Fork State-Aware Differential Fuzzing for Blockchain Consensus Implementations

Wonhoi Kim[1*], Hocheol Nam[1*], Muoi Tran[2], Amin Jalilov[1], Zhenkai Liang[3], Sang Kil Cha[1], Min Suk Kang[1†]
[1]*KAIST*, {wh.kim, hcnam, amin, sangkilc, minsukk}@kaist.ac.kr
[2]*ETH Zürich*, dutran@ethz.ch
[3]*National University of Singapore*, liangzk@comp.nus.edu.sg

***Abstract*—Blockchain networks allow multiple client implementations of the same consensus algorithm by different developers to coexist in the same system. Ensuring correct implementations among these heterogeneous clients is crucial, as even slight semantic discrepancies in their implementations can lead to safety failures. While existing fuzzing frameworks have discovered implementation flaws in blockchain, they suffer from several challenges in testing them with sequences of conflicting blocks, called forks. Existing tools fail to adequately assess the fork-handling processes in blockchain implementations when relying on traditional code coverage feedback, which lacks the granularity needed to navigate the diverse and complex fork-handling scenarios. This paper introduces FORKY, a fork state-aware differential fuzzing framework designed to detect implementation discrepancies within the critical fork-handling process with its novel fork-aware mutation and fork-diversifying feedback mechanisms. We test FORKY on the two most influential blockchain projects: Bitcoin and Ethereum, which are the representatives of the two major blockchain consensus algorithm families, Proof-of-Work (PoW) and Proof-of-Stake (PoS) consensus algorithms.**

***Index Terms*—Blockchain, consensus, differential fuzzing**

I. INTRODUCTION

The open nature of blockchain projects allows heterogeneous implementations of clients with the same consensus algorithm to coexist in the same network. Major blockchain projects, such as Bitcoin and Ethereum, have maintained a strong community-driven, open client development culture, where multiple independent teams of developers implement their consensus clients in different languages and maintain them independently. As a result, blockchains are often operated by multiple families of clients, each of which has multiple versions coexisting at the same time. For example, as of March 2024, only 39.2% of Bitcoin clients run the most up-to-date Bitcoin Core version 26.0.0, whereas 60% run older versions of Bitcoin Core and 1% run several other client families (e.g., btcd, bcoin) [58], [19]. Ethereum's four major client families (i.e., Prysm, Lighthouse, Teku, Nimbus) are used by 37%, 33%, 19%, 10%, respectively [18], [49].

Having some implementation diversity in distributed networks is desirable in general as the risk of single point of failures can be reduced; yet, special care must be taken to ensure that all these heterogeneous client implementations correctly agree on the same blockchain states. In blockchain consensus algorithms, even a slight semantic difference between two implementations of the same algorithm can lead to a critical safety violation.

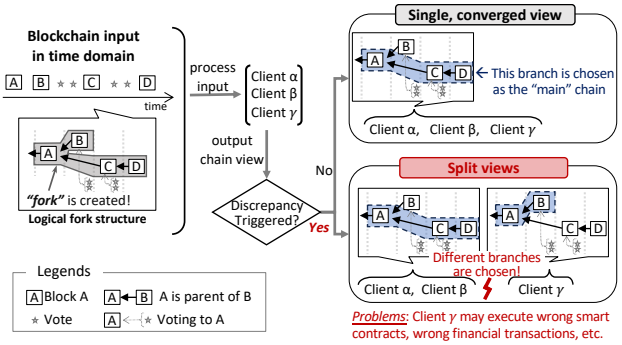


Fig. 1. An example of how discrepancies in blockchain clients' fork-handling logic can lead to split views among clients.

A simple example in Figure 1 illustrates how discrepancies in implementation can lead to safety violations in blockchain networks. Three clients (or nodes) α, β, and γ receive block messages (□) and votes (★) from the blockchain network in a particular sequence. These clients encounter a *fork*, a condition in blockchain inputs where two blocks (B and C) extend from the same parent block (A), creating two competing branches.[1] Since competing branches, such as B and C, may contain conflicting transactions, only one branch should eventually become the main (or canonical) branch. Ideally, if the blockchain's consensus algorithm operates correctly (without implementation bugs), all client nodes in the network should eventually converge to a single view, agreeing on the same branch. However, discrepancies in the consensus implementations across multiple clients may lead to divergent views (e.g., α and β may select a different branch than γ). These divergent views, whether permanent or transient, can cause serious issues, as a client operating on an incorrect view might mishandle transactions or smart contracts based on their divergent blockchain state.

This observation suggests a need for a practical testing tool capable of evaluating multiple blockchain clients with inputs exhibiting forks, particularly, to find discrepancies in the fork-handling logic. While recent studies [17], [37], [57] have made

* Both authors contributed equally to this work.
† Corresponding author.

[1]In this paper, the terms "forks" and "branches" specifically refer to those within the blockchain data structure, *not* to be confused with software code forks and branches.

strides in fuzzing blockchain consensus implementations, current blockchain fuzzers fall short in directly testing fork-handling logics and finding discrepancies in implementation.

A key technical challenge for existing fuzzers lies in their inability to grasp the complex state transitions found in blockchain consensus, particularly concerning fork-handling mechanisms. Blockchain consensus algorithms are distinct from traditional software in that they are highly stateful and have heavy state transitions. Particularly, the arrival of each new block and other auxiliary messages (e.g., votes) can significantly alter the blockchain state; e.g., one new block can tip the balance in favor of one branch over another as seen in Figure 1. The primary challenge in testing consensus implementations is to find an input sequence of blocks potent enough to significantly influence the states crucial to fork-handling logic. Unfortunately, conventional code coverage feedback methods may not be effective enough for identifying such important blockchain inputs, as two different blockchain inputs could yield the same branch coverage, yet only one might alter the blockchain state in a meaningful way.

Current fuzzers, including Fluffy [57], Tyr [17], and LOKI [37], can *partially* handle this problem by randomly varying blockchain transactions and block orders. However, relying only on code coverage feedback [57] or random block generation based on a coarse-grained state model [17], [37], these tools lack the ability to precisely represent intricate fork structure layout of blockchain inputs, limiting their ability to test the fork-handling logic in blockchain consensus algorithms reliably and effectively.

In this paper, we tackle this challenge by addressing the intricacies of analyzing blockchain states, especially fork structures, within fuzzing processes. Our approach is the first to consider the complex blockchain fork states, which we coin as *fork state-aware* fuzzing, testing a wide range of blockchain states to detect critical discrepancies in fork-handling logic found in consensus algorithms. We employ two main strategies: fork-aware mutation strategies and a fork-diversifying feedback mechanism. The former leverages a deep understanding of fork structures to test diverse blockchain states, thus enhancing the detection of fork-handling logic discrepancies beyond traditional mutation methods. The latter introduces a novel feedback mechanism aimed at uncovering previously unseen fork-handling events, overcoming the limitations of conventional code coverage feedback. By selecting fuzz inputs that present new and distinct fork events, our approach ensures a wider range of blockchain states are tested, leading to more effective and comprehensive testing outcomes.

We design and implement FORKY, an open-sourced differential fuzzer based on our proposed fork state-aware fuzzing methodology. FORKY effectively test fork-handling logic in Proof-of-Work (PoW) and Proof-of-Stake (PoS) blockchain consensus algorithms. Our evaluation of FORKY across 34 Bitcoin clients and 4 Ethereum clients demonstrates its effectiveness in finding several discrepancies in them.

In summary, our contributions are as follows:

- We propose a new fork state-aware fuzzing methodology aimed at testing fork-handling logic within blockchain consensus algorithms, which is complementary to existing blockchain fuzzers.
- We present fork-aware mutation strategies that generate inputs exhibiting many fork conditions, and a new fork-diversifying feedback mechanism that can reliably explore unseen fork-handling cases.
- We present FORKY, a differential fuzzer that implements our approach and evaluates it on Bitcoin's PoW and Ethereum's PoS consensus algorithms, finding several discrepancies in their fork-handling implementations.
- We open-source FORKY for its wider use in other PoW and PoS blockchains [3].

II. BACKGROUND

In this section, we first introduce several basic blockchain terminologies to understand the rest of the paper.

Blockchains. A blockchain (e.g., Bitcoin [38], Ethereum [13]) is a distributed ledger storing transactions in a chain of blocks that are cryptographically linked together. The blockchain is commonly maintained by a peer-to-peer (P2P) network of nodes or clients, which store a replica of the blockchain locally. Blockchain client nodes collectively follow a consensus algorithm to agree on the validity of individual blocks, aiming to have exactly the same view (i.e., identical copies of blocks). Typically, each block consists of the hash of its previous block in the blockchain (called the parent block) and transactions.

Users generate transactions that are periodically grouped into a new block that extends the head (i.e., the latest block of the *winning* branch) by special clients (e.g., miners in PoW consensus or validators in PoS consensus). A new block is then circulated in the P2P network and independently validated by clients.

Proof-of-Work algorithm. Many early blockchains, such as Bitcoin, use the Proof-of-Work (PoW) algorithm to achieve consensus among untrusted client nodes. The PoW algorithm operates through specialized client nodes, known as miners, tasked with creating blocks. To create a block, miners compete in finding a specific mathematical solution known as a nonce, that, when combined with block data, produces a hash with a required number of leading zeros. A chain of blocks with greater cumulative computational effort is regarded as the *winning* branch if multiple branches exist in the network. In general, the longest branch is regarded as the winning branch as the series of blocks that required the most substantial computational effort to construct necessitate smaller hash values.

Proof-of-Stake algorithm. More recent blockchains have adopted the Proof-of-Stake (PoS) algorithm, which selects validators to create blocks based on their stake (i.e., the amount of cryptocurrency they hold). In 2022, Ethereum switched to PoS from PoW with the introduction of the Gasper protocol [14]. In Gasper, one validator is chosen at random as the proposer, who is responsible for creating a new block and proposing it to the network. Other validators must attest to the validity

of the proposed block by casting votes (or attestations) in a timely manner; otherwise, they are penalized for rule violation (i.e., lose their deposit). When two-thirds of validators vote correctly on the chain head for two epochs in a row, the block can not be reverted by other forks.

Fork and fork-choice rules. When a client receives two or more blocks extending the same parent block, it has multiple branches of blocks in its local blockchain copy, or a fork. Forks are resolved quickly following a pre-defined set of rules (e.g., favoring the longest branch in Bitcoin). The set of rules is called the *fork-choice rules* (because they determine which branch to choose when handling a fork) and may vary depending on the consensus algorithms in different blockchain projects [38], [14]. Note that it is natural for a PoW or PoS blockchain to have forks in the first place because miners or validators may create blocks while being unaware of the existence of other blocks extending the same parent block.

Reorganization. When a newer branch gets chosen by the fork-choice rules and thus has to overwrite the already accepted branch, some blockchain states need to be rolled back (e.g., transactions in the overwritten blocks are marked as unspent) and some new state changes need to be applied (e.g., transactions in the new branch are marked as spent). We call this process a block reorganization (or simply a reorg) and it is a critical part of the fork-choice rules in consensus algorithm.

III. Threat Model

Discrepancies in the fork-handling implementation of blockchain clients pose risks due to potential exploitation by malicious actors. The adversaries we consider in this paper aim to create inconsistent blockchain states (e.g., different branches being chosen by different clients) in PoW/PoS blockchain networks. If attacks are successfully mounted, target clients would have a blockchain state that differs from the rest of the network. Consequently, services relying on these inconsistent blockchain states may experience undesired outcomes. For instance, a cryptocurrency exchange receiving blockchain states from a targeted client might produce incorrect trading results, or a mining pool might generate a block that is rejected by the rest of the network, thereby wasting its mining power.

The impact of these attacks depends on the size of the affected client population in the network. When the target nodes of an attack constitute the majority of the network, the adversary can cause global damage, making his/her branch the canonical chain permanently. One reorg bug we discover in Ethereum (see §VIII-B) would fall into this category, as the target client families, Prysm and Teku, together account for more than 50% of the Ethereum network. In contrast, when the target nodes are the minority, the created fork (or view split) would eventually be resolved. However, the temporary split can offer a window of opportunity for attacks. Several centralized exchanges (such as Binance or Bithumb) rely on one-block confirmation [7], [12], meaning a short-lived (e.g., 1-2 block time) state inconsistency in the target clients can be exploited to cause localized damage. The chain-tip switching bug in Bitcoin (see §VIII-A) would be an example of such local-damage attacks. Or, the temporary fork may persist for an extended period with certain strategies. The recent balancing attack strategy [40] that can maintain a long-lived fork (by balancing two branches for several slots) in Ethereum 2.0 is one example.

To exploit these discrepancies in fork-handling logic, adversaries with some mining power or staking power should create and send one or more blocks and/or attestations to the affected clients. It is a realistic attack capability in modern PoW/PoS blockchains because rational miners or validators in practice often use their power to generate blocks with malicious intent when the expected return is higher than the benign block reward. For instance, Bitcoin has recently experienced intentionally created invalid blocks in both the mainnet and testnet by rational miners [2], and Ethereum has had more than 200 validators slashed for attesting rule violations since its transition to PoS in 2022 [6].

Scope. Our main goal is to identify implementation discrepancies in fork-handling logic that can be exploited to cause view splits in PoW/PoS blockchain networks. Confirming whether a view split due to a discrepancy is permanent or temporary is beyond the scope of this work because it depends on external factors such as the affected client population, as discussed above. We target the fork-handling logic implementations of PoW/PoS systems, not necessarily covering the entire blockchain implementations. This makes FORKY complementary to other, more general blockchain fuzzers [57], [17], [37].

IV. Forky Overview

In this section, we introduce a motivating example that highlights a specific challenge in blockchain consensus fuzzing. Following this, we briefly outline how FORKY tackles this challenge by employing fork-aware mutation strategies and fork-diversifying feedback mechanisms.

A. Motivating Example

Blockchain consensus presents unique challenges to fuzzing due to the interconnected nature of multiple blocks through persistent state variables. Refer to the motivating example in Figure 2, which showcases a simplified PoW blockchain consensus code. This code contains persistent blockchain state variables and two primary functions for block processing. Although our system operates across different programming languages, we have provided this example code in simplified `C` style to facilitate easier understanding.

First of all, the global state variables play a crucial role in storing critical blockchain states, undergoing consistent updates throughout the execution of input blocks. The state variable `chain` manages the global state of the blockchain, which encompasses all blocks (in hash format) from the genesis block, and the reference pointer to the latest block in the main chain, called the head. Additionally, the variable `mempool` holds instances of *coin*, which represents the cryptocurrency containing information of the amount and cryptographic proof of ownership.

```
1  // Global persistent state variables
2  ChainState chain;  // Chain state variable
3  Mempool mempool;   // Mempool (unspent coins)
4
5  function ProcessBlock(Block b) {
6    ...
7    if (sha256d(b) < chain.target_pow) {
8      if (chain.contains(b.parent)) {
9        chain.add(b);
10       if (b.cum_work > chain.cum_work) { // Begin Fork choice
11         if (b.parent == chain.head) {
12           chain.head = b;
13           chain.cum_work += b.work;
14           mempool.delete(used_coin);
15           mempool.add(new_coin);
16           ... // Update other persistent state variables
17         } else { Reorganize(b); }
18 } } } }
19 function Reorganize(Block b) {
20   common_ancestor = FindCommonAncestor(chain.head, b);
21   ... // Revert blocks from the current head to common_ancestor
22   for (block in [common_ancestor, ..., b]) {
23     if (chain.Verify(block.header) == VALID) {
24       chain.head = b; ... // Update other persistent state variables
25       if (mempool.contain(used_coin)) {
26         mempool.delete(used_coin);
27         mempool.add(new_coin);
28       } else { bug(); }
29 } } }
```

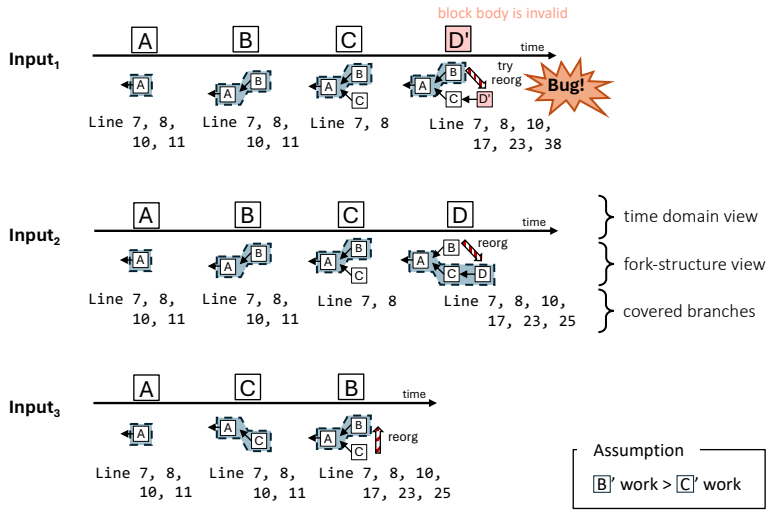


Fig. 2. A simplified consensus program (left) and three example inputs (right). We illustrate how a series of blocks in each input lead to different fork-structure states, resulting in different covered branches of the consensus program.

When a client receives a series of blocks, it processes each block in sequence. For each block, the function `ProcessBlock()` is invoked first, which checks the proof of work (Line 7) and whether the block extends any block in the chain (Line 8), Next, a fork-choice operation begins; cumulative works of `b` and `chain_head` (i.e., head of the winning branch) are compared (Line 10). If the new block extends the current head (Line 11), then the client adds the block at the end of the winning branch and updates `chain`, `mempool`, and other persistent state variables (Line 12-16). Otherwise, the `Reorganize()` function is invoked.

The `Reorganize()` function rolls back all transactions in the old branch up to the `common_ancestor`, where the new branch diverges from. Note that many details are omitted for brevity; yet, we highlight the multi-level, nested conditions (Line 23, 25) for the `bug()` to be triggered (Line 28). This bug is triggered when the mempool does not have the memory of the coin to delete (Line 25) and this condition depends on the global state variables, which could have been altered by previous block processing (e.g., block body contains duplicate coins). Note that this simplified bug example is inspired by one of our own case studies outlined in §VIII-A.

At first glance, satisfying the conditions within each function to trigger the bug may not appear so challenging. For instance, by presenting a series of blocks $[\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}']$ to the client (see the example Input_1 in Figure 2), one could induce a fork — notably at the insertion of `C` (as shown in fork-structure view) with the assumption that block `B`'s work exceeds that of block `C`. At the submission of $\texttt{D}'$, the block penetrates Line 17 and triggers reorganization because it tries to extend the losing chain (Line 11). During the reorganization, $\texttt{D}'$ is found to contain a coin (i.e., transaction) that is not present in the mempool (Line 25) while its header is valid (Line 23), which in turn triggers the bug (Line 28).

Generating such a precise sequence of inputs with state-of-the-art fuzzers is, however, far from straightforward. Fuzzers lacking an understanding of blockchain's persistent state variables, like the fork structure, would require a rare chance to append a block at the end of the losing chain and satisfy the condition in Line 11 to activate `Reorganize()`. Furthermore, penetrating through nested conditions to reach Line 28 would require a series of uncommon mutations to the input blocks and result in meaningful changes of the values of state variables. Taking Input_1 as an example, a specific coin can be added to *mempool* through state changes (Line 14, 15) during the reception of C. This coin will trigger the bug after the reception of D', if it gets deleted twice during reorganization (Line 26) and eventually dissatisfying the condition at Line 25. Although there exists a slim chance to discover a block input that satisfies all these conditions, it is undesirable to depend on extremely rare luck through random mutations.

Some might still argue that with enough time, after random mutations of blocks and transactions, state-of-the-art fuzzers would ultimately discover inputs that form a specific fork structure. For instance, Input_2 in Figure 2 already triggers `Reorganize()` (Line 17), visits Line 25, but not the bug (Line 28) yet. Through some random mutations of block `D`, a fuzzer could potentially find a critical input similar to Input_1. However, we show that reliably testing such a specific sequence in blockchain is *not* as straightforward as it appears. Even if a sequence of input blocks $[\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}]$ is generated by randomly attempting various block orders, achieving a reliable, meaningful test is challenging because code coverage is *not* sensitive enough to the changes in the blockchain states. For instance, consider two block sequences $\text{Input}_2 = [\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}]$ and $\text{Input}_3 = [\texttt{A}, \texttt{C}, \texttt{B}]$, which achieve the *same* branch coverage because both should handle new fork cases and resolve them through reorganization. Thus, if Input_3 is already present in the test case corpus, the fuzzer would miss the opportunity to add Input_2 to the corpus, despite its importance in covering significant inputs like Input_1.

B. Our Approach

FORKY tackles the highlighted challenges by emphasizing the analysis of persistent blockchain states, specifically fork

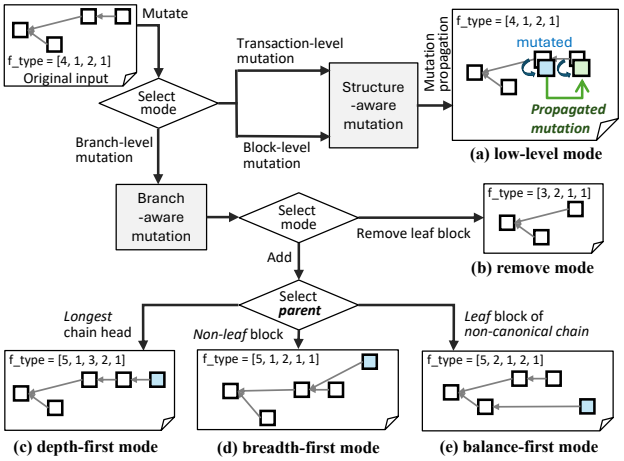


Fig. 3. Overview of mutation operations in FORKY.

structures, during fuzzing processes. With the knowledge of the fork structure of fuzz inputs, FORKY performs fork state-aware fuzzing to produce inputs that exhibit a wide range of blockchain states.

FORKY is a differential fuzzing framework that tests multiple target blockchain clients using the same blockchain inputs to detect discrepancies in their output blockchain states, such as view splits. Inputs from the shared corpus are mutated to trigger potential discrepancies and the notable ones are later added to the corpus. We highlight the two components that address the outlined challenges.

1) Fork-aware mutation strategies (§V). FORKY performs state-aware fuzzing and is able to generate inputs exhibiting highly diverse blockchain states, enhancing the discovery of critical fork-handling logic discrepancies. Unlike conventional structure-aware mutations, which randomly generate blockchain inputs at the transaction and block levels, our mutation strategies are expressly designed to produce inputs with forks. These strategies are also designed to promote or avoid certain types of fork-handling events to render the testing more effective.

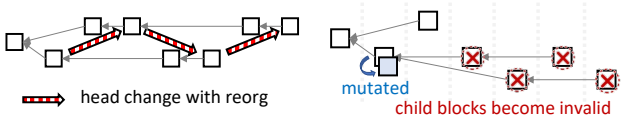
2) Fork-diversifying feedback mechanism (§VI). Our second contribution is a new feedback mechanism that can explore unseen fork-handling events. Our feedback complements code coverage feedback by evaluating the novelty of fork events tested compared to those previously examined. By selecting fuzz inputs showcasing new and different fork events for the corpus, we ensure a broader array of blockchain states are tested, leading to more efficient and effective testing, like testing with the critical Input_1 in Figure 2.

V. FORK-AWARE MUTATION STRATEGIES

The goal of our fork-aware mutation design is to effectively adjust fuzz inputs with the knowledge of the fork structure of the inputs. To achieve this, we propose a number of mutation strategies for FORKY.

A. Mutating Input Fork Structures

In PoW/PoS blockchains, the fork structure of a blockchain input represents a tree of blocks, where each block can contain



(a) Rare but interesting forks **(b) Ineffective mutation**

Fig. 4. Example fork mutations we promote (a) and avoid (b).

multiple transactions. Thus, we mutate the input fork structure at three levels: branch-level, block-level, and transaction-level, as shown in Figure 3. FORKY first determines the mutation level (branch, block, or transaction) and then applies the corresponding mutation operations to the input. Figure 3 showcases five mutation modes ((a)–(e)) that FORKY uses for making effective fork-structure mutations. Let us present three main mutation strategies of FORKY and how these mutation modes are used.

Mutation strategy 1. Mutating with fork structures. This first strategy describes how FORKY in general mutates the fork structure of the input to test various fork-handling logics effectively. Mutating the input at the branch level, FORKY modifies the fork structure of the input. For example, the remove mode in Figure 3(b) deletes one block from the input fork structure. Figure 3(c)–(e) show the three systematic mutation modes that FORKY uses to add a new block to the input. The depth-first mode in Figure 3(c) makes the chain longer by adding a new block to the longest branch of the tree. The breadth-first mode in Figure 3(d) adds a new block at non-leaf nodes, creating a new branch. The balance-first mode in Figure 3(e) is a special, blockchain-specific mode we introduce to create highly unlikely fork structures with long-lived competing branches; see more details later in this section. FORKY uses a combination of these operations to test diverse fork structures.

Notably, FORKY favors creating the fork structure that has *not* been tested before. FORKY evaluates whether the fork structure of the new mutated input has a new fork or overlaps with existing fork structures before conducting the mutation. For this, FORKY checks the isomorphism of the unordered rooted tree of the mutated inputs. To be more specific, FORKY assigns the *tree isomorphism code* [55] of the root block of the mutated input (denoted as 'f_type' in the Figure 3) and checks whether it is a new fork structure, which can be computed in linear time [28], [1]. This way, the mutation can continuously guide the fuzzer to test unseen types of fork structure and thus potentially unseen fork-handling cases.

Mutation strategy 2. Promoting rare fork structures. The above mutation strategy above already enables testing of inputs with various forms of fork structures. Yet, it would require a significant amount of fuzz energy to test inputs with certain types of fork structures. A blockchain input with long-lived competing branches is one such example. As shown in Figure 4(a), two long-lived branches of blocks compete as they grow over time, trigger unique fork-handling logics; e.g., overwriting and rolling back may block and transaction variables repeatedly over multiple fork-handling events. A mere fork-aware mutation (e.g., Figure 3(c) and (d)) may not

be able to create such long-lived competition between branches effectively. For this, we introduce a special mutation operation, the *balance-first* mode in Figure 3(e), that explicitly promotes the competition between branches by empowering the non-canonical yet long-lived branch. This mode is selected with a certain probability, offering a chance to create highly unlikely fork structures with long-lived competing branches.

Mutation strategy 3. Avoiding ineffective mutations. When the mutation is conducted at the block or transaction level, the mutation operation may affect the validity of the blocks or transactions in the fork structure. While this is useful for testing the robustness of the target client in general, it may lead to ineffective fuzzing in the blockchain context. See Figure 4(b) as an example, where a single block mutation that makes the block invalid may significantly reduce the efficacy of testing because the mutation invalidates many other blocks in the fork structure of blocks. To handle this, FORKY propagates the mutation operations according to the fork-structure view of the blocks in the input. Figure 3(a) shows the low-level mode that propagates the mutation operations in the downward order of blocks (i.e., from parent to all child blocks) to avoid invalidating many blocks in the fork structure.

B. Mutation for PoS: Votes and Times

There exist two more dimensions in blockchain inputs that have been introduced by the Proof-of-Stake (PoS) consensus algorithms: votes and arrival times. In PoS systems, each validator node can cast explicit votes (as known as attestations in Ethereum) to blocks that it considers valid and more appropriate to be included in the main chain. And the precise arrival time of each block and vote is also critical in PoS because it can affect the weight of the block in the fork-choice rules [4]. Therefore, for the same given fuzz input, mutating minute timing information and altering few votes in PoS systems can create various fork-handling events.

Mutation strategy 4. Voting-and-timing mutation. FORKY mutates several fields of the votes (attestations in Ethereum) messages. For example, FORKY randomly changes the target block of the vote at each mutation operation. Also, the arrival time of each vote is mutated to test scenarios where some votes arrive late or early, which is critical in PoS systems.

VI. FORK-DIVERSIFYING FEEDBACK

FORKY's fork-aware mutation strategies enhance its ability to test various mutations of block fork structures. However, the new mutation strategies alone are insufficient to test many critical fork structures because the code coverage feedback is insensitive to blockchain state changes, as demonstrated in §IV-A. In this section, we introduce a novel feedback mechanism based on fork structures designed to overcome the shortcomings of traditional code coverage feedback. This new mechanism is intended to complement, rather than replace, the existing code coverage-based feedback.

A. High-level Intuition

Recall from §IV-A that consensus implementations in blockchain are *stateful*, thus discerning interesting inputs such as Input_2 in Figure 2 is hard due to the insensitivity of traditional code coverage metrics. Losing Input_2 in the corpus would significantly reduce the chance of finding the discrepancy (which can be triggered with Input_1).

Addressing this challenge would require FORKY to continuously explore more diverse blockchain states or fork structures in addition to finding new code paths. This additional feedback mechanism can help FORKY to find fuzz inputs with more diverse fork structures even when they do not necessarily increase code coverage. In the PoW and PoS blockchain contexts, we characterize this additional feedback as *fork-diversifying* feedback.

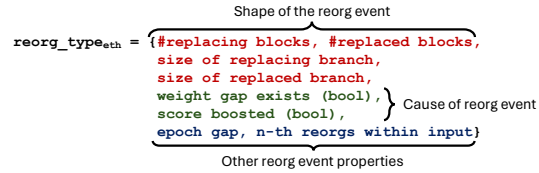
B. New Feedback

We propose an additional fork-diversifying feedback mechanism to complement code coverage. We leverage our domain knowledge to quantitatively define the types of fork-handling events and use these to add feedback on top of code coverage. With this additional feedback and the knowledge of fork types, FORKY finds the inputs that test new fork-handling events and eventually test previously uncovered code paths more effectively. We empirically find that the fork-diversifying feedback shows noticeably faster discrepancy detection performance in both Bitcoin and Ethereum.

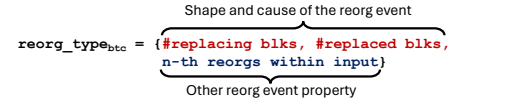
Types of forks. We quantify fork types within the input to enable continuous generation of new fork varieties. This approach aims to examine previously untested fork-handling logics. Our quantification centers on how a fork in a blockchain input is resolved (or how blocks are *reorganized*) by the consensus algorithm. Specifically, a reorganization represents a form of fork-handling operation, requiring the rollback (or restoration) of many transactions and blocks. By focusing on fork reorganizations, we can quantitatively assess the diversity of fork-handling logics. Notably, different reorganization strategies have recently been exploited in attacks against Bitcoin and Ethereum, as evidenced in literature [39], [40], [41], [42], [44], [47], [60]. By designing a quantitative metric of fork reorganization, we can expect to test a broader range of fork reorganization events, thereby uncovering more potential discrepancies in the fork-handling implementations.

To that end, we define *reorganization type* in the context of the environment and complexity in which each reorganization happens. For example, some reorg events can occur between long, competing branches while some may involve a single block reorganization. Moreover, the replacing block can be in a slot later than the replaced block (i.e., ex-post reorgs [47]) or the replacing block can be in the earlier slot (i.e., ex-ante reorgs [47]). We define a reorganization type to reflect these different styles of reorganization events with varying complexities. A unique reorganization type is defined as the relationship between two branches, critical condition for reorganizations (e.g., block order, timing, voting), number of reorganized blocks, etc.

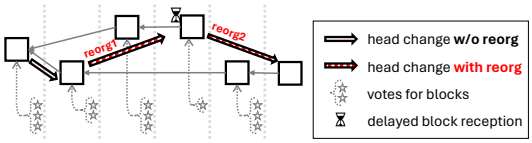
Let us present the reorganization type definition for Ethereum first because reorganization events in Ethereum's PoS consensus algorithm are more complex than those in



(a) Reorg type for Ethereum PoS consensus



(b) Reorg type for Bitcoin PoW consensus



✓ reorg1's type = {2, 1, 3, 1, 1, 0, 0, 1}

✓ reorg2's type = {3, 2, 5, 3, 1, 1, 0, 2}

(c) A blockchain input that triggers two reorg events (see the two different reorg types)

Fig. 5. The reorg type for (a) Bitcoin PoW and (b) Ethereum PoS consensus algorithm, and (c) examples of reorg type in Ethereum.

Bitcoin's PoW consensus algorithm. The Ethereum reorganization type in Figure 5(a) contains eight elements. The first four (i.e., the `number of replacing/replaced blocks` and the `size of replacing/replaced branch`) provide insights into the characteristics of the corresponding reorganization event (e.g., 2-slot size ex-ante reorganization [47]). Note that the `size of replacing/ replaced branch` is measured by the difference between the slot number of each head of the branch and the slot number of the common ancestor of the branches. The next two elements (i.e., the presence of a weight gap and the occurrence of a score boost) describe the underlying cause of the reorganization; e.g., when neither a weight gap nor a score boost is observed in a reorganization event, it implies that the reorganization was triggered due to a tie-break rule. The next `epoch gap` indicates the number of epochs a reorganization event spans; e.g., it has a non-zero value when a fork straddles two or more consecutive epochs. The inclusion of this element is important because recent attacks [5] have shown that some attacks can occur at the boundary of epochs. Finally, to help promote long-lived competition, the type includes the counter for how many reorgs occur within a given test case to differentiate multiple reorganization events in a single test case. Reorganization events in Bitcoin's PoW consensus algorithm are relatively simpler than those of Ethereum. Figure 5(b) illustrates a simpler reorganization type of Bitcoin.

We provide an example of reorganization types in Ethereum in Figure 5(c). This specific example input triggers two reorganization events; see two red-striped arrows. While these two reorganization events look similar and the second does not increase code coverage, we assign different reorganization types to them. According to our reorganization type definition for Ethereum found in Figure 5(a), the two types differ in several elements (see the figure for details) and thus our fuzzer can potentially store this input as a new interesting seed in the corpus.

Finally, let us explain how our fork-diversifying feedback mechanism adds novel fork inputs to the corpus. After mutating and testing a blockchain input, FORKY evaluates it to compute reorganization types present in the input. If an input exhibits a reorganization type that has not been previously encountered (when compared to the record FORKY maintains throughout the fuzzing campaign), it is added to the corpus. Note that this feedback mechanism does not replace the existing code coverage feedback; instead, it enriches it by adding more interesting inputs to the corpus. This feedback mechanism enhances the likelihood of uncovering yet-untested discrepancies in the implementations of various fork-handling logics.

VII. Implementation and Evaluation

We present our implementation of FORKY for Bitcoin and Ethereum clients (§VII-A) and their performance (§VII-B).

A. Implementation

With FORKY, we test Bitcoin and Ethereum — the two largest (in terms of market capitalization) PoW and PoS blockchain projects, respectively. Since their PoW and PoS consensus algorithms are incompatible with each other, we test them separately with two fuzzers: FORKY-Bitcoin and FORKY-Ethereum. We open source both implementations [3].

FORKY is built on top of the libFuzzer library [35] using C++ (for FORKY-Bitcoin) and Rust (for FORKY-Ethereum). Particularly, we base our implementation on code coverage feedback, corpus maintenance, and seed input selection from the default libFuzzer framework and further improve it with our fork-aware mutation and fork-diversifying feedback. All Bitcoin clients run in the regression test mode (i.e., *regtest*). For FORKY-Ethereum, we build on top of the beacon-fuzz [48] and use the spec test environment [24] that has been used for executing test cases.

For FORKY-Bitcoin, we test a total of 30 Bitcoin clients across four client families. This includes Bitcoin Core (C++) from 0.15.0 to 27.0 (26 versions); Bitcoin Knots (C++) 22.0 and 23.0 (2 versions); btcd (Go) 22.1, 23.0, and 23.1 (3 versions); and bcoin (Javascript) 2.0.0 to 2.2.0 (3 versions). For FORKY-Ethereum, we test a total of four Ethereum consensus[2] clients across four client families. This includes Lighthouse 3.5.1 (Rust); Prysm 4.0.7 (Go); Teku 23.6.2 (Java); and Nimbus 23.5.1 (Nim).

B. Evaluation

We evaluate how effective FORKY's main features, i.e., the fork-aware mutation and fork-diversifying feedback mechanism, are at finding discrepancies in fork-handling logic. For this, we compare the proposed full implementation of

[2]After switching to PoS in September 2022, Ethereum's execution clients, such as Geth, no longer handle consensus logic and thus execution clients are out of the scope of this work.

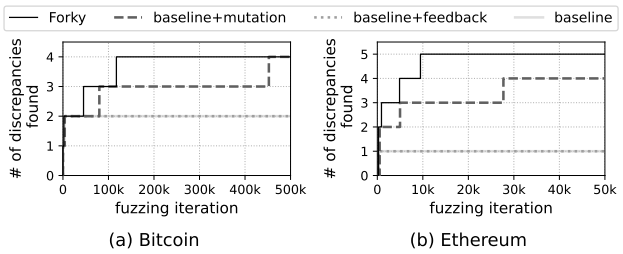


Fig. 6. Performance of FORKY and its variants.

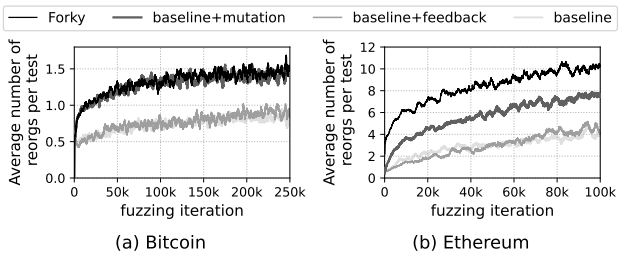


Fig. 7. Numbers of reorgs triggered per test case with FORKY and its variants.

FORKY with several variants of FORKY, which are inferior implementations of FORKY. We show that the FORKY's core features are indeed effective at finding discrepancies.

- 'baseline' denotes a coverage-guided structure-aware fuzzer,
- 'baseline+mutation' denotes a coverage-guided structure-aware fuzzer with the fork-aware mutation,
- 'baseline+feedback' denotes a coverage-guided structure-aware fuzzer with the fork-diversifying feedback mechanism, and
- 'Forky' (i.e., 'baseline+mutation+feedback') denotes a coverage-guided structure-aware fuzzer with the fork-aware mutation and fork-diversifying feedback.

1) Findings: Table I lists the discrepancies detected by FORKY. It identifies six previously unknown discrepancies, two instances of rediscovered known bugs, and one variant of an existing bug. In Bitcoin, one discrepancy (#4) could lead to a view split in bcoin. One known bug (#2) and a new variant of a known bug (#3) are detected. Also, one unknown bug (#1) could disrupt chain growth by inhibiting the appending of new blocks. In Ethereum, FORKY detects discrepancies in the tick operation (#6) and the future attestation processing logic (#5-8), which could cause view splits among Ethereum clients. The discrepancy (#9) can cause resource exhaustion due to orphan threads. The list also includes bugs in the standardized test suites (i.e., Ethereum's spec tests). This suggests that the current testing environment for Ethereum consensus clients is unsatisfactory to test the correctness of the fork-resolution logic in these Ethereum clients.

Fixing these discrepancies requires different levels of effort. The discrepancies (#1,3) in Bitcoin Core can be mitigated by adding additional incorrect block verification [10]. The discrepancy (#2) in Bitcoin Core can be fixed by adding code for SegWit handling [36]. The bcoin bug (#4) can be mitigated by reverting the chain tip after block body validation if the validation fails. The orphan thread issue (#9) in Nimbus can be prevented by adding 2-3 lines of code for exception handling. The remaining discrepancies in Ethereum (#5-8) require more substantial changes to the clients' fork-resolution logic because they stem from ambiguous specifications. For example, Prysm and Teku would have to change their core data structure to address the conflicting future attestations (#7).

2) Performance: Figure 6 shows the number of discrepancies found as FORKY and three variants progress. In both Figure 6(a) and Figure 6(b), we clearly see that the full FORKY fuzzer finds the discrepancies faster than all other inferior versions of FORKY, showing the overall effectiveness of FORKY. In Figure 6(a), the 'baseline+mutation' strategy also finds all Bitcoin discrepancies within 500K iterations (about 12 hours of execution) but it took longer to find all discrepancies than FORKY. Figure 6(b) shows that the 'baseline+mutation' fails to find one discrepancy (i.e., #7 in §VIII-B) within 50K iterations (about 166 hours of execution). We stop at 100K iterations but it still does not find this discrepancy. In both Bitcoin and Ethereum, the 'baseline+feedback' fuzzers show no improvement over the 'baseline' fuzzers, showing that the fork-diversifying feedback alone would not be useful.

3) Number of Reorgs: To better understand the effectiveness of FORKY's fork-aware mutation and fork-diversifying feedback, we measure the number of reorgs per test case during the fuzzing campaigns. Figure 7 shows that FORKY indeed generates more reorgs than all other variants. Also, the 'baseline+mutation' fuzzers generate much more reorgs than the 'baseline' fuzzers, indicating that the fork-aware mutation is indeed the key to generating more reorgs. Each line in Figure 7 represents the moving average (with a window size of 2,000) of the number of reorgs.

The effectiveness of the fork-diversifying feedback in generating more reorgs though is not universal across Bitcoin and Ethereum, and it appears to be dependent on the existence of the fork-aware mutation in the fuzzing campaign. In our Bitcoin testing, the fork-diversifying feedback does not generate any noticeably more reorgs than the strategies without the feedback. This is because the reorg types in Bitcoin have a much lower dimension of freedom than those in Ethereum; see Figure 5(a) and Figure 5(b) for comparison. The limited dimensions of freedom in Bitcoin reorg types make the traditional coverage feedback dominate our new feedback. Yet, the fork-diversifying feedback shows a significant impact on the number of reorgs in Ethereum, especially when it is used with the fork-aware mutation. The effectiveness of the feedback reduces to none though when it is used without the fork-aware mutation (see the 'baseline+feedback' and 'baseline' lines), which confirms that the fork-diversifying feedback alone is not useful.

Notice the number of reorgs in Bitcoin is generally lower than that in Ethereum. This is because the PoW consensus algorithm in Bitcoin is much simpler than the PoS consensus algorithm in Ethereum, and thus it is less likely to trigger reorgs in Bitcoin.

4) Diversity of Reorgs: We measure whether the fork-diversifying feedback increases the diversity of reorg types in test cases. We use the reorg type definitions in Figure 5(a)

TABLE I
DISCREPANCIES FOUND BY FORKY.

#	Platforms	Clients	Implications	Descriptions	Known	Status
1	*	Bitcoin Core	denial-of-service	Incorrectly handled reorg failure repeats every block arrival, preventing its verification	No	Reported**
2	Bitcoin	Bitcoin Core	view split*	Clients reject blocks containing SegWit data in regression test mode	Yes [8]	N/A
3		Bitcoin Core	crash, double-spending	Verifying a double-spending block results in a valid confirmation or a crash	Yes [54]	N/A
4		bcoin	view split*	Inconsistent block verification selects different branches after a reorg	No	Confirmed**
5	Ethereum	Prysm	view split*	Prysm spec test fails to handle future attestations	No	Confirmed [30]
6		Prysm	view split*	Prysm spec test fails to correctly handle multiple steps at a slot border	No	Confirmed [29]
7		Prysm, Teku	view split*	Prysm and Teku handle conflicting future attestations in a non-deterministic way	No	Confirmed [33]
8		Teku	view split*	Teku's spec test fails to handle some future attestations	No	Reported [32]
9		Nimbus	orphan thread	Nimbus' spec test may face resource exhaustion when multiple test cases fail	No	Reported [31]

* When exploited, a view split may lead to a permanent chain split or a temporary one that gives adversaries a window of opportunity to launch attacks; see §III.
** Directly reported to developers.

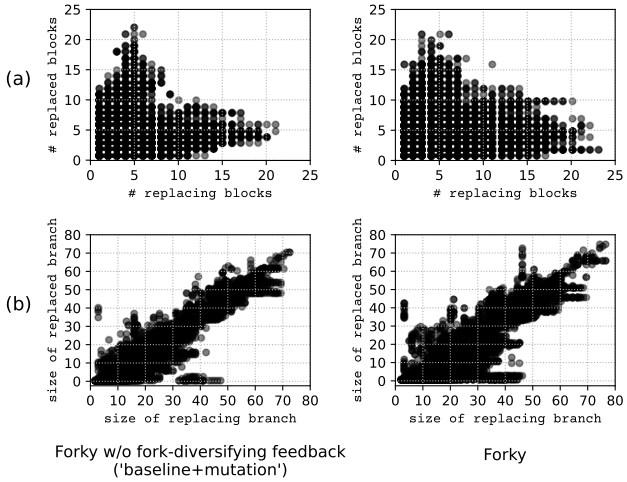


Fig. 8. Diversity of tested reorgs with FORKY and 'baseline+mutation' in Ethereum.

to analyze how diverse the tested reorg types are. Note that, when it comes to the diversity of reorg types, we focus on the fuzzing on Ethereum, not Bitcoin, since the effect of the fork-diversifying feedback is shown to be marginal in Bitcoin.

Figure 8 shows the diversity of several elements in the reorg type that are used in feedback during fuzzing. The results on the left-hand side of the figure are from a fuzzing campaign with the 'baseline+mutation' fuzzers while the results on the right-hand side are from FORKY (thus, including the fork-diversifying feedback). Figure 8(a) and Figure 8(b) show how (`#replacing blocks`, `#replaced blocks`) and (`size of replacing branch`, `size of replaced branch`) are distributed in all the feedback used in each fuzzing campaign. The full 'Forky' fuzzers generate a more diverse set of reorgs (i.e., covering more combinations of these two features) than the one without the feedback.

5) Quantitative Comparison with Other Tools: Last, we provide a quantitative comparison between FORKY and other existing blockchain fuzzers, Fluffy [57], and LOKI [37]. We emphasize that a comprehensive comparison between these fuzzers with distinct scopes and specialized targets is inherently challenging. Yet, we offer our best effort for this comparison with several caveats attached. In particular to compare FORKY with Fluffy [57], and LOKI [37], we re-implement FORKY for Ethereum 1.0 (i.e., PoW Ethereum before 2022) using go-fuzz [56] on go-ethereum 1.9.24 (Geth) [53] and denote it as FORKY-Eth1.0 for this particular comparison.

We run FORKY-Eth1.0, Fluffy [57], and LOKI [37] for 6 hours[3]. Table II compares the performance of the three fuzzers in terms of statement coverage and the number of unique reorg types. We select five packages in Ethereum 1.0 and measure the statement coverage of each fuzzer to highlight the relative strengths of FORKY-Eth1.0 and the other fuzzers. FORKY-Eth1.0 achieves much higher coverage in the `core` and `trie` packages compared to Fluffy and LOKI. This is a clear indication of the relative strengths of FORKY in testing the central components of the blockchain system including block verification and fork-handling logic. Fluffy [57] tops in the `core/vm` package with extensive fuzz testing on EVM bytecode. LOKI [37] shows a substantially high coverage in the `p2p` and `rlp` (a message serialization method) packages because it performs extensive testing on peer-to-peer networking and serialization methods.

TABLE II
PERFORMANCE COMPARISON BETWEEN FORKY, FLUFFY, AND LOKI.

		FORKY-Eth1.0*	Fluffy [57]	LOKI [37]
statement coverage	`core`	**2,089**	1,512	1,890
	`trie`	**664**	463	397
	`rlp`	558	424	**669**
	`core/vm`	126	**1,061**	107
	`p2p`	33	91	**2,135**
unique reorg types		**212**	N/A	0**

* FORKY re-implemented for PoW-based Ethereum 1.0 for comparison.
** May increase to 1 depending on the block time setting.

FORKY-Eth1.0 tests 212 unique reorganization types while LOKI and Fluffy do not test any unique reorg types. Fluffy is shown to be totally incapable of triggering any reorgs because it uses a block data structure only as a container for transactions. While LOKI triggers 0 unique reorg types in this fair comparison (with the same Ethereum block time setting), we find that LOKI can often trigger one (but not more) unique reorg type when the block time is set smaller than the default 12 seconds. The tested reorg type in LOKI is yet limited to a simplest reorg type where one block replaces another due to network delays. This quantitative comparison shows that FORKY is clearly more effective at testing fork-handling logic in blockchains than the existing state-of-the-art blockchain fuzzers. This shows that FORKY's superior capability of testing fork-handling logic complements other blockchain fuzzers, ensuring thorough testing across all critical packages. We attach the *qualitative comparison* with other fuzzers (including Tyr [17]) in §X.

[3]Tyr [17] has not released its code, rendering comparison challenging.

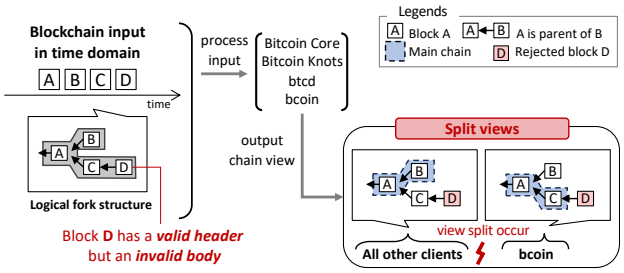


Fig. 9. Case 1 (#4): All bcoin clients switch their main chain to $[A \leftarrow C]$ when the header of block D is validated, and do *not* switch it back when the rest of block D data turns out to be invalid. All other clients do not switch their main chain until block D is fully verified.

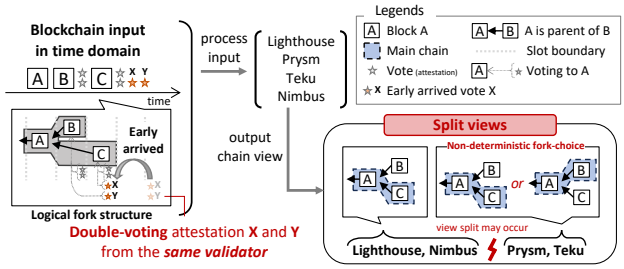


Fig. 10. Case 2 (#7): When double-voting attestations X and Y from a same validator arrive earlier, clients that use `queue` switch to $[A \leftarrow C]$. Prysm and Teku produce non-deterministic fork-choice results because they use `map` while others use `queue` for handling future attestations.

VIII. Case Studies

We present one case study in Bitcoin (§VIII-A) and another in Ethereum (§VIII-B). We end this section with a discussion on the implications of our findings in §IX-B.

A. Case 1: Reorganization Bug in Bitcoin (#4)

With Forky, we have found a semantic bug that causes every bcoin client we test (version 2.0.0, 2.1.0, and 2.2.0) to have their main chains different from the ones in all other Bitcoin clients. The root cause of this discrepancy is a subtle, inconsistent interaction between block verification and chain-head management in different Bitcoin clients. Bcoin builds the longest chain of blocks (i.e., the main chain) based on the validity of their headers only while other clients verify the entire block (header and transactions) before adding it to the main chain. This shows that Forky successfully generates a test case with a fork structure that triggers this bug in certain bcoin clients while the state-of-the-art differential testing tools fail to do so. We illustrate detailed steps of this bug in Figure 9.

B. Case 2: Data Structure Bug in Ethereum (#7)

Forky also discovers a subtle bug in Ethereum's fork-choice logic that causes non-deterministic fork-choice results in Prysm and Teku clients when handling future attestations. Future attestations occur when attestations arrive early from future slots due to local time differences or network delays. Figure 10 illustrates the impact of this bug. When this bug is triggered, Prysm and Teku produce non-deterministic fork-choice results while Lighthouse and Nimbus produce deterministic results.

The root cause of this discrepancy is the *different data structures* used in these clients to handle future attestations. The `map` data structure (used in Prysm and Teku) does not guarantee the order of arrived future attestations while `queues` (used in Lighthouse and Nimbus) guarantees their submission order. The problem arise from the fact that the Ethereum consensus specification does not offer a precise interpretation for handling future attestations.

IX. Discussion

A. Limitations of Forky

While Forky has shown its effectiveness in testing fork-handling logic in Bitcoin and Ethereum, it still has a limited scope of fuzzing capability, features, and target blockchain systems. We discuss these limitations in this section.

Limited fuzzing capability. There are a few fork resolution scenarios we have not covered in Forky mainly due to the practical resource limitations. For example, discrepancies that are triggered only when a fork straddles two or more epochs are not covered in Forky because it would require too many blocks in an input. We also have used only up to 128 validators for a committee for each epoch because we cannot simulate all 890K validators for our tests due to resource constraints.

Limited features. We focus on detecting errors in consensus algorithms and particularly their fork resolution mechanisms. Our tool may, therefore, miss errors in other parts of the blockchain client implementations. For example, Forky is not designed to detect bugs in the crypto libraries, peer-to-peer network protocol implementations, data parsing, etc.

Limited experimentations on other blockchains. We have implemented and tested Forky on Bitcoin and Ethereum 2.0 and Ethereum 1.0 the model consensus algorithms for many other PoW/PoS blockchains. Therefore, many copycats of Bitcoin and Ethereum, such as Litecoin [34] and Dogecoin [52], would require near-zero effort for Forky to be applied. Different PoW and PoS consensus algorithms may require custom changes to Forky. Forky's design, however, is based on the general principles of PoW and PoS consensus algorithms and thus we believe that Forky can be applied to other blockchains without forklift changes. To be specific, our fork-centric fuzzing strategies may need to be re-implemented based on the domain knowledge of each target blockchain system (e.g., re-defining the structure of block, transaction, or consensus messages) while the core of Forky can be reused.

B. Implications and Responses

Many findings from Forky have been acknowledged by the developers. For example, the bcoin developers have acknowledged the bug (§VIII-A) through private communication. The Prysm and Teku teams have also acknowledged that the handling of future attestations in Prysm (#5) is different from the two other clients. Some have not been acknowledged yet due to the lack of timely responses from the developers.

So far, developers of the affected clients tend to refuse to fix the discrepancies as they consider them to be not critical or exploitable at present. Two most cited reasons are (1) the discrepancies require someone to invest a significant amount

of PoW power to craft a malicious input to trigger them, or (2) the discrepancies are triggered only when validators are willing to be slashed (e.g., by sending double-voting attestations).

Yet, we argue that these discrepancies are better to be fixed, regardless of their exploitability at present, for two reasons. First, it is now in the realm of possibility that some adversaries with mining or staking power would risk their power to cause these discrepancies. There is a growing number of incidents where rational miners or validators are willing to risk their PoW or PoS power to gain other sources of profit, such as exploiting MEV opportunities [21]; e.g., some Bitcoin miners intentionally created invalid blocks in both the mainnet and testnet [2], and some (more than 200 so far) validators have been slashed for attestation rule violations [6]. Bugs like Case 1 and Case 2 in this section could be triggered by such miners or validators.

Second, perhaps more fundamentally, a latent bug that is unexploitable at present because of external constraints (e.g., particular input sanitization placed currently) is essentially a latent bug that may be exploitable in the future when the codebase of the affected clients propagates to other projects or when the constraints are relaxed due to code changes. Blockchain codebases typically have regular updates and are often used as a reference implementation for other blockchain projects [46], [59].

C. Testing Framework in Ethereum

One unexpected result from our evaluation is that there are not significantly many bugs in Ethereum's fork resolution mechanism compared to that in Bitcoin, even though the former is much more complex than the latter. We believe that this is because Ethereum has practiced more systematic testing than Bitcoin.

Ethereum's official implementation spec. Bitcoin does not have a formal specification of their client implementations [11]. Instead, they have a reference implementation (i.e., Bitcoin Core) that is considered as the de facto specification. Ethereum, in contrast, has improved the state-of-the-art by providing an official specification of its client implementations, written in Python, called PySpec [25]. When third-party developers implement their own Ethereum clients, they refer to this implementation spec to make sure that their implementations are consistent with the official one.

Spec test. In addition to having an official implementation spec, Ethereum has developed a systematic testing suit, called *spec test* [24], to test its diverse client implementations. The spec test is a collection of test cases that are designed to test the client implementations. While the spec test is a great tool for testing Ethereum clients, it supports *only manual testing*. Our contribution with FORKY provides useful complementary tools to the spec test by automating the testing process and diversifying the testing scenarios.

X. Related Work

We discuss related work to FORKY, the first fork state-aware differential fuzzing for PoW and PoS blockchains.

TABLE III
COMPARATIVE ANALYSIS OF FORKY, TYR, LOKI, AND FLUFFY.

	FORKY	Tyr [17]	LOKI [37]	Fluffy [57]
Differential testing	Yes	No	No	Yes
Feedback on code coverage	Yes	Yes	No	Yes
State-awareness	Yes (focusing on forks)	Limited*	Coarse-grained**	No
Linear chain structure-aware	Yes	Yes	No	Yes
Fork structure-aware	Yes	No	No	No
Testing of forks	Comprehensive	Occasional	Occasional	No
Ability to test rare forks	Yes	No	No	No
Supported protocol types	PoW/PoS	PoW/BFT***	PoW/BFT	PoW

* States are confined to selected state variables: the number of elected leaders, local transaction pool, local blockchain data, and the height of the chain.
** States are defined by the observed consensus message types, resulting in coarse-grained client states.
*** BFT stands for Byzantine Fault Tolerant protocols, such as PBFT [15], DiemBFT [22].

Fuzzing blockchain systems. Researchers have applied fuzzing to find bugs in the blockchains. Bitcoin Core developers implement a grey-box fuzzer [9], [27] and participate in an open-source fuzzing framework called OSS-Fuzz by Google [43]. Ethereum clients have applied fuzzing on a functional level with structure-aware inputs [45], [50], [20], [51]. Some of the earliest studies are EVMLab [23] and EVMFuzzer [26], which use differential fuzzing to find discrepancies in the EVM states between different Ethereum clients. A work Chronos [16] focuses on finding timeout bugs by fuzzing the deep-priority transient delay in the distributed system. While these efforts may find certain types of bugs, such as crash and memory corruption, they lack a specific focus on finding semantic bugs in consensus implementations by testing fork-handling logic.

Fuzzing blockchain consensus. A few recent studies, such as Tyr [17], LOKI [37], and Fluffy [57], try to find bugs in blockchain consensus implementations. While these fuzzing tools are shown to be effective, they are different from FORKY in several aspects, as illustrated by our qualitative comparison in Table III. In terms of fuzzing types, FORKY and Fluffy are differential testing tools whereas Tyr and LOKI test individual clients. Also, FORKY, Tyr, and Fluffy use code coverage feedback, whereas LOKI does not. Additionally, the types of blockchain protocols supported vary among these tools. Tyr and Fluffy are only capable of recognizing the linear chain structures (i.e., transactions and blocks form a linear chain) but unaware of how chains form forks and how forks are resolved. Therefore, Tyr and LOKI can only occasionally test forks, relying on rare lucky circumstances (See Table II).

XI. Conclusion

One unique feature of blockchain consensus algorithms is resolving conflicting inputs according to clearly defined fork-choice rules. Testing this highly critical feature in multiple client implementations has been a challenge in existing work. With our tool FORKY, we show that automated testing of fork-handling logic is feasible and effective in finding implementation discrepancies in PoW and PoS blockchains.

Acknowledgment

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2025-RS-2023-00259099) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

REFERENCES

[1] A. Abboud, A. Backurs, T. D. Hansen, V. Vassilevska Williams, and O. Zamir, "Subtree isomorphism revisited," *ACM Transactions on Algorithms (TALG)*, vol. 14, no. 3, pp. 1–23, 2018.

[2] O. Adejumo, "Marathon Digital confirms it mined invalid Bitcoin block," *CryptoSlate*, 2023, https://cryptoslate.com/marathon-digital-confirms-mining-invalid-bitcoin-block/.

[3] anonymous authors, "Forky repository," https://github.com/NetSP-KAIST/forky/releases/tag/v0.0.1-alpha.1, 2024.

[4] A. Asgaonkar, "Proposer LMD Score Boosting," https://github.com/ethereum/consensus-specs/pull/2730, 2021.

[5] ——, "Fork choice bugfix disclosure," https://notes.ethereum.org/@djrtwo/2023-fork-choice-reorg-disclosure, 2023.

[6] beaconcha.in, "Slashed validators," https://beaconcha.in/validators/slashings, 2022.

[7] Binance, "Binance reduces the number of confirmations required," https://www.binance.com/en/support/announcement/binance-reduces-the-number-of-confirmations-required-for-deposits-withdrawals-on-btc-and-eth-networks-360030775291, 2019.

[8] Bitcoin Core, "Todo: fix the code to support segwit block," https://github.com/bitcoin/bitcoin/blob/8b67698420e23db20cfbb9228ca01a68c8ddc10c/src/test/test_bitcoin.cpp#L45.

[9] ——, "Fuzz-testing Bitcoin Core," 2016, https://github.com/bitcoin/bitcoin/blob/v0.14.0rc1/doc/fuzzing.md.

[10] ——, "CVE-2018-17144 Full Disclosure," 2018, https://bitcoincore.org/en/2018/09/20/notice.

[11] Bitcoin.org, "Bitcoin Reference," 2009, https://developer.bitcoin.org/reference/intro.html.

[12] Bithumb, "Bithumb deposit and withdrawl status," https://en.bithumb.com/coin_inout/compare_price, 2015.

[13] V. Buterin, "Ethereum white paper: A next-generation smart contract and decentralized application platform," *Ethereum White Paper*, 2014.

[14] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining GHOST and Casper," 2020.

[15] M. Castro, B. Liskov *et al.*, "Practical Byzantine Fault Tolerance," in *Proc. USENIX OSDI*, 1999.

[16] Y. Chen, F. Ma, Y. Zhou, M. Gu, Q. Liao, and Y. Jiang, "Chronos: Finding Timeout Bugs in Practical Distributed Systems by Deep-Priority Fuzzing with Transient Delay," in *Proc. IEEE S&P*, 2024.

[17] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *Proc. IEEE S&P*, 2023.

[18] clientdiversity.org, "Ethereum Client Diversity," 2021, https://clientdiversity.org.

[19] "Coin Dance: Bitcoin Nodes Summary," 2021, https://coin.dance/nodes.

[20] ConsenSys, "Teku," https://consensys.net/knowledge-base/ethereum-2/teku, 2018.

[21] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *Proc. IEEE S&P*, 2020.

[22] Diem Association, "Welcome to the diem project," https://www.diem.com/en-us/, 2019.

[23] Ethereum Foundation, "EVM lab utilities: Utilities for interacting with the Ethereum virtual machine," 2017, https://github.com/ethereum/evmlab.

[24] ——, "Ethereum Proof-of-Stake Consensus Spec Tests," https://github.com/ethereum/consensus-spec-tests, 2019.

[25] ——, "Executable Python Spec (PySpec)," https://github.com/ethereum/consensus-specs/blob/dev/tests/core/pyspec/README.md, 2020.

[26] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "EVMFuzzer: Detect EVM Vulnerabilities via Fuzz Testing," in *Proc. ACM ESEC/FSE*, 2019.

[27] A. Groce, K. Jain, R. van Tonder, G. T. Kalburgi, and C. L. Goues, "Looking for Lacunae in Bitcoin Core's Fuzzing Efforts," in *Proc. ICSE-SEIP*, 2022.

[28] J. E. Hopcroft and R. E. Tarjan, "Isomorphism of planar graphs," in *Proc. Symposium on the Complexity of Computer Computations*. Springer, 1972, pp. 131–152.

[29] "Time moves during fork choice spec test," https://github.com/prysmaticlabs/prysm/issues/12884, 2023.

[30] "Consider adding attestation deferring test vectors in the fork-choice spec test," https://github.com/ethereum/consensus-specs/issues/3498, 2023.

[31] "Not flushing taskpool when failure happens during spec test," https://github.com/status-im/nimbus-eth2/issues/5415, 2023.

[32] "fork choice spec test seems not fully handling future attestation," https://github.com/Consensys/teku/issues/7804, 2023.

[33] "Data structure discrepancy on deferred attestation," https://github.com/Consensys/teku/issues/7805, 2023.

[34] Litecoin Project, "Litecoin - Open source P2P digital currency," https://litecoin.org/, 2011.

[35] LLVM, "libfuzzer – a library for coverage-guided fuzz testing," 2003, https://llvm.org/docs/LibFuzzer.html.

[36] E. Lombrozo, J. Lau, and P. Wuille, "Segregated Witness (Consensus layer)," 2015, https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.

[37] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "LOKI: state-aware fuzzing framework for the implementation of blockchain consensus protocols," in *Proc. NDSS*, 2023.

[38] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf, 2008.

[39] R. Nakamura, "Analysis of bouncing attack on ffg," https://ethresear.ch/t/analysis-of-bouncing-attack-on-ffg/6113, 2019.

[40] J. Neu, E. N. Tas, and D. Tse, "Ebb-and-flow protocols: A resolution of the availability-finality dilemma," in *Proc. IEEE S&P*, 2021.

[41] ——, "Two more attacks on proof-of-stake GHOST/Ethereum," in *Proc. ACM ConsensusDay*, 2022.

[42] M. Neuder, D. J. Moroz, R. Rao, and D. C. Parkes, "Low-cost attacks on ethereum 2.0 by sub-1/3 stakeholders," in *Proc. GTiB*, 2020.

[43] "OSS-Fuzz - Google's continuous fuzzing service for open source software," 2016, https://github.com/google/oss-fuzz.

[44] U. Pavloff, Y. Amoussou-Guenou, and S. Tucci-Piergiovanni, "Ethereum Proof-of-Stake under Scrutiny," in *Proc. ACM/SIGAPP SAC*, 2023.

[45] Prysmatic Labs, "Prysm," https://prysmaticlabs.com, 2018.

[46] P. Reibel, H. Yousaf, and S. Meiklejohn, "Short Paper: An Exploration of Code Diversity in the Cryptocurrency Landscape," in *Proc. FC*, 2019.

[47] C. Schwarz-Schilling, J. Neu, B. Monnot, A. Asgaonkar, E. N. Tas, and D. Tse, "Three Attacks on Proof-of-Stake Ethereum," in *Proc. FC*, 2022.

[48] Sigma Prime, "beacon-fuzz," https://github.com/sigp/beacon-fuzz, 2019.

[49] ——, "Blockprint," https://github.com/sigp/blockprint, 2021.

[50] ——, "Lighthouse," https://lighthouse.sigmaprime.io, 2021.

[51] status.im, "Nimbus," https://nimbus.team, 2018.

[52] The Dogecoin Foundation & Dogecoin Project, "Dogecoin - An open-source peer-to-peer digital currency," https://dogecoin.com/, 2013.

[53] The go-ethereum Authors, "go-ethereum: Official go implementation of the ethereum protocol," https://geth.ethereum.org/, 2013.

[54] "CVE-2018-17144," 2018, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17144.

[55] G. Valiente, *Tree Isomorphism*. Springer Berlin Heidelberg, 2002, pp. 151–251.

[56] D. Vyukov, "go-fuzz: randomized testing for go," https://github.com/dvyukov/go-fuzz, 2015.

[57] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in ethereum via multi-transaction differential fuzzing," in *Proc. USENIX OSDI*, 2021.

[58] A. Yeow, "Global Bitcoin nodes distribution," 2021, https://bitnodes.io/.

[59] X. Yi, Y. Fang, D. Wu, and L. Jiang, "BlockScope: Detecting and Investigating Propagated Vulnerabilities in Forked Blockchain Projects," in *Proc. NDSS*, 2023.

[60] M. Zhang, R. Li, and S. Duan, "Max Attestation Matters: Making Honest Parties Lose Their Incentives in Ethereum PoS," in *Proc. USENIX Security*, 2024.