ĐArcher: Detecting On-Chain-Off-Chain Synchronization Bugs in Decentralized Applications

Anonymous Author(s)

ABSTRACT

Since the emergence of Ethereum, blockchain-based decentralized applications (DApps) have become increasingly popular and important. To balance the security, performance, and costs, a DApp typically consists of two layers: an on-chain layer to execute transactions and store crucial data on blockchain, and an off-chain layer to interact with users. A DApp needs to proactively synchronize its off-chain layer with the on-chain layer, otherwise, the inconsistent data in the off-chain layer could mislead users and cause undesirable consequences, e.g., loss of transaction fees. However, transactions sent to blockchain are not guaranteed to be executed and could even be reversed after execution due to chain reorganization. Such non-determinism in the transaction execution is unique to blockchain and DApp developers may fail to perform the on-chain-off-chain synchronization accurately due to their unfamiliarity of the complex transaction lifecycle.

In this work, we investigate the challenges of synchronizing on-chain and off-chain data in Ethereum-based DApps. We present two types of bugs that could result in inconsistencies between the on-chain and off-chain layers. To help detect such on-chain-off-chain synchronization bugs, we introduce a state transition model to guide the testing of DApps, and propose two effective oracles to facilitate the automatic identification of bugs. We build the first testing framework, *ĐArcher*, to detect on-chain-off-chain synchronization bugs in DApps. We have evaluated *ĐArcher* on 11 popular real-world DApps. *ĐArcher* achieves high precision (99.3%), recall (87.6%), and accuracy (89.4%) in bug detection and significantly outperforms the baseline methods. It has found 15 real bugs in the 11 DApps. So far, six of the 15 bugs have been confirmed by the developers and three have been fixed. These promising results demonstrate the usefulness of *ĐArcher*.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging.**

KEYWORDS

Software testing, Decentralized applications, DApps, Blockchain

ACM Reference Format:

Anonymous Author(s). 2021. *ĐArcher*: Detecting On-Chain-Off-Chain Synchronization Bugs in Decentralized Applications. In *Proceedings of The 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Decentralized Applications (DApps) are software applications running on a decentralized network like blockchain. Since the emergence of Ethereum [11], a blockchain platform that supports Turing-complete smart contracts, blockchain-based DApps have drawn much attention from both academia and industry. As of February 2021, there are over three thousand DApps and 100 thousand active DApp users, making over two million transactions per day [?].

As shown in Fig. 1, DApps are typically deployed as web applications, consisting of two layers: *on-chain* and *off-chain*. The former comprises a set of smart contracts that store and update crucial data on the blockchain.[1] The latter contains a user-friendly front-end client and an optional centralized service outside the blockchain [53]. Take Giveth [20], a popular DApp for charitable donation on Ethereum, as an example. It comprises a set of on-chain smart contracts, and an off-chain front end backed by a centralized cache server. To avoid frequent communication with the blockchain and facilitate users' queries, the off-chain layer usually stores processed or analyzed results of certain important on-chain data. Therefore, a DApp's state is composed of an *on-chain state* and an *off-chain state*, referring to the data stored at the on-chain (e.g., data in Giveth's smart contracts) and the off-chain (e.g., database of Giveth's cache server) layers, respectively. The on-chain and off-chain states of a DApp are not necessarily the same. DApps usually maintain a mapping between the on-chain and off-chain states. Since blockchain is autonomous and the on-chain state may change out of the control of DApps, DApps need to proactively synchronize their off-chain states and keep them consistent with the on-chain states. We call such a process *on-chain-off-chain synchronization*.

On-chain-off-chain synchronization can be complicated. Changes are made to the on-chain state of a DApp by sending transactions to blockchain. However, transaction executions are non-deterministic due to the decentralized nature of blockchain [59]. On the one hand, transactions sent to blockchain are not necessarily executed and could be dropped silently [10]. On the other hand, executed transactions could be reversed as a result of chain reorganization [27]. Such non-determinism is unique to blockchains, and does not exist in conventional centralized services or distributed systems. If the non-determinism is not carefully considered and dealt with in the development of a DApp, inconsistencies between its off-chain and on-chain states may arise when the DApp is running. Such inconsistencies can cause catastrophic consequences. It is because users typically take actions (e.g., buying and selling) according to the off-chain state shown at the front-end client of a DApp. Stale

ESEC/FSE 2021, 23 - 27 August, 2021, Athens, Greece

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

[1] To save transaction costs, blockchain typically only stores crucial data [55].

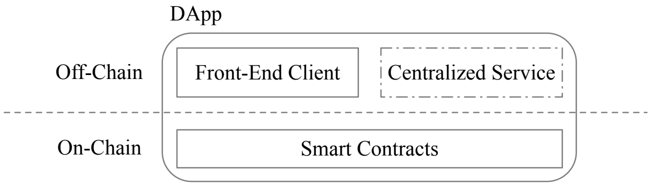

Figure 1: The Architecture of A Typical DApp

data at the off-chain layer can mislead users into taking wrong actions that result in irreversible changes on blockchain or financial losses. For example, in a DApp like Giveth, if a donation transaction is reversed on the blockchain while the DApp's off-chain layer is unaware of the reverse and does not update the donation status, subsequent transactions to withdraw the donation will fail, causing the loss of transaction fees.[2] We refer to such bugs that stem from non-deterministic transaction execution and induce inconsistencies between on-chain and off-chain state as *on-chain-off-chain synchronization bugs*.

While several bug detection techniques have been proposed to assure the quality of DApps, they are either general tools for specifying test cases and assertions [56] or focus only on the on-chain layer by finding defects in smart contracts [22, 25, 32, 38, 51]. In other words, none of them can effectively pinpoint on-chain-off-chain synchronization bugs in DApps. This motivates us to investigate the on-chain-off-chain synchronization bugs and devise testing techniques to detect such bugs.

On-chain-off-chain synchronization bugs could occur in DApps on all blockchain platforms. Due to the impact of the Ethereum blockchain, our work focuses on Ethereum-based DApps. To ease presentation, we may simply refer to Ethereum-based DApps as DApps in the following.

There are two major challenges in detecting on-chain-off-chain synchronization bugs in DApps. First, we need to expose the non-determinism of the transaction executions, which is not considered by existing testing environments, so that on-chain-off-chain synchronization bugs can be better revealed. Second, we need a decidable criterion to mechanically judge the existence of the bugs. To address the first challenge, we study the causes of the synchronization bugs and propose a state transition model for the lifecycle of transactions on the Ethereum blockchain. Guided by the model, we are able to trigger the scenarios where transactions are dropped or reversed to test DApps. To address the second challenge, we propose two test oracles based on the following key observation: the off-chain state of a DApp should stay the same if the corresponding on-chain state is unchanged and a violation of this requirement would indicate the existence of bugs in the state synchronization process. With these oracles, we are able to define assertions on the off-chain states to effectively reveal on-chain-off-chain synchronization bugs in DApps.

We implement our approach as a DApp testing framework, called *ĐArcher*, and evaluate it using 11 popular real-world DApps. These DApps vary in scale, ranging from hundreds to tens of thousands of lines of code, and have received at least 100 stars on GitHub. Our experiment results show that *ĐArcher* is effective. It is able to detect on-chain-off-chain synchronization bugs in all of the DApp subjects. We manually check warnings reported by *ĐArcher*, group the ones with the same root cause, and submit 15 issue reports on GitHub. So far, bugs in six issue reports have been confirmed by developers. These bugs can cause transaction failures, continuous errors being prompted in the DApp client UI, or incorrect information being displayed to users.

This paper makes three major contributions:

- To our best knowledge, this is the first study that examines the bugs in the on-chain-off-chain synchronization process of DApps. We formulate the problem with a state transition model of transaction lifecycle. Using the model, we present the challenges in on-chain-off-chain synchronization and the causes of the synchronization bugs.
- We propose two test oracles and develop a testing framework, *ĐArcher*, to detect on-chain-off-chain synchronization bugs in Ethereum-based DApps. *ĐArcher* is open-source on GitHub[3].
- We evaluate *ĐArcher* on 11 real-world DApps and conclude that *ĐArcher* can detect bugs with high precision (99.3%), recall (87.6%) and accuracy (89.4%). We submit 15 issue reports to developers and six have been confirmed.

2 BACKGROUND

2.1 Two-Layer Architecture of DApps

In Section 1, we have briefly introduced the architecture of DApps. This section further explains several important concepts in detail.

2.1.1 On-chain and off-chain layers. The two-layer architecture of DApps is to balance security, maintainability, performance, and costs [53]. Blockchain (e.g., Ethereum [17]), as a decentralized ledger, offers highly secure data store with programmable smart contracts in the on-chain layer. However, storing data and computations on blockchain incurs a high latency and requires paying a non-negligible amount of transaction fees. Besides, interacting with blockchain with low-level bytecode is also not friendly for ordinary users. The off-chain layer is meant to improve the performance and reduce costs using a user-friendly front-end and centralized services (optional), without sacrificing too much on security [52].

2.1.2 On-chain and off-chain states. As mentioned in Section 1, DApps execute key logics implemented in smart contracts, and store crucial data on the blockchain as the on-chain state. Users usually take actions in the off-chain layer according to the results of some calculations involving on-chain data. Since it is expensive to store such intermediate results back onto the blockchain and it is also inefficient to repetitively query the on-chain state to perform the calculations, DApps usually store a simplified view of their on-chain state and some relevant calculation results at the off-chain layer as off-chain state. Note that the off-chain layer may also store other data irrelevant to the on-chain state. In this paper, we do not include such data in our definition of the off-chain state.

[2] Ethereum users need to pay fees for each transaction, no matter the transaction succeeds or not [17].

[3] https://github.com/Troublor/darcher

2.1.3 On-chain-off-chain synchronization. DApp users send transactions to blockchain via the off-chain layer to make changes to the on-chain state. During the lifecycle of each transaction, blockchain emits various events. DApps can monitor such events to keep their off-chain state synchronized with the corresponding on-chain state. This process is easy to implement if the transactions are executed deterministically on a centralized service or database. However, this is not the case on blockchain, as will be explained in the next subsection.

2.2 Non-deterministic Transaction Execution

Transactions sent to blockchain will be broadcast to miners throughout the network. Conceptually, miners on a blockchain collectively maintain a pool of transactions awaiting execution. A transaction is added to the transaction pool when it is received by a miner. A transaction is executed non-deterministically on the blockchain from two aspects. First, a transaction may not be executed after it has been sent by DApp users. Second, an executed transaction can be reversed from the transaction history. In the following, we explain how such non-determinism arises.

2.2.1 Sent transactions may not be executed. A miner can select which transactions to execute from the pool, when a new block has been mined. It is possible that a transaction is not selected and keeps staying in the pool. An old transaction in the pool can be invalidated by a new one with the same nonce, which is the index of a transaction sent from the user [10, 17]. The invalidated transactions will be dropped and never get executed on the blockchain. In addition, miners can silently drop transactions for various reasons (e.g., due to the size limit of the transaction pool).

2.2.2 Executed transactions may be reversed. A newly mined block is not necessarily added to the blockchain. When multiple miners concurrently mine a new block [59], the blockchain will fork multiple chains of blocks. To resolve this problem, the blockchain will validate only the longest chain and invalidate the others. The process is known as *chain reorganization* [37]. If the execution of a transaction is recorded by a block in an invalidated chain, the transaction will be reversed and put back to the pool. In practice, chain reorganization can hardly affect blocks with a sufficient number of *confirmations*, which are the succeeding blocks on the same chain [18].

Such non-determinism in transaction execution complicates the interactions and synchronization between the on-chain and off-chain layers of a DApp. Bugs can arise if the DApp handles the non-determinism inappropriately. In Section 3, we will give a real-world example of such bugs.

3 MOTIVATING EXAMPLE

In this section, we present a code snippet adapted from Giveth[20], to illustrate the on-chain-off-chain synchronization in DApps, and an on-chain-off-chain synchronization bug [21].

Listing 1 shows a code snippet for the "withdraw donation" functionality in the front-end client of Giveth, which allows users to withdraw the donations they receive. The front-end client of Giveth uses web3.js [47], an official Ethereum JavaScript library, to send a transaction to an underlying smart contract "liquidPledging" and call its "withdraw" function at line 3. The client registers a callback for handling the "transactionHash" event, which occurs when the transaction is added to the blockchain transaction pool (line 4). Since the transaction is to modify the on-chain state concerning the donation amount, the off-chain state is also updated accordingly (line 6). After the update, a corresponding withdraw record is created and stored in an off-chain centralized database (line 11).

The code in Listing 1 may not work correctly because the "withdraw donation" transaction may remain in the transaction pool indefinitely or be dropped without execution. The occurrence of the "transactionHash" event only signifies that a transaction has been added to the transaction pool, rather than the execution of the transaction. Even if the transaction has been executed, it can still be reversed later. Therefore, it is possible that the off-chain state is updated while the on-chain state remains unchanged (i.e., when the withdraw transaction is dropped or reversed). Such state inconsistencies can lead to many undesirable consequences. For example, let us consider the following scenario. A user of Giveth sends a "withdraw donation" transaction to the smart contract but the transaction is dropped by blockchain. However, due to the bug, the Giveth client incorrectly shows that the donation has been withdrawn and the user decides to donate the withdrawn cryptocurrencies to another community. In such a scenario, the new donation transaction may fail, causing unnecessary loss of transaction fee and poor user experience.

It is difficult to expose on-chain-off-chain synchronization bugs using existing DApp testing tools. For example, Ganache [8] is one of the most commonly used blockchain environments for testing Ethereum-based DApps. Like testing on centralized databases, transactions sent to Ganache blockchain are executed immediately. Therefore, when testing Giveth using Ganache, the on-chain and off-chain states are always consistent. The aforementioned bug can never be detected.

To tackle this problem, we propose to model the non-determinism in transaction execution using transaction lifecycle. Our approach simulates the non-deterministic transaction execution process and drives DApps to systematically traverse the lifecycle of each transaction. We also propose effective test oracles to automatically detect inconsistencies between the on-chain and off-chain states.

4 METHODOLOGY

In this section, we first propose a state transition model to capture the lifecycle of transactions on Ethereum blockchain. We then discuss the challenges in on-chain-off-chain synchronization, and identify two types of synchronization bugs that may arise in DApps. After that, we propose our framework, *ĐArcher*, to detect on-chain-off-chain synchronization bugs in DApps.

4.1 Transaction Lifecycle

To detect on-chain-off-chain synchronization bugs, we model the non-determinism in the lifecycle of a transaction on Ethereum blockchain using a state transition model as shown in Fig. 2. A transaction starts its lifecycle at the *Created* state by constructing the required arguments at the DApp's off-chain layer. After that, the transaction is sent to blockchain and transits to the *Pending* state, meaning that the transaction is added to the transaction pool

```
function withdrawDonation() { // withdraw donation from one crowdfunding project
    contract.liquidPledging
        .withdraw(...arguments) // send withdraw transaction to smart contract
        .once('transactionHash', hash => { // function called when transaction is sent
            let txHash = hash; //
            updateExistingDonation(existingDonation, amount); // update existing donation after withdraw

            const withdrawRecord = {...}; // create a new withdraw record
            feathersClient
                .service('/donations')
                .create(withdrawRecord) // save the withdraw record in the centralized database
                .catch(onError);
        });
}
```

Listing 1: Withdraw Donation Functionality in the DApp Giveth

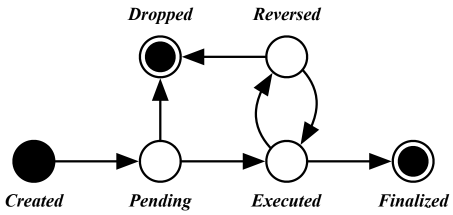

Figure 2: State Transition Model of the Ethereum Transaction Lifecycle

awaiting execution on blockchain. A transaction may remain in the *Pending* state indefinitely as miners are free to select more profitable transactions for execution.

A *Pending* transaction can be dropped and transit to the *Dropped* state in two cases. First, users may send a duplicated transaction to override the previous transaction or offer a higher fee to increase the chance of execution. Second, the transaction may be deleted silently by miners, due to the capacity limit of the transaction pool, or malicious behaviours of miners. In the latter case, the DApp is not informed that a transaction has been dropped, and it is also hard for the DApp to proactively check whether a transaction is dropped on the blockchain or not.

A *Pending* transaction transits to the *Executed* state when it is executed and included in a block. As discussed in Section 2, an executed transaction can be reversed when the blockchain is reorganized. When it happens, the transaction will transit from the *Executed* state to the *Reversed* state and be put back to the transaction pool, awaiting execution. Similar to the *Pending* state, transactions at the *Reversed* state can also be dropped.

While in theory executed transactions can be reversed, in practice, a transaction whose execution has been logged by a block with a sufficient number of confirmations can be considered finalized, i.e., the transaction will transit from the *Executed* state to the *Finalized* state. The number of confirmations required varies according to the security requirements of the DApp [43]. To avoid problems caused by reversed transactions, one common practice adopted by DApp developers is to use the result of a transaction at the off-chain layer only after the transaction transits to *Finalized* state [18].

To estimate the frequency of transaction state transitions, we collect Ethereum traffic data from Etherscan [13] and a Ethereum full node maintained by us. We find that the average number of transactions submitted to Ethereum per second is 32.5, while the average number of transactions executed by Ethereum per second (TPS) is 16.4, which indicates that around half of the transactions cannot be immediately executed after submission. We also monitor the Ethereum mainnet for over 4 months and observe that chain reorganization happens every 24.43 blocks, i.e., every 11.06 minutes. The average number of transactions reversed per hour due to such chain reorganizations is 16.69. These statistics show that the drop and reverse rates of transactions on Ethereum are non-negligible, and it is necessary for DApps to consider the non-deterministic transaction execution to avoid on-chain-off-chain synchronization bugs.

4.2 On-Chain-Off-Chain Synchronization Bugs

In Section 3, we have presented an on-chain-off-chain synchronization bug in a DApp named Giveth. On-chain-off-chain synchronization bugs occur when DApps fail to maintain the consistency between their on-chain states and off-chain states. In the Giveth bug, developers do not consider the situation, in which a *Pending* transaction is *Dropped* by the blockchain, and inconsistencies are thus induced. Such inconsistencies can result in erroneous off-chain states, which might mislead users. Further operations based on these erroneous states can cause unexpected changes on the blockchain or waste transaction fees.

Since DApps are a new kind of software applications, the lack of understanding in state transitions in transaction lifecycle may make it difficult for developers to assure proper on-chain-off-chain synchronization. We observe that developers often assume that the transactions submitted by their DApp would be *Executed* and eventually *Finalized* on blockchain, while ignoring the situations where (1) a *Pending* or *Reversed* transaction is dropped silently by the blockchain, or (2) an *Executed* transaction is reversed due to chain reorganization. As a result, on-chain-off-chain-synchronization bugs often arise. In this paper, we focus on studying on-chain-off-chain synchronization bugs triggered in the two situations. We refer to them as Type-I and Type-II bugs, respectively. Fig. 3 illustrates how these two types of bugs could occur with our transaction lifecycle model. We will introduce them in detail below.

4.2.1 Type-I Bugs. As Fig. 3(a) shows, Type-I bugs occur because the off-chain state is prematurely updated when the transaction is in *Pending* state, but in fact, the transaction is later dropped.

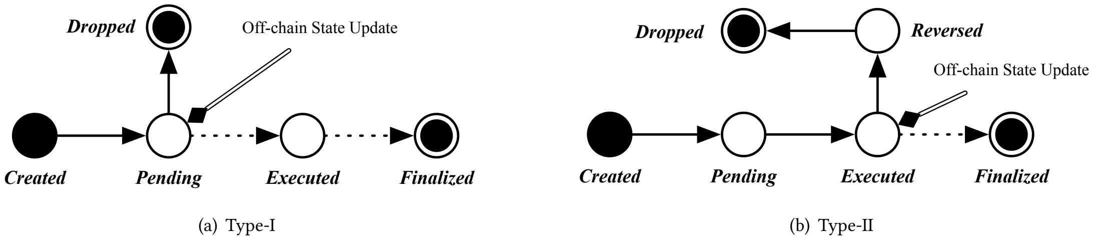

Figure 3: Transaction Lifecycles for the Two Types of On-Chain-Off-Chain Synchronization Bugs. Solid arrows depict the actual state transitions, and dashed arrows depict the transitions assumed by DApps. For Type-I bugs, the off-chain state is prematurely updated when the transaction is *Pending*. For Type-II bugs, the off-chain state is updated when the transaction is *Executed*, but the updated state is not reverted when the transaction is reversed.

The bug in Giveth as shown in Listing 1 is a Type-I bug. Giveth immediately updates its off-chain state (lines 4–13) when the donation withdrawal transaction is submitted to blockchain. It does not further check the state of the transaction and properly deal with potential state changes.

4.2.2 Type-II Bugs. As Fig. 3(b) shows, Type-II bugs occur because the off-chain state is updated after the transaction transits to the *Executed* state, but the executed transaction is later reversed due to chain reorganization. Issue #8260 of Augur [31], a popular DApp for prediction markets [30], is a Type-II bug. Augur updates its off-chain state when the transactions that create markets are executed but does not revert the off-chain state when the executed transactions are reversed. Consequently, the front-end client of Augur would display markets that do not exist on blockchain. Any further operations on these markets will result in “Market Not Found” errors.

These two types of bugs are not well studied in the literature. Existing techniques either focused on testing smart contracts [22, 25, 32, 38, 51], which are only concerned with the on-chain layer of DApps, or do not fully consider the entire lifecycle of transactions when testing DApps [8]. Take a popular DApp testing environment Ganache [8] as an example. When using Ganache, DApp developers are encouraged to configure the testing blockchain to make transactions immediately executed [23], while the possible state transitions from *Pending* to *Dropped* or from *Executed* to *Reversed* are ignored [24]. As a result, Type-I and Type-II bugs can easily slip into real-world DApps. This motivates us to propose *ÐArcher*, a testing framework to effectively detect the two types of on-chain-off-chain synchronization bugs in DApps.

4.3 The *ÐArcher* Testing Framework

From the above discussion, we can see that to trigger on-chain-off-chain synchronization bugs, we need to systematically emulate transaction state transitions. Guided by our transaction lifecycle model, *ÐArcher* controls the execution of each transaction in its testing environment to drive the transaction to traverse possible states. However, triggering bugs alone is not sufficient. Effective testing also requires oracles to judge the existence of bugs. In the following, we present our design of test oracles in *ÐArcher*, and then explain how to leverage the oracles to detect on-chain-off-chain synchronization bugs during testing.

4.3.1 Test Oracles. It is challenging to design oracles for detecting on-chain-off-chain synchronization bugs. Although the on-chain and off-chain states in DApps need to be consistent, they are not necessarily identical. Off-chain states are often maintained as a simplified version of the corresponding on-chain states to facilitate front-end user actions. For instance, in our motivating example, Giveth’s cache server processes on-chain donations and stores processed data, instead of making an exact copy of the on-chain state. The processed data contains additional information, such as the index of donations, to facilitate user queries, but such information is not saved on blockchain to reduce the cost of interacting with smart contracts. In addition, updates of the off-chain state depend on not only the changes of the on-chain state, but also the program logic of the off-chain layer. Therefore, it is hard to specify the mapping between the on-chain and off-chain states as well as directly check whether the off-chain state is consistent with the on-chain state. To address the challenge, we design oracles that check on-chain-off-chain state consistency by only comparing the off-chain states of the DApp when the concerned transaction is at different lifecycle stages. In this way, the mapping between on-chain and off-chain states is not required. In the following, we present the test oracles.

Test oracle for Type-I bugs. For Type-I bugs, we observe that, when a transaction t is added to the transaction pool (i.e., transits from the *Created* state to the *Pending* state), the DApp should not prematurely update the off-chain state as if t is executed and finalized. Since DApps are not informed when transactions transit from the *Pending* state to the *Dropped* state, any premature updates of the off-chain state are likely to remain when the transactions are dropped on the blockchain. Assertion 1 helps detect such bugs.

Assertion 1. *For each transaction* t*,* $\sigma(t, Created) \neq \sigma(t, Finalized)$ *implies* $\sigma(t, Pending) \neq \sigma(t, Finalized)$.

In the above formulation, $\sigma(t, s)$ denotes the off-chain state of the DApp under test when the transaction t is at the transaction lifecycle state s. The clause $\sigma(t, Created) \neq \sigma(t, Finalized)$ means that the transaction t results in an update to the off-chain state. The clause $\sigma(t, Pending) \neq \sigma(t, Finalized)$ specifies that the off-chain state should not be updated as if the transaction has been *Finalized*, when the transaction has been just sent to the transaction pool. Note that this assertion allows DApps to make changes to the off-chain state when the concerned transaction is *Pending*, but the changes should not indicate that the transaction has been *Finalized*. Violations of Assertion 1 indicate the existence of Type-I bugs.

Test oracle for Type-II bugs. When a transaction is executed (i.e., transits from the *Pending* state to the *Executed* state), some DApps may update their off-chain states. Type-II bugs occur when the executed transaction is reversed due to chain reorganization but the updated off-chain state is not reverted accordingly. *ĐArcher* leverages Assertion 2 to check for Type-II bugs.

Assertion 2. *For each transaction t,* $\sigma(t, Pending) = \sigma(t, Reversed)$.

Since reversed transactions are put back to the transaction pool, the off-chain state of the DApp when transaction t is at the *Pending* state should be the same as that when t is at the *Reversed* state. Violations of Assertion 2 indicate the existence of Type-II bugs.

4.3.2 Lifecycle Emulation & Assertion Checking. As discussed earlier, in existing blockchain testing environments such as Ganache [8], transactions are directly *Executed* and *Finalized* once submitted to blockchain. These transactions are never dropped or reversed, and thus Type-I and Type-II bugs cannot be triggered. To address the limitation, *ĐArcher* implements a blockchain environment that is able to control the state of transactions. Instead of executing a transaction immediately after it is submitted to the blockchain, *ĐArcher* drives the transaction to traverse lifecycle states in the following order: *Created* → *Pending* → *Executed* → *Reversed* → *Executed* → *Finalized*. Such a traversal allows our proposed oracles to be evaluated for each transaction to detect the two types of on-chain-off-chain synchronization bugs. Specifically, when there is a state transition, *ĐArcher* fetches and stores the off-chain state of the DApp under testing. After the state traversal terminates, the fetched off-chain state is checked against the Assertions 1 and 2. If there is any assertion violation, *ĐArcher* will report a bug. More details of bug detection will be further introduced in Section 5.

5 IMPLEMENTATION

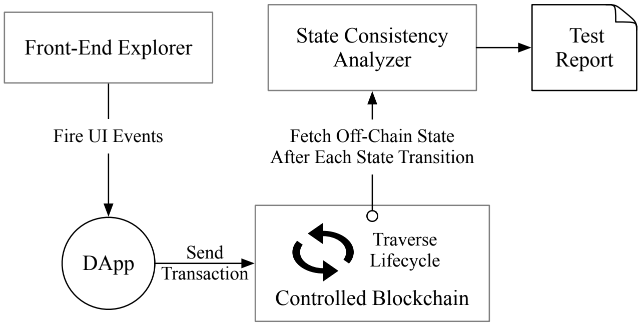

Figure 4: Overview and Workflow of *ĐArcher*

Fig. 4 shows an overview of *ĐArcher*. Given a DApp, the front-end explorer fires UI events to exercise the DApp to explore the functionalities that involve sending transactions. Once a transaction is sent, *ĐArcher* leverages a controlled blockchain to execute it and traverse its lifecycle states in the order mentioned in Section 4.3.2. During the state traversal process, *ĐArcher* keeps fetching the off-chain state of the DApp, whenever there is a state change. The state consistency analyzer then checks such collected off-chain states to detect bugs.

We have open sourced *ĐArcher* on GitHub[4]. In the following, we will present more details on how *ĐArcher* is implemented.

5.1 The Front-End Explorer

As mentioned earlier, the front-end explorer fires UI events to exercise DApps. In practice, developers can use any applicable tool or manually write test cases for the purpose. In the current implementation of *ĐArcher*, we choose to integrate a popular web testing tool, Crawljax [33], to generate GUI events to test DApps, which are often web-based applications. Crawljax infers a state-flow graph when testing a web application. GUI events are fired at those states that can transit to unvisited states. The state-flow graph is updated whenever new states are discovered during testing. The exploration will stop when all of the states in the graph have been visited. Such model-based testing can help exercise DApps to interact with the blockchain.

5.2 The Controlled Blockchain

We implement a controlled blockchain in *ĐArcher* based on Geth [12], a popular Ethereum client. When the DApp under test submits a transaction to our controlled blockchain, *ĐArcher* will drive the transaction to traverse its lifecycle states according to the order mentioned in Section 4.3.2.

5.3 The State Consistency Analyzer

As mentioned in the overview, during testing, *ĐArcher* keeps collecting the off-chain state data for bug detection. As different DApps may maintain the off-chain states in different ways (e.g., using various databases such as MySQL and MongoDB, or browser storage such as IndexedDB and localStorage), it is hard to design a tool to automatically identify runtime values that represent the off-chain states for all DApps. As a workaround, we build our off-chain state fetcher for different data storage, including databases, browser storage, and HTML elements. Users of our tool only need to configure a few rules to specify the variables and fields that constitute the off-chain state of their DApp.

To minimize the runtime overhead, *ĐArcher* does not instrument DApps. Since it is hard to determine when a DApp finishes updating its off-chain state at runtime, *ĐArcher* would wait for a period of time when a transaction's lifecycle state changes, before fetching the off-chain state. The waiting time is configurable in *ĐArcher*.

After the state traversal process completes, *ĐArcher* checks the fetched off-chain states to identify the two types of on-chain-off-chain synchronization bugs according to the two oracles proposed in Section 4.3.1.

6 EVALUATION

We evaluate *ĐArcher* on real-world web-based DApps. Specifically, we investigate the following research questions:

- **RQ1 (Bug Detection Capability)**: Can *ĐArcher* effectively detect on-chain-off-chain synchronization bugs in real DApps?

[4] https://github.com/Troublor/darcher

- **RQ2 (Efficacy of Our Oracles)**: How effective are our proposed oracles? Can on-chain-off-chain synchronization bugs be detected using existing oracles?
- **RQ3 (Usefulness)**: Can *ĐArcher* detect on-chain-off-chain synchronization bugs that are useful to developers?

6.1 Subjects

To collect subjects for evaluation, we search for Ethereum DApps on GitHub using the keyword *ethereum* with constraints *stars:>=100, language:JavaScript* or *language:TypeScript*. The first constraint is to ensure the popularity of the selected subjects. The second constraint is to help find web-based DApps. The search returns 254 projects. We manually check each project to exclude those that are not web applications, or are libraries, blockchain clients, block explorers, transaction trackers, frameworks, operating systems, etc. Projects that are tagged as deprecated or archived are also excluded because they are not actively maintained. Furthermore, we exclude those that are claimed to be examples, tutorials, or starters, as we are interested in real DApps rather than toy examples. After this filtering process, 21 DApps remain. Finally, since *ĐArcher* is a dynamic testing framework, which requires executing transactions, we exclude those projects that we fail to deploy on our local blockchain by following the provided instructions. After the above filtering, we collect a total of 11 popular real-world DApps for experiments. This is in line with an empirical study made by Wu et al. [55] that few DApps are fully open-source. Table 1 provides the information of these subjects. The selected subjects differ in scales. The smallest DApp has less than 1,000 lines of code, while the largest DApp has more than 30,000 lines of code. The purposes of the subjects are also quite diversified.

6.2 Experiment Design

In this subsection, we explain how we set up our experiments, derive off-chain states, and establish the ground truth to validate bug detection results. We also introduce the baselines, against which *ĐArcher* will be compared.

Experiment Setup. We deploy the subjects locally on our controlled blockchain. We set the *ĐArcher*'s waiting time for DApps to update their off-chain states to be 15 seconds, which is aligned with the average block interval time on Ethereum mainnet [13]. We run *ĐArcher* on each DApp for one hour. We observe that this is sufficient for *ĐArcher* to reach the saturation of test coverage during experiments. We repeat the testing process ten times for each DApp to mitigate the randomness of Cralwjax.

Off-Chain State Derivation. One challenge in the experiments is to identify the data fields that compose off-chain states. As we have discussed in Section 5.3, different DApps may maintain off-chain states in different ways. As a workaround, we manually explore each DApp, and mechanically derive the off-chain state by including those data fields that are updated when transactions are directly executed (i.e., going through the states *Created* → *Pending* → *Executed* → *Finalized*). The intuition behind is that we assume developers have tested their DApp to assure that the off-chain state is properly updated when transactions undergo such "normal" executions.

Table 1: The DApps Used in Our Experiments

DApp	Stars	LOC (JS)	Commits	Purpose
AgroChain [1]	106	291	40	Agricultural Supply Chain
Augur [30]	264	68,972	40,971	Prediction Markets
DemocracyEarth [16]	1,305	29,149	3,394	Governance for DAOs
ETH Hot Wallet [39]	197	7,295	284	Wallet
Ethereum Voting Dapp [36]	348	238	31	Voting
Giveth [20]	257	30,708	3,131	Charitable Donation
Heiswap [48]	100	1,687	87	Anonymous Transfer
MetaMask [15]	4,091	81,898	11,072	Wallet
Multisender [44]	165	1,421	33	One-to-Many Transfer
PublicVotes [41]	149	628	34	Voting
TodoList Dapp [2]	127	1,076	28	Todo List

Result Validation. To evaluate the precision and recall of *ĐArcher* for each DApp, we manually reproduce all transactions generated during testing and make them go though the lifecycle to identify potential inconsistencies between the on-chain and off-chain states. This process is independently performed by two authors. The results are cross-checked for consistency. It is worth mentioning that the recall metric we used here is for evaluating *ĐArcher*'s capability of catching Type-I and Type-II bugs once they arise during the processing of transactions. We do not aim to evaluate how many of all possible on-chain-off-chain synchronization bugs can be detected by *ĐArcher* since it is hard to obtain the ground truth and the Crawljax front-end explorer may not be able to trigger all possible transactions during testing.

Baselines. Since there is no prior work for detecting on-chain-off-chain synchronization bugs in DApps, we construct two baseline methods by replacing oracles used by *ĐArcher* with the ones used for detecting smart contract vulnerabilities [25] and web application faults [4]. Specifically, Baseline-I would report bugs if the execution of a transaction violates the assertions defined by ContractFuzzer [25]. Baseline-II would report bugs if runtime errors occur in the JavaScript console during the testing process [4]. Except for the differences in oracles, the two baseline methods use exactly the same tests for each DApp and traverse the lifecycle of each transaction in the same way as *ĐArcher*. We do not compare *ĐArcher* with existing blockchain testing environments such as Ganache [8], since none of them is able to emulate the lifecycle of transactions, and thus neither Type-I nor Type-II bugs can be triggered by them.

6.3 RQ1: Bug Detection Capability

Table 2 presents the results of *ĐArcher* and baselines, including the overall test coverage, precision, recall, and accuracy. Since *ĐArcher* aims to test the correctness of synchronization between the on-chain and off-chain layers in DApps during the transaction execution process, we measure the test coverage in terms of how well transaction submission API call sites are covered during testing[5]. Such API call sites are essentially the "starting point" of on-chain-off-chain synchronization. Their coverage could indicate the diversity of the tests generated by the Front-End Explorer. Note that since some functionalities relying on Ethereum public blockchain services

[5] We are able to measure the coverage of transaction submission API call sites because DApps use designated APIs [6, 45, 46] provided by the official libraries (e.g., web3.js [47]) of Ethereum to interact with the blockchain.

(e.g., Uniswap [50]) are unavailable on the controlled blockchain of *ÐArcher*, API call sites for such functionalities are not considered in our coverage measurement. In the following, we will discuss the experiment results in detail.

Coverage. *ÐArcher* triggers a total of 3,134 transactions and covers 70% of the transaction submission API call sites when testing the 11 DApps. It achieves an overall accuracy of 89.4% in deciding whether a transaction handling process contains bugs or not.

False Positives. Altogether, there are 385 transactions violating Assertion 1 and 1,862 transactions violating Assertion 2. There is no transaction violating both assertions. Among these transactions, there are 16 exhibiting Type-I bugs, which are found to be false positives (FPs). There are no false positives for those detected to exhibit Type-II bugs. We find that all of the 16 FPs are related to the DApp Giveth. They arise from the unexpected delays in updating the off-chain states in Giveth's cache server. For instance, when a transaction reaches the *Pending* state, Giveth makes a partial update to the off-chain state, with a flag indicating that the transaction is awaiting execution. The flag should be cleared when the transaction is *Finalized*. However, it is not cleared within 15 seconds, so that *ÐArcher* fetches the same off-chain state as the one fetched when the transaction is *Pending*. In this case, *ÐArcher* reports a violation of Assertion 1 but in fact it is a false positive. If *ÐArcher* waits for a longer period before fetching the off-chain state, the violation will not be reported. Despite the FPs, *ÐArcher* still achieves an overall precision of 99.3% in detecting on-chain-off-chain synchronization bugs in the collected DApps.

False Negatives. There are 52 and 264 transactions exhibiting Type-I and Type-II bugs, respectively, which are missed by *ÐArcher*. After investigating the corresponding transactions, we find two major reasons for the false negatives (FNs). First, *ÐArcher* may fetch the off-chain state before it is inappropriately updated, in the same way as we have mentioned in the previous Giveth example. Second, transactions may depend on each other. An on-chain-off-chain synchronization bug may occur if a transaction v is executed based on the interim result of another transaction u, while u is reversed and dropped. FN occurs if the DApp does not update the off-chain state for transaction v (neither Assertion 1 or 2 will be violated if the off-chain state is unchanged). Despite such cases, the recall of *ÐArcher* is still quite high and reaches 87.6% in our experiments.

Answer to RQ1: *ÐArcher* can effectively detect on-chain-off-chain synchronization bugs in DApps with high precision (99.3%), recall (87.6%), and accuracy (89.4%).

6.4 RQ2: Efficacy of Our Oracles

We answer RQ2 by comparing *ÐArcher* with the two baseline methods that employ existing oracles. The results of baselines are also presented in Table 2.

As we can see from the table, Baseline-I only generates warnings for seven transactions. After inspecting the seven transactions whose execution violates vulnerability assertions, we find that the underlying smart contract contains the exception disorder vulnerability according to the bug definition by Jiang et al. [25]. However, these 7 warnings are all FPs in terms of on-chain-off-chain synchronization bugs. Smart contract vulnerability oracles cannot detect any on-chain-off-chain synchronization bugs in the experiment. It is because the detection of on-chain-off-chain synchronization bugs requires the examination of both on-chain and off-chain layers of a DApp, whereas the oracles for smart contract vulnerabilities examine only the on-chain layer.

Baseline-II is able to reveal some on-chain-off-chain synchronization bugs. For example, a bug [31] in Augur results in a runtime error with message "*Uncaught (in promise) Error: execution reverted*", when a transaction is reversed. However, runtime error oracle is not effective, compared to our proposed oracles. Our experiments show that the overall precision, recall, and accuracy of Baseline-II are only 56.9%, 10.1% and 20.7%, respectively, which are significantly worse than *ÐArcher*. In addition, the runtime error messages generated may not provide useful information about the root causes of on-chain-off-chain synchronization bugs. For instance, the message "Error: PollingBlockTracker - encountered error fetching block:" generated in the testing of MetaMask gives little hint to the occurrence of a synchronization bug.

Answer to RQ2: Our proposed oracles significantly outperform the existing ones, in terms of detecting on-chain-off-chain synchronization bugs.

6.5 RQ3: Usefulness

We answer RQ3 by reporting bugs detected by *ÐArcher* to the developers and communicating with them. Although *ÐArcher* reports warnings for thousands of transactions in the experiments, lots of them are repeated explorations of the same functionalities in DApps. To avoid overwhelming developers, we group the warnings with the same root cause into a single issue to report to the developers. Table. 3 lists the IDs of the GitHub issues, in which we report on-chain-off-chain synchronization bugs to the DApp developers.

In total, we have reported 15 bugs, of which six have been confirmed by developers and three have been fixed. Developers provide positive feedbacks to our reported bugs. For example, developers of Giveth respond "Syncing two backend (cache server and blockchain) is delicate and complex job, and we hope is solved soon", and the bugs in Giveth were fixed one and a half month after we reported them. The comment indicates that the reported on-chain-off-chain synchronization bugs are real and the on-chain/off-chain synchronization is complex. Detection of the bugs is useful to developers in improving the quality of DApps.

Due to the complexity of on-chain/off-chain synchronization, developers are likely to improperly handle scenarios where transactions are dropped or reversed. For instance, the developers of Giveth are aware of the possibility that transactions can be reversed after execution and Giveth is designed to only update the off-chain state when transactions reach *Finalized* state. However, as discussed in Section 3, the handling of some transactions still contains faults, and the off-chain state is updated prematurely without waiting for the transaction to be finalized. Another example is Augur, which maintains a rollback table that stores the metadata used to revert the off-chain state when a transaction is reversed

Table 2: Experiment Results of *ĐArcher* and Baselines

DApp	API Call Site Coverage	Total Txs.	*ĐArcher*									Baseline-I						Baseline-II					
			TP		FP		FN		Pre.	Rec.	Acc.	Contract Vulnerability Oracle						Runtime Error Oracle					
			I	II	I	II	I	II				TP	FP	FN	Pre.	Rec.	Acc.	TP	FP	FN	Pre.	Rec.	Acc.
AgroChain	75.0%	417	60	160	0	0	40	0	100.0%	84.6%	90.4%	0	0	260	-	0.0%	37.6%	0	0	260	-	0.0%	37.6%
Augur	66.7%	164	0	24	0	0	0	25	100.0%	49.0%	84.8%	0	7	49	0.0%	0.0%	65.9%	49	115	0	29.9%	100.0%	29.9%
DemocracyEarth	100.0%	78	0	78	0	0	0	0	100.0%	100.0%	100.0%	0	0	78	-	0.0%	0.0%	0	0	78	-	0.0%	0.0%
ETH Hot Wallet	100.0%	140	140	0	0	0	0	0	100.0%	100.0%	100.0%	0	0	140	-	0.0%	0.0%	58	0	82	100.0%	41.4%	41.4%
Ethereum Voting Dapp	100.0%	376	0	140	0	0	0	236	100.0%	37.2%	37.2%	0	0	376	-	0.0%	0.0%	0	0	376	-	0.0%	0.0%
Giveth	63.6%	353	27	0	16	0	11	0	62.8%	71.1%	92.4%	0	0	38	-	0.0%	89.2%	0	80	38	0.0%	0.0%	66.6%
Heiswap	100.0%	323	0	322	0	0	0	1	100.0%	99.7%	99.7%	0	0	323	-	0.0%	0.0%	16	0	307	100.0%	5.0%	5.0%
MetaMask	100.0%	225	0	225	0	0	0	0	100.0%	100.0%	100.0%	0	0	225	-	0.0%	0.0%	1	0	224	100.0%	0.4%	0.4%
Multisender	100.0%	334	0	332	0	0	0	2	100.0%	99.4%	99.4%	0	0	334	-	0.0%	0.0%	0	0	334	-	0.0%	0.0%
PublicVotes	100.0%	289	142	146	0	0	1	0	100.0%	99.7%	99.7%	0	0	289	-	0.0%	0.0%	133	0	156	100.0%	46.0%	46.0%
TodoList Dapp	100.0%	435	0	435	0	0	0	0	100.0%	100.0%	100.0%	0	0	435	-	0.0%	0.0%	0	0	435	-	0.0%	0.0%
Total/Overall	70.0%	3,134	369	1,862	16	0	52	264	99.3%	87.6%	89.4%	0	7	2,547	0.0%	0.0%	18.5%	257	195	2,290	56.9%	10.1%	20.7%
Average	77.3%	284.9	33.5	169.3	1.5	0	4.7	24.0	96.6%	85.5%	91.2%	0.0	0.6	231.5	0.0%	0.0%	17.5%	23.4	17.7	208.2	71.7%	17.5%	20.6%

Txs. is short for transactions. We report the number of TPs, FPs and FNs for Type-I and Type-II bugs separately in the detection results of *ĐArcher*. We use Pre., Rec., and Acc. as abbreviations for precision, recall, and accuracy, respectively. Precision is marked as "-" when the tool reports no bugs. Average results are arithmetic means of the results for all subjects.

Table 3: Real Bugs Detected by *ĐArcher*

DApp	Issues	DApp	Issues
AgroChain	#7, #8	Heiswap	#29
Augur	#8260[f]	MetaMask	#10120[c]
DemocracyEarth	#561	Multisender	#34[c]
ETH-Hot-Wallet	#41	PublicVotes	#11, #12
Ethereum Voting Dapp	#28	TodoList Dapp	#5[?]
Giveth	#1103[f], #1605[f], #1792[c]		

[f] The reported bugs have been confirmed and fixed.
[c] The reported bugs have been confirmed.
[?] The developers have asked for Pull Requests.
Other issues have not received responses from developers.

on blockchain. However, the rollback table is cleared unexpectedly when the page is refreshed after the transaction is executed. In such cases, the off-chain state does not get reverted when a transaction is reversed, causing inconsistencies between off-chain and on-chain states. *ĐArcher* is able to catch the above on-chain-off-chain synchronization bugs, and they have been confirmed and fixed by developers. This demonstrates that on-chain-off-chain synchronization bugs could still occur in those DApps whose developers have already considered the non-determinism of transaction execution, and *ĐArcher* is able to help developers catch the hidden bugs.

Answer to RQ3: Among 15 bugs reported to developers, six have been confirmed and three have been fixed. Responses from developers show that *ĐArcher* is useful in detecting on-chain-off-chain synchronization bugs.

7 DISCUSSIONS

7.1 Synchronization Strategies

To better understand how developers of DApps synchronize on-chain and off-chain states, we further investigate the synchronization strategies adopted in the 11 DApps used in our evaluation. We observe three common strategies.

Periodic Polling. The most straightforward way to synchronize on-chain and off-chain states is to poll the on-chain state periodically. For instance, DemocracyEarth [16] registers a daemon task that periodically checks the on-chain state and updates the off-chain state accordingly. This strategy is effective to keep the off-chain state consistent with the on-chain state, during the lifecycle of transactions. Nevertheless, periodic polling is inefficient if the DApp is complicated. Redundant synchronization work is performed repeatedly, and a lot of communication and computation overhead would be induced.

Passive Waiting. The passive waiting strategy copes with the non-determinism of transaction execution by updating the off-chain state only when a transaction reaches *Finalized* state in its lifecycle. That is to say, as long as the non-determinism still exists, i.e., transactions are not yet finalized, the DApp does not update its off-chain state. For instance, Giveth [20] adopts this strategy by counting the number of confirmations for each transaction after it is executed. Only when a transaction has enough confirmations will Giveth update its off-chain state in the centralized cache server. This strategy could save a lot of communication and computation overhead compared to the periodic polling strategy, since the DApp only needs to track the transactions that are not yet finalized. However, it could induce inevitable delay in the DApp, influencing user experiences, because after the transaction is executed, users must wait until there are enough confirmations.

Aggressive Updating. The aggressive updating strategy is another choice for DApp developers. This strategy is intended to keep the off-chain state closely synchronous with the on-chain state, which means the DApp updates its off-chain state when transactions are executed and reverts the off-chain state when transactions are reversed. For instance, Augur adopts this strategy in its implementation. When a transaction is sent to blockchain, i.e., at *Pending* state, the off-chain state is not updated. The update only takes place when the transaction is executed. If the transaction is reversed due to blockchain reorganization, Augur will also revert its off-chain state accordingly. This strategy offers better user experience than the passive waiting strategy in that users can see the updates of the off-chain state immediately when their transactions are executed or reversed. However, it is more error-prone to revert the off-chain state, which might involve many data fields, when transactions are reversed on the blockchain.

7.2 Limitations and Future Work

Our work is subject to two limitations. First, precise identification of the off-chain states is important to the test effectiveness of *ĐArcher*. In our experiments, we assume that the off-chain states are appropriately updated when the transactions are completed in a straightforward manner. This may not always hold. Second, *ĐArcher* assumes that each update of an off-chain state, if ever happens, would be completed within a fixed time period, which is to be manually specified. If the period is set to be too large, the time efficiency of *ĐArcher* is compromised. If the period is set to be too small, *ĐArcher* may miss the detection of some on-chain-off-chain synchronization bugs. Furthermore, the period can vary across DApps. A possible solution to these two limitations is to analyze the source code of DApps to determine what compose the off-chain states and when they are updated, so that the efficiency and effectiveness of each transaction's analysis could be improved. However, the dynamic and reflective nature of JavaScript [42] imposes other challenges to perform a sound and complete analysis on DApps. As such, we leave these two limitations to be addressed in our future work.

Furthermore, as discussed in Section 6.3, it could be the case that the consequence of on-chain-off-chain synchronization bugs is not reflected in DApps' off-chain states, e.g., sending a transaction dependent on the interim result of another transaction, which gets reversed or dropped. Future work can be made to detect the on-chain-off-chain synchronization bugs exhibited in such scenarios.

7.3 Threats to Validity

The limited number of DApp subjects poses an external threat to the validity of our evaluation results. We mitigate this threat by selecting popular real-world DApps of different sizes and purposes from GitHub to improve their representativeness. More subjects are needed to fully address this threat in future research.

Threats to internal validity may arise from the way we interpret the experiment results. The specification of off-chain states for each DApp poses a threat, which we mitigate by mechanically deriving off-chain states with the assumption discussed in Section 6.2. During the reproduction of transactions, we confirm that our assumption holds for all DApp subjects. We also report our detected bugs to the DApp developers, and ask for their feedbacks to confirm the effectiveness and usefulness of *ĐArcher*.

8 RELATED WORK

In this section, we briefly review the existing work related to the problem that *ĐArcher* aims to address.

8.1 DApp Development and Testing

In 2017, Porru et al. [40] introduced the concept of blockchain-oriented software engineering, and pointed out the challenges and research directions on the development and testing. Since then, DApps, as a kind of blockchain-oriented software, start to attract attentions from software engineering researchers. Wessling et al. [52, 53] discussed the design choices of architecture of a software that involves blockchain, showing the benefits and drawbacks of DApps. Wu et al. [54, 55] conducted empirical studies on Ethereum-based DApps to show the popularity, growth, development practice, cost, and the open-source status quo. They pointed out the research direction in the synchronization between on-chain and off-chain layer of a DApp, but no further study is conducted. Wu et al. [56] proposed a framework, Kaya, for testing DApps. However, their framework only provides tools to facilitate the manual creation and execution of test cases for DApp. Unlike our work, Kaya does not target at any bugs that are specific to DApps, and does not propose oracles to help automatically reveal bugs in DApps.

8.2 Smart Contract Testing

Lots of studies have been conducted in the past several years to analyze and test smart contracts. Various approaches have been proposed to detect vulnerabilities with symbolic execution [32, 34], fuzzing [25, 26, 28, 38, 57], static analysis [5, 14, 22, 35, 49, 51, 58], mutation testing [7], or machine learning [29] techniques. Multiple empirical studies have also been conducted to review and verify the effectiveness and efficiency of smart contract vulnerability analysis tools [9, 19]. However, these studies only focus on the testing on the on-chain layer of a DApp, i.e., smart contracts. As shown in our evaluation, in which we adopt contract vulnerability oracles from ContructFuzzer [25], testing smart contracts only and neglecting the testing of the interaction between on-chain and off-chain layers cannot ensure the correctness of a DApp. Our work, instead, takes the interaction between the on-chain and off-chain layers into consideration and can detect bugs during the synchronization of the two layers.

8.3 Test Case Generation for Web Applications

Since *ĐArcher* targets at web-based DApps, test case generation in web applications is relevant to our work. Mesbah et al. [33] proposed Crawljax, which can derive a state flow graph for web applications and generate tests to traverse the graph. Since Crawljax is well-maintained and capable of automatically generating test cases with good coverage to explore web applications, we integrate it into *ĐArcher* to trigger the interaction with smart contract for testing DApps. Other test generation tools, such as SubWeb [3] and DIG [4], are usually built based on Crawljax, with the aim to further increase the coverage of generated tests. Since these tools require web applications to have page navigational models, which are not available in our DApp subjects, we did not integrate them into our framework. Nevertheless, the DApp Front-End Explorer component of *ĐArcher* is loosely coupled with other components. It is convenient to migrate to other web testing tools to explore functionalities of DApps and detect on-chain-off-chain synchronization bugs based on our proposed oracles.

9 CONCLUSION

In this paper, we study the development challenges of DApps caused by the non-deterministic transaction execution on the blockchain and the on-chain-off-chain synchronization bugs thus induced. We propose *ĐArcher*, an automated testing framework to detect two common types of such bugs in DApps. Our experiments on 11 real-world DApps show that *ĐArcher* is effective in detecting on-chain-off-chain synchronization bugs and significantly outperforms the baseline methods, in terms of precision, recall, and accuracy. Feedbacks from real-world DApp developers also confirm the effectiveness and usefulness of *ĐArcher*.

REFERENCES

[1] Kerala Blockchain Academy. 2021. AgroChain: Agricultural Supply Chain Dapp with Micro-Finance. https://github.com/Kerala-Blockchain-Academy/AgroChain.

[2] Manuel Beaudru. 2021. TodoList Dapp. https://github.com/mbeaudru/ethereum-todolist.

[3] Matteo Biagiola, Filippo Ricca, and Paolo Tonella. 2017. Search Based Path and Input Data Generation for Web Application Testing. In *Search Based Software Engineering (Lecture Notes in Computer Science)*, Tim Menzies and Justyna Petke (Eds.). Springer International Publishing, Cham, 18–32. https://doi.org/10.1007/978-3-319-66299-2_2

[4] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-Based Web Test Generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Tallinn Estonia, 142–153. https://doi.org/10.1145/3338906.3338970

[5] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *arXiv:1809.03981 [cs]* (Sept. 2018). arXiv:1809.03981 [cs]

[6] ChainSafe Systems. 2021. Web3.Eth.Contract - Methods.MyMethods.Send. https://web3js.readthedocs.io/en/v1.3.0/web3-eth-contract.html#methods-mymethod-send.

[7] P. Chapman, D. Xu, L. Deng, and Y. Xiong. 2019. Deviant: A Mutation Testing Tool for Solidity Smart Contracts. In *2019 IEEE International Conference on Blockchain (Blockchain)*. 319–324. https://doi.org/10.1109/Blockchain.2019.00050

[8] ConsenSys Software Inc. 2021. Ganache. https://trufflesuite.com/ganache.

[9] 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 ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[10] Ethereum Foundation. 2021. Canceling or Replacing a Transaction after It's Been Sent. https://kb.myetherwallet.com/en/transactions/checking-or-replacing-a-tx-after-sending/.

[11] Ethereum Foundation. 2021. Ethereum.Org. https://ethereum.org/en/.

[12] Ethereum Foundation. 2021. Go Ethereum. https://geth.ethereum.org/.

[13] Etherscan. 2021. The Ethereum Blockchain Explorer. https://etherscan.io/.

[14] J. Feist, G. Grieco, and A. Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[15] ConsenSys Formation. 2021. MetaMask Browser Extension. https://github.com/MetaMask/metamask-extension.

[16] Democracy Earth Foundation. 2021. Sovereign Dapp: Create and Participate in Digital Autonomous Organizations (DAOs). https://github.com/DemocracyEarth/sovereign.

[17] Gavin Wood. 2020. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf.

[18] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna Austria, 3–16. https://doi.org/10.1145/2976749.2978341

[19] Asem Ghaleb and Karthik Pattabiraman. 2020. How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 415–427. https://doi.org/10.1145/3395363.3397385

[20] Giveth Decentralized Altruistic Community. 2020. A Community of Makers Building the Future of Giving. https://github.com/Giveth/giveth-dapp.

[21] Giveth/giveth-dapp Issue #1103. 2020. Giveth-Dapp Seems to Assume That Transactions Are Guaranteed to Be Executed. https://github.com/Giveth/giveth-dapp/issues/1103.

[22] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving out-of-Gas Conditions in Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 1–27. https://doi.org/10.1145/3276486

[23] Truffle Blockchain Group. 2021. Ganache-Cli Options. https://github.com/trufflesuite/ganache-cli#options.

[24] Truffle Blockchain Group. 2021. Handle Chain Reorgs in newHeads Event. https://github.com/trufflesuite/ganache-core/issues/91.

[25] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, Montpellier France, 259–269. https://doi.org/10.1145/3238147.3238177

[26] 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 2019)*. Association for Computing Machinery, New York, NY, USA, 363–373. https://doi.org/10.1145/3293882.3330560

[27] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. 2020. A Survey on the Security of Blockchain Systems. *Future Generation Computer Systems* 107 (June 2020), 841–853. https://doi.org/10.1016/j.future.2017.08.020

[28] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 65–68.

[29] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-Gram: Towards Semantic-Aware Security Auditing for Ethereum Smart Contracts. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 814–819. https://doi.org/10.1145/3238147.3240728

[30] PM Research Ltd. 2021. Augur. https://github.com/AugurProject/augur.

[31] PM Research Ltd. 2021. Augur Issue #8260. https://github.com/AugurProject/augur/issues/8260.

[32] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[33] A. Mesbah, A. van Deursen, and D. Roest. 2012. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 35–53. https://doi.org/10.1109/TSE.2011.28

[34] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[35] Bernhard Mueller. [n.d.]. Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit.

[36] Mahesh Murthy. 2021. Ethereum Voting Dapp: Simple Ethereum Voting Dapp Using Truffle Framework. https://github.com/maheshmurthy/ethereum_voting_dapp.

[37] Satoshi Nakamoto. [n.d.]. Bitcoin: A Peer-to-Peer Electronic Cash System. *bitcoin.org* ([n. d.]), 9.

[38] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[39] PaulLaux. 2021. ETH-Hot-Wallet: Ethereum Wallet with ERC20 Support - a Web Wallet. https://github.com/PaulLaux/eth-hot-wallet.

[40] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli. 2017. Blockchain-Oriented Software Engineering: Challenges and New Directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 169–171. https://doi.org/10.1109/ICSE-C.2017.142

[41] Dominik Schiener. 2021. PublicVotes - Ethereum-Based Voting App. https://github.com/domschiener/publicvotes.

[42] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting Taint Specifications for JavaScript Libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 198–209. https://doi.org/10.1145/3377811.3380390

[43] Stephen Gornick. 2021. How Many Confirmations Do I Need to Ensure a Transaction Is Successful? https://bitcoin.stackexchange.com/a/8373.

[44] Roman Storm. 2021. Multisender: Batch Sending ERC20, ETH, Ethereum Tokens. https://github.com/rstormsf/multisender.

[45] ChainSafe Systems. 2021. Web3.Eth - sendSignedTransaction. https://web3js.readthedocs.io/en/v1.3.0/web3-eth.html#sendsignedtransaction.

[46] ChainSafe Systems. 2021. Web3.Eth - sendTransaction. https://web3js.readthedocs.io/en/v1.3.0/web3-eth.html#sendtransaction.

[47] ChainSafe Systems. 2021. Web3.Js. https://web3js.readthedocs.io/en/v1.3.0/.

[48] Kendrick Tan. 2021. Heiswap Dapp - Mix and Mask Your Ethereum Transactions! https://github.com/kendricktan/heiswap-dapp.

[49] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[50] Uniswap. 2021. Decentralized Trading Protocol. https://uniswap.org/.

[51] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–29. https://doi.org/10.1145/3360615

[52] F. Wessling, C. Ehmke, M. Hesenius, and V. Gruhn. 2018. How Much Blockchain Do You Need? Towards a Concept for Building Hybrid DApp Architectures.

In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 44–47.

[53] Florian Wessling, Christopher Ehmke, Ole Meyer, and Volker Gruhn. 2019. Towards Blockchain Tactics: Building Hybrid Decentralized Software Architectures. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 234–237. https://doi.org/10.1109/ICSA-C.2019.00048

[54] Kaidong Wu. 2019. An Empirical Study of Blockchain-Based Decentralized Applications. *arXiv:1902.04969 [cs]* (Feb. 2019). arXiv:1902.04969 [cs]

[55] Kaidong Wu, Yun Ma, Gang Huang, and Xuanzhe Liu. 2019. A First Look at Blockchain-Based Decentralized Applications. *arXiv:1909.00939 [cs]* (Sept. 2019). arXiv:1909.00939 [cs]

[56] Zhenhao Wu, Jiashuo Zhang, Jianbo Gao, Yue Li, Qingshan Li, Zhi Guan, and Zhong Chen. 2020. Kaya: A Testing Framework for Blockchain-Based Decentralized Applications. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 826–829. https://doi.org/10.1109/ICSME46990.2020.00103

[57] Valentin Wüstholz and Maria Christakis. 2020. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 789–800. https://doi.org/10.1145/3377811.3380388

[58] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng. 2020. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1029–1040.

[59] Shijie Zhang and Jong-Hyouk Lee. 2020. Analysis of the Main Consensus Protocols of Blockchain. *ICT Express* 6, 2 (June 2020), 93–97. https://doi.org/10.1016/j.icte.2019.08.001