SCVHunter: Smart Contract Vulnerability Detection Based on Heterogeneous Graph Attention Network

Feng Luo
University of Electronic Science and Technology of China
The Hong Kong Polytechnic University

Ruijie Luo
University of Electronic Science and Technology of China

Ting Chen*
University of Electronic Science and Technology of China

Ao Qiao
University of Electronic Science and Technology of China

Zheyuan He
University of Electronic Science and Technology of China

Shuwei Song
University of Electronic Science and Technology of China

Yu Jiang
Tsinghua university

Sixing Li
University of Electronic Science and Technology of China

ABSTRACT

Smart contracts are integral to blockchain's growth, but their vulnerabilities pose a significant threat. Traditional vulnerability detection methods rely heavily on expert-defined complex rules that are labor-intensive and difficult to adapt to the explosive expansion of smart contracts. Some recent studies of neural network-based vulnerability detection also have room for improvement. Therefore, we propose SCVHunter, an extensible framework for smart contract vulnerability detection. Specifically, SCVHunter designs a heterogeneous semantic graph construction phase based on intermediate representations and a vulnerability detection phase based on a heterogeneous graph attention network for smart contracts. In particular, SCVHunter allows users to freely point out more important nodes in the graph, leveraging expert knowledge in a simpler way to aid the automatic capture of more information related to vulnerabilities. We tested SCVHunter on reentrancy, block info dependency, nested call, and transaction state dependency vulnerabilities. Results show remarkable performance, with accuracies of 93.72%, 91.07%, 85.41%, and 87.37% for these vulnerabilities, surpassing previous methods.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Blockchain, Smart Contract, Vulnerability Detection

*Corresponding author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639213

ACM Reference Format:
Feng Luo, Ruijie Luo, Ting Chen, Ao Qiao, Zheyuan He, Shuwei Song, Yu Jiang, and Sixing Li. 2024. SCVHunter: Smart Contract Vulnerability Detection Based on Heterogeneous Graph Attention Network. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639213

1 INTRODUCTION

Blockchain revolutionized global transactions through decentralized cryptocurrencies. Bitcoin [48] paved the way by introducing automated transactions using scripts. However, the limited Turing-completeness of Bitcoin scripts confined them to currency transactions. The breakthrough came with the concept of smart contracts, which are Turing-complete programs executing predefined logic on the blockchain. Smart contracts opened doors to reshaping various industries, such as finance and gaming.

While smart contracts offer advantages, they've become attractive to attackers due to storing valuable cryptocurrency. As they run on permission-less networks, poorly designed contracts expose vulnerabilities to anyone [5]. Immutability on the blockchain worsens the situation, making it hard to fix vulnerable contracts once deployed. For example, the infamous DAO incident on Ethereum, an attacker exploited the reentrancy vulnerability to create a 70 million USD financial loss [18]. Hence, ensuring vulnerability-free smart contracts is crucial to protect assets from such incidents.

Various traditional methods for detecting smart contract vulnerabilities have been proposed, such as formal verification [27, 37, 58], symbolic execution [28, 46, 64], and fuzzing [35, 63]. However, these methods often depend on intricate fixed patterns or well-crafted test cases, requiring experts to deeply analyze vulnerabilities and convert their knowledge into complex rules specific to the type of vulnerability. While expert rules can enhance detection accuracy, they must be meticulously integrated into the analysis tool's source code for practical use. With the rapid growth of smart contracts, depending on experts to examine all vulnerability types, create corresponding rules, and modify source code becomes challenging.

In addition, deep learning-based methods have recently gained a lot of attention. Previous efforts include using LSTM for sequential analysis of contracts [51, 60], but this approach loses contract structural information. Zhuang et al. addressed this by translating smart contracts into graphs and using graph neural networks (GNN) for detection [79]. However, their model cannot handle heterogeneous graphs with multiple node types, so they remove secondary nodes and aggregate their features into primary nodes, resulting in information loss. Nguyen et al. proposed a contract heterogeneous graph based on the control flow and invocation graph for vulnerability detection [50], but all nodes have equal status in their graph structure without emphasizing key nodes specific to the vulnerability type. AME used a detection methods that fuse expert knowledge with neural networks, but their expert model requires writing to the source code and is therefore difficult to extend [43].

In this paper, we propose SCVHunter, a vulnerability detection method based on heterogeneous contract semantic graph (CSG) analysis. The main idea of SCVHunter is to convert smart contract source code into CSG by combining concise, user-supplied core node selection rules and using GNN for feature learning to detect vulnerabilities. This design is more extensible and eliminates the need to write complex expert rules into the source code, allowing core nodes to be specified by a single line of externally-entered commands, further improving accuracy while maintaining efficiency.

It is non-trivial to design SCVHunter because of the following challenges. **C1:** The data structures of contracts have many members and they are indexed differently in the EVM, which does not facilitate users to easily identify an element as a core node. Therefore, we elevate smart contracts to an intermediate representation (IR) with rich contract properties attached (§4.2) to facilitate users to specify core nodes using formal rules that restrict contract properties (§4.3). **C2:** Complex heterogeneous graphs require precise learning models to reduce the interference of benign nodes that are not related to vulnerabilities. Therefore, we design a graph neural network containing two layers of attention mechanisms at the node level and path level to give more attention to non-benign nodes (§4.4).

As a proof of concept, we use SCVHunter to detect reentrancy, block info dependency, nested call, and transaction state dependency vulnerabilities. We evaluate SCVHunter using a large annotated dataset of 1,200 labeled contracts we built and an open-source dataset of 47,587 contracts [14]. Extensive experiments are conducted, comparing SCVHunter with six neural network-based approaches and six state-of-the-art tools based on traditional techniques. The results demonstrate that SCVHunter achieves detection accuracy rates of 93.72%, 91.07%, 85.41%, and 87.37% for the mentioned vulnerabilities, respectively, outperforming existing methods significantly. Additionally, SCVHunter achieves true positive and true negative rates of 91.67% and 90.00% in real contract data not within the scope of manual labeling.

In summary, this work has three major contributions.

- We present SCVHunter, an extensible new approach for smart contract vulnerability detection using CSG and GNN.
- We design a novel IR-based approach for heterogeneous contract semantic graph generation and allow users to specify core nodes that contribute more to vulnerability detection.
- We compared SCVHunter with a total of 12 traditional vulnerability detection tools and neural network-based tools. To the best of our knowledge, this is the largest comparative experiment of its kind of work and the experimental results show that SCVHunter set a new state-of-the-art performance in smart contract vulnerability detection. Our tool and dataset have been released to the community at https://doi.org/10.6084/m9.figshare.24566893.v1 and we will release the source code upon acceptance of the paper.

2 BACKGROUND

2.1 Blockchain

Contract. Smart contracts are programs that define a set of rules for the governing of associated funds, typically written in a Turing-complete programming language (e.g., Solidity)[9]. After being compiled into bytecode, smart contracts will be deployed to the blockchain and executed according to the predefined program logic[7, 71]. After deployment, a smart contract cannot be modified. A smart contract can provide methods to be invoked by others, and emit events to inform other applications.

Fallback function. The fallback function is a unique feature of smart contracts compared to traditional programs. It is an arbitrary-programmable special function with no arguments or return values. If the function is marked by payable, the fallback function will also be executed automatically when receiving transfers [56].

Transfer and Call Instruction. In Solidity smart contracts, there are three methods to transfer tokens: $call.value()$, $send()$, and $transfer()$. Only $call.value()$ sends all the gas to the contract address for subsequent functions to execute, $send()$ and $transfer()$ are limited to 2300 gas units [8]. These three types of methods and other special behaviors (e.g., function call) generate CALL instructions during the process of compiling smart contracts into bytecode. A CALL instruction reads seven values from the top of the EVM stack. They represent the gas limitation, recipient address, transfer amount, input data start position, size of the input data, output data start position, and size of the output data, respectively.

2.2 Graph Neural Networks

Graph Neural Network (GNN) is a deep neural network designed to process graph-structured data of arbitrary structure [68]. This architecture combines graph broadcasting with deep learning, leveraging both graph structure and vertex attribute information for improved learning outcomes. GNN calculates new feature vectors layer by layer by aggregating domain node features. With multiple iterations, GNN learns structural information of adjacent nodes through feature vectors. This enables the model to handle complex graph data for tasks like graph or node classification, and edge prediction [80].

GNNs can be categorized as isotropic and anisotropic based on how they aggregate structural information from adjacent nodes. In the isotropic model, all edge-oriented nodes are treated equally, and each neighbor node contributes equally to the central node. An example is the graph convolutional network (GCN), which generalizes the convolution operation for graph data [41]. GCN aggregates graph information and updates the center vertex feature by convolving it with its neighbors' features. In contrast, the anisotropic model treats each edge direction differently, assigning weights to nodes and aggregating neighbor information based on those weights. An example is the graph attention network (GAT), which calculates weights for neighbor nodes using a self-attention

```
contract Victim {
  mapping(address => uint) public userBalance;
  function withdraw() {
    uint amount = userBalance[msg.sender];
    if (amount > 0) {
      require(amount > 0)
      msg.sender.call{value:amount}("");
      userBalance[msg.sender] = 0;}}}
```

```
contract Attacker {
  function() payable {
    Victim(msg.sender).withdraw();}
  function attack(address addr) {
    Victim(addr).withdraw();}}
```

Figure 1: Reentrancy

mechanism to identify more important neighbors [66]. However, these GNNs cannot handle graphs with multiple node and edge types and are limited to analyzing homogeneous graphs.

3 VULNERABILITIES DESCRIPTION

Similar to any other software, smart contracts may have vulnerabilities caused by various factors. Monika et al. [12] comprehensively studied the types of vulnerabilities and a taxonomy based on Smart Contract Weakness Classification (SWC) [59] and Decentralized Application Security Project (DASP) [49] mappings. These vulnerabilities threaten blockchain security. Chen et al. previously analyzed the impact level of various vulnerabilities in detail [6], and based on their results, we selected four of the most serious and widespread vulnerabilities to validate the effectiveness of SCVHunter. These vulnerabilities have also been widely selected for study in prior work due to their destructive nature [19, 21, 37, 43, 75, 79]. As discussed in §4.5, our approach easily extends to other vulnerabilities.

3.1 Reentrancy

The fallback mechanism of smart contracts allows an attacker to reenter the called function during a context switch [36], which may result in reentrancy vulnerabilities. When the smart contract C_1 uses the function f_1 containing the *call.value*() method to transfer money to the recipient contract C_2, the fallback function f_2 of C_2 will be automatically executed. By designing f_2, C_2 can call back to f_1 causing multiple calls and money transfers.

Example: Fig. 1 shows a reentrancy vulnerability example. The *Victim* contract acts as a bank, holding users' Ethers. Users can withdraw Ethers by calling the *withdraw*() function, which contains the vulnerability. The *Attacker* contract can exploit this vulnerability to steal funds from the *Victim* contract. First, the *Attacker* invokes the *withdraw*() function of the *Victim* contract to withdraw Ethers (step 1). The *Victim* contract sends Ethers to the attacker using *call.value* (step 2). Instead of resetting the attacker's balance to 0 as expected, the *call.value* triggers the *Attacker* contract's fallback function (step 3). In this fallback function, the attacker calls the *withdraw*() function again (step 4). This process continues in a loop until the Ethers in the *Victim* contract are depleted.

3.2 Block Info Dependency

Smart contracts can access block information (including *timestamp*, *hash*, *gaslimit*, and *number*) as execution context, but using those information as triggers to perform certain critical operations may lead to block info dependency vulnerability. For example, when

```
contract Roulette {
  uint public pastBlockTime;
  fallback() external payable {
    require(msg.value == 1 ether);
    require(block.timestamp != pastBlockTime);
    pastBlockTime = block.timestamp;
    if(block.timestamp % 15 == 0) { // winner
      payable(msg.sender).transfer(address(this).
          balance);}}}
```

Figure 2: Block Info Dependency

these properties of future blocks are used as seeds for generating random numbers to determine the winner of a lottery ticket. Note that most vulnerability detection tools currently only consider *timestamp* dependency vulnerabilities and ignore *hash*, *gaslimit* and *number* dependency vulnerabilities. Our method takes all of these variables into account.

Example: Fig. 2 shows a timestamp dependence vulnerability. The contract creates a lottery game using the fallback method (line 3), where each block allows one player to wager 1 ether to participate. As per the logic in line 7, the player has a 1/15 chance of winning the lottery. However, miners who mine blocks can freely set the timestamp in 900-second intervals [35]. Hence, if there's enough ether in the contract, miners are incentivized to select a block with $block.timestamp\%15 = 0$ to win the reward for the locked ether.

3.3 Nested Call

In the Ethereum execution environment, it is very expensive for contracts to use the CALL command for context switching (9000 gas paid for a non-zero value transfer as part of the CALL operation) [5, 72]. If a loop with unrestricted loop iterations contains CALL instructions, there is a risk that its gas cost will exceed the limit.

Example: In Fig. 3, an attacker can maliciously increase the number of loop iterations to cause an out-of-gas error by increasing the length of *member*. Once the out-of-gas error happens, this function cannot work anymore, as there is no way to reduce the loop iterations[5].

```
contract Roulette {
  for (uint i = 0; i<member.length; i++) {
    member[i].send(1 wei);}}
```

Figure 3: Nested Call

3.4 Transaction State Dependency

In smart contract context, *tx.origin* is a global variable that returns the original address of the transaction, but this method is unreliable because the address returned by this method depends on the status of the transaction. Therefore, if the contract uses *tx.origin* to check if the caller has the correct permissions for certain permission-sensitive functions, it could lead to serious consequences[6].

Example: In Fig. 4, the *Attacker* contract can make a permission check fail by utilizing the attack function (Line 7). By utilizing this method, anyone can execute sendMoney function (Line 3) and withdraw the Ethers in the contract.

4 SCVHUNTER

4.1 Architecture

Fig. 5 shows the overall design of SCVHunter, consisting of three main modules: `IR Translator`, `Contract Graph Generator`, and

```
contract Victim {
  address owner = owner address;
  function sendMoney (address addr) {
  require (tx.origin == owner);
  addr.transfer(1 Ether);}}
contract Attacker {
  function attack (address vim addr, address myAddr) {
  Victim vic = Victim (vim addr);
  vic.sendMoney(myAddr);}}
```

Figure 4: Transaction State Dependency

`Vulnerability Detector`. `IR Translator` converts smart contract source code into CGIR, a semantic property-rich intermediate representation suitable for our graph construction approach. `Contract Graph Generator` constructs a smart contract heterogeneous graph based on CGIR using user-specified core node selection rules. Finally, `Vulnerability Detector` processes the graph features and outputs the final detection results.

4.2 IR Translator

To facilitate the translation of smart contracts into heterogeneous graphs, `IR Translator` first converts the input smart contracts into a new IR in the form of Static Single Assignment (SSA) [53] called CGIR, which accurately represents control and data information. To meet the demands of smart contract heterogeneous graph generation, we assign special features to CGIR. We show the features and syntax of CGIR in §4.2.1 and §4.2.2, respectively.

4.2.1 Features. Distinct from previous studies on IR of smart contracts [2, 19], our CGIR has the following three salient features.

F1: Mappings, structs, and arrays are converted into immutable data structures. Solidity allows the manipulation of mappings, arrays, and structures accessed by dereferencing, so previous work used specific variable types to store the results of dereferencing [19]. Unlike them, our CGIR elevates these three types of data to separate wholes so that they can be treated as nodes when constructing CSG.

F2: Variables and functions with additional refined contract attributes. To facilitate the automatic generation of core nodes based on user-specified rules during the heterogeneous graph construction phase, for variables, our CGIR provides basic type and refined type annotations. The elementary type annotations cover the six elementary types (i.e. address, bool, string, int, uint, byte) as well as mapping, structure, and array types that are fully supported by Solidity nowadays. Refined type annotations provide a tailored representation of specific variables that are frequently associated with smart contract vulnerabilities. Specifically, Solidity contains

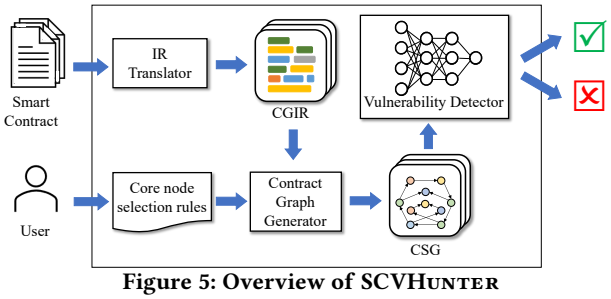


Figure 5: Overview of SCVHunter

```
mapping(address => uint256)
 public balance;
```

(a) Data structure a

```
mapping (address => uint256)
 userID;
mapping (uint256 => uint256)
 balances;
```

(b) Data structure b

```
struct asset {
 string name;
 uint amount;
}
mapping(address => asset)
 public balances;
```

(c) Data structure c

```
struct Checkpoint {
 uint fromBlock;
 uint value;
}
mapping (address =>
 Checkpoint[]) balances;
```

(d) Data structure d

Figure 6: Data structures for recording account information

13 variables built into the global namespace to describe block and transaction properties [17], but some of these variables can cause vulnerabilities (e.g. block.timestamp). These built-in variables are essentially variables with basic types, but to distinguish them more precisely from ordinary variables, their names are used directly as their types in our CGIR. For example, block.timestamp is a built-in variable of type uint that expresses the timestamp of the current block, and its type is noted as block.timestamp in CGIR.

Moreover, since the cryptocurrency-driven nature of the blockchain is the root cause that motivates attackers to launch attacks, the data structures used to store account balance information in smart contracts also require particular attention. Chen et al. [10] and He et al. [29] researched the data structure for recording account information in smart contracts in the past. Based on their results, we set CGIR to record the four data structures for storing account information as shown in Fig. 6 as balance-type special variables.

For functions, CGIR provides access privilege and call type annotations. The access privileges are extracted from the *require* statement and *modifier* contained in the function. Modifiers are inheritable properties of smart contracts and are used to alter the behavior of functions [15]. The *require* and *modifier* are often used to perform access checks when executing functions. For example, *require(msg.sender == owner)* can check if the caller is the contract owner. In addition, for function calls, eight call types are defined (see §4.2.2) to express the call information in greater detail.

F3: Construct fallback statements to explicitly represent the fallback function of the attack contract. Fallback mechanism is the cause of many smart contract vulnerabilities, and the attack contract can interact with the victim contract through the fallback function for the attack. To detect these vulnerabilities during the detection process for a single contract, inspired by the work of Rondon et al. [52], `IR Translator` constructs $fallback\ call.value()\ to\ f$ during the translation of a contract containing call.value() to CGIR, simulating that the preceding *call.value* operation triggers the fallback function of the attack contract, which in turn activates the subsequent process of calling back the function f in the victim contract. In this way, the fallback function is explicitly represented in the subsequently constructed heterogeneous graph.

4.2.2 Syntax. Fig. 7 illustrates the syntax of CGIR, an IR for modeling Solidity smart contracts. Type in CGIR has been elaborated in §4.2.1, and in the following, we briefly outline the statements and expressions of CGIR that are relevant to the rest of the paper.

Statements in CGIR include let binding, conditional, require, assertion, etc. As CGIR programs are assumed to be in SSA form, each variable must be assigned only once and defined prior to use. As mentioned previously, another feature of CGIR is that it contains fallback statement. Note that Solidity does not contain such fallback structures, we include it in CGIR to allow explicit expression of the attacker's exploit of the fallback mechanism.

Expressions in CGIR consist of variables, unsigned integer and boolean constants, binary operations, and data structure operations. Data structures include maps and structs, and we use the same notation for accessing/updating them. In particular, $e_1[e_2]$ yields the value stored at key (resp. field) e_1 of map (resp. struct) e_2. Similarly, $e_1[e_2 \dashv e_3]$ denotes the new map (resp. struct) obtained by writing value e_3 at key (resp. field) e_2 of e_1. As a simple example, we show the result of lifting the Victim contract in Fig. 1 to CGIR in Fig. 8(a).

4.3 Contract Graph Generator

After converting the smart contract to CGIR, `Contract Graph Generator` captures the control and data flow from the IR. Combining the core node rules entered by the user to extract node and edge type information, `Contract Graph Generator` constructs a heterogeneous CSG, denoted $\mathcal{G} = (V, E, \phi, \psi)$, consisting of a vertex set V and an edge set E. $\phi(V) \rightarrow A$ is a node-type mapping function. $\psi(E) \rightarrow R$ is an edge-type mapping function. A and R are sets of node types and edge types, respectively. CSG is a directed graph and models the execution flow.

4.3.1 Node. The nodes in the CSG represent variables or function calls in a smart contract. As mentioned earlier, not all nodes are of equal importance in the vulnerability detection process, and the types of important nodes are specific to the vulnerability type. Therefore, we formulate four types of nodes for CSG, i.e., core nodes, invocation nodes, variable nodes, and fallback nodes.

Core nodes. Core nodes symbolize critical variables or function invocations that are related to the existence of vulnerabilities. To make it effortless for users to specify core nodes when detecting vulnerabilities, we formalize the rules for selecting core variable nodes as well as core invocation nodes as follows.

$$varType(A) \vee varAssign(A) \vee InvType(A) \vee InvContained(A)$$

In this formulation, A represents vulnerability-related attributes (e.g., type, statements). The nodeType rule (resp. InvType rule) is queried for variables (resp. function call) that satisfy the node attribute A. The varAssign rule queries for variables assigned by variables that satisfy the A attribute. The InvContained rule is queried for function calls that internally contain nodes with the A attribute. All queried nodes are considered core nodes. Based on the above formula, Table 1 summarizes the detailed core node selection rules for analyzing the four types of vulnerabilities. For reentrancy, core nodes model (i) an invocation to the *call.value* function, (ii) the variable that corresponds to user balance, and (iii) variables that can directly affect user balance. For block info dependency, (i) invocations to block information, and (ii) variables assigned by block information are extracted as core nodes. For nested call, (i) all the loop condition variables, and (ii) invocations containing low-level call are considered core nodes. For transaction state dependency, (i)invocations to $tx.origin$, and (ii)invocations to *require* function are considered as core nodes.

Table 1: Selection rules for locating core node

Vulnerability type	Core node selection rule
Reentrancy	$varType(Balance) \vee varAssign(Balance) \vee InvContained(call.value)$
Block Info Dependency	$varType(block.information) \vee varAssignBy(block.information)$
Nested Call	$varType(for) \vee InvContained(LowCall)$
Transaction State Dependency	$varType(tx.origin) \vee InvContained(require)$

Invocation nodes. Function invocations that are not extracted as core nodes are modeled as invocation nodes.

Variable nodes. Similar to invocation nodes, variables that are not extracted as core nodes are modeled as variable nodes.

Fallback nodes. Many vulnerabilities in smart contracts are directly or indirectly related to the fallback function, however, such vulnerabilities are often manifested during the interaction with the attacking contract. To represent this process in CSG, we construct fallback nodes based on the fallback statement added in §4.2.1 to simulate the fallback function that inspires a virtual attack contract and interacts with the function under test.

4.3.2 Edge. To capture rich semantic relationships between nodes, we define two levels of edge classification (Table 2). Different nodes are connected based on statement descriptions in CGIR, creating a fine-grained first-level type set FT of edges with 11 types. These edges direct from the previous node, representing the preceding function call or variable, to the node representing the current function call or variable in the statement.For modeling the natural control flow transfer of code sequences, we employ SQ edges, connecting nodes in adjacent statements. This preserves the programming logic reflected in the source code sequence. For variable access in each statement, we use AC edges for the description. In particular, we can explicitly describe the fallback mechanism automatically based on the fallback statements given in CGIR. The first fallback edge is connected to the fallback node from the first *call.value*, whereas the second edge connects the fallback node to the higher-level function that contains the *call.value*.

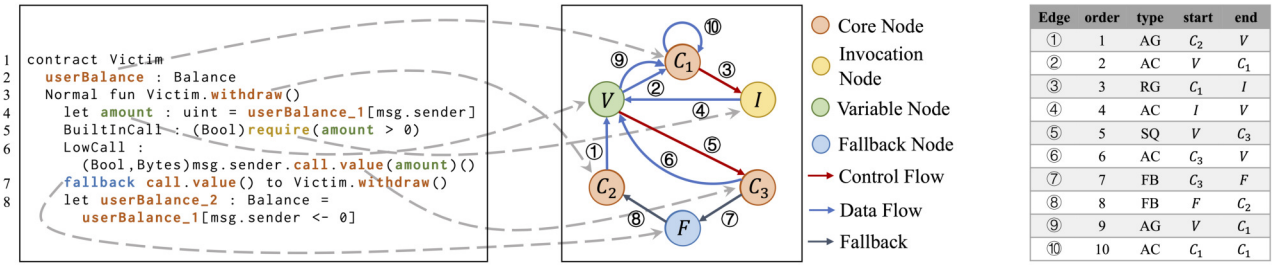
Furthermore, to more deeply portray the control and data dependency relationship, we construct a coarse-grained second-level classification ST inspired by [79], which categorizes all edges as control flow, data flow, and fallback edge. Control flow edges capture the control semantics of the code. Data flow edges track the use of variables, involving access to or modification of variables.

Table 2: Two-level classification of edges

Semantic	Statement	First-level	Second-level
assertion	`assert` e	AS	control flow
require	`require` e	RG	
if-then-else	`if` e	IF	
	`then` s	TH	
	`else` s	EL	
while loop	`while` e	WH	
for loop	`for` e	FO	
sequence	$s_1; s_2$	SQ	
assignment	`let` $x : \tau = e$	AG	data flow
access	$e_1 = e_2$	AC	
fallback	`fallback` *call.value* `to` f	FB	fallback edge

Types		**Statements**		**Expressions**	
t ::= `Address`	address	s ::= `let` $x : \tau = e$	assignment	e ::= n	constants
| `Bool`	boolean	| $s_1; s_2$	sequence	| `true`|`false`	boolean
| `String`	string	| `assert` e	assertion	| x	variable
| `Int`	integer	| `require` e	require	| $\ominus e$	unary operation
| `Uint`	unsigned integer	| `if` e `then` s_1 `else` s_2	if-then-else	| $e_1 \oplus e_2$	binary operation
| `Byte`	byte	| `while` e `do` s	while loop	| `mapping`$(t_i \mapsto e_i)$	mapping with t keys
| `Map`(t)	mapping with t keys	| `for` e `do` s	for loop	| `struct`$(x_i \mapsto e_i)$	struct
| `Struct` s	struct	| `skip`	skip	| $e_1[e_2]$	data structure index
| `Array` a	array	| `fallback` *call.value* `to` f	fallback statement	| $e_1[e_2 \dashv e_3]$	data structure update
| `Built-in`	built-in variables				
| `Balance`	balance				
Function Type		**Function Call Type**			
T ::= `Normal`	normal function	c ::= `LowCall`	low-level Solidity call	| `ExternalCall`	external function call
| `Ctor`	constructor	| `LibCall`	library call	| `EventCall`	event call
| `Fallback`	fallback function	| `BuiltInCall`	call built-in function	| `SendCall`	Solidity send
| `Mod`	modifier	| `InternalCall`	internal function call	| `TransferCall`	Solidity transfer

Figure 7: Syntax of CGIR programs



Edge	order	type	start	end
①	1	AG	C_2	V
②	2	AC	V	C_1
③	3	RG	C_1	I
④	4	AC	I	V
⑤	5	SQ	V	C_3
⑥	6	AC	C_3	V
⑦	7	FB	C_3	F
⑧	8	FB	F	C_2
⑨	9	AG	V	C_1
⑩	10	AC	C_1	C_1

(a) CGIR for Victim contract **(b) CSG for Victim contract** **(c) Edge Features**

Figure 8: CGIR conversion and CSG construction process

4.3.3 Node and edge features. `Contract Graph Generator` extracts different features based on node and edge types. For function invocation nodes, features are represented by a quadruple $F = (ID, Type, Caller, AccPer)$, with ID as the identifier, $Type$ as the node type, $Caller$ as the invoking entity, and $AccPer$ as the access privilege (see §4.2.1). For variable or fallback nodes, features are in a binary group $F = (ID, Type)$. Edges have features represented as a quadruple $F = (Order, Type, V_{start}, V_{end})$, with $Order$ denoting temporal order, $Type$ as the first level type of the edge, and V_{start} and V_{end} as start and end nodes. For example, Fig. 8(b) shows a CSG constructed based on CGIR of the Victim contract, and Fig. 8(c) shows its edge features.

4.4 Vulnerability Detector

Our CSG is able to reflect the semantic information of smart contracts more comprehensively, but it inevitably becomes more complex as smart contracts become more complex. This means that as the number of functions increases, there will be more and more non-vulnerable lines corresponding to benign nodes on the graph, causing significant interference to the detection. To address this problem, we design a `Vulnerability Detector` with a dedicated network model to mine the syntactic and semantic information in the CSG more thoroughly. Specifically, inspired by the work of [68], we design a node-level and path-level dual-layer attention mechanism based on meta-paths, a popular semantic capture structure.

Definition 4.1 (Meta-path). A metapath Φ is defined as a path in the form of $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \cdots \xrightarrow{R_l} A_{l+1}$, which defines a composite relation $R = R_1 \circ R_2 \circ \cdots \circ R_l$ between type A_1 and A_{l+1}, and $\circ$ denotes the composition operator on relations.

Fig. 9 shows the workflow of `Vulnerability Detector`. We use the two-layer attention mechanism to update graph node information. Next, we aggregate and input the node information into the convolution module to obtain high-level contract features. The high-level features' dimensionality is compressed using the MLP module to determine if the sample attributes contain vulnerabilities. In other words, we estimate the label $\tilde{y}$ for each smart contract, where $\tilde{y} = 1$ indicates the contract has a specific vulnerability, and $\tilde{y} = 0$ indicates the contract is secure.

4.4.1 Node-level Attention. Note the fact that the meta-path-based neighbors of each node play diverse roles in node embedding learning for vulnerability detection tasks, with a considerable number of benign nodes not connected with vulnerabilities. Thus directly aggregating all incoming messages from a node's neighbors to update the state vector tends to mask vulnerability information, since most of the information it learns is not vulnerable. To counter this issue, we introduce a node-level attention mechanism to achieve the effect of mimicking manual vulnerability analysis. By learning the importance of neighbors to each node, more weight is given to

nodes related to vulnerabilities, so that node embeddings contain more vulnerability-related features.

To enable node-level attention for different node types, we use node-type-specific transformation matrices M_{ϕ_i} for each node of type ϕ_i. Features of different nodes are projected into the same feature space using a linear transformation $h'i = M\phi_i \cdot h_i$, where h_i and $h'i$ are the original and projected features of node i, respectively. We then learn the node weights using a self-attentive mechanism [65]. In vulnerability detection, the set of meta-paths is specified as the first-level types of edges in the CSG. For a given pair of nodes (i, j) connected by a meta-path Φ, the importance of j to i is measured by node-level attention eij, defined as follows:

$$e_{ij}^{\Phi} = att_{node}(h_i', h_j'; \Phi) \tag{1}$$

where att_{node} denotes the deep neural network that computes node-level attention, which is a multilayer perceptron [38], whose parameters are learned automatically by backpropagation. The input to such a perceptron is a concatenation of two vectors h_i' and h_j'. Given a meta-path Φ, all node pairs of this meta-path share the same att_{node}. After obtaining the importance between node pairs based on the meta-path, we regularize them by the softmax function to obtain the weight coefficients a_{ij}^{Φ}:

$$a_{ij}^{\Phi} = softmax_j(e_{ij}^{\Phi}) = \frac{exp(\sigma(a_{\Phi}^T \cdot [h_i' || h_j']))}{\sum_{k \in \mathcal{N}_i^{\Phi}} exp(\sigma(a_{\Phi}^T \cdot [h_i' || h_k']))} \tag{2}$$

where σ denotes the activation function, $\mathcal{N}_i^{\Phi}$ denotes the meta-path based neighbors of node i (include itself), $||$ denotes the concatenate operation and a_{Φ} is the node-level attention vector for Φ.

Then, the metapath-based node i embedding aggregation is performed utilizing the projection features of the neighbors and the related weight coefficients. Due to the scale-free nature of the heterogeneous graph, it will cause a large variance of the graph data. To solve this issue, we use multi-headed attention to scale the node-level attention to make the training process more stable. More precisely, we repeat the node-level attention K times and connect the learned embeddings into semantics-specific embeddings as follows:

$$t_i^{\Phi} = \Big\|_{k=1}^{K} \sigma\Big(\sum_{k \in \mathcal{N}_i^{\Phi}} a_{ij}^{\Phi} \cdot h_j'\Big) \tag{3}$$

To further enhance the semantic information carried by our semantic-specific embeddings, we design feature vectors s of the secondary type Ψ specific to the edges of the CSG and enhance the semantic

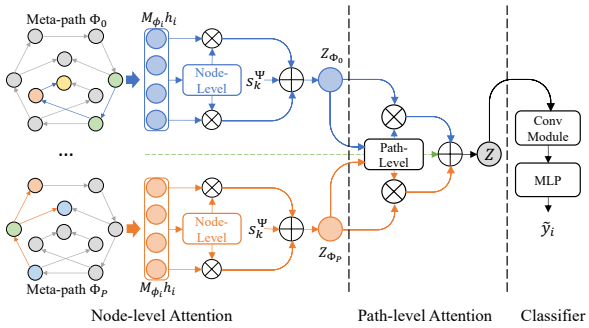


Figure 9: The workflow of the vulnerability detector

embeddings by $z_i^{\Phi} = t_i^{\Phi} + s_{\Psi}$. After the feature dimension processing, we can then obtain the P-group specific semantic node embedding by node-level attention, denoted as $\{Z_{\Phi_0}, Z_{\Phi_1}, \ldots, Z_{\Phi_P}\}, P = |FT|$.

4.4.2 Path-level Attention. Most of the vulnerabilities in smart contracts are not related to only one semantics but to multiple semantic states. To detect them, it is necessary to consider the various semantic information included in each node of the heterogeneous graph. However, in node-level attention, a specific semantic node embedding based on a meta-path reflects the node information in only one way. Therefore, we design path-level attention to perform more comprehensive node embedding learning by fusing the specific semantics embedded in each meta-path. Specifically, to comprehend the importance of each path, we first apply a layer of multilayer perceptron (MLP) to nonlinearly transform the embeddings of specific paths. Then the similarity of a path-level attention vector q to the nonlinear transformation of the semantic node-specific embeddings is used to measure the importance of the semantic embeddings of the specific semantic nodes of a given meta-path. Finally, the importance of each meta-path w_{Φ_i} is determined by averaging the importance of all semantic node-specific embeddings:

$$w_{\Phi_i} = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{V}} q^T \cdot tanh(W \cdot z_i^{\Phi} + b) \tag{4}$$

where W is the weight matrix, b is the bias, and q is the path-level attention vector. For a meaningful comparison, all the above parameters are shared for all meta-paths and semantics-specific embeddings. After obtaining the importance of each meta-path and normalizing it by the softmax function, we can obtain the weight of the meta-path Φ_i denoted by β_{Φ}:

$$\beta_{\Phi_i} = \frac{exp(w_{\Phi_i})}{\sum_{i=1}^{P} exp(w_{\Phi_i})} \tag{5}$$

which can be interpreted as the contribution of meta-path Φ_i to the vulnerability detection task. Obviously, the higher β_{Φ_i} is, the more important Φ_i is. Finally, we use the weights as coefficients to fuse these semantic specific embeddings to obtain the final embedding Z:

$$Z = \sum_{i=1}^{P} \beta_{\Phi_i} \cdot Z_{\Phi_i} \tag{6}$$

4.4.3 Classifier. In the final stage of the vulnerability detection task which is essentially the completion of graph classification, a more efficient classifier is needed to make the detection results of SCVHunter more accurate. Inspired by [39, 68], we design to let the connected nodes now convolve in two layers to capture high-level features that are more relevant to the vulnerability and then input the learned features into the MLP for binary classification.

4.5 Extensibility

Since CGIR is designed based on Solidity syntax rather than specific vulnerabilities, it applies to all smart contracts. The core node selection rules do not require any source code modification and only require simple user input to point out important elements related to new vulnerabilities. Therefore, SCVHunter can be easily extended to other vulnerabilities. Because of the obvious targeting of smart contract vulnerabilities, this task demands little effort and is less

Table 3: Filtered contracts and labeled contract distribution

Vulnerability	Filtered Contracts	Vulnerable	Non-vulnerable
Reentrancy	9,352	203	97
Block Info	24,689	184	116
Nested Call	28,438	187	113
Transaction State	4,301	180	120

error-prone. Moreover, users have the flexibility to experiment with different inputs to find the optimal core node selection rule.

The current design of SCVHunter allows it to extend to most types of vulnerabilities. As an example, we analyzed the work of Zhou et al. [78] covering 16 smart contract-level vulnerabilities and found that SCVHunter can extend to 13 of them, except for underpriced opcodes, direct calls to untrusted contracts, and outdated compiler versions. The first two vulnerabilities cannot be supported because SCVHunter currently applies to contract source code and does not support cross-contract call analysis, while the third vulnerability is caused by the Solidity compiler, which is not a defect of the contract itself, and is therefore out of the scope of our detection.

5 IMPLEMENTATION AND EVALUATION

Implementation. Our `IR Translator` is based on Slither [19], which converts smart contracts into an intermediate representation called SlithIR. We modified and extended it to ensure CGIR has the desired features and a syntax suitable for heterogeneous contract graph generation. We implemented `Contract Graph Generator` and `Vulnerability Detector` using over 3000 lines of Python code, with the neural network designed using Pytorch and DGL.

Experimental setup. All experiments were conducted on a virtual machine with a 16-core processor, 24G of RAM and NVIDIA Tesla P100 graphics card, which runs Ubuntu 18.04.

Dataset. To build a suitable dataset, we exerted significant effort, as no large dataset covered those four vulnerabilities. Previous methods used the intersection of multiple existing tools to build the dataset [1, 33, 45, 47], but this approach lacks complete accuracy due to errors in each tool. For accurate evaluation, we chose to build the dataset manually. We began with 47,587 contract source codes from the Ethereum Open Dataset [14] and performed preliminary filtering using four types of keywords: call.value, block information (timestamp, hash, number, gaslimit), loop operations (for, while), and tx.origin. The filtered contracts were then manually analyzed by 3 graduate students with 2 years of blockchain experience, and each vulnerability had 300 samples labeled. Table 3 shows the number of filtered contracts and the distribution of labeled samples.

Parameter setting. The default parameters of model are as follows: learning rate = 0.004, batch size =128, and dropout = 0.1. We use 5-fold cross-validation to ensure that the train set represents all patterns involved in the vulnerabilities and report the average results.

Evaluation metrics. For all tools, we evaluate their performance in terms of four aspects: *Accuracy*, *Precision*, *Recall*, and *F1 Score*.

Research Questions (RQs): RQ1: How much does SCVHunter outperform other neural network-based approaches? **RQ2:** How does SCVHunter compare to state-of-the-art traditional smart contract vulnerability detection methods? **RQ3:** What is the time overhead of SCVHunter for vulnerability detection tasks? **RQ4:** How our proposed core node specification with heterogeneous CSG helps contract vulnerability detection? **RQ5:** Can SCVHunter identify vulnerabilities in real-world smart contracts?

5.1 Comparison with Other Deep Learning Based Methods

To evaluate SCVHunter's effectiveness, we compare its performance to other neural network-based vulnerability detection methods. We selected six open source detection methods based on different neural network models for a comprehensive evaluation: Vanilla-RNN [24], LSTM [54], GRU [11], DR-GCN [79], TMP [79], and AME [44]. The first three models take the text sequence of the smart contract code as input, while the latter three models take smart contract graph data as input. These methods are representative of typical vulnerability detection applications using their respective models, and they are widely chosen as benchmark methods by recent smart contract vulnerability detection studies [3, 31, 44, 73, 79]. We present the results in Table 4. Since none of the methods used for comparison supports block info dependency vulnerabilities other than timestamp dependency, we only perform the detection of timestamp dependency vulnerabilities for a fair comparison.

For the reentrancy and timestamp dependency vulnerabilities detected by all the tools, SCVHunter improves the detection accuracy by 32.26%[1] and 33.43%, respectively, over GRU, the best performer in sequence modeling. As for the GNN-based tools, SCVHunter improves the detection accuracy against the reentrancy and timestamp dependency vulnerabilities by 5.09% and 5.32%, respectively, compared to the best available tool, AME. This result is because our heterogeneous CSGs contain more fine-grained semantic information than homogeneous graphs. Interestingly, among all neural network architectures experimented with, the four GNN-based approaches perform significantly better than the other three. This suggests that blindly treating smart contracts as sequences of text for vulnerability detection may not be appropriate, as it ignores source-level structural information such as data flow and control flow, leading to the omission of meaningful data from the analysis. Instead, using GNN for vulnerability detection from the perspective of contract graph modeling shows more potential.

For nested call and transaction state dependency vulnerabilities, SCVHunter shows more significant improvement. Specifically, compared to GRU, the best performer on nested call vulnerability detection, and LSTM, the best performer on transaction state dependency vulnerability detection, SCVHunter's accuracy is improved by 30.14% and 35.86%, respectively. This phenomenon is because the other three experimental GNN-based approaches do not support these two vulnerabilities, while the sequence model-based approach performs poorly for the reasons previously described.

> **Answer to RQ1:** For the 4 vulnerabilities, SCVHunter's accuracy improved by 5.09%, 5.32%, 30.14%, and 35.86%, respectively, over the neural network-based architecture that previously performed best on each vulnerability.

[1]This and subsequent improvements are percentage point comparisons.

Table 4: Performance comparison with neural network-based approaches

Methods	Reentrancy				Timestamp Dependency				Nested Call				Transaction State Dependency			
	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1
Vanilla-RNN	51.43	56.92	50.37	53.45	48.61	44.24	53.36	48.37	49.22	45.26	48.31	46.74	47.95	52.43	44.86	48.35
LSTM	57.68	65.93	53.42	59.02	53.44	59.17	52.12	55.42	55.06	56.33	47.61	51.60	53.47	60.24	50.83	55.14
GRU	61.46	67.61	55.19	60.77	57.22	60.39	52.16	55.97	55.27	54.32	48.98	47.55	51.51	63.44	53.72	58.18
DR-GCN	80.55	80.71	74.11	77.27	79.27	76.97	72.21	74.51	-	-	-	-	-	-	-	-
TMP	83.57	83.05	77.14	79.99	82.39	83.13	74.65	78.66	-	-	-	-	-	-	-	-
AME	88.63	87.15	82.25	84.63	85.33	84.26	83.78	84.02	-	-	-	-	-	-	-	-
SCVHunter	**93.72**	**90.60**	**88.84**	**89.71**	**90.65**	**87.52**	**84.13**	**85.79**	**85.41**	**82.69**	**82.33**	**82.51**	**87.37**	**83.42**	**83.61**	**83.51**

"-" means the corresponding tool does not support detecting this type of vulnerability.

5.2 Comparison with Traditional State-of-the-art Tools

Apart from comparing with neural network-based methods, we also assess SCVHunter against state-of-the-art traditional contract vulnerability detection tools. Following Chen et al.'s [5] strategy, we selected Oyente [46], Smartcheck [61], Securify [64], Contractfuzzer [35], Mythril [16], and Slither [19] as benchmark tools from top conferences, top journals, and related papers in Sec/SE. Considering the characteristics of these tools, we use Smartcheck, Securify, and Slither for source-level vulnerability detection, and Oyente, Contractfuzzer, and Mythril for bytecode-level vulnerability detection. To ensure fairness, we ran them on the bytecode versions of the same dataset. Table 5 shows the results.

Results of reentrancy vulnerability. The results in Table 5 show that traditional vulnerability detection tools do not perform well enough on our dataset for reentrancy vulnerability detection. Slither, the best performer, achieves 82.67% accuracy, while SCVHunter significantly improves accuracy by 11.05% to 93.72%. This poor performance of traditional tools is attributed to their heavy reliance on simple and fixed patterns for vulnerability detection. For example, Mythril detects reentrancy based on whether the *call.value* invocation is followed by any internal function call. Additionally, tools that process smart contract bytecode also impact performance due to missing semantics at the bytecode level. For instance, Oyente and Mythril fail to differentiate between the $send()$, $transfer()$, and $call()$ functions, causing misidentification of reentrancy vulnerabilities. In contrast, SCVHunter gains an advantage due to its semantic processing capabilities.

Results of block info dependency vulnerability. Smartcheck and Securify do not support block information dependency vulnerability detection, so we only compare SCVHunter with Oyente, Mythril, Contractfuzzer, and Slither. We observe that SCVHunter consistently outperforms the conventional methods on all four metrics, achieving a significant improvement in accuracy (91.07%) compared to Oyente, Contractfuzzer, Mythril, and Slither (37.29%, 63.94%, 76.62%, and 64.79% respectively). The low values of all four metrics for Oyente are primarily due to its limited support for block.timestamp dependency detection in block information dependency vulnerability. Additionally, Oyente's approach to block info dependency detection by cursorily checking the presence of block variables in functions also contributes to its lower performance.

Results of the other two vulnerabilities. Since only Mythril, Slither, and Smartcheck support nested call and transaction state dependency vulnerabilities, we compared our tool with them. We observed that SCVHunter outperformed all three tools on all metrics. Compared to the best Mythril, SCVHunter improved the detection accuracy of nested call and transaction state dependency vulnerabilities by 16.56% and 12.05%, respectively. We observe that this improvement is not as significant as when SCVHunter is compared to other deep learning-based methods. This suggests that traditional methods are better than sequence models for these two vulnerabilities when no other neural network architectures support them.

> **Answer to RQ2:** Compared to the state-of-the-art conventional tools available, SCVHunter improved its accuracy by at least 11.05%, 14.45%, 16.56%, and 12.05% on the four vulnerabilities experimented with.

5.3 Time Overhead

We evaluated the time overhead of SCVHunter for detecting a smart contract. To ensure accuracy, we conducted the evaluation with all background processes shut down in a clean environment. SCVHunter was tested five times, and the average time to detect a contract in our dataset was recorded. Fig. 10 presents the time consumption results for each of SCVHunter's modules. The average time consumption for detecting a smart contract was 0.10s for `IR Translator`, 0.35s for `Contract Graph Generator`, and 0.21s for `Vulnerability Detector`. Overall, the average time overhead of SCVHunter is 0.66s. The maximum time for SCVHunter to analyze a smart contract is 2.04s, while the simplest smart contract in the dataset (only 7 rows) requires only 0.23s. Most of the overhead of SCVHunter comes from constructing CSG.

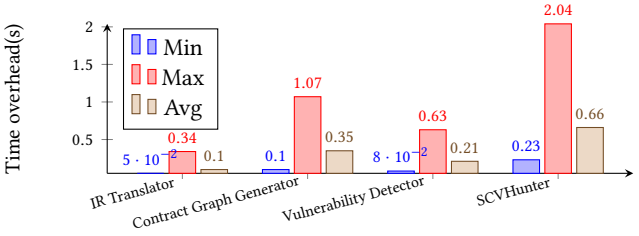


Figure 10: Time overhead of SCVHunter

Table 5: Performance comparison with traditional detection tools

Methods	Reentrancy				Block Info Dependency				Nested Call				Transaction State Dependency			
	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1
Oyente	63.25	55.31	44.26	49.17	37.29	24.62	28.34	26.35	-	-	-	-	-	-	-	-
Smartcheck	-	-	-	-	-	-	-	-	62.33	91.98	63.70	75.27	30.67	33.89	40.67	36.97
Contractfuzzer	67.15	61.42	47.26	53.42	63.94	67.42	64.52	65.94	-	-	-	-	-	-	-	-
Mythril	64.37	70.13	42.93	53.26	76.62	61.74	67.38	64.44	68.85	50.43	62.36	55.76	75.32	43.86	57.44	49.75
Securify	73.10	60.42	52.45	56.13	-	-	-	-	-	-	-	-	-	-	-	-
Slither	82.67	94.09	82.68	88.02	64.79	97.67	29.79	45.65	64.68	68.18	75.63	71.71	46.05	17.83	69.70	28.40
SCVHunter	**93.72**	**90.60**	**88.84**	**89.71**	**91.07**	**89.43**	**84.41**	**86.85**	**85.41**	**82.69**	**82.33**	**82.51**	**87.37**	**83.42**	**83.61**	**83.51**

Answer to RQ3: SCVHunter can analyze a smart contract in an average of 0.66 seconds.

5.4 Ablation Study

As mentioned earlier, the two main features of SCVHunter are that it allows users to specify the core node type and that it converts smart contracts into heterogeneous CSGs. To verify whether this design is effective, we conducted the following study.

Specifying the core node type. To evaluate the impact of this feature, we conducted experiments without entering core node selection rules. In this case, the contract graph constructor generated heterogeneous graphs containing the other three types of nodes only. We denoted this variant case as SCVHunterWCN (WCN for without core node) and compared it with the standard SCVHunter. Table 6 shows the results. We observed that the standard SCVHunter performed better. For reentrant vulnerability detection, the standard SCVHunter achieved 5.98%, 6.22%, 9.58%, and 7.97% improvement in accuracy, recall, precision, and F1 score, respectively. This result demonstrates the effectiveness of allowing core nodes to specify functionality and highlights the significant contributions of specific nodes in the vulnerability detection process, rather than all nodes having equal contributions.

Heterogeneous CSG. To assess the effectiveness of converting smart contracts into CSG, we compared the performance of SCVHunter based on homogeneous graph analysis with that of heterogeneous graph analysis. We modified `Contract Graph Generator` and transformed smart contracts into homogeneous graphs using the normalization method proposed by Zhuang et al. [79]. Similarly, the network design of `Vulnerability Detector` was downgraded to a GAT with a single layer of attention. We denoted this variant case as SCVHunterHOG (HOG for homogeneous graph). The empirical results are shown in Table 6. We observed that the standard SCVHunter significantly outperforms SCVHunterHOG in all aspects. This result demonstrates that vulnerability detection tasks based on our heterogeneous CSG are significant for improving performance, as the CSG holds more fine-grained semantic information in its graphical features.

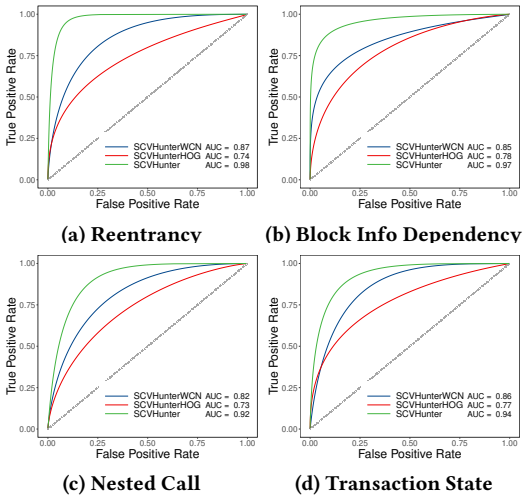
We also show the receiver operating characteristic curves (ROC) for SCVHunter, SCVHunterWCN and SCVHunterHOG in Fig. 11. The area under the curve (AUC) visually reflects the performance of vulnerability detection, and the higher the AUC, the better the performance. The results show that for the reentrancy vulnerability detection task, SCVHunter's AUC increases by 0.11 and 0.24, respectively. For the block info dependency vulnerability detection task, SCVHunter's AUC increases by 0.12 and 0.19, respectively. For the nested call vulnerability detection task, SCVHunter's AUC increases by 0.10 and 0.19, respectively. For the transaction state dependency vulnerability detection task, SCVHunter's AUC increases by 0.08 and 0.17, respectively.Therefore, our proposed insights are beneficial for smart contract vulnerability detection.

Answer to RQ4: SCVHunter's core node specification and heterogeneous CSG improve detection performance across the board.

5.5 Vulnerability Detection in the Wild

To evaluate SCVHunter's performance on smart contracts outside the labeled dataset, we randomly selected 500 smart contracts for each keyword category (totaling 2000 contracts) from the Ethernet Open Dataset [14], excluding the 1200 labeled contracts used in dataset construction. Similarly, we first performed the same keyword filtering as in the database construction process, because these keywords are the basic premise for the existence of the four vulnerabilities we are concerned with, and it makes no sense to conduct experiments away from this premise. Note that a contract may have multiple types of keywords, so after completing the filtering we remove the duplicate contracts and end up with 1673 contracts.

We applied the trained SCVHunter model to analyze 1673 contracts and identified 20 reentrancy, 27 block info dependencies, 18 nested calls, and 16 transaction state dependencies. To validate



(a) Reentrancy (b) Block Info Dependency

(c) Nested Call (d) Transaction State

Figure 11: ROC analysis of SCVHunter and its two variants on different vulnerability detection

Table 6: Comparison between SCVHunter and its variants on the four vulnerability detection tasks

Methods	Reentrancy				Block Info Dependency				Nested Call				Transaction State Dependency			
	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1	Acc	Recall	Precision	F1
SCVHunter-WCN	87.74	84.38	79.26	81.74	84.73	84.16	79.74	81.89	80.29	79.63	77.26	78.43	82.44	82.17	80.11	81.13
SCVHunter-HOG	80.99	79.86	73.62	76.61	76.62	79.17	70.44	74.55	73.35	71.47	70.23	70.84	74.11	76.39	69.82	72.96
SCVHunter	**93.72**	**90.60**	**88.84**	**89.71**	**91.07**	**89.43**	**84.41**	**86.85**	**85.41**	**82.69**	**82.33**	**82.51**	**87.37**	**83.42**	**83.61**	**83.51**

the results, we randomly selected 15 samples with vulnerabilities and 15 samples without vulnerabilities marked by SCVHunter for each vulnerability and manually reviewed them. For reentrancy, we found 0 false positive (FP) and 1 false negative (FN). For block information dependency, we found 2 FPs and 1 FN. For nested call, we found 1 FP and 2 FNs. For transaction state dependency, we found 2 FPs and 2 FNs. Upon analysis, we attributed these discrepancies to contracts with rare and complex control or assignment processes. For instance, conditional judgment variables associated with block.timestamp after 4 levels of assignment. We speculate that these complexities lead to overly complex generated CSGs, making it challenging to accurately learn the deep state automatically. We also used traditional tools to detect the discovered FPs and FNs. we found that a rule-based approach is more effective for such rare cases. For example, Mythril has no error in nested call and transaction state dependency detection. However, due to the lack of support for reentrancies associated with delegatecall and block information dependencies other than timestamp, it has 1 FN for reentrancy vulnerabilities and 2 FPs for block info dependency vulnerabilities.

Answer to RQ5: The overrall true positive and true negative rates when SCVHunter is applied to real contract vulnerability detection are 91.67% and 90.00%.

6 RELATED WORK

Traditional smart contract vulnerability detection. Some research utilizes traditional program analysis techniques to detect specific smart contract vulnerabilities [22, 23, 32, 42, 46, 62, 70, 76]. Oyente [46] and Nova [32] utilize symbolic execution to detect four vulnerabilities by traversing different execution paths of smart contracts. ReDetect [76] and MAR [70] combine symbolic execution and other techniques for reentrancy vulnerability detection. Mythril [16] and SmartCheck [61] use pattern matching to detect vulnerabilities based on predefined rules. AChecker [23] infers access control vulnerability in smart contracts through static data flow analysis. Securify [20] employs formal verification with logical languages for smart contract security. Tolmach et al. [62] perform formal analysis to verify the security of DeFi smart contracts. In addition, some studies use taint analysis, fuzzing, and other methods [4, 13, 26, 35, 57, 69]. Contractfuzzer [35] and Hfcontractfuzzer [13] employ fuzzing technology to detect various vulnerabilities in smart contracts. SWAT [57] performs vulnerability detection based on the Smart Contract Weakness Classification by matching patterns in Solidity code. However, compared to SCVHunter, which only requires the user to specify core nodes, the effectiveness of all these methods relies on the expert rules written into the source code, limiting their accuracy and extensibility.

Neural network-based vulnerability detection. Many attempts have been made to use deep neural networks for detecting smart contract vulnerabilities [3, 25, 30, 31, 34, 40, 43, 55, 67, 73, 74, 77]. For instance, ContractWard [67] employs five machine learning algorithms for vulnerability detection. SmartConDetect [34] uses a pre-trained BERT model to detect vulnerable code patterns. Some approaches use LSTM models, such as Qian et al. [51] and Hu et al. [31], to analyze contract fragments, but they may overlook structural information. DR-DCN and TMP [79] represent functions as graphs and utilize graph neural networks for vulnerability detection. Cai et al. [3] combine sliced joint graphs and graph neural networks for contract vulnerability detection. Peculiar [73] employs data flow graphs and pre-training techniques, but their isomorphic graph loses some semantic information. MANDO [50] combines contract control flow and invocation graphs, but all nodes have the same status in the graph structure, hindering the highlighting of nodes contributing significantly to vulnerabilities. In comparison, our heterogeneous CSG contains more relevant information and effectively represents important nodes at the graph level. Coupled with our graph neural network designed with a two-layer attention mechanism, we can learn richer semantic features.

7 THREATS TO VALIDITY

The main threat to internal validity is that SCVHunter currently only supports single-contract vulnerability detection. To address this problem, we will extend it for cross-contract detection in the future by linking CSGs for multiple contracts using the externally called function property defined in CGIR (§4.2.2). Threats to external validity concern our dataset. SCVHunter requires a large amount of data for training to learn enough features, and this phase may limit the ability of this approach to discover new classes of vulnerabilities where traditional methods may work better. To mitigate such threats, we can reduce the reliance on labeled data through semi-supervised learning when less training data is available.

8 CONCLUSION

In this paper, we propose SCVHunter, a novel contract vulnerability detection framework that combines heterogeneous CSGs and a graph neural network with two-layer attention. SCVHunter improves accuracy by allowing users to specify core node with significant contributions to vulnerability. We utilize graph neural networks to learn rich semantic features from contract graphs with control and data flow. Extensive experiments show that SCVHunter outperforms traditional vulnerability detection tools and neural network-based alternatives.

9 ACKNOWLEDGEMENT

The authors thank the anonymous reviewers for their constructive comments. This paper is partially supported by National Natural Science Foundation of China (62332004, U22B2029, 72304121).

REFERENCES

[1] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2021. Eth2Vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*. 47–59.

[2] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

[3] Jie Cai, Bin Li, Jiale Zhang, Xiaobing Sun, and Bing Chen. 2023. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *Journal of Systems and Software* 195 (2023), 111550.

[4] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–43.

[5] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. DefectChecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2189–2207. https://doi.org/10.1109/TSE.2021.3054928

[6] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. https://doi.org/10.1109/TSE.2020.2989002

[7] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, Yan Cheng, and Xiaosong Zhang. 2022. SigRec: Automatic Recovery of Function Signatures in Smart Contracts. *IEEE Transactions on Software Engineering* 48, 8 (2022), 3066–3086. https://doi.org/10.1109/TSE.2021.3078342

[8] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. Towards saving money in using smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. 81–84.

[9] Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, and Xiaosong Zhang. 2020. Understanding ethereum via graph analysis. *ACM Transactions on Internet Technology (TOIT)* 20, 2 (2020), 1–32.

[10] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1503–1520. https://doi.org/10.1145/3319535.3345664

[11] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[12] Monika di Angelo and Gernot Salzer. 2023. Consolidation of Ground Truth Sets for Weakness Detection in Smart Contracts. *arXiv preprint arXiv:2304.11624* (2023).

[13] Mengjie Ding, Peiru Li, Shanshan Li, and He Zhang. 2021. Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection. In *Evaluation and Assessment in Software Engineering*. 321–328.

[14] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[15] Ethereum. 2018. Function Modifiers. Retrieved Octorber 7, 2022 from https://docs.soliditylang.org/en/v0.8.17/contracts.html#function-modifiers

[16] Ethereum. 2018. Mythril: Security analysis tool for EVM bytecode. Retrieved Octorber 7, 2022 from https://github.com/ConsenSys/mythril

[17] Ethereum. 2018. Units and Globally Available Variables. Retrieved Octorber 7, 2022 from https://docs.soliditylang.org/en/v0.8.17/units-and-global-variables.html

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

[19] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[20] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. 2021. A survey on formal verification for solidity smart contracts. In *2021 Australasian Computer Science Week Multiconference*. 1–10.

[21] Asem Ghaleb and Karthik Pattabiraman. 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*. 415–427.

[22] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 728–739.

[23] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. *Proc. ACM ICSE* (2023).

[24] Christoph Goller and Andreas Kuchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, Vol. 1. IEEE, 347–352.

[25] Saroj Gopali, Zulfiqar Ali Khan, Bipin Chhetri, Bimal Karki, and Akbar Siami Namin. 2022. Vulnerability Detection in Smart Contracts Using Deep Learning. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1249–1255.

[26] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 557–560.

[27] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[28] Zheyuan He, Zhou Liao, Feng Luo, Dijun Liu, Ting Chen, and Zihao Li. 2022. TokenCat: detect flaw of authentication on ERC20 tokens. In *ICC 2022-IEEE International Conference on Communications*. IEEE, 4999–5004.

[29] Zheyuan He, Shuwei Song, Yang Bai, Xiapu Luo, Ting Chen, Wensheng Zhang, Peng He, Hongwei Li, Xiaodong Lin, and Xiaosong Zhang. 2022. TokenAware: Accurate and Efficient Bookkeeping Recognition for Token Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* (aug 2022). https://doi.org/10.1145/3560263 Just Accepted.

[30] Tianyuan Hu, Bixin Li, Zhenyu Pan, and Chen Qian. 2023. Detect defects of solidity smart contract based on the knowledge graph. *IEEE Transactions on Reliability* (2023).

[31] Teng Hu, Xiaolei Liu, Ting Chen, Xiaosong Zhang, Xiaoming Huang, Weina Niu, Jiazhong Lu, Kun Zhou, and Yuan Liu. 2021. Transaction-based classification and detection approach for Ethereum smart contract. *Information Processing & Management* 58, 2 (2021), 102462.

[32] Jianjun Huang, Jiasheng Jiang, Wei You, and Bin Liang. 2021. Precise Dynamic Symbolic Execution for Nonuniform Data Access in Smart Contracts. *IEEE Trans. Comput.* 71, 7 (2021), 1551–1563.

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

[34] Sowon Jeon, Gilhee Lee, Hyoungshick Kim, and Simon S Woo. 2021. Smartcondetect: Highly accurate smart contract code vulnerability detection mechanism using bert. In *KDD Workshop on Programming Language Processing*.

[35] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.

[36] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2020. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1695–1712.

[37] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[38] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[39] Min Li, Chunfang Li, Shuailou Li, Yanna Wu, Boyang Zhang, and Yu Wen. 2021. Acgvd: Vulnerability detection based on comprehensive graph via graph neural network with attention. In *International Conference on Information and Communications Security*. Springer, 243–259.

[40] Lin Liu, Wei-Tek Tsai, Md Zakirul Alam Bhuiyan, Hao Peng, and Mingsheng Liu. 2022. Blockchain-enabled fraud discovery through abnormal smart contract detection on Ethereum. *Future Generation Computer Systems* 128 (2022), 158–166.

[41] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 103–118. https://www.usenix.org/conference/nsdi23/presentation/liu-tianfeng

[42] Yiping Liu, Jie Xu, and Baojiang Cui. 2022. Smart Contract Vulnerability Detection Based on Symbolic Execution Technology. In *Cyber Security: 18th China Annual Conference, CNCERT 2021, Beijing, China, July 20–21, 2021, Revised Selected Papers*. Springer, 193–207.

[43] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. *arXiv preprint arXiv:2106.09282* (2021).

[44] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2021. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[45] Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad Reza Sadeghi, and Farinaz Koushanfar. 2021. ESCORT:

ethereum smart contracts vulnerability detection using deep neural network and transfer learning. *arXiv preprint arXiv:2103.12607* (2021).

[46] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 254–269. https://doi.org/10.1145/2976749.2978309

[47] Feng Mi, Zhuoyi Wang, Chen Zhao, Jinghui Guo, Fawaz Ahmed, and Latifur Khan. 2021. VSCL: automating vulnerability detection in smart contracts with deep learning. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.

[48] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[49] Nccgroup. [n. d.]. Decentralized Application Security Project. https://dasp.co/.

[50] Hoang H. Nguyen, Nhat-Minh Nguyen, Chunyao Xie, Zahra Ahmadi, Daniel Kudendo, Thanh-Nam Doan, and Lingxiao Jiang. 2022. MANDO: Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities. https://doi.org/10.48550/ARXIV.2208.13252

[51] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access* 8 (2020), 19685–19695.

[52] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-Level Liquid Types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 131–144. https://doi.org/10.1145/1706299.1706316

[53] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.

[54] Hasim Sak, Andrew W Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. (2014).

[55] Supriya Shakya, Arnab Mukherjee, Raju Halder, Abyayananda Maiti, and Amrita Chaturvedi. 2022. SmartMixModel: Machine Learning-based Vulnerability Detection of Solidity Smart Contracts. In *2022 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 37–44.

[56] Solidity. 2018. Solidity Document. Retrieved Octorber 7, 2022 from http://solidity.readthedocs.io/

[57] Nattawat Songsom, Warodom Werapun, Jakapan Suaboot, and Norrathep Rattanavipanon. 2022. The SWC-Based Security Analysis Tool for Smart Contract Vulnerability Detection. In *Proceedings of the 6th IEEE International Conference on Information Technology (InCIT)*. 1–6.

[58] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 555–571.

[59] SWC. [n. d.]. Smart Contract Weakness Classification. https://swcregistry.io/.

[60] Wesley Joon-Wie Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. 2018. Towards safer smart contracts: A sequence learning approach to detecting security threats. *arXiv preprint arXiv:1811.06632* (2018).

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

[62] Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. 2021. Formal analysis of composable DeFi protocols. In *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 149–161.

[63] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119.

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

[65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[66] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[67] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2021. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Transactions on Network Science and Engineering* 8, 2 (2021), 1133–1144. https://doi.org/10.1109/TNSE.2020.2968505

[68] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. 2019. Heterogeneous Graph Attention Network *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2022–2032. https://doi.org/10.1145/3308558.3313562

[69] Yajing Wang, Jingsha He, Nafei Zhu, Yuzi Yi, Qingqing Zhang, Hongyu Song, and Ruixin Xue. 2021. Security enhancement technologies for smart contracts in the blockchain: A survey. *Transactions on Emerging Telecommunications Technologies* 32, 12 (2021), e4341.

[70] Zexu Wang, Bin Wen, Ziqiang Luo, and Shaojie Liu. 2021. MAR: A Dynamic Symbol Execution Detection Method for Smart Contract Reentry Vulnerability. In *Blockchain and Trustworthy Systems: Third International Conference, BlockSys 2021, Guangzhou, China, August 5–6, 2021, Revised Selected Papers 3*. Springer, 418–429.

[71] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[72] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[73] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 378–389.

[74] Yingjie Xu, Gengran Hu, Lin You, and Chengtang Cao. 2021. A novel machine learning-based analysis model for smart contract vulnerability. *Security and Communication Networks* 2021 (2021), 1–12.

[75] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1029–1040.

[76] Rutao Yu, Jiangang Shu, Dekai Yan, and Xiaohua Jia. 2021. Redetect: Reentrancy vulnerability detection in smart contracts with high accuracy. In *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*. IEEE, 412–419.

[77] Jiale Zhang, Liangqiong Tu, Jie Cai, Xiaobing Sun, Bin Li, Weitong Chen, and Yu Wang. 2022. Vulnerability Detection for Smart Contract via Backward Bayesian Active Learning. In *Applied Cryptography and Network Security Workshops: ACNS 2022 Satellite Workshops, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, SiMLA, Rome, Italy, June 20–23, 2022, Proceedings*. Springer, 66–83.

[78] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2444–2461.

[79] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network.. In *IJCAI*. 3283–3290.

[80] Bo Zong, Qi Song, Martin Renqiang Min, Wei Cheng, Cristian Lumezanu, Daeki Cho, and Haifeng Chen. 2018. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In *International conference on learning representations*.