COBRA: Interaction-Aware Bytecode-Level Vulnerability Detector for Smart Contracts

Wenkai Li
Hainan University
Haikou, China
liwenkai871@gmail.com

Xiaoqi Li*
Hainan University
Haikou, China
csxqli@gmail.com

Zongwei Li
Hainan University
Haikou, China
lizw1017@gmail.com

Yuqing Zhang
University of Chinese Academy of Sciences
Beijing, China
zhangyq@nipc.org.cn

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 present SRIF (**S**ignatures **R**everse **I**nference from **F**unctions), 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.

CCS CONCEPTS

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

*Corresponding author

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695601

KEYWORDS

Ethereum, Bytecode, Smart contract, Function signature, Security

ACM Reference Format:
Wenkai Li, Xiaoqi Li, Zongwei Li, and Yuqing Zhang. 2024. COBRA: Interaction-Aware Bytecode-Level Vulnerability Detector for Smart Contracts. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695601

1 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 [24], while only around 1.19 million are open source to the public [39], accounting for approximately 1.79% of the total. With the advent of the smart contract layer, the Ethereum blockchain 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 caused an 3.6M ETH loss in the DAO event [7], and the error of arithmetic that led to an $80M loss in Compound Finance [46].

Symbolic execution (e.g., [38][40]), fuzz testing (e.g., [33][49]), and taint analysis (e.g., [43]) 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 [38]. Fuzzing tests attempt to expose flaws in the contract by using unreasonable inputs, but the presence of function interfaces may diminish their effectiveness [33]. 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 [43]. 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][30]), and other vulnerabilities (e.g., [21][45][44]). 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.

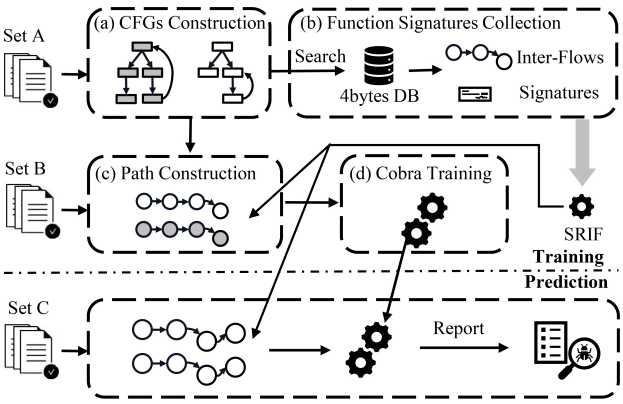

Figure 1: Overview of Framework.

In this paper, we propose COBRA, as demonstrated in Figure 1, a seq2seq structure with a novel encoder that *first* integrates semantic context and function interface for vulnerability detection. Our encoder in COBRA consists of the following four key components: **Part (a):** A semantic extraction process extracts semantic representations in the static single assignment (SSA) format, leveraging 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, which first collects public signatures for training, and then retrieves the undisclosed function interfaces from predictions. **Part (c):** A path sequence construction function that joint concatenates semantics and function interface features. Function interfaces and properties are mapped in parallel to form a function embedding. **Part (d):** A model training component that learns the vulnerability patterns from the joint 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 input parameters from the semantic context. Specifically, we infer the function properties by counting particular Opcodes and jointly mapping them as a function feature (§ 3.3, § 3.4).
- As far as we know, we are the *first* to present COBRA that integrates semantic context and ABI features into a novel encoder. COBRA generates an embedding of smart contracts to feed a decoder that classifies vulnerabilities (§ 3.5).
- 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, over 89% recall can be achieved with the inferred function representation (§ 4).
- We also open source some relevant datasets and codes at https://figshare.com/articles/dataset/22313074.

2 MOTIVATION & BACKGROUND

2.1 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

```
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 moeny 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

`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.

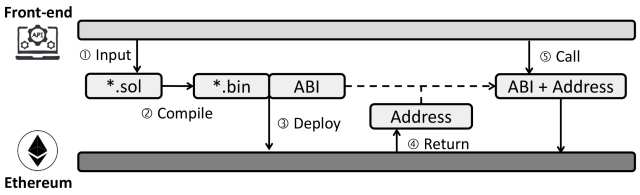

Figure 2: Deployment and Interaction of a Contract. The ①-④ represent the whole deployment process of a contract; the ⑤ and the return of ④ constitute an interactive process.

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 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.

2.2 Background

Smart Contract As a Turing-complete language, smart contract [6] is initially merged with blockchain in the 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[47]. Notably, it cannot be modified once the contract code has been released. After deployment, smart contracts can be interacted each other by invoking with specified function signatures.

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 [2]. EVM directly modifies the status information in the state database (StateDB) when a user account initiates a transfer request [10]. 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 [22]. 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 [48]. Moreover, all operations adhere to the gas mechanism [22]. 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 [48].

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 [3] to the function prototype string [9] and getting the first 4 bytes. Existing Function Signature libraries, such as the Ethereum Function Signature Database (EFSD) [1], 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.

Application Binary Interface ABI is an interpreter designed to facilitate communication between bytecode smart contracts on EVM [33]. 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.

2.3 Related Vulnerabilities

When smart contracts expand the programmability of blockchain systems, security problems also increase. The Decentralized Application Security Project (DASP) [26] is a project classifying smart contract vulnerabilities based on actual impact. In this paper, we will concentrate on five of these vulnerabilities in Table 1.

Table 1: 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 1 [27, 38]. The reentrancy can be discovered [7] 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 [8]. The Transaction Ordering Dependency (TOD) is also know as Time-Of-Check vs Time-Of-Use (TOCTOU) [32]. 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 [42].

3 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 the first step, the context information (i.e., Opcodes and SSA Opcodes) is extracted from the bytecode smart contracts. In the meantime, we crawl Etherscan [18] 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 represents 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

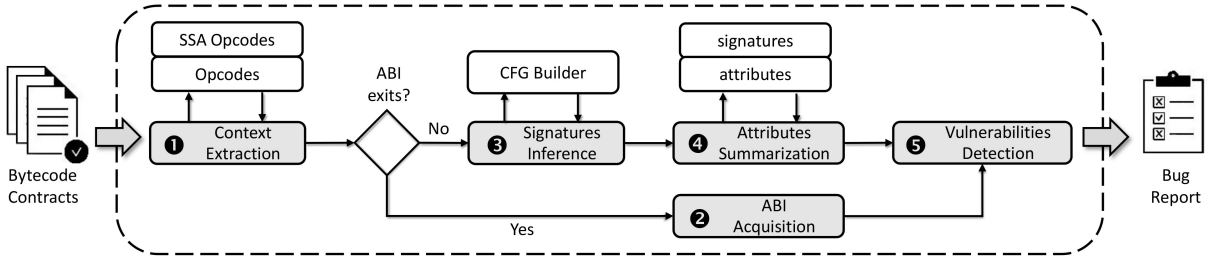

Figure 3: The Architecture of Our Work. The dashed box represents the primary steps of the detection process. The inputs are the bytecode smart contracts, and the output is a vulnerability report.

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.

Algorithm 1: Functions Context and Ids Acquisition

```
   input : A deployed bytecode smart contract bc
   output: two global map: functions context OpSeq, functions hashes Ids
1  BasicBlocks, eb ← CFG.countBasicBlocks(EVMAsm(bc));
2  pushValue, prePushValue ← None;
3  Function getFuncInfo(block, entry):
4      foreach instruction i of the block.ins do Ops ← i;
5      if entry then
6          if end of block compatible with JUMPI then
7              Assert length of block.ins > 2;
8              dest ← oprand of block.ins[-2];
9              OpSeq[dest] ←getFuncInfo(BasicBlocks[dest], false);
10             Ids[dest] ← None;
11         return Ops ;
12     for i in block.ins do
13         if i compatible with PUSHs then
14             prePushValue ← pushValue;
15             pushValue ← oprand of i;
16     if end of block compatible with JUMPI then
17         if prePushValue then
18             fnAddr, fnId ← pushValue, prePushValue;
19     else
20         fnAddr, fnId ← None;
21     if fnAddr compatible with BasicBlocks then
22         OpSeq[fnAddr] ← getFuncInfo(BasicBlocks[fnAddr], false);
23         Ids[fnAddr] ← fnId;
24         if end of block compatible with JUMPI then
25             dest ← ((endPc ep of block) + 1);
26             OpSeq[dest] ← getFuncInfo(BasicBlocks[dest], false);
27     return Ops ;
28 foreach entryBlock address eb of the BasicBlocks do
29     OpSeq[eb] ←getFuncInfo(BasicBlocks[eb], True);
```

3.1 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 [14], the Opcodes of contracts without compilation error can be gathered. 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 [13]. 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 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 three 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 previous 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. If the address in the blocks, the Opcodes and the hash in the block will be stored.

Furthermore, the precision of the EVM CFG builder [15] is crucial to the accuracy of the data we collected. For this reason, we keep an optimized EVM CFG recovery module in Elysium [20] to gather data for model training. The method in Elysium is proposed by [12] [4], which would extract more precise CFGs in this module.

3.2 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

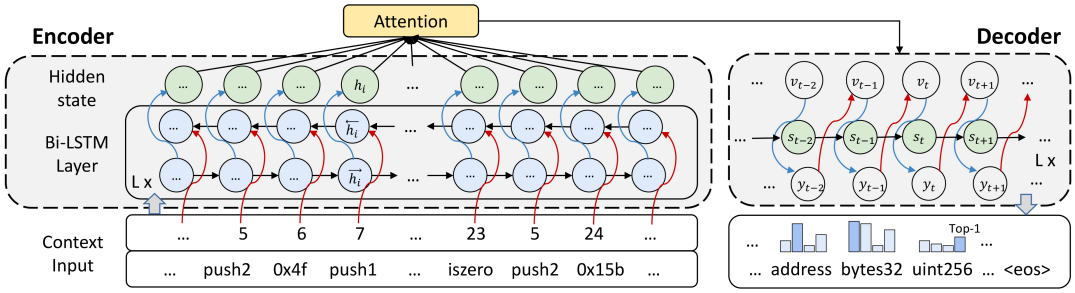

Figure 4: The Signatures Inference Model, SRIF. The dashed boxes represent the encoder and decoder, which are connected by an attention module.

have published ABI information. Therefore, for these bytecode contracts that do not disclose their ABI information, we propose an alternative approach in § 3.3 and § 3.4 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 convert it to TENSOR format and remove the function names.

3.3 Signature Inference

This section focuses primarily on deducing function parameters information from the function context obtained in the § 3.1. 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. Future work would then optimize the vulnerability detection model as an end-to-end framework, motivating our utilization of deep learning technology. However, we just infer these parameters tentatively in this paper.

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 § 3.1, the function id would be saved in 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 [9] 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 [5]. Therefore, to achieve a more adaptable process to get these input types in smart contracts, we utilize an encoder-decoder framework in Figure 4 to infer function parameters from the function context.

In § 3.1, we have gathered the Opcodes of basic block and function id for each function. Subsequently, we employed 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.

In the MLC task, 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. (1).

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

In Figure 4, 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 $|v|$. 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 $E \in \mathbb{R}^{k \times |v|}$ 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 [29], 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. (2), where h_i is the concatenation of the hidden states from both directions, indicating the ultimate representation of the i-th word.

$$h_i = [\overrightarrow{LSTM}(\overrightarrow{h_{i-1}}, c_i); \overleftarrow{LSTM}(\overleftarrow{h_{i-1}}, c_i)] \tag{2}$$

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. (3), eq. (4). 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 phrase.

$$w_{ti} = softmax(\boldsymbol{W_a^T} \tanh(\boldsymbol{U_a} s_t + \boldsymbol{O_a} h_i)) \tag{3}$$

$$v_t = \sum_{i=1}^{n} w_{ti} h_i \tag{4}$$

where s_t is the hidden state of decoder at t, which can be defined as following eq. (5),

$$s_t = LSTM(s_{t-1}, [g(y_{t-1}); v_{t-1}]) \tag{5}$$

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 [36]. 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 function 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. (6).

$$loss = -\alpha(1-p_t)^{\gamma} log(p_t) \tag{6}$$

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 [36] 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. (7), 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 category, and the average value represents the total loss.

$$loss' = \frac{\sum_{i=0}^{n} -\alpha_i(1-p_t)^{\gamma} log(p_t)}{n} \tag{7}$$

$$\alpha_i = \frac{\sum_{j=0}^{T} n_j}{n_i} \tag{8}$$

3.4 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 2, 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, we can conclude the Opcodes related to the *pure* property of functions [48].

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 2, 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.

Table 2: 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 keyword *payable* is mandatory in any function that involves asset transactions, where the type and amount of the asset depend on the message. After 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 according to the Ethereum Yellow Paper [48].

3.5 Vulnerability Detection

In this subsection, the main structure of our detection model will be presented. As Figure 5 shows, the raw bytecode smart contract will first be processed by context extraction (§ 3.1), signature inference (§ 3.3) and attributes summarization (§ 3.4). The sequence $X = \{x_1, x_2, ..., x_n\}$ contains n Opcodes. Using the rattle tool [13], 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 5, 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.

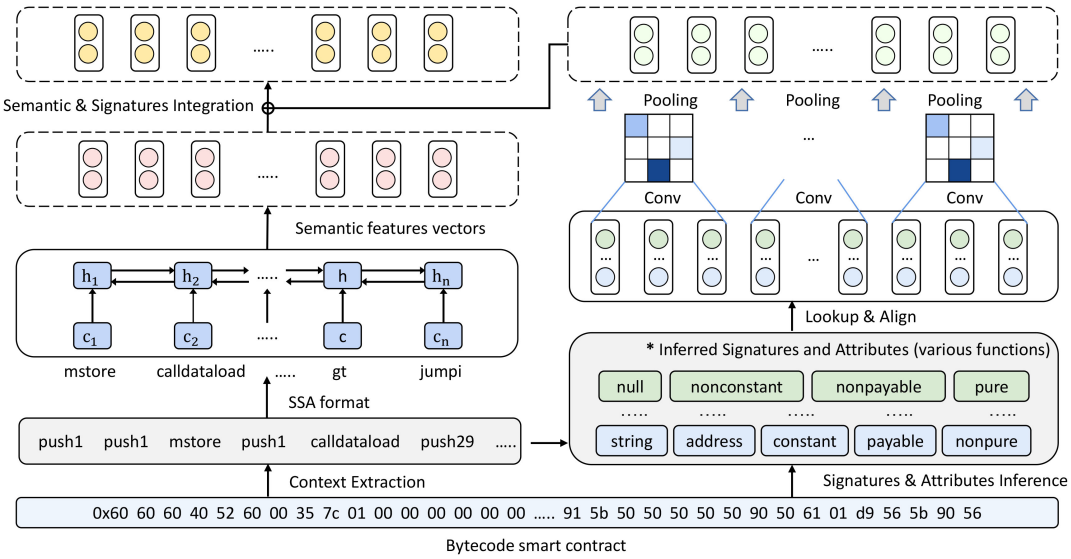

Figure 5: The Encoder of our Detection Model. The bottom layer is the hex 2-gram representation of the bytecode smart contract, which is converted into two types of data: the Opcode sequence and the function data. * indicates the part that can be replaced with raw ABI information. The box with yellow dots at the top of the diagram reflects the representation in the latent space.

If ABI cannot be obtained, we employ the method in § 3.3 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 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, then feed a CNN and average pooling layer. The factor that selecting CNN is to obtain local feature vectors for signatures 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 [16] in Figure 6 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. (9).

$$feature_l = N(Conv_2(\zeta(N(Conv_1(G))))) \tag{9}$$

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. (10), eq. (11).

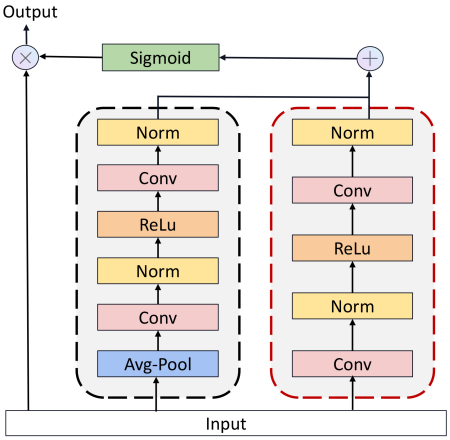

Figure 6: The MS-CAM in Our Structure. ⊗ means multiplication, ⊕ denotes addition. The black dashed box represents the global feature, and the red dashed box represents the local feature.

$$feature_g = GAP(feature_l) \tag{10}$$

$$output = G \otimes (\xi(feature_g \oplus feature_l)) \tag{11}$$

where the *GAP* represents a global average pooling layer, ⊗ means multiplication in the feature map, which is consistent with the MS-CAM. ⊕ 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 § 3.3. 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 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.

4 EXPERIMENTS

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

RQ1: **Is SRIF effective for the function signature inference?**
RQ2: **Is COBRA effective for the vulnerability detection?**
RQ3: **Is ABI or function signature in COBRA effective?**
RQ4: **Can COBRA detect new bugs in real-world environment?**

4.1 Experimental Setup

4.1.1 ***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 [51]. We use the SmartBugs [17] 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.

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 § 3.1, we gathered the function ids present in these contracts. These function signatures were then matched against the 4byte [1] database. Finally, 99,745 function signatures, along with the corresponding Opcodes, were collected.

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

4.1.3 ***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.

4.2 RQ1: 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 [25] 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 3, 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 3: 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 4 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 4: 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 RQ1. The SRIF achieves 94.76% F1-score on the test set, and it can recover more function signatures than Gigahorse.

4.3 RQ2 : 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 [17], and MANDO-GURU [41]. The recall value signifies the percentage of identified malicious classes among the actual malicious classes, evaluating when only inferred function signatures are available.

Result: As Table 5 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 6.

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 7 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 SRLF 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 this detection. After testing, Table 7 shows the recall can reach 89.46%.

Table 5: The Performance Comparison of Different Cells and Loss Functions.

Network Structures				
Cells	**Loss Function**	**F1-socre**	**Precision**	**Recall**
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 6: The Comparison Results of COBRA and other SOTA.

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

Table 7: 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%

Answer to RQ2. The COBRA achieves 93.45% F1-score on the test set, and 94.26% F1-score on the validation, which outperforms other methods.

4.4 RQ3: 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.

Result: Table 8 summarizes the case of distinct contract semantics and the addition of ABI separately. Note that all the data in Table 8 is done by the combination structure of LSTM with cross-entropy loss function. Table 8 provides insight into numerous conclusions. First, general results can be obtained using only context or SSA Opcode.

Table 8: 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%**

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 in the table indicate that using only global feature extraction can improve F1-score and recall, while MS-CAM can enhance precision.

Table 9: The Comparison of Different Architectures for Vulnerability Detection with Inferred Function Signatures.

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

As shown in Table 9, 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 9 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 in the contract.

Answer to RQ3. The ABI and function signatures can improve COBRA's detection performance to a certain degree.

4.5 RQ4: Can COBRA detect new bugs in real-world environment?

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 § 4.1.1, 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). An illustration of a potential contract vulnerability is presented in Listing 2, 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. 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 7, it is reasonable to conclude that `msg.caller` and `msg.value` are commonly present in the vulnerable contract.

```
contract PrivateBank {
  mapping (address=>uint) public balances;
  function CashOut(uint _am) {
    if(_am<=balances[msg.sender]) {
      if(msg.sender.call.value(_am)()){
        balances[msg.sender] -= _am; }
    }
  }
}
```

Listing 2: The Simplified Snippets of PrivateBank

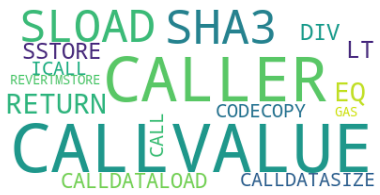

Figure 7: The Frequencies of SSA Opcodes in Reentrancy Vulnerabilities

Answer to RQ4. Utilizing COBRA, we find two previously undisclosed vulnerabilities, i.e., CVE-2023-36979 and CVE-2023-36980.

4.6 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 § 4.1.1), collected from real-world functions. Therefore, it is suitable for evaluation.

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.

5 RELATED WORK

Smart Contract Vulnerability Detection Many state-of-the-art works based machine learning have been presented for vulnerability detection [11][27][21][45][31][44][35][37][23], 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. [27] implement the fuzz function through the GRU module and combine it with symbolic execution techniques to achieve higher coverage. Gao et al. [21] 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. [45] 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. [31] 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. [44] proposed ESCORT, a multi-label detection tool that supports lightweight transfer learning. Different from these works, COBRA considers the advantages of function interface in the contract vulnerability detection task.

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 [25], Eveem [34]). 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 [9] 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 [19], Neural-FEBI [28], DeepInfer [50], and SRIF). Typically, OSD searches directly in EFSD [1] for function hashes to discover function signatures. Neural-FEBI identifies functions via a two-step process to generate a more precise CFG.

6 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 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.

7 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] 4byte. 2019. Ethereum Signature Database. https://www.4byte.directory
[2] Schmideg Adam, Szilágyi Péter, and Wilcke Jeffrey. 2013. Go Ethereum:Official Go implementation of the Ethereum protocol. https://geth.ethereum.org
[3] The Solidity Authors. 2019. Solidity documentation–contract abi specification. https://solidity.readthedocs.io/en/latest/abi-spec.html
[4] Andrea Benini, Mariano Ceccato, Filippo Contro, Marco Crosara, Mila Dalla Preda, and Michele Pasqua. 2023. Enhancing Ethereum Smart-Contracts Static Analysis by Computing a Precise Control-Flow Graph of Ethereum Bytecode. *Journal of Systems and Software*. (2023), 111653.
[5] Alex Beregszaszi, Kamil Śliwak, and et al. 2022. Solidity Releases. https://blog.soliditylang.org/category/releases
[6] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds.. In *Proc. of SP*. 161–178.
[7] Vitalik Buterin. 2016. CRITICAL UPDATE Re: DAO Vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability
[8] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering*. 48, 1 (2022), 327–345.
[9] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, and et al. 2022. SigRec: Automatic Recovery of Function Signatures in Smart Contracts. *IEEE Transactions on Software Engineering*. 48, 8 (2022), 3066–3086.
[10] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. 2022. WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts.. In *Proc. of ISSTA*. 703–715.
[11] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. 2018. Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology.. In *Proc. of WWW*. 1409–1418.
[12] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. 2021. EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode.. In *Proc. of ICPC*. 127–137.
[13] Crytic. 2018. rattle. https://github.com/crytic/rattle
[14] Crytic. 2020. Pyevmasm's documentation. https://pyevmasm.readthedocs.io/en/latest
[15] Crytic. 2023. evm_cfg_builder. https://github.com/crytic/evm_cfg_builder
[16] Yimian Dai, Fabian Gieseke, Stefan Oehmcke, Yiquan Wu, and Kobus Barnard. 2021. Attentional Feature Fusion.. In *Proc. of WACV*. 3559–3568.
[17] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts.. In *Proc. of ICSE*. 530–541.
[18] Etherscan. 2023. The Ethereum Blockchain Explorer. https://etherscan.io
[19] Ethervm. 2023. Online Solidity Decompiler. https://ethervm.io/decompile
[20] Christof Ferreira Torres, Hugo Jonker, and Radu State. 2022. Elysium: Context-Aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts.. In *Proc. of RAID*. 115–128.
[21] Zhipeng Gao. 2021. When Deep Learning Meets Smart Contracts.. In *Proc. of ASE*. 1400–1402.
[22] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: Detecting Gas-Related Vulnerabilities in Smart Contracts.. In *Proc. of ISSTA*. 728–739.
[23] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *Proc. of ICSE*. 1–12.
[24] Google. 2023. BigQuery - Ethereum Dataset. https://console.cloud.google.com/bigquery?p=bigquery-public-data&d=crypto_ethereum&page=dataset
[25] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts.. In *Proc. of ICSE*. 1176–1186.
[26] NCC Group. 2018. First Iteration of The Decentralized Application Security Project Top 10. https://dasp.co/index.html
[27] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts.. In *Proc. of CCS*. 531–548.
[28] Jiahao He, Shuangyin Li, Xinming Wang, Shing-Chi Cheung, Gansen Zhao, and Jinji Yang. 2023. Neural-FEBI: Accurate Function Identification in Ethereum Virtual Machine Bytecode. *Journal of Systems and Software*. 199 (2023), 111627.
[29] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*. 9, 8 (1997), 1735–1780.
[30] Huiwen Hu, Qianlan Bai, and Yuedong Xu. 2022. SCSGuard: Deep Scam Detection for Ethereum Smart Contracts.. In *Proc. of INFOCOM*. 1–6.
[31] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and et al. 2021. Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching. *IEEE Transactions on Information Forensics and Security*. 16 (2021), 2144–2156.
[32] Nikolay Ivanov, Qiben Yan, and Anurag Kompalli. 2023. TxT: Real-Time Transaction Encapsulation for Ethereum Smart Contracts. *IEEE Transactions on Information Forensics and Security*. 18 (2023), 1141–1155.
[33] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection.. In *Proc. of ASE*. 259–269.
[34] Tomasz Kolinko. 2018. Eveem/Panoramix–Showing Contract Sources. https://eveem.org
[35] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proc. of ISSTA*. 531–548.
[36] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection.. In *Proc. of ICCV*. 2980–2988.
[37] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding permission bugs in smart contracts with role mining. In *Proc. of ISSTA*. 716–727.
[38] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter.. In *Proc. of CCS*. 254–269.
[39] Ortner Martin and Eskandari Shayan. 2023. Smart Contract Sanctuary. https://github.com/tintinweb/smart-contract-sanctuary-ethereum
[40] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, and et al. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts.. In *Proc. of ASE*. 1186–1189.
[41] Hoang H. Nguyen, Nhat-Minh Nguyen, Hong-Phuc Doan, Zahra Ahmadi, Thanh-Nam Doan, and Lingxiao Jiang. 2022. MANDO-GURU: Vulnerability Detection for Smart Contract Source Code By Heterogeneous Graph Embeddings. In *Proc. of ESEC/FSE*. 1736–1740.

[42] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited.. In *Proc. of USENIX Security*. 1325–1341.
[43] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks.. In *Proc. of NDSS*. 1–15.
[44] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, and et al. 2023. Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning.. In *Proc. of NDSS*. 1–18.
[45] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution.. In *Proc. of USENIX Security*. 1361–1378.
[46] Blockonomist Staff. 2021. Compound Finance Mis-Rewarded around $80m Worth Comp to the Users. https://rekt.news/compound-rekt
[47] Samuel Steffen, Benjamin Bichsel, and Martin Vechev. 2022. Zapper: Smart Contracts with Data and Identity Privacy.. In *Proc. of CCS*. 2735–2749.
[48] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*. 151, 14 (2014), 1–32.
[49] Valentin Wüstholz and Maria Christakis. 2020. Targeted Greybox Fuzzing with Static Lookahead Analysis.. In *Proc. of ICSE*. 789–800.
[50] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. DeepInfer: Deep Type Inference from Smart Contract Bytecode. In *Proc. of ESEC/FSE*. 745–757.
[51] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-Ning Dai. 2020. XBlock-ETH: Extracting and Exploring Blockchain Data From Ethereum. *IEEE Open Journal of the Computer Society*. 1 (2020), 95–106.