

OPDIFFER: LLM-Assisted Opcode-Level Differential Testing of Ethereum Virtual Machine

JIE MA, Beihang University, China and Zhongguancun Laboratory, China

NINGYU HE, The Hong Kong Polytechnic University, China

JINWEN XI, Zhongguancun Laboratory, China

MINGZHE XING, Zhongguancun Laboratory, China

HAOYU WANG, Huazhong University of Science and Technology, China

YING GAO*, Beihang University, China and Zhongguancun Laboratory, China

YINLIANG YUE*, Zhongguancun Laboratory, China

As Ethereum continues to thrive, the Ethereum Virtual Machine (EVM) has become the cornerstone powering tens of millions of active smart contracts. Intuitively, security issues in EVMs could lead to inconsistent behaviors among smart contracts or even denial-of-service of the entire blockchain network. However, to the best of our knowledge, only a limited number of studies focus on the security of EVMs. Moreover, they suffer from 1) insufficient test input diversity and invalid semantics; and 2) the inability to automatically identify bugs and locate root causes. To bridge this gap, we propose OPDIFFER, a differential testing framework for EVM, which takes advantage of LLMs and static analysis methods to address the above two limitations. We conducted the largest-scale evaluation, covering nine EVMs and uncovering 26 previously unknown bugs, 22 of which have been confirmed by developers and three have been assigned CNVD IDs. Compared to state-of-the-art baselines, OPDIFFER can improve code coverage by at most 71.06%, 148.40% and 655.56%, respectively. Through an analysis of real-world deployed Ethereum contracts, we estimate that 7.21% of the contracts could trigger our identified EVM bugs under certain environmental settings, potentially resulting in severe negative impact on the Ethereum ecosystem.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → *Software security engineering*.

Additional Key Words and Phrases: Ethereum Virtual Machine, Differential Testing, Large Language Model

1 Introduction

Ethereum is the second-largest blockchain platform, supporting smart contracts as its killer application [6]. Numerous applications are built upon smart contracts, such as NFTs [26], decentralized finance [23], and games [16]. At the time of writing, the total market capitalization of Ethereum has reached up to \$307.21 billion [12]. To support the execution of smart contracts, the Ethereum Virtual Machine (EVM) serves a critical role as the core execution environment. Instead of running on each client node, EVMs can also be executed in local environments, such as simulating execution results for smart contracts during prototype testing [52].

Given that vulnerabilities in smart contracts can result in substantial financial losses, the security of smart contracts has drawn considerable attention from both academia and industry [52, 63, 66, 67]. In contrast, the security of the underlying EVM has been underexplored, which can result in even more severe negative impacts. CVE-2021-39137 [17], a consensus vulnerability in Go-Ethereum (Geth) [21], triggered a chain fork on Ethereum mainnet in August 2021. Due to Geth’s widespread

*Ying Gao and Yinliang Yue are the corresponding authors.

Authors’ Contact Information: [Jie Ma](#), Beihang University, Beijing, China and Zhongguancun Laboratory, Beijing, China, majie2023@buaa.edu.cn; [Ningyu He](#), The Hong Kong Polytechnic University, Hong Kong, China, ningyu.he@pku.edu.cn; [Jinwen Xi](#), Zhongguancun Laboratory, Beijing, China, xijw@zgclab.edu.cn; [Mingzhe Xing](#), Zhongguancun Laboratory, Beijing, China, xingmz@zgclab.edu.cn; [Haoyu Wang](#), Huazhong University of Science and Technology, Wuhan, China, haoyuwang@hust.edu.cn; [Ying Gao](#), Beihang University, Beijing, China and Zhongguancun Laboratory, Beijing, China, gaoying@buaa.edu.cn; [Yinliang Yue](#), Zhongguancun Laboratory, Beijing, China.

deployment as the dominant Ethereum client, this vulnerability allowed malicious contracts to fork Geth nodes from the mainnet as a result of inconsistent contract results, introducing double-spending risks. The vulnerability stemmed from memory corruption in the EVM precompiled contract datacopy, highlighting the necessity for rigorous EVM implementation testing.

To the best of our knowledge, only a few tools are available for testing the EVM, including EVMFuzzer [28], NeoDiff [45] and go evmlab [56, 59]. Specifically, EVMFuzzer is the first differential testing tool for EVM, using mutation on the source code of smart contracts to generate test inputs, while NeoDiff directly generates contract bytecode through predefined template. Meanwhile, go evmlab employs state transition tests based on similar pre-defined templates.

However, testing EVMs are hindered by some inherent limitations. On the one hand, *it is challenging to generate diverse and semantically-valid test inputs*. Current work either generates test inputs on source-code-level through compilation, which greatly reduces diversity [28], or only performs syntax-correct mutations on bytecode-level without considering semantic validity [45, 56, 59]. It is hard to find a balance between these two requirements. On the other hand, *it is challenging to achieve automated bug identification and root cause localization for identified EVM inconsistencies*. Current approaches directly provide EVM developers with contracts that trigger inconsistencies, which brings a huge time and manpower burden to their subsequent debugging work.

This work. We present OPDIFFER, an opcode-level differential testing framework for EVM, which can *generate semantically-valid and diverse test inputs* for differential testing and *automatically identify EVM implementation bugs with the corresponding root causes inside EVM*. We take advantage of Large Language Models (LLMs) with static analysis methods to extract opcode definitions from the specification, applying control-flow-oriented mutation and argument-oriented mutation to generate opcode-level test inputs. With collected results from differential testing, we focus on three critical metrics to identify bugs and propose a root cause localization algorithm leveraging LLMs to link the bug back to its corresponding implementation within the EVM. We have evaluated the effectiveness of OPDIFFER on nine EVM implementations covering multiple scenarios. Compared with state-of-the-art baselines, OPDIFFER increases code coverage by 71.06%, 148.40% and 655.56%, respectively. Moreover, OPDIFFER has successfully uncovered 26 distinct bugs, 22 of which have been confirmed, and three of which are assigned CNVD IDs.

Contribution. We make the following contributions in this work:

- We present OPDIFFER, an opcode-level differential testing framework for EVM, which can generate semantical-valid and diverse test inputs by taking advantage of LLMs and static analysis.
- We propose an automated bug identification and root cause localization algorithm, leveraging LLMs to locate the faulty implementations in EVMs to facilitate risk assessment and bug patch.
- OPDIFFER has identified 26 unknown bugs from nine EVM implementations, among which 22 have been confirmed or patched with our timely disclosure. Additionally, three vulnerabilities have been assigned with CNVD IDs.
- Our evaluation of real-world Ethereum contracts indicates that opcodes with faulty implementations are widespread (7.21% of existing contracts) within Ethereum mainnet.

2 Background

2.1 Ethereum Virtual Machine

Ethereum Virtual Machine. The Ethereum Virtual Machine (EVM) serves as the runtime environment for executing Ethereum smart contracts. Designed as a stack-based virtual machine, the EVM executes bytecode at a low level. The architecture of the EVM encompasses several foundational concepts that are essential to Ethereum’s execution[62]:

- *Stack*. The EVM operates with a stack-based architecture, maintaining a stack of up to 1024 elements, each represented as a 256-bit word.
- *Memory*. The memory model of EVM is a volatile, word-addressable byte array, with all memory locations initially defined as zero.
- *Gas*. To prevent network abuse and address challenges inherent to Turing completeness, all computations in the EVM incurs fees, measured in units of gas. The gas fees are charged on opcode level, whose price may fluctuate according to the concrete behavior of the opcode. For instance, the EVM opcode BALANCE has a variable gas cost: if the address queried by BALANCE has already been accessed within the same transaction, the gas cost is 100; otherwise, a higher dynamic gas charge of 2600 applies.
- *Storage*. The storage in EVM is a persistent key-value store which is part of the Ethereum state, with all locations initially set to zero. Storage can be read and written using the specific opcodes, like SLOAD and SSTORE.
- *Code*. The bytecode of the smart contract to be executed is stored in an immutable ROM within EVM, accessible only through specialized opcodes (e.g., CODESIZE, CODECOPY).
- *Program Counter*. The program counter (pc) in the EVM indicates the position of the next opcode to be executed. For opcodes that include operands (e.g., PUSH1 0x60), the pc is incremented by $1 + x$, where x represents the number of operands.

Ethereum Virtual Machine Opcodes. An EVM opcode is an atomic instruction that the EVM can interpret and execute. Opcodes are building blocks of smart contract bytecode and are used to perform various operations within the EVM. Each opcode is represented by a single byte (8 bits), which allows for a range of 256 possible opcodes (from 0x00 to 0xFF in hexadecimal). As of October 2024, Ethereum had undergone its latest mainnet upgrade, known as Dencun [25], which has expanded the EVM to include 149 opcodes.

Ethereum Specifications. The Ethereum network is a continuously evolving decentralized system. To effectively test the security of the EVM, it is essential to understand its design and security features based on Ethereum specifications. Currently, the primary specifications for Ethereum include *The Yellow Paper*, *Ethereum Improvement Proposals (EIPs)*, and *Ethereum Execution Client Specifications*.

- *The Yellow Paper*. The Ethereum Yellow Paper [62] provides a mathematically rigorous formal specification of the Ethereum protocol and the EVM execution model. While its mathematical formalism ensures precision, it presents comprehensive challenges for programmers and has not been updated beyond the Shanghai fork of April 2023 [25].
- *Ethereum Improvement Proposals*. Ethereum Improvement Proposals (EIPs) [13] are standards that define potential new features or processes for the Ethereum network. Once an EIP is accepted and implemented, it may modify or extend the specifications presented in the Yellow Paper, ensuring that the formal documentation remains consistent with protocol developments.
- *Ethereum Execution Layer Specifications*. For the purpose of providing a more programmer friendly and up-to-date specification of Ethereum than the traditional Yellow paper [60], the Ethereum Execution Layer Specification (EELS) [20] was introduced. The EELS serves not only as an EVM implementation, but also as a specification of the core components of an Ethereum execution client. It is written in Python, with a focus on clarity and readability. EELS provides a current and detailed description of the Ethereum execution client specifications, integrating the latest protocol updates to ensure continued compatibility with the Ethereum mainnet.

```

1 ++ func expModulus() *big.Int {
2 ++     m := &big.Int{}
3 ++     return m.Exp(big.NewInt(2), big.NewInt(256), nil)
4 ++ }
5
6 func (i *Int) Exp(e *Int) *Int {
7 --     i.Int.Exp(i.Int, e.Int, nil)
8 ++     i.Int.Exp(i.Int, e.Int, expModulus)
9     return i.toI256()
10 }

```

(a) How SealEVM implements the EXP opcode, as well as the patch.

```

1 def exp(evm: Evm) -> None:
2     base = Uint(pop(evm.stack))
3     exponent = Uint(pop(evm.stack))
4     exponent_bits = exponent.bit_length()
5     exponent_bytes = (exponent_bits + Uint(7))
6     charge_gas(
7         evm, GAS_EXPONENTIATION + GAS_EXPONENTIATION_PER_BYTE * exponent_bytes
8     )
9     result = U256(pow(base, exponent, Uint(U256.MAX_VALUE) + Uint(1)))
10    push(evm.stack, result)
11    evm.pc += Uint(1)

```

(b) The specification of the EXP opcode.

Fig. 1. A motivating example about the bug of the EXP opcode in SealEVM.

2.2 Differential Testing

Over the past decade, fuzzing has emerged as an exceptionally effective approach for uncovering software bugs [50]. Fuzzing generally requires a test oracle—a mechanism to identify whether the output is correct or erroneous. In cases where an oracle is not easily available, differential testing comes to use. Specifically, differential testing is used to identify bugs or inconsistencies by comparing the behavior or outputs of multiple implementations of the same functionality. In this approach, the same input is provided to each implementation, and their outputs are compared. Any discrepancies between the outputs can indicate unexpected behavior or potential bugs in one or more of the implementations. Differential testing first requires generating test inputs for target programs to execute. During the execution, the runtime information will be collected for comparison and further bug report. Differential testing is particularly effective for uncovering semantic or logical bugs that do not exhibit explicit erroneous behaviors, such as crashes or assertion failures.

3 Challenges & Solution

In this section, we will detail the challenges in performing differential testing for EVM by providing motivating examples in §3.1. Then, we will propose our high-level solution in §3.2.

3.1 Motivating Example & Challenges

Given the complexity of the EVM, fully avoiding implementation bugs remains challenging. To illustrate this, we present the following motivating example. Fig. 1a shows a bug (as well as its patch) in SealEVM [51], a custom third-party EVM implementation. Specifically, this bug is a mathematical calculation related bug within the implementation of opcode EXP. When executing the opcode EXP with crafted parameters, SealEVM will be stuck, disrupting the normal execution of subsequent opcodes. The root cause is that SealEVM lacks the modulus implementation defined in the specification, as shown at Line 9 of Fig. 1b. Triggering this bug requires 1) a deep understanding of the semantics of the EXP opcode in the specification, and 2) careful construction of the corresponding parameters as input.

<pre>def jump(evm: Evm) -> None: jump_dest = Uint(pop(evm.stack)) charge_gas(evm, GAS_MID) if jump_dest not in evm.valid_jump_destinations: raise InvalidJumpDestError evm.pc = Uint(jump_dest)</pre>	<pre>// bytecode 602056...6080 1 PUSH1 20 // Counter 3 JUMP // Jump to counter ... 20 PUSH1 80 // Not a JUMPDEST</pre>
(a) The specification of the JUMP opcode.	(b) A semantically invalid case.

Fig. 2. An example for illustrating a semantically-invalid case.

Table 1. A comparison of existing differential testing tools for EVM, where **Level** and **Approach** refer to their test inputs generation methods. The ●, ◐, and ○ symbols represent full, partial, and no semantic validity or diversity, respectively.

Tool	Level	Approach	Challenge #1		Challenge #2	
			Validity	Diversity	Bug Identification	Root Cause Localization
EVMFuzzer	source code	compilation	●	○	opcode length, gas	-
NeoDiff	bytecode	template	○	◐	type-hash, state-hash	Opcode
FuzzyVM	bytecode	template	○	◐	state transition	-

In summary, conducting differential testing against different EVM implementations presents significant challenges, which can be summarized into two aspects in Table 1.

Challenge #1: Generating test inputs that satisfy both diversity and semantic validity requirements is challenging. Testing the EVM implementation first requires the generation of test inputs. Currently, *source-code-level* and *bytecode-level* test input generation exist, which suffer from this challenge.

Specifically, source-code-level methods require source code as inputs to generate more test inputs. However, leveraging source code for testing has certain limits. According to the prior work of P. Ma *et al.* [43], only 2% of the contracts are open-source. In addition, our analysis of recently verified open source contracts (Etherscan dataset at §5.3) has shown that 88.42% contracts are ERC20 contracts. Those ERC20 contracts represent real-world business logic, and attackers are unlikely to open source malicious contracts or share their bytecode. This limits the characterization of malicious behaviors and hinders the effective testing of EVM implementations against corner cases. In other words, *the diversity of test inputs generated from source-code-level methods cannot satisfy the real-world requirements.*

As for bytecode-level methods, they directly take contract bytecode as inputs and perform mutation to enlarge the diversity of the test input corpus. However, the mutation only considers the syntax correctness without semantic constraints. For example, Fig. 2b illustrates a JUMP related bytecode snippet. Though it meets the syntax correctness, like stack balance, it does not consider that *each JUMP requires a corresponding JUMPDEST* as its semantic validity requirement, which is illustrated in Fig. 2a. Consequently, *the bytecode-level methods would generate test inputs with semantic invalidity.*

Challenge #2: It is challenging to achieve automated bug identification and root cause localization for EVM inconsistencies. Though differential testing does not require oracles, identifying the specific bug and localizing the corresponding root cause in EVM is still challenging. Specifically, different EVM adopts different programming languages and component architectures in their implementations. Moreover, they may behave distinctively on the same metric. For example, due to language-specific features, EVM implementations handle errors differently across platforms. Consequently, *identifying if bugs, i.e., discrepancies, really exist by comparing EVM behaviors is difficult.* As for the root cause localization, it also puts forward further requirements on the scalability of the method. Specifically, only providing bytecode test input is of limited help to developers, because they do not know which opcode or which logic problem in the EVM causes

this inconsistency. However, as mentioned above, different EVMs adopt different programming languages and architectural designs, leading to the challenges in locating the root cause.

Limitations of current tools. To the best of our knowledge, only a few tools are available for testing the EVM, *i.e.*, EVMFuzzer [28], NeoDiff [45], and FuzzyVM [56, 59]. We underline all existing tools have certain limitations.

Specifically, EVMFuzzer is the first differential testing tool for EVM, utilizing source codes of smart contracts as seeds and employing 8 predefined mutators to generate new contracts. Subsequently, these contracts are compiled into bytecode for execution within the EVM. Intuitively, its method is limited by **Challenge #1**. Moreover, it requires huge manual effort to confirm bugs and localize root causes, *i.e.*, the ones stated in **Challenge #2**.

As for NeoDiff, it is a feedback-guided differential fuzzing framework for EVMs. It directly generates contract bytecode based on predefined templates and mutates them with feedback information. Due to the use of human-crafted templates, the generated bytecode exhibits similar structures, leading to insufficient diversity (**Challenge #1**). Moreover, the excessive length of the NeoDiff generated bytecode complicates the process of bug identification and it is limited to reporting only the opcode responsible for the inconsistencies, lacking a comprehensive analysis of their underlying causes as discussed in **Challenge #2**.

FuzzyVM [59], maintained by developers from the Go-Ethereum Team [21], generate test inputs based on pre-defined templates and performs fuzzing on Geth. The actual execution is handled by Go evmlab [56], which facilitates differential testing with other EVM implementations. Specifically, FuzzyVM conducts state transition testing to find EVM bugs. State transition tests [15] first define the pre-state and execution environment of the EVMs, then execute a sequence of transactions to achieve a post-state. However, since the state test requires a well-defined execution environment setting and complicated transaction sequences, expert knowledge is necessary to write state tests effectively, which reflects the issues discussed in **Challenge #1**. Besides, FuzzyVM requires manual effort to review inconsistencies, which also faces the **Challenge #2**.

3.2 Solution

In response to these two challenges, we propose the following ideas: To tackle **Challenge #1**, we propose the concept of *opcode-level test inputs generation*. It refers to generating short snippets for each opcode based on specifications. To ensure the semantic validity and diversity of generated test inputs, we combine static analysis with Large Language Models (LLMs). Leveraging the advanced capabilities of LLMs in natural language understanding and the ICFG parsed by static analysis, as well as the *control-flow-oriented* and *argument-oriented* mutation strategies, we can generate semantically valid and diverse inputs tailored for differential testing. Regarding **Challenge #2**, we instrument the target EVMs and introduce three metrics designed to evaluate EVM implementation consistency for automated bug detection. For confirmed bugs, we propose a root cause localization algorithm leveraging LLMs to locate the responsible implementation in the source code.

4 Approach

This section presents the design of our framework, dubbed as OPDIFFER.

4.1 Overview

Fig. 3 illustrates the workflow of OPDIFFER. As we can see, it can be divided into three phases, *i.e.*, *LLM-assisted test input generation* (§4.2), *differential testing* (§4.3), and *bug identification & root cause localization* (§4.4). Specifically, OPDIFFER takes EELS (the specification for each opcode written in Python, see §2.1) as inputs. Then, OPDIFFER will generate test inputs, while considering diversity and semantical validity, with the help of LLMs and static analysis. Before conducting the

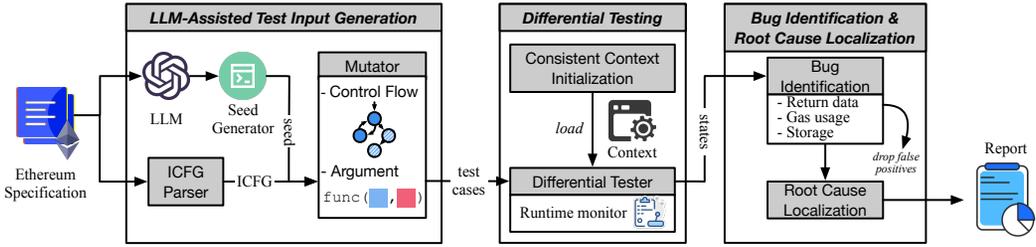


Fig. 3. The workflow overview of OPDIFFER.

differential testing, OPDIFFER will initiate the execution context for all EVMs. After the execution, the results and runtime data from instrumented EVMs will be parsed in a uniform format for bug identification, which will be further utilized for root cause localization. At last, OPDIFFER will output the corresponding bug reports.

4.2 LLM-Assisted Test Input Generation

Instead of using the existing source code of smart contract, which intertwines the implementation correctness of the compilers as described in **Challenge #1**, we employ a generation-based method to directly generate EVM bytecode for testing. Initially, we employed template-based methods to generate test inputs derived from the Ethereum Execution Layer Specification (EELS). However, this approach requires significant human effort to read the specification and manually design the templates. Considering the presence of code comments and definitions within specifications and the recent advancements demonstrated by Large Language Models (LLMs) in code generation and natural language processing tasks [31, 38, 40, 55, 64], LLMs offer an effective approach to interpret and generate test inputs based on opcode specifications. In order to improve the performance of LLMs on this specific type of task, we take advantage of the EELS to construct prompts. This phase can be further divided into four steps.

4.2.1 Step I: Opcode Seeds Construction. To facilitate the test input generation, we leverage LLMs to build initial seeds for each opcode. Specifically, we first explored the GitHub repository of EELS¹ [20], where the semantics of each opcode is defined and written in Python functions. For each opcode, we generate prompts to create the corresponding seed generators. Fig. 4 is a prompt used for constructing a seed generator for the EVM opcode BYTE. As we can see, the prompt consists of context, task description, output requirements, additional notes, and the opcode definition from EELS. Fig. 5 presents an example seed generator of the opcode BYTE opcode generated by LLMs. The seed generator can generate a bytecode sequence, *i.e.*, *seed*, that correctly invokes the BYTE opcode by placing valid parameters on the stack, ensuring the semantical validity.

Existing research [18] has demonstrated that GPT-4 could achieve high accuracy in Python code generation tasks, particularly for short code snippets. Our evaluation indicates that the length of the seed generator is an average of 14 lines of code. Furthermore, our experimental results validate GPT-4’s high accuracy, underscoring the feasibility of this approach.

4.2.2 Step II: Specification-Based ICFG Extraction. Though Step I illustrates that directly taking the EELS as input to construct seeds is feasible, since there may be function calls within the EELS (such as the `charge_gas` call at Line 4 in Fig. 4), considering only the specification function defined in EELS cannot cover its control flow completely. For example, if external function calls are not taken into consideration, only two paths of opcode BYTE will be generated, as shown in Fig. 6a. However, callee may introduce additional control flows, if these are overlooked, it can prevent the

¹EELS defines the EVM opcodes in the `src/ethereum/fork/vm/instructions` folder.

Prompt 1: Opcode seed generator construction

System: You are an expert in Ethereum Virtual Machine (EVM).

Task: Given the following opcode specification, you should write a Python function that generates executable bytecode test case for the given opcode.

Requirements:

1. The function should create a valid bytecode sequence, including any required push operations and memory operations for operands.
2. The function should also add the bytecode corresponding to the opcode itself.
3. Use random to generate random values where necessary, such as for operand lengths.
4. The function should return the bytecode as a hex string.

Note: There is no need to explain your answer, just return the Python code.

Input: EVM opcode BYTE and its specification.

```

1  def get_byte(evm: Evm) -> None:
    """
    For a word (defined by next top element of the stack), retrieve the
    Nth byte (0-indexed and defined by top element of stack) from the
    left (most significant) to right (least significant).
    evm : The current EVM frame.
    """
    # STACK
2  byte_index = pop(evm.stack)
3  word = pop(evm.stack)
    # GAS
4  charge_gas(evm, GAS_VERY_LOW)
    # OPERATION
5  if byte_index >= 32:
6      result = U256(0)
7  else:
8      extra_bytes_to_right = 31 - byte_index
        # Remove the extra bytes in the right
9      word = word >> (extra_bytes_to_right * 8)
        # Remove the extra bytes in the left
10     word = word & 0xFF
11     result = U256(word)
12     push(evm.stack, result)
    # PROGRAM COUNTER
13     evm.pc += 1

```

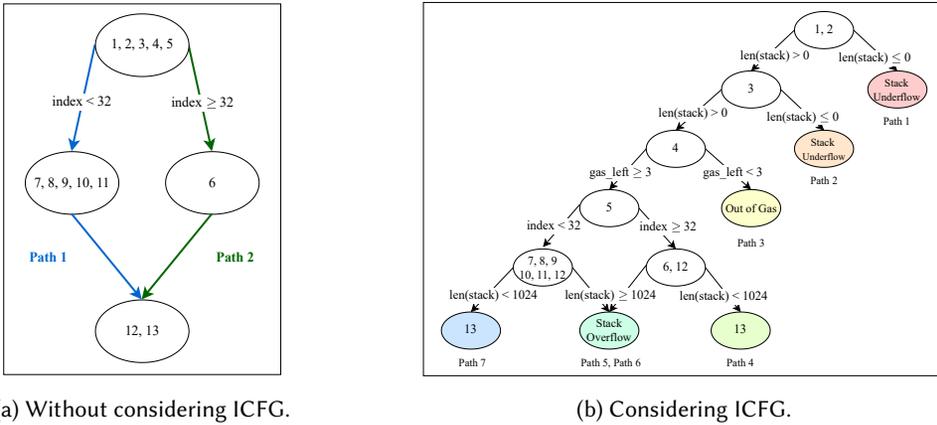
Fig. 4. Prompt for generating seed generator of the BYTE opcode.

generation of test inputs that cover the certain control paths each opcode may execute at runtime. As shown in Fig. 6b, the pop will introduce an additional control flow if the stack is empty, and the charge_gas will revert if the given gas has been exhausted. Therefore, we parse and extract the inter-procedural control flow graph (ICFG) for each opcode's specification, which could capture the control flow relationships across different functions.

4.2.3 Step III: Control-Flow-Oriented Mutation. To generate test inputs with diversity and semantical validity, we utilize the reasoning and inferential capabilities of LLMs to mutate the existing seeds to cover more paths defined in the control flow of opcode specification. Specifically, to ensure that the test inputs generated by LLMs remain valid and executable, we leverage the previous seeds from §4.2.1 to construct the input prompts. These seeds produced by seed generators serve as concrete examples that guide the LLMs in producing test inputs that adhere to the expected

```
def generate_byte_bytecode() -> str:
    # Initialize an empty bytecode array
    bytecode = bytearray()
    # Generate and push a random value (1 to 32 bytes) onto the stack
    length1 = random.randint(1, 32)
    push_bytecode_hex1 = generate_push_bytecode(length1)
    push_bytecode1 = bytes.fromhex(push_bytecode_hex1)
    bytecode.extend(push_bytecode1)
    # Push a byte index (0x0 to 0xff) onto the stack
    length2 = 1
    push_bytecode_hex2 = generate_push_bytecode(length2)
    push_bytecode2 = bytes.fromhex(push_bytecode_hex2)
    bytecode.extend(push_bytecode2)
    # Append the BYTE opcode (0x1a)
    bytecode.extend(b"\x1a")
    return bytecode.hex()
```

Fig. 5. A generated seed generator for the BYTE opcode.



(a) Without considering ICFG.

(b) Considering ICFG.

Fig. 6. The control flow of BYTE in EVM, where the specification and line numbers in Fig. 4.

structure and specifications, ensuring semantical validity. Considering that LLMs may struggle with interpreting hexadecimal bytecode seed, we convert the seed into its mnemonic representation to enhance the LLM’s comprehension of the example. The input of the prompt consists of the opcode name, an example bytecode seed in both its hexadecimal and mnemonic forms, and an inter-procedural control flow graph (ICFG). Leveraging their reasoning and inferential capabilities regarding control flow within the ICFG, LLMs are expected to mutate the seeds and output a list of valid bytecode, tailored for execution within the EVM environment. An example prompt for control-flow-oriented mutation is shown in Fig. 7.

4.2.4 Step IV: Argument-Oriented Mutation. Considering that reaching certain branches in the control flow paths outlined in the opcode specification requires operands to be boundary values (For example, Path 4 in Fig. 6b), LLMs may fail to mutate seeds to cover all the paths in the CFG. Moreover, relying solely on LLMs to produce diverse bytecode is economically expensive. Therefore, we propose an argument-oriented mutation strategy to mutate the arguments in the mutated seed to introduce diversity and randomness for differential testing. The Algorithm 1 illustrates the details of argument-oriented mutation strategy. Specifically, as shown in Line 15, Line 17 in Algorithm 1, we apply mutations to the operands following the PUSHX opcodes by replacing them with their boundary values or random values with a probability, where X is the byte length. This approach aims to introduce diversity in operand values, increasing the likelihood of reaching edge cases in the execution paths.

Prompt 2: Opcode control-flow-oriented mutation

System: You are an expert in Ethereum Virtual Machine (EVM).

Task: You will receive a JSON object with these four fields about an EVM opcode:

Requirements:

1. `opName`: The name of the EVM opcode.
2. `seed`: An example of a valid hex string bytecode seed for the opcode `BYTE`.
3. `seed_mnemonics`: The mnemonic representation of the seed, which is helpful to understand the bytecode.
4. `ICFG`: The inter-procedural control flow graph (ICFG) described in Python, covering all possible paths of the opcode's specification, constructed using the Graphviz dot tool. The bytecode provided in `seed` field represents one possible execution path in the ICFG.

Objective:

1. Generate a comprehensive set of test inputs that cover all possible paths through the opcodes as defined by the provided ICFG.
2. Each test input must be a valid hex string representing valid bytecode. You may refer to the format of the provided example, but you do not need to strictly follow it.
3. Return the test inputs in a comma-separated list format, where each element is the corresponding hex string.

Input format: A JSON object containing the four values above.

Output format: Only a comma-separated list, where each element is a bytecode string executable within the EVM environment.

EXAMPLE OUTPUT:

```
[hexstr_1, hexstr_2, ..., hexstr_n]
```

Note: No explanation is needed, just output the test inputs in a comma-separated list.

Fig. 7. Prompt for implementing control-flow-oriented mutation for opcode `BYTE`.

4.3 Differential Testing

In the second phase, `OpDIFFER` will conduct differential testing by taking the generated test inputs from §4.2 as inputs on EVMs. However, given the target EVMs are implemented in a different manner, we need to provide a consistent context for them. Moreover, instead of only examining the final state for each EVM, we also analyze the intermediate states during the differential testing.

4.3.1 Consistent Context Initialization. To avoid false alarm during differential testing, we initialize all EVMs in a consistent context, including *forks*, *account states*, and *global states*.

- *Forks*. Ethereum adopts *forks* to introduce technical upgrades or changes [25]. In the context of EVM, different forks may represent different instruction sets, and even functionalities. To ensure the compatibility among EVMs, we configure all EVMs under the same fork, *i.e.*, *Cancun* [25].
- *Account states*. Ethereum can be taken as a state machine, where transactions can lead to the changes of permanent data, *e.g.*, account balance. To avoid the distinctions brought by these account states, we initialize a set of accounts under the same addresses with identical balances and embedded codes (*i.e.*, smart contracts).
- *Global states*. Apart from account states, Ethereum has several inherent global attributes, which may affect the state of the whole blockchain, *e.g.*, chainid, block number, and timestamp. Similarly, to ensure the testing consistency, we must unify their values across EVMs.

Algorithm 1: The algorithm of argument-oriented mutation**Input:** *seed* – the EVM bytecode seed to be mutated*p* – the mutation probability*t* – the number of mutation iterations**Output:** *testinputs* – the set of test inputs after mutation

```

1 Function Mutation(seed, p, t):
2   testinputs  $\leftarrow$   $\emptyset$ ;
3   for iteration  $\leftarrow$  0 to t do
4     seedmut  $\leftarrow$  ""; // Initialize empty seed
5     for index  $\leftarrow$  0 to seed.length do
6       opcode  $\leftarrow$  seed[index : index + 2];
7       seedmut  $\leftarrow$  seedmut || opcode;
8       index  $\leftarrow$  index + 2;
9       if opcode  $\in$  PUSHX_SET then
10        byteslen  $\leftarrow$  getByteLen(opcode);
11        index  $\leftarrow$  index + 2  $\times$  byteslen;
12        r  $\leftarrow$  RandomFloat(0, 1);
13        if r < p then
14          if r < p/2 then
15            | operand  $\leftarrow$  00  $\times$  byteslen; // All zeros
16          else
17            | operand  $\leftarrow$  FF  $\times$  byteslen; // All ones
18          else
19            | operand  $\leftarrow$  RandomBytes(byteslen); // Random valid operand
20            | seedmut  $\leftarrow$  seedmut || operand;
21        testinputs  $\leftarrow$  testinputs  $\cup$  seedmut; // Add the mutated seed to the set
22  return testinputs

```

For each round of differential testing, the context should be initialized consistently as we mentioned above. However, we have to argue that different context setting may lead to different behaviors. For example, the Path 3 in Fig. 6b is related to the gas availability, which will be triggered exclusively by insufficient gas allocation. Thus, before each round of testing, we will randomize the context while ensuring its consistency among EVMs. Note that, since not all EVMs support custom context configurations, we had to modify the source code of certain EVM implementations to achieve such a consistency.

4.3.2 Differential Testing. After setting the context, OPDIFFER takes test inputs generated from the §4.2 as inputs and conducts differential testing. Besides the final state, the runtime intermediate states, e.g., stack and memory, are also considered by OPDIFFER for the following bug identification and root cause localization. Inspired by the EIP-3155 [46], we manually record the EVM state before the execution of each opcode in a same format. Specifically, for each opcode, we pay attention to the program counter, the opcode name, remaining gas, consumed gas, stack, memory, and other auxiliary data. This enables us to record necessary runtime information, ensuring comprehensive capture of execution details for each EVM.

4.4 Bug Identification & Root Cause Localization

With collected states from the differential testing in the second phase, we specifically pay attention to some fields that are critical to cause EVM bugs. By leveraging LLMs, we can further pinpoint the corresponding functions in EVMs, reducing the burden of debugging for developers.

4.4.1 Bug Identification. To identify bugs from EVM inconsistencies, we focus specifically on to the following three metrics abstracted from collected states in the second phase.

- *Return data.* The return data inside EVM is primarily utilized for messages calls to return values. It works similarly to a *hash* representing the outcome of execution, directly reflecting inconsistency between EVMs.
- *Gas usage.* As the EVM is designed to use gas to restrict the resource abuse, EVM will charge the gas for each opcode before the operation is actually executed. If the gas usage is inconsistent, it means that either opcode is not implemented according to the specification, or a different control flow is taken by an EVM.
- *Storage.* Storage is one of the permanent data stored on-chain. Intuitively, its changes can also be used as an indicator. In total, six opcodes are related to storage changes, *i.e.*, SSTORE, TSTORE, CREATE, CREATE2, SELFDESTRUCT, and REVERT². Before and after the execution of these six opcodes, we parse and record values in storage to identify if any inconsistency exists.

To ensure that the bug is reproducible, we will reproduce the identified bugs using the previously saved information. Specifically, we will re-execute the test input that triggered the bug on the EVMs with the same configuration as the prior run. In summary, we will extract these three metrics from all intermediate and final states to assess any discrepancies between EVM implementations.

4.4.2 Root Cause Localization. After bugs are identified as mentioned in §4.4.1, it is essential to locate their root causes to reduce duplicates and assist developers in fixing the corresponding issues. Thus, we propose a *root cause localization algorithm*, as shown in Algorithm 2.

Firstly, we compare all values in fields of all states generated from different EVM implementations to locate the specific opcode that triggered the inconsistency (Lines 2 – 4). With the opcode corresponding to the identified bug, we further need to figure out which part of the implementation in EVM is responsible for that. We utilize LLMs to extract the mapping relations between opcodes and their corresponding functions in EVMs (Line 5). Based on our analysis of the specification, we found that the logic in EELS for handling opcodes can be divided in to four phases: *stack*, *gas*, *operation*, and *pc*. Therefore, when inconsistencies are detected, the root cause is identified by pinpointing the specific phase where the discrepancy occurs (Lines 10 – 17). For instance, if the divergence is observed in the stack values, it indicates the issue lies in the implementation on handling the EVM stack. Finally, the responsible opcode, its corresponding function in EVMs, as well as the root cause identified by Algorithm 2 are returned.

5 Implementation and Evaluation

We implemented OPDIFFER with Python, building on top of EVMFuzzer. To parse the ICFG from the Ethereum Execution Layer Specification, we use Scalpel [37], a static analysis framework for Python. For the integrated LLM, we choose the latest version of GPT-4o, the state-of-the-art LLM from OpenAI. Our experiments aim to answer the following four research questions:

RQ1 How is the performance of OPDIFFER compared to baselines?

RQ2 How many real-world bugs in EVM implementations can be identified by OPDIFFER, and what are their root causes?

²Please refer to [53] for their detailed explanations.

Algorithm 2: Root Cause Localization Algorithm

Input: *testinput*: the EVM test input that triggered inconsistency
sc: the source code of the EVM

Output: *opcode*: the opcode responsible for the inconsistency
func: the function implementing the opcode
cause: the root cause

```

1 Function RootCauseLocalization(testinput, sc):
2   cause ← "";
3   runtime_info ← Execute(testinput) ;           // Get runtime information
4   opcode ← Compare(runtime_info) ;             // Get the responsible opcode
5   funcMap ← LLM4ExtractFunc(opcode, sc) ;    // Extract opcode-function map
6   foreach pair ∈ funcMap do
7     if pair.key is opcode then
8       func ← pair.value;
9       diff ← getDiff(runtime_info, opcode) ;    // Compare runtime information
10      if diff found in stack then
11        | cause ← cause || "stack handling implementation";
12      if diff found in gas then
13        | cause ← cause || "gas handling implementation";
14      if diff found in memory then
15        | cause ← cause || "operation execution implementation";
16      if diff found in pc then
17        | cause ← cause || "program counter handling implementation";
18      return (opcode, func, cause)

```

RQ3 What are the characteristics and the real-world impacts of the detected bugs?

RQ4 How does each component of OPDIFFER contribute to its code coverage?

Experiment Setup & Baseline Selection. All experiments were performed on a desktop running Ubuntu 20.04 with an Intel i7-12700 CPU and 128GB RAM. We selected the state-of-the-art EVM testing frameworks, EVMFuzzer [28], NeoDiff [45] and FuzzyVM [59] as baselines for our evaluation. For test input generation, EVMFuzzer operates at the source-code-level to produce bytecode for EVM execution, which faces challenges in §3.1. In addition, NeoDiff generates bytecode-level test inputs using predefined templates, but similarly struggles with a lack of diversity, as discussed in §3.1. FuzzyVM generates state test inputs using different strategies, which are also limited in diversity, as illustrated in §3.1.

Targeted EVMs. To systematically test the EVM implementations, we collected 49 EVM implementations in the Ethereum ecosystem to the best of our efforts. Out of them, we choose three categories of EVM as our targets, which together cover more than 97% execution clients on the Ethereum mainnet [2] and include all officially recommended standalone EVM implementations [24]. Besides, to test the completeness of minority third-party EVM implementations, we also select two relatively niche EVMs for analysis. Our target EVMs can be divided into three categories, *i.e.*, *Ethereum mainnet execution client's EVM*, *Ethereum officially recommended standalone EVM*, and *Third-party custom EVM*, as shown in Table 2. All these EVMs are actively maintained by developers, with their most recent commits pushed within the past two months.

Table 2. Overview of the selected EVM implementations.

Category	EVM Project	Language	Github Stars	Version
execution client	Geth [21]	Golang	47.4k	1.14.9
	Besu [33]	Java	1.5k	24.8.0
	Nethermind [47]	C#	1.3k	1.29.1
standalone EVM	Py-EVM [22]	Python	2.3k	0.8.0b1
	evmone [19]	C++	867	0.11.0
	ethereumjs [14]	TypeScript	2.6k	2.1.0
	revm [1]	Rust	1.6k	0.9.0
custom EVM	SealEVM [51]	Golang	16	0.3.0
	Chainmaker [48]	Golang	3	2.3.3

Table 3. Code coverage results for 12 hours.

Tool	Geth (All)	Geth (Opcode Funcs)	SealEVM (All)	SealEVM (Opcode Funcs)
OPDIFFER	54.40%	91.85%	90.10%	89.87%
EVMFuzzer	31.80%	44.12%	56.60%	57.38%
NeoDiff	21.90%	17.98%	N/A	N/A
FuzzyVM	7.20%	1.43%	N/A	N/A

5.1 RQ1: Comparison with Baselines

To compare the performance of OPDIFFER with baselines, we mainly focus on their code coverage on EVMs. As Geth is the only EVM supported by all tools, we first take Geth into consideration. As OPDIFFER is implemented based on EVMFuzzer, thus both of them support SealEVM. Finally, against Geth and SealEVM, we ran OPDIFFER, EVMFuzzer, NeoDiff and FuzzyVM for 12 hours. As both targeted EVMs are implemented in Golang, we take advantage of its official code coverage tool [29]. We measured not only the code coverage of the EVM as a whole, but also the specific functions responsible for opcode processing. We recorded the code coverage on an hourly basis, where the results are shown in Fig. 8 and Table 6.

Overall Results. As shown in Table 6, when considering Geth/evm as a whole, OPDIFFER achieves a code coverage of 54.4%, surpassing the coverage of EVMFuzzer, NeoDiff, and FuzzyVM by 71.06%, 148.40% and 655.56%, respectively. When only considering the opcode-related functions, OPDIFFER can cover 91.85% of statements, which is 108.18%, 410.85% and 6323.08% more than EVMFuzzer, NeoDiff and FuzzyVM, respectively. FuzzyVM reached the lowest coverage for Geth that only 1.43% of the lines in the opcode implementation functions were touched. This may be attributed to its inability to generate semantically valid and diverse test inputs. In the case of SealEVM, the results are similar. Compared to EVMFuzzer, OPDIFFER achieves 59.18% and 56.62% higher code coverage when considering SealEVM as a whole or only opcode-related functions. The reason behind such a significant improvement is that OPDIFFER proposes an opcode-level test input generation approach based on the EVM specification, which can generate semantically valid and diverse test inputs with LLM’s assistance. In contrast, due to EVMFuzzer generates bytecode through compilation, while NeoDiff and FuzzyVM utilizes templates-based approach for test input generation; these tools struggle to produce semantic-valid and diverse test inputs capable of covering certain statements in EVM implementations.

Coverage Trend. We measured the code coverage on an hourly basis. After twelve hours, the coverage results of all tools eventually converge, which are displayed in Fig. 8.

As shown in Fig. 8a, OPDIFFER achieves higher coverage than EVMFuzzer, NeoDiff and FuzzyVM throughout the entire testing process. The coverage of other tools has converged within less than one hour, indicating that existing tools fail to test the Geth implementations of EVM opcodes due to

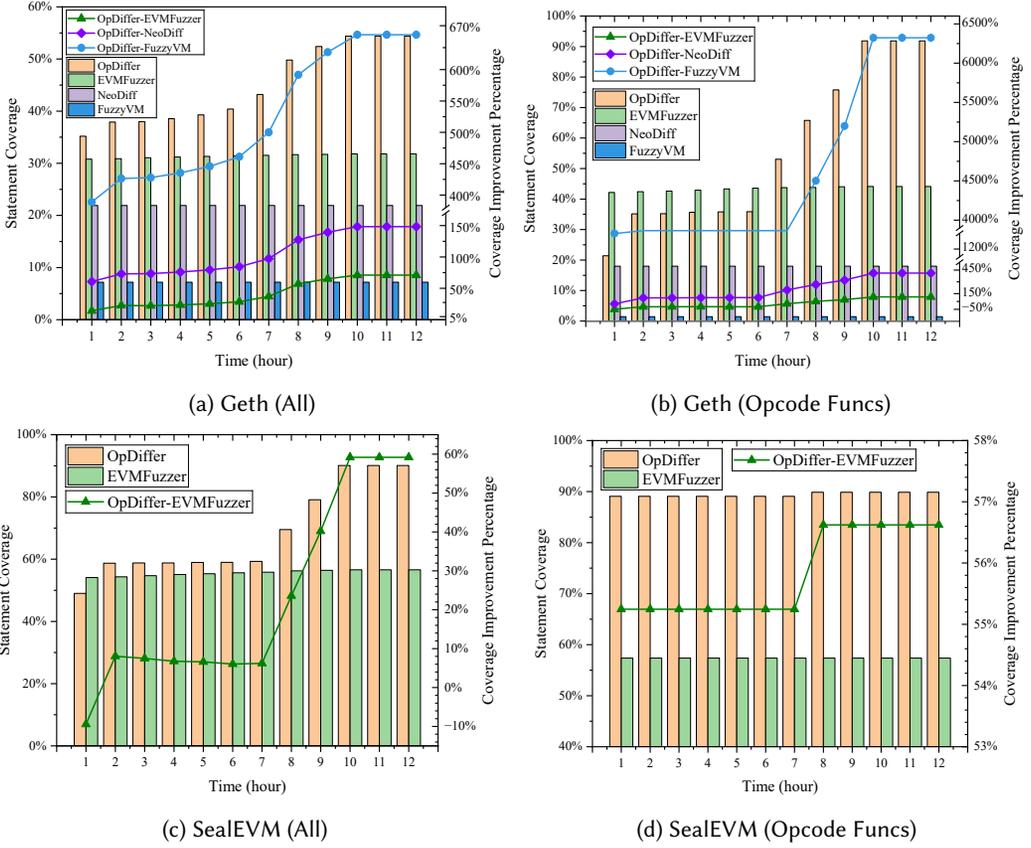


Fig. 8. Coverage results of OPDIFFER, EVMFuzzer, NeoDiff and FuzzyVM on Geth and SealEVM. **Bars** represent the achieved code coverage, while **lines** represent the coverage improvement by which OPDIFFER exceeds EVMFuzzer, NeoDiff and FuzzyVM.

lack of diversity of test inputs. Although baselines may generate more test inputs than OPDIFFER, their generated inputs cannot further improve the coverage across target EVM implementations. In contrast, OPDIFFER generates more diverse and semantically valid test inputs, resulting in its increase in coverage even after eight hours. In addition, Fig.8b demonstrates OPDIFFER’s coverage of opcode-related functions compared to baselines. While EVMFuzzer initially shows higher coverage by testing multiple opcodes simultaneously through compiler-generated bytecode, OPDIFFER’s depth-first strategy focuses on thorough testing of individual opcodes. Despite a lower initial coverage in the first five hours, OPDIFFER ultimately achieves 59.18% and 56.62% higher coverage than EVMFuzzer after 12 hours on SealEVM, as shown in Fig.8c and Fig. 8d.

Answer for RQ1: Compared with three state-of-the-art baselines, OPDIFFER achieves at most 148.40% and 410.84% coverage improvement when considering the EVM as a whole and only the opcode-related functions, respectively. The continuous growth of the code coverage proves the effectiveness of our proposed test input generation method.

Table 4. Details of all identified bugs of OPDIFFER, **Qty** is the number of affected opcodes, **Opcode** is an example affected opcode. Symbols ●, ◐, and ○ represent bugs assigned CNVD/CVE IDs, currently under review, or not reported, respectively.

No.	Project	State	Root Cause	Opcode	Qty	Issue	CNVD
1	Geth	Fixed	Network fork	PUSH0	6	#29782	○
2		Fixed	Instruction set	PREVRANDAO	1	#29722	◐
3		Fixed	Address setting	BALANCE	1	#30254	○
4		Fixed	Fee setting	BASEFEE	1	#30279	○
5	Besu	Fixed	Gaslimit setting	GASLIMIT	1	#7432	○
6		Fixed	Fee setting	BASEFEE	1	#7433	○
7		Fixed	Tracing	REVERT	1	#7608	○
8		Fixed	Entry point	N/A	N/A	#7430	○
9	Nethermind	Confirmed	Instruction set	PUSH0	6	#7629	○
10	EthereumJS	Confirmed	Address setting	N/A	N/A	#3236	●
11		Will fix	Memory tracing	N/A	N/A	#3561	○
12	Py-EVM	Fixed	Missing field	SELFDESTRUCT	1	#2176	●
13		Reported	Fee setting	N/A	N/A	#2162	●
14	revm	Confirmed	Fee setting	N/A	N/A	#1702	○
15		Reported	Address setting	N/A	N/A	#1704	○
16	EELS	Reported	Memory	N/A	N/A	#981	○
17		Reported	Description	N/A	N/A	#992	○
18	SealEVM	Will fix	Fee setting	N/A	N/A	#18	○
19		Fixed	Stack	CREATE2	1	#21	○
20		Fixed	Pc manipulation	N/A	N/A	#27	◐
21		Fixed	Memory	MSIZE	N/A	#28	○
22		Fixed	Storage	EXTCODESIZE	N/A	#30	○
23		Fixed	Math	EXP	6	#31	◐
24		Fixed	Stack	CREATE2	1	#1092	○
25	ChainMaker	Fixed	Pc manipulation	N/A	N/A	#1140	◐
26		Fixed	Math	EXP	6	#1173	◐

5.2 RQ2: Real-World EVM Bugs

We have shown that OPDIFFER can achieve higher coverage than the state-of-the-art baselines. We will further explore the capability of OPDIFFER in detecting real-world EVM bugs.

In total, OPDIFFER has identified 26 bugs, whose detailed information is shown in Table 4. To assist further investigation, we prepare detailed and reproducible steps and potential root causes for every identified bug. We collect all identified bugs as well as the output reports and report them to the corresponding EVM developers. As we can see, of the 26 identified bugs, 22 of them have been confirmed by the developers already, and 17 have been fixed at the time of writing. Additionally, the developers have indicated plans to fix two of the remaining issues in future updates. We further conducted manual analysis to identify if there are security vulnerabilities. Among the 26 confirmed bugs, three vulnerabilities have been assigned with China National Vulnerability Database (CNVD) IDs, while five of them are still under review by CNVD and NVD. From Table 4, we can observe that: 1) Regardless of which of the three categories the EVM falls into, bugs affect them uniformly. 2) Developers have an active attitude towards the bugs identified by OPDIFFER, as they promptly confirmed and fixed the bugs with our assistance. We attribute this responsiveness to the comprehensive and reproducible bug reports provided, which streamline the debugging and

Table 5. Statistics of two datasets that are used to evaluate the real-world impact for identified buggy opcodes.

Dataset	#Contracts	#Opcodes	#Buggy Opcodes	Buggy Rate
Gigahorse	10,000	55,777,530	1,874,626	3.36%
Etherscan	10,000	58,979,456	4,253,359	7.21%

validation processes for developers. 3) Execution client EVM implementations have the fewest security-related bugs due to extensive testing before their deployment. In contrast, standalone and custom EVMs remain vulnerable to several bugs, underscoring the critical importance of rigorous testing across all EVM implementations.

Answer for RQ2: Out of nine Ethereum mainnet execution client EVMs, standalone EVMs, and third-party custom EVMs, OPDIFFER can successfully identify 26 bugs. With our timely disclosure, 22 bugs have been confirmed, while 3 of them may lead to severe impacts and have been assigned CNVD IDs.

5.3 RQ3: Bug Characterization

To further illustrate the impact of identified bugs on blockchain systems, we assess the real-world consequences of these EVM bugs by analyzing smart contracts from the Ethereum mainnet. Additionally, we present case studies for two critical identified bugs.

Real-world impact. Out of 26 identified bugs, there are 21 opcodes that are responsible for them, accounting for 14.1% of all valid opcodes defined in the specification. To evaluate the real-world impact of our detected bugs, we assume that if a contract contains one of the buggy opcodes, when it is executed on the corresponding EVM with specific context, the bug can be triggered. We have to argue that this rate is the upper bound of real-world impacts of detected EVM bugs as there is still much effort left to exploit the vulnerability. However, an attacker could deploy malicious contracts containing buggy opcodes on the mainnet and send specific transactions designed to trigger the associated bugs in the affected EVM implementations. Such an attack could disrupt normal clients by causing issues during block synchronization and the execution of transactions.

Specifically, we collected two datasets. One is from Gigahorse [30], consisting of 10,000 compiled contract bytecode, while another is 10,000 recently verified contracts from Etherscan [27], as certain buggy opcodes are introduced in recent Ethereum forks, like PUSH0 and MCOPY. The second dataset was compiled using the latest version of the Solidity compiler (solc 0.8.28). Our analysis indicates that 88.42% of the contracts in this dataset conform to the ERC20 standard, which reflects the homogeneity of the source-code-level test input generation methods.

As presented in Table 5, our experimental results indicate that 1,874,626 and 4,253,359 opcodes respectively in the two datasets could be affected by our identified EVM bugs. The corresponding rates of affected opcodes are 3.36% and 7.21%, respectively. This evaluation demonstrates that the opcodes with EVM implementation bugs we detected are widely distributed across the Ethereum ecosystem, highlighting their widespread existence in real-world smart contracts. Although the proportion of potentially affected opcodes is currently around 3.36% on the Ethereum mainnet, this rate could increase when new compiler features are enabled and become more widely used.

Case #1. The first case is the 12th entry in Table 4, which has been assigned with CNVD-2024-40890. This is a bug of Py-EVM 0.8.0b1, which is a Python implementation of EVM. It is related to the Py-EVM implementation for opcode SELFDESTRUCT, as shown in Fig. 9. As we can see, before executing the opcode, Py-EVM will first look up the implementation function `opcode_fn` for the specific opcode (Line 2). However, because in the function for SELFDESTRUCT, the field `mnemonic` is not defined yet (Line 7). As a result, Py-EVM raises an error and halts the execution

```

1 for opcode in computation.code:
2     opcode_fn = computation.get_opcode_fn(computation.opcodes, opcode)
3     computation.logger.trace(
4         "OPCODE:_{0:x}_{1}({%s})_{2}|_{3}pc_{4}:%s",
5         opcode,
6 # Bug here: the mnemonic field for the SELFDESTRUCT func is not defined
7         opcode_fn.mnemonic,
8         max(0, computation.code.pc - 1),
9     )
10     ...

```

Fig. 9. Case study #1: the buggy implementation of SELFDESTRUCT in Py-EVM.

```

1 // ...
2 LondonBlock:                new(big.Int),
3 ArrowGlacierBlock:          nil,
4 GrayGlacierBlock:           nil,
5 TerminalTotalDifficulty:    big.NewInt(0),
6 TerminalTotalDifficultyPassed: true,
7 // Bug fix: add the following lines to support newly introduced opcodes
8 MergeNetsplitBlock:         nil,
9 ShanghaiTime:                &shanghaiTime,
10 CancunTime:                  &cancunTime
11 // ...

```

Fig. 10. Case study #2: the buggy implementation of PREVRANDAO in Geth, as well as the patch.

for following opcodes. Attacker could exploit this vulnerability by crafting malicious bytecode, enabling a denial-of-service (DoS) attack on Py-EVM.

Case #2. This case includes the 2nd entry in Table 4, related to the well-known Geth. According to statistics [2], Go-Ethereum (Geth) is the most widely used execution client on the Ethereum mainnet as of October 2024. We find that when executing the opcode PREVRANDAO, Geth/evm triggers a runtime error, indicating an invalid memory address or nil pointer dereference. This issue is caused by an incorrect instruction set configuration, preventing the opcodes introduced in the post-Paris forks from executing correctly. With our detailed reports, the developers confirmed and fixed this bug. As shown in Fig. 10, developers fixed this bug by adding correct configurations for the supported instruction set from Line 8 to 10.

Answer for RQ3: On the one hand, the opcode-related bugs we found are widely present in the real world and can be triggered after setting appropriate parameters for EVMs. On the other hand, we found that these bugs can be triggered by a single opcode, highlighting the necessity of opcode-level testing.

5.4 RQ4: Ablation Studies

To assess the contribution of each component of our differential testing framework OPDIFFER, we performed ablation studies focusing on three key components: the seed generator (Seed), the control flow mutator (CFMut), and the argument mutator (ArgMut). In a series of three independent experiments, we sequentially removed one component at a time and evaluated the impact on code coverage for both Geth and SealEVM. Specifically, in the first experiment, we removed the seed from the seed generator along with its mnemonics (as shown in Fig. 7) and generated test inputs without them. In the second experiment, we removed the ICFG (as shown in Fig. 7) and disabled the control-flow-oriented mutation. While in the third experiment, we disabled the argument mutator, using the raw output of LLMs as test inputs without mutation. The impact of each component was assessed based on the observed code coverage. As shown in Table 3, the results demonstrated that each component positively contributed to the code coverage of the target EVMs. Removing any component led to a reduction in coverage. Next, we will analyze the effect of each component.

Table 6. Code coverage results of ablation Study. "w/o" denotes "without," while "CFMut" and "ArgMut" refer to the control flow mutator and argument mutator, respectively.

Tool	Geth (All)	Geth (Opcode Funcs)	SealEVM (All)	SealEVM (Opcode Funcs)	Average Coverage Reduction
w/o Seed	48.30%	66.02%	71.20%	80.83%	-14.96%
w/o CFMut	52.70%	89.50%	88.60%	89.27%	-1.54%
w/o ArgMut	53.20%	90.36%	89.20%	89.28%	-1.05%
OPDIFFER	54.40%	91.85%	90.10%	89.87%	N/A

We first examined the impact of the seed generator. As shown in Table 3, removing the seed generator resulted in the largest drop in code coverage (average code coverage loss of -14.96%), followed by CFMut (-1.54%) and ArgMut (-1.05%). We argue that the seeds and its mnemonics from seed generator provide a structured and well-defined example, enabling the LLMs to reason and generate meaningful test cases. Without the ICFG, the LLMs failed to generate test cases that explore additional execution paths in the opcode definition, leading to a 1.54% reduction in coverage. Finally, we investigated the effect of the argument mutator. Since the argument mutator modifies arguments in the stack to generate random operands, disabling it restricted the exploration of corner cases in the specification, reducing the diversity of test inputs.

Answer for RQ4: Our ablation studies demonstrate that each component of OPDIFFER is essential for enhancing code coverage on target EVM implementations, underlining the necessity of integrating all three modules within OPDIFFER.

6 Discussion

Ethical Consideration. Our differential testing has detected 26 distinct bugs from various EVMs, of which 22 have been confirmed or fixed by developers. To meet the ethical standards, we provide detailed bug reports and reproducible test inputs to developers once after the analysis is done. Moreover, all identified vulnerabilities are reported to both developers and relevant security databases, including the CNVD and NVD, for further disclosure and remediation.

Limitation. 1) *Test input generation.* As we utilize LLM to generate test inputs, the main limitation lies on the LLM's restricted reasoning capabilities due to their intrinsic limitations. To solve this, we can introduce symbolic execution for a more accurate and comprehensive input generation, which is reserved for future work. To mitigate the potential unreliability of LLM's outputs caused by hallucinations, the self-refinement [44] mechanism can be applied to iteratively enhance the consistency and correctness of the output. 2) *Root cause localization.* Once we identified the responsible opcode, determining the underlying root cause for non-execution-stage bugs remains challenging, such as the bugs introduced at the EVM initialization stage. However, the localized responsible opcode can still aid the process by excluding irrelevant opcodes for the following manual inspection. 3) *Opcode specification compatibility.* Ethereum is evolving rapidly, which means that a new set of opcode specifications may be introduced in the future, like EOF [3]. However, we underline that OPDIFFER is scalable. By extracting the semantics of the corresponding specifications, we can equip LLMs with essential domain knowledge, enabling them to autonomously reason and generate effective test inputs. This method ensures that OPDIFFER remains adaptable to the evolving landscape of Ethereum's development.

7 Related Work

Blockchain Security. Much of the existing research on blockchain security has focused on smart contract, with notable contributions from various tools and frameworks designed to identify weaknesses in smart contracts [32, 34, 52, 54, 55, 57, 58]. While substantial advances have been

made in analyzing and detecting vulnerabilities in contract code, there is comparatively less focus on the security aspects of the foundational EVM. EVMFuzzer [28] is the first differential testing tool for EVM. NeoDiff [45] utilizes bytecode-level generation test input generation method to fuzz the EVM. FuzzyVM [59] and go evmlab [56] are official differential testing tools maintained by Go-Ethereum Team. However, as introduced in section §3.1, existing tools are limited in generating semantic-valid, diverse test inputs and lack an effective root cause analysis for identifying potential bugs. Besides, several studies have explored methods for testing blockchain consensus protocols [9, 41, 61, 65] and RPC services [35, 36] in Ethereum execution clients.

Differential Testing. Differential testing has been widely applied in both academia and industry, such as detecting issues in SSL/TLS implementations [5, 8, 49], compilers [31], runtime [4, 7, 10, 11, 39, 68]. Specifically, PyRTFuzz designs a framework for Python runtime testing, which can generate applications covering runtime APIs for fuzzing. WASMaker [7] uses real-world binaries to generate semantic-rich test inputs. WADIFF [68] integrates symbolic execution techniques for the generation of test inputs. OPDIFFER is fundamentally inspired by existing differential testing methodologies and employs targeted testing approaches specifically designed for the unique context of the Ethereum Virtual Machine.

8 Conclusion

In this paper, we present OPDIFFER, a differential testing framework for Ethereum Virtual Machine (EVM) based on opcode-level test input generation. By leveraging LLM’s reasoning and comprehension capabilities and static analysis methods, OPDIFFER can generate semantically valid and diverse test inputs according to the specifications. To assist following debugging processes, we also propose an automated bug identification and root cause localization algorithm. Compared with state-of-the-art baselines, our evaluation illustrates that OPDIFFER can achieve up to 410.8% code coverage improvement. Among nine EVM implementations spanning multiple scenarios, we have uncovered 26 previously unknown real-world bugs, 22 of which have been confirmed or fixed and three of them have been assigned CNVD IDs. Additionally, we investigated that 7.21% real-world smart contracts can trigger these bugs under certain environmental settings.

Data Availability

The dataset, artifact and the main experimental results of OPDIFFER are available at [42].

References

- [1] Blue Alloy. 2025. *Github revm repository*. Retrieved 2025-04-15 from <https://github.com/bluealloy/revm>
- [2] Ether Alpha. 2025. *Ethereum Client Diversity*. Retrieved 2025-04-15 from <https://clientdiversity.org>
- [3] Alex Beregszaszi, Paweł Bylica, Andrei Maiboroda, and Matt Garnett. 2021. *EIP-3540: EOF - EVM Object Format v1*. Retrieved 2025-04-15 from <https://eips.ethereum.org/EIPS/eip-3540>
- [4] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (Los Angeles, CA, USA) (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 351–364. doi:10.1145/3548606.3560624
- [5] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *2014 IEEE Symposium on Security and Privacy (SP) (Berkeley, CA, USA)*. IEEE, 114–129. doi:10.1109/SP.2014.15
- [6] Vitalik Buterin et al. 2013. *Ethereum white paper*. *GitHub repository 1* (2013), 22–23. Retrieved 2025-04-15 from <https://ethereum.org/en/whitepaper>
- [7] Shangdong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. 2024. WASMaker: Differential Testing of WebAssembly Runtimes via Semantic-Aware Binary Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1262–1273. doi:10.1145/3650212.3680358

- [8] Chu Chen, Pinghong Ren, Zhenhua Duan, Cong Tian, Xu Lu, and Bin Yu. 2023. SBDT: Search-Based Differential Testing of Certificate Parsers in SSL/TLS Implementations. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 967–979. doi:10.1145/3597926.3598110
- [9] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jiaguang Sun. 2023. Tyr: Finding Consensus Failure Bugs in Blockchain System with Behaviour Divergent Model. In *2023 IEEE Symposium on Security and Privacy (SP)* (San Francisco, CA, USA). IEEE, 2517–2532. doi:10.1109/SP46215.2023.10179386
- [10] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE '2019)*. 1257–1268. doi:10.1109/ICSE.2019.00127
- [11] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) (PLDI '16). Association for Computing Machinery, New York, NY, USA, 85–99. doi:10.1145/2908080.2908095
- [12] CoinMarketCap. 2025. *Ethereum price today*. Retrieved 2025-04-15 from <https://coinmarketcap.com/currencies/ethereum>
- [13] Ethereum community. 2025. *Ethereum Improvement Proposals*. Retrieved 2025-04-15 from <https://ethereum.org/en/eips>
- [14] Ethereum Javascript Community. 2025. *Github ethereumjs repository*. Retrieved 2025-04-15 from <https://github.com/ethereumjs/ethereumjs-monorepo>
- [15] Dan, Mario Vega, Mukul Kolpe, Spencer Taylor-Brown, and omahs. 2025. *State Transition Tests, Ethereum Execution Spec Tests*. Retrieved 2025-04-15 from https://ethereum.github.io/execution-spec-tests/main/tutorials/state_transition
- [16] DappRadar. 2025. *Top Ethereum Games*. Retrieved 2025-04-15 from <https://dappradar.com/rankings/protocol/ethereum/category/games>
- [17] National Vulnerability Database. 2021. *CVE-2021-39137 Detail*. Retrieved 2025-04-15 from <https://nvd.nist.gov/vuln/detail/CVE-2021-39137>
- [18] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Rongxiang Weng, Jingang Wang, Xunliang Cai, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. arXiv:2407.06153 [cs.SE] <https://arxiv.org/abs/2407.06153>
- [19] Ethereum. 2025. *Github evmone repository*. Retrieved 2025-04-15 from <https://github.com/ethereum/evmone>
- [20] Ethereum. 2025. *Github execution-specs repository*. Retrieved 2025-04-15 from <https://github.com/ethereum/execution-specs>
- [21] Ethereum. 2025. *Github Go Ethereum repository*. Retrieved 2025-04-15 from <https://github.com/ethereum/go-ethereum>
- [22] Ethereum. 2025. *Github Py-EVM repository*. Retrieved 2025-04-15 from <https://github.com/ethereum/py-vm>
- [23] Ethereum.org. 2025. *Decentralized finance (DeFi)*. Retrieved 2025-04-15 from <https://ethereum.org/en/defi>
- [24] Ethereum.org. 2025. *Ethereum Virtual Machine (EVM) implementations*. Retrieved 2025-04-15 from <https://ethereum.org/en/developers/docs/evm>
- [25] Ethereum.org. 2025. *The history of Ethereum*. Retrieved 2025-04-15 from <https://ethereum.org/en/history>
- [26] Ethereum.org. 2025. *Non-fungible tokens (NFT)*. Retrieved 2025-04-15 from <https://ethereum.org/en/nft>
- [27] Etherscan. 2025. *The Ethereum Blockchain Explorer*. Retrieved 2025-04-15 from <https://etherscan.io>
- [28] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. 2019. EVMFuzzer: detect EVM vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) (ESEC/FSE 2019). Association for Computing Machinery, New York, NY, USA, 1110–1114. doi:10.1145/3338906.3341175
- [29] Google. 2025. *Coverage profiling support for integration tests*. Retrieved 2025-04-15 from <https://go.dev/doc/build-cover>
- [30] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering*. 1176–1186. doi:10.1109/ICSE.2019.00120
- [31] Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco CA USA, 2023-11-30) (ESEC/FSE 2023). Association for Computing Machinery, 2201–2203. doi:10.1145/3611643.3617850
- [32] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. EOSAFE: Security Analysis of EOSIO Smart Contracts. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1271–1288. <https://www.usenix.org/conference/usenixsecurity21/presentation/he-ningyu>
- [33] Hyperledger. 2025. *Github Besu Ethereum Client repository*. Retrieved 2025-04-15 from <https://github.com/hyperledger/besu/>

- [34] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (Montpellier, France) (ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. doi:10.1145/3238147.3238177
- [35] Shinhae Kim and Sungjae Hwang. 2023. EtherDiffer: Differential Testing on RPC Services of Ethereum Nodes. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA) (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1333–1344. doi:10.1145/3611643.3616251
- [36] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Richard Tang, XiaoFeng Wang, and Xiapu Luo. 2021. As Strong As Its Weakest Link: How to Break Blockchain DApps at RPC Service. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. <https://www.ndss-symposium.org/ndss-paper/as-strong-as-its-weakest-link-how-to-break-blockchain-dapps-at-rpc-service/>
- [37] Li Li, Jiawei Wang, and Haowei Quan. 2022. Scalpel: The Python Static Analysis Framework. arXiv:2202.11840 [cs.SE] <https://arxiv.org/abs/2202.11840>
- [38] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*, 14–26. doi:10.1109/ASE56229.2023.00089
- [39] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (Copenhagen, Denmark) (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1645–1659. doi:10.1145/3576915.3623166
- [40] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 100, 13 pages. doi:10.1145/3597503.3639180
- [41] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jianguang Sun. 2023. LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. <https://www.ndss-symposium.org/ndss-paper/loki-state-aware-fuzzing-framework-for-the-implementation-of-blockchain-consensus-protocols/>
- [42] Jie Ma. 2025. *OpDiffer: LLM-Assisted Opcode-Level Differential Testing of Ethereum Virtual Machine*. doi:10.5281/zenodo.15195943
- [43] Pengxiang Ma, Ningyu He, Yuhua Huang, Haoyu Wang, and Xiapu Luo. 2024. Abusing the Ethereum Smart Contract Verification Services for Fun and Profit. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society. <https://www.ndss-symposium.org/ndss-paper/abusing-the-ethereum-smart-contract-verification-services-for-fun-and-profit/>
- [44] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. SELF-REFINE: iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (New Orleans, LA, USA) (NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2019, 61 pages.
- [45] Dominik Maier, Fabian Fäßler, and Jean-Pierre Seifert. 2022. Uncovering Smart Contract VM Bugs Via Differential Fuzzing. In *Reversing and Offensive-Oriented Trends Symposium (Vienna, Austria) (ROOTS'21)*. Association for Computing Machinery, New York, NY, USA, 11–22. doi:10.1145/3503921.3503923
- [46] Marius van der Wijden Martin Holst Swende. 2020. *EIP-3155: EVM trace specification [DRAFT]*. Retrieved 2025-04-15 from <https://eips.ethereum.org/EIPS/eip-3155>
- [47] NethermindEth. 2025. *Github Nethermind Ethereum client repository*. Retrieved 2025-04-15 from <https://github.com/NethermindEth/nethermind>
- [48] Beijing Academy of Blockchain and Edge Computing. 2025. *chainmaker document*. Retrieved 2025-04-15 from <https://docs.chainmaker.org.cn>
- [49] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. NeZha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP) (Berkeley, CA, USA)*. IEEE, 615–632. doi:10.1109/SP.2017.27
- [50] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP) (San Francisco, CA, USA)*. IEEE, 1974–1993. doi:10.1109/SP54263.2024.00137
- [51] SealSC. 2025. *Github SealEVM repository*. Retrieved 2025-04-15 from <https://github.com/SealSC/SealEVM>

- [52] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (Seattle, WA, USA) (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 322–333. doi:10.1145/3597926.3598059
- [53] smlXL Inc. 2025. *An Ethereum Virtual Machine Opcodes Interactive Reference*. Retrieved 2025-04-15 from <https://www.evm.codes/?fork=cancun>
- [54] Tianle Sun, Ningyu He, Jiang Xiao, Yinliang Yue, Xiapu Luo, and Haoyu Wang. 2024. All Your Tokens are Belong to Us: Demystifying Address Verification Vulnerabilities in Solidity Smart Contracts. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3567–3584. <https://www.usenix.org/conference/usenixsecurity24/presentation/sun-tianle>
- [55] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 166, 13 pages. doi:10.1145/3597503.3639117
- [56] Martin Holst Swende. 2025. *Github Go evmlab repository*. Retrieved 2025-04-15 from <https://github.com/holiman/goevmlab>
- [57] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)* (Vienna, Austria). IEEE, 103–119. doi:10.1109/EuroSP51992.2021.00018
- [58] Petar Tsankov, Andrei Dan, Dana Drachler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. doi:10.1145/3243734.3243780
- [59] Marius van der Wijden. 2025. *Github FuzzyVM repository*. Retrieved 2025-04-15 from <https://github.com/MariusVanDerWijden/FuzzyVM>
- [60] Sam Wilson. 2023. *Ethereum Execution Layer Specification*. Retrieved 2025-04-15 from <https://blog.ethereum.org/2023/08/29/eel-spec>
- [61] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. 2023. Randomized Testing of Byzantine Fault Tolerant Algorithms. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 101 (April 2023), 32 pages. doi:10.1145/3586053
- [62] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
- [63] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. 2024. Are We There Yet? Unraveling the State-of-the-Art Smart Contract Fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 127, 13 pages. doi:10.1145/3597503.3639152
- [64] Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. 2024. LLM4Fin: Fully Automating LLM-Powered Test Case Generation for FinTech Software Acceptance Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1643–1655. doi:10.1145/3650212.3680388
- [65] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 349–365. <https://www.usenix.org/conference/osdi21/presentation/young>
- [66] Wuqi Zhang, Zhuo Zhang, Qingkai Shi, Lu Liu, Lili Wei, Yepang Liu, Xiangyu Zhang, and Shing-Chi Cheung. 2024. Nyx: Detecting Exploitable Front-Running Vulnerabilities in Smart Contracts. In *2024 IEEE Symposium on Security and Privacy (SP)* (San Francisco, CA, USA). IEEE, 2198–2216. doi:10.1109/SP54263.2024.00146
- [67] Zhijie Zhong, Zibin Zheng, Hong-Ning Dai, Qing Xue, Junjia Chen, and Yuhong Nan. 2024. PrettySmart: Detecting Permission Re-delegation Vulnerability for Token Behaviors in Smart Contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 168, 12 pages. doi:10.1145/3597503.3639140
- [68] Shiyao Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. 2024. WADIFF: A Differential Testing Framework for WebAssembly Runtimes. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (Echternach, Luxembourg) (ASE '23)*. IEEE, 939–950. doi:10.1109/ASE56229.2023.00188