FIRE: Smart Contract Bytecode Function Identification via Graph-Refined Hybrid Feature Encoding

Yu Sun, Lingfeng Bao†, Xiaohu Yang
The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
22321296@zju.edu.cn, lingfengbao@zju.edu.cn, yangxh@zju.edu.cn

ABSTRACT

The growing popularity of smart contracts has spurred an increasing demand for efficient analysis of their bytecode. Reverse engineering plays a critical role in understanding and auditing smart contracts, with function identification being a key aspect. However, existing function identification techniques often struggle with scalability, accuracy, and adaptability across different contract versions. This paper presents FIRE (Smart Contract Bytecode Function Identification via Graph-Refined Hybrid Encoding), a novel approach to function identification in Ethereum smart contract bytecode. By leveraging hybrid encoding of basic blocks and incorporating a graph neural network (GNN) based on control flow graph (CFG), our method improves the effectiveness of function identification. The approach demonstrates strong generalization across contract versions and significantly reduces runtime. We evaluate FIRE on multiple datasets and show its superior performance compared to existing techniques, highlighting its potential for efficient smart contract bytecode analysis.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; • **Computing methodologies → Machine learning**.

KEYWORDS

Smart Contract, Reverse Engineering, Function Identification, Machine Learning, Graph Neural Network

ACM Reference Format:
Yu Sun, Lingfeng Bao†, Xiaohu Yang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, 22321296@zju.edu.cn, lingfengbao@zju.edu.cn, yangxh@zju.edu.cn . 2025. FIRE: Smart Contract Bytecode Function Identification via Graph-Refined Hybrid Feature Encoding. In *the 16th International Conference on Internetware (Internetware 2025), June 20–22, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3755881.3755883

1 INTRODUCTION

The rise of blockchain technology [42] has revolutionized various industries, with decentralized applications (dApps) built on platforms such as Ethereum [38] gaining substantial attention. Central to these applications are smart contracts [43], which are self-executing agreements with the terms encoded directly into the code. While these smart contracts offer powerful capabilities for automating processes and facilitating trustless interactions, they also pose distinct challenges related to security [37], optimization [19], and analysis [21]. Although Ethereum smart contracts are typically written in high-level languages like Solidity, they are ultimately deployed as low-level bytecode. This compiled form, characterized by inherent complexity and compiler optimizations, poses significant challenges for comprehension and analysis—even for experienced developers. Furthermore, a substantial portion of deployed contracts lack publicly available source code, making decompilation an essential process to regenerate analyzable code for critical security audits and vulnerability assessments. Consequently, efficiently decompiling Ethereum bytecode has become a vital aspect of smart contract analysis, particularly in ensuring the accuracy and security of contracts prior to deployment.

During decompilation, reconstructing the original high-level logic from the bytecode requires precise function identification, as functions encapsulate the core behaviors and interactions within a contract. The absence of explicit metadata and the compression optimization employed in bytecode make function identification a non-trivial task. Addressing this challenge is essential for ensuring the reliability of decompiled code, enabling binary analysis, security assessments [32] and vulnerability detection [30].

Traditional smart contract decompilers [5, 14, 15, 23] often rely on heuristics or pattern matching based on common bytecode sequences. For example, Gigahorse [14] discovers function entries by heuristically setting the call site pattern. These approaches, while useful, are inherently limited by their reliance on pre-defined rules and can be inaccurate when faced with more complex or optimized bytecode as compiler optimizations and unconventional control flow patterns often break the underlying assumptions of static pattern matching. Furthermore, as Ethereum contracts evolve and new patterns emerge, these heuristic-based methods often struggle with generalization across different contract versions, making them less effective in dynamic real-world scenarios.

Recent advancements [17, 24, 41, 44] in machine learning, particularly deep learning, have shown potential in automating and improving bytecode analysis. These approaches have demonstrated the capability to learn from large datasets, offering significant benefits in terms of effectiveness. However, the state-of-the-art neural models for smart contract function identification, neural-FEBI [17], only considers the bytecode as a sequence model and uses conditional random fields [22] (CRF) to predict function entries. They do not fully utilize the information of bytecode, resulting in some performance bottlenecks, especially in optimized bytecode data. Since

†Corresponding author, also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

Internetware 2025, June 20–22, 2025, Trondheim, Norway

ACM ISBN 979-8-4007-1926-4/25/06
https://doi.org/10.1145/3755881.3755883

bytecode is not completely organized for continuous execution, the state transitions learned in CRF may be overly dependent on training data. When the new compiler adopts additional optimizations that cause the organization of bytecode to change, neural-FEBI is difficult to generalize and adapt to the new version of bytecode data.

In this paper, we propose FIRE (Function Identification via Graph-Refined Hybrid Feature Encoding), a novel machine learning approach to function identification in Ethereum smart contracts. Our approach addresses the limitations of traditional decompilers and existing machine learning models , as discussed earlier, by integrating a graph-based refined hybrid feature encoding modeling strategy.

Specifically, we model the feature encoding of the basic blocks obtained by bytecode segmentation. For each basic block, we perform statistical feature analysis to obtain its local instruction frequency encoding, and fuse it with the temporal features output by the language model [20]. In addition, through static analysis, we construct a control flow graph [1] with basic blocks as nodes. The hybrid encoding is input into the graph neural network [39] and refined iteratively through the control flow graph structure. The refined feature encoding can be classified to determine whether the basic block is a function entry.

The combination of hybrid encoding and GNN refinement allows FIRE to leverage both local and global context in the bytecode, resulting in a more precise identification of function entry blocks. This advantage contributes to the robustness of our approach, enabling models trained on low-version data to be successfully transferred for predictions on higher-version data. Strong robustness means that FIRE does not have to face the difficulty of labeling new version data. Besides, the improvement in function entry identification accuracy also enhances the effectiveness of the function boundary identification task. Additionally, our approach achieves a significant boost in runtime performance compared to state-of-the-art models.

In summary, the major contributions of this work are as follows:

- We propose FIRE, a novel function identification approach for Ethereum smart contract bytecode, which integrates hybrid feature encoding and graph neural network refinement to enhance function entry identification accuracy.
- We demonstrate the effectiveness of our approach in addressing generalization challenges by ensuring high accuracy across different contract versions, outperforming existing heuristic-based methods and machine learning models.
- Our approach improves the accuracy of function entry identification, making function boundary identification more accurate and achieving state-of-the-art results.
- Our approach significantly improves runtime performance, providing a more efficient solution for large-scale contract analysis.

2 MOTIVATION

In this section, we provide a motivational example to introduce our approach.

Figure 1 shows a partial subgraph of the control flow graph of a smart contract bytecode [1]. There are four basic blocks (split by *JUMP*, *JUMPI* and *JUMPDEST* instructions) in the graph, which will be referred to by their starting instruction program counter address subscripts in the following text, namely $\mathcal{B}_{547}$, $\mathcal{B}_{2004}$, $\mathcal{B}_{2025}$, and $\mathcal{B}_{2091}$. Among them, $\mathcal{B}_{2004}$ with a yellow background color is a function entry block. From an inspection of the previous state-of-the-art approach and consideration of the current example, the following observations can be made.

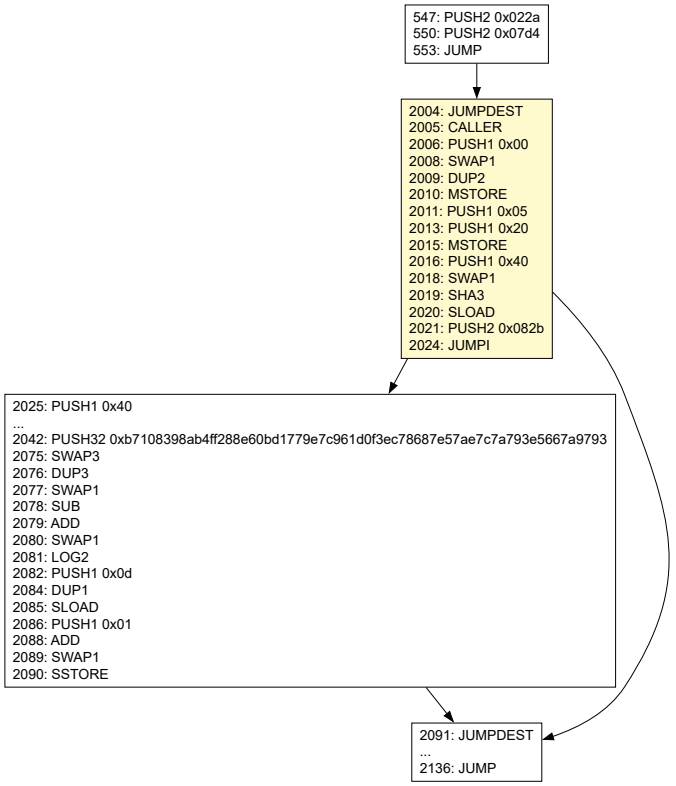


Figure 1: An example of a subgraph of a control flow graph generated from a smart contract bytecode.

Observation 1. Modeling bytecodes with the graph model is more appropriate than the sequence model. The state-of-the-art approach, neural-FEBI [17], treats the bytecode as a natural language sequence and performs part-of-speech tagging [33] by combining LSTM[18] with CRF [22] to achieve function entry classification. However, the relationships between blocks are often non-linear and can involve jumps, and branching that make sequence-based models less effective. Different versions of compilers have different degrees of optimization when organizing instructions, which may cause the dependencies captured by the sequence model to fail to generalize. As shown in Figure 1, a control flow graph (CFG) better captures these relationships by representing blocks of instructions as nodes and edges as control flow between them. The graph structure naturally reflects the dependencies and control flow of the contract, providing richer context than what a sequence model can represent, and thus improves the accuracy of function identification. For example, $\mathcal{B}_{547}$ is a classic function call pattern, which first pushes the return address into the stack, then

[1] https://etherscan.io/address/0x0000009d48b12597675a02fca9c317eadef152cb, compiled by solc-0.5.17 with optimization configuration.

pushes the calling function address, and then jumps to $\mathcal{B}_{2004}$. When using the graph neural network to predict $\mathcal{B}_{2004}$, the features of $\mathcal{B}_{547}$ will be integrated, making it easier for the model to determine that $\mathcal{B}_{2004}$ is a function entry. Therefore, graph models have great potential for this task and will learn more meaningful patterns and capture complex control flows.

Observation 2. The distribution of instruction types within a basic block can largely reflect the functionality of the block. Each basic block in a control flow graph typically contains a set of instructions that are executed consecutively. The types of instructions, whether arithmetic operations, stack operations, memory operations, or storage operations, can provide strong clues about the functionality of the block. There are usually some initialization operations at the function entry, such as setting up the stack frame, saving register status, etc. If a basic block contains many of these types of instructions, it may be the function entry. If there are many arithmetic operations in a basic block, it is very likely that some business operations are being performed, which are usually in a deeper function body. For example, $\mathcal{B}_{2004}$ first uses the *CALLER* instruction to obtain the caller address, which may be used to subsequently check the caller's identity or perform permission verification. The remaining instructions of $\mathcal{B}_{2004}$ are basically stack operations and read/write operations, which are generally a series of function initialization work. The instruction distribution shown in the above analysis shows that $\mathcal{B}_{2004}$ is likely to be a function entry. In addition, the instructions in $\mathcal{B}_{2025}$ are more like ordinary function body blocks that carry calculation and logging functionality. By examining the distribution and frequency of instruction types within each basic block, we can infer the role of that block within the overall contract and its relationship to other blocks. This observation highlights the importance of understanding the instruction set within blocks, which can serve as a vital feature for accurate function identification.

3 TASK DEFINITION

The function identification task can be decomposed into two subtasks with a sequential relationship, namely function entry identification and function boundary identification. Suppose a smart contract bytecode is accessible, which contains several functions $f_1, \cdots, f_n$. The two tasks are defined as follows:

Definition 1. Function Entry Identification: Given a bytecode, find a set of offset addresses $\{s_1, s_2, \cdots, s_n\}$, where s_i is the first byte offset address in each function f_i. Furthermore, the bytecode can be decomposed into several basic blocks. Therefore, the task can also be expressed as: Given a bytecode, find a set of basic blocks $\{b_1, b_2, \cdots, b_n\}$, where b_i is the basic block that starts with the first instructions in each function f_i.

Definition 2. Function Boundary Identification: Given a bytecode, find a set of $\{B_1, B_2, \cdots, B_n\}$, where each B_i consists of all basic blocks that contain the contents of function f_i, and each B_i can be represented as a set of basic blocks $\{b_1, b_2, \cdots, b_{n_i}\}$.

Our method mainly completes the first task by designing a neural network model to predict function entry. According to previous work [17, 34], based on the function entry predicted by the model, function boundaries can be identified by traversing the control flow through static analysis.

In addition, it is important to note that the compiler embeds a code segment called dispatcher at the beginning of the bytecode. The dispatcher stores the function selectors [8] of each public/external function, which is a function identifier. Before calling a public/external function, there is a basic block to determine whether the calling method matches the corresponding function selector. It is easy to infer the entry of these functions through static analysis. Therefore, we only consider private/internal functions in the evaluation of the function entry identification task.

4 APPROACH

Based on the motivation example, we propose FIRE to enrich the representation of basic blocks to better predict function entry. The overall architecture of FIRE is shown in Figure 2. We first build a control flow graph of the bytecode (§ 4.1), then perform hybrid feature encoding (§ 4.2) for each node in the graph, refine the encoding based on the control flow graph (§ 4.3), and finally conduct binary classification of function entry for each node.

4.1 Control Flow Graph Construction

To construct the control flow graph of the bytecode, we first split the bytecode and identify the basic blocks. Here we use basic blocks that are mainly divided by *JUMP*, *JUMPI* and *JUMPDEST* instructions instead of reachable blocks used in previous work [17]. This is because the reachable blocks are only split by the *JUMPDEST* instruction as a split point. This strategy causes some blocks to be merged with sequentially adjacent blocks, which destroys the topological relationship of the entire control flow graph. For example, if the JUMPI instruction does not jump, it should continue to execute downward, and the subsequent instructions should be treated as a single block until the next jump. However, the reachable block will not be split when it does not jump, so the block after the jump cannot successfully establish an adjacent relationship with the existing unsplited block. In our control flow graph (CFG), each basic block is represented as a node. By parsing the jump instruction at the end of each node, the connectivity between nodes can be determined. Simple direct jumps are easy to parse, that is, the jump address is pushed into the stack before the jump instruction. As for complex jumps, symbolic stack execution [2] is required through depth-first search to accurately obtain the jump address.

The CFG is crucial for understanding the program's execution path and enables the model to capture the relationships and transitions between different parts of the bytecode. These relationships are pivotal for accurately identifying function entries, as the function entry is typically characterized by the flow of control into the function's entry point. We leverage the CFG structure to enrich the feature representation of each basic block in subsequent steps of the approach.

4.2 Hybrid Feature Encoding

In addition to the graph structure, each node's own feature encoding scheme is the most fundamental key factor in ultimately identifying the function entry. As mentioned before (Observation 2), the instruction distribution within a basic block is an important distinguishing feature. Therefore, we use local frequency encoding as the most important node embedding in FIRE.

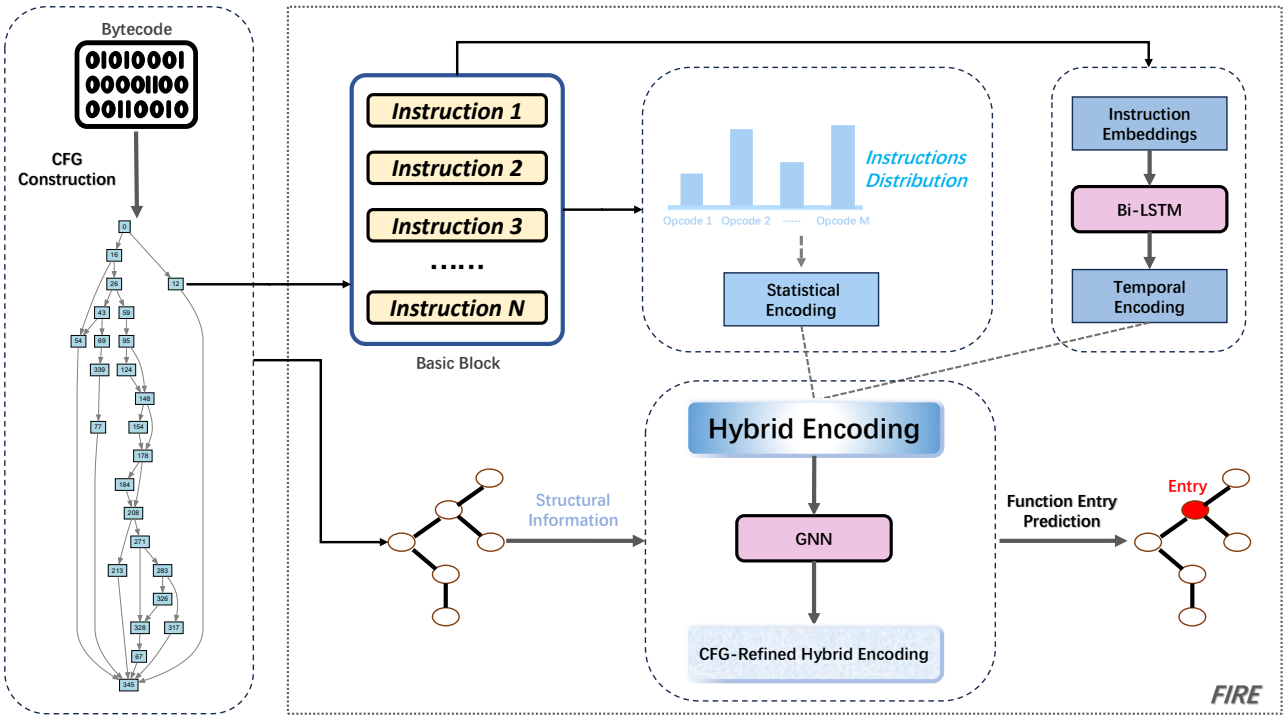


Figure 2: Overview of FIRE.

Specifically, we first count all the categories of instructions that appear in the training set, which has a total number of m. Next, we statistically calculate the instruction distribution in each node, calculate the frequency of each instruction in the node, and obtain its local frequency feature encoding $e^{freq} \in \mathbb{R}^{1\times m}$. The specific calculation method is as follows:

$$e_i^{freq} = \frac{count_i}{instrs\_num} \quad , i \in \{1, 2, \ldots, m\} \tag{1}$$

, where $count_i$ is the number of times the i-th instruction appears in node; $instrs\_num$ represents the total number of instructions in node.

It should be noted that local frequency encoding only contains statistical features, which lacks the sequential information of instructions as a kind of temporal data. We supplement this sequential information by introducing a language model. The language model also provides contextual information that aids in understanding the semantics of the instructions within the node, offering a more comprehensive feature set. We input the instructions of each node into a bidirectional LSTM [18], and the output result is average pooled to obtain the temporal information encoding e^{seq}:

$$e^{seq} = Pooling(BiLSTM(instructions)) \tag{2}$$

We concatenate the local frequency feature encoding and the local temporal feature encoding to obtain the final hybrid feature encoding e as the initial node embedding vector:

$$e = concatenate(e^{freq}, e^{seq}) \tag{3}$$

This hybrid encoding of statistical and temporal features allows the model to better represent the complexity of the bytecode.

4.3 CFG-based Refinement

Once the hybrid feature encoding is constructed, it is input into a graph neural network (GNN) that iteratively refines the encoding based on the structure of the control flow graph (CFG). The GNN allows us to capture higher-order relationships between basic blocks, as it propagates feature information across the graph edges. During this iterative refinement process, the GNN adjusts the feature representations by aggregating information from neighboring nodes, effectively enriching the feature vectors. This step enhances the model's ability to detect function entry points by considering not only the individual characteristics of each block but also the contextual and structural information provided by the CFG.

In fact, when choosing graph neural network, we use graph attention network (GAT) [36] to obtain refined feature encoding e_u^*:

$$e_u^* = \sum_{v\in\mathcal{N}_u} \phi W e_v \tag{4}$$

, where u is a node; $\mathcal{N}_u$ means u's neighborhood nodes including itself; ϕ represents the attention scores calculated by GAT. W is the trainable weights.

The GAT allow for the dynamic adjustment of feature importance through an attention mechanism, enabling each node to assign varying attention weights to its neighbors. By applying the attention mechanism, the GAT not only preserves the structural integrity

Table 1: Statistics of dataset.

	solc-0.4.25		solc-0.5.17	
	Unoptimized	Optimized	Unoptimized	Optimized
#Bytecode	30,265	30,260	9,090	9,099
#Average Function	22.0	21.8	31.76	30.74

of the CFG but also ensures that critical control flow information is emphasized in the iterative refinement process. This is particularly beneficial for identifying function entry points, where the flow of control often shifts significantly. The iterative nature of GAT further enriches the feature representation by progressively refining the embeddings of each basic block, ensuring that the final representation is highly discriminative and context-aware. The attention mechanism thus allows the model to focus on key transitions between blocks that indicate function entries, leading to a more accurate and efficient identification of function entries.

Finally, the graph-refined hybrid feature encoding of each basic block is sent to a classifier for binary classification to determine whether it is a function entry.

5 EXPERIMENT SETUP

5.1 Dataset

We conducted experiments on the dataset introduced by neural-FEBI [17], which contains 38,996 unique smart contracts. The contracts were compiled using two instrumented compiler versions (solc-0.4.25 and solc-0.5.17) and two compilation options (optimized and unoptimized), resulting in four distinct datasets, as shown in Table 1. The compiler with optimization configuration optimizes the bytecode by folding constants, inlining small functions, and eliminating redundant calculations to reduce gas costs and improve execution efficiency. This means that the bytecode with optimization configuration is more complex, which increases the difficulty of function identification.

5.2 Metrics

We use the same evaluation metrics as previous work [3, 17, 34], namely P (precision), R (recall) and F1. The calculation is as follows:

$$P = \frac{TP}{TP + FP} \tag{5}$$

$$R = \frac{TP}{TP + FN} \tag{6}$$

$$F1 = \frac{2PR}{P + R} \tag{7}$$

where TP is the number of true positive predictions, FP is the number of false positive predictions, and FN is the number of false negative predictions. The $F1$ score is the harmonic mean of precision and recall. The $F1$ score provides a single metric that balances both precision and recall, giving a more comprehensive view of the model's performance.

For the function entry prediction task, the above evaluation metrics are calculated by comparing the predicted entry block with the ground truth. The same is true for the function boundary recognition task, but a complete matching strategy is used when comparing, that is, all blocks of the predicted function body must completely match all blocks in the corresponding function ground truth. When the function boundary identification task times out or is interrupted by an error, these metrics will be recorded as 0.

5.3 Baseline

In our experiment, we set up three traditional decompilers and a state-of-the-art method that uses machine learning methods for function identification as baselines. They are introduced as follows:

- Gigahorse [14]: performs declarative program based on heuristical rules to conduct function identification in decompilation.
- Elipmoc [15]: is built on top of Gigahorse and merged into its code repository as Gigahorse 2.0. Elipmoc uses a transactional context sensitivity algorithm to improve the precision of function reconstruction, especially private functions.
- Shrnkr [23]: is an improvement on Elipmoc, adding shrinking context sensitivity analysis to improve precision.
- neural-FEBI [17]: uses the LSTM-CRF architecture to identify function entries and designs a static analysis framework for function boundary identification.

For another SOTA decompiler, Heimdall-rs [11] , we do not use it as a baseline because it does not have the ability to identify private/internal functions.

5.4 Implementation Setup

The experiments were conducted on a server equipped with an Intel(R) Xeon(R) Platinum 8358P CPU, featuring 128 cores running at 2.60 GHz, coupled with an NVIDIA A800 80GB PCIe GPU. The construction of the control flow graph of the bytecode is implemented using the EhterSolve [6] tool. We use the pytorch [29] framework to build the neural network. We use the cross-entropy loss to calculate the error and the AdamW [25] algorithm to train the network. The dimension of the instruction embedding is set to 64. The dimension of the sequential information encoding of the LSTM output is set to 32. We only use one layer of GAT to generate the graph-refined hybrid feature encoding, and its dimension is set to 128. For the running time, we set 120s as the timeout.

6 EXPERIMENT RESULTS

We assess our approach by trying to answer the following five research questions.

- **RQ1: To what extent can FIRE perform in the function entry identification task?**
- **RQ2: How does the ablation of several key components in FIRE perform?**
- **RQ3: To what extent can FIRE perform in real-world cross-version scenarios?**
- **RQ4: To what extent can FIRE improve the effectiveness of boundary identification?**
- **RQ5: To what extent can FIRE improve the efficiency of boundary identification?**

6.1 RQ1: To what extent can FIRE perform in the function entry identification task?

Motivation and Setting. FIRE has been carefully and uniquely designed for function entry identification. We hope to use this RQ

Table 2: Comparison of different models for the internal function entries identification.

Model	solc-0.4.25						solc-0.5.17					
	Unoptimized			Optimized			Unoptimized			Optimized		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Gigahorse	0.9579	0.8586	0.8799	0.9441	0.8270	0.8571	0.8756	0.6905	0.7361	0.8695	0.6835	0.7284
Elipmoc	0.9646	0.8642	0.8859	0.9520	0.8338	0.8644	0.9490	0.7493	0.8012	0.9326	0.7340	0.7843
Shrnkr	0.9727	0.8713	0.8934	0.9619	0.8427	0.8736	0.9701	0.7653	0.8194	0.9567	0.7542	0.8068
neural-FEBI	0.9958	**0.9983**	0.9955	0.9780	0.9366	0.9455	0.9928	0.9891	0.9877	0.9514	0.8455	0.8725
FIRE	**0.9973**	0.9955	**0.9956**	**0.9962**	**0.9936**	**0.9936**	**0.9942**	**0.9914**	**0.9911**	**0.9910**	**0.9922**	**0.9901**

to evaluate the effect of FIRE on the function entry identification task. Consistent with previous work [17], we divide the training set, validation set, and test set into a ratio of 4:1:5.

As shown in Table 2, the FIRE model outperforms all other models in almost every aspect, particularly in terms of F1-score, which is a balanced measure of both precision and recall. For the solc-0.4.25 dataset, FIRE achieves an F1-score of 0.9956 in the unoptimized setting and 0.9936 in the optimized setting, which are the highest among all models evaluated. Similarly, for the solc-0.5.17 dataset, FIRE achieves F1-scores of 0.9911 in the unoptimized setting and 0.9901 in the optimized setting.

The performance of the three compilers used as the baseline of traditional methods has gradually improved. Shrnkr, which has the most comprehensive context-sensitivity algorithm, performs the best, but is still far from the machine learning method. The F1-score of our method is about 0.1-0.2 higher than that of Shrnkr on different datasets. This shows that relying solely on heuristic rules to identify function entry has great limitations. Compared with neural-FEBI, our model outperforms it in all metrics, except for the recall of the solc-0.4.25-unoptimized dataset. This may be related to the incompleteness of the control flow graph. Some function entry blocks are not included in the control flow graph. We will explain this in detail in Section 7. However, on this dataset, FIRE shows comparable and not inferior performance to neural-FEBI, and the same is true on the solc-0.5.17-unoptimized dataset using non-optimized compilation options.

It is worth noting that on the dataset with non-optimized compilation options, the neural network-based models all performed very well. This may be because the function entry patterns hidden in the bytecode generated by non-optimized compilation are relatively simple, and the network can easily learn its rules. When the compilation option is changed to optimized configuration, the performance of all models except FIRE will experience a significant decline. The optimization options make the structure of the bytecode more complex, requiring the model to have more powerful modeling capabilities to maintain high performance.

FIRE uses graph-refined hybrid feature encoding to integrate local statistical features, local temporal features, and global structural features. This greatly enriches the semantic information, allowing it to still have excellent performance on datasets that use compilation optimization options. Regardless of whether compilation optimization options are used, FIRE can achieve a performance of more than 0.99 in F1-score, demonstrating strong algorithm stability. This positions FIRE as a highly effective solution for function entry identification in smart contract decompilation tasks.

Summary: *FIRE outperforms all other models in function entry identification. Regardless of whether the dataset uses compilation optimization options or not, FIRE can achieve an F1-score of more than 0.99, demonstrating its effectiveness and stability.*

6.2 RQ2: How does the ablation of several key components in FIRE perform?

Motivation and Setting. In the FIRE architecture, there are three key components, namely statistical encoding, temporal encoding, and the CFG-based refinement module. In this RQ, we hope to demonstrate the effectiveness of each component through ablation experiments. We set up three control methods and conducted experiments under the same settings as RQ1:

- MLP_{SE}**:** Only statistical feature encodings are used as input to the MLP [13] to demonstrate **observation 2**.
- **FIRE wo/ SE:** FIRE removes the statistical encoding component.
- **FIRE wo/ TE:** FIRE removes the temporal encoding component.
- **FIRE wo/ GR:** FIRE removes the CFG-based refinement component.

The ablation results in Table 3 reveal the necessity of all three components in FIRE. First, removing the CFG-based refinement module (FIRE wo/GR) causes the most significant performance drop across all scenarios, especially for solc-0.5.17 (e.g., F1 decreases from 0.9911 to 0.8467 in unoptimized cases). This suggests that structural pattern mining through CFG-based refinement is critical for capturing deep code logic features, particularly for newer compiler versions where code semantics may be more complex. Second, ablating the statistical encoding component (FIRE wo/SE) results in a more noticeable performance decline compared to removing the temporal encoding component (FIRE wo/TE). For example, in solc-0.4.25-unoptimized dataset, F1 drops from 0.9956 to 0.9835 when statistical encoding is removed, whereas removing temporal encoding only reduces F1 to 0.9897. And then, MLP_{SE}, which relies only on statistical feature encoding, also achieved good results. This suggests that statistical features provide a stronger foundational signal for function entry identification than temporal patterns in

Table 3: Comparison of experimental results after FIRE ablates various components.

Model	solc-0.4.25						solc-0.5.17					
	Unoptimized			Optimized			Unoptimized			Optimized		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
MLP_{SE}	0.8804	0.9426	0.8846	0.9021	0.9342	0.8949	0.8042	0.9170	0.8304	0.8068	0.9110	0.8300
FIRE wo/SE	0.9860	0.9840	0.9835	0.9854	0.9831	0.9827	0.9838	0.9820	0.9813	0.9818	0.9806	0.9794
FIRE wo/TE	0.9933	0.9904	0.9897	0.9919	0.9882	0.9870	0.9884	0.9840	0.9836	0.9851	0.9824	0.9800
FIRE wo/GR	0.9113	0.9462	0.9061	0.8940	0.9547	0.9006	0.7999	0.9489	0.8467	0.8241	0.9295	0.8492
FIRE	**0.9973**	**0.9955**	**0.9956**	**0.9962**	**0.9936**	**0.9936**	**0.9942**	**0.9914**	**0.9911**	**0.9910**	**0.9922**	**0.9901**

Table 4: Comparison of different neural models for the internal function entries identification in cross-version scenarios.

Model	0.4.25-U → 0.5.17-U			0.4.25-O → 0.5.17-O		
	P	R	F1	P	R	F1
neural-FEBI (40% training data)	0.6644	**0.9911**	0.7458	0.6236	0.8468	0.6530
neural-FEBI (100% training data)	0.6250	0.8362	0.6327	0.6222	**0.9924**	0.7069
FIRE (40% training data)	0.8109	0.9767	0.8432	**0.8850**	0.9738	0.9024
FIRE (100% training data)	**0.9074**	0.9830	**0.9280**	0.8816	0.9834	**0.9171**

instructions, which also shows that **Observation 2** is correct. Finally, while temporal encoding also contributes to performance, its removal has a relatively smaller impact, indicating that temporal dependencies provide supplementary but less critical information compared to statistical and structural features. Overall. the full FIRE architecture, with all components intact, outperforms the individual ablations, reaffirming the necessity of combining these techniques to achieve optimal results.

Summary: *All three components——statistical encoding, temporal encoding, and CFG-based refinement, are essential for FIRE's optimal performance, with CFG-based refinement being the most critical.*

6.3 RQ3: To what extent can FIRE perform in real-world cross-version scenarios?

Motivation. In Table 2, we can observe that the F1-score of all models on the high-version compiled data is lower than that on the low-version data. This is because high-version compilers usually introduce more advanced compilation optimization techniques. This also introduces a new question: Can low-version data be used for training to predict high-version data? This is actually a real-world scenario. The production of existing datasets relies on the modification of compilers to perform program instrumentation analysis to obtain ground-truth. If the model trained on low-version data cannot predict high-version data well, it is necessary to modify the high-version compiler to obtain high-version data for training, which is undoubtedly costly. Such a high annotation cost poses a challenge to the generalization of the model, so we hope to use this RQ to study the cross-version prediction performance of FIRE in real scenarios.

Setting. In RQ1, we have trained a model with excellent performance on low-version data using 40% of the data. We first evaluate the performance of these models trained with only 40% of the low-version data on the high-version data. And then, we set up a full low-version data training scenario to explore the model's efficiency in utilizing data and compare the overfitting effects of different models with increased training data. Since traditional decompilers do not rely on training, they do not have generalization problems. Here we only compare FIRE with neural-FEBI.

As shown in Table 4, in the 40% data training scenario, FIRE achieved an F1-score of 0.8432 and 0.9024 on the datasets with unoptimized compilation options and optimized compilation options, respectively. Compared with neural-FEBI, our results are 13% and 38% higher, respectively. This shows that FIRE is strongly robust and can handle the bytecode function entry identification task in real scenarios. In the full data training scenario, as shown in the Table 4, FIRE achieves an F1-score of 0.9280 and 0.9171 in the data without compilation optimization options and the data with compilation optimization options, respectively. This performance is about 47% and 30% higher than neural-FEBI, respectively.

It can be noted that the increase in low-version training data does not make FIRE suffer from the threat of overfitting. FIRE performs better when trained with all low-version data than when trained with only part of the low-version data. This shows that FIRE's careful design can capture more subtle function entry patterns, enabling it to use more data to enhance its performance and perform well in high-version data prediction. In contrast, neural-FEBI suffers from performance degradation when the training data increases.

In addition, the experimental results show that neural-FEBI performs well in the recall evaluation metric. However, its precision lags far behind, resulting in poor overall performance. This shows that neural-FEBI has learned rough patterns that are only applicable to low-version data, and has not captured the fine patterns that

Table 5: Comparison of different models for the function boundaries identification.

Model	solc-0.4.25						solc-0.5.17					
	Unoptimized			Optimized			Unoptimized			Optimized		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Gigahorse	0.3608	0.4200	0.3868	0.1661	0.1962	0.1791	0.2641	0.2946	0.2771	0.1239	0.1416	0.1309
Elipmoc	0.3627	0.4219	0.3886	0.1670	0.1972	0.1800	0.2814	0.3146	0.2956	0.1269	0.1450	0.1341
Shrnkr	0.3647	0.4243	0.3908	0.1680	0.1984	0.1810	0.2860	0.3192	0.3002	0.1280	0.1468	0.1357
neural-FEBI	0.9638	0.9629	0.9633	0.9440	0.9367	0.9400	0.9211	0.9146	0.9175	0.8438	0.8165	0.8290
FIRE	**0.9794**	**0.9777**	**0.9785**	**0.9732**	**0.9716**	**0.9723**	**0.9663**	**0.9566**	**0.9608**	**0.9592**	**0.9477**	**0.9524**

Table 6: Statistics of runtime (seconds) of different models (excluding timeout).

Model	solc-0.4.25				solc-0.5.17			
	Unoptimized		Optimized		Unoptimized		Optimized	
	Mean	Median	Mean	Median	Mean	Median	Mean	Median
Gigahorse	3.79	2.36	2.99	1.78	5.32	2.30	5.11	1.96
Elipmoc	3.07	2.41	2.82	2.20	4.70	2.40	5.60	2.05
Shrnkr	3.04	2.74	2.76	2.53	3.24	2.64	2.87	2.36
neural-FEBI	6.66	1.20	11.04	1.62	7.52	0.97	20.46	3.69
FIRE	**1.25**	**0.91**	**1.06**	**0.67**	**1.94**	**0.84**	**1.87**	**0.63**

really affect the function boundary, making its generalization far inferior to FIRE.

These results demonstrate that FIRE not only performs well within a single version of the Solidity compiler but also exhibits strong generalization capabilities across different compiler versions. Whether trained with a limited amount of data (40%) or a larger dataset (100%), FIRE maintains superior performance compared to neural-FEBI, making it highly effective for real-world cross-version scenarios.

Summary: *FIRE achieving an F1-score over 0.9 in cross-version prediction tasks, significantly outperforming the SOTA. It effectively avoids overfitting even with an increasing amount of low-version training data, showcasing high data utilization efficiency.*

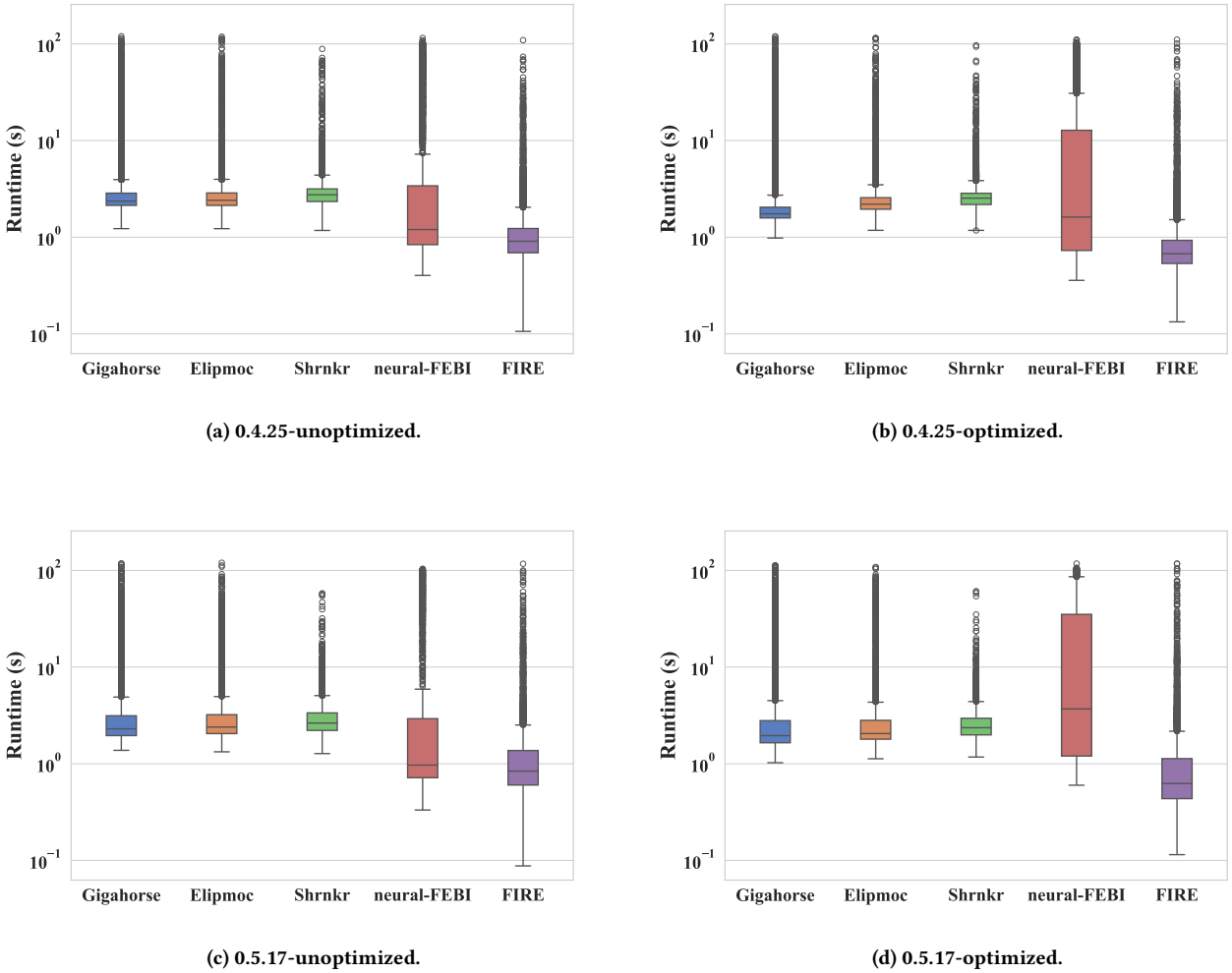
6.4 RQ4: To what extent can FIRE improve the effectiveness of boundary identification?

Motivation and Setting. Identification of the function entry is a prerequisite for identification of the function boundary, and the two together constitute the overall task of function identification. FIRE uses the boundary detection framework in neural-FEBI for boundary identification. This RQ aims to assess the enhancement in the effectiveness of function boundary identification achieved by FIRE.

The results, as presented in Table 5, clearly demonstrate FIRE's superiority in boundary identification tasks. FIRE achieves the best results across all metrics and datasets. On the most challenging solc-0.5.17-optimized dataset, the F1-score improves by 15% compared with neural-FEBI. This demonstrates the improvement in function entry identification accuracy brought by FIRE, and also brings stable improvements to downstream function boundary identification tasks. The traditional decompilers—-Gigahorse, Elipmoc, and Shrnkr, show consistently lower performance across all metrics. For instance, Gigahorse, Elipmoc, and Shrnkr all struggle with low precision and recall, particularly in the optimized settings, where their F1-scores range from 0.1791 to 0.3002 in solc-0.4.25 and from 0.1309 to 0.2771 in solc-0.5.17. The reason why neural-FEBI and FIRE are much higher than them may be due to the boundary detection framework used by neural-FEBI. Since the function entry prediction output of FIRE is very consistent with the input of this framework, it can pass on the advantages established in the upstream task to the downstream task.

It can be noted that in the solc-0.4.25-unoptimized dataset, neural-FEBI and FIRE have similar performance in the function entry identification task, but FIRE is slightly better in the function boundary identification task. This is because neural-FEBI times out when predicting some data and cannot produce function boundary results. The study of operating efficiency will be described in the next research question.

Summary: *FIRE significantly enhances the effectiveness of function boundary identification, achieving the best results across all metrics and datasets, with an impressive 15% improvement in F1-score over neural-FEBI on the most challenging dataset.*



(a) 0.4.25-unoptimized.

(b) 0.4.25-optimized.

(c) 0.5.17-unoptimized.

(d) 0.5.17-optimized.

Figure 3: Runtime of function identification on test set (excluding timeout).

6.5 RQ5: To what extent can FIRE improve the efficiency of boundary identification?

Motivation and Setting. In this RQ, we evaluate the efficiency of the FIRE model in function boundary identification tasks, focusing on its runtime performance compared to other models. The goal is to assess whether FIRE not only achieves high accuracy but also does so in a computationally efficient manner, which is crucial for real-world applications.

The results in Figure 3 illustrate the runtime performance distribution of various models on the test sets, excluding any instances where the process timed out. As can be seen from the figure, FIRE consistently exhibits much lower runtimes than the other models, particularly in the optimized settings.

Table 6 presents detailed statistics on the runtime for each model, including the mean and median runtimes in seconds. FIRE outperforms all other models in terms of both mean and median runtime, demonstrating its efficiency in boundary identification. For solc-0.4.25, FIRE achieves a mean runtime of 1.25 seconds in the unoptimized setting and 1.06 seconds in the optimized setting, with median runtimes of 0.91 seconds and 0.67 seconds, respectively. These times are significantly faster than the other models, which range from 2.99 seconds (Gigahorse) to 11.04 seconds (neural-FEBI) in the optimized setting. Similarly, for solc-0.5.17, FIRE's mean runtime is 1.94 seconds in the unoptimized setting and 1.87 seconds in the optimized setting, with median runtimes of 0.84 seconds and 0.63 seconds, respectively. These results are again much faster compared to the other models, with the slowest model, neural-FEBI, taking 20.46 seconds on average in the optimized setting.

According to our analysis, the reason why neural-FEBI has a low operating efficiency is due to the model architecture it adopts. The CRF module in neural-FEBI involves complex inference over a large number of possible label sequences, which significantly increases computational complexity. Moreover, its function boundary detection framework involves multiple operations of Viterbi

Decoding [27], which leads to a large number of timeouts. As for our approach, excluding the construction of CFG, the speed of FIRE in function boundary detection is in the millisecond level.

The efficiency of FIRE is particularly notable when considering the performance trade-off. Despite its significantly faster runtime, FIRE maintains the highest performance in terms of precision, recall, and F1-score, as demonstrated in the previous sections. This makes FIRE not only an accurate solution for function boundary identification but also a highly efficient one, making it suitable for large-scale and real-time applications in smart contract analysis.

Summary: *FIRE significantly improves the efficiency of function boundary identification, achieving much lower runtimes compared to other models, while maintaining high accuracy.*

7 THREATS TO VALIDITY

Internal Validity. First, if the control flow graph of the bytecode fails to be constructed, FIRE cannot perform function identification. FIRE uses EtherSolve [6], the state-of-the-art control flow graph construction tool, to build the control flow graph. If EtherSolve fails, FIRE will not be able to obtain the graph structure for reasoning. However, such scenarios are very rare. In the data set, only about 0.04% of the contracts cannot be processed by EhterSolve. FIRE also has the ability to degenerate and retain the availability of prediction functionality when the graph structure cannot be obtained.

In addition, the quality of the CFG generated by EtherSolve also affects the effect of function identification. Although EtherSolve is the current state-of-the-art method, there may be some deviations in the generation of its control flow graph. We observed that EtherSolve may mistakenly identify the contract termination block in advance, and the rest is regarded as a data segment of the bytecode. However, there may be a small amount of function body content in these bytecodes regarded as data segments. Fortunately, this inaccuracy has little impact in our dataset, affecting only 0.01% of the functions. Moreover, our experimental results show that these defects will not have much impact on the final results.

External Validity. Due to the inherent overfitting problem in machine learning [40], FIER may face some generalization challenges. We used many common regularization methods such as L2 regularization to mitigate the impact of overfitting. And we aggregated local statistical features, local sequential features, and global structural features through graph-refined hybrid feature encoding. This makes the feature modeling source of basic blocks richer and makes the model performance less sensitive to the differences between different compilers. In RQ3, we simulated a real scenario where a model trained with low-version data predicts high-version data. The experiment shows that FIRE has sufficient generalization and is not subject to the risk of overfitting caused by the increase in low-version training data.

8 RELATED WORK

8.1 Smart Contract Decompilation

Smart contracts are typically written in high-level programming languages such as Solidity [9], which are then compiled into bytecode running on Ethereum Virtual Machine (EVM). EVM bytecode is lower-level than bytecodes of other languages such as JVM [35] bytecode, making its reverse engineering process more difficult than other languages.

Vandal [5] is a logic-driven decompiler that translates bytecode into an intermediate representation in the form of logical relations. Gigahorse [14] performs declarative program based on logical rules to produce a 3-address intermediate representation that is easier to analyze. Panoramix [28] is an official decompiler used by Etherscan [10], which has a certain ability to regenerate source code. Elipmoc [15] is built on top of Gigahorse and merged into its code repository as Gigahorse 2.0. Elipmoc uses a transactional context sensitivity algorithm to improve the precision of function reconstruction, especially private functions. Dedaub [7] is a web application that includes a decompilation function developed on the basis of Elipmoc. EthervmDec [11] is another online decompiler that can parse Ethereum deployment contracts. Heimdall-rs [4] is a smart contract bytecode analysis tool written in Rust [26] language, which has extremely fast decompilation speed. Shrnkr [23] is an improvement on Elipmoc, using a shrinking context sensitivity analysis algorithm, achieving the state-of-the-art decompiler.

Among these decompilers, Gigahorse, Elipmoc, and Shrnkr, which are used as our experimental baselines, are the most commonly used decompilers. Other decompilers are either not open source, cannot infer private functions, or have poor performance.

8.2 Function Identification

Function identification has always been a hot topic in decompilation. In addition to the several smart contract decompilers mentioned in the previous section, there are also many static analysis tools in C language that use rule-based methods to infer function boundaries such as Dyninst [16] and IDA Pro [12].

Function identification can be decomposed into two subtasks: function entry identification and boundary identification. The first task can be easily modeled as a classification task. And due to the existence of a large amount of binary data, training a classifier through machine learning has become an important way to identify function entry. Rosenblum et al. [31] first introduced machine learning methods for function identification and predicted function entry by combining logistic regression with conditional random fields. ByteWeight [3] is also a supervised learning method that uses weighted prefix trees to speed up model training. Shin et al. [34] used recurrent neural networks to improve the accuracy of function identification.

The low-level EVM bytecode is very different from C binaries. It lacks a lot of high-level meta information, making its function identification difficult. Neural-FEBI [17] is the only one that uses machine learning methods to perform function identification on smart contract bytecode. It regards bytecode as a natural language sequence, fuses features through dual-granularity BiLSTM representation, and performs part-of-speech tagging through conditional random fields, achieving a certain prediction effect. The method proposed in this paper starts from the perspective of control flow graph and combines local information with global information to achieve better prediction results.

9 CONCLUSION

In this paper, we introduced FIRE, an efficient and accurate method for identifying function boundaries in Ethereum smart contract bytecode. By integrating CFG-refined hybrid feature encoding, FIRE not only enhances the effectiveness of function identification but also improves runtime efficiency, making it highly scalable. Our results show that FIRE outperforms existing methods, particularly in terms of real-world cross-version generalization. Future work will focus on further refining the approach for more complex scenarios or other smart contract languages.

ACKNOWLEDGMENTS

This work is supported by the Zhejiang Province "JianBingLingYan+X" Research and Development Plan (2025C02020).

DATA AVAILABILITY

The replication package, which includes the source code, datasets and experiment results, can be found at https://github.com/augustus618/FIRE.

REFERENCES

[1] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.
[2] 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.
[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*. 845–860.
[4] Jonathan Becker. 2024. *Heimdall-rs.* https://github.com/Jon-Becker/heimdall-rs
[5] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).
[6] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. 2021. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 127–137.
[7] Dedaub. 2024. Dedaub Decompiler. https://app.dedaub.com/decompile.
[8] Solidity Documentation. 2024. *ABI Specification: Function Selector.* https://docs.soliditylang.org/en/v0.8.26/abi-spec.html#function-selector
[9] Solidity Documentation. 2024. *Solidity Documentation.* https://docs.soliditylang.org
[10] Etherscan. 2024. *Ethereum (ETH) blockchain explorer.* https://etherscan.io/
[11] EthervmDec. 2024. *Online Solidity Decompiler.* https://ethervm.io/decompile
[12] Justin Ferguson and Dan Kaminsky. 2008. *Reverse engineering code with IDA Pro.* Syngress.
[13] Matt W Gardner and SR Dorling. 1998. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment* 32, 14-15 (1998), 2627–2636.
[14] 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)*. IEEE, 1176–1186.
[15] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: Advanced decompilation of ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.
[16] Laune C Harris and Barton P Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News* 33, 5 (2005), 63–68.
[17] 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.
[18] S Hochreiter. 1997. Long Short-term Memory. *Neural Computation MIT-Press* (1997).
[19] Wen Hu, Zhipeng Fan, and Ye Gao. 2019. Research on smart contract optimization method on blockchain. *IT Professional* 21, 5 (2019), 33–38.
[20] Kun Jing and Jungang Xu. 2019. A survey on neural network language models. *arXiv preprint arXiv:1906.03591* (2019).
[21] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Ethereum smart contract analysis tools: A systematic review. *Ieee Access* 10 (2022), 57037–57062.
[22] John Lafferty, Andrew McCallum, Fernando Pereira, et al. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Icml*, Vol. 1. Williamstown, MA, 3.
[23] Sifis Lagouvardos, Yannis Bollanos, Neville Grech, and Yannis Smaragdakis. 2024. The Incredible Shrinking Context... in a decompiler near you. *arXiv preprint arXiv:2409.11157* (2024).
[24] Wenkai Li, Xiaoqi Li, Zongwei Li, and Yuqing Zhang. 2024. Cobra: Interaction-aware bytecode-level vulnerability detector for smart contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1358–1369.
[25] I Loshchilov. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).
[26] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.
[27] J Omura. 1969. On the Viterbi decoding algorithm. *IEEE transactions on information theory* 15, 1 (1969), 177–179.
[28] palkeo. 2024. Panoramix. https://github.com/palkeo/panoramix.
[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[30] Peng Qian, Zhenguang Liu, Qinming He, Butian Huang, Duanzheng Tian, and Xun Wang. 2022. Smart contract vulnerability detection technique: A survey. *arXiv preprint arXiv:2209.05872* (2022).
[31] Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code.. In *AAAI*. 798–804.
[32] Sara Rouhani and Ralph Deters. 2019. Security, performance, and applications of smart contracts: A systematic survey. *IEEE Access* 7 (2019), 50759–50779.
[33] Helmut Schmid. 1994. Part-of-speech tagging with neural networks. *arXiv preprint cmp-lg/9410018* (1994).
[34] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *24th USENIX security symposium (USENIX Security 15)*. 611–626.
[35] Robert F Stärk, Joachim Schmid, and Egon Börger. 2012. *Java and the Java virtual machine: definition, verification, validation.* Springer Science & Business Media.
[36] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *stat* 1050, 20 (2017), 10–48550.
[37] Zeli Wang, Hai Jin, Weiqi Dai, Kim-Kwang Raymond Choo, and Deqing Zou. 2021. Ethereum smart contract security research: survey and future research opportunities. *Frontiers of Computer Science* 15 (2021), 1–18.
[38] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[39] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.
[40] Xue Ying. 2019. An overview of overfitting and its solutions. In *Journal of physics: Conference series*, Vol. 1168. IOP Publishing, 022022.
[41] 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*. 745–757.
[42] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. 2017. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)*. Ieee, 557–564.
[43] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (2020), 475–491.
[44] Di Zhu, Feng Yue, Jianmin Pang, Xin Zhou, Wenjie Han, and Fudong Liu. 2022. Bytecode similarity detection of smart contract across optimization options and compiler versions based on triplet network. *Electronics* 11, 4 (2022), 597.