EtherDiffer: Differential Testing on RPC Services of Ethereum Nodes

Shinhae Kim
The Affiliated Institute of ETRI
Daejeon, South Korea
shinhae1106@nsr.re.kr

Sungjae Hwang*
Sungkyunkwan University
Suwon, South Korea
sungjaeh@skku.edu

ABSTRACT

Blockchain is a distributed ledger that records transactions among users on top of a peer-to-peer network. Among all, Ethereum is the most popular general-purpose platform and its support of smart contracts led to a new form of applications called decentralized applications (DApps). A typical DApp has an off-chain frontend and on-chain backend architecture, and the frontend often needs interactions with the backend network, e.g., to acquire chain data or make transactions. Therefore, Ethereum nodes implement the official RPC specification and expose a uniform set of RPC methods to the frontend. However, the specification is not sufficient in two points: (1) lack of clarification for non-deterministic event handling, and (2) lack of specification for invalid arguments. To effectively disclose any deviations caused by the insufficiency, this paper introduces EtherDiffer that automatically performs differential testing on four major node implementations in terms of their RPC services. EtherDiffer first generates a non-deterministic chain by multi-concurrent transactions and propagation delay. Then, it applies our key techniques called property-based generation and type-preserving mutation to generate both semantically-valid and semantically-invalid-yet-executable test cases. EtherDiffer executes the test cases on target nodes and reports any deviations in error handling or return values. The evaluation showed the effectiveness of our test case generation techniques with the success ratios of 98.8% and 95.4%, respectively. Also, EtherDiffer detected 48 different classes of deviations including 11 implementation bugs such as crash and denial-of-service bugs. We reported 44 of the detected classes to the specification and node developers and received acknowledgements as well as bug patches. Lastly, it significantly outperformed the official node testing tool in every technical aspect. We believe that our research findings can contribute to more stable DApp ecosystem by reducing the inconsistencies among nodes.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → **Program analysis**; • **Security and privacy** → *Distributed systems security*.

*Sungjae Hwang is the corresponding author.

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

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616251

KEYWORDS

blockchain, ethereum nodes, rpc services, differential testing

ACM Reference Format:
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 (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616251

1 INTRODUCTION

A blockchain is a distributed ledger on top of a peer-to-peer network and records transactions among users. It is technically a growing list of data structures called blocks, and each peer shares the same copy of the list to prevent any arbitrary modifications. A block contains various transactional and network state data such as transactions, event logs, and account states. When a user submits a transaction, one of the peers forms a block with the transaction and broadcasts it to the network. Once received, each peer individually validates the incoming block and appends it at the end of its chain. Since introduced by Satoshi Nakamoto in 2009 [36], blockchain technology has made much progress and became a top future trend [33].

Out of all, Ethereum is the most popular general-purpose blockchain platform with 199 billion dollars of market capitalization [7]. It consists of Ethereum peers referred to as "nodes" and supports general-purpose smart contracts. The contracts are programs that are implemented in high-level programming languages like Solidity [44] and deployed on the network. Users can execute their functions by submitting transactions with contract addresses and arguments specified. As smart contracts enable the automation of service logic, there has been a popularity increase in decentralized applications (DApps). According to statistics [6], there exist 2,855 Ethereum DApps spanning over various services like payments and auctions, and 31,590 users made a total of 79,550 transactions in a single day. Also, the top 10 DApps have the transaction volume of approximately 6.88 billion dollars in total [8].

A decentralized application consists of an off-chain frontend and an on-chain backend. The frontend is what users interact with, and the backend is a number of smart contracts to handle core application logic. However, the off-chain component needs frequent interactions with its backend network for e.g., acquiring chain states or making new transactions. Therefore, Ethereum nodes offer remote procedure call (RPC) services [15] and expose a set of RPC methods that the frontend can rely on. Meanwhile, there are a number of different implementations of Ethereum nodes in different programming languages. According to the official Ethereum page [16]:

> "This makes the network stronger and more diverse. The ideal goal is to achieve diversity without any client dominating to reduce any single points of failure."

To enable uniform RPC services among nodes, the Ethereum Foundation defines an official specification [12] that their RPC method implementations should comply to. The specification has a brief description of each method and defines regular expressions for valid arguments and return values. However, *the specification is not sufficient* in two points. First, it has *no clarification* on expected behaviors in case of non-deterministic chain events. For example, while there was a question in Ethereum Magicians forum [38]:

> "What's the expected behavior of `eth_getLogs` if the blockHash does not correspond to any block? This is not just a theoretical consideration, since chain reorganizations might cause a blockHash to no longer be valid.",

the questioner confirmed that some implementations return -32000 error code with an "unknown block" message, while others simply return an empty array. Second, the specification does not state *anything* on return values in case of invalid arguments such as values out of the allowed ranges. While there was an attempt to standardize error codes among nodes [2], the proposal has been stagnant for more than three years. It can be problematic if nodes behave differently as at least 63% of DApps rely on third-party node providers [40] that maintain a pool of various types of nodes and seamlessly serve with a different one based on black-box load balancers [27]. For example, a bug report in Infura, the most dominant node provider, showcases that a user unconsciously experienced inconsistent results when they queried the latest block number [5].

To effectively investigate any deviations in terms of RPC services, this paper presents *the first syntax- and semantics-aware* differential testing approach. Also, we implemented EtherDiffer that applies our approach to four different node implementations that together take up approximately 99.7% of the main Ethereum network [19]. To facilitate better understanding, we define a few terms as follows:

Definition 1 (*Non-deterministic Chain*). If a generation mechanism produces a different chain every time, the state of which is unpredictable, the resulting chain is called non-deterministic.

Definition 2 (*Semantically-Validness*). A test case is semantically-valid if all method call arguments match the expected semantics.

Definition 3 (*Semantically-Invalidness*). A test case is semantically-invalid if any argument disconforms to the expected semantics.

Particularly, this research overcomes three technical challenges: 1) the generation of semantically-valid test cases, 2) the generation of semantically-invalid-yet-executable test cases, and 3) the enforcement of chain state consistency among nodes prior to test case executions. EtherDiffer first constructs a local network that consists of four nodes, each from a different implementation, as well as auxiliary nodes for chain evolution. Then, it generates a non-deterministic chain by making multiple-and-concurrent transactions that actively produce event logs and state changes. Also, we made minimum instrumentation on the nodes to mimic real-world propagation delay and trigger non-deterministic chain events.

On top of the generated chain, EtherDiffer overcomes the first two challenges by our key techniques: property-based generation and type-preserving mutation. We first defined a domain-specific language (DSL) that captures the type and semantic requirement, which we call "property," for each method argument. Then, we converted the specification of RPC-triggering methods into our DSL, which we denote as spec_{DSL}. Based on spec_{DSL}, EtherDiffer generates semantically-valid template code where all arguments satisfy their properties. The template code is yet another valid test case but not bound to a specific RPC-serving node. Also, EtherDiffer stochastically mutates one of the arguments to turn the template code semantically-invalid, while preserving the executability. Then, it produces a set of four test cases by binding the template code with each target node and cross-checks their execution return values.

The last challenge is to enforce the chain state consistency among nodes when executing test cases as its inconsistency can lead to false alarms. EtherDiffer implements a two-phase architecture: generation and testing phase. During the generation phase, the auxiliary nodes operate to evolve the chain. Then, once the chain reaches the configured height, their operations stop and EtherDiffer begins its testing phase after all target nodes are synced. Also, we implemented a save-and-restore strategy as a transaction-sending test case changes the original chain state. The evaluation showed that our techniques generated both semantically-valid and invalid-yet-executable test cases with the success ratios of 98.8% and 95.4%, respectively. Also, EtherDiffer detected 48 deviation classes including 11 implementation bugs such as crash and denial-of-service bugs. We reported 44 of the detected classes and received acknowledgements for our research findings as well as bug patches. Lastly, EtherDiffer significantly outperformed the official testing tool for Ethereum nodes in every technical aspect.

In short, the contributions of this paper are the followings:

- **We present an approach that generates semantically-valid test cases based on a specification. Also, we present a mutation strategy to produce invalid test cases while preserving the executability.** Our approaches can be applied in other domains as well if specifications exist.
- **We present EtherDiffer that automatically reports deviations among four Ethereum node implementations that take up 99.7% of the mainnet.** EtherDiffer found 48 classes of deviations, which include 11 implementation bugs. We believe it can contribute to the stability of DApp ecosystem and publicly opened the tool implementation.[1]

2 BACKGROUND

In this section, we explore the concepts highly related to our research. We first explain the two factors that make blockchain non-deterministic. Then, we present the architecture of typical DApps as well as the necessity of using JavaScript libraries in development.

2.1 Blockchain Non-determinism

Blockchain is technically a growing list of blocks, and specifically-configured nodes called miners create them. However, as blockchain operates on a large user base, multiple transactions concurrently reside on the network, and it is upon the miner's decision which and how many transactions to include in a block. Therefore, it is

[1] [Link for EtherDiffer] https://github.com/JosephK95/EtherDiffer-public

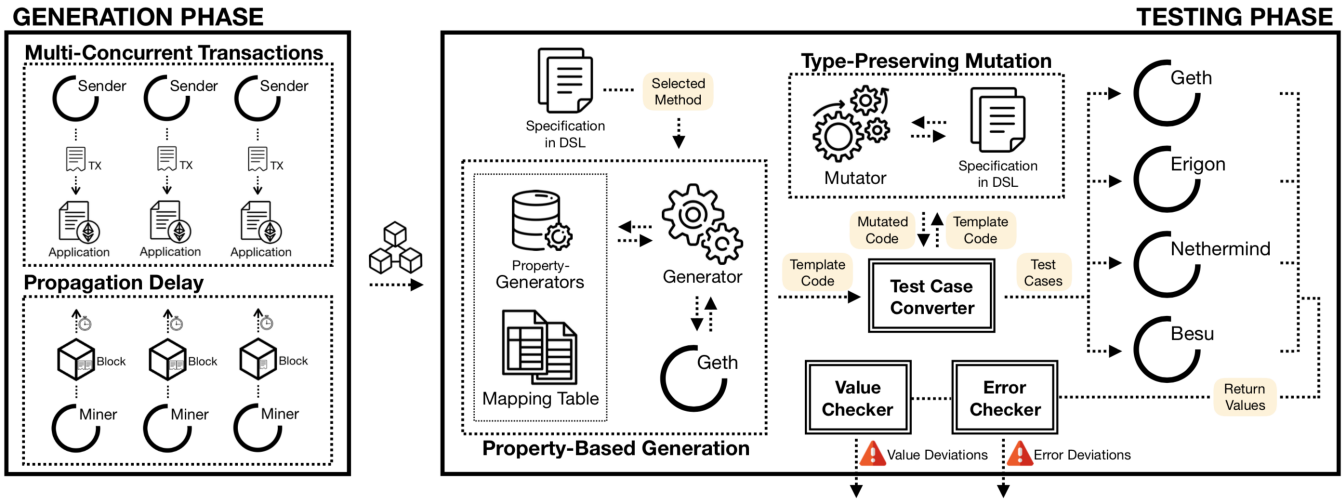

Figure 1: Overall Architecture of EtherDiffer

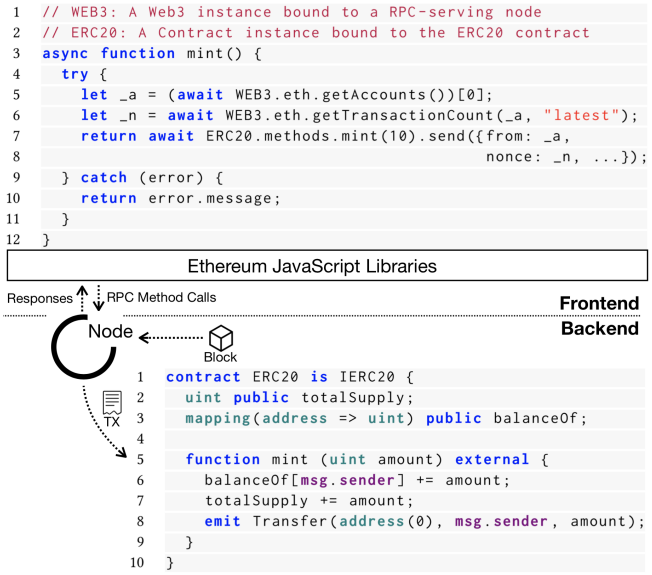

Figure 2: Architecture of ERC20 Decentralized Application

completely non-deterministic which transactions a new block will consist of. Also, the inclusion of different transactions leads to different elements in other block fields as well. For example, blocks contain event logs emitted from smart contract executions. Therefore, only the logs from chosen contract transactions become part of the blocks. Another key factor that makes blockchain non-deterministic is propagation delay. Ethereum has become a significantly large network with 5,649 nodes [19]. While each node propagates the new block to its peers once received, a time difference in block reception is inevitable. Therefore, the chain can have two blocks with the same parent when a different miner created another block before perceiving the existence of a new block. Such event is called "temporal fork," and the chain evolves with either one of the blocks, leading to "chain reorganization." The left-over blocks drop off the canonical chain and become orphaned. However, their headers can be still part of the canonical chain as "uncles" if subsequent miners include them in their blocks under creation.

The Merge. Since the initial launch in 2015, Ethereum has adopted proof-of-work (PoW) consensus mechanism [18] where miners formulate blocks by solving cryptographic puzzles. Recently, Ethereum switched to proof-of-stake (PoS) consensus mechanism [17] where block proposers and validators participate in block formulation by staking their Ether balances. However, the non-deterministic factors remain exactly the same except that uncles no longer exist.

2.2 Ethereum DApps and JavaScript Libraries

A decentralized application consists of an user-interfacing frontend and backend smart contracts. About 75% of DApps have web-based frontends implemented in JavaScript [40], and the frontends need frequent interactions with the backend network. However, as they are off-chain, Ethereum nodes serve as a bridge to the network by exposing a set of RPC methods. For example, the user interface of a crowdfunding DApp can display the pledge history by making an `eth_getLogs` method call. Also, the frontend can update the chain state by calling `eth_sendTransaction` method upon a new user pledge. However, interacting directly with nodes is not practical for two reasons. First, it is cumbersome to make syntactically-valid method calls as a lot of methods have error-prone parameters such as 20 bytes-encoding hexadecimal addresses or an array of 32 bytes-encoding hexadecimal strings. Also, more importantly, nodes have no support of event notifications. As blockchain has non-deterministic characteristics, it is unpredictable when a new transaction will be mined and confirmed. Users would have to make numerous repetitions of method calls to track their transactions. Therefore, DApps leverage Ethereum JavaScript libraries that are simple wrappers of RPC methods but support input formatters to filter out miss-typed arguments and callback-based event notifications. Figure 2 shows an example ERC20 decentralized application.

The frontend first acquires two types of information by `getAccounts` and `getTransactionCount` library methods at line 5 and 6: an address which it has ownership of and the number of transactions made by the address called nonce. Then, it sends a minting transaction to its backend ERC20 smart contract by `send` method, which resolves with the transaction receipt once mined successfully. The nonce should be incremented by one after each transaction, and invalid nonce values can lead to various unexpected results including transaction rejection or long-term pending [20]. However, as mentioned earlier, users can unconsciously experience RPC service from different nodes. Therefore, it can be crucial if any node returns a deviated nonce value for the `getTransactionCount` method call.

3 METHODOLOGY

To effectively investigate the deviations of RPC service, we implemented EtherDiffer that automatically performs differential testing on four Ethereum node implementations. In this section, we first explain the overall architecture of EtherDiffer and how it generates non-deterministic chains. Then, we present our domain-specific language as well as two key techniques to generate test cases based on the specification converted into our DSL.

3.1 Overview

EtherDiffer first constructs a local network that consists of four nodes, each from a different implementation, as well as mining nodes for chain evolution. The network is configured to activate all mainnet features except the only difference that it adopts PoW mechanism for block creation. However, the difference has no impact on outcomes as the testing proceeds only after a consensus is reached. Figure 1 shows the overall architecture upon the network construction. EtherDiffer begins the generation phase where the network generates a non-deterministic chain (sec. 3.2). Once the chain reaches the configured height, EtherDiffer switches over to the testing phase where it generates test cases and cross-checks their execution results. For the test case generation, EtherDiffer leverages web3.js, the official Ethereum JavaScript library [3]. Particularly, we defined a domain-specific language that captures the syntactic and semantic requirements for method arguments (sec. 3.3) and converted the library specification into our DSL, which we denote as spec_{DSL}. EtherDiffer first selects a method from spec_{DSL}, and Generator produces semantically-valid template code where all arguments satisfy their requirements (sec. 3.4). In addition, Mutator stochastically changes one of the arguments to a semantically-invalid value while preserving its type (sec. 3.5). Lastly, Test Case Converter instantiates a set of four test cases by binding the template code with each target node. Once the executions are completed, Error Checker reports if only a subset of nodes throw errors while the others return values, and Value Checker reports if their return values are not consistent to one another. The test cases are wrapped with `try-catch` statements to property identify errors while preserving the executions of EtherDiffer.

3.2 Non-deterministic Chain Generation

As explained in section 2.1, the two factors that make blockchain non-deterministic are multiple-and-concurrent transactions and propagation delay. Likewise, EtherDiffer applies the factors to its local network and generates a non-deterministic chain.

$$
\begin{array}{rcl}
s & := & \overline{\texttt{typedef}\ t}\ \ \overline{t <: t}\ \ \overline{\texttt{propdef}\ p}\ \ \overline{d} \\
t & := & t_p \mid \{\overline{[\texttt{opt}]^?\ k\colon t}\} \mid \texttt{t\_Array}(t) \mid t \cup t \\
p & := & p_s \mid p_d \mid \{\overline{[\texttt{opt}]^?\ k\colon p}\} \mid \texttt{p\_Array}(p) \mid p \cup p \\
d & := & t\ \ M(\overline{[\texttt{opt}]^?\ t \Vdash p\ [\vee\ t \Vdash p]^?}) \mid t\ \ c.M(t \Vdash p) \\
c & := & C.F_{<v_s, v_v>}(\overline{v_a})
\end{array}
$$

Figure 3: Syntax of our Domain-Specific Language

Multiple-and-Concurrent Transactions. EtherDiffer deploys applications, each of which consists of one or two smart contracts that together provide a service. We leveraged the applications from Solidity by Example [10], a well-known database that covers a wide range of real-world applications. We filtered those out that only contain simple logic such as proxies and leveraged the remaining 12 applications. Upon their deployments, EtherDiffer operates auxiliary nodes that repeatedly and concurrently send transactions to the contracts. Particularly, each node is dedicated to a single application to maximize the number of available transactions at a time. Also, we defined transaction sequences for each application that guarantee the execution completion while actively triggering log emissions and state changes. In total, EtherDiffer generates a chain based on 33 sequences consisted of 109 valid transactions.

Propagation Delay. To further induce non-determinism in our chain, e.g., presence of temporal forks, there should be time delay in block propagations among nodes. In other words, mining nodes should create another block before perceiving the existence of a new block. To trigger such occasions, we instrumented the mining nodes to delay their block propagations within the period of a block creation time multiplied by six. This reflects the policy that mining nodes can include uncle headers that are up to six-generations apart from their blocks under creation.

3.3 Domain-Specific Language

For test case generation, EtherDiffer leverages the official Ethereum JavaScript library, web3.js. To automate the processing of the library specification, we defined a domain-specific language that captures the types and semantic requirements for method arguments. Figure 3 shows the syntax of our DSL. The specification s consists of type definitions, subtype relations, property definitions, and method declarations. A type t can be one of the primitive types t_p, an object type $\{\overline{[\texttt{opt}]^?\ k\colon t}\}$, an array type, or a union of two types. The primitive types not only include basic JavaScript types like `t_Number` and `t_String`, but also more constrained types like `t_Address` and `t_Hex32`. An object type $\{\overline{[\texttt{opt}]^?\ k\colon t}\}$ is a collection of key-type pairs, some of which can be optional. Also, the DSL captures subtype relations between types ($\overline{t <: t}$). A property p is a semantic requirement that an argument should satisfy. A primitive property is either a "static" property p_s or a "dynamic" property p_d, depending on whether its value generation relies on other method calls. For example, the properties `p_Gas` and `p_Value` are static as

```
t_TxObj  getTransactionFromBlock(((t_Number ∪ t_String) ≀ p_BlockNumber) ∨ (t_Hex32 ≀ p_BlockHash),
                                 t_Number ≀ p_TxIdx@(p_BlockNumber ∪ p_BlockHash))
t_FeeObj  getFeeHistory(t_Number ≀ p_Range(1,1024),
                        (t_Number ∪ t_String) ≀ p_BlockNumber,
                        t_Array(t_Number) ≀ p_Array(p_Range(0,100))
```

(a) Specification in DSL

(c) Mapping Table (p_TxIdx··)

Method Name: getBlock

Arguments: [@_, false]
Postfixes: [".transactions.index( )"]

(b) Semantically-Valid Template Code

```
let aa0 = "latest";
let aa1 = 1; // getBlock(aa0, false).transactions.index()
return getTransactionFromBlock(aa0, aa1);
```

(d) Semantically-Invalid Mutated Code

```
let aa0 = "latest";
let aa2 = 757; // mutation code for an invalid index
return getTransactionFromBlock(aa0, aa2);
```

Figure 4: Example Code Generation Based on Specification in DSL

users can designate values as they want when sending transactions. On the other hand, a dynamic property `p_Nonce@(p_EOA)` needs the return value of `getTransactionCount` method call with its associated address as first argument. An object property $\{\overline{[\texttt{opt}]^?\ k\colon\ p}\}$ defines the requirement for an object type argument, where each key k should satisfy the property p if exists. To systematically derive property definitions, we first extracted all existing parameters and clustered those that accept the same type of arguments based on their descriptions. Also, in case of any ambiguity, we referred to the official Ethereum page for further clarification. A method declaration d either declares a standard method $(t\ \ M(\overline{[\texttt{opt}]^?\ t \wr p\ [\vee\ t \wr p]^?}))$ or a contract-interaction method $(t\ \ c.M(t \wr p))$. A standard method is a simple wrapper of a node-exposing RPC method with one-to-one correspondence. A contract-interaction method is yet another wrapper but facilitates RPC method calls to interact with smart contracts. A declaration specifies the type and property $(t \wr p)$ of each parameter as well as its return type. Particularly, standard methods have zero or more parameters, each of which can be optional, while contract-interaction methods have a single parameter. Also, the parameters in standard methods can be disjunctive to allow different types of arguments. Figure 4-(a) shows the declarations of two methods, which we will refer to throughout this section. The upper method has two parameters of a block and a transaction index and returns the corresponding transaction in the block. The lower method retrieves a history of transaction fees in the maximum 1024 blocks backwards from the second parameter block. For every block, it sorts the fee values in ascending order and returns the percentiles specified by the third parameter. A contract transaction c is a prefix for contract-interaction methods and corresponds to calling the member function F in contract C with arguments $\overline{v_a}$. The subscripted values v_s and v_v represent transaction-level arguments, `msg.sender` and `msg.value`, respectively. In total, we defined 32 types, 6 subtype relations, 31 properties, and 31 method declarations. Also, as the testing proceeds in the same local network, we leveraged the 109 transactions from the generation phase as well.

3.4 Property-Based Template Code Generation

EtherDiffer selects a target method from spec_{DSL}, and Generator generates semantically-valid template code based on the GenTC function in Algorithm 1. The template code is valid JavaScript code but not bound to any RPC-serving node yet. For each property of the selected method (Line 3), Generator first "instantiates" the property if necessary (Line 4). Then, it generates a valid argument by applying different approaches depending on property types (Line 5). Lastly, Generator produces a target method call with the generated arguments (Line 7).

Algorithm 1: Template Code Generation

```
Input  : Selected method in DSL (d)
Output : Generated template code (stmts)
1 function GenTC(d)
2     stmts ← [], args ← []
3     for p in d.props() do
4         p ← InstProp(p)
5         stmts, arg ← GenArg(stmts, p)
6         args.push(arg)
7     stmts ← GenCall(stmts, d, args)
8     return stmts
```

Algorithm 2: Type-Preserving Mutation

```
Input  : Template code (stmts)
Output : Mutated template code (stmts')
1 function MutateTC(stmts)
2     call ← stmts.pop()
3     arg ← SelectArg(call)
4     p ← PropLookUp(arg)
5     stmts, arg' ← GenArg(stmts, p)
6     stmts' ← AppendNewCall(stmts, call, arg, arg')
7     return stmts'
```

Property Instantiation. There exist several properties that should be associated with specific accounts and others. For example, when sending transactions, nonce parameter should receive the transaction count of the sender account. Therefore, once a sender address is generated, Generator instantiates the nonce property with the address. Another example is the transaction index parameter of `getTransactionFromBlock` in Figure 4-(a). As it is associated with the first parameter block, Generator instantiates the index property upon the generation of a block argument.

Static Property. A primitive property, which captures the semantics of a primitive type parameter, is "static" if its value generation is possible without method calls. Static properties in most

Table 1: Success Ratios of Valid and Invalid Test Case Executions

	Test Cases	Generated Properties					Total
		Static	Dynamic	Object	Array	Union	
✔	2,965 (98.8%)	1,431 (98.8%)	1,275 (98.8%)	965 (97.6%)	279 (99.6%)	150 (99.3%)	4,110 (98.6%)
✘	35 (1.2%)	17 (1.2%)	15 (1.2%)	24 (2.4%)	1 (0.4%)	1 (0.7%)	58 (1.4%)
	3,000	1,448	1,290	989	280	151	4,168

(a) Valid Test Cases (Error ? No: ✔, Yes: ✘)

	Test Cases	Mutated Types				Total
		Primitive	Object	Array	Union	
✔	2,863 (95.4%)	1,435 (99.4%)	819 (86.8%)	- (0%)	609 (99.3%)	2,863 (95.4%)
✘	137 (4.6%)	8 (0.6%)	125 (13.2%)	- (0%)	4 (0.7%)	137 (4.6%)
	3,000	1,443	944	-	613	3,000

(b) Invalid Test Cases (Lib. Error ? No: ✔, Yes: ✘)

cases capture the user-assigning transaction parameters but also other parameters whose values can be statically generated based on their descriptions, e.g., a random integer in the specified range for `p_Range(1,1024)`. We defined a Property-Generator for each static property, which Generator leverages in case of a static property parameter. For non-trivial cases like user-assigning values, we designed the Property-Generator to return one of the values used in the specification code snippets as arbitrary value generation can lead to execution failures like out-of-gas errors. Figure 4-(b) shows an example of semantically-valid template code in a simplified format. Generator produced "`latest`" for the first parameter as Property-Generator of `p_BlockNumber` can return one of the pre-defined strings.

Dynamic Property. A primitive property is "dynamic" if its value generation relies on method calls. Dynamic properties capture the parameters that receive chain-stored values such as transaction and block hashes. Therefore, it is necessary to make method calls at runtime to acquire a valid property value. Also, it is often that methods should be called with certain arguments and/or postprocess their return values. Therefore, we made a Mapping Table for each dynamic property where each entry contains a method name, fixed arguments if necessary, and postfixes that return a valid property value when appended to the return value. Generator selects one of the entries and chains with the method call result to generate a dynamic property argument. In Figure 4-(b), the index property was first instantiated to `p_TxIdx@"latest"` upon the generation of `aa0`. Then, Generator looked up the Mapping Table in Figure 4-(c) and made a `getBlock` method call with the associated "`latest`" value (denoted as `@_`) and `false` to acquire the block without transaction details. Lastly, it accessed the transaction attribute and returned a valid index by applying the helper `index` method.

Compound Property. Aside from primitive properties, there exist a number of object properties that capture object type parameters such as transactions and log filters. Generator recursively generates a value for each object field following the above approaches and merges into an object. An array property captures an array type parameter, and Generator recursively generates values based on its base property and forms them into an array. For a union property, it randomly selects one of the properties and generates its value.

3.5 Type-Preserving Mutation

To investigate any deviations against semantically-invalid arguments as well, Mutator stochastically turns the generated template code into semantically-invalid code. While EtherDiffer enables users to configure the mutation probability, it is set to 50% by default. The mutation is based on the MutateTC function in Algorithm 2 and satisfies two characteristics: single-argument mutation and type preservation. To effectively examine one property at a time, Mutator randomly selects a single argument in the target method call (Line 3). In case of compound type arguments, it selects one of the object fields or array elements. Also, it is essential to preserve the argument type as the library filters out miss-typed arguments and does not make RPC method calls in such cases. Therefore, Mutator looks up a property from spec_{DSL} that has the same type with but not equivalent to the original argument property (Line 4). It also takes subtype relations into account during property selection. Then, Mutator generates a valid argument for the selected property (Line 5) and appends the new method call, argument of which is replaced with the generated value (Line 6). This approach ensures both semantically-invalidness and executability as it makes use of an unintended property with a compatible type. Figure 4-(d) presents the mutated code from the template code. Mutator selected `p_Range(1,1024)` and replaced the original index with a semantically-invalid integer value while preserving `t_Number`. Finally, Test Case Converter instantiates a set of four test cases from possibly-mutated template code, and Error Checker and Value Checker report in case of respective deviations.

4 EVALUATION

EtherDiffer examines the deviations among four major Ethereum nodes that take up 99.7% of the main network. Particularly, we chose their most recent stable versions by the time of evaluation as our targets: Geth v1.10.21-stable, Erigon 2022.07.04-alpha, Nethermind v1.13.6, and Hyperledger Besu v22.4.4. To evaluate the effectiveness of our techniques and disclose any deviations, we set up the following research questions. Also, we compare EtherDiffer with the official node testing tool maintained by the Ethereum Foundation.

- **RQ1 (Effectiveness of Test Case Generation):** How much do the semantically-valid test cases complete their executions without errors? Also, how much do semantically-invalid test cases preserve their executability without library errors?
- **RQ2 (Deviation and Bug Detection Capability):** How many deviations and bugs has EtherDiffer detected?
- **RQ3 (Comparison with the Official Tool):** How effective is EtherDiffer compared to the official testing tool for node implementations?

Table 2: Overview of All Deviation Classes Found by EtherDiffer

Category	Deviation Class	Report	RC
Invalid Argument Handling (sec. 4.2.1)	**(IAH-1)** Number of Transactions in Invalid Block	✓S*	‡
	(IAH-2) Past Event Logs in Invalid Block	✓S*	‡
	(IAH-3) Transaction at Certain Index in Invalid Block	✓S*	‡
	(IAH-4) Number of Uncles in Invalid Block	✓S*	‡
	(IAH-5) Uncle at Certain Index in Invalid Block	✓S*	†‡
	(IAH-6) Code of Specific Contract in Invalid Block	✓S*	‡
	(IAH-7) Balance of Specific Address in Invalid Block	✓S*	‡
	(IAH-8) Nonce of Specific Account in Invalid Block	✓S*	‡
	(IAH-9) Storage of Specific Contract in Invalid Block	✓S*	‡
	(IAH-10) Account State Structure of Unknown Account	✓S*	‡
	(IAH-11) Past Event Logs in Wrong Block Range	✓S*	‡
	(IAH-12) Transaction Fee History in Wrong Block Range	✓S*	‡
	(IAH-13) Transaction in Block at Invalid Index	✓S*	‡
	(IAH-14) Uncle in Block at Invalid Index	✓S*	‡
Gas Estimation and Local Execution (sec. 4.2.2)	**(GL-1)** Improper Max Fee Parameter for Gas Estimation	✓S*	‡
	(GL-2) Low Gas Limit Parameter for Gas Estimation	✓S	‡
	(GL-3) High Gas Limit Parameter for Gas Estimation	✓S*	‡
	(GL-4) Invalid Transaction Type for Gas Estimation	✓S*	‡
	(GL-5) Invalid Account Nonce for Gas Estimation	✓S*	‡
	(GL-6) Insufficient Funds in Account for Gas Estimation	✓S*	‡
	(GL-7) Improper Max Fee Parameter for Local Execution	✓S*	‡
	(GL-8) Low Gas Limit Parameter for Local Execution	✓S*	‡
	(GL-9) Invalid Transaction Type for Local Execution	✓S*	‡
	(GL-10) Invalid Account Nonce for Local Execution	✓S*	‡
Uncle Access (sec. 4.2.3)	**(UA-1)** Block Structure that Matches Uncle Hash	✓S*	†‡
	(UA-2) Uncle at Certain Index in Uncle Block	✓S*	†‡
	(UA-3) Transaction at Certain Index in Uncle Block	✓S*	†‡
	(UA-4) Number of Uncles in Uncle Block	✓S*	†‡
	(UA-5) Number of Transactions in Uncle Block	✓S*	†‡
Method Support (sec. 4.2.4)	**(MS-1)** No Method for Pending Transaction List	✓[E*,B]	‡
	(MS-2) No Method for Account State Structures	✓E*	
	(MS-3) No Method for List of Account Addresses	✓E	‡
	(MS-4) No Method for Current Protocol Version	✓G	‡
Fields and Formats (sec. 4.2.5)	**(FF-1)** Field Difference in Uncle Block Structures	✓[G,E*]	†‡
	(FF-2) Field Difference in Event Log Structures	(AD)	†‡
	(FF-3) Field Difference in Canonical Block Structures	(AD)	‡
	(FF-4) Field Difference in Receipt Structures	(AD)	
	(FF-5) Field Difference in Transaction Structures	✓B*	‡
	(FF-6) Format Inconsistency in Contract Storage Keys	✓B*	‡
	(FF-7) Format Inconsistency in Storage Slot Values	(AD)	‡
Implementation Bugs (sec. 4.2.6)	**(IB-1)** Invalid Total Difficulty in Uncle Headers	✓[G,N*,B]	†‡
	(IB-2) Wrong Values in Account State Structures	✓N*	‡
	(IB-3) Wrong Log Index in Event Log Structures	✓B*	†‡
	(IB-4) Wrong Log Identifier in Event Log Structures	✓B*	†‡
	(IB-5) Wrong Transaction Fee History in Block Range	✓B*	‡
	(IB-6) Wrong Gas Estimation for Contract Executions	✓[N*,B*]	
	(IB-7) Crash Bug while Retrieving Past Event Logs	✓E*	†‡
	(IB-8) DoS while Retrieving Past Event Logs	✓B*	†‡

✓S: Specification ✓G: Geth ✓E: Erigon ✓N: Nethermind ✓B: Besu
✓[..]: Multiple Nodes **(AD)**: Acceptable Deviation *: Confirmed or Patched
†: Detection Failure of RPC-COMPAT Due to Chain Generation (sec. 4.3.1)
‡: Detection Failure of RPC-COMPAT Due to Test Case Generation (sec. 4.3.2)

Table 3: Overview of Error Handling and Uncle Access Deviations of Node Implementations

	IAH-1	IAH-2	IAH-3	IAH-4	IAH-5	IAH-6	IAH-7	IAH-8	IAH-9	IAH-10	IAH-11	IAH-12	IAH-13	IAH-14
• ***Geth***	`null`	`[]`	`null`	`null`	`null`	**Error**	**Error**	**Error**	**Error**	$\textbf{Val}_{\text{acc}}$	`[]`	$\textbf{Val}_{\text{fee}}$	`null`	`null`
• ***Erig.***	`null`	`[]`	`null`	`null`	`null`	$\textbf{Val}_{\text{code}}$*	**Error**	**Error**	**Error**	-**	**Error**	$\textbf{Val}_{\text{fee}}$	`null`	`null`
• ***Neth.***	**Error**	**Error**	**Error**	**Error**	**Error**	**Error**	**Error**	**Error**	**Error**	$\textbf{Val}_{\text{acc}}$	**Error**	**Error**	**Error**	**Error**
• ***Besu***	`null`	`[]`	`null`	`null`	`null`	**Error***	`null`	0	`null`	**Error**	`[]`	**Error**	`null`	`null`

GL-1	GL-2	GL-3	GL-4	GL-5	GL-6	GL-7	GL-8	GL-9	GL-10	UA-1	UA-2	UA-3	UA-4	UA-5
Error	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	**Error**	**Error**	$\textbf{Val}_{\text{call}}$	$\textbf{Val}_{\text{call}}$	$\textbf{Val}_{\text{block}}$	$\textbf{Val}_{\text{block}}$	`null`	$\textbf{Val}_{\text{num}}$	$\textbf{Val}_{\text{num}}$
Error	$\textbf{Val}_{\text{gas}}$	**Error**	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	**Error**	**Error**	$\textbf{Val}_{\text{call}}$	$\textbf{Val}_{\text{call}}$	$\textbf{Val}_{\text{call}}$	**Error**	`null`	**Error**	`null`	`null`
$\textbf{Val}_{\text{gas}}$	**Error**	$\textbf{Val}_{\text{gas}}$	**Error**	**Error**	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{call}}$	$\textbf{Val}_{\text{call}}$	**Error**	**Error**	`null`	**Error**	**Error**	**Error**	**Error**
$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	$\textbf{Val}_{\text{gas}}$	**Error**	**Error**	$\textbf{Val}_{\text{call}}$	$\textbf{Val}_{\text{call}}$	$\textbf{Val}_{\text{block}}$	$\textbf{Val}_{\text{block}}$	$\textbf{Val}_{\text{tx}}$	$\textbf{Val}_{\text{num}}$	$\textbf{Val}_{\text{num}}$

*: Erigon and Besu can return **Error** and `null` respectively, depending on the argument. **: Erigon does not support the method. (see MS-2)

4.1 Effectiveness of Test Case Generation

4.1.1 Semantically-Valid Test Cases. EtherDiffer leverages our property-based technique to generate semantically-valid test cases. To evaluate its effectiveness, we generated 3,000 valid test cases and checked whether they successfully completed executions without any errors. Table 1-(a) shows the result as well as the numbers of generated properties in corresponding test cases. The result confirms the effectiveness of our technique as 2,965 valid test cases completed their executions and 4,110 out of the total 4,168 properties belonged to the execution-completed ones. This ensures that our technique generates a semantically-valid argument with 98.6% confidence or even higher as some of the arguments in the execution-failed test cases can be still valid. We manually investigated the 35 failed test cases and confirmed that each argument was yet semantically-valid but resulted in errors when combined in most cases. For example, `getPastLogs` method receives two block number arguments that together specify a block range to retrieve event logs from. While EtherDiffer succeeded at generating a valid argument for each block number, there were occasions when the starting number was greater than the ending number, producing errors. We believe that our future work to add semantic requirements in argument combinations can eliminate the errors.

4.1.2 Semantically-Invalid Test Cases. EtherDiffer applies our type-preserving mutation strategy to convert semantically-valid test cases into invalid ones. To evaluate its effectiveness, we performed mutations on 3,000 valid test cases and checked whether the invalid test cases did not trigger library errors and returned

results including node-produced errors. Table 1-(b) shows the result as well as the mutated types in corresponding test cases. The result confirms that our mutation technique effectively preserves the executability with 95.4% confidence as 2,863 invalid test cases produced no library errors. One thing to note is that there were no mutations for arrays as each array type consists of a single property and thus has no mutation candidate. We manually investigated the 137 failed test cases and confirmed that while each mutation successfully preserved the type, they mostly failed as the library performs additional checks on transaction object arguments to reduce transaction failures. For example, the majority of cases occurred as the mutated gas value was smaller than the minimum required gas. We believe that a more sophisticated mutation approach based on intersecting with the original property can further improve the effectiveness.

4.2 Deviation and Bug Detection Capability

To detect any deviations among Ethereum node implementations, we conducted 10 iterations of EtherDiffer. Particularly, we configured each iteration to generate a 300-height non-deterministic chain and 600 test case sets with the default mutation probability. In total, our differential testing was based on 6,000 test case sets consisting of 24,000 test cases, and EtherDiffer detected error deviations and value deviations in 1,536 and 1,693 test case sets, respectively. We randomly selected 300 error-deviating and 300 value-deviating sets for manual investigation and confirmed that only 11 deviating sets were false positives, accounting for 1.8% of the total. The false positives were derived from the deviations as well, as the node implementations returned different values during the process of dynamic property generation. For the remaining true positives, we clustered them into six major categories and 48 different classes as shown in Table 2. We analyzed each class further and reported 44 deviation classes to either RPC specification developers or node developers based on the following criteria:

- (Report to Specification: 29 Deviation Classes) The deviation class was caused by the lack of clear specification such as error handling and uncle access.
- (Report to Node Developers: 15 Deviation Classes) The deviation class was caused by node implementations that either disconform to the specification or have bugs.

We did not make reports for the remaining four classes as they correspond to simply defining additional fields in data structures. For the rest of this section, we briefly explain each of the deviation categories found by EtherDiffer.

4.2.1 Invalid Argument Handling. EtherDiffer detected 14 deviation classes in terms of handling invalid arguments of RPC method calls. The classes can be grouped into four sub-categories based on the argument types: (1) invalid blocks, (2) unknown accounts, (3) wrong block ranges, and (4) invalid indices. While there exist many RPC methods with a block parameter, EtherDiffer found that node implementations handle nine of the methods in different ways when a given block argument is invalid. Table 3 shows the return values from each implementation. As the columns from IAH-1 to IAH-9 show, the return values vary in the range of the following: `null`, `[]`, an error, or even a normal value. Another finding is that even a single implementation handles the methods inconsistently. For example, while Geth returns `null` or `[]` for five of the methods, it just throws an error for the rest. Additionally, node implementations have divergent handling logic on five methods that receive an account, a block range, or an index as an argument. When an invalid argument is given, the range of possible return values is the same as that of the invalid block case. Particularly, some implementations just return a normal account or transaction fee history even if they do not actually exist. Although all the fields are set with zero, this can still lead to misconception of DApp developers.

Finding 1: *Ethereum node implementations have divergent handling logic on invalid arguments.*

Finding 2: *Even a single node implementation handles each method in an inconsistent way.*

4.2.2 Gas Estimation and Local Execution. The core component of a decentralized application is its backend smart contracts which implement the service logic. And as actual interactions with smart contracts require transaction fees, the RPC interface offers two ways of interactions without fees and enables users to validate their transactions in advance: gas estimation and local contract execution. However, while the interaction requires an object argument with various fields designated, EtherDiffer found that each implementation performs validation on a very different set of fields. In other words, while some implementations return an error in case of an invalid field, the others just return a normal result: an estimated gas value or a contract return value. The columns from GL-1 to GL-10 in Table 3 summarize the return values when invalid field values are given. As the results show, none of the implementations share the same set of fields to validate. For example, while Geth throws an error against improper fee in gas estimation, Nethermind throws an error against low gas limit and invalid transaction type and nonce. Also, as in invalid argument handling, even a single implementation does not validate fields consistently. For example, while Geth has no validation on gas limit in gas estimation, it throws an error against low gas limit in local execution. These deviations can be problematic as DApp developers usually rely on the results and make actual transactions with the same field values.

Finding 3: *Ethereum node implementations validate different parameters in gas estimation and local execution.*

4.2.3 Uncle Access. As explained earlier, the RPC specification has no clarification with regard to non-deterministic event handling. While uncle blocks are indicators of such occurrences, EtherDiffer found five deviation classes in terms of handling them. Particularly, node implementations returned deviated results when an uncle hash is passed to five methods that expect a block hash argument. The argument is still semantically-valid as an uncle hash is yet another block hash. However, as a canonical chain only includes the headers of once-valid uncle blocks, the return values from accessing their body elements are up to the node implementations. The columns from UA-1 to UA-5 in Table 3 show the return values for each method, and the results show that the implementations have different levels of support for original uncle block data: partial, full, or none. While Geth supports most of the methods, it does not provide access to the transactions existed in uncle blocks. Besu fully supports uncle block access, and the other two nodes provide no support by returning either an error or `null`.

```
async function erigon() {
  try {
    let aa0 = "41";
    ...
    let aa3 = "0x7f7B7c0992cCC777626EF18Cc3578D0d3b56a376";
    let aa4 = [null];
    let aa5 = "0x4a817c800"; // mutation code
    return await web3_erigon.eth.getPastLogs({
      fromBlock: aa0, toBlock: aa5, address: aa3, topics: aa4
    });
  } catch (error) {
    return "[ERROR] " + error.message;
  }
}
```

```
async function besu() {
  try {
    let aa0 = "genesis";
    ...
    let aa2 = "0x31aB061154876beb39912216E96F76756Bb3EFe1";
    let aa3 = [null];
    let aa4 = "1000000000000000000"; // mutation code
    return await web3_besu.eth.getPastLogs({
      fromBlock: aa0, toBlock: aa4, address: aa2, topics: aa3
    });
  } catch (error) {
    return "[ERROR] " + error.message;
  }
}
```

Figure 5: Test Cases for Crash and Denial-of-Service Bugs

Finding 4: *Ethereum node implementations provide different levels of support for original uncle block data.*

4.2.4 Method Support. The RPC specification defines a uniform set of methods that Ethereum nodes should implement. However, EtherDiffer found that the nodes have not implemented some of the methods. Particularly, Geth and Besu have no support for a single method, and Erigon lacks implementations for three methods that return a pending transaction list or account data. We confirmed that the nodes have not clearly noted the non-implementations in their documentations except for one case. Even more, we found a case where the documentation specified all the required information to make a valid method call including example code snippets. This can give false intuitions to DApp developers that their application code is operating in the expected manner.

Finding 5: *Ethereum node implementations do not provide consistent sets of RPC methods to users.*

4.2.5 Fields and Formats. EtherDiffer found two kinds of syntactic deviations: structure fields and return value formats. While the implementations define the core blockchain data structures in their own programming languages, the structures consist of slightly different sets of fields. Particularly, we found deviations in five structure definitions: event logs, uncle blocks, canonical blocks, transactions, and receipts. For example, while Geth and Erigon do not define transactions field for uncle blocks, Nethermind and Besu have the field with an empty array value. Also, only Besu does not include type field for transactions. We analyzed the results and reported those that disconform to the specification. In addition, while the RPC interface offers two methods to access contract storage, node implementations return its keys and values in different formats such as the length of hexadecimal storage values.

Finding 6: *The definitions of core data structures are slightly different in node implementations.*

Finding 7: *Ethereum node implementations return storage keys and values in inconsistent formats.*

4.2.6 Implementation Bugs. The last category consists of eight classes of implementation bugs. The total difficulty field in block headers stores the value of accumulated mining work until the block. However, we found a bug in each of three node implementations that they either produced `undefined` or 0 in case of an uncle block header. Also, there was a bug in Nethermind that returned zero for non-zero storage values and proofs. Besu had three implementation bugs with regard to the handling of event logs and transaction fee history. Particularly, Besu returned wrong values for log indices and identifiers at all times. Also, it returned arbitrary fee values additionally when the number of requested blocks exceeded that of available blocks. There existed two more bugs in Nethermind and Besu that they produced wrongly estimated gas values for contract executions. Lastly, we found two critical bugs in Erigon and Besu that either crashed the node or blocked it from serving users. When a significantly large value was used as the ending block number for event log retrievals, a crash occurred in the handler method of Erigon, and Besu did not serve any subsequent RPC requests until resolved with a timeout. Figure 5 shows two example test cases to trigger the bugs. The `erigon` function tries to acquire all event logs originated from `aa3`. However, the block range is wrong as the `aa5` variable holds an invalid block value, and Erigon crashed with a "method handler crashed" message when executed. Erigon incorrectly assumed that `toBlock` is always less than the maximum 32-bit unsigned integer. After we reported this bug, it took less than a day until the patch, which demonstrates the high severity of the bug. Similarly, the `besu` function tries to retrieve all event logs originated from `aa2`. However, the `aa4` variable holds an invalid block value, and Besu falls into denial-of-service for a specific time period with a "Thread blocked" message.

Finding 8: *Ethereum nodes have various implementation bugs, which include crash and DoS bugs.*

4.2.7 Discussion: Real-World Impacts. After we reported 29 deviation classes, our findings received an acknowledgement from the specification developers. Moreover, the developers even approved to contribute our test cases to their official repository [26]:

> "thank you for these reports, this is very helpful",
> "would be very happy to have you contribute your tests"

Similarly, we reported 15 deviation classes to node developers and received an acknowledgement of our implementation [25]:

> "super interested in this project ··· like to read more"

The developers have either patched or confirmed 13 of the reported classes, while they had their own rationale for the rest. One thing to note is that some uncle-related deviation classes may lose their value in the main network due to the merge. However, they still have practical impacts as there exist Ethereum-based PoW networks with 3.63 billion dollars of market capitalizations [34]. Also, while EtherDiffer can fail at detecting PoS-specific deviations such as handling of block finality, we leave it as future work.

Table 4: Comparison of RPC-COMPAT and ETHERDIFFER

(a) Chain Generation			(b) Test Case Generation		
299	Transactions	4,762	71.4%	Coverage	100%
0	Event Logs	2,737	5.5%	Unique TCs	81.8%
0	Chain Events	1,001	0%	Error TCs	28.9%

RPC-COMPAT (left), ETHERDIFFER (right)

4.3 Comparison with the Official Tool

As Ethereum nodes are the entry points for end users and thus significant, the Ethereum Foundation has developed its own testing platform called Hive [13] and actively validates the node implementations with various tools on top of it. Particularly, it offers a testing tool called RPC-COMPAT [14], which generates a chain and validates the consistency of the implementations with a set of test cases. As RPC-COMPAT is designed for conformance tests, we closely compare the tool with ETHERDIFFER in terms of two technical aspects in this section: chain generation and test case generation. For fairness, we configured RPC-COMPAT to generate a 300-height chain and 600 test cases and compared with a single iteration of ETHERDIFFER.

4.3.1 Chain Generation. We first compared the non-determinism in their generated chains as it is the key characteristic of blockchain and significant for deviation detection. Particularly, the previous section 2.1 explained that blockchain is non-deterministic in terms of block elements and chain events. Therefore, we extracted and analyzed the transactions and event logs in each of their chains as they are the main block elements. Also, we counted the number of chain event occurrences based on the existence of uncle headers. As Table 4-(a) shows, ETHERDIFFER significantly outperformed RPC-COMPAT in all three aspects of chain generation. More surprisingly, our investigation confirmed that RPC-COMPAT generates a chain in a completely deterministic manner that every block except the pre-defined genesis contains a single, exactly-equivalent transaction with no event log. Also, it does not produce non-deterministic chain events such as temporal forks at all. This leads to a significant degradation in its effectiveness and failures of detecting *at least 13 deviation classes* shown in Table 2. On the other hand, ETHERDIFFER applies multi-concurrent transactions and generates a non-deterministic chain where each block contains unpredictable numbers of transactions and event logs. Also, the block elements are diverse as they are based on 109 different transactions. Lastly, the propagation delay in our network successfully triggered a substantial number of non-deterministic chain events.

4.3.2 Test Case Generation. We then compared the test case generation of each tool. While both tools successfully generated 600 test cases, we evaluated them in the following three aspects: (1) the coverage of RPC methods in the specification, (2) the ratio of unique test cases, and (3) the ratio of test cases that triggered node-produced errors. The first aspect is to ensure that the test cases cover the complete list of RPC methods except the mining-related and Geth-specific ones, while the latter two are to evaluate the test case diversity for each method. Table 4-(b) shows the overall results, and the coverage confirms that RPC-COMPAT validates only a subset of RPC methods. This is due to its design shortcomings such as the lack of certain block elements and non-deterministic chain events. Furthermore, RPC-COMPAT has significantly low test case diversity as it generates only one or two "static" test cases for each method. Our evaluation confirmed that 94.5% of the generated test cases were duplicates with no support for error-triggering test cases. These partial support for RPC methods and low test case diversity significantly degrade its effectiveness and result in failures of detecting *at least 45 deviation classes* shown in Table 2. On the other hand, ETHERDIFFER validates the complete list of RPC methods. Also, it dynamically generates test cases, and our evaluation confirmed that 81.8% were unique. Lastly, 28.9% of the unique test cases triggered node-produced errors, which demonstrates that ETHERDIFFER sufficiently examines error handlings as well.

4.4 Threats to Validity

The evaluation results showed that ETHERDIFFER effectively generated test cases and detected a variety of deviation classes. Also, it significantly outperformed the official node testing tool in terms of non-deterministic chain and test case generation. However, while our test case generation relies on the manually-converted specification, there could be human errors during the process of conversion. Also, as ETHERDIFFER does not cover all possible arguments for RPC method calls, some other deviations can possibly remain undiscovered. We believe that our conversion was correct as the specification is well-structured with specific patterns and the descriptions provide sufficient details to extract and cluster properties. Also, the success ratios of test case generation confirmed the validity of our conversion. Besides, we believe that our findings of deviation classes have already produced high impacts to the community as acknowledged by both specification and node developers.

5 RELATED WORK

Analysis of Smart Contracts and DApps. As on-chain smart contracts play a vital role in the core logic of DApps, ensuring their correctness and security is crucial for the development of reliable DApps. Prior research efforts have disclosed various vulnerabilities in smart contracts [23, 32, 37] and have developed automated tools for their detection [1, 4, 22, 32, 39, 42, 45]. Additionally, several measurement studies have been conducted to assess the strengths and weaknesses of these tools [30, 35, 46]. Moreover, with the increasing popularity of DApps, researchers have extended their investigation beyond smart contracts to encompass the entire DApp ecosystem. For example, Zhang et al. [47] identified synchronization bugs arising from inconsistencies between on-chain and off-chain states, and Li et al. [27] demonstrated a denial-of-service attack on Ethereum RPC services. Also, Su et al. [31] explored past real-world attacks on Ethereum DApps. However, despite these valuable contributions to the correctness and security of smart contracts and DApps, none of them has investigated the deviations among different node implementations in terms of their RPC services. Also, the previous work had impacts on specific DApps only, whereas ETHERDIFFER contributes to the ecosystem in general as all DApps depend on the RPC services of Ethereum nodes.

Differential Testing. Differential testing has proven to be an effective method for identifying deviations in software and has been successfully applied in diverse domains, such as JVMs [43, 50, 51], deep learning systems [24], and x86 disassemblers [41]. Recently, it

has also been adopted in the blockchain domain. Fu et al. [48] introduced EVMFuzzer, a tool that mutates real-world smart contracts using eight predefined mutators. By executing the mutated contracts, EVMFuzzer identified inconsistencies in opcode sequence execution and gas usage across four EVM implementations. Another framework, NeoDiff [9], focuses on the virtual machine of Neo smart contracts and directly generates test bytecodes using bytecode-level mutators. This approach produces valid opcode sequences not attainable by high-level programming language compilers. Similarly, EVMLab [11] deploys random test bytecodes and invokes them within a single transaction to evaluate different Ethereum clients. Yang et al. [49] developed Fluffy, a tool designed to detect consensus bugs in Ethereum that can lead to errorneous states within the blockchain. Notably, Fluffy applies multiple transactions and enables the detection of bugs that cannot be detected with a single transaction. Ma et al. [21] introduced LOKI, which also detects consensus bugs but uniquely employs a dynamic state model generated from real-time consensus information. The dynamic information enables LOKI to detect complex consensus bugs in four different blockchain platforms including Ethereum that would be bypassed with fixed types of inputs. Although the frameworks have effectively detected deviations among nodes, none of them is designed to investigate the RPC services. The closest tool to EtherDiffer is rpc-compat maintained by the Ethereum Foundation. However, our evaluation showed that rpc-compat has significantly low effectiveness in both technical aspects.

6 CONCLUSION

In this research, we aimed at finding any deviations among four major node implementations in terms of their RPC services. The deviations can exist as the RPC specification does not clarify the expected behaviors in case of non-deterministic chain events and invalid arguments. To effectively detect such deviations, we proposed two test case generation techniques and implemented EtherDiffer, an automatic differential testing tool. EtherDiffer first generates a non-deterministic chain by multiple-and-concurrent transaction injection and propagation delay. Then, it applies our techniques to generate both semantically-valid and invalid-yet-executable test cases. EtherDiffer executes them on each target node and reports if any error deviations or value deviations are detected. Our evaluation showed that EtherDiffer generated both valid and invalid test cases with the success ratios of 98.8% and 95.4%, respectively. Also, it detected 48 deviation classes, which include 11 implementation bugs such as crash and denial-of-service bugs. We reported 44 of the detected classes to the specification and node developers and received acknowledgements as well as bug patches. Lastly, EtherDiffer significantly outperformed the official node testing tool in terms of both chain and test case generation. We believe that our research findings can stabilize the ecosystem of decentralized applications by eliminating the inconsistencies among nodes.

7 DATA AVAILABILITY

To contribute to openness in science, we disclose the replication package [28] and evaluation datasets [29] to the public. As they are archived on a preserved digital repository, anyone can download the package and datasets at any time from the links.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00688,AI Platform to Fully Adapt and Reflect Privacy-Policy Changes), (No. 2022-0-01199; Graduate School of Convergence Security, Sungkyunkwan University), and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2022R1F1A1074495)

REFERENCES

[1] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor and Prateek Saxena. 2019. Exploiting the Laws of Order in Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 363–373. https://doi.org/10.1145/3293882.3330560

[2] Paul Bouchon. 2022. *Remote Procedure Call Specification.* Retrieved January 4, 2023 from https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1474.md#error-codes

[3] ChainSafe. 2016. *Web3.js - Ethereum JavaScript API.* Retrieved January 4, 2023 from https://github.com/ChainSafe/web3.js/tree/v1.7.4

[4] Chenguang Zhu, Ye Liu, Xiuheng Wu, and Yi Li. 2022. Identifying Solidity Smart Contract API Documentation Errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–13. https://doi.org/doi.org/10.1145/3551349.3556963

[5] Infura Community. 2019. *How Does the Load Balancer Work?* Retrieved January 4, 2023 from https://community.infura.io/t/how-does-the-load-balancer-work/1090

[6] ConsenSys. 2020. *Ethereum by the Numbers - May 2020.* Retrieved January 23, 2023 from https://consensys.net/blog/news/ethereum-by-the-numbers-may-2020/

[7] Cryptoslate. 2023. *Coin Rankings.* Retrieved January 23, 2023 from https://cryptoslate.com/coins/

[8] DappRadar. 2023. *Top Ethereum Dapps.* Retrieved January 23, 2023 from https://dappradar.com/rankings/protocol/ethereum

[9] Dominik Maier, Fabian Fäßler and Jean-Pierre Seifert. 2021. Uncovering Smart Contract VM Bugs via Differential Fuzzing. In *Proceedings of the 5th Reversing and Offensive-oriented Trends Symposium (ROOT)*. 11–22. https://doi.org/10.1145/3503921.3503923

[10] Smart Contract Engineer. 2022. *Solidity by Example.* Retrieved January 4, 2023 from https://solidity-by-example.org/

[11] Ethereum. 2019. *EVM Lab Utilities.* Retrieved July 24, 2023 from https://github.com/ethereum/evmlab

[12] Ethereum. 2022. *Ethereum JSON-RPC Specification.* Retrieved January 4, 2023 from https://ethereum.org/en/developers/docs/nodes-and-clients/

[13] Ethereum. 2023. *Hive - Ethereum End-to-End Test Harness.* Retrieved January 4, 2023 from https://github.com/ethereum/hive

[14] Ethereum. 2023. *Rpc-Compat Simulator.* Retrieved January 10, 2023 from https://github.com/ethereum/hive/tree/master/simulators/ethereum/rpc-compat

[15] Ethereum.org. 2022. *JSON-RPC API.* Retrieved January 4, 2023 from https://ethereum.org/en/developers/docs/apis/json-rpc/

[16] Ethereum.org. 2022. *Nodes and Clients.* Retrieved January 4, 2023 from https://ethereum.org/en/developers/docs/nodes-and-clients/

[17] Ethereum.org. 2022. *Proof-of-Stake.* Retrieved January 11, 2023 from https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/

[18] Ethereum.org. 2022. *Proof-of-Work.* Retrieved January 11, 2023 from https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/

[19] Ethernodes. 2023. *Ethereum Mainnet Statistics.* Retrieved January 23, 2023 from https://ethernodes.org

[20] Ethereum Stack Exchange. 2016. *What Happens When a Transaction Nonce is Too High?* Retrieved January 4, 2023 from https://ethereum.stackexchange.com/questions/2808/what-happens-when-a-transaction-nonce-is-too-high

[21] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li and Jiaguang Sun. 2023. LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols. In *Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2023.24078

[22] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. VULTRON: Catching Vulnerable Smart Contracts Once and for All. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 1–4. https://doi.org/doi.org/10.1109/ICSE-NIER.2019.00009

[23] Sungjae Hwang and Sukyoung Ryu. 2020. Gap between Theory and Practice: An Empirical Study of Security Patches in Solidity. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 542–553. https://doi.org/10.1145/3377811.3380424

[24] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. Dlfuzz: Differential Fuzzing Testing of Deep Learning Systems. In *Proceedings of the 26th*

ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). 739–743. https://doi.org/10.1145/3236024.3264835

[25] JosephK95. 2022. *Erigon RPC Service Consistency against Other Client Implementations #1*. Retrieved July 24, 2023 from https://github.com/ledgerwatch/erigon/issues/4962

[26] JosephK95. 2022. *Proposal: Error Handling Specification #1*. Retrieved July 24, 2023 from https://github.com/ethereum/execution-apis/issues/286

[27] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Tang, XiaoFeng Wang, and Xiapu Luo. 2021. As Strong As Its Weakest Link: How to Break Blockchain DApps at RPC Service. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*. 616–633. https://doi.org/10.14722/ndss.2021.23108

[28] Shinhae Kim and Sungjae Hwang. 2023. *EtherDiffer: Differential Testing on RPC Services of Ethereum Nodes (Code)*. Retrieved August 10, 2023 from https://doi.org/10.6084/m9.figshare.23913096.v1

[29] Shinhae Kim and Sungjae Hwang. 2023. *EtherDiffer: Differential Testing on RPC Services of Ethereum Nodes (Data)*. Retrieved August 9, 2023 from https://doi.org/10.6084/m9.figshare.21936555.v1

[30] Shinhae Kim and Sukyoung Ryu. 2020. Analysis of Blockchain Smart Contracts: Techniques and Insights. In *Proceedings of the 5th IEEE Secure Development Conference (SecDev)*. 65–73. https://doi.org/10.1109/SecDev45635.2020.00026

[31] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing and Baoxu Liu. 2021. Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. 1307–1324.

[32] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 254–269. https://doi.org/10.1145/2976749.2978309

[33] Bernard Marr. 2022. *The Top 10 Tech Trends In 2023 Everyone Must Be Ready For*. Retrieved January 11, 2023 from https://www.forbes.com/sites/bernardmarr/2022/11/21/the-top-10-tech-trends-in-2023-everyone-must-be-ready-for

[34] MiningPoolStats. 2023. *Mining Pool Stats*. Retrieved January 23, 2023 from https://miningpoolstats.stream/

[35] Monika Angelo and Gernot Salzer. 2019. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *Proceedings of the 1st IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. 69–78. https://doi.org/10.1109/DAPPCON.2019.00018

[36] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. *Decentralized Business Review* (2008).

[37] Nicola Atzei, Massimo Bartoletti and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts. In *Proceedings of the 6th International Conference on Principles of Security and Trust (POST)*. 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

[38] Fellowship of Ethereum Magicians. 2018. *Remote Procedure Call Specification*. Retrieved January 4, 2023 from https://ethereum-magicians.org/t/remote-procedure-call-specification/1537

[39] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 67–82. https://doi.org/10.1145/3243734.3243780

[40] Evgeny Ponomarev. 2019. *DApp Survey Results 2019*. Retrieved January 4, 2023 from https://medium.com/fluence-network/dapp-survey-results-2019-a04373db6452

[41] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. N-version Disassembly: Differential Testing of x86 Disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*. 265–274. https://doi.org/10.1145/1831708.1831741

[42] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 1–29. https://doi.org/10.1145/3360615

[43] Sungjae Hwang, Sungho Lee, Jihoon Kim and Sukyoung Ryu. 2021. JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. 1708–1718. https://doi.org/10.1109/ICSE-Companion52605.2021.00073

[44] Solidity Team. 2022. *Solidity*. Retrieved January 4, 2023 from https://soliditylang.org/

[45] Teng Zhou, Kui Liu, Li Li, Zhe Liu, Jacques Klein and Tegawendé F Bissyandé. 2021. SmartGift: Learning to Generate Practical Inputs for Testing Smart Contracts. In *Proceedings of the 37th IEEE International Conference on Software Maintenance (ICSM)*. 23–34. https://doi.org/10.1109/ICSME52107.2021.00009

[46] Thomas Durieux, João F. Ferreira, Rui Abreu and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 530–541. https://doi.org/10.1145/3377811.3380364

[47] Wuqi Zhang, Lili Wei, Shuqing Li, Yepang Liu, and Shing-Chi Cheung. 2021. ÐArcher: Detecting On-Chain-Off-Chain Synchronization Bugs in Decentralized Applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 553–565. https://doi.org/10.1145/3468264.3468546

[48] 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 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1110–1114. https://doi.org/10.1145/3338906.3341175

[49] Youngseok Yang, Taesoo Kim and Byung-Gon Chun. 2021. Finding Consensus Bugs in Ethereum via Multi-Transaction Differential Fuzzing. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 349–365.

[50] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep Differential Testing of JVM Implementations. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 1257–1268. https://doi.org/10.1109/ICSE.2019.00127

[51] 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 (PLDI)*. 85–99. https://doi.org/10.1145/2908080.2908095

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