Interaction-Aware Vulnerability Detection in Smart Contract Bytecodes

Wenkai Li, Xiaoqi Li, Yingjie Mao, Yuqing Zhang

***Abstract*—The detection of vulnerabilities in smart contracts remains a significant challenge. While numerous tools are available for analyzing smart contracts in source code, only about 1.79% of smart contracts on Ethereum are open-source. For existing tools that target bytecodes, most of them only consider the semantic logic context and disregard function interface information in the bytecodes. In this paper, we propose COBRA, a novel framework that integrates semantic context and function interfaces to detect vulnerabilities in bytecodes of the smart contract. To our best knowledge, COBRA is the first framework that combines these two features. Moreover, to infer the function signatures that are not present in signature databases, we propose SRIF, automatically learn the rules of function signatures from the smart contract bytecodes. The bytecodes associated with the function signatures are collected by constructing a control flow graph (CFG) for the SRIF training. We optimize the semantic context using the operation code in the static single assignment (SSA) format. Finally, we integrate the context and function interface representations in the latent space as the contract feature embedding. The contract features in the hidden space are decoded for vulnerability classifications with a decoder and attention module. Experimental results demonstrate that SRIF can achieve 94.76% F1-score for function signature inference. Furthermore, when the ground truth ABI exists, COBRA achieves 93.45% F1-score for vulnerability classification. In the absence of ABI, the inferred function feature fills the encoder, and the system accomplishes an 89.46% recall rate.**

***Index Terms*—Ethereum, Bytecode, Smart contract, Function signature, Security**

I. Introduction

Detecting vulnerabilities in smart contracts is a crucial task in blockchain systems, which are distributed ledgers that publicly record transactions. Until May 2024, there are about 66 million deployed contracts [2], while only around 1.19 million are open source to the public [3], accounting for approximately 1.79% of the total. With the advent of the smart contract layer, Ethereum has gained enhanced functionality. However, as the use of smart contracts proliferates, numerous fragile code snippets are exploited maliciously. For example, the reentrancy vulnerability that led to a 3.6M ETH loss in the DAO event [4], and the error of arithmetic that caused an $80M loss in Compound Finance [5].

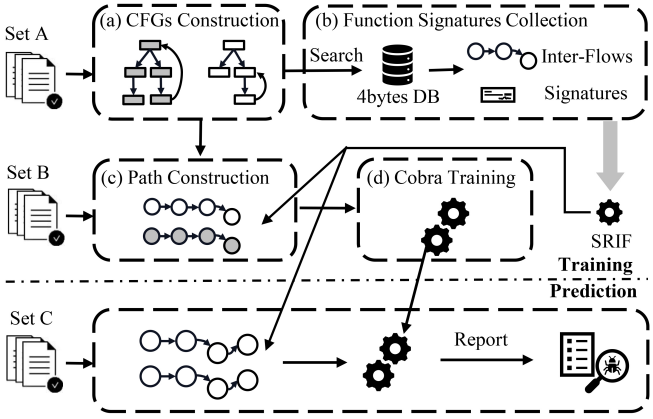

Fig. 1: Overview of Framework.

Previous studies have leveraged several dynamic and static methods to detect contract vulnerabilities. Symbolic execution (e.g., [6][7]), fuzz testing (e.g., [8][9]), and taint analysis (e.g., [10]) are viable attempts for detecting smart contract vulnerabilities. Moreover, they rely on the control flows within functions to some extent. Symbolic execution leverages formal methods to analyze all variables, including function parameters, to calculate potential sequences of vulnerabilities mathematically [6]. Fuzzing tests attempt to expose flaws in the contract by using unreasonable inputs, but the presence of function interfaces may diminish their effectiveness [8]. Taint analysis tracks and identifies whether tainted source data will be maliciously processed to expose a vulnerability, and functional interfaces can be the source of the taint [10]. Recent advancements in technology and the availability of large datasets have given rise to new approaches, such as machine learning, which has been used to analyze transactions and accounts to detect Ponzi schemes (e.g., [11][12]), and other vulnerabilities (e.g., [13][14][15]).

However, these tools utilized the semantic execution sequence of the source code. They do not prioritize the role of function interfaces in their detection process, even when analyzing bytecode. In addition, the source code of smart contracts can be transformed into bytecode and application binary interface (ABI) after compilation. Therefore, inspired by the deployment and interactions of contracts, we jointly learn the semantic features of contracts and the function interfaces to detect malicious contracts.

In this paper, we propose COBRA, as demonstrated in Figure 1, a deep learning-based framework that *first* integrates seman-

Wenkai Li, Xiaoqi Li, Yingjie Mao are with the School of Cyberspace Security, Hainan University, Haikou, 570228, China. E-mail: cswkli@hainanu.edu.cn, csxqli@ieee.org, yingjiemao@hainanu.edu.cn;

Yuqing Zhang is with National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing, 100049, China. E-mail: zhangyq@nipc.org.cn.

Corresponding author: Xiaoqi Li

This manuscript is an extended version of our work [1]. It has been extended more than 40% over the ASE conference version, including: (1) Enhancement of the analysis of the experiments (§ IV). (2) Addition of case analysis with the exploits detected by our framework (§ IV). (3) Elaboration on the extensive discussion of existing literature (§ V). (4) Optimization of the deeper analysis of the detected vulnerabilities (§ V).

tic context and function interface for vulnerability detection. As a sequence-to-sequence learning structure, the encoder in COBRA consists of the following four key components:

Part (a): A semantic extraction process extracts the semantic context in static single assignment (SSA) format, utilizing control flow graph (CFG) construction from bytecode contracts.

Part (b): A function signatures recovery component, which contains the application binary interface (ABI), signatures collection, and SRIF. SRIF first collects public signatures for training, and then retrieves the undisclosed function interfaces.

Part (c): A path sequence construction function that concatenates semantics and function interfaces. Function interfaces and properties are connected to form a function embedding.

Part (d): A model training component that learns the vulnerability patterns from the semantic and function representations.

The main contributions of this paper are as follows:

- To the best of our knowledge, we are the *first* to propose SRIF utilizing a seq2seq structure to extract function parameters from the semantic context. Moreover, we infer the function properties by counting particular Opcodes, jointly mapping as a function feature (§ III-C, § III-D).
- As far as we know, we are the *first* to present COBRA that integrates semantic context and function interface features, generating an embedding of smart contracts. The embedding is used to classify vulnerabilities (§ III-E).
- We integrate inferred function features and semantic information to discover vulnerabilities. Experimental results show that over 94% F1-score can be implemented if raw ABI is available, and over 89% recall can be achieved with the inferred function feature (§ IV).
- We also open source some relevant datasets and codes at https://figshare.com/articles/dataset/22313074.

As the extended version of the conference paper [1], the following significant extensions are provided.

- We present the first empirical study of function parameter and compiler version distributions across the Ethereum blockchain, while exploring SRIF's detection performance for diverse compiler environments.
- To evaluate the utilization degree of computing resources by different models, we select LSTM, Transformer, and BERT to analyze the neuron coverage rate, output accuracy, and model parameters. It is found that LSTM could achieve a 6.95% point increase in accuracy with at least 8 times fewer parameters than Transformer and BERT.
- To explore the opcode distribution in the vulnerable contract bytecode, we leverage COBRA to analyze the opcode frequencies of 5 types of vulnerable contracts. Specifically, we remove the invalid instructions that do not concern data operations using SSA opcodes, enhancing the interpretability of the fragile contract bytecode.

The remainder of the paper is organized as follows. Section II provides the background of the paper and Section III details the implementation of COBRA. In Section IV, we show the experimental results to demonstrate the effectiveness of our proposed method. The discussion of COBRA is conducted in Section V. Finally, we review related literature in Section VI and conclude our work in Section VII.

```
1 contract Victim {
2    mapping(address => uint256) balances;
3    ...
4    function withdraw(address add, uint amount){
5       require(balances[add]>amount);
6       add.call.value(amount)();
7       balances[add] -= amount;
8    }
9 }
10 contract Attacker {
11    address victim;
12    function setAddr(address add) public{
13       victim = add
14    }
15    function attack() payable{
16 //deposit money on Victim with call operation
17       deposit_call(money)
18       victim.call(bytes4(keccak256("withdraw(
             address add, uint amount)")), money/2);
19    }
20    //fallback function
21    function () payable{
22       //Reentrancy
23       victim.call(bytes4(keccak256("withdraw(
             address add, uint amount)")), this.msg.
             value);
24    }
25 }
```

Listing 1: The Simplified Snippets of a Malicious Interaction

II. Motivation & Background

A. Motivation

The targeted malicious interaction in this paper is shown in Listing 1, providing an understandable source code format. In the Solidity contracts, the attacker initially records the victim's address in the `setAddr()` function (line 12). The attack starts at line 15, with the `Attacker` depositing funds into the `Victim` contract and retrieving half via a `call` operation. Since the `call` lacks a return function specification, execution proceeds to the fallback function (line 21) without altering the `balances`. Consequently, line 23 recursively retrieves funds regardless of the check at line 5, resulting in an error when the victim's balance is insufficient. In this process, an attack pattern exploiting vulnerabilities is automated into the `Attacker` contract. It harnesses the call operation to interact with compatible function interfaces, executing the attack logic.

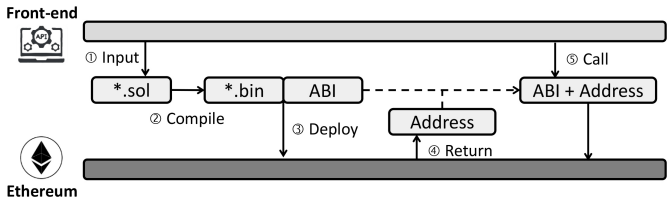

Fig. 2: Deployment and Interaction of a Contract. The ①-④ represent the whole deployment process of a contract; the ⑤ and the return of ④ constitute an interactive process.

As Figure 2 illustrates, the smart contract source code is compiled to produce bytecode and an application binary interface (ABI). The ABI specifies the standardized format for interactions with the contract, including information about functions (e.g., type, name, input parameters, output parameters, and properties). In the Ethereum virtual machine (EVM),

a function selector identifies the function signature from various functions based on the first four bytes of the CALLDATA. After the contract is deployed on the blockchain, its address is returned to the front end, which can then use the ABI to call specific contracts.

B. Background

1) Smart Contract: As a Turing-complete language, smart contract [16] is initially merged with blockchain in Ethereum. Ethereum's smart contract is compatible with various programming languages, including Solidity, Rust, Vyper, etc. The functional architecture permits smart contracts to communicate with other contracts. The contract is the code deployed on the blockchain, and the deployment process requires only a single transaction containing the compiled code[17]. Notably, it cannot be modified once the contract code has been released. After deployment, smart contracts can be interacted with each other by invoking with specified function signatures.

2) Ethereum Virtual Machine: The Ethereum virtual machine (EVM) is the execution environment for smart contracts, and nodes in the network can be connected through clients such as Geth [18]. EVM directly modifies the status information in the state database (StateDB) when a user account initiates a transfer request [19]. If an account submits a transaction request, EVM examines the data field in the message for a function entry to the contract based on the function signature. The interpreter converts the bytecode in StateDB to the Opcode to execute more advanced functionality [20]. The EVM opcodes occupy the hexadecimal bits 0x00-0xFF, with each byte containing only one opcode. These instructions can operate all types of data, including stack data (e.g., PUSH, POP), memory data (e.g., MSTORE, MLOAD), storage data (e.g., SLOAD, SSTORE). Further, it can perform arithmetic operations (e.g., ADD), jump the program counter (e.g., JUMP), and so on [21]. Moreover, all operations adhere to the gas mechanism [20]. Each Opcode necessitates a specific quantity of gas to execute. When the required amount of gas exceeds the threshold, the operation will be rolled back [21].

3) Function Signature: The function signature comprises the function's name and its arguments in the form of *functionName(param1, param2, ...)*. In the event of interactions between contracts, functional signatures become crucial. Since the function name can be defined arbitrarily, the function's behavior depends more on the number of arguments, the type of arguments, and the function id than on its name. The function id can be determined by applying the *Keccak-256* hash algorithm [22] to the function prototype string [23] and getting the first 4 bytes. Existing Function Signature libraries, such as the Ethereum Function Signature Database (EFSD) [24], are utilized to extract function ids for their function parameter types and numbers. The function hash is stored in the first four bits of the CALLDATA, and the called contract retrieves which function is called by extracting the function id. With the function hash in the CALLDATA, the EOAs or contract accounts can invoke the bytecode contract through the request from the front end.

4) Application Binary Interface: ABI is an interpreter designed to facilitate communication between bytecode smart contracts on EVM [8]. Since smart contracts are deployed with bytecode format in Ethereum, ABI decodes bytecode contracts into a human-readable language to facilitate interaction. Each ABI produces the following five components, 1) function types, 2) function names, 3) function input parameters, 4) function output parameters, and 5) function properties. The function types include *constructor*, *fallback*, and *receive*. In Ethereum, the *receive* type identifies a send/receive function, indicating that the function can receive and transfer Ether. A contract may contain only one *fallback* function with no parameters or return values. The *fallback* function is executed when the call request is not sent to any function of a contract. When a contract is created, its *constructor* function is called to initialize its state.

C. Related Vulnerabilities

When smart contracts expand the programmability of blockchain systems, security problems also increase. The Decentralized Application Security Project (DASP) [25] is a project classifying smart contract vulnerabilities based on actual impact. In this paper, we will concentrate on five of these vulnerabilities in Table I.

TABLE I: Related Vulnerabilities in DASP

Categories	**Alias**
Reentrancy Vulnerability	Recursive Call
Arithmetic Vulnerability	Overflow, Underflow
Unchecked Low Level Calls	Unchecked Send
Transaction Ordering Dependency	Front-Running, TOCTOU
Time Manipulation	Timestamp Dependency

A variety of works have been yielded for studying these attacks in Table I [6], [26]. The reentrancy can be discovered [4] when multiple recursive calls are made to withdraw assets before updating the balance state. The integer overflow, floating-point precision loss, and division by zero are all arithmetic vulnerabilities. Developers risk compromising the program's security if they fail to verify the variables' scope. Unchecked low-level calls occur when the return values of the calls are not effectively handled in the contract, resulting in coin loss [27]. The Transaction Ordering Dependency (TOD) is also know as Time-Of-Check vs Time-Of-Use (TOCTOU) [28]. By giving higher gas, the miners were incentivized to preempt other transactions, resulting in alterations to the initial states of the contract. A time manipulation vulnerability exists when a timestamp within a block is exploited to trigger a security event. The smart contract has access to the variables in block (e.g., `timestamp`, `difficulty`), the `block.timestamp` can be modified to cause unexpected issues when many contracts call it simultaneously [29].

D. Neuron Coverage

Neuron Coverage (NC) [30] quantifies the proportion of activated neurons in a neural network when processing a given test suite. Formally, it is defined as eq. (1),

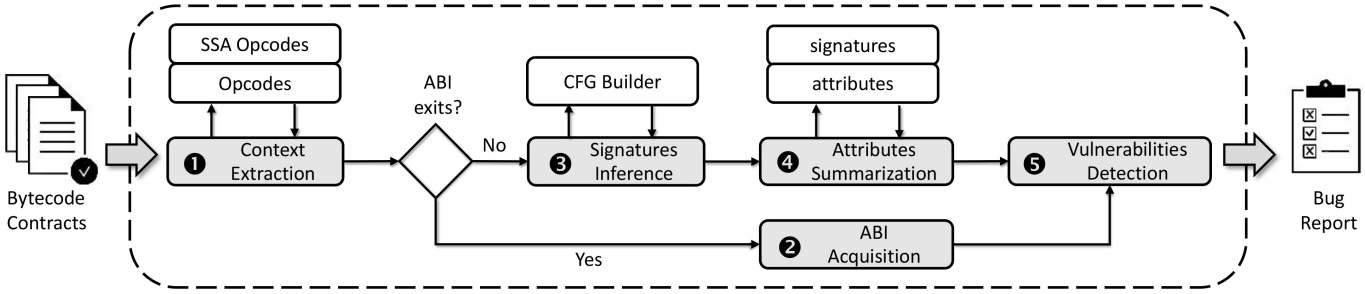

Fig. 3: The Architecture of Our Work. The dashed box represents the primary steps of the detection process. The inputs are the bytecode smart contracts, and the output is a vulnerability report.

$$NC = \frac{|n \in N | \forall i \in T, A(n,i) > \theta|}{|N|} \quad (1)$$

where N denotes the complete set of neurons in the network, T represents the collection of test inputs, $A(n,i)$ indicates the activation value of neuron n when processing input i, θ is the predefined activation threshold. A neuron is considered activated when its output value exceeds the specified threshold θ for any given test input. The metric ranges from 0 to 1, with higher values for a more complete neuron-level test coverage.

III. COBRA

In this section, we describe the primary methods taken to implement our two-stage approach, as well as the vulnerability detection model in Figure 3. As the input to our approach, the bytecode smart contract performs the following steps: ❶ *Context Extraction*, ❷ *ABI Acquisition*, ❸ *Signatures Inference*, ❹ *Attributes Summarization*, and ❺ *Vulnerabilities Detection*.

During operations in ❶-❷ or ❶-❸-❹, the dataset for detection model is processed. In step 1, the context information (i.e., Opcodes and SSA Opcodes) is extracted from the bytecode smart contracts. In the meantime, we crawl Etherscan [31] for its raw ABI data based on the address of the contract. If the ABI information is collected, both the semantic information and the ABI are represented as embedding, feeding our encoder module to obtain the contract's hidden representation. If the ABI data is absent, the steps ❸ and ❹ aim to infer the function signatures and attributes to recover the function features in ABI based on semantical context. The function signature includes the function name and parameters. More specifically, since there are only finite function signatures in EFSD, also known as 4BYTE, we propose the SRIF structure for inferring function signatures of contracts. The ❺ is the vulnerabilities detection module, where the COBRA with a novel encoder was presented to combine the processed semantic context and function representations of contracts. Finally, a bug report is generated, which contains classifications of the vulnerabilities in the bytecode smart contracts.

A. Context Extraction

We first extracted the Opcode in both original and SSA format from the bytecode smart contracts. By decompiling the bytecode with reverse engineering method [32], the Opcodes of contracts without compilation error can be gathered.

Algorithm 1: Functions Context and Ids Acquisition

input : A deployed bytecode smart contract bc
output: two global map: functions context $OpSeq$, functions hashes Ids

```
BasicBlocks, eb ← CFG.countBasicBlocks(EVMAsm(bc));
pushValue, prePushValue ← None;
Function getFuncInfo(block, entry):
    foreach instruction i of the block.ins do Ops ← i;
    if entry then
        if end of block compatible with JUMPI then
            Assert length of block.ins > 2;
            dest ← oprand of block.ins[-2];
            OpSeq[dest] ←getFuncInfo(BasicBlocks[dest], false);
            Ids[dest] ← None;
        return Ops ;
    for i in block.ins do
        if i compatible with PUSHs then
            prePushValue ← pushValue;
            pushValue ← oprand of i;
    if end of block compatible with JUMPI then
        if prePushValue then
            fnAddr, fnId ← pushValue, prePushValue;
    else
        fnAddr, fnId ← None;
    if fnAddr compatible with BasicBlocks then
        OpSeq[fnAddr] ← getFuncInfo(BasicBlocks[fnAddr], false);
        Ids[fnAddr] ← fnId;
        if end of block compatible with JUMPI then
            dest ← ((endPc ep of block) + 1);
            OpSeq[dest] ← getFuncInfo(BasicBlocks[dest], false);
    return Ops ;
foreach entryBlock address eb of the BasicBlocks do
    OpSeq[eb] ←getFuncInfo(BasicBlocks[eb], True);
```

SSA Opcode is an intermediate representation of Opcode that preserves the semantic information by removing data operations in the stack such as PUSH, POP, SWAP, and DUP [33]. Then, we construct the CFG in the execution order to obtain Opcodes. It entails separating the contract into basic blocks by searching for instructions about the end of basic blocks. For example, the Opcodes around jump (e.g., JUMP, JUMPI), and others (e.g., STOP, SELFDESTRUCT, RETURN, REVERT, INVALID, SUICIDE) will result in the stop of sequential execution. The resultant basic blocks will execute sequentially, beginning with the first instruction as the entry point and

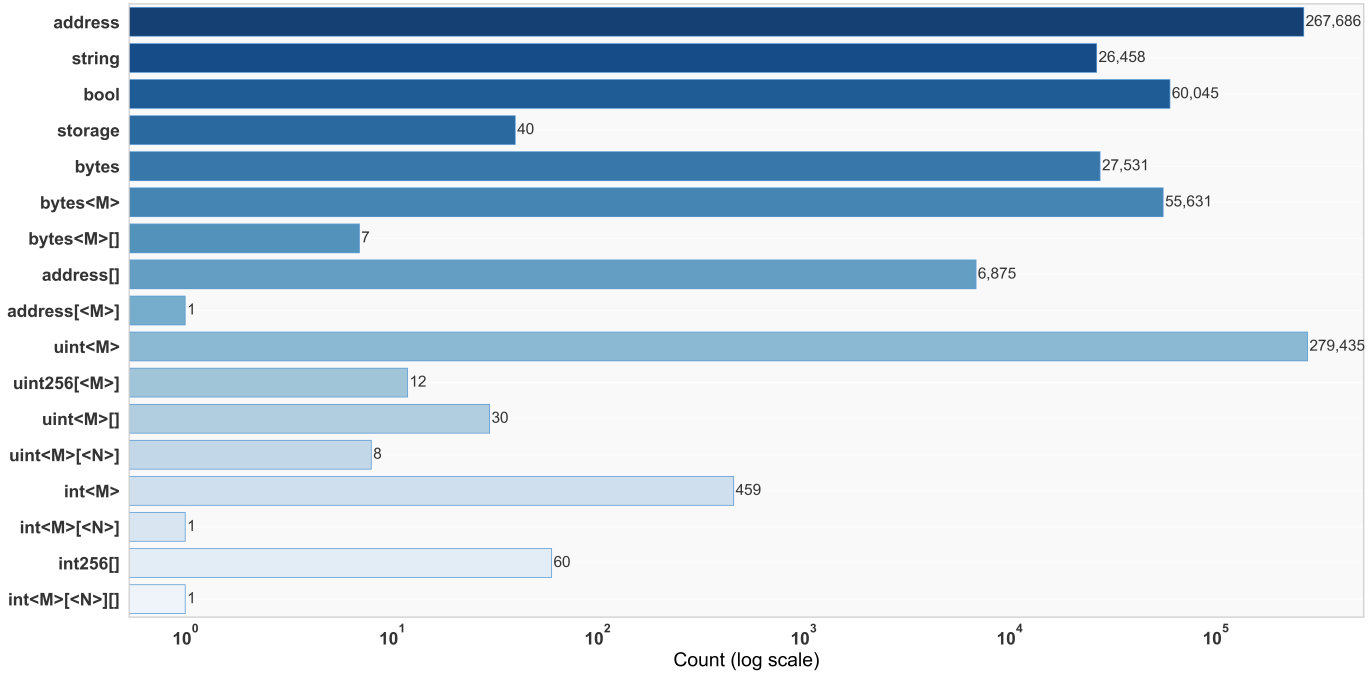

Fig. 4: The Label Distribution on the Ethereum Blockchain. The <M> and <N> are placeholders for a particular data length. The x-scale represents the number of occurrences of the type, which is measured in log to prevent long-tailed effects in the data plot. The y-scale is represented by all parameter types included in the statistics.

concluding with the last instruction as the outlet. The CFG is completed by constructing edges between the blocks based on the control flow. There are 3 types of basic block conversion, including conditional jump (e.g., JUMPI), unconditional jump (e.g., JUMP), and fall to next block. As a function block, the execution block must be processed from its entry. In addition, the function selector requires the block ending with JUMPI to obtain the function hashes, identifying different functions.

In the Algorithm 1, the presence or absence of the final instruction JUMPI is used to determine whether to proceed to the following function in the program. With the CFG construction, the function hashes and Opcodes of each sequentially executed function block are obtained. For each block, we evaluate its eligibility as an entry for a function. Specifically, on line 5, if the block is an entry and the end instruction is JUMPI, we extract the next function id. The Opcodes of the next function are then stored at $OpSeq$, and the address is the key. At line 12, if not an entry, we retrieve the last two push values `pushValue`, `prePushValue` within the block. On line 17, if the block ends with a JUMPI, the `pushValue` and `prePushValue` are used as the address and its hash value.

Furthermore, the precision of the EVM CFG builder [34] is crucial to the accuracy of the data we collected. For this reason, we keep an optimized EVM CFG recovery module in Elysium [35] to gather data for model training. The method in Elysium is proposed by [36] [37], which would extract more precise CFGs in this module.

B. ABI Acquisition

Etherscan, an Ethereum blockchain browser, allows us to crawl ABI information via contract addresses to examine real-time information such as blocks, transactions, miners, accounts, etc. Similarly to the accessibility of contracts, some bytecode contracts do not make their ABI information publicly available. To demonstrate this, we gathered 96,200 bytecode contracts in block-number order, of which 15,026 ABIs are available. Only 15.62% of these contracts have published ABI information. Therefore, for these bytecode contracts that do not disclose their ABI information, we propose an alternative approach in § III-C and § III-D for obtaining the function inputs and function attributes from the contracts. The format of the ABI stored in the blockchain is JSON. To ensure compatibility with the following format for function parameters and properties, we remove the function names.

Considering the label space of the function parameters, we collect the function interfaces in the first two million blocks on the Ethereum blockchain, and the number distribution of each parameter type is summarized. As Figure 4 shows, we only collected 17 types of parameters in total, and the address, string, bool, bytes, and uint types appeared more frequently.

C. Signature Inference

This section focuses primarily on deducing function parameters information from the function context obtained in the § III-A. Although 4BYTE contains publicly available function signatures, the library is incompatible with non-public function ids. In order to initially infer function parameters from bytecode information, we employ a structure called SRIF.

In accordance with the architecture of the EVM, this stack virtual machine does not store the runtime data, and the function and data operations are embedded within the Opcode. As described in § III-A, the function id would be saved in

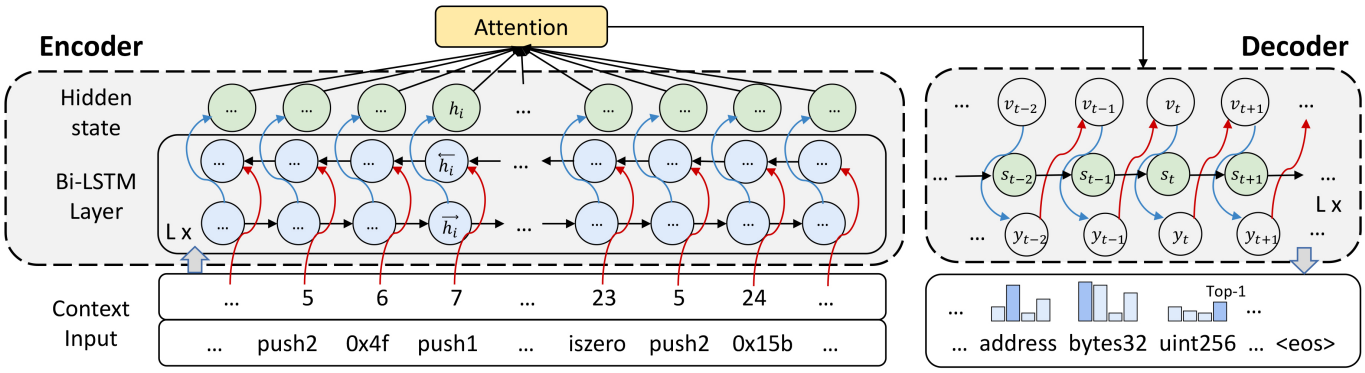

Fig. 5: The Signatures Inference Model, SRIF. The dashed boxes represent the encoder and decoder, which are connected by an attention module.

the call data and transferred into the called function. Some Opcodes can interact with call data, e.g., CALLDATALOAD, CALLDATASIZE, and CALLDATACOPY. Simultaneously, these data contain fixed rules for function parameter encapsulation, which can be inferred using specific rules. For instance, over 30 rules were included in SIGREC [23] for deducing function signatures from the bytecode of Solidity and Vyper. Such type-specific inferring rules may be affected by Solidity's version iterations for variable types, and Solidity has experienced more than 100+ released version updates over the past few years [38]. Therefore, to achieve a more adaptable process to get these input types in smart contracts, we utilize an encoder-decoder framework in Figure 5 to infer function parameters from the function context.

In § III-A, we have gathered the Opcodes of basic block and function id for each function. Subsequently, we employ CFGBuilder to establish the control flow graph (CFG) among all basic blocks. A depth-first search (DFS) algorithm with a designated depth was utilized to capture the sequence of contexts for all basic blocks within the function. We approached the task of parameter prediction as a multi-label classification (MLC) problem, considering the abundance of parameters and label categories involved. As shown in Figure 4, because the number of smart contract function parameter types is not high, the label space explosion problem of MLC does not occur.

Given the labels $L = \{l_1, l_2, ..., lm\}$, the primary objective is to generate the optimal sequence of label subset y^* from each sentence $\{w_1, w_2, ..., w_n\}$. A subset s of m labels is constructed from L to x. The task can be defined as maximizing $P(y|x)$, which can be computed by the following eq. (2).

$$P(y|x) = \prod_{i=1}^{n} p(y_i|y_1, y_2, ..., y_i, x) \tag{2}$$

In Figure 5, we take a context with n words $w_1, w_2, ..., w_n$ as example. As input, the word format must be converted to a machine-readable format. We collected possible words as vocabulary $|\nu|$. Suppose $i \in [0, n]$, where w_i is converted to a number, and then one-hot encoding is applied to w_i. An embedding matrix $E \in \mathbb{R}^{k \times |\nu|}$ extends each encoded w_i into a k-dimensional embedding vector e_i. Then the input context can be expressed as $c = \{e_1, e_2, ..., e_n\}$.

To extract the semantic information of each word bidirectionally, we adopt a recurrent neural network, LSTM [39], to acquire the context word meaning and its semantic features. The hidden state h_i of each word embedding vector e_i can be obtained by computing eq. (3), where h_i is the concatenation of the hidden states from both directions, indicating the ultimate representation of the i-th word.

$$h_i = [\overrightarrow{LSTM}(\overrightarrow{h_{i-1}}, c_i); \overleftarrow{LSTM}(\overleftarrow{h_{i-1}}, c_i)] \tag{3}$$

When invoking a function method, the number and sequence of parameters must be correct. Therefore, it is necessary to extract the correct number and order of labels from the original sentence features. In addition, the attention mechanism can identify valid words advantageous to the result from the input sentence. To predict the classification from the hidden states to the variable length, we employ a decoder module with an attention mechanism. Specifically, the context feature vector v_t obtained from attention at time step t is calculated as the eq. (4), eq. (5). Where w_{ti} is the weight of i-th word at t. $\boldsymbol{W_a^T}$, $\boldsymbol{U_a}$, $\boldsymbol{O_a}$ are determined during the training phase.

$$w_{ti} = softmax(\boldsymbol{W_a^T} \tanh(\boldsymbol{U_a} s_t + \boldsymbol{O_a} h_i)) \tag{4}$$

$$v_t = \sum_{i=1}^{n} w_{ti} h_i \tag{5}$$

where s_t is the hidden state of decoder at t, which can be defined as following eq. (6),

$$s_t = LSTM(s_{t-1}, [g(y_{t-1}); v_{t-1}]) \tag{6}$$

where the $g(y_{t-1})$ represents the label with the highest probability of distribution y_{t-1} at previous time step $t-1$. Notably, the probability distribution y_t is generated by a liner layer and a $softmax$ function. Specifically, the hidden states are transformed to the output size using a linear layer with an activation function. The resulting output is then passed through a $softmax$ function to obtain the probability distribution y_t.

In the training phase, we employ the focal loss function [40]. In our multi-classification task, an imbalanced category distribution may hinder the training process and prevent the model from converging to the extremes. The focal loss func-

tion was initially introduced for object detection tasks in the computer vision domain, where identifying positive and negative samples may present a wide disparity of difficulty. The focal loss is calculated as eq. (7).

$$loss = -\alpha(1-p_t)^{\gamma} log(p_t) \tag{7}$$

where the α controls the weight of positive and negative samples on loss, while p_t represents the probability of the ground truth category. γ is also a parameter that controls the value of $(1-p_t)$ to reduce the model's emphasis on easy-to-classify samples close to the ground truth. We followed the original assumption [40] that $\gamma = 2$ for focal loss. For the value of α, we made a few minor adjustments so that each class has a more balanced concentration. In eq. (8), the n_i is the number of types for the i-th parameter, $0 < i \leq T$, where T is the total number of classes. The α_i value for class i is $\sum_{j=0}^{T} n_j/n_i$. Each α_i is substituted for α to determine the loss value of each class, and the average value is the total loss.

$$loss' = \frac{\sum_{i=0}^{n} -\alpha_i(1-p_t)^{\gamma} log(p_t)}{n} \tag{8}$$

$$\alpha_i = \frac{\sum_{j=0}^{T} n_j}{n_i} \tag{9}$$

D. Attributes Summarization

Additionally, we deduce the attributes (i.e., the state mutability and payable) of functions in Solidity. State mutability indicates whether the states of a function can be updated, i.e., functions that are neither *pure* nor *view* type. *View* functions imply that the function state cannot be modified. Prior to the Solidity 0.5.0 version, *view* functions were referred to as *constant*. The *payable* property must be declared when a function transfers or receives Ether in Ethereum. To differentiate these properties and apply them to vulnerability detection, we classify them as *constant*, *pure*, and *payable*. According to Table II, we summarize the Opcodes that can modify state variables and transfer Ether. For each function, the context is examined for state modification operations and call messages with Ether. Consequently, the *constant* and *payable* properties are inferred, respectively. In particular, functions declared as *pure* neither modify nor read any state variables and, therefore, consume no gas. Based on Ethereum gas consumption[21], we get the Opcodes related to the *pure* property of functions.

TABLE II: The Table of States Operation Opcode

Checked Attributes	Related Behaviors	Analyzed Opcodes
View	Storage Modification	SSTORE
	Events Emitting	LOG0, LOG1, LOG2, LOG3, LOG4
	Child Contract Creation	CREATE, CREATE2
	Self-destruct	SELFDESTRUCT
	Low-level Calls	CALL, CALLCODE, DELEGATECALL
Payable	Transaction in Assets	CALLVALUE
Pure	Gas Consumption	STOP, RETURN, REVERSE

The *view* function is not permitted to alter the state variable, so we are supposed to identify statements that can modify the state variable. In Table II, we focus on storage modification, events emitting, child contract creation, self-destruct, and low-level calls at the Opcode level. SSTORE first reads the *key* and *value* from the stack and then writes the *value* at the *key* address, which might overwrite the storage, modifying the state variables of contracts. Moreover, the contract's transaction information is saved in the state variable. When an event is invoked, the arguments are written to the transaction log, causing a change in the states. Thus, LOG0, LOG1, LOG2, LOG3, LOG4 should be focused when emitting an event with different topics, even if LOGs do not affect the states. Additionally, creating or suiciding contracts will add or remove the storage and code of the contract, which is saved in the state variable. Hence, CREATE, CREATE2, and SELFDESTRUCT are identified as modifying the state variable. Furthermore, low-level calls (e.g., CALL, CALLCODE, and DELEGATECALL) are not permitted in the *view* function, and the STATICCALL Opcode is used to replace these calls, which prohibits states modification.

The keyword *payable* is mandatory in any function that involves asset transactions, where the type and amount of the asset depend on the message. After the Solidity 0.5.2 version, the CALLVALUE is used to obtain the value of the call.

The *pure* function cannot read or modify states. To ensure that the function consumes no gas, we identify all Opcodes that are disallowed in *pure* functions based on Ethereum's gas calculation. We exclude stack operations such as PUSH, POP, SWAP, and DUP, leaving only STOP, RETURN, and REVERSE as valid instructions for zero gas consumption [21].

E. Vulnerability Detection

In this subsection, the main structure of our detection model will be presented. As Figure 6 shows, the raw bytecode smart contract will first be processed by context extraction (§ III-A), signature inference (§ III-C) and attributes summarization (§ III-D). The sequence $X = \{x_1, x_2, ..., x_n\}$ contains n Opcodes. Using the rattle tool [33], we convert X to a SSA format $S = \{s_1, s_2, ..., s_m\}$. Since the stack operations are eliminated, and the Opcodes are arranged in the order of execution, more precise semantic information can be obtained. One-hot encoding is utilized to convert S into a k-dimensional embedding, and then feed a Bi-GRU layer to obtain the representations of the contract semantics in latent space, $h(j) = BiGRU(h_{j-1}, s_j)$, where $0 < j \leq k$. In Figure 6, the inferred function signature can be replaced with ABI information. When the ABI data is retrieved, it is converted into machine-readable form by looking up the vocabulary. Convolutional neural network (CNN) and average pooling layer transform the ABI into a function feature representation, before concatenating semantic and function representations to form the final feature representation of the contract.

If ABI cannot be obtained, we employ the method in § III-C to get the contract parameters for each function. Variation in the dimension of function signatures is inevitable, given multiple functions in each contract. Concurrently, the

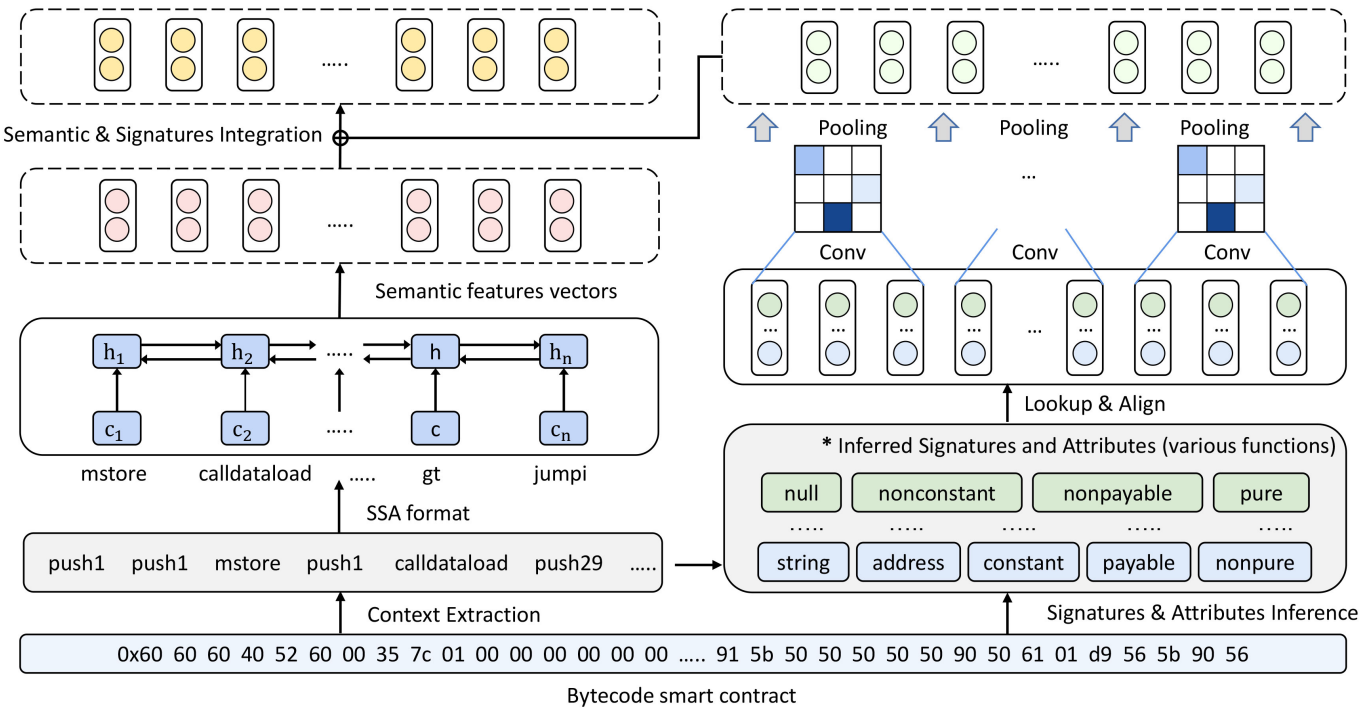

Fig. 6: The Encoder of our Detection Model. The bottom layer is the hex 2-gram representation of the bytecode smart contract, which is converted into two types of data: the Opcode sequence and the function data. * indicates the part that can be replaced with raw ABI information. The box with yellow dots at the top of the diagram reflects the representation in the latent space.

number of arguments varies, necessitating the alignment of sequences with varying lengths. Assume that each function parameter prediction process has z associated parameter types $T = \{t_1, t_2, ..., t_z\}$, and let F_i represent the i-th sequence of functions ordered by the first basic block address $\{type_1^i, type_2^i, ..., type_b^i\}$, containing a total of b arguments. Finally, we concatenate the sequence of parameters $\{attr_1^i, attr_2^i, attr_3^i\}$ for ap functions, where ap is the number of functions in the contract. In order to get a representation of the function signatures at the contract level, we cluster the functions of each contract and then feed a CNN and average pooling layer. The factor that selects CNN is to obtain local feature vectors for signature data, and its convenience is another factor in our task. Specifically, we use a lookup dict to convert each parameter to a uniform numeric format and blank padding to align these parameters from various functions.

Supposing that $g_{ij} = type_j^i$ is the i-th function of j-th parameter in contract X, where $0 < i \leq ap, 0 < j \leq n$. We first feed them to several convolutional layers to get the features of each function, $G_i' = \boldsymbol{W_1}[g_{i1}, g_{i2}, ..., g_{i(1+k-1)}; attr_1^i, attr_2^i, attr_3^i] + \boldsymbol{b_1}$ where the $\boldsymbol{W_1}, \boldsymbol{b_1}$ are the training parameters and k represents the kernel size. To obtain the hidden features among all the functions, MS-CAM [41] in Figure 7 is applied to obtain the features in the certain dimension d_1. MS-CAM was originally proposed for integrating features with different dimensions. We focus on the local features expressed by a certain function and the global features expressed by all functions. In this way, we can capture the related features of the individual functions. The local feature representation $feature_l$ can be expressed as the following eq. (10).

$$feature_l = N(Conv_2(\zeta(N(Conv_1(G))))) \tag{10}$$

where $G = \{g_1, g_2..., g_{ap}\}$, N denotes the normalization layer, $Conv_1$ denotes shrinking the input sizes on dimension d_1, while $Conv_2$ denotes expanding the size on d_1 back to its original size. ζ represents the $ReLu$ activation function. The global feature $feature_g$ and output are calculated as the eq. (11), eq. (12).

$$feature_g = GAP(feature_l) \tag{11}$$

$$output = G \otimes (\xi(feature_g \oplus feature_l)) \tag{12}$$

where the GAP represents a global average pooling layer, $\otimes$ means multiplication in the feature map, which is consistent with the MS-CAM. $\oplus$ denotes addition after adjustment, while broadcasting in MS-CAM. ξ represents the $Sigmoid$ activation function.

Then, a $ReLu$ and a pooling layer are utilized to get the feature representation of the function parameters and attributes, $G_i^* = ReLU(G_i')$, $G = Pool(G_1^*, G_2^*, ..., G_{ap}^*)$. Thus, the final feature representation of the contract X is $h_x = [h(1), ..., h(k); G]$.

Following our model's encoder, we obtain an implicit feature representation h_x incorporating contract semantic and function signature information. To demonstrate the efficacy of our encoder, we adopt the same architecture as the decoder of SRIF model in § III-C. We employ an attention structure for h_x, and a recurrent neural network is fed to decode the hidden states and identify the vulnerabilities. Moreover, since there is

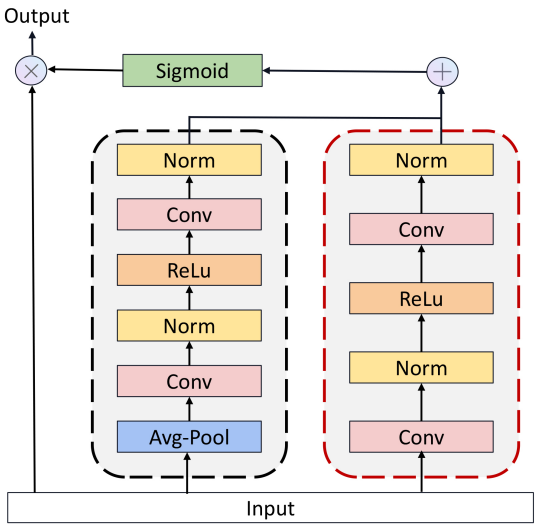

Fig. 7: The MS-CAM in Our Structure. ⊗ means multiplication, ⊕ denotes addition. The black dashed box represents the global feature, and the red dashed box represents the local feature.

no same label in each prediction, a mask mechanism is used to get distinct results at different time steps. For example, if y_j has the highest probability at time step $t-1$, the initial value of a label y_j is set to negative infinity at time t, and set the initial value of all labels except y_j to 0. To alleviate the class imbalance problem in the multi-class detection task, we use the focol loss function in the vulnerability detection process, which has been introduced in eq. (8) in § III-C.

IV. Experiments

In this section, we present the results of experiments to evaluate the performance of our framework, answering the following questions:

RQ1: **Why the RNN is chosen for SRIF and COBRA?**
RQ2: **Is SRIF effective for the function signature inference?**
RQ3: **Will SRIF be affected by different compiler versions?**
RQ4: **Is COBRA effective for the vulnerability detection?**
RQ5: **Is ABI or function signature in COBRA effective?**
RQ6: **Can COBRA detect new bugs in the real world?**

A. Experimental Setup

1) ***Datasets****:* To generate a labeled bytecode smart contract dataset I with a sufficient amount of ground truth for proper evaluation, we collect 13,948 deployed bytecode smart contracts from the XBLOCK_ETH dataset [42]. We use the SMARTBUGS [43] framework to identify vulnerabilities in these contracts and label them accordingly. Due to the fact that each part of the tool has its own specialized vulnerabilities, we utilize various state-of-the-art modules to collect as much accurate ground truth data as possible. For instance, we utilize the OYENTE to identify reentrancy, and MYTHRIL is maintained for arithmetic, unchecked low-level calls, and transaction ordering dependency vulnerabilities detection. Especially, the time manipulation is labeled by the CONKAS in SMARTBUGS, which is renewed by the community. In addition, we run each contract for a minimum of 30 minutes to ensure maximum reliability. After filtering out contracts that can not be detected due to version incompatibility and disassembly errors, we have obtained 8,267 processed contracts with their corresponding vulnerability labels.

Table III presents our dataset composition, comprising 790 analyzed contracts. Among these, we identified 790 instances of reentrancy vulnerabilities, 4,609 instances of arithmetic vulnerabilities, 1,764 cases of unchecked low-level calls, 1,351 transaction order dependencies, and 1,292 timing manipulation issues. Notably, 7,190 contracts were found to be vulnerability-free. The total amount of vulnerability exceeds the total number of contracts because most vulnerable contracts contain multiple different types of vulnerability.

TABLE III: The Label Distribution of the Dataset I.

Vulnerability Types	Existing Amount
Reentrancy Vulnerability	790
Arithmetic Vulnerability	4,609
Unchecked Low Level Calls	1,764
Transaction Ordering Dependency	1,351
Time Manipulation	1,292
No Vulnerability	7,190

Another dataset II, also derived from XBLOCK_ETH, contains only contracts in bytecode format without vulnerability labels. We collected a total of 6,024 contracts for the phase of function signature inference. Using the function Opcodes and hashes acquisition method described in § III-A, we gathered the function ids present in these contracts. These function signatures were then matched against the 4byte [24] database. Finally, 99,745 function signatures, along with the corresponding Opcodes, were collected.

2) ***Evaluation Metrics****:* We use F1-score, precision, and recall as evaluation metrics. The precision is the likelihood of each classification being accurately identified. Recall indicates the probability of discovering all possible results. The F1-score represents the harmonic mean of precision and recall.

3) ***Environments****:* Two experimental environments exist in the whole experiments, (1) Intel(R) Xeon(R) W-2255 CPU + 256GB RAM + 2 × GeForce RTX 3090 with the operating system of Windows Server 2019 and (2) Intel(R) Core(TM) I7-12700 CPU and 32GB RAM with the system of Ubuntu 20.04. In (1), we labeled contracts with vulnerability classes and trained models; in (2), we collected all bytecode smart contracts used in our framework from the XBLOCK_ETH.

B. RQ1: Why is the RNN chosen for SRIF and COBRA?

Motivation: The reason why we choose the LSTM rather than other models (e.g., Transformer, and BERT) is (1) computational resource efficiency, and (2) robust handling of variable-length sequences.

Approach: To comparatively assess model utilization efficiency across identical data conditions, we evaluate neuron coverage performance on the dataset described in § IV-A1. Our experiments involve training 3 distinct architectures, i.e.,

Transformer, BERT, and LSTM, on dataset II, followed by coverage computation using the designated test partition. The dataset is divided following a 60% : 40% training and test ratio to ensure consistent evaluation conditions. Note that the conventional notion of a "neuron" requires careful interpretation in RNNs, as these architectures generate vector-based hidden states rather than discrete neuronal outputs [44]. Thus, we conduct coverage analysis by examining the hidden state vector at the testing layer. For Transformer and BERT, we follow [45] setting the learning rate l = 0.0001, hidden size h = 128, batch size β = 64, the number of attention heads $heads$ = 4, and the number of Transformer layers L = 4. We set the threshold θ = 0 in the neuron coverage, which means that all the neurons whose parameter value is not 0 are valid.

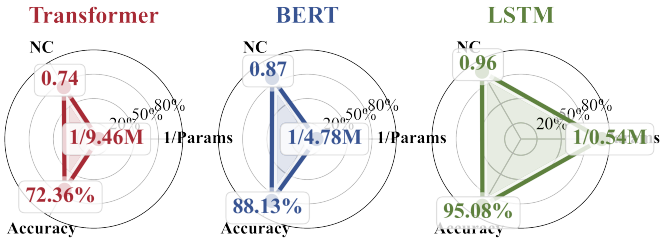

Fig. 8: The Neuron Coverage Comparison of Different Model Types. The red color represents the experimental data of the Transformer architecture, the blue color represents the experimental data of the BERT architecture, and the green color represents the experimental data of the LSTM. The three dimensions NC, Accuracy, and 1/Params are neuron coverage, accuracy, and the inverse of parameter amount, respectively.

Result: Figure 8 demonstrates the performance variations across Transformer, BERT, and LSTM when evaluated along three critical dimensions: neuron coverage (NC), detection accuracy (Accuracy), and parameter efficiency (i.e., 1/Params). LSTM achieves superior performance with 0.96 neuron coverage and 95.08% detection accuracy, outperforming BERT's 88.13% by 6.95 percentage points and Transformer's 72.36% by 22.72 percentage points. Furthermore, LSTM maintains the performance advantage while demonstrating significantly greater parameter efficiency 1/0.54M, exceeding BERT's efficiency 1/4.78M by over 8 times and Transformer's 1/9.46M by around 17.5 times. The experimental data shows that LSTM can achieve better detection results with fewer model parameters in our dataset and tasks.

Answer to RQ1. LSTM demonstrates superior neuron coverage, detection accuracy, and parameter efficiency relative to Transformer and BERT architectures.

C. *RQ2: Is* SRIF *effective for the function signature inference?*

Motivation: We first verify the effectiveness of SRIF on function signature data. It serves as the foundational element for ensuring the efficacy of COBRA.

Approach: To evaluate the effectiveness of various network structures, we divide the dataset II into training, validation, and test sets with proportions of 60%, 20%, and 20%, respectively. After training, performance results from various model structures are collected during validation, and the best network structure is evaluated on the test set. Due to the fact that each function call procedure is composed of distinct basic blocks, the flow between each block is uncertain and diverse, resulting in multiple branches. Therefore, we employ the DFS algorithm to obtain the Opcode of each function in the executing flow. To determine the optimal depth, we compare the instances of 1, 2, and 3 depths, respectively. Furthermore, we compare the SRIF with Gigahorse [46] on a subset of the test set. The contracts are compiled manually in the Gigahorse tool and their reverse recovered function signature results are collected. It is worth noting that at this stage, we strictly control the number and order of function parameters, and when the number and order are inconsistent, we consider that the function signature decision fails.

Result: We compare the LSTM, GRU cells, and different depths in SRIF. According to the results in Table IV, the ideal results are obtained when the depth is 1. In light of this result, we presume that the information most relevant to function parameters is stored in the first basic block of the function, which is the location of the function entry. The results indicate that the LSTM owns 95.46% F1-score, which has a more favorable performance and is better suited for the stage of function inference.

TABLE IV: The Results of Function Parameter Inference in Different Depths and Cells with Focal Loss Function

Network Structures				
Cells	**Max Depths**	**F1-socre**	**Precision**	**Recall**
GRU	3	93.19%	90.51%	96.04%
LSTM	3	95.23%	94.14%	96.36%
GRU	2	94.85%	93.75%	95.96%
LSTM	2	95.20%	93.99%	96.44%
GRU	1	95.18%	94.16%	96.23%
LSTM	**1**	**95.46%**	**94.17%**	**96.78%**

Furthermore, we use the cross entropy loss function for training, obtaining the F1-score, precision, and recall rate of 94.46%, 93.81%, and 95.62%, respectively. Note that the cell and depth are LSTM and 1 separately. The result reveals that the focal loss function has a certain improvement effect than the cross entropy loss function.

The model for function parameters inference consists of 540,771 parameters when LSTM and focal loss function is employed. As Table V shows, SRIF can achieve 94.76% F1-score, 93.49% precision, and 96.06% recall, which indicates that it can achieve high performance in function signature inference of smart contracts.

TABLE V: The Measures of Function Parameters Inference.

Metrics for Testing	**SRIF Performance**
Test Precision	93.49%
Test Recall	96.06%
Test F1-Score	94.76%

Moreover, we randomly select 12 contracts (a total containing 120 functions) in the test set, comparing the SRIF with the Gigahorse. Due to the problem of time consumption, only

part of the contracts are selected in this paper. During the selection, all the contracts in the test set are randomly shuffled and divided into 12 equidistant intervals, and the contracts are randomly selected from each interval. This sufficient randomness gives some validity to the results. In this process, we find that Gigahorse can successfully recover 98 function signatures, but SRIF successfully recovers 110 function signatures.

Answer to RQ2. The SRIF achieves 94.76% F1-score on the test set, and it can recover more function signatures than Gigahorse.

D. RQ3: Will SRIF be affected by different compiler versions?

Our evaluation of SRIF's effectiveness utilizes a comprehensive dataset of unoptimized, open-source smart contracts compiled across multiple Solidity versions. The dataset construction leverages two key resources: (1) version metadata from Etherscan [31] and (2) compilation verification through the official EVM Solc compiler. From the initial two million blockchain blocks, we extracted 13,948 reliability-focused contracts. These represent 85 distinct Solidity compiler versions spanning v0.4.11 through v0.8.30.

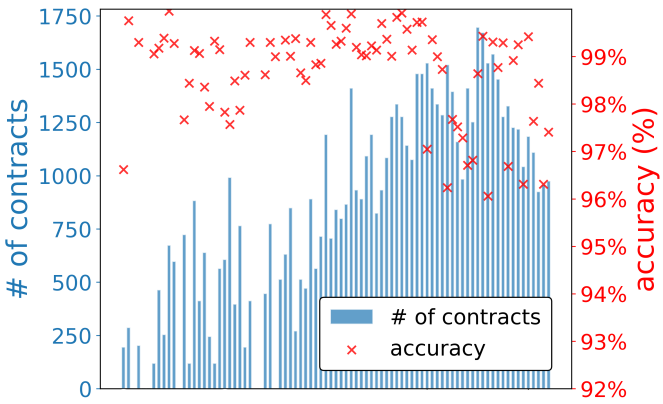

Fig. 9: Performances of SRIF for Different Solidity Compiler Versions. The blue bars represent the number of contracts under different compiler versions, and the × indicates the inferred accuracy under different compiler versions.

For each compiler version, we evaluate SRIF's accuracy performance. Figure 9 presents these results, sorted by Solidity version in ascending order, along with the corresponding number of contracts per version, ranging from 1 to 1750. Our experimental results demonstrate SRIF's robust performance across all tested compiler versions. In the Solidity compilers, the tool maintains consistently high accuracy, with no observed case falling below 96% across all 85 versions. This finding confirms that SRIF's accuracy remains stable regardless of compiler version evolution.

Answer to RQ3. The evaluation results indicate that SRIF maintains stable performance across different compiler configurations. It achieves consistent accuracy, with ≥96% success rate for all 85 Solidity compiler versions tested.

E. RQ4: Is COBRA effective for the vulnerability detection?

Motivation: We evaluate COBRA's ability to detect vulnerabilities in our collected dataset.

Approach: We utilize the dataset I, which contains 8,267 labeled contracts. We start by conducting comparisons among various LSTMs, GRU cells, loss functions, and other tools using the validation set. Subsequently, we assess COBRA's performance on the test set to derive the final evaluation results. Additionally, we explore a scenario where only inferred function signatures are accessible. The F1-score serves as the metric, representing the harmonic mean of precision and recall, thereby evaluating the COBRA, Mythril [43], and MANDO-GURU [47]. The recall signifies the percentage of identified malicious classes among the actual malicious classes, evaluating when only inferred signatures are available.

Result: As Table VI shows, under the combination of context and ABI, the GRU with focal loss function can achieve the best F1-score (94.26%) in our evaluation. Notably, our model can achieve even higher recall on cross-entropy. One of the possible reasons is that the focal loss function in our experiments might result in more training for the more challenging classes, reducing generalization for the simpler categories. Nevertheless, GRU can perform better than LSTM. Furthermore, after we compare Mythril and MANDO-GURU with COBRA, we find that COBRA can achieve the best performance as shown in Table VII.

TABLE VI: The Performance Comparison of Different Cells and Loss Functions.

Network Structures		F1-socre	Precision	Recall
Cells	**Loss Function**			
LSTM	Cross Entropy	92.38%	90.82%	94.01%
LSTM	Focal Loss	87.96%	84.82%	90.49%
GRU	Cross Entropy	93.17%	90.40%	96.13%
GRU	**Focal Loss**	**94.26%**	93.12%	95.42%

TABLE VII: The Comparison Results of COBRA and SOTA.

Name	F1-score
Mythril	65.57%
MANDO-GURU	91.06%
COBRA	94.26%

When combining context, the global features of ABI with GRU, and the focal loss function, the model contains a total of 3,130,115 parameters. In the test phase, the results in Table VIII show that 93.45% of the F1-score, 91.56% of the precision, and 95.42% of the recall score can be obtained. Moreover, we utilize the SRIF with attributes summarization method to generate alternative function information for contracts that do not expose ABI. As a reminder, the method needs to support the ability to discover as many vulnerabilities as possible. Therefore, we take recall as the metric for detection. After testing, Table VIII shows the recall can reach 89.46%.

Answer to RQ4. The COBRA achieves 93.45% F1-score on the test set, and 94.26% F1-score on the validation, which outperforms other methods.

TABLE VIII: The Measures of Vulnerabilities Classification.

Metrics for Testing	COBRA Performance
Test Reminder Recall	89.46%
Test Precision	91.56%
Test Recall	95.42%
Test F1-Score	93.45%

F. RQ5: Is ABI or function signature in COBRA *effective?*

Motivation: We conduct experiments to explore scenarios where different function interfaces are available, i.e., application binary interface (ABI) or function signatures.

Approach: To demonstrate that ABI information is valuable for vulnerability detection, we summarize the case of distinct contract semantics and the addition of ABI separately. The experiments involved with ABI are conducted by the combination structure of LSTM with cross-entropy loss function. Furthermore, regarding the function signatures, we make a comparison of different network structures. When no public ABI is available, inferred function signature representations and semantic features are used as latent features of contracts. Since we expect the COBRA with only function signatures to discover as many malicious classes as possible, recall is utilized to evaluate this situation.

TABLE IX: The Performance Comparison of Different Composite Structures.

Network Structures	F1-socre	Precision	Recall
Context	76.27%	70.21%	83.47%
Context + ABI	**92.38%**	**90.82%**	**94.01%**
Context + ABI + MS-CAM	**92.14%**	**91.34%**	**92.96%**

Result: Table IX summarizes the case of distinct contract semantics and the addition of ABI separately. Note that all the data in Table IX is done by the combination structure of LSTM with cross-entropy loss function. Table IX provides insight into numerous conclusions. First, general results can be obtained using only context or SSA Opcode. The relatively low precision indicates that the features cannot effectively specify the vulnerabilities. In addition, after combining ABI, we compared the different ABI features using the global feature and the whole MS-CAM. The results presented indicate that using only global feature extraction can improve F1-score and recall, while MS-CAM can enhance precision.

TABLE X: The Comparison of Different Architectures for Vulnerability Detection with Inferred Function Signatures.

Structures	LSTM		GRU	
	—	MS-CAM	—	MS-CAM
Recall	87.79%	89.01%	89.78%	**90.92%**

As shown in Table X, the MS-CAM can obtain better performance when combined with GRU, and the recall value can increase to 90.92%. Therefore, it is evident from the results in Table X that the combination of local and global features of function features is more beneficial for vulnerability detection. The GRU and modified MS-CAM can improve the detection.

Answer to RQ5. The ABI and function signatures can improve COBRA's detection performance to a certain degree.

G. RQ6: Can COBRA *detect new bugs in the real world?*

Motivation: We discuss the vulnerability detection capability of COBRA and analyze the vulnerability we detected.

Approach: We test in Xblock-ETH except for the contracts included in § IV-A1, which are detected using COBRA. These contracts exist on the Ethereum mainnet. During this process, COBRA uses GRU and focal loss function. When the ABI is not publicly available, we add the MS-CAM module.

Result: We identify two previously undiscovered interactive vulnerabilities (CVE-2023-36979 and CVE-2023-36980), which can not be detected by Mythril, Oyente, and MANDO-GURU. An illustration of a potential contract vulnerability is presented in Listing 2, which could be exploited. The contract depicts a casino game developed on the Ethereum blockchain. The global variable instance Casino has the uint balance, which ranges from 0 to $2^{256} - 1$. If the user needs to play the game, they need to store tokens into `casino.balance` according to the `casinoDeposit()` function.

```
function casinoDeposit() {
    if (msg.sender == casino.addr)
        casino.balance += msg.value;
    else
        msg.sender.send(msg.value);
}
// Bet on Number
function betOnNumber(uint number) public returns
     (string) {
    // Input Handling
    address addr = msg.sender;
    uint betSize = msg.value;
    if (betSize < casino.bettingLimitMin ||
        betSize > casino.bettingLimitMax) {
        // Return Funds
        if (betSize >= 1*10**18)
            addr.send(betSize);
        return "Please choose an amount";
    }
    if (betSize * 36 > casino.balance) {
        // Return Funds
        addr.send(betSize);
        return "Casino has insufficient funds";
    }
    if (number < 0 || number > 36) {
        // Return Funds
        addr.send(betSize);
        return "Please choose a number";
    }
    // Roll the wheel
    privSeed += 1;
    uint rand = generateRand();
    if (number == rand) {
        uint winAmount = betSize * 36;
        casino.balance -= (winAmount - betSize);
        addr.send(winAmount);
        return "Win!";
    }
    else {
        casino.balance += betSize;
        return "Wrong number.";
    }
}
```

Listing 2: The Simplified Snippets of **Casino**

We can find that it only limits the situation of guessing correctly, and when it is correct, the money can be withdrawn. However, for the 35 lines of correct guessing at Listing 2,

the `send()` is used to transfer the money amount. If the transfer is not successful, it will only return false and will not block execution, so the return value needs to be checked. However, the owner does not check the return value in the contract. Moreover, If the gas exceeds 2300, the transaction will fail. But the `fallback()` function needs at least 2300. Therefore, if the attacker designs the contract to call `betOnNumber(uint number)` using the `call` method in an attack function, and then jumps to the `fallback` function when the call ends. It causes the gas to exceed 2300, causing the `send()` function to fail.

Impact: Based on CVSS (Common Vulnerability Scoring System) assessments [48], the analyzed vulnerabilities are categorized as medium-risk severity. As for CVE-2023-36979, we will introduce the details in the § V-C. It originates from improper authorization mechanisms in low-level call operations, enabling malicious manipulation of the contract's balance state variable, which subsequently disrupts expected payment distributions to users due to flawed contract logic. Meanwhile, CVE-2023-36980 in Listing 2 stems from insufficient validation checks on monetary-related parameters, permitting unauthorized alteration of transfer values that ultimately result in permanent financial losses for contract participants. Both security vulnerabilities demonstrate significant financial implications by exposing user assets to substantial risks of exploitation and irreversible damage.

Answer to RQ6. Utilizing COBRA, we find two previously undisclosed vulnerabilities, i.e., CVE-2023-36979 and CVE-2023-36980.

V. Discussion

A. Properties of COBRA

COBRA achieves high performance by combining functional interfaces and contract semantics features. Compared to other deep learning technologies, i.e., MANDO-GURU, COBRA benefits from greater information features through its function interfaces. In SRIF and COBRA, we use the RNN to achieve the sequential dependency of opcode execution.

When learning the semantic features of a contract, both SRIF and COBRA employ a bidirectional RNN structure to apprehend the opcode sequence features stemming from the sequential execution of a contract. In addition, SRIF uses RNN architecture to sequentially decode the function interface parameters. Given the relatively limited variety of function parameters in Ethereum smart contracts, the label space during decoding is not excessively vast.

The deep learning architecture implemented in both COBRA and SRIF provides inherent extensibility for emerging signature and vulnerability patterns. When novel categories are identified, the system accommodates them through two straightforward modifications: (1) incorporation of additional training samples representing the new type, and (2) dynamic adjustment of the focal loss α parameter at eq. (9) to maintain appropriate class weighting based on updated vulnerability frequency distributions.

B. Runtime Overhead

COBRA needs to obtain function interface information first, and then integrate it with contract semantics.

When using ABI data exposure, this paper only needs to input it into the model at the same time as the contract code, which will not affect the time consumption of COBRA.

When the ABI data is not public, through the experiments of § IV-C in this paper, the time taken by the SRIF model to parse the function signature of a function is about 0.5667 seconds. This has a negligible impact on the time of the model in this paper. Moreover, in our dataset, the length of opcodes of several smart contracts exceeds 16,600. They are quite complex, leading to over 30 minutes of analysis by the symbolic execution tool Mythril. COBRA detected the entire test set only took a few minutes, performing relatively better. More importantly, the function-attribute interface summarization module can be determined to search for a fixed element from a specified sequence, even if the time complexity of the sequential search algorithm is linear $O(n)$, and does not consume too much time.

We compared the time consumption of Mythril, MANDO-GURU, and COBRA in the test set, but the results had a huge difference, so we did not intentionally record the data. We used Mythril to analyze the test set, which took several hours. In particular, when analyzing contracts with opcode lengths over 7300, it took almost ten minutes each. However, the COBRA detected the entire test set only took a few minutes (with fair CPU usage). It may be caused by the natural advantages of deep learning, making the timing too different. For the MANDO-GURU, a single contract takes several minutes (with fair CPU usage).

C. Case Study

1) Arithmetic Vulnerability: In Figure 10, we present the statistics of arithmetic vulnerabilities and summarize each contract's SSA Opcode by measuring dataset in [49][43].

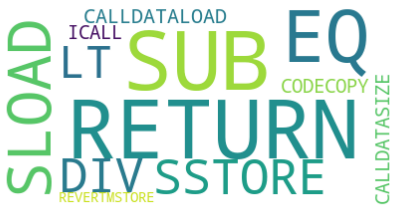

Fig. 10: The Frequencies of SSA Opcodes in Arithmetic Vulnerabilities

To identify the Opcodes associated with the arithmetic vulnerability, we performed certain operations to eliminate the Opcodes (e.g., CALLDATALOAD), which were almost present in any contract. We examined the frequency of each Opcode in each contract, dividing the total number of occurrences by the total number of contracts. To eliminate the effect of generic Opcodes, we select Opcodes with values close to 1, which can access data appearing in every weak contract. Figure 10 illustrates that numerous data operation instructions

(e.g., SUB, EQ, DIV, and LT) exist. We also collect input parameters for these functions. Considering that our framework only detects integer overflows and underflows, we collect the function parameters and attributes containing integer types. Our function inference method demonstrates that 67.86% of function contracts with integer-type parameters exist. In the meantime, these functions are not *view* type, which has included instructions for modifying the state variables (e.g., `address.balance`, `block`, `tx`, and `msg`).

```
1 contract Roulette {
2   uint ps;
3   struct Casino {
4     address addr;
5     uint balance;
6   }
7   Casino casino;
8   function bet(uint num) public {
9     address addr = msg.sender;
10    uint betSize = msg.value;
11    ps += 1;
12    uint rand = genRand();
13    uint randC = (rand + 1) % 2;
14    if (rand != 0 && (randC == num)) {
15      casino.balance -=  betSize * 2;
16      addr.send(betSize * 2); }
17    else { casino.balance += betSize; }
18  }
19  function genRand() private returns (uint) {
20    ps = ((ps * 3 + 1) / 2) % 10 ** 9;
21    uint bn = block.number;
22    uint d = block.difficulty;
23    uint t = block.timestamp;
24    uint g = block.gaslimit;
25    uint rand = (ps + bn + d + t + g) % 37;
26    return rand;
27  }
28 }
```

Listing 3: The Simplified Snippets of **Roulette**

In Listing 3, we give an example of the arithmetic vulnerability that we detected. Lines 11 and 17 of the code exhibit the potential for integer overflows, but we will focus on line 17 in this analysis. On line 3, a *struct* with an *address* and a *uint* type is defined and stored in *memory*. The variables `ps` and `balance` are unsigned integral numbers with a default range of $[0, 2^{256}-1]$. The input parameter of the function `bet` is a *uint*. When `balance` and `betsize` are out of range, such as when the `balance` is $2^{256}-1$, even a `betsize` of 1 can cause the `casino` instance to overflow, resulting in significant economic loss. Furthermore, the overflow in line 11 causes `ps` to become zero, and as a result, its random seeds can be deduced. As demonstrated in this example, it is challenging to identify whether an overflow has occurred by solely examining the SSA Opcode sequence. If the overflow in the function does not modify the state, it may result in a logic error without causing any direct economic loss. Therefore, our approach incorporates function parameters and function state types to determine whether a state operation occurs.

2) Time Manipulation Vulnerability: We have also identified another type of vulnerability in the contract in Listing 3, the time manipulation vulnerability. Despite the contract's reliance on extensive calculations to generate pseudo-random numbers, malicious miners can obtain the block information, including the timestamp, and disclose it to attackers. It may enable attackers to derive the random numbers from the calculation method in the source code and subsequently exhaust tokens from the contract.

Fig. 11: The Frequencies of SSA Opcodes in Time Manipulation Vulnerabilities

In Figure 11, we present a summarization of the SSA Opcodes associated with the timestamp manipulation vulnerability. The Opcodes include TIMESTAMP, which indicates that the contract accesses the block dependency of timestamp attribute; CALLDATALOAD, which retrieves the call data; SSTORE, which stores data to storage; and DIV, which denotes data manipulation. These instructions indicate that contracts with these vulnerabilities are likely to read call data and timestamp, process the data, and modify the storage. By leveraging function properties, we can determine whether a function tends to modify state variables. Furthermore, we conducted a comprehensive analysis of all state variables. Our findings indicate that 90.42% of contracts are not *payable*. 92.57% of the functions that contain the payable type modify the state variables.

3) Unchecked Low-Level Call Vulnerability: Additionally, we have distinguished unchecked low-level call vulnerabilities in Listing 3. Specifically, on line 16, the contract's transfer method utilizes the `send`, which is considered a low-level call. This approach sends tokens or ETHs to another contract address and returns a boolean value indicating the success or failure of the transaction. However, the contract does not include any checks on the transaction status, which will result in an incomplete or unsuccessful transaction. Although the vulnerability in this contract does not directly affect the balance of the contract, it can negatively impact users when the contract does not send tokens or ETHs. Therefore, we consider it to be a true positive case.

Fig. 12: The Frequencies of SSA Opcodes in Unchecked-Low-Level Call Vulnerabilities

We conducted an analysis of Solidity assembly Opcodes depicted in Figure 12 and found that the ADDRESS and BALANCE Opcodes are frequently utilized in these contracts with unchecked-low-level calls. These Opcodes are instrumental in capturing the address and balance of contracts, respectively, and are often used in functions that read state variables. Furthermore, we surveyed a significant disparity of over

4.27 times between the number of payable and non-payable functions in the analyzed contracts. Among *payable* functions, 54.68% have modified the state, while 45.32% have read state variables. At the Opcode level, low-level calls in Solidity can be categorized into three types: CALL, DELEGATECALL, and CALLCODE. However, in the semantic context gathered in the SSA format, DELEGATECALL and CALLCODE are not frequently detected. Consequently, analyzing fragile contracts containing such low-level calls (e.g., dangerous delegatecall) would yield true negatives.

4) Transaction Ordering Dependency: Exploiting a transaction ordering dependency (TOD) vulnerability largely depends on the execution order. When the vulnerability occurs, indicating that the transaction is profitable, state variables and balance-related operations may be shared between the two transactions. Consequently, examining the SSTORE and BALANCE Opcodes is necessary, as SSTORE stores the state variables, and BALANCE retrieves the account's balance. Line 14 of Listing 3 also reveals the possibility of TOD. If transaction t_1 wins the bet by transferring the correct value, transaction t_2 could front-run the t_1. Mythril failed to identify the TOD, which caused the report to be a false positive.

5) Reentrancy Vulnerability: As described at § IV-G, the reentrancy vulnerability in our database shows a tendency. In this issue, some operations, such as utilizing transfer and send methods, require the *payable* type, which is associated with the CALLVALUE instruction. Additionally, when using the `call.value`, the attacker can access the target function's signature by utilizing the SHA3 instruction. Furthermore, an examination of the reentrancy vulnerability reveals that certain Opcodes, such as CALLER, are frequently utilized. Consequently, as evidenced by the results depicted in Figure 13, it is reasonable to conclude that `msg.caller` and `msg.value` are commonly present in the vulnerable contract.

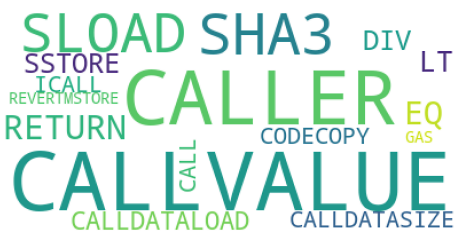

Fig. 13: The Frequencies of SSA Opcodes in Reentrancy Vulnerabilities

An illustration of a potential contract vulnerability is presented in Listing 4, which could be exploited through reentrancy. Specifically, When tokens are sent to an address without checking the balance, attackers can repeatedly call the function and exhaust its resources. To address this issue, protective measures, such as utilizing transfer and send methods, require the *payable* type, which is associated with the CALLVALUE instruction. Additionally, when using the `call.value` function call, the attacker can access the target function's signature by utilizing the SHA3 instruction. Furthermore, an examination of the reentrancy vulnerability reveals that certain Opcodes, such as CALLER, are frequently utilized. Consequently, as evidenced by the results depicted in Figure 13, it is reasonable to conclude that `msg.caller` and `msg.value` are commonly present in the vulnerable contract.

```
1 contract PrivateBank {
2   mapping (address=>uint) public balances;
3   function CashOut(uint _am) {
4     if(_am<=balances[msg.sender]) {
5       if(msg.sender.call.value(_am)()){
6         balances[msg.sender] -= _am; }
7     }
8   }
9 }
```

Listing 4: The Simplified Snippets of **PrivateBank**

D. Limitations

This section describes the limitations of our research, which is restricted by time, reverse engineering, and raw data.

1) Time: The two-stage framework is presented for detecting vulnerabilities in bytecode-level smart contracts, which is necessary due to the limited availability of ABI information. In the presence of raw ABI data, we can extract the ABIs from Etherscan, while in the absence of raw ABI data, we employ the SRIF function to infer the function signatures and properties of the contracts. However, this inferred data does not contain the output parameters and types of functions. The extraction of ABI, function parameters, and attributes must be completed before the execution of COBRA, with function parameters and attributes being inferred using the SRIF. As a result, the construction of the framework requires a great deal of time. We have taken the first step toward resolving this issue by proposing a deep learning-based framework for function signature inference. In the future, we will combine this method with our encoder to build an end-to-end deep learning framework for improved vulnerability detection.

2) Reverse Engineering: For the structure of SRIF and COBRA, they both rely on reverse engineering of the EVM. First, they might be affected by the accuracy of Opcode acquisition. For Opcode, we use the PYEVMASM module to facilitate integration development directly in Python programming language. However, this module also results in a loss of precision. It does not support all versions of SOLC, which is the Solidity compiler in Ethereum. On the other hand, SRIF also relies on CFG recovery in both original and SSA format Opcodes. In our implementation, the CFG module in EtherSolve is utilized for inferring more precise CFG, which verifies whether the CFG contains unreachable basic blocks to improve accuracy. Although we could improve the accuracy of CFG and reduce the impact of noise to a certain extent, the volume of data required for deep learning in SRIF and COBRA is still substantial.

3) Raw Data: In the absence of raw ABI, our approach is limited to providing alerting functionality only. In the 4byte signature library, there are far more than 1 million unique records. Nevertheless, according to our survey, only around 15.62% of 96,200 contracts are publicly available to ABI. As the experiments (§ IV) demonstrated, even though our function parameter inference method SRIF could achieve relatively good results, there is still an apparent limitation in vulnerability detection compared to the raw ABI. Therefore,

we have a relatively reduced capacity for detecting contracts that keep ABI private. However, as the recovery capability of the function interface is optimized and the amount of ABI disclosed increases, we would have a significant potential for improving our approach.

E. Threat to Validity

Internal validity. The contracts in XBlock_ETH are from the Ethereum blockchain, where the ground truth labels for function signatures are derived from 4bytes (See § IV-A1), collected from real-world functions. Therefore, it is suitable for evaluation. Additionally, COBRA relies on reverse engineering the Ethereum virtual machine. For bytecode data, it needs to be converted to an opcode sequence. However, there are errors in the current method of obtaining opcode sequences, and different versions of compilers will lead to accuracy loss. Even some popular compiler tools such as pyevmasm [50] do not support all versions, resulting in a lack of accuracy. However, COBRA exploits an intermediate representation in the SSA form and thus alleviates inaccurate translations of opcode sequences. In addition, the EtherSolve tool [36] is used in this paper to achieve the accurate construction of the CFG graph to reduce invalid basic blocks in the recovery process.
External validity. SRIF has some randomness when compared with Gigahorse. We split the test set into equal numbers of groups and randomly selected contracts from each group, ensuring as fair a comparison as possible.

F. Future Direction

This section outlines potential enhancements for subsequent research. First, our current implementation of SRIF focuses solely on input parameters while excluding return value specifications in function signatures. The ablation study presented in §IV-E reveals that incorporating ABI output data enhances vulnerability detection recall by approximately 2.06 percentage points. This finding motivates planned extensions to integrate output parameter analysis into SRIF's framework. Second, while both SRIF and COBRA employ recurrent neural network architectures for computational efficiency rather than the Transformer-based models, the emergence of advanced large language models (LLMs) presents new opportunities. Future investigations will evaluate the feasibility of adapting state-of-the-art LLMs (e.g., GPT-4, Gemini, Claude) for simultaneous function signature inference and vulnerability identification, contingent upon overcoming current problems, including computational constraints, model robustness, error bounds, and explainability of predictions.

VI. Related Work

A. Smart Contract Vulnerability Detection

Many state-of-the-art works based machine learning have been presented for vulnerability detection [11][26][13][14][51] [15][52][53][54][55][56][57][58][45][59][60], as the increasing of smart contract bugs and machine learning technology. Chen et al. [11] use machine learning methods to detect Ponzi schemes. Ether flow graphs are constructed by analyzing Opcode and account information, and features are designed for classifying source code contracts. He et al. [26] implement the fuzz function through the GRU module and combine it with symbolic execution techniques to achieve higher coverage. Gao et al. [13] transform the code into an abstract syntax tree (AST) and then serialize the tree based on the nodes. After learning the feature vector of the sequence, the vector threshold is used to determine whether the feature is a vulnerability. So et al. [14] collect enough sequences of vulnerabilities through symbolic execution to train a language model. The tool detects Ether leaking and suicidal contracts in source code. Huang et al. [51] utilized an unsupervised graph embedding algorithm to embed the sliced CFG in the graph and then performed similarity calculations on the sliced vectors. Sendner et al. [15] proposed ESCORT, a multi-label detection tool that supports lightweight transfer learning. Different from these above works, COBRA takes the advantages of the function interface into account when detecting vulnerabilities in smart contracts.

B. Function Signature Recovery

Different from Java Virtual Machine (JVM), EVM does not retain function signature information in bytecode data. Call data stores function parameters that can only be accessed via particular processing. Numerous ways utilize these databases to recover function signatures by developing parameter acquisition procedures (e.g., Gigahorse [46], Eveem [61]). Gigahorse introduces the "CALLPRIVATE" directive to identify private function calls. Eveem, utilizing symbolic execution techniques, performs symbolic and algebraic computations of the execution trace. When a layout containing *offset* and *num* fields is found, it is regarded as an array. SigRec [23] searches for call data in execution traces related to CALLDATACOPY and CALLDATALOAD and creates specific inference rules to recover various function signatures. Furthermore, there are also methods for locating function information without establishing rules (e.g., OSD [62], Neural-FEBI [63], and DeepInfer [64]). Typically, OSD searches directly in EFSD [24] for function hashes to discover function signatures. Neural-FEBI identifies functions via a two-step process to get a more precise CFG.

VII. Conclusion

We present COBRA, a novel framework for detecting vulnerabilities in Ethereum smart contracts at the bytecode level. COBRA employs semantic and function interface features for vulnerability detection. Additionally, we introduce the SRIF, a function signature recovery technique that handles cases where ABI is not disclosed. We also conduct minor adjustments to MS-CAM to learn both global and local features for functions in each contract. Our experiments demonstrate that the SRIF can accurately and efficiently predict function parameters, achieving a 94.76% F1-score on the dataset of 99,745 signatures. COBRA equipped with publicly available ABI exhibit 93.45% F1-score in vulnerability detection. However, even without publicly available ABI, the recall rate remains above 89%. Therefore, recovering ABI information remains a crucial consideration, and further research into more effective techniques is necessary.

VIII. ACKNOWLEDGMENTS

This work is sponsored by the National Natural Science Foundation of China (No.62362021 and No.62402146), CCF-Tencent Rhino-Bird Open Research Fund (No.RAGR20230115), and Hainan Provincial Department of Education Project (No.HNJG2023-10).

REFERENCES

[1] W. Li, X. Li, Z. Li, and Y. Zhang, "Cobra: Interaction-aware bytecode-level vulnerability detector for smart contracts," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1–12.

[2] Google, "Bigquery - ethereum dataset." 2025. [Online]. Available: https://console.cloud.google.com/bigquery?p=bigquery-public-data\&d=crypto_ethereum\&page=dataset

[3] O. Martin and E. Shayan, "Smart contract sanctuary." 2025. [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary-ethereum

[4] V. Buterin, "Critical update re: Dao vulnerability." 2025. [Online]. Available: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability

[5] B. Staff, "Compound finance mis-rewarded around $80m worth comp to the users." 2025. [Online]. Available: https://rekt.news/compound-rekt

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter." in *Proceedings of the ACM SIGSAC conference on computer and communications security (CCS)*, 2016, pp. 254–269.

[7] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, and et al., "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts." in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189.

[8] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection." in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 259–269.

[9] V. Wüstholz and M. Christakis, "Targeted greybox fuzzing with static lookahead analysis." in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, p. 789–800.

[10] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018, pp. 1–15.

[11] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, "Detecting ponzi schemes on ethereum: Towards healthier blockchain technology." in *Proceedings of the World Wide Web Conference (WWW)*, 2018, pp. 1409–1418.

[12] H. Hu, Q. Bai, and Y. Xu, "Scsguard: Deep scam detection for ethereum smart contracts." in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2022, pp. 1–6.

[13] Z. Gao, "When deep learning meets smart contracts." in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1400–1402.

[14] S. So, S. Hong, and H. Oh, "Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution." in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 1361–1378.

[15] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, and et al., "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning." in *Proceedings of the Network and Distributed System Security Symposium(NDSS)*, 2023, pp. 1–18.

[16] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds." in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 161–178.

[17] S. Steffen, B. Bichsel, and M. Vechev, "Zapper: Smart contracts with data and identity privacy." in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, p. 2735–2749.

[18] S. Adam, S. Péter, and W. Jeffrey, "Go ethereum:official go implementation of the ethereum protocol." 2025. [Online]. Available: https://geth.ethereum.org

[19] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: Uncovering vulnerabilities in wasm smart contracts." in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analys (ISSTA)*, 2022, pp. 703–715.

[20] A. Ghaleb, J. Rubin, and K. Pattabiraman, "etainter: Detecting gas-related vulnerabilities in smart contracts." in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 728–739.

[21] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger." *Ethereum project yellow paper.*, vol. 151, no. 14, pp. 1–32, 2014.

[22] T. S. Authors, "Solidity documentation–contract abi specification." 2025. [Online]. Available: https://solidity.readthedocs.io/en/latest/abi-spec.html

[23] T. Chen, Z. Li, X. Luo, X. Wang, T. Wang, Z. He, and et al., "Sigrec: Automatic recovery of function signatures in smart contracts." *IEEE Transactions on Software Engineering.*, vol. 48, no. 8, pp. 3066–3086, 2022.

[24] 4byte, "Ethereum signature database." 2025. [Online]. Available: https://www.4byte.directory

[25] N. Group, "First iteration of the decentralized application security project top 10." 2025. [Online]. Available: https://dasp.co/index.html

[26] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts." in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 531–548.

[27] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum." *IEEE Transactions on Software Engineering.*, vol. 48, no. 1, pp. 327–345, 2022.

[28] N. Ivanov, Q. Yan, and A. Kompalli, "Txt: Real-time transaction encapsulation for ethereum smart contracts." *IEEE Transactions on Information Forensics and Security.*, vol. 18, pp. 1141–1155, 2023.

[29] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited." in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 1325–1341.

[30] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 1–18.

[31] Etherscan, "The ethereum blockchain explorer." 2025. [Online]. Available: https://etherscan.io

[32] Crytic, "Pyevmasm's documentation." 2025. [Online]. Available: https://pyevmasm.readthedocs.io/en/latest

[33] ——, "rattle." 2025. [Online]. Available: https://github.com/crytic/rattle

[34] ——, "evm_cfg_builder." 2025. [Online]. Available: https://github.com/crytic/evm_cfg_builder

[35] C. Ferreira Torres, H. Jonker, and R. State, "Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts." in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defense (RAID)*, 2022, pp. 115–128.

[36] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode." in *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 127–137.

[37] A. Benini, M. Ceccato, F. Contro, M. Crosara, M. Dalla Preda, and M. Pasqua, "Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode." *Journal of Systems and Software.*, p. 111653, 2023.

[38] A. Beregszaszi, K. Śliwak, and et al., "Solidity releases." 2025. [Online]. Available: https://blog.soliditylang.org/category/releases

[39] S. Hochreiter and J. Schmidhuber, "Long short-term memory." *Neural computation.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[40] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection." in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2980–2988.

[41] Y. Dai, F. Gieseke, S. Oehmcke, Y. Wu, and K. Barnard, "Attentional feature fusion." in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2021, pp. 3559–3568.

[42] P. Zheng, Z. Zheng, J. Wu, and H.-N. Dai, "Xblock-eth: Extracting and exploring blockchain data from ethereum." *IEEE Open Journal of the Computer Society.*, vol. 1, pp. 95–106, 2020.

[43] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts." in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 530–541.

[44] W. Huang, Y. Sun, X. Zhao, J. Sharp, W. Ruan, J. Meng, and X. Huang, "Coverage-guided testing for recurrent neural networks," *IEEE Transactions on Reliability*, vol. 71, no. 3, pp. 1191–1206, 2021.
[45] T. Wang, X. Zhao, and J. Zhang, "Tmf-net: Multimodal smart contract vulnerability detection based on multiscale transformer fusion," *Information Fusion*, vol. 122, pp. 103 189–103 204, 2025.
[46] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts." in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1176–1186.
[47] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, "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 (ESEC/FSE)*, 2022, pp. 1736–1740.
[48] FIRST, "Common vulnerability scoring system," 2025. [Online]. Available: https://www.first.org/cvss/
[49] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion." in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI)*, 2021, pp. 2751–2759.
[50] Crytic, "Pyevmasm's documentation." 2025. [Online]. Available: https://github.com/crytic
[51] J. Huang, S. Han, W. You, W. Shi, B. Liang, J. Wu, and et al., "Hunting vulnerable smart contracts via graph embedding based bytecode matching." *IEEE Transactions on Information Forensics and Security.*, vol. 16, pp. 2144–2156, 2021.
[52] Q. Kong, J. Chen, Y. Wang, Z. Jiang, and Z. Zheng, "Defitainter: Detecting price manipulation vulnerabilities in defi protocols," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analys (ISSTA)*, 2023, pp. 531–548.
[53] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 716–727.
[54] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," in *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1–12.
[55] Y. Chen, Z. Sun, Z. Gong, and D. Hao, "Improving smart contract security with contrastive learning-based vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–11.
[56] X. Wang, S. Tian, and W. Cui, "Contractcheck: checking ethereum smart contracts in fine-grained level," *IEEE Transactions on Software Engineering*, vol. 50, no. 7, pp. 1789–1806, 2024.
[57] L. Chen, H. Wang, Y. Zhou, T. Wong, J. Wang, and C. Zhang, "Smarttrans: Advanced similarity analysis for detecting vulnerabilities in ethereum smart contracts," *IEEE Transactions on Dependable and Secure Computing*, 2025.
[58] C. Liu, Z. Sang, L. Duan, J. Wang, W. Ni, and W. Wang, "Anomaly detection services for blockchain smart contracts with unknown vulnerabilities," *ACM Transactions on Software Engineering and Methodology*, 2025.
[59] X. Xie, H. Wang, Z. Jian, Y. Fang, Z. Wang, and T. Li, "Blockgram: Mining knowledgeable features for efficiently smart contract vulnerability detection," *Digital Communications and Networks*, vol. 11, no. 1, pp. 1–12, 2025.
[60] Y. Huang, S. Fang, J. Li, B. Hu, J. Tao, and T. Zhang, "Deep smart contract intent detection," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2025, pp. 124–135.
[61] T. Kolinko, "Eveem/panoramix–showing contract sources." 2025. [Online]. Available: https://eveem.org
[62] Ethervm, "Online solidity decompiler." 2025. [Online]. Available: https://ethervm.io/decompile
[63] J. He, S. Li, X. Wang, S.-C. Cheung, G. Zhao, and J. Yang, "Neural-febi: Accurate function identification in ethereum virtual machine bytecode." *Journal of Systems and Software.*, vol. 199, p. 111627, 2023.
[64] K. Zhao, Z. Li, J. Li, H. Ye, X. Luo, and T. Chen, "Deepinfer: Deep type inference from smart contract bytecode," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 745–757.

Wenkai Li is currently pursuing the doctor's degree in the School of Cyberspace Security at Hainan University, China. Previously, he received a master's degree in the School of Cyberspace Security at Hainan University. His research lies in smart contract security and malicious behavior analysis, focusing on enhancing blockchain security through software and data analytics. He is also exploring the integration of artificial intelligence, such as graph neural networks and large language models.

Xiaoqi Li is an associate professor at Hainan University. Previously, he was a researcher at the Hong Kong Polytechnic University. He received his Ph.D. in Computer Science from Hong Kong Polytechnic University, MSc in Information Security from the Chinese Academy of Sciences, and BSc in Information Security from Central South University. His current research interests include Blockchain/Mobile/System Security and Privacy, Ethereum/Smart Contract, Software Engineering, and Static/Dynamic Program Analysis. He received best paper awards from INFOCOM'18, ISPEC'17, CCF'18, and an outstanding reviewer award from FGCS'17.

Yingjie Mao is currently pursuing a master's degree in the School of Cyberspace Security at Hainan University, China. Previously, he received a B.E. degree from the Southwest University of Science and Technology. His current research interests include Blockchain Security/Privacy and Large Language Model.

Yuqing Zhang is the Director of the Chinese National Computer Network Intrusion Prevention Center, Deputy Director of the Chinese National Engineering Laboratory of Computer Virus Prevention Technology, Vice Dean of the School of Computer and Control Engineering at the Chinese Academy of Sciences, and Professor at Hainan University. He received his Ph.D. from Xi'an University of Electronic Science and Technology. He has presented over 100 papers and 7 national/industry standards. His current research interests include Network Attacks and Prevention, Security Vulnerability Mining and Exploitation, IoT System Security, AI Security, Data Security, and Privacy Protection.