DeepInfer: Deep Type Inference from Smart Contract Bytecode

Kunsong Zhao
The Hong Kong Polytechnic University
China
kunsong.zhao@connect.polyu.hk

Zihao Li
The Hong Kong Polytechnic University
China
cszhli@comp.polyu.edu.hk

Jianfeng Li
Xi'an Jiaotong University
China
jfli.xjtu@gmail.com

He Ye
KTH Royal Institute of Technology
Sweden
heye@kth.se

Xiapu Luo*
The Hong Kong Polytechnic University
China
csxluo@comp.polyu.edu.hk

Ting Chen*
University of Electronic Science and Technology of China
China
brokendragon@uestc.edu.cn

ABSTRACT

Smart contracts play an increasingly important role in Ethereum platform. It provides various functions implementing numerous services, whose bytecode runs on Ethereum Virtual Machine. To use services by invoking corresponding functions, the callers need to know the function signatures. Moreover, such signatures provide crucial information for many downstream applications, e.g., identifying smart contracts, fuzzing, detecting vulnerabilities, etc. However, it is challenging to infer function signatures from the bytecode due to a lack of type information. Existing work solving this problem depended heavily on limited databases or hard-coded heuristic patterns. However, these approaches are hard to be adapted to semantic differences in distinct languages and various compiler versions when developing smart contracts. In this paper, we propose a novel framework *DeepInfer* that first leverages deep learning techniques to automatically infer function signatures and returns. The novelties of *DeepInfer* are: 1) *DeepInfer* lifts the bytecode into the Intermediate Representation (IR) to preserve code semantics; 2) *DeepInfer* extracts the type-related knowledge (e.g., critical data flows, constant values, and control flow graphs) from the IR to recover function signatures and returns. We conduct experiments on Solidity and Vyper smart contracts and the results show that *DeepInfer* performs faster and more accurate than existing tools, while being immune to changes in different languages and various compiler versions.

CCS CONCEPTS

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

KEYWORDS

Smart Contract, Type Inference, Deep Learning

*Corresponding authors.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616343

ACM Reference Format:
Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. *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 '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616343

1 INTRODUCTION

Cryptocurrencies have shown a prevalent trend in both industry and academia in recent years. With the emergence of Ethereum [45], one of the largest decentralized platform, programmable cryptocurrency services enter a new era [11, 12, 29, 34, 61]. Smart contracts, as the key component of Ethereum, enable developers and users release cryptocurrencies, deploy applications, and utilize services on the Ethereum blockchain without the intervention of the trusted third parties [2, 33, 58, 69]. A smart contract is implemented by high-level languages (e.g., Solidity [52] and Vyper [57]), then compiled into the bytecode executed on Ethereum Virtual Machine (EVM), and finally deployed on Ethereum. The functions of a smart contract will be registered as the Application Binary Interface (ABI), making others easily invoke and implement their functionalities [9, 53].

Table 1: An overview of existing studies for recovering function signatures and returns in smart contracts.

Approach	Scalability		Accuracy		Techniques
	Languages	Compilers	Signature	Return	
EBD [43]	-	-	◐	○	FSD
Eveem [17]	○	○	◐	○	FSD+SA
Gigahorse [21]	○	○	◐	○	SA
SigRec [9]	◐	◐	◐	○	SA
DeepInfer	●	●	●	●	DL

● Full ◐ Partial ○ No support
FSD: Ethereum function signature database, SA: static analysis, DL: deep learning

The ABI consists of function signatures and returns. When invoking a function, users need to know the signatures because it defines the calling rules [53]. Function signatures comprise a function id and a list of parameter types in which function id is derived from the first 4 bytes of the Keccak-256 hash result of a function name and the corresponding list of parameter types in source code

[53]. Moreover, the function returns specify the format of return values of the function [53]. Identifying function signatures and returns is a crucial step to analyze the behavior of a function in smart contracts because one needs to first know how a function is called and what is returned before the evaluation. For example, previous studies adopted function signatures to recognize different types of smart contracts [5, 13, 16, 19, 24] and utilized function arguments to generate more mutant test cases to facilitate the fuzzing tools for detecting more vulnerabilities [9, 23, 27]. Moreover, checking returns of a function can avoid vulnerabilities, such as the unchecked call return value weakness [54].

Problem: function signatures and returns are black boxes for users. This is because nearly 99% of the deployed contracts are closed-source [18, 26]. The arguments of function signatures and returns are represented as the 256-bit words without the type information and debug information in the bytecode, which makes it hard to be recovered [1]. As shown in Table 1, some studies contribute to recovering function signatures in smart contracts. For example, EVM Bytecode Decompiler (EBD) [43] searches function signatures from their Ethereum Function Signature Database (FSD). Eveem [17] and Gigahorse [21] rely on the hard-coded heuristics to restore function signatures in which Eveem also incorporates the knowledge from the FSD. Chen et al. [9] proposed a static analysis-based tool SigRec that manually designed 31 heuristic patterns according to the access rules for different parameter types in the EVM and employed the symbolic execution technique to recover function signatures from the bytecode of a smart contract.

Unfortunately, these approaches still suffer from the following limitations.

Limitation 1 (L1): The existing methods for the function signature recovery heavily depend on either an incomplete database involving a fraction of functions in the wild [9] or restricted matching patterns designed by human experts. The limited database is hard to cover all the functions on Ethereum, where as the fixed matching patterns will be invalid when the smart contracts are developed by new programming languages with the evolution of the Ethereum ecosystems.

Limitation 2 (L2): Despite the existing tool designing a set of arguments access rules that has shown the ability to recovering types of arguments, its accuracy is still less than satisfactory, especially when the access patterns encounter even a slight change due to the compiler version upgrades.

Limitation 3 (L3): All the existing studies merely focus on recovering function signatures but the inference of the returns of a function is ignored. Since the aim of smart contracts is to execute transactions on Ethereum platform, both the signatures and returns of a contract function are indispensable components for analyzing the functionality and checking whether some desired operations are performed correctly [39, 42].

With deep learning techniques achieving promising results over traditional pattern-based methods in many software engineering tasks [49, 60, 67], one can leverage these data-driven techniques to learn implicit knowledge to infer function signatures and returns. However, it is challenging to design deep learning-based inference models for this purpose.

Challenge 1 (C1): The inference model needs to automatically learn how different types of arguments are operated in the EVM bytecode rather than ponderously depends on the function database or manually extracts a limited set of access patterns focusing on specific languages. This requires that the model actually grasps the differences in different types of arguments which are language-independent.

Challenge 2 (C2): As the compiler version upgrades, the patterns to access the arguments will be changed. This requires that the model is scalable, i.e., it needs to be free from the characteristics of various compiler versions.

Challenge 3 (C3): For function signature inference, we can force the model to understand the access patterns of different types of arguments and then use such knowledge to predict the signatures of other functions from a prepared type list. Unfortunately, it is infeasible for inferring some complex types (e.g., array) because we cannot determine how the number of nesting layers or dimensions there should be and restrict it to a limited set of types in advance. That is, it will suffer from an out-of-vocabulary (OOV) issue [22, 25]. This requires the model to decide which types need to be inferred from a finite set of types and which types need to dynamically determine the nesting depth or dimensions during the inference.

Challenge 4 (C4): Recovering returns is more difficult than inferring signatures, because the return values are stored in the memory and returned at the end of function execution [42, 52]. This requires the model to be able to understand the semantic of the whole function rather than a specific part. Despite there also exist some studies for function signature inference in other scenarios [3, 41, 47], none of them supports type inference from the bytecode of smart contracts because they target the source code.

In this paper, we present *DeepInfer*, a novel deep learning-based framework to automatically recover function signatures and returns from the EVM bytecode without any human intervention. To make the model deal with various languages, *DeepInfer* first lifts the bytecode compiled from different languages into the IR in which the language-specific and compiler-specific operations are stripped (**C1, C2**). Then, it conducts a definition/use analysis to extract critical information (e.g., data flows and constant values) that are highly relevant for the inference of signatures from the IR (§ 3.3). To make the model have the potential to recover various types of arguments, we design a two-stage inference framework in which the basic and complex types are handled individually (§ 2.2). Specifically, *DeepInfer* recognizes the basic and complex types according to the knowledge extracted and learned from the IR. For the basic types, *DeepInfer* recovers the actual type by selecting the one who has the maximum probability among the list of predefined types collected from the official documentations because of the limited number of types (§ 3.4). Since the complex types cannot be restricted in a finite set due to the above-mentioned OOV issue, *DeepInfer* employs a sequence generation model to dynamically generate the possible structure of such types according to the knowledge learnt from the critical information (§ 3.5) (**C3**). To recover returns, *DeepInfer* constructs the control flow graph (CFG) that captures the function semantic by means of the structured graph representation from the IR and employs the graph neural network to excavate the implicit semantic knowledge, aiming at promoting the model to understand the functionality (§ 3.6) (**C4**).

We conduct experiments on open-source smart contracts written by Solidity and Vyper. We collect unique functions from these

open-source smart contracts and use them to evaluate the accuracy of *DeepInfer*. The experimental results show that, overall, *DeepInfer* obtains the top-5 accuracy of 0.980 and 0.937 for signature inference in both Solidity and Vyper smart contracts, respectively. Compared with the baselines in Table 1, *DeepInfer* obtains an average accuracy improvement by 117.1% for recovering function signatures. Moreover, *DeepInfer* achieves the top-5 accuracy of 0.968 and 0.924 for recovering function returns, respectively. Further experiments show that *DeepInfer* performs more than 24 times faster than baseline methods. Meanwhile, *DeepInfer* is not affected by different languages and various compiler versions.

In summary, this paper makes the following major contributions:

- We develop *DeepInfer*, a novel framework that extracts and learns function access-related information from the lifted three-address code to recover function signatures. Moreover, *DeepInfer* is able to understand code semantics for recovering function returns.
- We conduct comprehensive experiments on real-world smart contracts written by Solidity and Vyper, and evaluate the accuracy of *DeepInfer*. The results show that *DeepInfer* obtains an average accuracy improvement by 117.1% compared to the existing tools for recovering function signatures. *DeepInfer* also has the unique potential to recover function returns and is immune to changes in different languages and various compiler versions.
- *DeepInfer* is the first work on recovering function signatures and returns from EVM bytecode based on deep learning inference, to the best of our knowledge. On the contrary to the state-of-the-art work [9], our process is done in a fully automatic manner without any human intervention.

2 BACKGROUND

2.1 Ethereum & Smart Contracts

Ethereum [45] is a second-generation blockchain-based platform that provides more flexible distributed computing abilities by incorporating smart contracts [28]. There are two types of accounts in Ethereum blockchain: External Owned Account (EOA) that can be treated as a wallet keeping assets (e.g., Ether) and smart contract which is created by either EOA or other smart contracts [14].

Smart contracts are executable programs that can run on the Ethereum blockchain, which implement various functionalities satisfying the requirements of end-users. It can be executed in a completely decentralized manner and does not depend on any trusted third parties [36]. Smart contracts are written by high-level programming languages (e.g., Solidity and Vyper) and compiled into the bytecode executing on the EVM.

2.2 Function Invocation in EVM

The EOA and COA accounts can call a smart contract function by sending an invocation message that consists of the address of the smart contract where the invoked function is located and a special filed *call data* which carries information about the function invoked and actual arguments [9]. The *call data* field appears as a sequence of bytes whose first 4 bytes refer to the function id and the rest is the arguments [53]. For example, to invoke the function shown in Listing 1, the *call data* field starts from the corresponding function id *0xea7cabdd* followed by two specific types of arguments: *_tokenId* (a one-dimensional dynamic array) and *owner* (an address).

There are two instructions read arguments from the *call data* field to EVM, including *CALLDATALOAD* and *CALLDATACOPY*. *CALLDATALOAD* first loads the top element in the stack of EVM and uses it as the offset to locate data in the *call data* field. It then loads 32 bytes data starting from the offset from the *call data* [45]. Eventually, the loaded data is put into the stack of EVM. Different from *CALLDATALOAD* that reads a fixed-size data into the stack, *CALLDATACOPY* loads a variable-size data from *call data* to the memory of EVM [45]. It first loads three elements from the top stack, including the memory location for storing the data, the location of *call data* for loading the data, and the length of data to be loaded. There are many supported types in Solidity and Vyper in their official documentations [52, 57], which can be treated as basic types and complex types according to whether one type is enumerable. The basic types include address, string, struct, bool, bytes, (u)int$_M$ (where $M \in \{8, 16, \ldots, 256\}$), and bytes$_N$ (where $N \in \{1, 2, \ldots, 32\}$). The complex types contain various arrays that are derived from the basic types except for struct, but can have variable nesting depth and sizes, such as one/multi-dimensional array with compile-time fixed or dynamic sizes and the nested array [52, 57]. Readers can see more access differences of these types in the literature [9, 53].

```
function checkAllOwner(uint256[] _tokenId, address owner) public
        view returns (bool) {
    ......
    for(uint i=0;i<_tokenId.length;i++){
        if(owner != zombieToOwner[_tokenId[i]]){
            return false;
        }
    }
    return true;
}
```

Listing 1: A function sample with two arguments and one return whose function id is *0xea7cabdd*.

2.3 Motivating Example

Listing 1 illustrates an example of a function deployed in Ethereum. The function consists of two arguments and one return value in which the first parameter is the dynamic array and the second one is the address. This function checks whether all tokens in the dynamic array *_tokenId* belong to a particular *owner* and returns true if so, false otherwise. There are different EVM instructions to load these two types of parameters. For the second parameter, the EVM first uses a *CALLDATALOAD* instruction whose operand is the start point of this parameter in the *call data* field. Then it loads a 32-byte data and uses an *AND* instruction for masking to a fixed-length (i.e., 32 bytes) [9, 52]. The masked result is the actual parameter value that can be used by following instructions. To recover the type of this parameter, we can make the model learn the operation instructions involved and constant values that are used for masking, and then select the type with the maximum probability from a fixed type list because the basic types are limited as mentioned above.

However, *the type inference for the first parameter is difficult because there is no dimension information of the array in the bytecode and thus is unknown during the compilation.* Assume the actual parameters of *_tokenId* in the *call data* is [0xa, 0xb, 0xc] and the arrangement of the data is displayed in Fig. 1. The first 4 bytes refer to the function id and the *offset* records the start point of this array. Moreover, *num* is the size of the actual elements in this array, followed by their individual real data values (i.e., 0xa, 0xb, and 0xc). Hence, to access an array element (e.g., 0xb), the EVM

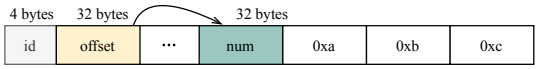


Figure 1: The data arrangement of a dynamic array uint256[].

uses two *CALLDATALOAD* instructions to load the items *offset* and *num* to identify the start point of this array, followed by another *CALLDATALOAD* instruction to obtain the actual value 0xb according to its address. To infer such types of arguments, *DeepInfer* designs a generative deep learning model which has the potential to automatically generate nested structures by learning critical data flows and involved constant values. This is because the forms of arrays are various, such as one/multi-dimensional static/dynamic arrays and nested arrays, whose nested depths and dimensions cannot be limited into a fixed set.

On the other hand, *recovering the type of the return value is difficult because they are stored in memory during EVM running.* Different from the type recovery of signatures that holds explicit operation instructions to load and access the function arguments, return values are stored into or loaded from the memory. This prevents the model from collecting obvious instruction operations. To infer the return type, we make the model to understand the functionality of the contract function and determine what is the type of the return value.

3 FRAMEWORK

3.1 Overview

Fig. 2 demonstrates the overall framework of *DeepInfer* that takes as input the EVM bytecode of smart contracts and finally outputs the signatures and returns of each function in it. Specifically, *DeepInfer* first lifts the bytecode into IR in which the complex stack operations are stripped but the useful instruction operations are reserved in the form of readily comprehensible three-address code. Then, *DeepInfer* recognizes all functions from the generated IR according to the function boundaries (§ 3.2). After that, *DeepInfer* extracts function access-related information (such as dataflow features and constant values) and constructs control flow graphs (CFG). This is because the function access-related information indicates how each parameter is loaded and used in the EVM for recovering function signatures whereas the CFG is related to understanding the functionality for recovering function returns (§ 3.3). Based on these information, *DeepInfer* trains the deep learning models to recover the basic types of arguments by building the classification model (§ 3.4), generates the complex types of arguments by training the sequence generation model (§ 3.5), and infers the function returns by understanding the whole functionality (§ 3.6).

3.2 Lifting & Function Recognition

For the input EVM bytecode of a smart contract, *DeepInfer* first uses Gigahorse [21] to parse the bytecode into the register-based IR (i.e., the three-address code) that consists of a clause <*opcode, operand*$_1$*, operand*$_2$*, ..., operand*$_n$*, result*> ($n > 0$), in which the *result* is the output of the instruction *opcode* with the operands (e.g., *operand*$_1$). If a variable appears in *result*, we consider it a variable definition operation. Moreover, if a variable appears in *operand*, we consider it is used. Note that each variable can be defined only once, i.e., it holds a unique value, but can be used multiple times [21]. We use the IR rather than smart contract bytecode in the following steps of *DeepInfer* because it simplifies stack operations to clauses, which makes it easy to analyze and extract the access patterns of function arguments. More specifically, *DeepInfer* first recognizes the JUMP instructions to find the boundaries of basic blocks and then conducts the context-sensitive and flow-sensitive analysis to process the register-based IR at the contract level. Besides, it speculates the entrance and exit of each basic block to recognize the function boundaries and generates the register-based IR at the function level. Each basic block is connected to one or more precursor and successor basic blocks except for the entrance and exit. The precursors refer to the ones that may be executed before the execution of a basic block. The successors means the ones that may be executed after the execution of a basic block. After generating the register-based IR for a smart contract, *DeepInfer* uses a regular expression to extract all the public functions by matching the function declarations. Next, we introduce how does *DeepInfer* extract critical features from such IR for model training.

3.3 Feature Extraction

After obtaining the three-address code of each function, *DeepInfer* extracts function access-related features from it. The key insights here lie in three aspects. First, in order to infer the types of arguments, *DeepInfer* intensively learns how each parameter is loaded and used by the instructions. This is a necessary step because different types of parameters are operated by distinct opcodes. For example, two *ISZERO* instructions will be used when the loaded parameter is a *bool* whereas only the instruction *BYTE* can be used to assess each byte data for a parameter with the type *byte32* [9]. Thus, *DeepInfer* proactively learns such access patterns automatically according to the instruction flows. Second, some types (e.g., int) are with different bit sizes which are represented as the mask value defined by the constant values in bytecode. For example, the type *uint8* requires 31 bits of zeros to mask the left side of the data whereas another type *uint248* simply needs 1 bit. Thus, *DeepInfer* is carefully designed to collect such constant values from the definition of them (§ 3.2). Third, unlike the arguments that utilize different instructions to access distinct types of parameters, the return values are stored in the memory and returned after the function is executed, which means that there are no specific instruction operations. Thus, *DeepInfer* needs to understand the whole functionality to infer the returns rather than learn the access patterns from the instruction flows.

3.3.1 Definition/Use Analysis. The aim of definition/use analysis is to trace a set of relations between variables in which one variable is used by others. As aforementioned (§ 2), a parameter is accessed by using either *CALLDATALOAD* or *CALLDATACOPY* instruction [9], thus, we do not need to analyze the definitions and uses of all variables. On the contrary, we fully focus on analyzing the variables containing the definitions or uses of each of the parameters in a function. We can achieve such variable analysis by using the definition/use analysis based on the parsed IR. Thanks to such IR that assigns a unique variable name to represent the execution result of each

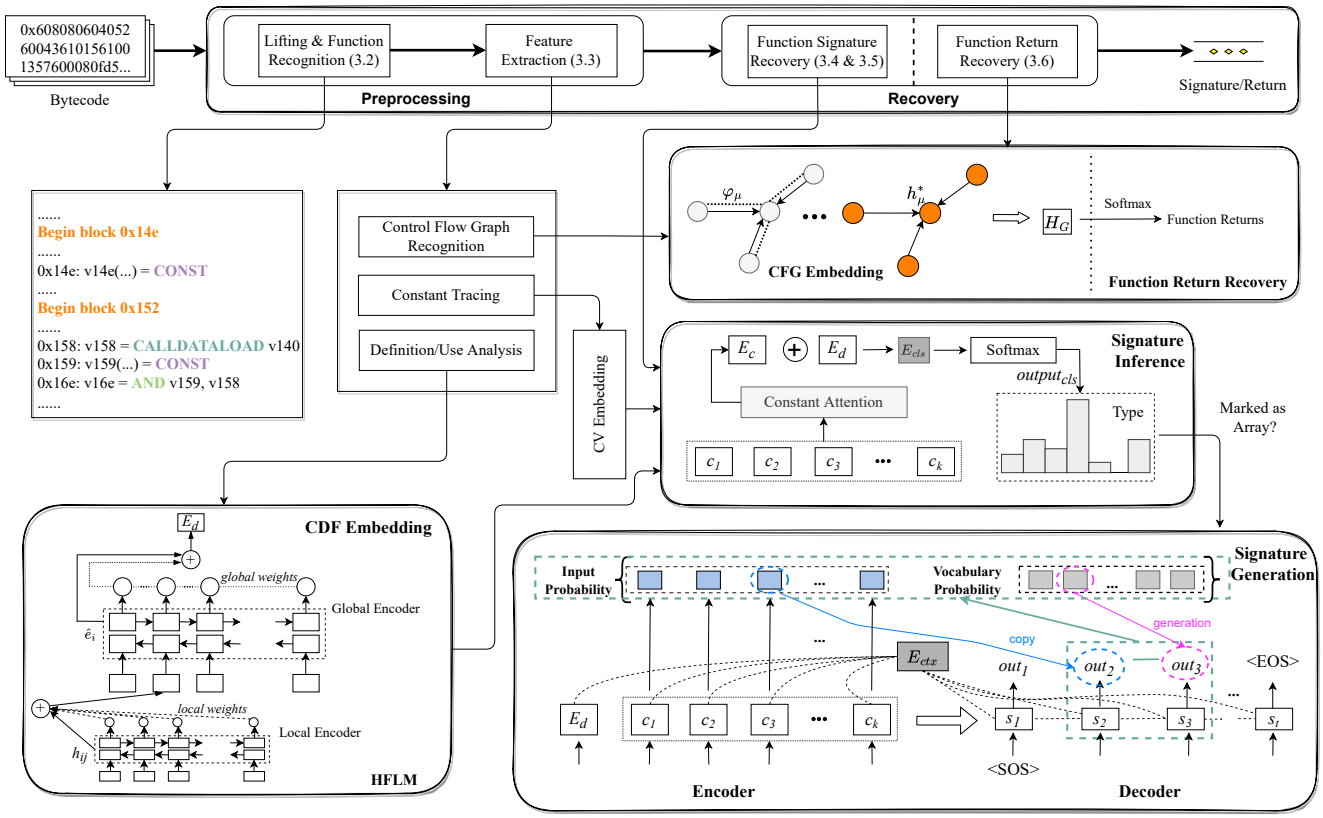


Figure 2: The overall framework of *DeepInfer*.

instruction, *DeepInfer* can search which variable definitions contain the two instructions and treat these variables as the start points, aiming at collecting the parameter access-related paths. Since the clauses in the IR will use the assigned variables as the operands, *DeepInfer* can recursively look for the clauses whose operands depend on the variable definitions of the two instructions. As a result, *DeepInfer* can obtain a set of instruction sequences where each sequence starts from either *CALLDATALOAD* or *CALLDATACOPY* instruction and we call each instruction sequence the Critical Data Flow (CDF) in the following.

3.3.2 Constant Tracing. The goal of constant tracing is to collect a set of constant values that are relevant to types of function arguments. The analysis of definitions and uses can generate a set of CDF starting from *CALLDATALOAD* or *CALLDATACOPY*. However, only using CDFs to build the signature inference model is inadequate because some types not only use type-related access instructions but depend on values to determine their sizes (e.g., *uint8*) or the dimensions (e.g., *address[2]*). Such values are also defined in the IR with the keyword *CONST*. One intuitive solution is to collect all the definitions that contain *CONST*. However, not all the values defined by *CONST* are related to the parameter access operations because some conditional jumps (e.g., *JUMPI*) or operations that consume constant values (e.g., *CALLDATALOAD*) can also involve such values. To solve this issue, we start with each CDF and collect the variables that appear in the operands. For each variable, we recursively backtrack to where it is defined until the initial clause whose *opcode* is the keyword *CONST* is found. Thus, we collect the corresponding operand in this clause as the constant value (CV). As a result, we can obtain a set of CVs for each CDF. By traversing all the CDFs, we can collect all the constant values associated with the arguments access operations.

3.3.3 Control Flow Graph Construction. The above features can be used to determine the access patterns and the corresponding sizes or dimensions while recovering function signatures. However, while recovering function returns, there is no related operation manifestation in the IR because all the return values are stored in the memory, i.e., there is a lack of explicit instructions for identifying specific operations about types that contained in returns. This requires finding a representation that not only contains the implicit characteristics of each function but can also be learned by the deep learning model. As an alternative, CFG abstractly represents the possible flows of all basic blocks in a function using the structured graph representation. It can reflect how each statement is executed during the program running [44, 70]. Besides, the structured presentation of a CFG also supports the use of the learnable model to dig for implicit features (i.e., its functionality). These benefits

encourage us to extract the CFG from the IR. Because the IR has been clearly divided into the basic blocks (§ 3.2), we can use regular expression matching to retrieve these basic blocks and treat each basic block as the node of a CFG. Besides, we generate the edges of a CFG by matching the precursors and successors of each basic block. As a result, we can obtain the structured CFG for each function in which nodes represent the basic blocks and edges refer to the jump relationships.

3.4 Function Signature Inference

One aim of *DeepInfer* is to recover function signatures by learning how different types of arguments are operated in the bytecode. There are many types of arguments used in the high-level languages (e.g., Solidity and Vyper) in which some basic types (e.g., uint, int, string, bool, etc) can be restricted into a fix-length set because they are enumerable but other complex types (i.e., array) cannot because they can hold unlimited dimensions and arbitrary sizes [53, 57]. Thus, we cannot simply treat the signature recovery as a traditional classification task. To solve this problem, we propose a two-stage framework that determines different types using distinct strategies. Specifically, we first mark types that are not enumerable with a general sign (i.e., Array) and all the types can be restricted into a limited type list. Thus, *DeepInfer* trains a classification model to determine the type of a parameter by selecting the one with the maximal prediction probability from this type list. Then, for the types that are marked as Array, *DeepInfer* trains another generative model to generate their actual types. In this subsection, we will introduce how to construct the classification model to learn the parameter access patterns and leave the detail of constructing the generative model in the next subsection (§ 3.5).

3.4.1 CDF Embedding. After extracting a set of CDFs and CVs from the IR of a function, *DeepInfer* trains a model to learn access patterns of different types of arguments. For the collected CDFs, we retain the opcodes for model training because opcodes preserve the real operations during the code execution. Since the obtained opcodes are not numeric and cannot be used as the input of type recovery model, *DeepInfer* first trains a Word2Vec model based on these opcodes to produce the initial embedding vectors. Specifically, it treats the opcodes extracted from a CDF as a sentence and employs the Continuous Bag of Word (CBOW) technique [40] that predicts a word (i.e., opcode) using its contextual words to generate an embedding mapping matrix $\mathbb{W}$.

Since there are multiple CDFs in each function and all of them together form the access patterns of arguments, inspired by previous work [65], *DeepInfer* designs the Hierarchical Flow Learning Mechanism (HFLM) to learn the access patterns among the opcodes in these CDFs. Specifically, assume the set of opcode sequences extracted from the CDFs is $\Theta = \{\theta_1, \theta_2, \ldots, \theta_m\}$ where $\theta_i = \{\theta_{i1}, \theta_{i2}, \ldots, \theta_{in}\}$ is the i-th sequence. m and n represent the number of CDFs and the length of the CDF, respectively. For each opcode, *DeepInfer* first initializes it with the embedding matrix, i.e., $e_{ij} = \mathbb{W}\theta_{ij}$. Then, for each θ_i, it uses the bi-directional LSTM [72] to incorporate the contextual operations from two directions and produce the hidden embedding vectors:

$$h_{ij} = [\overrightarrow{LSTM}(e_{ij}) \mid \overleftarrow{LSTM}(e_{ij})] \tag{1}$$

where h_{ij} is the hidden embedding vector of the j-th opcode in θ_i and $[\cdot|\cdot]$ refers to the concatenate operation.

Local Encoder. After obtaining the vector of each opcode, *DeepInder* next generates the embedding vector of a CDF. Since there exist instructions that merely use the variable defined by previous instructions as the target address and the type-related instructions will occur after such instructions, simply summing or averaging operation over the opcodes will introduce some irrelevant noise information. To make the model pay more attention to type-related operations, we employ the attention mechanism to calculate local weights from these opcodes and aggregate them into the representation of the CDF:

$$e_i = \sum_q h_{ij}[exp\left(f_l\left(h_{ij}\right)\right) / \sum_q exp\left(f_l\left(h_{ij}\right)\right)] \tag{2}$$

where e_i is the embedding vector of i-th CDF; $exp()$ refers to the exponential function; f_l represents the linear layer followed by the ReLU activation function [20]. We call the above operations the local encoder as shown in Fig. 2, because they focus on a single CDF in a function.

Global Encoder. According to the above operations, we can obtain the embedding vector for each CDF. Then, we introduce how to generate the overall embedding vector at function level because our aim is to recover the signatures for a function. There are multiple CDFs can be extracted from a function and they are accessed in the same order as the arguments that appear in the function declaration. Thus, we treat the CDFs as a sequence and use their embedding vectors (i.e., e_i) as the model input. We also use the bi-directional LSTM to update the embedding vectors of each CDF:

$$\hat{e}_i = [\overrightarrow{LSTM}(e_i) \mid \overleftarrow{LSTM}(e_i)] \tag{3}$$

where $\hat{e}_i$ is the hidden embedding vector of i-th CDF generated by the model.

Similarly, not all CDFs play the same importance in recovering the type of one parameter because the EVM will use one or more different instructions to access distinct types of parameters. Hence, *DeepInfer* uses the attention mechanism to calculate global weights from the embedding vectors of all the CDFs and incorporate them into an overall representation, forcing the model to concentrate on the parts of interested CDFs while recovering different types of parameters:

$$E_d = \sum_p \hat{e}_i[exp\left(f_g\left(\hat{e}_i\right)\right) / \sum_p exp\left(f_g\left(\hat{e}_i\right)\right)] \tag{4}$$

where E_d refers to the overall representation vectors of CDFs; f_g represents the linear layer followed by the ReLU activation function.

By using the above equations (1) - (4), *DeepInfer* can output the overall CDF embedding vector for each function. Next, we will introduce how to deal with constant values.

3.4.2 CV Embedding. As for constant values, since there is no order relations between them, *DeepInfer* builds a lookup table W_c in which each constant is initialized randomly and updated during the model training. Assume a set of constant values collected in the IR of each function is $\Phi = \{\phi_1, \phi_2, \ldots, \phi_k\}$ in which k is the number of unique constant values. *DeepInfer* embeds each constant into its initial embedding vector by inquiring about the lookup table, i.e., $c_k = W_c\ \phi_k$. We will explain how these constant values are aggregated

into the CDF embedding vectors to enhance the model inference ability when building the classification model and the generative model in the following.

3.4.3 Classification Model. As mentioned above, we have restricted all the types into a limited list. Thus, we can naturally regard the signature inference as a classical multi-classification task. Specifically, given the embedding vectors of CDFs E_d and CVs $c_1, c_2, \ldots, c_k$, we expect the model to learn not only the access operations from CDFs but also the possible sizes from CVs. Thus, *DeepInfer* first uses a linear function f_c to map each constant embedding c_i into the hidden vector $\hat{c_i}$ and then employs the attention mechanism to learn to pay more attention to the constant value that are relevant to a specific type, i.e., constant attention:

$$E_c = \sum_k [exp(\hat{c_i}) / \sum_k exp(\hat{c_i})] c_i \tag{5}$$

where E_c is the overall representation of the CVs in each function.

To learn the operation patterns and the constant information simultaneously, *DeepInfer* concatenates these two embedding vectors to form the representation vector for each function:

$$E_{cls} = [E_d | E_c] \tag{6}$$

According to the representation E_{cls}, *DeepInfer* can infer the most possible type for the arguments by selecting the one that has the maximal prediction probability over the type list:

$$output_{cls} = argmax\,(W_1 E_{cls} + b_1) \tag{7}$$

where $output_{cls}$ is the predicted type of a parameter; W_1 and b_1 represent the trainable weight matrix and bias, respectively.

Recall that we mark all the types of arrays as the general signal Array. If *DeepInfer* predicts a parameter as the type Array, its actual type still needs to be further determined. We will introduce the details in the next subsection.

3.5 Function Signature Generation

Different from enumerable types, types that are marked as Array can include infinite nested structure and arbitrary size, which makes it impossible to maintain a limited type list holding all the possible situations. To solve this problem, *DeepInfer* designs a sequence generation model to dynamically generate the possible array type by learning the type-related information extracted from the IR. Specifically, *DeepInfer* follows the architecture of sequence to sequence learning which consists of an encoder and a decoder to achieve this goal as shown in Fig. 2.

Encoder. Different from the classical sequence learning encoder that takes the tokenized sequence as inputs and makes each token learn the contextual token information, *DeepInfer* directly treats the embedding vector of CDFs (i.e., E_d) and all the embedding vectors of CVs (i.e., c_k) as the input, but doesn't require them to recognize the context because there is no order relationships between the inputs. *DeepInfer* utilizes such an input form because we expect it can not only predict the basic types of an array (e.g., uint, bool, address, etc) according to implicit knowledge from CDFs but also determine the dimensions according to the CVs. Thus, *DeepInfer* adopts the input embedding vectors $\mathcal{X} = \{c_0, c_1, c_2, \ldots, c_k\}$ ($c_0 = E_d$) to guide the type generation in the decoding procedure.

Decoder. *DeepInfer* uses vanilla Recurrent Neural Network (RNN) as the basic architecture of the decoder which reads an input token and combines it with embedding vectors from the encoder to generate the target token. Assume the vocabulary of basic types is $\mathcal{V} = \{v_1, \ldots, v_r\}$ where r represents the vocabulary length, *DeepInfer* can predict the basic type of an array from this vocabulary according to the input information. However, the nested depth or dimension size of an array cannot be foreseen because such values need to be determined dynamically according to the inputs. That is, it suffers from the out-of-vocabulary (OOV) issue [22, 25]. To deal with this situation, *DeepInfer* introduces the copy mechanism [22, 51] that can dynamically copy or generate the most possible token as the decoder output.

Given the encoder output $\{c_0, c_1, c_2, \ldots, c_k\}$ and the current (i.e. step t) decoder input hidden status s_t, *DeepInfer* employs the attention mechanism to calculate the current context vector:

$$E_{ctx} = \sum_{i=0}^{k} \alpha_i c_i \tag{8}$$

$$\alpha_i = s_t W_g c_i / \sum_i s_t W_g c_i \tag{9}$$

where α_i is the weight score and W_g is the trainable weight matrix.

Then, *DeepInfer* predicts the probability of the output token at current step by either selecting the most possible token from the vocabulary $\mathcal{V}$ or copying the token from the input sequence $\mathcal{X}$:

$$out_t = p_g gene_t + (1 - p_g) copy_t \tag{10}$$

where the out_t is the final probability distribution at step t; p_g refers to the probability a token needs to be generated; $gene_t$ and $copy_t$ means the probability distributions over the vocabulary $\mathcal{V}$ or the input sequence $\mathcal{X}$, respectively:

$$p_g = \sigma\left(W_{ctx} E_{ctx} + W_s s_t + W_p s_{t-1} + b_g\right) \tag{11}$$

$$gene_t = softmax\,(W_V [s_t | E_{ctx}] + b_V) \tag{12}$$

$$copy_t = \sum_{i:x_i=y} att_i^t \tag{13}$$

where σ is an activation function; W_{ctx}, W_s, W_p, b_g, W_V, and b_V are the trainable parameters; y is a token and $x_i \in \mathcal{X}$; att_i^t refers to the attentive probability distribution over $\mathcal{X}$ at step t.

The decoder starts with the token <SOS> and outputs all possible tokens step by step until a special token <EOS> appeared following the process in previous work [22, 51]. To make the model better learn the correct output token sequence, *DeepInfer* introduces the teacher forcing technique [68] during the model training. This technique employs the scheduled sampling [8] that makes the input distribution between training and generation as similar as possible to force the output tokens being subject to truly array types.

3.6 Function Return Recovery

As described above, arguments access has explicit instruction operations in the EVM bytecode whereas the returns of a function is stored into the memory. Thus, the model needs to understand the functionality for inferring function returns. CFG reflects how each statement is executed during the program running [44][70] and we expect the model to understand return-related knowledge from the CFG. Graph Neural Network (GNN), due to its powerful

structural learning ability, has been widely used in many code comprehension related tasks [4][66]. Because of the graph structure representation of the CFG, it is natural that using the GNN model to learn its implicit information. In this work, *DeepInfer* adopts the Graph Attention Network (GAT) to learn the knowledge from the CFG, because not all the execution paths are associated with the return values and GAT can automatically give more focus on the execution paths related to the function returns.

Concretely, assume the node set of a CFG is $\Omega = \{\omega_1, \omega_2, \ldots, \omega_\mu\}$ where μ represents the number of basic blocks. We extract the instruction sequence from each basic block and use their opcodes to train another CBOW model for producing a new embedding matrix $\mathbb{W}'$ similar to § 3.4.1. We don't use the previous mapping matrix $\mathbb{W}$ because our aim here is to make the embedding of each opcode capture its contextual operations at basic block level rather than the signature-related CDFs. For each basic block, *DeepInfer* uses $\mathbb{W}'$ to initialize the opcodes included in it and averages them to produce the node embedding vectors.

Next, *DeepInfer* employs the GAT to update the embedding vectors of each node by incorporating its neighborhood nodes with different weights:

$$h_\mu^* = \sum_{v \in \mathcal{N}_\mu} \varphi_\mu W_r h_v^* \tag{14}$$

where h_μ^* is the updated embedding vector of the μ-th node and $\mathcal{N}_\mu$ means the neighborhood nodes of the μ-th node. W_r is trainable parameters. φ_μ is the weight score across $\mathcal{N}_\mu$:

$$\varphi_\mu = \frac{\exp\left(\text{LeakyReLU}\left(W_r'[h_\mu^*|h_v^*]\right)\right)}{\sum_{v \in \mathcal{N}_\mu} \exp\left(\text{LeakyReLU}\left(W_r'[h_\mu^*|h_v^*]\right)\right)} \tag{15}$$

where LeakyReLU [37] is the activation function and W_r' is a trainable parameter.

After the node embedding vectors are updated by multi-layer GAT, *DeepInfer* aggregates all the node embedding vectors to produce the representation vectors H_G for a CFG using the similar operation as Eq. (2) and Eq. (4). Finally, *DeepInfer* uses a softmax layer to output the probability distribution.

4 EVALUATION

4.1 Experimental Setup

4.1.1 Dataset. To evaluate the accuracy of *DeepInfer*, we follow the previous work [10] to instrument an Ethereum node to obtain the runtime bytecode from deployed smart contracts. As a result, we collect 47,598,631 bytecode for Solidity smart contracts and 75,627 bytecode for Vyper smart contracts. We merely keep the open-source smart contracts because the ground-truth can be obtained from their source code. We remove the duplicated smart contracts and use the Etherscan API [6] to collect their ground-truth. According to the statistic, 99.9% functions with equal or less than nine arguments and returns. Following the processing principle in previous work [15], we focus on the functions with no more than nine arguments or returns. As a result, we obtain 292,064 unique functions for Solidity contracts and 5,003 unique functions for Vyper contracts, respectively. For each language, we randomly select 80% of the collected data as the training set and the remainder is treated as the test set.

4.1.2 Implementation. We implement *DeepInfer* in about 2,100 lines of code using Python 3. The IR is parsed from the bytecode using the *Gigahorse* tool [21]. We employ *multiprocessing* package with 64 processes to parallelize the data processing pipeline. Since there are multiple arguments or returns for each function, we first train a model to predict the number of arguments or returns and separately train the models for arguments or returns on each position. We iteratively train the models to relieve the data deficiency issue and enhance the knowledge learned by the model [62–64, 71]. For the model construction, we adopt Pytorch framework [48] to implement our HFLM component in which two bi-directional LSTM layers are used. We embed each input token into a 100-dimensional embedding vector and the size of hidden layer is set as 200. To optimize the model parameters, we select the Adam algorithm [30] as the optimizer to update the gradient. During the generation process, we adopt beam search technique [55] with a beam width of 10 to produce the output sequence. To understand the functionality from the CFG, we use two layers of the GAT to learn and update node embedding vectors.

Table 2: Evaluation Results of *DeepInfer*

Compiler	Metric	Signature		Return	
		Number	Type	Number	Type
Solidity	Top-1	0.983	0.835	0.871	0.749
	Top-3	0.995	0.944	0.971	0.889
	Top-5	0.998	0.966	0.988	0.943
Vyper	Top-1	0.721	0.634	0.800	0.535
	Top-3	0.958	0.840	0.990	0.782
	Top-5	0.994	0.897	0.998	0.841

4.2 RQ1: How is the accuracy of *DeepInfer*?

4.2.1 Motivation. The aim of *DeepInfer* is to recover the function signatures and returns by learning the access patterns or understanding the functionality from EVM bytecode, respectively. This question is designed to explore to what extent *DeepInfer* can recover the function signatures or returns.

4.2.2 Approach. To answer this question, we train different models to deal with distinct tasks. For function signature inference, *DeepInfer* trains a model to predict the number of arguments and then other models are trained to determine the type at each position over a limited type list (§ 3.4). For the types that are marked as Array, since the infinite nesting depth and arbitrary size of them, *DeepInfer* trains generative models to recover the actual type (§ 3.5). For function return inference, *DeepInfer* trains classification models that can understand the functionality to predict the number and types of returns (§ 3.6). We evaluate the accuracy of *DeepInfer* with all the collected functions in open-source Solidity and Vyper contracts, respectively. We report the average top-k accuracy of *DeepInfer*, which is the ratio of correct inference occurred in the most probable k ($k \in \{1, 3, 5\}$) candidates to the total number of unique ground truths.

4.2.3 Results. Table 2 illustrates the accuracy of *DeepInfer* for recovering the number and types of signatures and returns, respectively. When considering the top-1 suggestion, *DeepInfer* achieves

the accuracy of 0.983, 0.835, 0.871, and 0.749 for the number and type inference of arguments and returns in Solidity, respectively. It obtains the top-1 accuracy of 0.721, 0.634, 0.800, and 0.535 in Vyper smart contracts, respectively. When taking the top-5 suggestion into account, the accuracy goes up to 0.998, 0.966, 0.988 and 0.943 for Solidity, and 0.994, 0.897, 0.998 and 0.841 for Vyper, respectively. It shows the average accuracy improvements in recovering signatures and returns by 13.3% and 38.6% in Solidity and Vyper, respectively.

We manually investigate the incorrect inference and summarize the causes of failures as follows.

```
function register(bytes _domain, address _address) external {
    ......
    addresses[_domain] = _address;
}
```

Listing 2: An inaccurate recovery in case 1

Case 1: *DeepInfer* will confuse the types *bytes* and *string* when there is no operation on the parameter itself in smart contracts. As shown in Listing 2, despite *DeepInfer* correctly predict the second parameter as the type *address* but it inaccurately predicts the first parameter as *string* instead of *bytes*. This is because both *bytes* and *string* are loaded by the same instruction *CALLDATACOPY*. They differ only in that the former can be accessed by the *BYTE* instruction but the latter cannot. However, in this case, the first parameter is used as the index without any manipulation for itself, i.e., the *BYTE* instruction is absent in the corresponding IR.

```
function getContractVersionCount(bytes32 _name) external ... {
    ......
    return addressStorageHistory[_name].length;
}
```

Listing 3: An inaccurate recovery in case 2

Case 2: *DeepInfer* will confuse the types *bytes32* and *uint256*. When loading the byte sequence, it will be masked by zeros on the low-order side. Instead loading the unsigned integer will result in the masking by zeros on the high-order side. However, when loading the types *bytes32* or *uint256*, they have already reached the maximum length (i.e., 32 bytes) so no masking operation is needed, i.e., they have the same loading instructions. The differences between them are that the former can be accessed by *BYTE* instruction, whereas the latter can be used for arithmetic operations. As a result, as shown in Listing 3, *DeepInfer* incorrectly predict the *bytes32* as the *uint256*, because there are no byte access or arithmetic operations and this parameter is only used as the index.

```
function verify(address[2] tokens, uint256[8] args) external{
    address depositToken = tokens[0];
    address issueToken = tokens[1];
    uint256 totalIssueAmount = args[0];
    uint256 interestRate = args[1];
    uint256 maturity = args[2];
    uint256 issueFee = args[3];
    uint256 minIssueRatio = args[4];
    ......
}
```

Listing 4: An inaccurate recovery in case 3

Case 3: There are some missing constant values due to the compilation optimization of smart contracts, which misleads the prediction of *DeepInfer*. As shown in Listing 4, the arguments of the function are one-dimensional static array whose sizes needed to be inferred by incorporating the information from constant values. However, since the optimization operation is activated during the compilation, the constant values related to the size are missing. As a result, *DeepInfer* inaccurately infers the size of the array.

```
function getRiskAndValue(bytes32 _result) public returns (uint80
    , uint128) {
    bytes memory riskb = sliceFromBytes32(_result, 0, 16);
    bytes memory valueb = sliceFromBytes32(_result, 16, 32);
    return (uint80(toUint128(riskb)), toUint128(valueb));
}
```

Listing 5: An inaccurate recovery in case 4

Case 4: *DeepInfer* fails to infer some rare types. As shown in Listing 5, very few functions adopt the type *uint80* as the return type. This causes the model can simply learn extremely limited knowledge about this type during the model training process. As a result, *DeepInfer* outputs an incorrect inference.

Answer to RQ1: The top-5 accuracy of *DeepInfer* is 0.974 for Solidity and 0.933 for Vyper.

4.3 RQ2: How does *DeepInfer* perform compared with existing tools?

4.3.1 Motivation. Since there are some existing tools that decompile bytecode into human-readable pseudocode or directly recover function signatures from the bytecode, this question is designed to explore whether *DeepInfer* performs better than existing techniques.

4.3.2 Approach. We compared *DeepInfer* with three state-of-the-art decompilers (including EBD [43], Eveem [17], and Gigahorse [21]) and one symbolic execution-based static analysis technique SigRec [9] in terms of signature recovery. All the four baseline approaches support the bytecode of smart contracts as input. If one method can correctly infer the types of signatures or returns at a position, we treat this as a successful prediction. We report the average accuracy of these baselines in Solidity and Vyper smart contracts, respectively.

4.3.3 Results. Table 3 presents the average results for *DeepInfer* and baseline methods. From this table, we can find that the average top-1 accuracy of *DeepInfer* achieves average improvements by 85.9% and 148.3% compared with the baselines while recovering function signatures in Solidity and Vyper, respectively. Despite *DeepInfer* obtaining nearly the same top-1 accuracy as SigRec when predicting the types of arguments in Solidity, it achieves an improvement by 8.4% when considering the top-5 suggested types. Besides, the top-1 accuracy of *DeepInfer* achieves an improvement by 13.8% compared with SigRec when dealing with Vyper Smart contracts. This is because SigRec depends on the static heuristic rules designed by human experts, which limits its ability to expand to cover newly developed and compiled contracts involving new access patterns. Instead, *DeepInfer* can automatically learn the access patterns of arguments from the bytecode without any human intervention. We can see from this table that none of the baselines support recovering function returns. Instead, *DeepInfer* achieves the top-5 accuracy of 0.968 and 0.924 while recovering function returns in Solidity and Vyper, respectively.

Table 3: Average Results for *DeepInfer* and Baselines

Compiler	Approach	EBD	Eveem	Gigahorse	SigRec	*DeepInfer*		
						Top-1	Top-3	Top-5
Solidity	Signature	0.459	0.521	0.327	0.904	0.900	0.967	0.980
	Return	-	-	-	-	0.818	0.936	0.968
Vyper	Signature	0.208	0.271	0.216	0.589	0.670	0.888	0.937
	Return	-	-	-	-	0.674	0.891	0.924

Answer to RQ2: The top-1 accuracy of *DeepInfer* for recovering function signatures achieves average improvements by 85.9% and 148.3% across the baselines in Solidity and Vyper, respectively. In addition, only *DeepInfer* can recover function returns with the top-5 accuracy by 0.968 for Solidity and 0.924 for Vyper.

4.4 RQ3: How efficient is *DeepInfer*?

4.4.1 Motivation. As the static analysis techniques for function signature recovery (e.g., SigRec) are time-expensive due to the program simulation execution and path exploration [7], *DeepInfer* depends on well-trained deep learning models to directly predict function signatures. This question is designed to evaluate how efficient is *DeepInfer* while recovering function signatures and returns.

4.4.2 Approach. To explore what is the performance overhead of *DeepInfer*, we execute and compare it with other baselines. We exclude the approach EBD because it depends on a limited FSD and searches the function signatures from this database if exists, which just consumes a negligible amount of time. For *DeepInfer*, since the training process can be done offline, we compare its predictive time cost. We report the average time consumption for recovering signatures and returns of each function in seconds.

Table 4: Average Time Consumption

Approach	$DeepInfer_S$	SigRec	Gigahorse	Eveem	Speedup	$DeepInfer_R$
Time(s)	0.08	0.40	0.57	5.09	24.25x	0.003

$DeepInfer_S$ and $DeepInfer_R$ refer to the inference for signatures and returns, respectively.

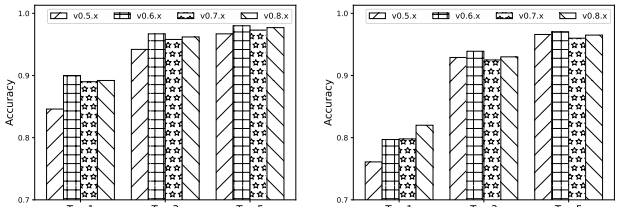
4.4.3 Results. Table 4 elaborates the average time consumption of *DeepInfer* and other three baselines for recovering function signatures. *DeepInfer* only spends 0.08 seconds for recovering signatures of each function, which is on average over 24 times faster than other baselines. Besides, *DeepInfer* spends 0.003 seconds for recovering function returns on average. This is because different from the static analysis-based techniques that run on central processing units, *DeepInfer* adopts deep learning as the infrastructure which naturally supports the operation acceleration of the graphics processing units. In addition, *DeepInfer* can be fully trained offline and then used to predict online.

Answer to RQ3: *DeepInfer* is over 24 times faster than the baselines on average.

4.5 RQ4: How does the compiler version affect *DeepInfer*?

4.5.1 Motivation. The compiler versions are frequently upgraded which will introduce the new characteristics and operations to avoid potential vulnerabilities [56]. This question is designed to explore how the changes of compiler versions impact *DeepInfer*.

4.5.2 Approach. To evaluate the impact of various compiler versions on *DeepInfer*, we category all the open-source Solidity smart contracts according to their main versions. We train the models on the lower versions and test whether *DeepInfer* still works on the high version that is not visible during the model training. For example, we train the models on the smart contracts with the versions 0.4.x to 0.7.x and assess its accuracy on the high version 0.8.x. We select the smart contracts whose version is large than 0.4.x because the counterpart is too little to support the model training (less than 1%). We report the average accuracy of *DeepInfer* on different compiler versions.



(a) Signature (b) Return

Figure 3: Evaluation results of *DeepInfer* under various compiler versions.

4.5.3 Results. Fig. 3 presents the average accuracy of *DeepInfer* under different compiler versions. We can find that *DeepInfer* achieves nearly the same performance for recovering function signatures and returns across various compiler versions, respectively. Despite the state-of-the-art tool SigRec has the ability to recover the function signatures as shown in § 4.3, it will be completely disabled when the compiler version is larger than 0.8.0. This is because the design of SigRec relies on the access patterns of compiler versions. Instead, *DeepInfer* is built upon the IR which strips the compiler-related operations. Such design makes *DeepInfer* insensitive to the changes of compiler versions. It also demonstrates that *DeepInfer* actually learns the implicit knowledge that is relevant to function signatures and returns.

Answer to RQ4: *DeepInfer* is immune to the changes in compiler versions.

5 DISCUSSION

In this section, we will discuss the limitations of *DeepInfer* and potential threats to validity encountered.

First, *DeepInfer* cannot recover the function signatures if the IR is not correctly parsed during the phase of lifting. *DeepInfer* adopts the state-of-the-art lifting tool Gigahorse [21] to convert the bytecode into the register-based IR. If Gigahorse crashes, the IR cannot be properly lifted, *DeepInfer* will not be able to extract the valid access patterns, resulting in a failed recovery. Similar situation will also occur while recovering returns. But we found that such situation is rare (nearly 0.2%), which implies that *DeepInfer* will hardly be affected. We will try to extend such tool to support valid

lifting operation, especially for the Vyper smart contracts in the future.

Second, the accuracy of inferring some rare types is not as high as that of inferring other types. This is because *DeepInfer* designs a deep learning-based architecture which is inherently limited by the training data itself due to the data-hungry characteristics of deep learning techniques [31, 59]. One solution is to employ some data-enhancement techniques that preserve the semantics of the bytecode and we leave this as the immediate future work.

On the other hand, there are some possible threats to validity during our experiments. The threats to internal validity lie in the tuning of hyper-parameters in the models. To relieve this validity, we fine-tune the batch size from 16 to 128 and the learning rate from 1e-5 to 1e-3 to make the model be fully trained. In addition, we set the hidden size as 200 dimensions and adopt some default settings (i.e., stacking two LSTM or GAT layers) for experiments. Other parameter combinations may also improve the accuracy of *DeepInfer* and we leave this exploration as the future work.

6 RELATED WORK

6.1 Signature Inference in Smart Contracts

Abi-guesser [50] was developed to infer the types of ABI-encoded data. Chen et al. [9] was the first to develop a static analysis-based tool called SigRec to recover function signatures in smart contracts. Specifically, SigRec first disassembled the bytecode and then proposed type-aware symbolic execution that explored how a parameter was manipulated in EVM instructions. It designed different patterns for different types according to the specific type-related operations in EVM and symbolically executed EVM instructions to recover parameter types.

However, abi-guesser can only deal with very simple data formats, which limits the ability in real-world smart contracts. In addition, since the accessing rules used by SigRec heavily rely on the expert knowledge, it is hard to be extended. Thus, we propose the first deep learning based method that can automatically learn the accessing rules or understand functionalities to recover function signatures and returns from the bytecode of smart contracts without any manual efforts.

6.2 Deep Learning for Signature Inference

Although very little work focused on function signature inference in smart contract, this topic been explored much more in other scenarios with the help of deep learning techniques. Chua et al. [15] was the first to introduce deep learning into function type recovery from C/C++ binaries. They regarded instructions as a sequence and employed the RNN model to learn the semantics to recover types. However, some important information related to function signatures was stripped off during the compilation process. To deal with this limitation, Lin et al. [35] proposed ReSIL that injected compiler-optimization-related domain information into the instructions to assist the inference of function signatures. Pei et al. [46] pretrained a model to learn the operational semantics from asssembly instructions with generative state modeling and then used the model to infer the types for C/C++ binaries. Lehmann et al. [32] tried to recover types from WebAssembly binaries. They defined grammars of high-level type languages and employed the neural machine translation architecture to recover these high-level types based on the DWARF debugging information.

In addition to the above studies recovering signatures from binaries, there also exist work focused on signature recovery from source code. Malik et al. [38] developed a LSTM-based model called NL2Type that utilized the code comments, function names and parameter names from the source code to predict the function types in JavaScript. Allamanis et al. [3] developed a graph neural network based method equipped with a novel triplet loss function which can learn the syntactic and semantic from source code to predict types in Python functions. Mir et al. [41] first parsed Python source code into ASTs and then extracted identifers, contextual information and visible type hints. They incorporated code semantics learnt by using a RNN model from code contexts with type hints and employed kNN algorithm to infer types. Peng et al. [47] proposed HiTyper that combined deep learning with static analysis for type prediction in Python. They first constructed a type dependency graph (TDG) from source code and then conducted forward static type inference along the TDG according to type dependencies. For some variables that would impact the types of many others, HiTyper trained a similarity-based deep learning model to recommend possible types.

Different from the above studies, we focus on recovering function signatures and returns from the bytecode of smart contracts without any human intervention. Since the smart contracts are designed to perform especial actions on the blockchain (e.g., transactions), the above-mentioned tools cannot capture some domain-specific operations, such as *CALLDATALOAD* and *CALLDATACOPY*. Thus, it is necessary to develop a new tool for the bytecode of smart contracts. To the best of our knowledge, we are the first to design such an automatic deep learning model for this purpose.

7 CONCLUSION

We present *DeepInfer*, a novel deep learning-based framework to automatically recover function signatures and returns from the bytecode of Solidity and Vyper smart contracts without any human intervention. The experimental results demonstrate that *DeepInfer* is more accurate and efficient than existing tools under different languages and distinct compiler versions. In the future, we plan to conduct a deep analysis on each component of the *DeepInfer* framework, and explore other alternative model components (such as Transformers) and parameter combinations to improve the model performance. In addition, we will extend our tool to support smart contracts running on other blockchains.

8 DATA AVAILABILITY

Our experimental materials are available at https://github.com/sepine/DeepInfer.

ACKNOWLEDGEMENT

We sincerely thank the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong RGC Projects (No. PolyU15219319, PolyU15222320, PolyU15224121) and National Natural Science Foundation (No. 62202405).

REFERENCES

[1] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2019. SAFEVM: a safety verifier for Ethereum smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 386–389.

[2] Maher Alharby and Aad Van Moorsel. 2017. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372* (2017).

[3] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st Acm SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 91–105.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[5] Monika di Angelo and Gernot Salzer. 2020. Characterizing types of smart contracts in the ethereum landscape. In *International Conference on Financial Cryptography and Data Security*. 389–404.

[6] Etherscan API. 2019. Etherscan documentation. https://docs.etherscan.io/.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[8] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* 28 (2015).

[9] Ting Chen, Zihao Li, Xiapu Luo, Xiaofeng Wang, Ting Wang, Zheyuan He, Kezhao Fang, Yufei Zhang, Hang Zhu, Hongwei Li, et al. 2021. Sigrec: Automatic recovery of function signatures in smart contracts. *IEEE Transactions on Software Engineering (TSE)* 48, 8 (2021), 3066–3086.

[10] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, et al. 2019. Dataether: Data exploration framework for ethereum. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1369–1380.

[11] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ting Wang, Teng Hu, Xiuzhuo Xiao, Dong Wang, Jin Huang, and Xiaosong Zhang. 2019. A large-scale empirical study on control flow identification of smart contracts. In *Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11.

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

[13] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1503–1520.

[14] Weimin Chen, Xinran Li, Yuting Sui, Ningyu He, Haoyu Wang, Lei Wu, and Xiapu Luo. 2021. Sadponzi: Detecting and characterizing ponzi schemes in ethereum smart contracts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 2 (2021), 1–30.

[15] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security)*. 99–116.

[16] Monika Di Angelo and Gernot Slazer. 2020. Wallet contracts on Ethereum. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 1–2.

[17] Eveem. 2019. Eveem. https://eveem.org.

[18] Michael Fröwis and Rainer Böhme. 2017. In code we trust? In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. 357–372.

[19] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. 2019. Detecting token systems on ethereum. In *International Conference on Financial Cryptography and Data Security*. 93–112.

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*. 315–323.

[21] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1176–1186.

[22] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393* (2016).

[23] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 531–548.

[24] Zheyuan He, Shuwei Song, Yang Bai, Xiapu Luo, Ting Chen, Wensheng Zhang, Peng He, Hongwei Li, Xiaodong Lin, and Xiaosong Zhang. 2022. TokenAware: Accurate and Efficient Bookkeeping Recognition for Token Smart Contracts. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2022).

[25] Ziniu Hu, Ting Chen, Kai-Wei Chang, and Yizhou Sun. 2019. Few-Shot Representation Learning for Out-Of-Vocabulary Words. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[26] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. 2021. Hunting vulnerable smart contracts via graph embedding based bytecode matching. *IEEE Transactions on Information Forensics and Security (TIFS)* 16 (2021), 2144–2156.

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

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

[29] Jerome Kehrli. 2016. Blockchain 2.0-from bitcoin transactions to smart contract applications. *Niceideas, November. Available at: https://www. niceideas. ch/roller2/badtrash/entry/blockchain-2-0-frombitcoin* (2016).

[30] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[31] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. 2020. The open images dataset v4. *International Journal of Computer Vision* 128, 7 (2020), 1956–1981.

[32] Daniel Lehmann and Michael Pradel. 2022. Finding the dwarf: recovering precise types from WebAssembly binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 410–425.

[33] Shipeng Li, Jingwei Li, Yuxing Tang, Xiapu Luo, Zheyuan He, Zihao Li, Xi Cheng, Yang Bai, Ting Chen, Yuzhe Tang, et al. 2023. BlockExplorer: Exploring Blockchain Big Data via Parallel Processing. *IEEE Trans. Comput.* (2023).

[34] Zihao Li, Jianfeng Li, Zheyuan He, Xiapu Luo, Ting Wang, Xiaoze Ni, Wenwu Yang, Chen Xi, and Ting Chen. 2023. Demystifying DeFi MEV Activities in Flashbots Bundle. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[35] Yan Lin, Debin Gao, and David Lo. 2022. ReSIL: Revivifying Function Signature Inference using Deep Learning with Domain-Specific Knowledge. In *Proceedings of the 12th ACM Conference on Data and Application Security and Privacy*. 107–118.

[36] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 630–641.

[37] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, Vol. 30. 3.

[38] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 304–315.

[39] Jan Midtgaard and Thomas P Jensen. 2012. Control-flow analysis of function calls and returns by abstract interpretation. *Information and Computation* 211 (2012), 49–76.

[40] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[41] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 2241–2252.

[42] Benoît Montagu and Thomas Jensen. 2021. Trace-based control-flow analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 482–496.

[43] MrLuit. 2019. EVM bytecode decompiler. https://github.com/MrLuit/evm.

[44] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly detection using program control flow graph mining from execution logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 215–224.

[45] Ethereum Yellow Paper. 2022. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf.

[46] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 690–702.

[47] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. 2019–2030.

[48] PyTorch. 2022. PyTorch Documentation. https://pytorch.org/docs/stable/index.html.
[49] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems (NeurIPS)* 33 (2020), 20601–20611.
[50] Samczsun. 2022. Abi-guesser. https://github.com/samczsun/abi-guesser.
[51] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).
[52] Solidity. 2022. Solidity Documentation v0.8.17. https://docs.soliditylang.org/en/v0.8.17.
[53] Contract ABI Specification. 2022. Solidity Documentation v0.8.17. https://docs.soliditylang.org/en/v0.8.17/abi-spec.html.
[54] SWC-104. 2020. Unchecked call return value. https://swcregistry.io/docs/SWC-104.
[55] Transformers. 2022. Hugging Face Transformers. https://github.com/huggingface/transformers.
[56] Solidity v0.8.0. 2020. Solidity v0.8.0 Breaking Changes. https://docs.soliditylang.org/en/v0.8.17/080-breaking-changes.html.
[57] Vyper. 2022. Vyper Documentation. https://vyper.readthedocs.io/en/stable/.
[58] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: A practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1410–1422.
[59] Xiaozhi Wang, Ziqi Wang, Xu Han, Wangyi Jiang, Rong Han, Zhiyuan Liu, Juanzi Li, Peng Li, Yankai Lin, and Jie Zhou. 2020. MAVEN: A massive general domain event detection dataset. *arXiv preprint arXiv:2004.13590* (2020).
[60] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 3034–3040.
[61] Pengcheng Xia, Haoyu Wang, Bingyu Gao, Weihang Su, Zhou Yu, Xiapu Luo, Chao Zhang, Xusheng Xiao, and Guoai Xu. 2021. Trade or trick? detecting and characterizing scam tokens on uniswap decentralized exchange. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)* 5, 3 (2021), 1–26.
[62] Zhiwen Xie, Runjie Zhu, Kunsong Zhao, Jin Liu, Guangyou Zhou, and Jimmy Xiangji Huang. 2021. Dual gated graph attention networks with dynamic iterative training for cross-lingual entity alignment. *ACM Transactions on Information Systems (TOIS)* 40, 3 (2021), 1–30.
[63] Zhiwen Xie, Runjie Zhu, Kunsong Zhao, Jin Liu, Guangyou Zhou, and Xiangji Huang. 2020. A contextual alignment enhanced cross graph attention network for cross-lingual entity alignment. In *Proceedings of the 28th International Conference on Computational Linguistics (COLING)*. 5918–5928.
[64] Kun Xu, Linfeng Song, Yansong Feng, Yan Song, and Dong Yu. 2020. Coordinated reasoning for cross-lingual knowledge graph alignment. In *Proceedings of the AAAI conference on Artificial Intelligence (AAAI)*, Vol. 34. 9354–9361.
[65] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT)*. 1480–1489.
[66] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems (NeurIPS)* 33 (2020), 3872–3883.
[67] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1385–1397.
[68] Wen Zhang, Yang Feng, Fandong Meng, Di You, and Qun Liu. 2019. Bridging the gap between training and inference for neural machine translation. *arXiv preprint arXiv:1906.02448* (2019).
[69] Peilin Zheng, Zibin Zheng, and Xiapu Luo. 2022. Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 740–751.
[70] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* 32 (2019).
[71] Hao Zhu, Ruobing Xie, Zhiyuan Liu, and Maosong Sun. 2017. Iterative entity alignment via knowledge embeddings. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
[72] Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. 2015. Long short-term memory over recursive structures. In *International Conference on Machine Learning (ICML)*. 1604–1612.

Received 2023-02-02; accepted 2023-07-27