Augmenting Smart Contract Decompiler Output Through Fine-Grained Dependency Analysis and LLM-Facilitated Semantic Recovery

Zeqin Liao, Yuhong Nan, *Member, IEEE*, Zixu Gao, Henglong Liang, Sicheng Hao, Peifan Ren, and Zibin Zheng, *Fellow, IEEE*

***Abstract*—Decompiler is a specialized type of reverse engineering tool extensively employed in program analysis tasks, particularly in program comprehension and vulnerability detection. However, current Solidity smart contract decompilers face significant limitations in reconstructing the original source code. In particular, the bottleneck of SOTA decompilers lies in inaccurate function identification, incorrect variable type recovery, and missing contract attributes. These deficiencies hinder downstream tasks and understanding of the program logic. To address these challenges, we propose SmartHalo, a new framework that enhances decompiler output by combining static analysis (SA) and large language models (LLM). SmartHalo leverages the complementary strengths of SA's accuracy in control and data flow analysis and LLM's capability in semantic prediction. More specifically, SmartHalo constructs a new data structure - Dependency Graph (DG), to extract semantic dependencies via static analysis. Then, it takes DG to create prompts for LLM optimization. Finally, the correctness of LLM outputs is validated through symbolic execution and formal verification. Evaluation on a dataset consisting of 465 randomly selected smart contract functions shows that SmartHalo significantly improves the quality of the decompiled code, compared to SOTA decompilers (e.g., Gigahorse). Notably, integrating GPT-4o mini with SmartHalo further enhances its performance, achieving a precision of 91.32% and a recall of 87.38% for function boundaries, a precision of 90.40% and a recall of 88.82% for variable types, and a precision of 80.66% and a recall of 91.78% for contract attributes.**

***Index Terms*—Smart contract, decompilation, static analysis, large language model.**

Received 12 February 2025; revised 1 October 2025; accepted 12 October 2025. Date of publication 20 October 2025; date of current version 12 December 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB2704100, in part by the National Natural Science Foundation of China under Grant 62572497, in part by NSFC-RGC Collaborative Research under Grant 62461160332, and in part by Guangdong Zhujiang Talent Program under Grant 2023QN10X561. Recommended for acceptance by G. Fraser. *(Corresponding author: Yuhong Nan.)*

The authors are with the School of Software Engineering, Sun Yat-sen University, Zhuhai 519082, China, and also with GuangDong Engineering Technology Research Center of Blockchain, Zhuhai 519082, China (e-mail: liaozq8@mail2.sysu.edu.cn; gaozx9@mail2.sysu.edu.cn; lianghlong@mail2.sysu.edu.cn; haosch@mail2.sysu.edu.cn; renpf@mail2.sysu.edu.cn; nanyh@mail.sysu.edu.cn; zhzibin@mail.sysu.edu.cn).

Digital Object Identifier 10.1109/TSE.2025.3623325

I. Introduction

DECOMPILER is a specific type of reverse engineering tool widely used for program analysis, such as program comprehension and vulnerability detection [1]. For smart contracts written in Solidity, the goal of the decompiler is to recover the machine-executable code (e.g., the EVM bytecode [2]) back to the original source code written by developers (e.g., the Solidity program [3]).

While decompilers recover abstractions that enhance code readability and are widely used by reverse engineers, decompilers are incapable of fully reconstructing the original developer-written code, as the compiler irreversibly subverts crucial information for optimization [4]. In particular, critical information such as variable types, function boundaries, variable names, and annotation are absent in contract bytecode, and are unrecoverable by decompilers. As pointed out by prior research [5], [6], [7], these limitations make the state-of-the-art solidity decompilers, such as Gigahorse [8] and Etherscan decompiler [9], share the following three deficiency in their output: (1) inaccurate function identification: current decompilers often fail to precisely determine function boundaries, leading to the omission of important functions or erroneous function range identification. For example, decompilers often incorrectly recover multiple functions as a single function, which obviously increases complexity in certain tasks (e.g., function-level similarity comparison). (2) inaccurate variable type recovery: decompilers may produce type errors that are inconsistent with static domain rules. For instance, the decompiler often ignores predefined type (e.g., the return value of hash function *keccak256* is the type of *bytes32*), while recovering them as the type of *uint256* uniformly. (3) lack of contract attributes. Smart contracts employ state variables to record critical contract attributes (e.g., *Asset*, *identity*, *router*). Although these contract attributes are explicitly stated in the source code through meaningful variable names and annotations, they are entirely omitted in the decompiled code. As highlighted in prior work [5], contract attribute is crucial for vulnerability detection tasks.

Recent advancements [5], [6], [10], [11], [12] start optimizing or reconstructing the absent contract information such as variable type by analyzing the context of decompiler output, even when these information is not part of the contract bytecode. These works are far from satisfied, due to their limited scope and approach-wise weaknesses. More specifically, SmartDagger [5], a framework for detecting cross-contract vulnerability, can partially recover contract attributes (e.g., asset) for state variables from the decompiled bytecode. Neural-FEBI [12] is designed for identifying function boundaries, but it does not support boundary recovery for complex functions such as modifier functions or functions inherited from other contracts/subcontracts. SigRec [11], VarLifter [6] and DeepInfer [10] can partially recover the variable types such as parameter types of already-known function signatures. However, SigRec [11] and VarLifter [6] are heavily relied on pre-defined heuristics according to EVM instructions. Besides, both SmartDagger [5], Neural-FEBI [12] and DeepInfer [10] are based on deep-learning models (e.g., deep neural networks) for information recovery. Therefore, their capability are quite limited to the model-scale and training datasets. For example, they suffers performance degradation when confronted with newly-emerging or rare contracts that are not exist in the training datasets.

Our work. In this paper, we propose SmartHalo, a novel framework designed to optimize the output of existing smart contract decompilers. In particular, SmartHalo aims at recovering *function boundaries*, *variable types*, and *contract attributes* through a novel combination of static analysis (SA) and large language models (LLMs). The output of SmartHalo can facilitate a number of downstream tasks in program comprehension and security analysis. For example, eliminating false positives and false negatives for vulnerability detection.

Our key insight is that (1) Software exhibits the natural patterns - programmers tend to utilize similar code structures, contract attributes, variable types, and function boundaries in comparable contexts [13]. This repetitiveness enables predicting highly probable contract attributes, function boundaries and variable types for similar contexts. (2) SA and LLMs can collaboratively enhance the output of existing decompilers. Specifically, the advantage of SA lies in its soundness, as it is accurate in handling the optimization targets with complex static constraints [14]. Consider the optimization of variable types as an example. The types of state variables (e.g., *address*) are inherently related to the program expressions that access or modify them (e.g., *msg.sender*). In the meantime, the strength of LLM lies in its completeness, as it possesses the flexibility to predict the optimization targets that lack static constraints [15]. For instance, with the remarkable capability of few-shot in-context learning, LLMs are flexible to predicting the types for rare local variables which are incapable of inferring their types based on the types of other variables.

SmartHalo extracts the program dependencies within the program through SA, and then leverages the superior generalization capabilities of LLMs to optimize the decompiled code after learning these program dependencies. Firstly, SmartHalo extracts three types of dependencies from the decompiled code, including state dependency (e.g., read and write on state variable), control-flow dependency, and type dependency, to construct a fine-grained Dependency Graph (DG) (see Section IV-A). The DG contains static domain knowledge (i.e., dependencies) necessary for decompiler output optimization, which facilitates the LLMs in capturing them across the entire smart contract. Secondly, SmartHalo uses the DG to create prompts that include optimization target contexts (i.e., functions or variables), optimization result candidates (e.g., type or attribute), and chain-of-thoughts that represent inference steps of static analysis for optimization targets. By learning the prompts, LLMs enable the joint optimization on decompiler output with the advantages of both SA and LLMs (see Section IV-B).

A common challenge in LLM-adaptation is to handling its potential hallucination. In our task, LLMs unexpectedly alter the program behaviors of the original decompiled code, and introduce inference errors that contradict common static domain rules (e.g., type knowledge)(see Section IV-C). As countermeasure, SmartHalo conducts the rigorous correctness verification for the LLM output. Specifically, SmartHalo utilizes symbolic execution and formal verification to validate the program-behavior equivalence between the original decompiled code and optimized code. In addition, SmartHalo integrates a set of static rules (e.g., type rules) to identify and reject inference errors.

Evaluation. To evaluate the effectiveness of SmartHalo, we randomly select 500 functions from the largest open-source contract dataset [16], and manually labelled a dataset containing 456 pairs of source code and decompiler outputs of smart contract functions (44 functions encountered decompilation error). Noting that the dataset size is similar to those used in SOTA studies [4], [17]. The evaluation results indicate that, compared to the original decompiler output (i.e., Gigahorse [8]), SmartHalo (with GPT-3.5) improves the precision and recall of function boundary identification by 20.30% and 30.03%. Further, SmartHalo significantly outperforming tool SOTA VarLifter [6], effectively enhances the precision and recall of variable type inference by 13.51% and 77.08%. Additionally, SmartHalo significantly outperforming SOTA tool SmartDagger [5], successfully improves the precision and recall of contract attributes by 44.69% and 80.86% With the help of SmartHalo, 60.22% of optimized codes can directly be recompiled using the Solidity compiler, whereas the original decompiled output can not support recompilation. We also prove that the optimized output of SmartHalo enhances the effectiveness of downstream tasks, i.e., vulnerability detection. For example, SmartHalo enhances the precision by 21.96% and the recall by 38.00% on detecting integer overflow, improve the precision by 16.67% for contract attack identification.

In summary, this paper makes the following contributions:

- We highlight the key limitations of current smart contract decompiler outputs that resistant various program analysis tasks in this domain.
- We propose SmartHalo, a novel framework for optimizing smart contract decompiler output in a generic manner. We propose a set of novel mechanisms that combine static analysis and large language models, to establish accurate and flexiable optimization in decompiler output.
- We perform extensive evaluation to show the effectiveness of SmartHalo. We demonstrate the efficacy of SmartHalo via three downstream tasks for vulnerability detection.

- To benefit future research, we release the artifact of SmartHalo, as well as the datasets[1].

II. Background and Motivation

A. Smart Contract Decompiler

Smart contract is a specific type of program running on the blockchain [18], which has enabled a wide range of application scenarios in the digital world [19], such as Decentralized Finance [1], Supply Chain Management [20], and Internet of Things [21]. Recently, the investigation report show that more than 99% smart contracts do not disclose their source code, smart contract decompilation become increasingly important.

Decompilation involves the recovery of variables, functions, and control-flow abstractions through various program analysis methods. Further, it then utilizes heuristic rules to synthesize these abstractions, thereby reconstructing the high-level code representation, commonly referred as decompiled code [15]. Recently, an investigation report reveals that over 99% of smart contracts do not make their source code publicly available, with only their bytecode being accessible [22]. Therefore, smart contract decompilation is becoming increasingly important.

Smart contract decompilation faces significant challenge in accurately restoring bytecode to its original source code [23]. This challenge is primarily caused by two factors: (1) the compiler may lose crucial information about the original program. For example, the lexical parsing stage of the compiler fails to propagate *variable types*, *variable names*, *annotation* and *function boundary* into the bytecode [24]; and (2) the heuristic rules employed by existing decompilers exhibit low coverage rates [4], This which make the decompiler difficult to adapt to varying contract programming styles or patterns.

B. Deficiencies in Decompiler Output

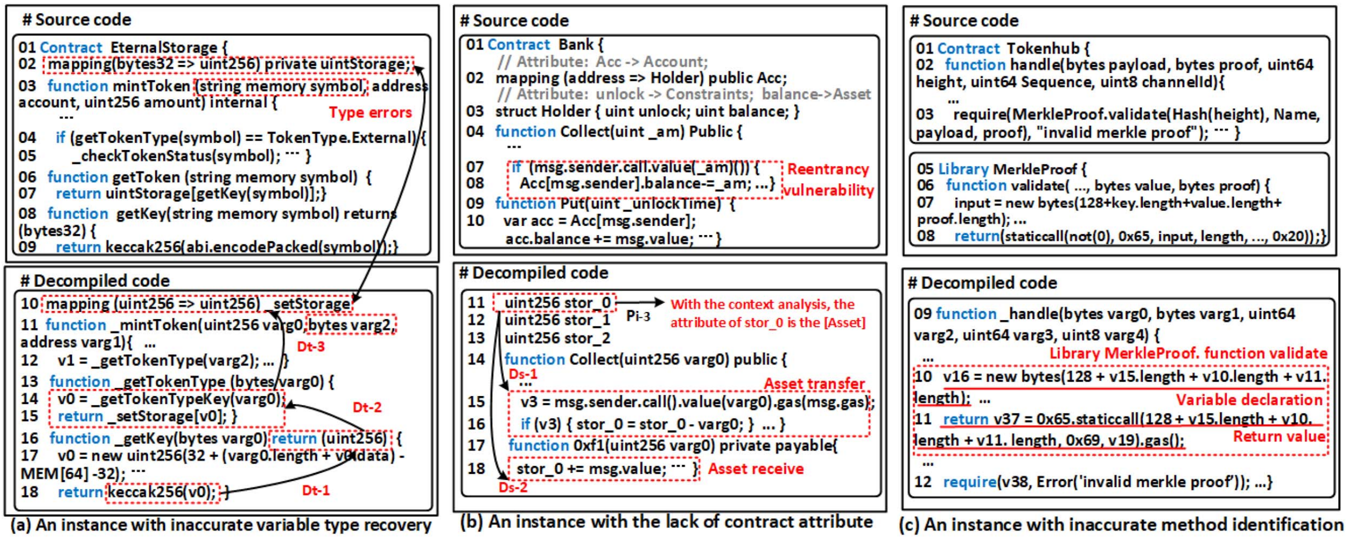
The current decompiler output exhibits certain deficiencies that obstruct effective comprehension and analysis of the target program. Below we take smart contracts of Fig. 1, which contain source code and corresponding decompilation code, as the instances for illustrating the deficiencies.

Deficiency-1. Inaccurate Variable Type Recovery. Limited by the insufficiently generalized heuristic rule, the SOTA decompilers are inaccurate in recovering the variable type. This deficiency is evident in the example shown in Fig. 1(a). In the source code, the state variable *uintStorage* is a mapping type that maps *bytes32* to *uint256* (line 2). However, the decompiler incorrectly assigns it as a mapping *uint256* to *uint256* (line 10). Additionally, while decompilers narrow down the range of possible types for the parameter *varg2* and assign it with *bytes* (line 11), the correct type of this parameter is actually a *string array* (line 3). Obviously, the SOTA decompilers commit errors in type inference. The notable efforts in this domain are SigRec [11], VarLifter [6] and DeepInfer [10]. SigRec and DeepInfer focus primarily on inferring function signatures and recovering parameter types within those signatures. Since these tools concentrate on a subset of variable types, they are insufficient to address the broader deficiency of type recovery (e.g., variable *uintStorage*). VarLifter exhibit low coverage rates, because it heavily relied on pre-defined heuristics according to EVM instructions. In addition, the inaccuracies in variable type recovery result in significant difficulty in identifying specific vulnerabilities, such as overflow vulnerabilities [25].

Deficiency-2. Lack of Contract Attribute. Due to the limited storage of blockchain, smart contract utilizes state variables (i.e., global variables) to record the key contract attributes (e.g., asset, identity, router). While contract attributes are crucial for the vulnerability analysis task, they are exceptionally difficult to be inferred from decompiled code [5]. We take an example in Fig. 1(b) for illustration. Contract *Bank* exhibits a Reentrancy vulnerability because it records the asset balance after transferring assets. An adversary could exploit this by re-entering the contract to transfer assets without recording the change of asset balance (line 7-8). Thus, correctly identifying the state variable *Acc.balance* as an *Asset* attribute is critical for detecting this vulnerability. While the contract attribute is explicitly stated in the source code through meaningful variable names (i.e., *Acc.balance*), it is totally missed at the bytecode level and is labeled as *store_0* in its decompiled code (line 11). Hence, inferring the contract attribute for the variable *store_0* in the decompiled code becomes a formidable challenge. Without the contract attribute, it is difficult to pinpoint the vulnerability. Prior work (i.e., SmartDagger [5]) trains a neural machine learning model using a corpus of 1,200 smart contracts to recover contract attributes for state variables within smart contracts. However, due to limitations in model capacity and the size of the training set, SmartDagger degrade its accuracy of contract attribute prediction when encountering with newly-emerged or rare contracts.

Deficiency-3. Inaccurate function Identification. SOTA decompilers are notably ineffective in identifying functions within smart contracts due to their reliance on low-coverage heuristic rules. For instance, the Gigahorse [8] decompiler identifies functions by detecting call sites that are invoked recurrently. The heuristic rules result in poor performance when encountering complex functions, particularly inherited functions. In contrast to languages like C++ and Java, where inheritance is clearly delineated, smart contract inheritance involves embedding all base subcontracts $(B_1, B_2, \ldots, B_n)$ as code blocks directly into the inheriting contract A, without retaining explicit call information [3]. Hence, the heuristic rules of Gigahorse failed in this case. As can be seen in Fig. 1(c), the inherited function *validate* is explicitly stated in the source code (line 6-8), but it is absent at the bytecode level. Our manual investigation reveals that the *validate* function is integrated into the *handle* function (lines 10-11). Consequently, SOTA decompilers like Gigahorse are unable to reconstruct such inherited functions, which are critical for downstream program analysis tasks such as cross-contract call flow analysis [5], [7] and component analysis [26].

[1] https://github.com/Janelinux/SmartHalo



```
# Source code
01 Contract  EternalStorage {
02  mapping(bytes32 => uint256) private uintStorage;
03  function mintToken (string memory symbol, address
account, uint256 amount) internal {
      ...
04    if (getTokenType(symbol) == TokenType.External){
05       _checkTokenStatus(symbol); ··· }
06  function getToken (string memory symbol)  {
07    return uintStorage[getKey(symbol)];}
08  function  getKey(string memory symbol) returns
(bytes32) {
09    return keccak256(abi.encodePacked(symbol));}

# Decompiled code
10 mapping (uint256 => uint256) _setStorage
11 function _mintToken(uint256 varg0, bytes varg2,
address varg1){  ...
12    v1 = _getTokenType(varg2); ... }
13  function _getTokenType (bytes varg0) {
14    v0 = _getTokenTypeKey(varg0);
15    return _setStorage[v0]; }
16  function _getKey(bytes varg0) return (uint256) {
17    v0 = new uint256(32 + (varg0.length + v0.data) -
MEM[64] -32); ···
18    return keccak256(v0); }
```

(a) An instance with inaccurate variable type recovery

```
# Source code
01 Contract  Bank {
     // Attribute:  Acc -> Account;
02  mapping (address => Holder) public Acc;
     // Attribute:  unlock -> Constraints;  balance->Asset
03  struct Holder { uint unlock; uint balance; }
04  function Collect(uint _am) Public {
      ...
07    if  (msg.sender.call.value(_am)()) {
08      Acc[msg.sender].balance-=_am; ...}
09  function Put(uint _unlockTime)  {
10    var acc = Acc[msg.sender];
      acc.balance += msg.value; ··· }

# Decompiled code
11   uint256 stor_0
12   uint256 stor_1
13   uint256 stor_2
14  function Collect(uint256 varg0) public {
     ...
15    v3 = msg.sender.call().value(varg0).gas(msg.gas);
16    if (v3) { stor_0 = stor_0 - varg0; }  ... }
17   function 0xf1(uint256 varg0) private payable{
18    stor_0 += msg.value; ··· }
```

Reentrancy vulnerability
Pi-3
With the context analysis, the attribute of stor_0 is the [Asset]
Ds-1
Asset transfer
Asset receive
Ds-2

(b) An instance with the lack of contract attribute

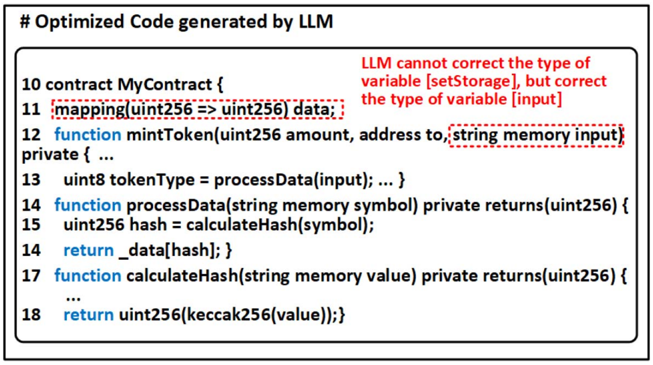
```
# Source code
01 Contract  Tokenhub {
02  function handle(bytes payload, bytes proof, uint64
height, uint64 Sequence, uint8 channelId){
      ...
03     require(MerkleProof.validate(Hash(height), Name,
payload, proof), "invalid merkle proof"); ··· }

05 Library MerkleProof {
06  function validate( ..., bytes value, bytes proof) {
07     input = new bytes(128+key.length+value.length+
proof.length); ...
08     return(staticcall(not(0), 0x65, input, length, ..., 0x20));}

# Decompiled code
09 function _handle(bytes varg0, bytes varg1, uint64
varg2, uint64 varg3, uint8 varg4) {
  ...
10  v16 = new bytes(128 + v15.length + v10.length + v11.
length);  ...
11  return v37 = 0x65.staticcall(128 + v15.length + v10.
length + v11. length, 0x69, v19).gas();
  ...
12  require(v38, Error('invalid merkle proof')); ...}
```

(c) An instance with inaccurate method identification

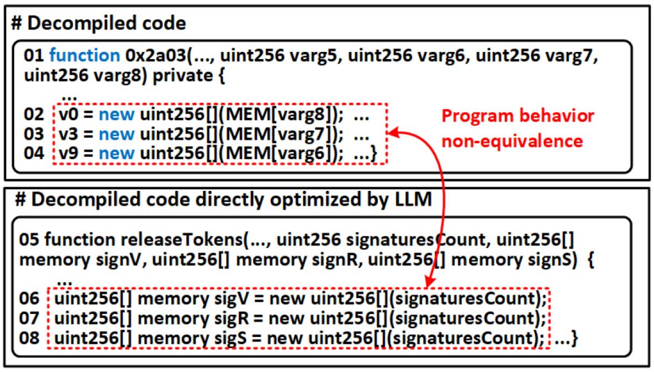
Fig. 1. Three motivating examples for illustrating the limitations of current decompiler output.



```
# Optimized Code generated by LLM
10 contract MyContract {
11  mapping(uint256 => uint256) data;
12  function mintToken(uint256 amount, address to, string memory input)
private { ...
13    uint8 tokenType = processData(input); ... }
14  function processData(string memory symbol) private returns(uint256) {
15    uint256 hash = calculateHash(symbol);
14    return _data[hash]; }
17  function calculateHash(string memory value) private returns(uint256) {
      ...
18    return uint256(keccak256(value));}
```

LLM cannot correct the type of variable [setStorage], but correct the type of variable [input]

Fig. 2. The decompiled code optimized by LLM for the instance in Fig. 1(b).



```
# Decompiled code
01 function 0x2a03(..., uint256 varg5, uint256 varg6, uint256 varg7,
uint256 varg8) private {
     ...
02  v0 = new uint256[](MEM[varg8]);  ...
03  v3 = new uint256[](MEM[varg7]);  ...
04  v9 = new uint256[](MEM[varg6]);  ...}

# Decompiled code directly optimized by LLM
05 function releaseTokens(..., uint256 signaturesCount, uint256[]
memory signV, uint256[] memory signR, uint256[] memory signS)  {
     ...
06  uint256[] memory sigV = new uint256[](signaturesCount);
07  uint256[] memory sigR = new uint256[](signaturesCount);
08  uint256[] memory sigS = new uint256[](signaturesCount); ...}
```

Fig. 3. An optimization error reported by LLM inference in terms of program-behavior non-equivalence.

C. Our Solution

We propose our strategies for optimizing the decompiler output, combining with a discussion on potential solutions for the deficiencies presented in motivating examples of Fig. 1.

Strategy-1: Using SA to extract control-flow dependency, type dependency and state dependency. First, type dependency is crucial for the variable type recovery. For example, the type error for variable *uintStorage* can be corrected by using SA to infer type dependencies, as illustrated in Fig. 1(a). Initially, the output of the predefined function *keccak256* is of type *bytes32* (D_{t_1}). Subsequently, the type of variable $v0$ depends on the return value type of the function *_getTokenKey*, which is also *bytes32* (D_{t_2}). Finally, the key type of the variable *setStorage* depends on the type of $v0$ (D_{t_3}). Consequently, the type of *setStorage* should be corrected to *mapping(bytes32 = uint256)*. Second, control-flow dependency is critical for the function boundary identification. For instance, as shown in Fig. 1(c), while static constraints (i.e., call information) are unavailable in function *validate*, we can manually infer its boundaries by analyzing the contextual control flow, i.e., the function typically declares a new variable at the beginning (line 10) and returns the value at the end (line 11). Third, state dependency is important for contract attribute inference. We take the motivating example in Fig. 1(b) as an instance for illustration. By analyzing the dependency of the state variable, we can find that state variable *stor_0* is written in line 16 (D_{s_1}) and line 18 (D_{s_2}). By manually analyzing their context usage, we can infer that they are utilized for asset transfer and asset receive, respectively. Hence, we can infer the attribute for *stor_0* as the *[Asset]*.

Strategy-2: Empowering LLMs to enrich semantics. With the remarkable capability of few-shot in-context learning [27], [28], LLMs are flexible to predicting the optimization target (i.e., variables or functions), which are rarely encountered in the program and often incapable of inferring them based on other variables or functions. We take Fig. 1(a) again as an instance for illustration. The type error for variable *input* can be corrected by using LLMs for prediction. Fig. 2 presents code optimized by LLM. Benefiting from the extensive training set and model capabilities, LLM can assign the variable *input* with *string memory* correctly (line 12). In addition, owing

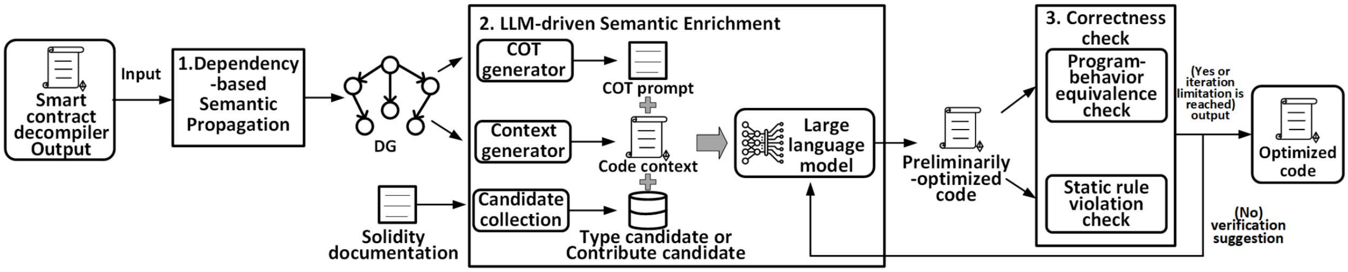

Smart contract decompiler Output
Input
1.Dependency -based Semantic Propagation
DG
Solidity documentation
2. LLM-driven Semantic Enrichment
COT generator
COT prompt
Context generator
Code context
Candidate collection
Type candidate or Contribute candidate
Large language model
Preliminarily -optimized code
3. Correctness check
Program-behavior equivalence check
Static rule violation check
(Yes or iteration limitation is reached) output
(No) verification suggestion
Optimized code

Fig. 4. The overview of SmartHalo.

to its remarkable model capacity, LLMs are more effective and generalizable in learning static domain knowledge (i.e., dependencies) for optimization predictions, compared to SA's manually crafting heuristic rules based on the same knowledge [29]. For instances, instead of manually summarizing heuristic rules, we can utilize LLMs to learning the control-flow context for function boundary prediction, and to analyze the contextual usage of state variables to infer their contract attributes.

Inspired by the above analysis, our objective is to harness the advantages of both SA and LLMs to optimize decompiler output. Therefore, SmartHalo extracts three types of program dependencies including type dependency, state dependency and control flow dependency within the program through SA. Subsequently, SmartHalo leverages the superior generalization capabilities of LLMs to optimize the decompiled code after learning these program dependencies.

III. Design of SmartHalo

To overcome the deficiencies, we propose a novel decompiler output optimization framework, called SmartHalo, based on hybrid SA and LLM. SmartHalo takes the decompiler output as input, and finally outputs the optimized code. Fig. 4 shows the overview of SmartHalo, which integrates the following mechanisms to ensure its effectiveness.

Dependency-based Semantic Extraction. In this mechanism, SmartHalo aims to extract critical semantics from the decompiled code through a fine-grain dependency analysis, which is then propagated into the LLM for further processing. To this end, our work complements and extends previous research [5], [6], [10], [11] by considering two new types of dependencies: namely, state dependency and type dependency. Specifically, SmartHalo analyzes three types of dependencies as the crucial program semantics, subsequently utilizing them to construct a Dependency Graph (DG). The extracted semantics (i.e., dependencies) represent the critical static domain knowledge necessary for decompiler output optimization, which facilitates the LLMs in capturing them across the entire smart contract (See Section IV-A).

LLM-driven Semantic Enrichment. With the semantic representation (i.e., DG) propagated by the above mechanism, SmartHalo constructs inference prompts to guide LLM to enrich the semantic of decompiler output. Specifically, according to the DG, SmartHalo utilizes our designed generator to constructs the context for optimization targets (i.e., functions or variables), for determining the code range related to optimization targets. Subsequently, SmartHalo further generates chain-of-thoughts that represent inference steps of static analysis for optimization targets, based on the dependencies among the DG, for teaching LLMs how to infer variable types, contract attributes and function boundaries. In addition, SmartHalo collects the candidates for variable types and contract attributes from the Solidity documentation [3]. Lastly, LLM learns from the above three types of prompt, and optimizes variable types, contract attributes and function boundaries for decompiled code (See Section IV-B).

Correctness Verification. However, verifying correctness for the optimized code is far from straightforward. First, due to the hallucination that commonly exist in LLMs, LLMs generates the random and unpredictable outputs that unexpectedly alter the program behaviors of the original decompiled code. For example, Fig. 3 illustrates a program behavior non-equivalence error caused by LLM optimization. In the original decompiled code, variables *v0*, *v3* and *v9* depends on parameters *varg6*, *varg7* and *varg8*, respectively (line 02-04). However, in the optimized code, variables *sigV*, *sigR* and *sigS* incorrectly depend on the same parameter *signaturesCount* (line 06-08). Second, LLMs also introduce errors that contradict common static domain rules (e.g., type rules), further complicating correctness verification. We take Fig. 2 again as an instance for illustrating such type of errors. In line 18, the value returned by the predefined function *keccak256* is type of *bytes32*. According to Solidity documentation [3], *bytes32* cannot be directly converted into *uint256*, result in convertion errors.

SmartHalo implements two specific checks for establishing correctness verification. In the program-behavior equivalence check, SmartHalo utilizes symbolic execution and formal verification to verify the program-behavior equivalence between the original decompiled code and optimized code. In the static rule violation check, SmartHalo summarizes a set of static rules that are related to types. Note that static rule violation check donot consider function boundaries and contract attributes because there are no explicit static rules related to them in smart contract [3]. By applying these static rules, SmartHalo rejects the inference errors. (see Section IV-C)

IV. Approach Details

A. Dependency-Based Semantic Extraction

Basic control-flow analysis. To enable control-flow analysis at the level of decompiled code, we develop a Solidity-oriented control-flow analysis component for SmartHalo, which

$$\begin{aligned} e \in \mathit{Expr} ::= & v \mid c \mid e \ \mathit{blop} \ e \mid e \ \mathit{numop} \ e \mid e \ \mathit{cmpop} \ e \mid \\ & e \ \mathit{bitop} \ e \mid (e, \ldots, e) \mid [e, \ldots, e] \mid e(e, \ldots, e) \mid \\ & e[e] \mid e[e:\ e] \mid e.v \end{aligned}$$

Fig. 5. The syntax of expression for typing in solidity.

provides both code parsing and control/data-flow graph construction. Built on a Tree-sitter architecture, the component performs analysis without requiring compilation, which eliminates environment configuration and build overhead. Such property makes the component particularly suitable for processing decompiled pseudocode. Specifically, the control-flow analysis component (i) employs Tree-sitter [30] to construct a concrete syntax tree for each decompiled code file, (ii) transforms the syntax tree into a three-address intermediate representation (IR), and (iii) leverages this IR together with function-call information to produce the corresponding control-flow and data-flow graphs.

Contract Dependency Identification. SmartHalo focus on identifying dependencies including the type dependency, state dependency, and control-flow dependency.

- *Type dependency.* Type dependency refers to the dependency that the type of variable v_a correlates to another variable v_b or a specific expression e_b. Fig. 5 shows the syntax of all the expressions that generate types in smart contracts. Given the decompiled code, SmartHalo identifies the type dependency according to these syntaxes effectively.
- *State dependency.* State dependency refers to the dependency between different state variables or dependency between state variables and expressions, such as read and write dependency on state variables. SmartHalo utilize SmartState [31], a SOTA state dependency analyzer to identify state dependency from given smart contracts.

Dependency Graph Construction. Subsequently, SmartHalo proposes the dependency graph (DG) as the representation of critical semantics for smart contracts.

The DG constructed by SmartHalo can be represented as a tripe $G_c = (N_c, E_c, X_e)$. Specifically, SmartHalo encodes the following information: (1) the nodes of DG can be denoted as a set of all variables and related expressions in the decompiled code. Here, we use N_v to represent the variable nodes, and leverage N_e to represent the expression nodes. Hence, we have $N_c := \{N_v \cup N_e\}$; (2) the edges of DG can be represented as a set of control-flow dependency edges E_d, state dependency edges E_s and type dependency edges E_t. Similarly, we have $E_c := \{E_d \cup E_s \cup E_t\}$. (3) $X(E_c) \rightarrow \{DFD, SD, TD\}$ is a labeling function that maps an edge to one of the above three dependencies.

After generating control-flow graph, SmartHalo constructs the DG by incrementally adding state dependency edges and type dependency edges to the control-flow graph. Specifically, firstly, SmartHalo searches the whole control-flow graph to locate the variables and expressions that are related to type dependency. Further, SmartHalo utilizes SmartState to identify the variables and expressions that are related to state dependency. Finally, SmartHalo finds out the sources and targets among all these variables and expressions, and uses the direct edge to connect them in pair.

B. LLM-Driven Semantic Enrichment

In this subsection, SmartHalo first generates domain-aware inference prompts according to the DG. Then, SmartHalo utilizes the inference prompts to guide the LLM to optimize the decompiler output. SmartHalo considers three types of inference prompts, including code contexts, inference candidates, and Chain-of-Thought (COT) prompts.

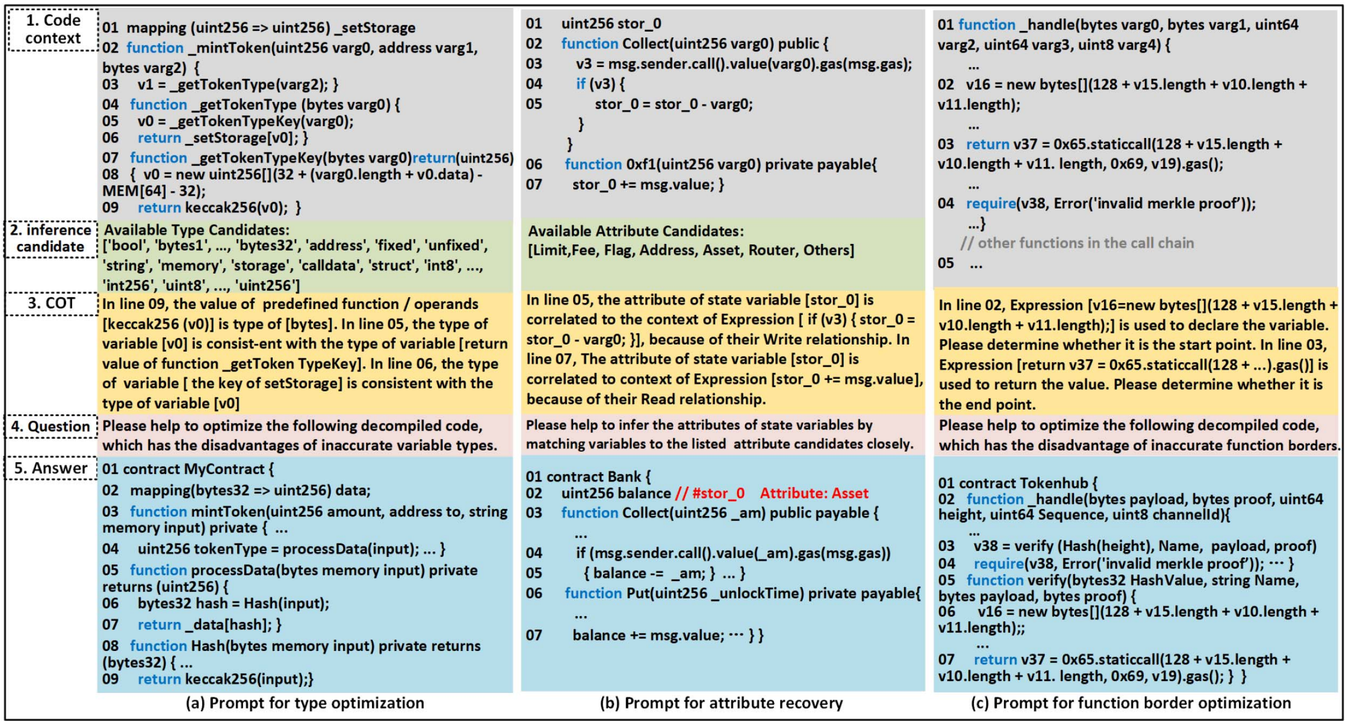
Code Context. SmartHalo adapts different strategies to generate code snippets for variables and functions. The strategy for variables applies to optimizing variable type and contract attribute, and another strategy for functions is used to optimize function boundaries.

- *Variables.* To pinpoint the context related to target variables precisely, SmartHalo utilizes code slicing to identify code context fragments from the decompiled code for target variables, based on the DG. To this end, SmartHalo first generates the slicing DG based on the original DG. On the original DG, SmartHalo starts from target variable nodes and performs the forward and backward traversals (i.e., the same or opposite direction to DG edges). In this way, SmartHalo finds out all the nodes that have the dependency on target variables and generates a slicing subgraph of DG. With the slicing DG, SmartHalo combines the corresponding expressions together to generate a code snippet as the context prompt.
- *functions.* To generate the context prompt for target functions, SmartHalo searches the DG to find out the call chains that target functions lie in. Then, SmartHalo combines the corresponding functions in these call chains together to generate a code snippet as the context prompt.

In this way, SmartHalo narrows down the original decompiled code to the much smaller code snippet, which contains the information only related to decompiler output optimization of target variables or functions. We present the code context generation for the instance of Fig. 1, as highlighted in the gray part of Fig. 6. For type optimization, SmartHalo identifies all the expressions that have type dependency related to variable *setStorage*, and integrates them as the code context prompt. For attribute recovery, SmartHalo extracts the corresponding expressions that have state dependency on state variable *stor_1*, and aggregates them as the code context prompt. Finally, SmartHalo extract the functions from the call chain that function *_handle* lies in to form the code context prompt.

Inference Candidate. Subsequently, SmartHalo prompts the LLM with the candidates of variable type and contract attribute. In this way, SmartHalo can help the LLM narrow down the selection range to improve the accuracy.

Variable type candidates. Solidity is a statically typed language that provides a specific number of built-in types. In addition, the user-defined value type in Solidity is actually an alias that creates a zero cost abstraction over an elementary value type, so SmartHalo directly utilizes the elementary value type to represent it. According to Solidity documentation [3],



1. Code context

```
01  mapping (uint256 => uint256) _setStorage
02  function _mintToken(uint256 varg0, address varg1,
bytes varg2)  {
03     v1 = _getTokenType(varg2); }
04   function _getTokenType (bytes varg0) {
05     v0 = _getTokenTypeKey(varg0);
06     return _setStorage[v0]; }
07   function _getTokenTypeKey(bytes varg0)return(uint256)
08   {  v0 = new uint256[](32 + (varg0.length + v0.data) -
MEM[64] - 32);
09     return keccak256(v0);  }
```

```
01    uint256 stor_0
02    function Collect(uint256 varg0) public {
03       v3 = msg.sender.call().value(varg0).gas(msg.gas);
04       if (v3) {
05           stor_0 = stor_0 - varg0;
         }
       }
06     function 0xf1(uint256 varg0) private payable{
07      stor_0 += msg.value; }
```

```
01 function _handle(bytes varg0, bytes varg1, uint64
varg2, uint64 varg3, uint8 varg4) {
      ...
02   v16 = new bytes[](128 + v15.length + v10.length +
v11.length);
      ...
03   return v37 = 0x65.staticcall(128 + v15.length +
v10.length + v11. length, 0x69, v19).gas();
      ...
04   require(v38, Error('invalid merkle proof'));
      ...}
    // other functions in the call chain
05   ...
```

2. inference candidate

Available Type Candidates:
['bool', 'bytes1', ..., 'bytes32', 'address', 'fixed', 'unfixed', 'string', 'memory', 'storage', 'calldata', 'struct', 'int8', ..., 'int256', 'uint8', ..., 'uint256']

Available Attribute Candidates:
[Limit,Fee, Flag, Address, Asset, Router, Others]

3. COT

In line 09, the value of predefined function / operands [keccak256 (v0)] is type of [bytes]. In line 05, the type of variable [v0] is consist-ent with the type of variable [return value of function _getToken TypeKey]. In line 06, the type of variable [the key of setStorage] is consistent with the type of variable [v0]

In line 05, the attribute of state variable [stor_0] is correlated to the context of Expression [if (v3) { stor_0 = stor_0 - varg0; }], because of their Write relationship. In line 07, The attribute of state variable [stor_0] is correlated to context of Expression [stor_0 += msg.value], because of their Read relationship.

In line 02, Expression [v16=new bytes[](128 + v15.length + v10.length + v11.length);] is used to declare the variable. Please determine whether it is the start point. In line 03, Expression [return v37 = 0x65.staticcall(128 + ...).gas()] is used to return the value. Please determine whether it is the end point.

4. Question

Please help to optimize the following decompiled code, which has the disadvantages of inaccurate variable types.

Please help to infer the attributes of state variables by matching variables to the listed attribute candidates closely.

Please help to optimize the following decompiled code, which has the disadvantage of inaccurate function borders.

5. Answer

```
01 contract MyContract {
02   mapping(bytes32 => uint256) data;
03   function mintToken(uint256 amount, address to, string
memory input) private {  ...
04     uint256 tokenType = processData(input); ... }
05   function processData(bytes memory input) private
returns (uint256) {
06     bytes32 hash = Hash(input);
07     return _data[hash]; }
08   function Hash(bytes memory input) private returns
(bytes32) { ...
09     return keccak256(input);}
```

```
01 contract Bank {
02    uint256 balance // #stor_0    Attribute: Asset
03    function Collect(uint256 _am) public payable {
        ...
04       if (msg.sender.call().value(_am).gas(msg.gas))
05         { balance -=  _am; }  ... }
06     function Put(uint256 _unlockTime) private payable{
        ...
07      balance += msg.value; ··· } }
```

```
01 contract Tokenhub {
02   function _handle(bytes payload, bytes proof, uint64
height, uint64 Sequence, uint8 channelId){
      ...
03    v38 = verify (Hash(height), Name,  payload, proof)
04    require(v38, Error('invalid merkle proof')); ··· }
05   function verify(bytes32 HashValue, string Name,
bytes payload, bytes proof) {
06     v16 = new bytes[](128 + v15.length + v10.length +
v11.length);;
       ...
07    return v37 = 0x65.staticcall(128 + v15.length +
v10.length + v11. length, 0x69, v19).gas(); }  }
```

(a) Prompt for type optimization

(b) Prompt for attribute recovery

(c) Prompt for function border optimization

Fig. 6. The prompt for the instance in Fig. 1.

we collect the variable type candidates, as can be seen in the inference candidates (highlighted in green part) of Fig. 6(a).

Contract attribute candidates. In contrast to variable type, contract attribute is far from straightforward and should be summarized by representative smart contract dataset. We summarize the contract attribute candidates as a set, as can be seen in the inference candidates (highlighted in green part) of Fig. 6(b). And the detail of summarizing process are illustrated in Section V-A.

Chain-of-Thought Prompt. The COT prompts are a series of intermediate reasoning steps in static analysis. SmartHalo prompts the LLM with COT Prompt, for teaching LLM how to reason the correct function boundaries, variable type and contract attribute from the perspective of static domain knowledge.

SmartHalo utilizes the slicing DG of the code slicing phase to generate the COT prompts. Given the slicing DG, SmartHalo organizes the nodes according to the distance between the nodes and the target variable node (i.e., hop counts). Specifically, the hop count of the target variable node is set as 0, and the hop counts of other nodes are calculated by their distance. SmartHalo translates each hop in the DG into a sentence of dependency description. Inspired by prior work [29], we summarize the effective COT prompt templates, as shown in Table I. As can be seen, each optimization tasks involve at least one kind of COT prompt template. Specifically, variable type optimization relies on the categories of both type dependency and control-flow dependency. Further, variable type optimization depends on the categories of both static dependency and control-flow dependency. And function boundary optimization only requires the category of control-flow dependency. To generate the COT prompt, SmartHalo fills the corresponding variable [TYPE], variable [NAME], [USAGE] and [STATEMENT] into the templates.

To illustrate the COT prompt generation, we present the COT prompt for the instance of Fig. 1, as highlighted in yellow part of Fig. 6. For type optimization, as discussed above, there are three type dependency edges (i.e., D_{t_1}, D_{t_2}, D_{t_3}) for the instance of Fig. 1(a). As shown in Fig. 6(a), SmartHalo utilizes $Expression \rightarrow Variable$ template to describe D_{t_1}, and leverages $Variable \rightarrow Variable$ template to describe D_{t_2} and D_{t_3}. The sentences of all the corresponding edges are connected together according to the order of type dependency, to form the COT prompt for type optimization.

For contract attribute recovery, as mentioned above, there are two dependency edges (i.e., D_{s_1}, D_{s_2}) for the instance of Fig. 1(b). As shown in Fig. 6(b), SmartHalo utilizes $Statevariable \rightarrow Expression$ template to describe D_{s_1} and and D_{s_2}. The sentences of these edges are aggregated together as the COT prompt for attribute recovery.

Fig. 6(c) shows the COT prompt for function boundary optimization on the instance of Fig. 1(c). SmartHalo utilizes $Return - Value$ and $Variable - Declaration$ templates to describe statements that are likely to be the start and end points of a function.

C. Correctness Verification

Program-behavior Equivalence Check. SmartHalo aims to check the program-behavior equivalence between functions of original decompiled code and functions of optimized decompiled code. We use m and m' to represent the two versions of the

TABLE I
CHAIN-OF-THOUGHT PROMPT TEMPLATE FOR LLM-DRIVEN SEMANTIC ENRICHMENT

Category	Type	Template
Type dependency	Variable → Variable	The type of variable [NAME] is consistent with the type of variable [NAME]
	Type → Expression	The value of predefined function/operands[NAME] of expression [STATEMENT] is type of [TYPE]
	Type → Variable	The variable of [NAME] is assigned from [TYPE]
	Expression → Variable	The type of variable [NAME] depends on expression [STATEMENT]
	Variable → Expression	The operand(s)/target(s)/key(s)/value(s) of expression [STATEMENT] is/are [TYPE].
State dependency	State variable → State variable	The attribute of state variable [NAME] is correlated to the attribute of state variable [NAME]
	State variable → Expression	The attribute of state variable [NAME] is correlated to the context usages of Expression [STATEMENT] (Write)
	Expression → State variable	The attribute of state variable [NAME] is correlated to the context usages of Expression [STATEMENT] (Read)
Control flow dependency	Call Site	Expression [STATEMENT] seems to be a call site to another function.
	Modifier	Expressions with a range from [STATEMENT] to [STATEMENT] seems to be a modifier function.
	Return Value	Expression [STATEMENT] is used to return the value. Please determine whether it is the end point.
	Variable Declaration	Expression [STATEMENT] is used to declare the variable. Please determine whether it is the start point.

Note: [NAME] indicates the names of variable nodes, [STATEMENT] indicates the statements of expression nodes, [TYPE] indicates the types of variable nodes. [USAGES] indicates the usages for the variable nodes.

TABLE II
DISTRIBUTION AND CHARACTERISTIC OF THE EVALUATION DATASET

Metric	Evaluation dataset
Average lines	163
Functionalities	Cross-Chain interoperability, Market Mechanisms & Price Discovery, Governance, Risk Management & Liquidation, Security & Consensus, Token Management, Value Transfer & Incentive Distribution, Entertainment & Probabilistic Gamebling
Solidity	From 0.4.22 to 0.8.25
Deployment	From April 2018 to April 2024

same function in the original decompiled code and optimized decompiled code.

SmartHalo conducts the symbolic execution on functions m and m' for generating symbolic summary s and s'. If functions m and m' are equivalent in terms of program behaviors, their symbolic summary s and s' are also equivalent in terms of functionality. An equivalence assertion is a first-order logic formula Φ that is used to assert such equivalence [32].

$$\Phi = \neg(s \Leftrightarrow s^{'}) \tag{1}$$

SmartHalo utilize formal verification method to analyze symbolic summaries s and s', and determine whether equivalence assertion Φ is satisfied. If Φ is satisfied, function m is non-equivalent to function m' in terms of program behaviors, otherwise, they are equivalent to each other. Further, SmartHalo integrates the formal verification method that relies on SMT solver (i.e., Z3 solver [33]), to find out satisfying assignments for equivalence assertion Φ.

Note that since the symbolic summaries generally contain non-linear constraints, it is difficult for the Z3 solver to solve them. Such non-linear constraints commonly result in the solving failure. We eliminate most of the solving failure by utilizing methods proposed in ARDiff [34].

Rule-based Type Violation Check. SmartHalo traverse throughout the optimized code, then leverages violation rejection rules to identify and reject incorrect variable types predicted by the LLM.

Here, we summarize the violation rejection rules integrated by SmartHalo in Fig. 7. Each violation rejection rule is composed of two parts, including specific premises (contents above the line) and conclusions (contents below the line). They are organized as follows.

$$\pi \vdash e : \theta \tag{2}$$

where π represents the context that contains lists that assign variable types to expression patterns. In this form, e refers to the expression, and we use $e_1, \ldots, e_n$ to represent different expressions. θ is the variable types. We use $\theta_1, \ldots, \theta_n$ to represent different variable types. A rule in this form is called a judgment or assignment. Our goal is to get the context π that assigns variable types to all the variables in code.

Finally, the incorrect optimization results predicted by LLM would be integrated with verification suggestions to form the violation information. SmartHalo retransmits the violation information into LLM to infer the correct function boundaries, variable types, and contract attributes anew. Further, the interaction can be iterated continually between LLM-driven semantic enrichment and correctness verification, until SmartHalo amends all the errors identified by correctness verification, or a maximum iteration limit is reached.

V. EVALUATION

Research Questions. We summarize the following research questions for evaluating SmartHalo.

RQ1. How effective is SmartHalo in terms of decompiler output optimization?

RQ2. How does SmartHalo perform compared to other state-of-the-art mechanisms?

RQ3. How effective are individual components of SmartHalo in terms of helping SmartHalo improve the precision of decompiler output optimization?

RQ4. Can SmartHalo generalize to real-world complex contracts, different decompilers or different LLMs?

RQ5. What is the efficiency of SmartHalo in terms of both runtime and monetary costs?

RQ6. How effective is the output of SmartHalo in terms of helping downstream program analysis task?

$$\frac{}{\pi \vdash c : \theta} \tag{Constant}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad \pi \vdash e_2 : \theta_2, \quad \widetilde{\theta} = \{bool, int\}}{\pi \vdash e_1[bitop]e_2 : \theta \wedge \widetilde{\theta}, \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta}, \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}} \tag{LShift, RShift}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad \widetilde{\theta} = \{bool, int\}, \quad \pi \vdash e_2 : \theta_2, \quad \theta' = getMorePreciseType(\theta_1 \wedge \widetilde{\theta}, \quad \theta_2 \wedge \widetilde{\theta})}{\pi \vdash e_1[numop]e_2 : \theta', \quad \pi \vdash \theta_1 \wedge \widetilde{\theta}, \pi \vdash \theta_2 \wedge \widetilde{\theta}} \tag{Numeric Operations}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad \pi \vdash e_2 : \theta_2, \quad \theta' = \{\Gamma, Array, Tuple\}}{\pi \vdash e_1[cmpop]e_2 : bool, \quad \pi \vdash e_1 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta}, \quad \pi \vdash e_2 : \theta_1 \wedge \theta_2 \wedge \widetilde{\theta}} \tag{Lt, LtE, Gt, GtE}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad ..., \quad \pi \vdash e_n : \theta_n}{\pi \vdash (e_1, ..., e_n) : Tuple[\theta_1, ..., \theta_n], \quad \pi \vdash (e_1, ..., e_n) : Array[\theta_1, ..., \theta_n]} \tag{Tuple, Array}$$

$$\frac{\pi \vdash e : \theta, \quad \widetilde{\theta} = \{str, bytes\}, \quad \theta' = getElementType(\theta_1 \wedge \widetilde{\theta})}{\pi \vdash [for]v[in]e : \theta', \quad \pi \vdash e : \theta \wedge \widetilde{\theta}} \tag{Comprehension}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad \pi \vdash e_2 : \theta_2}{\pi \vdash e_1[blop]e_2 : \theta', \quad Union[\theta_1, \theta_2]} \tag{Boolean Operation}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad \pi \vdash e_2 : \theta_2, \quad \widetilde{\theta} = \{bool, int, byte\}}{\pi \vdash e_1[bitop]e_2 : \theta \wedge \widetilde{\theta}, \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta}, \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta}} \tag{Bitor, BitAnd, BitXor}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad \pi \vdash e_2 : \theta_2}{\pi \vdash e_1[cmpop]e_2 : bool} \tag{Eq,NotEq,Is,IsNot}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad \pi \vdash e_2 : \theta_2, ... \quad \pi \vdash e_n : \theta_n, \quad \widetilde{\theta} = \{Callable[[\theta_1, \theta_2, ..., \theta_n], \theta]\}, \quad \theta' = getReturnType(\theta \wedge \widetilde{\theta})}{\pi \vdash e(e_1, ..., e_n) : \theta} \tag{Call}$$

$$\frac{\pi \vdash e_1 : \theta_1, \quad \pi \vdash e_2 : \theta_2, \quad \widetilde{\theta_1} = \{str, bytes\}, \quad \widetilde{\theta_2} = \{int, bool\}, \quad \theta' = getElementType(\theta_1 \wedge \theta_2)}{\pi \vdash e_1[e_2] : \theta', \quad \pi \vdash e_1 : \theta_1 \wedge \widetilde{\theta_1}, \quad \pi \vdash e_2 : \theta_2 \wedge \widetilde{\theta_2}} \tag{Slice}$$

Fig. 7. The static rules for type violation check.

A. Implementation and Evaluation Setup

Implementation. SmartHalo was implemented with around 1,799 lines of code in Python 3.8.10. We utilize ChatGPT (GPT-3.5) to support the LLM component of SmartHalo, due to the significantly expensive overhead of GPT fee. Note that SmartHalo is designed to be model-agnostic and unlimited to a certain large language model, it offers the flexibility to integrate alternative large language models. All evaluation experiments were conducted on an Ubuntu 20.04 server equipped with an Intel i9-10980XE CPU (3.0GHz), an RTX 3090 GPU, and 250GB of RAM.

Evaluation Dataset and Ground-truth Establishment. Our evaluation dataset derives from a largest open-source smart contract dataset [16] as of Apirl 2024, totaling 963,151 smart contracts. From this original dataset, we randomly sampled 500 functions to construct the evaluation dataset. Ultimately, we filter out the functions the decompiler cannot handle, and obtain 456 pairs of source code and decompiler outputs of functions, for forming the final evaluation dataset. *Noting that evaluation dataset size is similar to those used in SOTA studies [4], [17]*. Here, we utilize function pairs instead of entire contracts because GPT integrated by SmartHalo requires shorter code snippets as input.

Furthermore, we established the ground truth for the evaluation dataset. Specifically, the ground truth for function boundaries and variable types was determined by directly comparing the decompiled code with the source code, as these aspects are explicitly stated in the source code. In contrast, the ground truth for contract attributes was established through manual summarization by our domain researchers. To mitigate bias, the manual analysis procedure was conducted in a rigorous manner. The process consists of three steps. In the first step, six researchers with at least two years of relevant domain experience were invited and organized into three pairs. Among these, two pairs were designated as annotators, while the remaining pair, composed of researchers with at least four years of experience, served as referee experts. In the second step, each annotator within a pair independently conducts the manual investigation. The third step is cross-validation and quality control. For each pair of annotators, they perform the cross-validation and discuss the labels until they obtain a consensus on the classification results. If consensus cannot be reached, the referee experts are consulted to provide the final decision. In addition, we calculate the average Cohen's Kappa coefficient to measure the consistency of labeling results, which is 0.772, thus indicating a substantial agreement between each pair of researchers.

As mentioned earlier, contract attributes refer to the semantic meaning represented by state variables (e.g., asset, address). To ensure comprehensive coverage of contract attribute categories, we also randomly sampled an additional 1,000 smart contracts from the original dataset and used a Large Language Model (LLM) to analyze the usage patterns of state variables within them. With the usage patterns, our experts manually summarize

TABLE III
Comparison Results Between the Optimized Code Generated by SmartHalo and Original Decompiler Output Produced By the Dedaub Decompiler, For Evaluate the Overall Effecctiveness of SmartHalo

Metrics	Function Boundary					Variable Type					Contract Attribute					Recompilation Failure
	TP	FP	FN	Prec.	Recall	TP	FP	FN	Prec.	Recall	TP	FP	FN	Prec.	Recall	Rate
Original decompiler	316	149	310	67.96%	50.48%	676	410	975	62.25%	42.22%	N/A					100%
SmartHalo	504	67	122	88.26%	80.51%	1349	113	252	92.27%	84.26%	752	350	75	68.06%	90.93%	39.78%

“N/A” : the corresponding tool does not support optimizing this deficiency.

the representative categories as [*Limit, Fee, Flag, Address, Asset, Router, Others*].

Evaluation metrics. We utilize four metrics for evaluation:

- *function boundary match.* We confirm the boundary prediction is correct for a given function, by manually investigating whether the starting and ending points are totally match those of sourcecode-level function.
- *Type match.* Similar to DIRTY[15], We consider a type prediction to be correct only if the predicted type fully matches the ground truth type, including data layout, and the type and name of any fields if applicable.
- *Contract attribute match.* Similar to type match, we determine a attribute prediction to be correct only if the predicted attribute fully matches the ground truth attribute.
- *Recompilation failures.* It represents errors yielded by the compiler when recompiling decompiled outputs. This indicates bugs or limitation in decompiler output [35].

B. Overall Effectiveness

To address RQ1, we evaluate the overall effectiveness of SmartHalo by comparing the optimized code generated by SmartHalo with the original decompiler output produced by the Dedaub decompiler (i.e., the web version of Gigahorse [8]) using four key metrics. To achieve this, we run SmartHalo on the evaluation dataset. Additionally, we conduct a detailed analysis of the reasons behind optimization failures. As can be seen in Table III, the optimized code generated by SmartHalo significantly improves all four metrics compared to the original decompiler output. For example, the precision and recall of function boundary identification are enhanced by 20.30% and 30.03%, and the precision and recall of variable type inference are increased by 30.02% and 42.04%. Moreover, while the original decompiler output lacks any contract attribute information, SmartHalo successfully recovers the contract attributes with a precision of 68.06% and a recall of 90.93%. Additionally, unlike the original decompiler output, which is not suitable for recompilation, 60.22% of the optimized code generated by SmartHalo can be recompiled using a compiler.

Optimization failures. We manually inspect all the optimization failures (i.e., false positives and false negatives) of the four metrics. Firstly, we leverage the open card sorting approach similar to prior research for the error taxonomy construction. We found that SmartHalo struggles when encountering the following types of functions or contracts: E1. low-level and delegated calls, E2. inline assembly, E3. off-chain reliance, E4. inheritance structure or E5. others. Then, we compute the rates for each type of optimization failure in the above-stated taxonomy. Lastly, we conduct the root-cause mapping between the SmartHalo’s optimization failure and core components (i.e., static analysis and LLM). We found that most of optimization failures in E1 are caused by the incorrect facts generated by Gigahorse (i.e., the decompiler of SmartHalo’s static analysis module). These optimization failures can be further eliminated by integrating a more accurate decompiler in the future. We found that most of optimization failures in E2 and E5 are caused by the limited model capability of GPT-3.5. As proved by our experiment in Table VII, SmartHalo equipped with GPT-4o mini can achieve a precision rate of 91.32% and a recall rate of 87.38% for function boundaries, a precision rate of 90.40% and a recall rate of 88.82% for variable types, and a precision rate of 80.66% and a recall rate of 91.78% for contract attributes. Any static analysis approach like ours is fundamentally limited to addressing optimization failures in E3 and E4, as optimization failures in E3 require analyzing the off-chain data source and optimization failures in E5 require analyzing the inheritance relationship which is actually missed at the bytecode level in smart contract.

C. Comparison to Prior Work

To address RQ2, we compare SmartHalo with recent works.

SigRec is not open-sourced, and we are unable to reproduce SigRec because its core algorithm (i.e., TASE) lacks disclosure of essential implementation details. The model dataset for DeepInfer is unavailable for download; moreover, the method omits critical information regarding model inputs, architecture, hyperparameters, and training protocols. Consequently, we could not reproduce DeepInfer. Finally, we were unable to conduct a comparative evaluation between SmartHalo and Neural-FEBI, as the publicly released Neural-FEBI repository omits key modules and contains erroneous model artifacts.

Indeed, SmartHalo is expected to outperforms SigRec, DeepInfer and Neural-FEBI, as they can recover only a subset of variable types/function boundaries. Specifically, Neural-FEBI does not support boundary recovery for complex functions such as modifier functions or functions inherited from other contracts/subcontracts. Conversely, SmartHalo optimizes boundaries for all functions. And SigRec and DeepInfer only partially recover known function signatures and parameter types. Conversely, SmartHalo optimizes types for all variables

Fortunately, we contacted the authors of SmartDagger [5] and obtained the artifact. Further, we download the artifact of VarLifter [6] from the open-source repository. Hence, we

TABLE IV
COMPARISON RESULTS BETWEEN SMARTHALO, VARLIFTER AND SMARTDAGGER IN TERMS OF CONTRACT ATTRIBUTE OR VARIABLE TYPE

Metrics	Contract Attribute					Variable Type				
	TP	FP	FN	Prec.	Recall	TP	FP	FN	Prec.	Recall
SmartDagger [5]	111	364	641	23.37%	10.07%	N/A				
VarLifter [6]	N/A					115	31	1486	78.76%	7.18%
SmartHalo	752	350	75	68.06%	90.93%	1349	113	252	92.27%	84.26%

"N/A" means the corresponding tool does not support this optimization metric.

run SmartHalo and SmartDagger over the evaluation dataset to compare their precision and recall in terms of contract attribute. And we run SmartHalo and Varlifter over the evaluation dataset to compare their precision and recall in terms of variable type.

As can be seen in Table IV, SmartHalo significantly outperforms SmartDagger and Varlifter. In terms of contract attribute, compared to SmartDagger, the precision and recall of SmartHalo are improved by 44.69% and 80.86%. Further, we manually investigate the optimization failures generated by SmartDagger. Our findings indicate that the performance of SmartDagger degrades when encountering our dataset, which includes many newly-emerging smart contracts, due to limitations in its model capability and training dataset. In terms of variable type, compared to VarLifter, the precision and recall of SmartHalo are improved by 13.51% and 77.08%. Similarly, we also manually inspect the optimization failures generated by VarLifter. Our investigation results shows that VarLifter can only recover a small subset of variable types, owing to its low-coverage heuristic rules. And for most variable types, the output is marked as unknown, resulting in a large number of false negatives.

Conversely, SmartHalo equipped with GPT-3.5 achieves a higher precision of 68.06% and a higher recall of 90.93% for contract attributes, and exhibits a higher precision of 92.27% and a higher recall of 84.26% for variable types. When equipped with GPT-4o mini, SmartHalo achieves a even higher precision of 80.65% for contract attributes, and presents a more higher recall of 88.81% for variable types.

D. Ablation Study

Actually, SmartHalo is composed of two components including static analysis and large language models. To answer RQ3, we compare the Dedaub decompiler, LLM (GPT-3.5) and SmartHalo in terms of function boundaries and variable types, to evaluate the contribution of two components to the overall effectiveness of SmartHalo. Note that we do not compare them in terms of contract attributes because LLM is incapable of outputting the contract attribute without our inference prompt (e.g., contract attribute candidate). To conduct this comparison, we further ran the LLM over the evaluation dataset.

As can be seen in Table V, compared with Dedaub decompiler, LLM presents a limited improvement in precision and recall for function boundary identification. While LLM improves the precision of variable type inference by 10.01%, but LLM drop the recall of variable type inference by 18.80%. We manually inspect the optimization failures of LLM. Our manual investigation results shows that, (1) for function boundary identification, although LLM is capable of recovering a part of inherited functions, it also produces incorrect function boundary prediction which undermines the correct function boundaries in the original decompiler output due to the absence of static knowledge prompts. (2) for variable type inference, the recall of SmartHalo decreases rapidly, because a certain number of state variables and their types are unexpectedly deleted in the optimized code generated by LLM alone.

Benefiting from the inference prompts generated by static analysis, SmartHalo improves the precision and recall of function boundaries by 19.23% and 29.23%, as well as enhances the precision and recall of variable type by 15.01% and 60.84%, compared to LLM alone. In conclusion, static analysis and the large language model collaboratively contribute to the improvement of decompiler output optimization in SmartHalo.

E. Generalizability Ability

To answer RQ4, we evaluate the generalizability of SmartHalo in terms of three aspects: 1) real-world complex contract, 2) cross-LLM applicability and 3) cross-decompiler adaptability.

Feasibility to complex and diverse contracts. As discussed in Sections IV-A and IV-B, SmartHalo has two advantages: 1) both its SA and the LLM components are designed to be general, which enable SmartHalo to address diverse and complex real-world scenarios effectively; and (2) the prompts generated by static analysis embed rich code-context information which captures all the contents relevant to the optimization targets across the whole contract, which enables SmartHalo to perform contract-level optimization.

To evaluate this feasibility, we utilize the complex-contract dataset introduced by previous research [36], which aggregates 682 real-world DApps from 1,199 audit reports. From this corpus, we randomly sampled 50 smart contracts (i.e., around 900 functions in total) for evaluation. Manual investigation of these 50 contracts confirmed that (1) they are highly complex, which extensively cover inheritance, modifiers, and complex storage patterns; and (2) they are sufficiently diverse, which possess more than 12 mainstream smart-contract scenarios.

As shown in Table VI, SmartHalo exhibits good performance on all evaluation metrics when optimizing these complex and diverse contracts at whole-contract optimization granularity. For example, SmartHalo improves the recall of function boundary identification by 20.44%; increases the precision and recall

TABLE V
COMPARISON RESULTS BETWEEN SMARTHALO AND LLM-FACILITATED APPROACH

Metrics	Function Boundary					Variable Type				
	TP	FP	FN	Prec.	Recall	TP	FP	FN	Prec.	Recall
Original decompiler output (Baseline)	316	149	310	67.96%	50.48%	676	410	925	62.25%	42.22%
LLM without static inference prompt	321	144	305	69.03%	51.28%	375	144	1226	72.26%	23.42%
SmartHalo	504	67	122	88.26%	80.51%	1349	113	252	92.27%	84.26%

TABLE VI
COMPARISON RESULTS OVER COMPLEX CONTRACT DATASET, FOR EVALUATING SMARTHALO'S FEASIBILITY TO REAL-WORLD CONTRACTS

Metrics	Function Boundary					Variable Type					Contract Attribute				
	TP	FP	FN	Prec.	Recall	TP	FP	FN	Prec.	Recall	TP	FP	FN	Prec.	Recall
Original decompiler	519	223	381	69.95%	57.67%	2955	867	874	77.31%	76.18%	N/A				
SmartHalo	703	267	197	72.47%	78.11%	3607	222	272	94.20%	92.98%	433	65	464	86.95%	48.27%

"N/A" : the corresponding tool does not support optimizing this deficiency.

TABLE VII
THE LLMS EVALUATED IN OUR STUDY

Model Type	Model Name	Open-Source	Reasoning	Size
General LLM	GPT-3.5	✗	✗	-
	GPT-4o mini	✗	✗	-
	Llama-3 (Local)	✓	✗	7B
	Deepseek-v3	✓	✓	671B
Code LLM	Qwen-2.5-coder	✓	✗	32B

of variable type inference by 16.89% and 16.80%; and achieve the recovery of contract attributes with a precision of 86.95% and a recall of 48.27%, whereas the original decompiled outputs lacks the contract attributes. In summary, SmartHalo can be effectively applied to a wide range of complex real-world contract types, and it exhibits good scalability to enables contract-level decompilation optimization.

Applicability to different LLMs. Here, we further investigate SmartHalo's generalizability on integrating different LLMs. The study of the kind of LLM that can better drive SmartHalo is not the focus of our work. Conversely, we aim to explore whether SmartHalo maintains a good performance across different LLMs. To this end, we selected a diverse set of LLMs for evaluation, which considers dimensions such as general LLM versus code LLM, open-source versus closed-source models, reasoning-enabled versus non-reasoning models, and different versions of the same LLM (see Table VII). Among these LLMs, "Llama 3" is actually a small-scale model (7B parameters) that we deploy in local environment.

As shown in Table VIII, SmartHalo maintains good performance across the three metrics when applying to different LLMs. For GPT-4o mini, the precision and recall of function boundary identification are improved by 23.26% and 36.90%, the precision and recall of variable type inference are increased by 28.15% and 46.60%, and the contract attributes are successfully identified with a precision of 83.66% and a recall of 90.93%. For Deepseek-R1, the recall of function boundary identification is enhanced by 41.37%, the precision and recall of variable type inference are increased by 19.47% and 28.42%, and the contract attributes are successfully recovered with a precision of 83.91% and a recall of 86.11%. For Qwen-2.5-coder, the precision and recall of function boundary identification are improved by 95.60% and 85.90%, the precision and recall of variable type inference are increased by 23.89% and 13.31%, and the contract attributes are successfully identified with a precision of 70.54% and a recall of 88.88%. In particular, SmartHalo continues to perform well on the locally-deployed 7B-parameter Llama-3 model (see the third row of Table VIII), which indicates that SmartHalo can support nearly zero-cost local deployment.

In summary, SmartHalo exhibits good generalizability for a wide range of LLMs regardless of their model type, open-source status, reasoning capability, or model version.

Adaptability to different decompilers. Lastly, we evaluate SmartHalo's ability to deal with discrepancies among different Solidity decompilers. To this end, we selected three mainstream decompilers, Gigahorse (i.e., Elipmoc), Heimdall, and Panoramix, all of which have been widely applied in prior studies and industry [37], [38]. Specifically, we run these three decompilers on the 456 functions contained in our evaluation dataset to obtain their respective decompiled outputs. Subsequently, SmartHalo is run on the decompiled code produced by each decompiler, and the results of three evaluation metrics are collected for comparative analysis.

As illustrated in Table IX, evaluation results show that SmartHalo improves both precision and recall across all the metrics for different decompilers. For Heimdall, the precision and recall of function boundary identification are enhanced by 12.19% and 19.80%, the precision and recall of variable type inference are increased by 16.67% and 32.36%, and the contract attributes are successfully recovered with a precision of 46.63% and a recall of 99.52%, whereas the original decompiled outputs lack the contract attributes. For Panoramix, the precision and recall of function boundary identification are improved by 20.54% and 22.84%, the precision and recall of variable type inference are increased by 26.15% and 43.60%, and the contract attributes are successfully identified with a precision of 81.76% and a recall of 81.86%, whereas the original decompiled outputs lack the contract attributes. These results indicate that SmartHalo is generic to different Solidity decompilers.

TABLE VIII
EVALUATION RESULTS ACROSS SMARTHALO'S VERSIONS WITH DIFFERENT LLMS, FOR EVALUATING THE SMARTHALO'S APPLICABILITY TO DIFFERENT LLMS

Setup	Function Boundary					Variable Type					Contract Attribute				
	TP	FP	FN	Prec.	Recall	TP	FP	FN	prec.	recall	TP	FP	FN	prec.	recall
SmartHalo with GPT-3.5	504	67	122	88.26%	80.51%	1349	113	252	92.27%	84.26%	752	350	75	68.06%	90.93%
SmartHalo with GPT-4o mini	547	52	79	91.32%	87.38%	1422	151	179	90.40%	88.82%	759	182	68	80.66%	91.78%
SmartHalo with Llama	345	177	281	66.09%	55.11%	777	468	824	62.41%	48.53%	499	310	328	61.68%	60.34%
SmartHalo with Deepseek-R1	575	397	51	59.16%	91.85%	1131	253	470	81.72%	70.64%	626	120	201	83.91%	86.11%
SmartHalo with Qwen	544	25	82	95.60%	86.90%	889	143	712	86.14%	55.53%	735	307	92	70.54%	88.88%

TABLE IX
EVALUATION RESULTS ACROSS DIFFERENT DECOMPILERS, FOR EVALUATING SMARTHALO'S ADAPTABILITY TO DIFFERENT DECOMPILERS

Metrics	Function Boundary					Variable Type					Contract Attribute				
	TP	FP	FN	prec.	recall	TP	FP	FN	prec.	recall	TP	FP	FN	prec.	recall
Dedaub	316	149	310	67.96%	50.48%	676	410	975	62.25%	42.22%	N/A				
SmartHalo on Dedaub	504	67	122	88.26%	80.51%	1349	113	252	92.27%	84.26%	752	350	75	68.06%	90.93%
Heimdall	316	149	310	67.96%	50.48%	984	1082	617	47.63%	61.46%	N/A				
SmartHalo on Heimdall	440	109	186	80.15%	70.28%	1502	834	99	64.30%	93.82%	823	942	4	46.63%	99.52%
Panoramix	225	149	401	60.16%	35.94%	583	348	1018	62.62%	36.41%	N/A				
SmartHalo on Panoramix	368	88	258	80.70%	58.78%	1281	162	320	88.77%	80.01%	677	151	150	81.76%	81.86%

"N/A" : the corresponding tool does not support optimizing this deficiency.

TABLE X
THE AVERAGE TIME AND TOKENS FOR ANALYZING EACH METHOD IN THE EVALUATION DATASET

Efficiency of SmartHalo	Avg.time(s)	Avg.tokens
Dependency-based semantic extraction	1.94	-
LLM-driven semantic enrichment	8.95	4272 ($ 0.00065)
Correctness verification	13.10	4648 ($ 0.00071)
Total	23.99	8920 ($ 0.00136)

F. Efficiency

To address RQ5, we execute SmartHalo on our evaluation dataset and measure both its runtime and monetary costs.

As shown in Table X, the proposed approach can establishes high-quality optimization with an efficient and cost-effective manner. For example, we compute the average time for each step of SmartHalo: 1.94 s for the dependency-based semantic extraction module, 8.95 s for the LLM-driven semantic enrichment module and 13.10 s for the correctness verification module. For monetary costs, the GPT-4o mini-powered version of SmartHalo consumes an average of 8,920 tokens for each function, which incurs an average expense of $0.00136. With our statistical analysis, the average number of functions per contract within the evaluation dataset is approximately 34. Accordingly, we can roughly estimate that, SmartHalo costs an average expense of $0.046 and an average runtime of 815.66 s per smart contract.

In addition, with the rapid development of LLMs, an increasing number of open-source alternative LLMs has emerged. As mentioned earlier, SmartHalo can be deployed with a local, cost-free, small-scale LLama-7B model and achieve good performance, as shown in Table VIII. In this way, SmartHalo's monetary cost can be reduced to nearly zero.

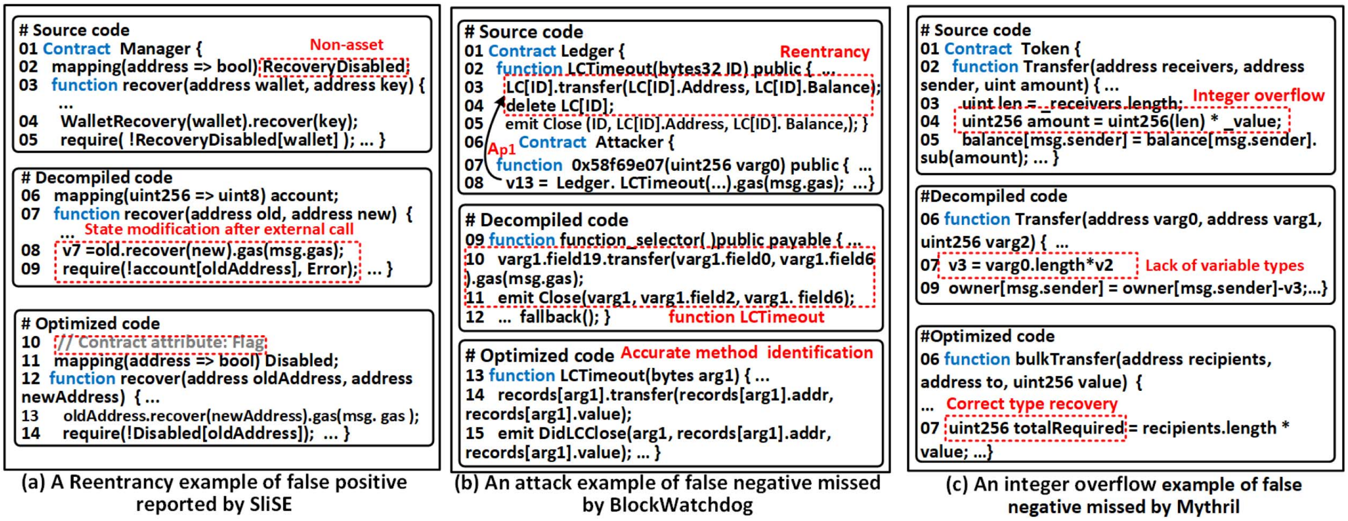
In sum, SmartHalo is proved to be cost-effective in terms of runtime and monetary costs, which indicate its practical value.

Parameter setting influence. Further, we evaluate how the settings of iteration limit influence the cost of correctness verification. Specifically, we run the correctness verification module with different iteration limit ($n = 1, 2, 3...$), and compute the convergence rate, runtime and monetary costs of evaluated functions. The evaluation results show that, when $n \geq 3$, both the runtime and monetary costs stop increasing, as the convergence rate of correctness verification module reaches 100%. Hence, we set the iteration limit n to 3 in our evaluation setup.

VI. EFFECTIVENESS FOR DOWNSTREAM TASK

These experiments were designed to assess how the contract attributes, function boundaries, and variable types recovered by SmartHalo contribute to their respective downstream tasks.

Contract attributes for Reentrancy vulnerability detection. Prior research [5] shows that the effectiveness of contract attribute recovery reflects on eliminating the false positives for Reentrancy detection. Hence, We evaluate how the contract attributes optimized by SmartHalo contribute to improving the precision of Reentrancy detection. To this end, we utilize the latest manually-labeled DApp dataset introduced in previous research [39], with a total of 81 positive labels for Reentrancy vulnerabilities. To conduct this evaluation, we run the SOTA tool SliSE [39] and another tool that integrates SliSE with SmartHalo over the manually-labeled DApp dataset to evaluate their precision. As can be seen in Table XI, SliSE+SmartHalo improve the precision to 80.41%, whereas the precision of SliSE is only 72.16%. We further manually inspect all the false positives reported by these two tools. We found that 8 of 27 false positives reported by SliSE can be eliminated by SmartHalo, with the help of more accurate contract attributes recovered by SliSE+SmartHalo. Fig. 8(a) illustrates a Reentrancy example of a false positive reported by SliSE. In this instance, SliSE erroneously flags the function *recover* due to its adherence



```
# Source code
01 Contract  Manager {
02   mapping(address => bool) RecoveryDisabled;
03   function recover(address wallet, address key) {
      ...
04    WalletRecovery(wallet).recover(key);
05    require( !RecoveryDisabled[wallet] ); ... }
```

```
# Decompiled code
06   mapping(uint256 => uint8) account;
07   function recover(address old, address new)  {
      ...
08    v7 =old.recover(new).gas(msg.gas);
09    require(!account[oldAddress], Error); ... }
```

```
# Optimized code
10   // Contract attribute: Flag
11   mapping(address => bool) Disabled;
12  function recover(address oldAddress, address
newAddress)  { ...
13     oldAddress.recover(newAddress).gas(msg. gas );
14     require(!Disabled[oldAddress]);  ... }
```

(a) A Reentrancy example of false positive reported by SliSE

```
# Source code
01 Contract Ledger {
02   function LCTimeout(bytes32 ID) public { ...
03    LC[ID].transfer(LC[ID].Address, LC[ID].Balance);
04    delete LC[ID];
05    emit Close (ID, LC[ID].Address, LC[ID]. Balance,); }
06   Contract  Attacker {
07   function  0x58f69e07(uint256 varg0) public {  ...
08    v13 =  Ledger. LCTimeout(...).gas(msg.gas);  ...}
```

```
# Decompiled code
09 function function_selector( )public payable { ...
10   varg1.field19.transfer(varg1.field0, varg1.field6
).gas(msg.gas);
11   emit Close(varg1, varg1.field2, varg1. field6);
12   ...  fallback(); }
```

```
# Optimized code
13 function LCTimeout(bytes arg1) { ...
14   records[arg1].transfer(records[arg1].addr,
records[arg1].value);
15   emit DidLCClose(arg1, records[arg1].addr,
records[arg1].value); ... }
```

(b) An attack example of false negative missed by BlockWatchdog

```
# Source code
01 Contract  Token {
02   function Transfer(address receivers, address
sender, uint amount) { ...
03     uint len = _receivers.length;
04    uint256 amount = uint256(len) * _value;
05     balance[msg.sender] = balance[msg.sender].
sub(amount); ... }
```

```
#Decompiled code
06 function Transfer(address varg0, address varg1,
uint256 varg2) {  ...
07  v3 = varg0.length*v2
09  owner[msg.sender] = owner[msg.sender]-v3;...}
```

```
#Optimized code
06 function bulkTransfer(address recipients,
address to, uint256 value)  {
...
07 uint256 totalRequired = recipients.length *
value; ...}
```

(c) An integer overflow example of false negative missed by Mythril

Fig. 8. The sample false alarms reported by SOTA tools. These FPs and FNs can be eliminated with the integration of SmartHalo.

TABLE XI
COMPARISON RESULTS FOR EVALUATING THE EFFECTIVENESS OF SMARTHALO ON THE REENTRANCY DETECTION

Reentrancy	Precision		
	TP	FP	Rate
SliSE	70	27	72.16%
SliSE+SmartHalo	78	19	80.41%

TABLE XII
COMPARISON RESULTS FOR EVALUATING THE EFFECTIVENESS OF SMARTHALO ON THE ATTACK IDENTIFICATION

Attack Contract Identification	Recall		
	TP	FN	Rate
BlockWatchdog	15	3	83.33%
BlockWatchdog+ SmartHalo	18	0	100.00%

to a pre-defined Reentrancy pattern, where function *recover* modifies state variables after the external call (lines 8 and 9). Upon meticulous inspection of the source code, we found that the contract attribute of the state variable *RecoveryDisable* belong to *[non-asset]* (line 02). This attribute indicates that the *recover* cannot be exploited for profit gain by adversaries, thus *recover* does not pose a Reentrancy risk. In contrast, SmartHalo accurately identifies the contract attribute of this state variable as [Flag] (line 10), effectively eliminating this false positive.

Function identification for attack identification. Prior research [7] shows that attack identification relies heavily on the precise cross-contract call flow analysis between victim and attacker contracts, which effectively models the attack path. Incorrect function identification can lead to missed external calls and corresponding call flows, thereby impeding the identification of attack path. Therefore, the effectiveness of function boundary optimization is reflected in reducing false negatives in attack identification. To assess how function boundaries optimized by SmartHalo enhance the recall of attack identification, we utilize the latest attack contract datasets introduced by previous research [7], comprising 18 pairs of attacker and victim contracts from 15 real-world incidents. Further, we compare the SOTA tool BlockWatchdog [7] with an enhanced version that integrates SmartHalo with BlockWatchdog. As shown in Table XII, BlockWatchdog+SmartHalo enhances the recall by 16.67% compared with BlockWatchdog alone. Additionally, we manually inspect all the false negatives reported by BlockWatchdog, finding that BlockWatchdog+SmartHalo successfully eliminates all false positives through more accurate function identification optimized by SmartHalo. Fig. 8(b) presents an example of false negative missed by BlockWatchdog. In this case, function *LCtimeout* contains a Reentrancy vulnerability, where it logs the transfer results (line 4) after transferring the asset (line 3). Further, an adversary could exploit this by invoking the function *LCTimeout* to trigger the Reentrancy vulnerability (path A_{p1}). BlockWatchdog fails to detect this attack path (i.e., false negative), because function *LCtimeout* is unavailable for BlockWatchdog at the decompiled code level [7]. In contrast, SmartHalo accurately identify the function *LCTimeout* (line 13-15), thereby eliminating this false negative.

Variable types for Integer-overflow vulnerability detection. Prior research [25], [40] demonstrate that detecting integer-overflow vulnerabilities heavily depends on accurately identifying the upper and lower bounds of variables based on their types. Incorrect variable type recovery results in both false positives and false negatives [25]. Therefore, the effectiveness of variable type optimization is reflected in its ability to reduce false negatives and false positives in integer-overflow vulnerability detection. We further evaluate the effectiveness of the variable types optimized by SmartHalo on the precision and recall of integer-overflow vulnerability detection. For this evaluation, we utilize integer overflow datasets collected from real-world attacks in previous research [41], which includes 50 vulnerable contracts with integer overflow vulnerabilities.

TABLE XIII
COMPARISON RESULTS FOR EVALUATING THE EFFECTIVENESS OF SMARTHALO ON INTEGER-OVERFLOW VULNERABILITY

Integer Overflow Vulnerability	Precision			Recall		
	TP	FP	Rate	TP	FN	Rate
Mythril	13	5	72.22%	13	37	26.00%
Mythril+ SmartHalo	32	2	94.18%	32	18	64.00%

Further, we compare the SOTA tool Mythril [42] with a modified version that integrates Mythril with SmartHalo. As shown in Table XII, Mythril+SmartHalo enhances the precision by 21.96% and the recall by 38.00% compared with Mythril alone. Additionally, we manually investigate all the false negatives and false positives reported by Mythril. We found that 60% of the false positives and 51.35% of the false negatives reported by Mythril can be eliminated by Mythril+SmartHalo, thanks to the more accurate variable types optimized by SmartHalo. Fig. 8 shows an example of false negative missed by Mythril. In this case, the function *batchTransfer* contains a *uint256* variable *amount*, which is correlated with two arguments, *receivers* and *value*, both of which can be passed arbitrarily via external calls (line 4), leading to an integer overflow vulnerability.

Mythril detects integer overflow based on two conditions: (1) scanning for key EVM instructions (e.g., ADD, SUB, MUL, etc.), and (2) invoking Z3 solver with the maximum bit-width derived from the variable type to determine whether integer overflow occurs. Due to the absence of type information for the variable *v3* (line 07) in the decompiled code, Mythril fails to identify the maximum bit-width, resulting in a false negative report. Conversely, SmartHalo correctly identifies the type of variable *v3* as a *uint256* type (line 07), thus eliminating the false negative.

VII. DISCUSSION AND LIMITATION

SmartHalo shares the following advantages in decompiler output optimization: (1) As evidenced by evaluation, SmartHalo is obviously effective in optimizing function boundaries, variable types and contract attributes, demonstrating superior performance compared to previous approaches; (2) SmartHalo establishes a novel combination of static analysis (SA) and large language models (LLMs), enabling the joint optimization on the decompiler output with the advantages of both SA and LLMs, thereby overcoming the limitation of prior works. (3) SmartHalo implements a rigorous correctness verification for the optimization output to eliminate the optimization errors.

SmartHalo has good adaptability to newly-deployed contracts due to its flexible and generic core components: (1) Static analysis employs typical control-flow analysis to extract the dependencies that are generic to all contracts. These dependency-extraction methods are equally applicable to newly deployed contracts; (2) The LLMs is adept at few-shot in-context learning, which refers to the capability of LLMs to instantaneously comprehend and address new and unseen cases/tasks during the inference stage, based solely on a small number of known examples (shots). This helps SmartHalo to be feasible to the newly-deployed contracts.

Although our evaluation of SmartHalo focuses on the function level, SmartHalo is actually capable of supporting contract-level optimizations. Specifically, the prompts generated by static analysis embed complete code-context information which captures all the contents relevant to the optimization targets across the whole contract, which enables SmartHalo to perform contract-level optimization. For example, the prompt templates of Table I introduce the contextual information of modifiers (i.e., row 10), which guides the LLM to reconstruct the corresponding modifier functions; and as highlighted by the yellow part of Fig. 6(a), the COT prompts produced by static analysis effectively capture the dependencies between key–value pairs within complex storage patterns (e.g., mappings and structs), this capability facilitates the recovery of the complex storage variable types (i.e., mapping (bytes 32 = uint256) data). Further, as evidenced by Section V-E, we have evaluated SmartHalo's feasibility to complex and diverse contracts. And the results confirm that SmartHalo exhibits good scalability for contract-level decompilation optimization.

The evolution of LLM APIs presents potential challenges to the reproducibility of evaluation results. To address this concern, we have implemented the following strategies. Firstly, we explicitly list the exact API configurations for each model in the SmartHalo's repository. These details encompass the server provider, model identifier, version/release date, invocation endpoints, context length limitations, and relevant environment variables. Further, for open-source models, except for API configurations, we provide an access link of the repository corresponding to each model version used in this paper. Finally, as evidenced by Section V-E, SmartHalo maintains a good performance on the evaluated smaller language model. In future work, we can explore integrating more alternative fine-tuned smaller language models into SmartHalo to further reduce the effects of evolving LLMs on reproducibility.

A current limitation is that SmartHalo cannot recover inheritance structures within contracts. The reason for this limitation is that, inheritance relationship and class information are missed at the bytecode level in smart contracts. Particularly, state-of-the-art efforts [9], [19] in other domains (e.g., C++, Java) also cannot recover the inheritance structure for smart contract, because they mainly rely on class information. Given the difficulty of this task, we leave the exploration of inheritance structure recovery to future work.

Threats to validity. An external threat to validity is that the number of smart contract functions used in our evaluation here is relatively small (i.e., 456 smart contract functions). However, we consider the dataset size sufficient for the following reasons. Firstly, the dataset was randomly selected from the largest real-world contract repository. Secondly, the dataset size is similar to those used in SOTA studies [4], [17]. Thirdly, our manual investigation confirms that the dataset involves diverse and mainstream application scenarios of smart contract[2].

An internal threat to validity is the predefined type rules employed by SmartHalo for correctness verification (Section IV-C). SmartHalo remains a highly generalizable framework for the following reasons: First, SmartHalo focuses on optimizing Solidity decompiler output. This setup is aligned with recent contract analysis research [1], [43]. Second, most contracts are written in Solidity [1] and are widely deployed on major blockchains (e.g., Ethereum, TRON, and BNB). Third, the static rules(see Fig. 7) are generic across different Solidity versions, as they reflect knowledge of fundamental types (e.g., array). A review of Solidity's release-changelogs indicates that these rules remained unchanged across all Solidity's major updates [44]. Fourth, SmartHalo is backward-compatible, can easily adapt to contract evolution with corresponding knowledge.

VIII. Related Work

Smart Contract Decompilation. The decompilation of smart contracts has garnered significant attention due to the prevalence of bytecode-only smart contracts [45] Gigahorse [8] translates smart contracts from low-level EVM bytecode into a 3-address code representation. Elipmoc [37] is the 2.0 verson of Gigahorse, which employs a novel context sensitivity called transactional sensitivity to achieve a more effective static abstraction. Erays [46] and EtherSolve [47] aim to recover the CFG from the EVM bytecode. Recently, SigRec [11] has been proposed to recover the signatures of public functions. DeepInfer [10] leverages deep learning techniques to infer function signatures.

Decompilation Optimization. The existing decompilation optimization research mainly focus on other languages such as C++ and Java [48], [49], which can be divided into the following two categories [50]. The first category aims at recovering the semantic information from the assembly code or intermediate language. Among them, previous works, including DEBIN [14], OSPREY [51], and BDA [52], analyze the program dependency via the static analysis. However, they are with low coverage due to their reliance on heuristic rules. Another group of works, including Nero [53], NFRE [54], SYMLM [55], focus on designing encoder-decoder architecture models to predict the function name for the binary. The other category of work focuses on optimizing variable name, variable type, and structure of code generated by decompiler, including DIRE [56], DIRTY [15] and DeGPT [4].

IX. Conclusion

In this paper, we introduced SmartHalo, a novel framework designed to address critical deficiencies in existing Solidity smart contract decompilers. By leveraging the strengths of static analysis (SA) and large language models (LLMs), SmartHalo significantly enhances the accuracy and readability of decompiled code. Extensive evaluation demonstrates that SmartHalo outperforms state-of-the-art decompilers, showing substantial improvements in function boundary identification, variable type inference, and contract attribute recovery. We also showed that the output of SmartHalo can significantly enhance the effectiveness of downstream tasks.

[2] Note that the manual investigation results are available via https://figshare.com/s/5d4fc1ad2312a4be9370?file=49346911

References

[1] Z. Liao et al., "SmartAxe: Detecting cross-chain vulnerabilities in bridge smart contracts via fine-grained static analysis," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 249–270, 2024.

[2] "Ethereum virtual machine." Accessed: Aug. 1, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/evm/

[3] "Solidity." Accessed: Mar. 23, 2024. [Online]. Available: http://solidity.readthedocs.io/

[4] P. Hu, R. Liang, and K. Chen, "DeGPT: Optimizing decompiler output with LLM," in *Proc. Netw. Distrib. Syst. Secur. Symp.* vol. 267622140, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID

[5] Z. Liao, Z. Zheng, X. Chen, and Y. Nan, "SmartDagger: A bytecode-based static analysis approach for detecting cross-contract vulnerability.," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, Virtual Event, South Korea, 2022, pp. 752–764.

[6] Y. Li, W. Song, and J. Huang, "VarLifter: Recovering variables and types from bytecode of solidity smart contracts," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 1–29, 2024.

[7] S. Yang, J. Chen, M. Huang, Z. Zheng, and Y. Huang, "Uncover the premeditated attacks: detecting exploitable reentrancy vulnerabilities by identifying attacker contracts," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.,* 2024, pp. 1–12.

[8] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Montreal, QC, Canada. Piscataway, NJ, USA: IEEE Press, May 2019, pp. 1176–1186.

[9] "EtherScan." Accessed: Mar. 23, 2024. [Online]. Available: https://etherscan.io/

[10] K. Zhao, Z. Li, J. Li, H. Ye, X. Luo, and T. Chen, "DeepInfer: Deep type inference from smart contract bytecode," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., (ESEC/FSE)*, San Francisco, CA, USA, 2023, pp. 745–757.

[11] T. Chen et al., "SigRec: Automatic recovery of function signatures in smart contracts," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 3066–3086, Aug2022.

[12] J. He, S. Li, X. Wang, S.-C. Cheung, G. Zhao, and J. Yang, "Neural-Febi: Accurate function identification in Ethereum virtual machine bytecode," 2023, *arXiv:2301.12695*.

[13] L. Dramko, J. Lacomis, E. J. Schwartz, B. Vasilescu, and C. L. Goues, "A taxonomy of c decompiler fidelity issues," in *Proc. 33th USENIX Secur. Symp. (USENIX Secur.)*, 2024, pp. 379–396.

[14] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1667–1680.

[15] Q. Chen, J. Lacomis, E. J. Schwartz, C. L. Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur.),* 2022, pp. 4327–4343.

[16] "Ethereum open-source smart contract." Accessed: Sep. 6, 2024. [Online]. Available: https://xblock.pro/\#/dataset/65

[17] W. Ma et al., "Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications," 2024, *arXiv:2403.16073*.

[18] J. Su and M. Jiang, "A hybrid entropy and blockchain approach for network security defense in SDN-based iiot," *Chin. J. Electron.*, vol. 32, no. 3, pp. 531–541, 2023.

[19] J. Yao, B. Yang, T. Wang, and W. Zhang, "A distributed self-tallying electronic voting system using the smart contract," *Chin. J. Electron.*, vol. 33, no. 4, pp. 1063–1076, 2024.

[20] W. Wang, G. Zhang, H. Hu, and S. Ding, "Blockchain-based platform for the compliance inspection of the ready-mixed concrete supply chain," *Tsinghua Sci. Technol.*, vol. 30, no. 2, pp. 769–781, 2024.

[21] F. A. Al-Yarimi, R. Salah, and K. Mohamoud, "Blockchain-driven secure data sharing framework for edge computing networks," *Tsinghua Sci. Technol.*, vol. 30, no. 3, pp. 978–997, 2024.

[22] "Smart contract statistic." Accessed: Mar. 23, 2024. [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary

[23] A. Jaffe, J. Lacomis, E. J. Schwartz, C. L. Goues, and B. Vasilescu, "Meaningful variable names for decompiled code: A machine translation approach," in *Proc. 26th Conf. Program Comprehension*, 2018, pp. 20–30.

[24] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in Ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[25] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in Ethereum," in *Proc. 4th Int. Conf. Cryptogr., Secur. Privacy*, 2020, pp. 110–115.

[26] K. Sun, Z. Xu, C. Liu, K. Li, and Y. Liu, "Demystifying the composition and code reuse in solidity smart contracts," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., (ESEC/FSE)*, 2023, pp. 796–807.

[27] X. Li, J. Li, H. Ma, and H. K. Lo, "A channel-independent transformer approach for ride-hailing demand prediction with internet sentiment and transport capacity," *Fundamental Res.,*

[28] Y. Wang, Y. Qing, K. Huang, C. Dang, and Z. Wu, "Preformer Mot: A transformer-based approach for multi-object tracking with global trajectory prediction," *Fundamental Res.*, pp. 1–9, Jan. 2025

[29] Y. Peng, C. Wang, W. Wang, C. Gao, and M. R. Lyu, "Generative type inference for Python," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 988–999.

[30] "Tree-sitter." Accessed: Jul. 31, 2024. [Online]. Available: https://github.com/tree-sitter/tree-sitter

[31] Z. Liao, S. Hao, Y. Nan, and Z. Zheng, "SmartState: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, 2023, pp. 980–991.

[32] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 226–237.

[33] L. M. de Moura and N. S. Bjørner, "Z3: An efficient SMT solver," in *Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[34] S. Badihi, F. Akinotcho, Y. Li, and J. Rubin, "ARDiff: Scaling program equivalence checking via iterative abstraction and refinement of common code," in *Proc. 28th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE),* Virtual Event, USA, 2020, pp. 13–24.

[35] D. Williams-King et al., "Egalito: Layout-agnostic binary recompilation," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 133–147.

[36] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, and M. Ye, "DAppSCAN: Building large-scale datasets for smart contract weaknesses in DApp projects," *IEEE Trans. Softw. Eng.*, vol. 50, no. 6, pp. 1360–1373, Jun. 2024.

[37] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis, "Elipmoc: Advanced decompilation of Ethereum smart contracts," in *Proc. ACM Program. Lang.*, vol. 6, Apr. 2022, pp. 1–27.

[38] X. Su, H. Liang, H. Wu, B. Niu, F. Xu, and S. Zhong, "Disco: Towards decompiling EVM bytecode to source code using large language models," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 2311–2334, 2025.

[39] Z. Wang, J. Chen, Y. Wang, Y. Zhang, W. Zhang, and Z. Zheng, "Efficiently detecting reentrancy vulnerabilities in complex smart contracts," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 161–181, 2024.

[40] J. Sun, S. Huang, C. Zheng, T. Wang, C. Zong, and Z. Hui, "Mutation testing for integer overflow in ethereum smart contracts," *Tsinghua Sci. Technol.*, vol. 27, no. 1, pp. 27–40, 2021.

[41] S. Zhou, Z. Yang, J. Xiang, Y. Cao, M. Yang, and Y. Zhang, "An ever-evolving game: Evaluation of real-world attacks and defenses in Ethereum ecosystem," in *Proc. 29th USENIX Conf. Secur. Symp. (SEC)*, USENIX Assoc., 2020.

[42] Consensys, "Mythril." Accessed: Mar. 23, 2024. [Online]. Available: https://github.com/Consensys/mythril

[43] H. Liu et al., "Using my functions should follow my checks: understanding and detecting insecure openzeppelin code in smart contracts," in *Proc. 33rd USENIX Secur. Symp. (USENIX Secur.)*, USENIX Assoc., 2024, pp. 3585–3601.

[44] "Solidity release changelogs." Accessed: Mar. 23, 2024. [Online]. Available: https://docs.soliditylang.org/en/v0.8.28/080-breaking-changes.html

[45] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF Con.,* vol. 25, no. 11, pp. 1–29, 2017.

[46] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. D. Bailey, "Erays: Reverse engineering ethereum's opaque smart contracts," in *Proc. 27th USENIX Secur. Symp., USENIX Secur.* 2018, pp. 1371–1385.

[47] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, "EtherSolve: Computing an accurate control-flow graph from Ethereum bytecode," in *Proc. 29th IEEE/ACM Int. Conf. Program Comprehension (ICPC)*, Madrid, Spain, 2021, pp. 127–137.

[48] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," 2020, *arXiv:2012.08680*.

[49] J. Peng, Y. Wang, J. Xue, and Z. Liu, "Fast cross-platform binary code similarity detection framework based on CFGS taking advantage of NLP and inductive GNN," *Chin. J. Electron.*, vol. 33, no. 1, pp. 128–138, 2024.

[50] C. Pang et al., "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *Proc. IEEE Symp. Security Privacy (SP),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 833–851.

[51] Z. Zhang et al., "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *Proc. IEEE Symp. Secur. Privacy (SP),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 813–832.

[52] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, "BDA: Practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," in *Proc. ACM Program. Lang.*, 2019, pp. 1–31.

[53] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–28. 2020.

[54] H. Gao, S. Cheng, Y. Xue, and W. Zhang, "A lightweight framework for function name reassignment based on large-scale stripped binaries," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2021, pp. 607–619.

[55] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "SymLM: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 1631–1645.

[56] J. Lacomis et al., "Dire: A neural approach to decompiled identifier naming," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2019, pp. 628–639.