LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols

Fuchen Ma*, Yuanliang Chen*, Meng Ren*, Yuanhang Zhou*, Yu Jiang*✉,
Ting Chen†, Huizhong Li‡, and Jiaguang Sun*
*School of Software, Tsinghua University, KLISS, BNRist, Beijing, China
† University of Electronic Science and Technology of China, Chengdu, China
‡ WeBank, ShenZhen, China

Abstract—Blockchain consensus protocols are responsible for coordinating the nodes to make agreements on the transaction results. Their implementation bugs, including memory-related and consensus logic vulnerabilities, may pose serious threats. Fuzzing is a promising technique for protocol vulnerability detection. However, existing fuzzers cannot deal with complex consensus states of distributed nodes, thus generating a large number of useless packets, inhibiting their effectiveness in reaching the deep logic of consensus protocols.

In this work, we propose LOKI, a blockchain consensus protocol fuzzing framework that detects consensus memory-related and logic bugs. LOKI fetches consensus states in real-time by masquerading as a node. First, LOKI dynamically builds a state model that records the state transition of each node. After that, LOKI adaptively generates the input targets, types, and contents according to the state model. With a bug analyzer, LOKI detects the consensus protocol implementation bugs with well-defined oracles. We implemented and evaluated LOKI on four widely used commercial blockchain systems, including Go-Ethereum, Meta Diem, IBM Fabric, and WeBank FISCO-BCOS. LOKI has detected 20 serious previously unknown vulnerabilities with 9 CVEs assigned. 14 of them are memory-related bugs, and 6 are consensus logic bugs. Compared with state-of-the-art tools such as Peach, Fluffy, and Twins, LOKI improves the branch coverage by an average of 43.21%, 182.05%, and 291.58%.

I. INTRODUCTION

Consensus protocols are considered as backbones of blockchain systems. They are designed to coordinate nodes to achieve consistency in a distributed network. Various consensus protocols are used in blockchain systems, such as POW [1], POS [2], Raft [3], and PBFT [4]. Due to the complexity of node interactions in blockchain systems, it is hard to avoid bugs in the implementation of consensus protocols. Consensus protocols always play an essential role in blockchain systems so any bug may have severe consequences. For example, three vulnerabilities were recently exposed in the POS consensus

protocol of Ethereum [5]–[7]. By leveraging these bugs, attackers have conducted a DoS attack on Ethereum nodes. The attack also increased the profits of individual validators by short-range recolonizing the underlying consensus chain [8].

Fuzz testing is a promising technique among the approaches to protocol vulnerability detection. One of the most commonly used protocol fuzzers is Peach [9]. Based on user-defined data models, Peach generates test inputs and feeds them to the target program. Peach has successfully detected numerous vulnerabilities in industrial protocols. However, Peach requires human effort to define the protocol's specifications and only performs fuzz testing according to static state models. In the area of blockchain, researchers have recently developed a tool named Fluffy [10] to test the transaction execution process of Ethereum. As a differential fuzzing tool, Fluffy generates a multitude of transactions, feeds them to different Ethereum clients, and has successfully detected two previously unknown bugs in Ethereum by analyzing the differences in the execution results. In addition to the fuzzing tools, researchers have recently developed a tool named Twins [11], [12] to examine the behaviors of BFT systems under Byzantine attacks. As a unit-test generator, Twins implements three types of Byzantine behaviors by duplicating correct node behaviors in a mocking network environment. However, when it comes to consensus protocol vulnerability detection, the complex consensus states of distributed nodes bring the following challenges that existing approaches cannot handle.

The first challenge is that the states of consensus nodes are dynamic, making it difficult to construct their state models precisely in real-time, which leads to ineffective testing. The state of a consensus node consists of its consensus phase and packet sequences. The consensus phase represents the stage of target nodes in the consensus process, and the packet sequences contain all the packets received and sent by LOKI nodes. Without knowing such states, most fuzzing packets will be rejected. For instance, sending 'Preprepare' packets to a leader node will not be accepted in a PBFT system. However, it is hard for existing tools to obtain consensus states in real-time. The entry and exit of consensus nodes are dynamic, and node states are various and independent.

The second challenge is that the inputs to the consensus protocol are multidimensional, making it hard to generate high-quality test inputs. Generally, the input to the consensus protocol consists of three dimensions: target, type, and content. The types and contents of packets should be dynamically

---

adjusted according to the current states of different target nodes. However, Peach and Fluffy can only send fixed types of inputs to settled targets. As for Twins, it only sends packets the same as the correct nodes, which can hardly cover the error-prone paths of the protocols. Another challenge that needs to be addressed is how to effectively use the achieved state information to guide the generation of test inputs.

To address the challenges above, we propose LOKI, a fuzzing framework for detecting vulnerabilities in the implementation of blockchain consensus protocols. LOKI is fully involved in the consensus process by masquerading itself as a consensus node. It fetches the real-time states of other nodes. Then LOKI constructs a dynamic message-driven state model which records the state transitions of each node by analyzing message sequences in the system. In each fuzzing round, LOKI leverages a message guider to determine the next message target and type and mutate the content according to the state model. During the node execution, LOKI uses a bug analyzer to detect memory-related and consensus logic bugs. It uses Address Sanitizer [13] as well as programs' panic mechanisms for memory bug detection. Furthermore, for the consensus logic bugs detection, the analyzer is equipped with two oracles to detect logic bugs that violate liveness [14] and safety [14] properties. In this way, LOKI continuously participates in the execution of the consensus protocol process, and performs effective fuzz testing on it.

We implemented LOKI and evaluated its effectiveness on the consensus protocols of 4 commercial blockchain systems: Go-Ethereum, Meta Diem, IBM Fabric, and WeBank FISCO-BCOS. Results show that LOKI covers 43.21%, 182.05%, and 291.58% more branches than Peach, Fluffy, and Twins. In addition, LOKI has found 20 previously unknown vulnerabilities in these protocols (5 in Go-Ethereum, 3 in Diem, 5 in Fabric, and 7 in FISCO-BCOS.) with 9 CVEs assigned (while others are still in the review): CVE-2021-35041, CVE-2021-43667, CVE-2021-40243, CVE-2021-42219, CVE-2021-43668, CVE-2021-43669, CVE-2022-45196, CVE-2021-46359, and CVE-2022-28936. While Peach and Fluffy only detect 1 of these bugs, respectively, and Twins detects none. We also evaluate LOKI to illustrate its testing overhead. The result shows that LOKI performs 16,203 fuzz iterations on average during 12 hours.

This paper makes the following contributions:

- We propose a method of node masquerading to fetch real-time states of distributed nodes for effective testing of blockchain consensus protocol implementation.
- We implement and evaluate LOKI on four widely used blockchains. LOKI employs a state model builder to construct and update state models promptly and a message guider to produce high-quality messages.
- We open-source LOKI[1] for practical usage. Compared with state-of-the-art tools: Peach, Fluffy, and Twins, LOKI increases the branch coverage by 43.21%, 182.05%, and 291.58% on average. Besides, LOKI detects 20 serious previously unknown vulnerabilities with 9 CVE ids. They all have been confirmed and repaired by the maintainers.

---

[1]LOKI is available at: https://github.com/ConsensusFuzz/LOKI

II. BACKGROUND

A. Blockchain Consensus Protocols

A blockchain system leverages a consensus protocol to achieve overall system reliability. Blockchain consensus protocols are derived from game theory and CAP theorem [15]. Game theory based consensus protocols discourage nodes from violating the consensus by increasing the cost of cheating. Typical protocols of this type include POW and POS. POW stands for 'Proof of Work', a consensus schema used by the Bitcoin system and Ethereum. POW sets the difficulty and rules for the work that a block generation node should do. POS stands for 'Proof of Stake'. This algorithm selects consensus validators in proportion to their quantity of holdings. Game theory protocols are widely used in the permissionless blockchain. The other type is the protocol based on the CAP theory [15]. Commonly used protocols of this type contain PBFT [4], Raft [3] and HotStuff [16]. These consensus protocols ensure the availability and partitioning tolerant by achieving eventual consistency. CAP based protocols are commonly used in consortium blockchain systems due to better performance and faster processing speed.
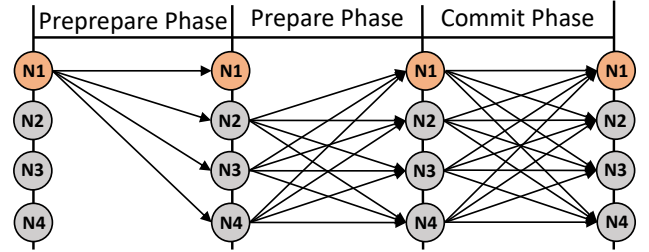

Fig. 1. PBFT needs three phases for consensus. N1 is the leader node.

In general, the state of consensus protocols is more complex than traditional protocols, which other nodes can easily influence. Figure 1 shows the consensus process of 4 nodes using the PBFT protocol. PBFT defines three phases to achieve the consensus. During the Preprepare phase, a chosen leader broadcasts *PrePrepare* packets containing transaction sequences. After receiving a *PrePrepare* packet, a node executes and verifies the transactions. A node will enter the Prepare phase and broadcast a *Sign* packet if the result is correct. Currently, the state of such a node can be described as 'a non-leader node with 1 *Sign* packet sent and n *Sign* packets received' in Prepare phase. After collecting *Sign* packets from more than two-thirds of the nodes, the node broadcasts *Commit* packets. During the Commit phase, the node commits the locally-cached blocks to the database when receiving *Commit* packets from more than two-thirds of the nodes.

When the leader has failed, another node in the PBFT protocol will trigger a timeout. It suspects something is wrong with the current leader. Then it initiates a view change process to negotiate a new leader with other nodes by broadcasting a *Viewchange* packet to move the system to view + 1. If a node receives *Viewchange* packets from two-thirds of the nodes, it will switch to the new view and calculate the new leader.

B. Fuzzing Technique

Fuzzing is considered an effective technique for program vulnerability detection. A fuzzer continuously produces new inputs for the testing program and tries to trigger software bugs. Fuzz testing techniques can be divided into black-box, gray-box, and white-box. Black-box fuzzing produces test cases without knowing the testing programs' behaviours and implementation. Moreover, the gray-box fuzzer leverages the feedback of the programs (such as the coverage) to guide the generation of new test inputs. While white-box fuzzing always constructs test inputs according to the source code of the testing program.

LOKI is designed as a gray-box fuzzer as it leverages the consensus state information to guide its fuzzing decision. Compared with a black-box fuzzer, LOKI is more effective in generating test inputs according to the feedback information. Compared with a white-box fuzzer, LOKI is more general. Blockchain systems differ in languages and architectures and always have a large amount of code. Applying white-box fuzzers to such systems needs massive human efforts. In addition, the source code of blockchain systems is not available in some situations (such as AntChain [17]).

III. MOTIVATION

A. Consensus Protocol Implementation Bugs

Due to the complexity of blockchain node interactions, it is hard to avoid bugs in the implementation of consensus protocols. Figure 2 shows a vulnerability in the consensus protocol of Hyperledger Fabric [18], [19].

```
1  func ChannelHeader(env *cb.Envelope)
         (*cb.ChannelHeader, error) {
2  + if env == nil {
3  +   return nil, errors.New("Invalid envelope
         payload. can't be nil")
4  + }
5    envPayload, err :=
         UnmarshalPayload(env.Payload)
6    if err != nil {
7      return nil, err
8    }
9    ...
10   return chdr, nil
11 }
```

Fig. 2.   Nil pointer bug in the consensus protocol of Fabric. Line 5 tries to access an invalid memory if 'env' is nil. Lines start with '+' describe the code to fix this bug.

Line 5 shows that if the parameter 'env' is nil, the operation 'env.Payload' will try to access invalid memory and raise a SEGV signal. As a result, the leader will crash immediately. If any node is hacked, the whole blockchain will eventually crash. Though Raft only supports CFT, it needs to guarantee essential services. It has been confirmed by the Fabric developers and assigned a CVE identifier: CVE-2021-43667. Developers have fixed it by adding a nil pointer checker, as shown in lines 2-4. Such vulnerabilities in consensus protocols can lead to DoS attacks. Unfortunately, they tend to be hidden in the deep logic of the protocol implementation, making them hard to be detected.

B. Challenges to Detect Such Bugs

Compared with traditional protocols, consensus protocols tend to have more complex states which are dynamically updated during the execution. Several unique state transitions of different nodes should be executed to detect this bug.
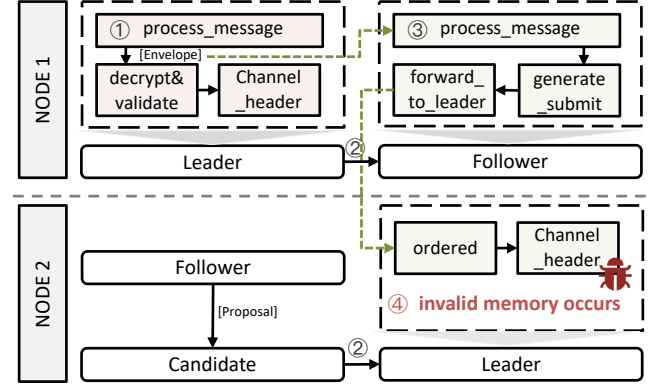

Fig. 3.   The follower forwards an Envelope message with nil payload to the leader. The leader then processes the message without validating it and crashes immediately.

Figure 3 describes the state transition process of two consensus nodes and the trigger of the bug: ① At first, $node_1$ was the leader and $node_2$ was a follower. All the 'Envelop' messages will be decrypted and validated by $node_1$. As the red bars at the top left show, all the invalid messages will be rejected in such a process. ② An election process is proposed. Two key state transitions were triggered – $node_2$ is elected as the new leader while $node_1$ becomes the follower. ③ At this moment, if there are some unprocessed 'Envelope' packets in $node_1$, then a new state arrives, where those packets will be forwarded to the new leader by the follower. ④ $node_2$ processes these packets by the function 'ordered', which does not validate the request and calls the function 'Channel_header' directly. The leader panics when trying to access the nil pointer.

Existing tools like Peach cannot effectively detect this bug. Peach sends pre-defined types of packets to fixed targets based on a static state model. It cannot fetch the real-time states of each node and update its fuzzing strategies. Without the guidance of particular states, Peach generates lots of invalid packets. For example, it may constantly send an 'Envelope' packet to a 'Candidate' node or a 'Proposal' packet to a 'Follower' node. These packets will not be accepted, making it hard for Peach to detect such bugs. While Fluffy only focuses on vulnerabilities in the transaction execution process. It generates a multitude of transactions and feeds them to different clients. Without fuzzing on consensus packets, it is unable for Fluffy to detect such bugs either.

To effectively test consensus protocols, a fuzzer needs to fetch the consensus states in real-time and dynamically update its fuzzing strategy according to the current state. In this example, that is to fetch the states that occur in step ② and ③ and send 'Envelope' packets according to the transition conditions to a follower node. LOKI is designed as a masquerading node that participates in the consensus process for fetching precise states of each node. In this example, a dynamic state model is constructed for tracking all state transitions of $node_1$ and $node_2$ in real-time. LOKI first receives a packet and sets the
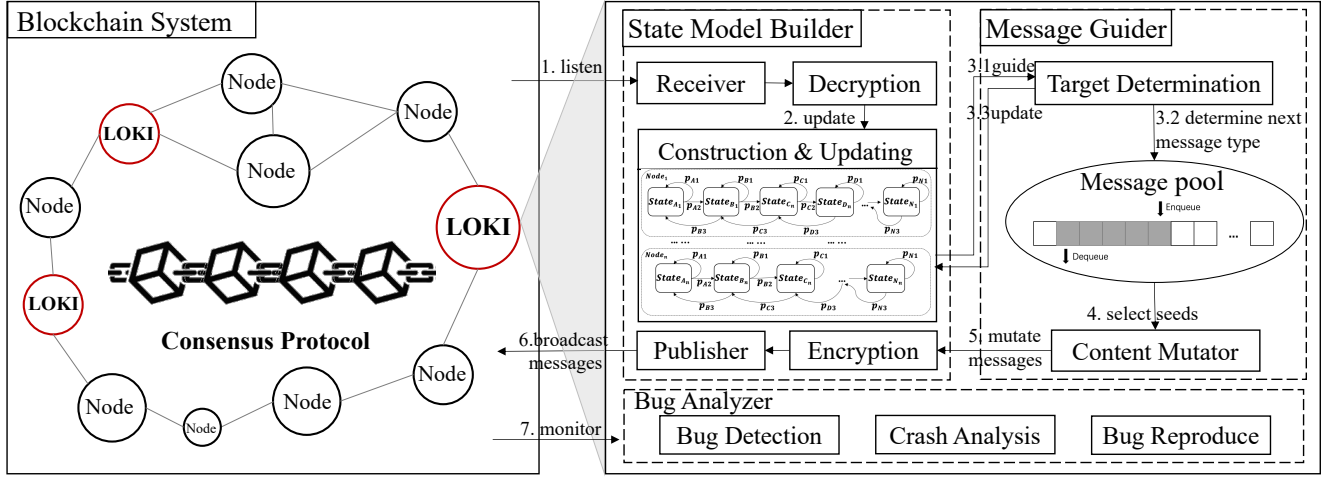
Fig. 4. An overview of LOKI. (1) Module Receiver first listens all the messages, decrypts and analyses them. (2) Next, State Module Builder constructs and updates a state model according to the message sequences. (3) Message Guider determines next message type based on the state model, and updates it. (4) LOKI selects corresponding seeds from the message pool. (5) Content Mutator mutates seeds according to message specifications, and generates new messages. (6) Finally, LOKI encrypts and signs the messages, broadcasts them. (7) The Bug Analyzer monitors the execution information of blockchain system all the time. And LOKI proceeds to the next iteration of the fuzzing process until termination.

state of node 2 to the leader and node 1 to the follower. Afterward, due to the unprocessed 'Envelope' packets in node 1, LOKI fetches an unusual transition that forwards these packets to the new leader. Guided by this transition, LOKI then constructs malicious 'Envelope' packets to the follower node and triggers this bug.

IV. LOKI DESIGN

LOKI is designed to cover the core implementation of consensus protocols for most blockchain systems, from public blockchain, i.e., Ethereum, to consortium blockchains, i.e., Diem [20], Fabric [19] and FISCO-BCOS [21]. By camouflaging the fuzzing nodes into the blockchain system as normal nodes, LOKI fetchs the consensus states in real time and automatically constructs and updates a state model, based on which, LOKI adjusts its fuzzing strategies dynamically. With the help of corresponding bug oracles, LOKI detects consensus protocol implementation bugs in real time.

Figure 4 illustrates an overview of LOKI. There are three key components in LOKI: State Model Builder for fetching real-time states, constructing and updating a dynamic state model; Message Guider for guiding testing states and mutating messages; Bug Analyzer for bug detection, analysis and reproducing. (1) Module Receiver first monitors, decrypts and extracts all the messages from the blockchain system. (2) Based on the received messages, State Model Builder constructs and updates a local state model. Then, LOKI assigns an initial probability for each state transition, and starts the fuzzing process. (3) Based on the state model, Message Guider determines the target state, chooses the next message type and dynamically updates the probability. (4) According to the determined message type, LOKI selects corresponding seeds from the message pool, and delivers them to the Content Mutator. (5) Content Mutator receives those messages and mutates them based on message specifications. Then it generates plenty of new fuzzed messages. (6) Finally, module Encryption encrypts and signs the messages, broadcasts the fuzzed messages through Publisher to the target nodes in blockchain

system. (7) The Bug Analyzer monitors the execution status of nodes in blockchain system. If any consensus bugs or memory-related crashes are detected, then the runtime information will be recorded and the bugs will be collected, analyzed and reproduced. Afterwards, LOKI proceeds to the next iteration (from step 3 to step 6) of the fuzzing process until termination.

A. State Model Builder

The dynamic state model is the key guiding information for the fuzzing process. Formally, *a state* is defined as a triple $< p, R, S >$, where p means consensus phase, R presents the set of received messages and S is the set of sent messages. Consensus phase is defined as the stage of the target nodes in the consensus process, which is automatically inferred by the received messages. For example, for a PBFT system, when a node receives "prepare" messages from others, then it is in the "prepare phase". Once LOKI join the blockchain network, it immediately starts the message loop for listening and collecting messages. All messages are decoded and extracted and key information such as message type, sender and receiver are recorded. Then the State Model Builder divides them into several message sequences by their message types, the identification of senders and the EPOCH numbers. All messages are sorted by the timestamp to record the temporal correlations. High-frequency patterns can be mined from the message sequences and converted into several different state chains. The detailed process is mainly divided into two phases: (1) State model construction and (2) State model updating.

(1) State model construction: To illustrate the state model construction, we take PBFT in FISCO-BCOS as an example, which is presented in Figure 5. When LOKI node joins the blockchain network, it first sends and receives messages as normal nodes do. All messages are analysed in real time, the message type, identifications of senders/receivers and timestamp information are recorded and sorted. Then, based on the $NodeID$ (identifications of message senders/receivers), all messages are divided into several message sequences by LOKI.
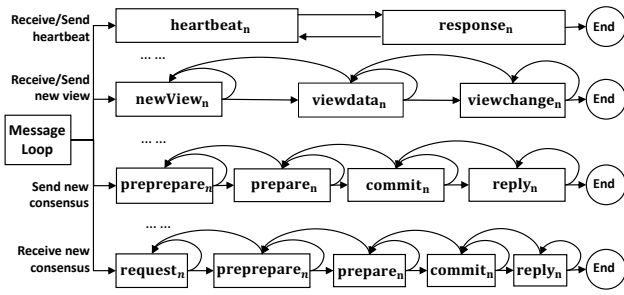
4

Fig. 5. An example of the state model in PBFT of FISCO-BCOS, all state chains are identified by the nodes' ID. They begin from the same message loop node, and end when related nodes exit. This model contains PBFT's formal specifications and also contains other patterns such as heartbeat messages.

Finally, LOKI dynamically analyzes all message sequences, and mines the high-frequency messages patterns in them. If a pattern occurs in all message sequences, LOKI identifies it as high-frequency pattern. The message types are converted to state node, and the following relations are converted to the state transitions. The states and transitions are formed like a chain, which we call the state chain. For example, the State Model Builder can easily find that for each message sequence, every heartbeat request is followed by a heartbeat response, then a heartbeat request to the same node is sent. So the heartbeat pattern – request and response are followed by each other is mined. Then a state chain is constructed, state *heartbeat* and state *response* links each other. Similarly, LOKI automatically figures out that for all message sequences, *viewdata* messages always follow *newView* messages, and *viewchange* messages always follow *viewdata* messages. Then the high-frequency pattern ($newView \rightarrow viewdata \rightarrow viewchange$) is mined and the corresponding state chain is constructed.
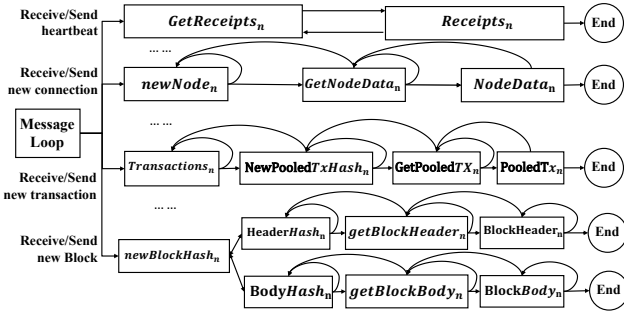

Fig. 6. An example of the state model in the POW protocol of Go-Ethereum.

To help understand the state model construction better, we also take pow in Go-Ethereum for example, as presented in Figure 6. After analyzing all message sequences, the State Model Builder can easily find that every *GetReceipts* is followed by a *Receipts*, which is similar to the heartbeat mechanism of Fabric. LOKI also detects that *BodyHash* and *HeaderHash* messages always follow *newBlockHash* messages, then are followed by *getBlockHeader*, *getBlockBody* messages and *BlockHeader*, *BlockBody* messages repsectively. Thus, the state chains '$newBlockHash \rightarrow HeaderHash \rightarrow getBlockHeader \rightarrow BlockHeader$' and '$newBlockHash \rightarrow BodyHash \rightarrow getBlockBody \rightarrow BlockBody$' are constructed.

By automatically mining high-frequency patterns in these message sequences, more and more state chains have been constructed gradually. When there is no new pattern mined, the construction process ends. The state model is formed as a tree, all state chains are derived from the initial state ($MessageLoop$), which we call the root node. Finally, to help explore abnormal message sequences, the State Model Builder also creates transitions for each state to its own state and its previous state. Then it assigns each state transition with an initial probability (1/n if there are n edges). The 'probability' means the possibility that Message Guider selects the next message type to transit to the target state. This probability is dynamically adjusted, if a transition triggers new bugs or makes new coverage, its probability increases.

During the state model building process, LOKI may not capture all states, but it will continuously explore new message patterns by trying unobserved state transitions during the recursive state model updating process as we will describe in the following. As a heuristic search algorithm with randomness, LOKI's core idea is sending mutated messages to test uncommon behaviours. LOKI constantly tries to cover these behaviours during the state model updating phase.

Algorithm 1: State Model updating

Input : Root node of the state model tree: $root$,
Vector of state chains: $chains$,
Message received: msg

1 **Function** treeUpdate(*root, chains, msg*):
2   // new message type is found
3   **if** checkNewMsgType *(msg)* **then**
4   record *(msg)*;
5   newchain = newChain *(msg)*;
6   *chains*.add(newchain);
7   **async:**
8   newAnalyser().trackFollows(msg);
9   **if** *isNewFollows(msg')* **then**
10          $newchain$.links(msg') ;
11   **end**
12   **end async**
13   **end**
14   // message from new node
15   **if** checkNewMsgId *(msg)* **then**
16   chain = getChainBytype *(msg*.type);
17   chain.updateState(msg);
18   *root*.newchild(chain);
19   **end**
20   // message from existing state chain
21   chain = getChainById *(msg*.Id);
22   chain.updateState(msg);
23   **if** *chain.state == end* **then**
24   *root*.removeChild(chain);
25   **end**
26 **End Function**

---

(2) State model updating: Once the LOKI's state model construction ends, the model updating and fuzzing processes begin. Algorithm 1 illustrates the process of updating the state model tree. For every message received from the network or generated by LOKI, function 'treeUpdate' first checks whether the message is a new message type. If a new type is found,

LOKI records it and creates an analyzer for tracking subsequent messages asynchronously. All following messages will be recorded and analysed for mining new potential patterns (if new patterns found, then links them to the new state chain). Otherwise, it checks whether the message comes from a new node. If the sender is new, LOKI will create a state chain from existing state patterns, and add it into the root node. If the message's id matches existing state chain, then get the chain, update its state. Finally, State Model Builder checks whether the current state chain comes to the end state. If so, LOKI will remove it from the state model tree. The model updating process runs through the fuzzing process. If the state chain has not been updated for a long time, LOKI will try edges that don't exist in the model with a low probability. By performing such byzantine behaviours, LOKI continuously explores new message patterns and self-improves the state model.

B. Message Guider

Figure 7 shows the flow chart of Message Guider. It first receives a message type list containing all the message types in the blockchain system. Then, based on the state model with dynamic transition probability, the Message Guider determines the target fuzzing state and selects the next message type which can trigger this state transition. According to the determined type, Seed Selector selects the corresponding messages from the message pool, and sends them to the Content Mutator for mutation based on the message specification. Finally, the Signer and the Publisher sign the mutated messages and send them to other consensus nodes. All messages which contribute to new states and state transitions are regarded as interesting message seeds, and are stored in the message pool.
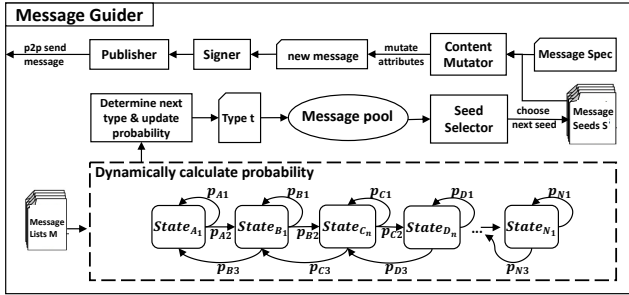

Fig. 7. Flow chart of Message Guider. The guider decides the following fuzzing destinations and message type, mutates the content and sends the mutated message.

Target state determination: Once the state model has been constructed, LOKI will start the fuzzing process immediately. Based on the state model, the Message Guider needs to determine the next target state and select the related message type to reach it. Algorithm 2 illustrates the process of how the Message Guider determines the target state. First, the Message Guider traverses all the neighbourhood nodes. For each connection of a neighbouring node, LOKI maintains a separate state model for them. Those models are distinguished by their $NodeID$. They are generated or eliminated with the dynamic entry and exit of nodes. Message Guider uses depth-first search to get corresponding state chains from the state model by their current states. Then for each state chain, the Message Guider gets their transition probability models, and

calculates a random number based on them. Finally, each state chain transfers its state according to the transition probability p and returns the corresponding message type to reach the target state. More specifically, once the next message types have been determined, related messages will be mutated and sent to the neighbourhood nodes for execution. This process is asynchronous, and a new thread worker is awakened to wait for the feedback information, as shown in lines 10 - 16. If any message can reach new states or trigger new state transitions, the corresponding probability p will increase accordingly.

Considering that the LOKI node maintains a separate state model for each connected node and that the state transfer for each state model is based on the transition probability. Furthermore, different probabilities p may lead to different next message types for different nodes. As a result, LOKI node may sent different types of messages to different nodes. With this Byzantine behaviour, LOKI performs continuous and efficient fuzz testing to the target blockchain systems.

Algorithm 2: Target state determination

Input : Vector of neighbourhood nodes: $peers$
Output: Vector of next message types: $msgs$

1 **Function** DetermineNextMsg(*peers, root*):
2   msgs = []
3   **for** $peer \in peers$ **do**
4   // dfs root, get corresponding state chain ;
5      $root = peer$.getStateModel($peer$.Id);
6      $chain = root$.getchainByState($peer$.State);
7      $p = $ Random ($chain$.getProbability());
8      $nextmsg = chain$.transfer(p);
9      $msgs$.push ($nextmsg$);
10   **async:**
11   fuzzedmsg = mutate (nextmsg);
12   feedback = p2p.send(peer, fuzzedmsg);
13   **if** isNewState *(feedback) or* isnewTransition *(feedback)* **then**
14          $chain$.updateProbability (p);
15   **end**
16   **end async**
17   **end**
18   **return** $msgs$
19 **End Function**

Content mutation: The performance of the fuzzing process is influenced by the quality of the input seeds. Since messages in the blockchain system are various, highly structured and well-formatted, the quality of input messages is determined by the message specification. LOKI automatically synthesises the specifications by extracting and analyzing the message stream(which means message binary data received from the blockchain) in the blockchain system.

To illustrate the process of content mutation, Figure 8 takes the $Preprepare$ message in Fabric for example. All messages in Fabric are serialized into the Protocol Buffers (protobuf) format [22]. According to its encoding structure [23], specification information – pair $< type, size >$ will be constructed and divided by their message types.

Figure 9 gives an example of the specification for the $Preprepare$ message. Each pair represents a field in the
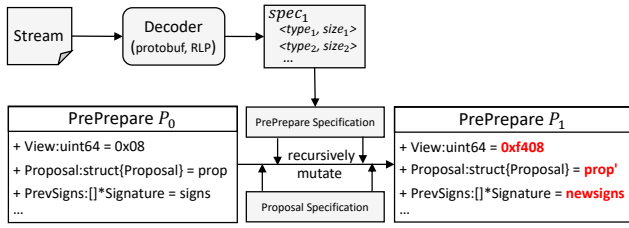
Fig. 8. Flow chart of the mutation process. LOKI recursively mutates message fields based on specifications.

message. The size is defined by a number or a keyword 'unfixed'. The struct field contains the pair of each element. Based on the specifications, LOKI first recursively searches

Fig. 9. An example of the generated spec for the $Preprepare$ message. Each pair represents a field of the message and the size is calculated in bytes.

them ($Preprepare$ and $Proposal$ specifications), and mutates their fields: all the fields are mutated by the corresponding mutators. More specifically, LOKI mutates different types of data in different ways:

- **For the numeric type**, a numerical mutator will randomly convert it to another number, especially to some border value such as INT_MAX and 0. For example, the field 'View' in Figure 8 are mutated from $0x08$ to $0xf408$.
- **For the string type**, a string mutator will modify it by flipping its bytes or bits which is similar to the mutation strategies in AFL [24].
- **For the structure types** (from prop to prop'), LOKI recursively mutates each field. The basic types such as numeric/string fields are mutated directly, and the structure fields are handled recursively.
- **For the cryptography field** such as hash or signature, LOKI uses the inherent cryptographic components provided by each blockchain platform to process the corresponding functions. To masquerade itself as a normal node, LOKI node is implemented in the plug-in mode – the LOKI core components are plugged into the original normal node implementation. Thus, each LOKI node contains a normal node which has already implemented the cryptographic components. LOKI uses them directly to process hash/sign. LOKI mutates the message contents before the hash or sign functions are performed. In this way, LOKI can always generate message stream which has valid cryptography related fields.

To ensure the validity of the seeds after mutation, LOKI maintains the full structure of the message packages. This is the main novelty of LOKI's mutation strategies. Thus, the mutated messages will still be valid when deserialized by other nodes. Finally, the new message P_1 is mutated and sent to the target node for fuzzing.

Seed selection: All messages in the seed pool are categorized by their message types. As shown in Figure 10, there are a serial of seeds queues divided by message types in the message pool. (1) Once the Message Guider determines which
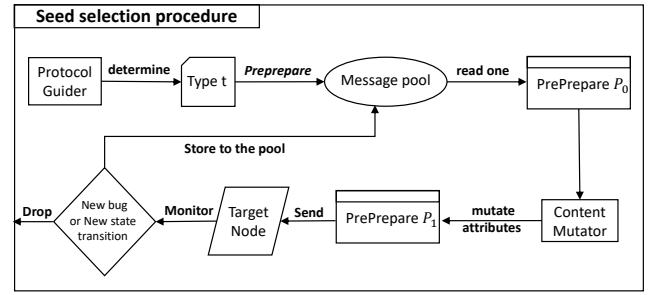
type to use in the next iteration, for example, a "preprepare" message, then the seed selector will dequeue the "preprepare" p_0 message queue for further mutation process. (2) p_0 message is mutated recursively according to the specifications. (3) Then a new "preprepare" message p_1 is generated. LOKI will send it to the target node and monitor its execution information. (4) If there are new bugs, states or state transitions found in the target node, the new 'preprepare' message will be regarded as interesting and saved to the seed pool for the next iteration.

C. Bug Analyzer

Bug Analyzer is designed to monitor the runtime information of target nodes and identify their exceptional behaviors, as show in Figure 11. There are two main types of bugs LOKI can detect: memory-related bugs and consensus bugs. Memory monitor is responsible for analyzing memory model and detecting memory-related bugs of target nodes. Consensus monitor is designed to analyze data states of blockchain ledger and identify consensus bugs.
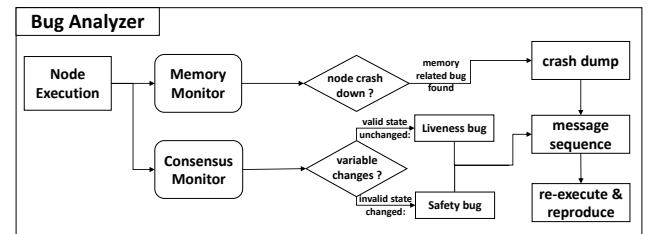

Fig. 11. The workflow of the bug analyze process. LOKI monitors the memory problems and consensus exceptions while nodes process messages.

Memory-Related Bug: LOKI detects memory-related bugs by observing whether a node crashes down. Since many vulnerabilities such as heap overflow usually won't directly crash the node, we use Address Sanitizer (ASAN) [13] for detecting and analyzing the latent bugs. With the help of ASAN and the programs' panic mechanisms, LOKI can capture and record the crash dump information of target nodes immediately. Based on them, the call stack and the root cause of the crash can be analyzed with tools like GDB [25] or LLDB [26]. It should be mentioned that the memory bug analyzer targets on all non-LOKI nodes. This is because we only want to detect memory related bugs generated under the original logic of the consensus protocols, without the influence of LOKI's logic.

Consensus Bug: LOKI can also be extended with oracles to detect consensus bugs that violate liveness [14] and safety [14] properties of blockchain system. Liveness means a valid transaction in a blockchain system is committed eventually. Safety means illegal transactions in blockchain systems will never be committed. LOKI first continuously sends a set of valid transactions to the blockchain system. Then, LOKI marks state variables that transactions intend to change and monitors these variables. If they are not changed for a long time, LOKI considers there is a liveness-violating bug. For the safety bug, LOKI regularly applies mutation strategies on the transaction field in the consensus packets to create invalid transactions and marks related states. Then LOKI monitors these state variables the transactions intend to change. If their values are changed, LOKI considers there is a safety-violating bug. Details of the consensus bug definition can be reffered to [27].

The oracles are generated based on the original definitions in the PBFT paper [4]. For example, safety is defined as 'the replicated service satisfies linearizability: it behaves like a centralized implementation that executes operations atomically one at a time.' This is an abstract definition at the design level, while our safety oracle is a concrete definition at the implementation level. For example, double spending is a concrete bug that can be captured by our rules and be prevented if the replicated service satisfies linearizability. In the blockchain context, our safety oracle can capture bugs that violate the formal definition in the original paper.

Bug Reproduce: LOKI records all received messages of a node and sorted them by the timestamp. We find the first states that the monitors output bugs while processing the test cases, which we call triggering states. We also find the last state that can reach the triggering states when used as the starting point of nodes execution, which we call the starting state. When a bug occurs, LOKI replays these messages between starting state and triggering states to reproduce the bug and help analyze the root cause.

V. IMPLEMENTATION

We implement our framework, LOKI, on four commonly-used commercial blockchain platforms, including one public blockchain – Go-Ethereum and three consortium blockchains – Diem, Fabric and FISCO-BCOS. They are chosen because they are diverse. As shown in Table I, all the blockchain systems are implemented in different languages. In addition, their consensus protocols are various. Implementation on these blockchain systems can demonstrate that LOKI is a cross-platform and language-free framework. We also open-sourced LOKI in Github repository. LOKI is implemented in Rust. We choose Rust because it has a good support of Foreign Function Interface (FFI), which allows LOKI's function to be easily called by target blockchain systems. Besides, we also introduce the basic file structure of LOKI in the README.md of the repository. Furthermore, for some languages which do not support ASAN such as Go, LOKI checks whether there is a bug by the panic mechanism.

To masquerade itself as a normal node, LOKI node is implemented in the plug-in mode – the LOKI core components are plugged into the original normal node implementation. In this way, LOKI can be quickly adapted to a blockchain system.

TABLE I.   DETAIL INFORMATION ABOUT 4 BLOCKCHAIN SYSTEMS TO EVALUATE LOKI AND THE CONSENSUS PROTOCOLS THEY USED.

Blockchain	Consensus	Company	Language	Version
Ethereum	POW	Ethereum Org	Go	#eb94896 [28]
Diem	DiemBFT	Meta	Rust	#4b3bd1e [29]
Fabric	Raft	IBM	Go	#cd88c60 [30]
FISCO-BCOS	PBFT	WeBank	C++	#09fb48e [31]

Figure 12 presents the implementation of LOKI Node, which can be divided into three parts. The first part is the Normal Node part, which contains all the functions of a normal node, including message handling, field cryptography and signature, block generation, etc. Although this part is specific to the target blockchain systems, their implementations are available and can be used directly. The second part is the LOKI Core Components, which is implemented for constructing the state model and dealing with the fuzzing process. It is totally independent and free from the target blockchain systems. These components can be reused when adapt to a new blockchain system. The third part is the Interface Adaptation, which is responsible for uncoupling the normal node and the core components of LOKI. This part needs to be developed from scratch when adapt to a new system.
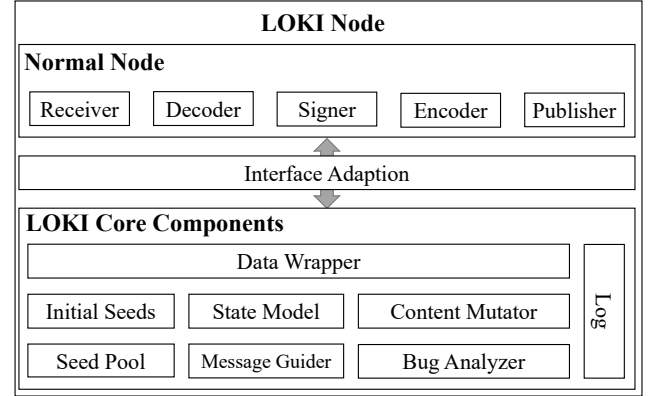

Fig. 12.   The implementation of LOKI node, contains three parts – Normal Node part for node masquerading and message handling. LOKI Core Components part for state model and fuzzing process. Interface Adaption for uncoupling Normal Node part and LOKI Core Components.

Initial Workload: LOKI utilizes the testing transaction generation programs in each blockchain system to trigger the whole consensus process. For FISCO-BCOS, we use the provided stress testing scripts [32]. This script generates transactions based on some predefined contracts. For Diem, we use the 'cluster-test' program [33] provided by the developers. This program generates mint transactions. For Fabric, we use the transactions generated based on the contracts in Fabric samples [34]. For Ethereum, we utilize a transactions firing tool named chainhammer [35].

Initial Seeds: Initial seeds are critical to the performance of the fuzzing because they determine the initial fuzzing direction. High-quality initial seeds are hard to write manually, while random seeds cannot ensure coverage. In our implementation, LOKI collects the messages in the network as the initial seeds. The messages here mean the consensus packages sent by the normal nodes and the network means the blockchain network. Once LOKI nodes join the blockchain network, all messages

from the normal nodes will be listened to, recorded, executed and tracked. Any messages which contribute to new states or transitions will be regarded as interesting seeds and stored into the message pool as the initial seeds.

Message Parser and Encryptor: The messages in diverse blockchain systems are different in both structure and format. More specifically, the encoding, decoding, cryptography and signature processes of those messages are unique. However, benefit from the plug-in mode implementation, each LOKI node contains a normal node which has already implemented the encoder, decoder, cryptographer and signer. As inherent functions, LOKI uses these components directly. For example, Fabric and Diem use protobuf to serialize and deserialize the messages, so we use the $Marshal$ and $Unmarshal$ packages. Ethereum and FISCO-BCOS utilize RLP [36] to encode and decode messages, so we directly use the eth/rlp package.

Effort of Adaptation: LOKI is implemented such that each module is encapsulated and loosely coupled. Therefore, when adapting LOKI to a new system, developers only need to implement four interfaces related to a specific chain and invoke LOKI's initialization routines at startup. The interfaces are: (1) interfaces to wrap and unwrap data that convert between the LOKI's chain-independent packet structure representation and the specific chain's message structure; and (2) interfaces to send and receive packets that send the mutated packets to Normal Node and pass the received message packets to LOKI Core Components through the Rust's FFI.

General steps to adapt LOKI are as follows: 1) Download LOKI source code. 2) Write 4 interfaces related to specific structures in system under test (SUT). 3) Start SUT. 4) Start fuzzing engine by calling LOKI initialization function. 5) Finish fuzzing. We take LOKI adaption in FISCO-BCOS for example, we only wrote 533 lines of code. The detailed adaption process is introduced in Appendix IX-E. For more examples, please refer to the repository of LOKI.[2]

VI. EVALUATION

To evaluate the effectiveness of LOKI, we compared it with three state-of-the-art tools: Peach [9], Fluffy [10] and Twins [11], [12]. We ran a blockchain network of 10 nodes. The binary of Diem and FISCO-BCOS are hardened by AddressSanitizer [13] to detect latent bugs. The initial seeds for all experiments are the same. For LOKI, we set up a group with 10 nodes for each target blockchain system and set 3 of them as LOKI nodes. For Peach, we also built a 10-node group and set the related ports of all the nodes in Peach's publisher. In this way, Peach can continuously send the generated packages to all nodes in the group. As for Fluffy, we followed its documentation [37] to start the testing. For Twins, we ran the 'cargo xtest' command for the pre-defined strategies [38] in Diem's source code. All the experiments are conducted several times on a 64-bit machine with 128 cores (AMD EPYC 7742). The OS of the machine is Ubuntu 20.04.1 LTS, and the main memory is 512 GB. We design experiments to address the following research questions:

- **RQ1:** Is LOKI effective in finding implementation bugs of real-world consensus protocols?

- **RQ2:** Can LOKI cover more code logic of consensus protocols compared with state-of-the-art tools?
- **RQ3:** What is the overhead of LOKI?
- **RQ4:** How LOKI performs under multiple LOKI nodes.

A. Bugs in Consensus Protocols

We ran LOKI and Peach on all four blockchain systems for 24 hours. Since Fluffy is only designed for the Ethereum platform, and Twins only supports Diem, we ran Fluffy and Twins for 24 hours on Go-Ethereum and Diem, respectively. LOKI detects 20 previously unknown consensus protocol implementation vulnerabilities, including 5 in Go-Ethereum, 3 in Diem, 5 in Fabric, and 7 in FISCO-BCOS. Among them, 14 vulnerabilities are memory-related, and the other 6 are consensus logic bugs that violate the liveness or safety property. Their details are listed in Table II.

All the bugs have been confirmed and repaired by the corresponding vendors, and at the time of paper submission, 9 bugs have been assigned with CVEs in U.S. National Vulnerability Database (while others are still in the review). 4 of the bugs (#1, #2, #10, #17) are of the type 'Invalid Memory'. They allow the programs to access invalid memory addresses and throw SIGSEGV or SIGBUS signals. 7 of the bugs (#4, #6, #7, #9, #11, #12, #15) are of the type 'Unexpected Panic'. This type of bug arises because the system cannot correctly handle some unique inputs, causing the entire node to crash. 2 of the bugs (#14, #16) are related to memory free. Specifically, bug #14 fails to free the unused memory and continuously consumes system resources. Bug #3 is of the 'Data Race' type. This bug results in a deadlock and triggers a denial-of-service attack. 4 of the bugs (#5, #8, #18, #19) are logic bugs that violate the liveness property of the blockchain system. This kind of bug may prevent valid transactions from being committed, and more seriously, the whole system will stop producing new blocks. And 2 of the bugs (#13, #20) are logic bugs that violate the safety property of the system. These bugs lead to the confirmation of illegal transactions.

Peach is able to detect 1 bug (#15) in Table II. The reason is that this bug occurs in the start phase of the consensus protocol. It can be occasionally triggered with little state information. As for Fluffy, it can also detect 1 of these bugs (#4). Fluffy constructs a set of transactions that call EIP-1283 and finally triggers this bug. As for Twins, it detects none of these bugs because Twins only supports three kinds of byzantine behaviors, all of which cannot trigger the hidden states of the protocol in Diem. The unique vulnerabilities detected by LOKI need to be triggered by specific packet sequences along with real-time state transitions, which Fluffy, Peach, and Twins do not support.

1) Accuracy of LOKI: LOKI does not have false positives in our evaluation. For memory-related bugs, LOKI detects node crashes either led by program panic mechanisms or ASAN. As for consensus logic bugs, LOKI has no false positives either by checking the final states of a transaction. If a valid transaction's state is not changed or an invalid transaction's state has been changed, there is a bug.

False negatives are hard to collect because we do not know exactly how many bugs are in these platforms. However, we added an experiment where we used LOKI to reproduce 12

9

#	Platform	Bug Type	Bug Description	Identifier
1	Go-Ethereum	Invalid Memory	SIGBUS: read a invalid memory when generating DAG on multiple nodes.	CVE-2021-42219
2	Go-Ethereum	Invalid Memory	SIGSEGV: nil pointer in newBlockIter during block sync with fast mode.	CVE-2021-43668
3	Go-Ethereum	Data Race	Resource access conflict in dialScheduler when miner enters network.	Bug#23965
4	Go-Ethereum	Unexpected Panic	VM crashes when executing multiple transactions in system contract EIP-1283.	Bug#23866
5	Go-Ethereum	Liveness	The chain indexer that caused repeated "chain reorged during section processing" errors.	Bug#24447
6	Diem	Unexpected Panic	The address conflicts with other processes when restarting consensus nodes.	Bug#1339041
7	Diem	Unexpected Panic	The validator node try to fetch an unreachable hash from cache.	Bug#9753
8	Diem	Liveness	Malicious nodes cause the failure of processing some transactions and stucks the chain	Bug#10228
9	Fabric	Unexpected Panic	Orderer crashes down after receiving an invalid config message.	Bug#15828
10	Fabric	Invalid Memory	Leader fails after receiving a nil payload message forwarded by followers.	CVE-2021-43667
11	Fabric	Unexpected Panic	Orderer breakdowns when marshalling an invalid envelope formation.	Bug#18529
12	Fabric	Unexpected Panic	Leader in consensus protocol crashes down when parsing an invalid Envelope Header.	CVE-2021-43669
13	Fabric	Safety	Repeatedly creating channel after receiving requests with the same Channel name.	CVE-2022-45196
14	FISCO-BCOS	Memory Unfree	Memory is not freed when dealing with sustained consensus packets.	CVE-2021-35041
15	FISCO-BCOS	Unexpected Panic	Private key cannot be parsed by consensus protocol.	CVE-2021-40243
16	FISCO-BCOS	Bad Free	Front service of a consensus node attempts to free an unallocated memory.	Bug#72
17	FISCO-BCOS	Invalid Memory	Read an invalid memory when starting a block sync process.	Bug#71
18	FISCO-BCOS	Liveness	Bug in checking txpool limit when receive transactions from p2p.	CVE-2021-46359
19	FISCO-BCOS	Liveness	Block not be executed if the synchronization execute it before the addExecutor.	Bug#2132
20	FISCO-BCOS	Safety	A fake proposal's header leads to the successful consensus of illegal blocks.	CVE-2022-28936

recent bugs [39] for 4 platforms (3 bugs for each platform) and found 2 of them cannot be detected by LOKI. These 2 bugs are related to data races, which are hard to be reproduced. This indicates that LOKI has a false negative rate of 16.67%. The bugs are listed in Section IX-H.

2) Case Study: Now we use two cases to illustrate how the bugs detected by LOKI affect the whole system. **The first case is the bug #20 listed in Table II**. This bug is assigned with a CVE ID: CVE-2022-28936. It is found in version 3.0 of FISCO-BCOS. The bug is a consensus logic bug that violates the safety property. The code snippet in the following figure describes detailed information about the vulnerability.

```
1  void verifyProposal(PublicPtr fromNode,
2      PBFTProposalInterface::Ptr proposal,
3      function<void(Error::Ptr, bool)> handler)
4  {
5  // Bug Fix: add the checker for the block header
6  + auto block = m_blockFactory->
7  +   createBlock(proposal->data());
8  + auto blockHeader = block->blockHeader();
9  + if(blockHeader->number()!=proposal->index())
10 + { if (handler)
11 +   { auto error = std::make_shared<Error>
12 +     (-1, "Invalid proposal");
13 +     handler(error, false);
14 +   } return;
15 + }
16 // Bug here: just verify the data of block
17 m_txPool->asyncVerifyBlock
18     (fromNode,proposal->data(),handler);
19 }
```

Fig. 13. A bug that breaks the safety property in the implementation of PBFT protocol in FISCO-BCOS release-3.0.0.

As shown in Figure 13, FISCO-BCOS uses a function named 'verifyProposal' to check the blocks in a new proposal from other nodes. This function takes in three parameters. The first is the source node of the proposal, and the second is the pointer of the proposal. While the third is a callback function to handle the error. As shown in lines 17 and 18, it calls the function 'asyncVerifyBlock' to check the proposal's block data. However, it overlooks the verification of the block header and leads to the successful consensus of the illegal block. Thus, during the fuzz testing, LOKI fakes a block's header when it fetches that it is the leader, and the consensus nodes will not detect the invalid headers. However, during the execution phase, the execution nodes will always fail to recognize these blocks and leave them for further execution. Thus, more invalid blocks are accumulated, and all the memory will finally be consumed. Line 6 to line 15 shows the code to fix this bug: add a checker for the block's header and stop invalid blocks from reaching consensus.

This bug can only be detected by LOKI because it only occurs in the state where the leader sends blocks with illegal headers. LOKI can fetch the state that it is the leader and mutate different fields of the packets, thus triggering this bug. Peach cannot fake itself as a leader, other nodes will not accept its proposal, and the bug will never appear.

The second one is the bug #2 listed in Table II. This bug was found in the block sync process of Go-Ethereum (version 1.10) in the fast mode. Figure 14 shows the detail of this vulnerability. The function 'find' is designed to find (key, value) a pair whose key is greater than or equal to the given key. First, it will get an $indexBlock$ through the function 'getIndexBlock' by current Reader r, and then call the function 'newBlockIter' based on this indexBlock. The function 'newBlockIter' is used to create a new $blockIter$ for traversing all the (key, value) pairs in a block. However, if the current node cannot fetch a valid block from the local blockchain, the function 'getIndexBlock' will return a nil pointer. Since there is no nil checker in the caller, the program crashes when receiving the nil pointer. This bug has been assigned with a CVE ID: CVE-2021-43668.

This vulnerability is hard to be detected because, in most cases, the function 'getIndexBlock' would always return a normal block. A long sequence of fuzzed messages and transactions should be constructed and sent to the target node to trigger it. After receiving those well-fuzzed messages and

```
1   // Find key/value pair based on the given key
2   func (r *Reader) find(key []byte, ...) (...) {
3     ...
4     indexBlock, rel, err := r.getIndexBlock(true)
5     if err != nil {return}
6     defer rel.Release()
7     index := r.newBlockIter(indexBlock, nil, nil,
          true)
8     defer index.Release()
9     ...
10  }
11
12  func (r *Reader) newBlockIter(b *block, ...)
        *blockIter {
13    bi := &blockIter{
14      tr:          r,
15      block:       b,
16      ...
17      // block *b is a nil pointer
18      riLimit:     b.restartsLen,
19      offsetStart: 0,
20      ...
21    }
22    ...
23  }
```

Fig. 14.   Nil pointer in Go-Ethereum's fastsync mode. If leveraged by malicious nodes, an honest node in the fastsync process can be broken down.

transactions, the target node constructs an abnormal block and stores it in the local blockchain. However, the 'getIndexBlock' function will fail to deal with such a block and return a nil pointer, eventually causing a node crash.

B. Code Coverage

In order to evaluate whether LOKI can cover more code logic of the target systems, we built a local network with 10 nodes (3 of them are LOKI nodes) for each blockchain. We calculated how many branches are covered in 24 hours. It should be noted that we only calculate the coverage for evaluating the effectiveness, and LOKI will not collect the code coverage at runtime. The results are shown in Table III.

In conclusion, LOKI covers 10,058 branches on Go-Ethereum (Geth), which is 48.04% and 182.05% more than Peach's 6,794 and Fluffy's 3,566. While on Fabric and FISCO-BCOS, LOKI covers 12,117 and 14,794 branches, respectively, which is 31.96% and 66.79% more than Peach's 9,182 and 8,870. And for Diem, LOKI covers 31,534 branches. This improves the coverage by 26.05% and 291.58% by Peach's 25,018 and Twins' 8,053 branches.

TABLE III.   BRANCH COVERAGE OF LOKI AND OTHER TOOLS. '-' MEANS THAT THE TOOL DOES NOT SUPPORT THE BLOCKCHAIN.

	Go-Ethereum	Diem	Fabric	FISCO-BCOS
LOKI	10058	31534	12117	14794
Peach	6794	25018	9182	8870
Fluffy	3566	-	-	-
Twins	-	8053	-	-

From the first column, we can see that LOKI covers more than twice as many branches as Fluffy. The reason is that Fluffy is designed to test the transaction execution logic in EVM. Many consensus processes, such as leader election,

block commitment, and view changing, cannot be tested by Fluffy. As for Peach, LOKI covers over 40% branches on Go-Ethereum, Fabric, and FISCO-BCOS because it dynamically decides the type and the content of the sent messages according to the state models. While Peach only tests fixed targets with settled types of messages under static state models.

While on Diem, LOKI covers 291.58% more branches than Twins. The reason is that Twins is a unit test generator. It can just mock a network environment for BFT protocol testing. Thus it cannot cover the code used in the actual runtime of blockchain consensus protocols such as block verification and block commitment. Besides, LOKI covers 26.05% more branches than Peach on Diem. Diem defines plenty of safety rules for the consensus process, unlike the other three blockchain systems. All the consensus packets that do not belong to the current stage will not be accepted and will not affect the following consensus process. The mutated packages constructed by LOKI trigger more assertions in such rules, which leads to the increment of coverage.

To observe the coverage trend, we record the coverage every 20 minutes for 2 hours. After that, the coverage of LOKI and other tools basically converge (only less than 1% coverage improvement). Similarly, the state model constructed by LOKI is basically completed. However, LOKI can still generate boundary values to trigger more bugs. The results are shown in Figure 15. The blue bars in each figure refer to the coverage of LOKI, while the orange bars describe the coverage of Peach. The red bars in figure (a) represent the coverage of Fluffy. And the purple bars in figure (b) represent the coverage of Twins. The lines in each figure show the increment percentages of LOKI compared with Peach.

According to Figure 15, LOKI's coverage grows significantly in the first 40-80 minutes on Diem, Fabric, and FISCO-BCOS. While on Go-Ethereum, the coverage of LOKI constantly increases rapidly before 100 minutes. This is because Go-Ethereum uses POW consensus protocol, which requires the consensus nodes to calculate a hash value that meets a specific condition. Therefore, each consensus round takes a relatively long time. Accordingly, LOKI needs more time to update the states, leading to a slower convergence. Peach and Fluffy's coverage proliferates in the first 60-80 minutes. After that, the generated packets can hardly cover more new branches as they do in the beginning. As a unit test generator, the coverage of Twins does not change over time.

Besides, we can also find that the slope of the percentage growth of LOKI is always positive. This means that LOKI's coverage has always been growing faster than Peach's. The main reason is that LOKI can always construct more high-quality input packets than other tools, benefiting from the extracted message specifications and the dynamic real-time state models.

C. Overhead of LOKI

In this section, we evaluate the testing overhead of LOKI on the fuzzing iteration. We compared LOKI with Peach and Fluffy on the 4 blockchain platforms in 24 hours (Twins is not a fuzzer). The result is shown in Figure 16. LOKI has an average fuzzing iteration of 16,203 on 4 platforms. While Peach performs 260 iterations on average. As for Fluffy, the fuzzing iteration is 4,919 on Ethereum.
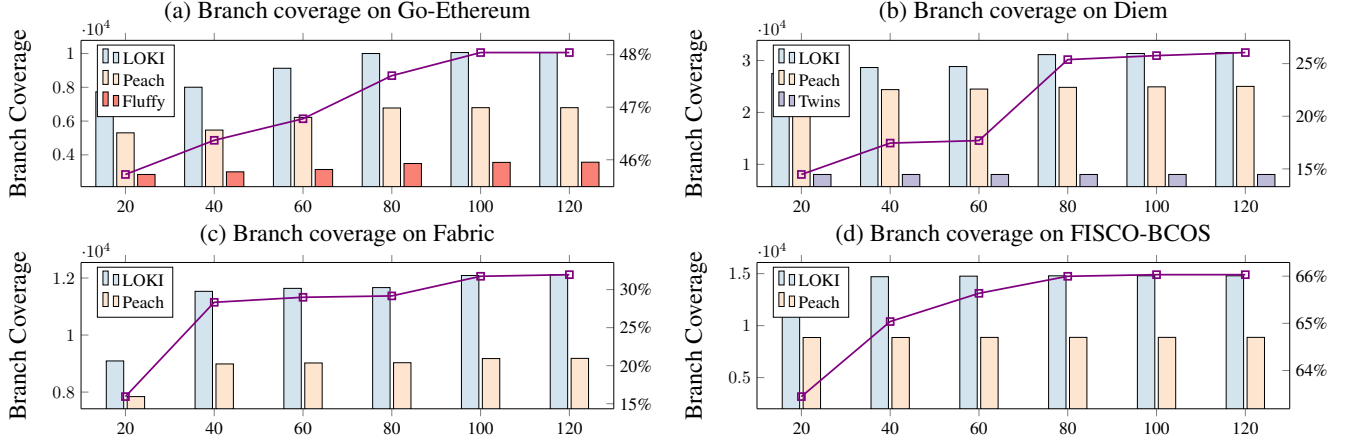
Fig. 15. Coverage trends evaluated for LOKI, Peach and Fluffy on Go-Ethereum, Diem, Fabric and FISCO-BCOS. **The bars** in the figure represent the coverage while **the lines** describe the increment percentage of LOKI compared with Peach. The coverage of Fluffy is zero for Diem, Fabric and FISCO-BCOS because it only supports Ethereum. The same is for Twins on Go-Ethereum, Fabric and FISCO-BCOS. After two hours, the coverage of all tools grow slowly (only less than 1% coverage improvement is observed). But after 2 hours, LOKI can still generate some boundary values to trigger more bugs.
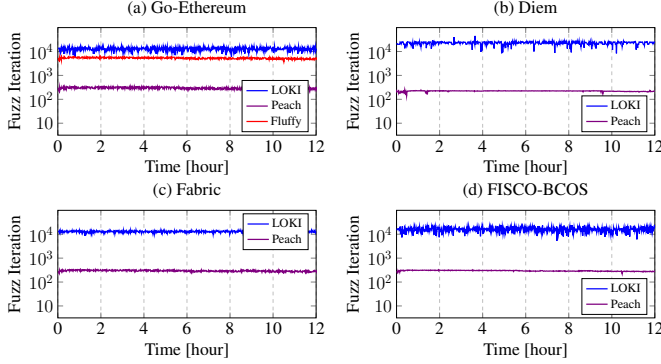

Fig. 16. Fuzz iterations of LOKI, Peach and Fluffy in 12 hours.

Fig. 17. Branch coverage under different numbers of LOKI nodes. The branch coverage is generally the same under various LOKI nodes.

Peach has a relatively low fuzzing iteration because it generates each test case from scratch. While LOKI produces new test inputs by mutating specific fields of the packets in the message pool. As for Fluffy, it is less efficient than LOKI because the test case is more complex. Targeting the virtual machine, Fluffy leverages multiple transactions as test inputs. However, LOKI generates new test cases by constructing or mutating protocol packets, which are much smaller than blocks and transactions.

D. Effectiveness under multiple LOKI nodes

We also conducted an experiment to find out the effect of multiple LOKI nodes. We set up a group of 10 nodes for each blockchain with 1, 2, and 3 LOKI nodes and calculated the branch coverage. The results are shown in Figure 17. We only collected the coverage for 2 hours because, after that, the coverage basically converged for all groups. The result shows that the coverage finally converges at a similar value (with only +/-5% of the variance) under various LOKI nodes. More LOKI nodes may slightly accelerate the coverage increment.
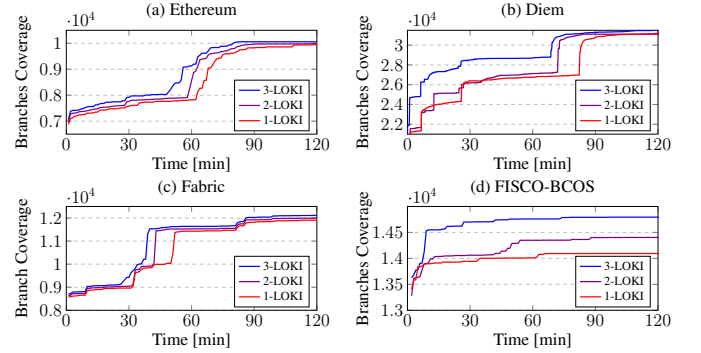
VII. DISCUSSION

In this section, we will discuss some advantages and limitations of LOKI, and our future work.

Scalability on Different Platforms. With the lightweight plug-in design, LOKI has strong scalability across languages and blockchain systems. For non-blockchain distributed systems, LOKI only applies to those with a similar consensus protocol to the blockchain, such as the IPFS cluster [40] with the Raft protocol. IPFS is a distributed file system that seeks to connect all computing devices with the same system of files. Though we have tested the Raft protocol of Fabric, they are different in the implementation (language and architecture). Besides, the workload is different when adapting to a non-blockchain system. For a blockchain system, the workload is the transaction based on smart contracts, while for IPFS, the workload is an I/O-related operation such as file uploading. In our experiment, we successfully migrated LOKI to a popular distributed management software ipfs-cluster [41].

We created a 3-peer IPFS cluster, which holds two normal peers and one fuzzing peer equipped with LOKI. The fuzzing peer will collect packets from other nodes to perceive the current consensus state and perform targeted mutations on the

returned packets. Table IV shows that after running for 24 hours, LOKI covered 1,946 branches. Compared with Peach, LOKI improves the coverage of consensus protocol logic by 43.09% through precise state perception and efficient mutation. A nil pointer dereference happens in syncing snapshots and log entries from the leader node. The root cause is that the order of internal variables in *followerReplication* structure violates the requirement for atomic operations. The corresponding developers have fixed this bug.

TABLE IV. BRANCH COVERAGE ON IPFS-CLUSTER WITHIN 24 HOURS.

	LOKI	Peach	Fluffy	Twins	Improvement
Coverage	1946	1360	-	-	+ 43.09% / - / -

Generality on Other Distributed Protocols. Currently, LOKI mainly focuses on detecting bugs in blockchain consensus protocols. This is because consensus protocols are the backbones of blockchain systems. Any bugs within these protocols may result in severe consequences. For example, a recent vulnerability [42] discovered in Hyperledger Fabric's consensus protocol can crash an orderer node and lead to a DoS attack. However, a limitation of LOKI may be that it is not suitable for some generally distributed protocols, such as gossip-based protocols. To build the state model of consensus nodes, LOKI requires some expectation that the messages received indicate a node's current state rather than a past state. In some general distributed protocols, however, the network packets cannot represent the current state of the protocol. In these scenarios, LOKI may not implement the state chains correctly, affecting its effectiveness. Thus, for now, LOKI focuses on blockchain consensus protocols.

Consensus Protocols with Concurrency LOKI utilizes the EPOCH number to divide the received messages for concurrent protocols. The consensus protocol we tested for FISCO-BCOS is the Parallel PBFT. This protocol introduces a pipeline mechanism that operates different consensus states with concurrency. LOKI can also support such protocols. For example, during the Commit phase, LOKI may receive commit messages as well as prepare messages at the same time. However, they contain different EPOCH numbers. LOKI divided the state model according to the EPOCH numbers and updated different state chains based on these messages. As we illustrated in Section VI, LOKI covered 14,797 branches and found 7 bugs in the Parallel PBFT of FISCO-BCOS, proving that LOKI is also effective on concurrent protocols.

VIII. RELATED WORK

Vulnerability Detection in Blockchain. Usually, the security of blockchain systems contains two layers, 1) security of the application layer (smart contracts), and 2) security of the backbone layer (EVM, consensus protocols, etc.).

Recently, many researchers have devoted themselves to the development of smart contract vulnerability detection tools. Some work uses static analysis methods to construct intermediate expressions and rule paradigms to match specific vulnerability patterns, such as Zeus [43], SmartCheck [44] and Securify [45] for reentrancy or overflow vulnerabilities, Mad-Max [46] for gas-related vulnerabilities and Pied-Piper [47] for contract backdoors. Some work use symbolic values as

input to simulate the execution of the program. For example, Oyente [48], Mythril [49] and their extensions [50]–[53]. However, these tools only support vulnerability in single contract, Pluto [54] models the inter contract call process to handle this problem. There are also some dynamic tools focus on smart contract vulnerability detection, such as ContractFuzzer [55], sFuzz [56] and ReGuard [57]. Guided by coverage feedback, they select and randomly mutate inputs within the seed pool. Another tool named V-Gas [58] uses fuzzing technique to find the high-gas consumption inputs to avoid out-of-gas situations. In addition, authors of SCStudio [59], [60] have proposed a contract vulnerability detection tool by integrating some basic tools selected through their previous empirical study [61]. However, all the above mentioned tools are designed for smart contracts and cannot be adapted to consensus protocol testing.

As for the backbone layer, there is also targeted research work. For example, EVMLab [62] generates random bytecode of contracts and invokes them with a single transaction. EVMFuzzer [63] and Fluffy [10] generate multi-transactions and use different EVMs as cross-referencing factors to observe abnormal behaviours. EVM* [64] and Sereum [65] further realize real-time blocking of dangerous transactions based on opcode sequence analysis. While Twins [11], [12] is designed as an automated unit test generator of Byzantine attacks.

Protocol Testing. Protocol Testing is a method of checking communication protocols in the domains of Switching, Wireless, VoIP, Routing, etc. The widely used detection methods include formal verification, symbolic execution, fuzzing and so on. SNAKE [66] is a tool that automatically finds performance and resource exhaustion attacks on unmodified transport protocol implementations. McMillan et al. [67] developed a formal specification of QUIC based on the draft standards documents, and used this specification to generate test inputs and validate output results for implementations of QUIC. SymbexNet [68] is a symbolic execution based tool for network protocol implementations. It automatically generates high-coverage test input packets for manual rules violations detection Whalen et al. [69] performs anomaly detection of network protocols with the minimal HMM architecture inferred from data.

Fuzzing is also an effective method to detect vulnerabilities in protocol implementation. Traditional fuzzers can be divided into two categories: mutation-based and generation-based. The mutation-based ones, represented by AFL [24] and LibFuzzer [70], collect initial seeds and use them to generate new inputs by some byte/bit level operations. The generation-based ones, such as Sulley [71], Peach [9], and Peach* [72], require user-provided data models to obtain the format specification of each element, then utilize it to complete test seeds. There are also some work optimized these tools from the perspective of reducing manpower expense. For example, Polar [73] uses static code analysis and dynamic taint analysis technology to automatically extract some critical protocol information. PAVFuzz [74] focuses on the protocols used in the vehicle system, learns the relations between two data elements in different states, and uses these relations to calculate and update the dynamic mutation weights.

Main Difference. Different from the above work, LOKI is a novel and effective fuzzing framework for blockchain consensus protocol implementations. For most protocol testing

tools, they cannot handle blockchain consensus protocols in an effective way for three reasons: 1) They rely on a pre-defined static state model for testing, limiting them to sending fixed testing packets. 2) They mutate the packet in one dimension, limiting them from exploring most of the state spaces. 3) Traditional fuzzers are separated from the test object and do not care about the internal state of the system. In contrast, LOKI disguises itself as a normal node and generates high-quality inputs based on the model based on a dynamic state model. In this way, it can achieve deeper code logic and precisely determine the type as well as the target of next packet. In addition, LOKI is not limited to specific objects, and can be easily extended to different languages and platforms.

IX. CONCLUSION

In this paper, we propose LOKI, a state-aware fuzzing framework for the implementation of blockchain consensus protocols. Designed as a masquerading node, LOKI fetches the consensus state in real time and constructs a state model to record state transitions for consensus nodes. We implement and evaluate LOKI on four commercial blockchain systems. The results show that LOKI improves the branch coverage by an average of 43.21%, 182.05% and 291.58% compared with state-of-the-art tools Peach, Fluffy and Twins. LOKI detected 20 previously unknown vulnerabilities with 9 CVE IDs, while Peach and Fluffy only detected 1 of them and Twins found none. Our future work will focus on enhancing LOKI with fined coverage feedback and adapt LOKI on more blockchain consensus protocols.

REFERENCES

[1] J. FRANKENFIELD, "Proof of work," https://www.investopedia.com/terms/p/proof-work.asp, 2021, accessed at October 23, 2021.

[2] ——, "Proof of stake," https://www.investopedia.com/terms/p/proof-stake-pos.asp, 2021, accessed at October 23, 2021.

[3] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[4] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[5] CVE-2021-42764, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42764, 2021.

[6] CVE-2021-42765, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42765, 2021.

[7] CVE-2021-42766, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42766, 2021.

[8] C. Schwarz-Schilling, J. Neu, B. Monnot, A. Asgaonkar, E. N. Tas, and D. Tse, "Three attacks on proof-of-stake ethereum," *arXiv preprint arXiv:2110.10086*, 2021.

[9] M. Eddington, "protocol-fuzzer-ce," https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce, 2021, accessed at October 23, 2021.

[10] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in ethereum via multi-transaction differential fuzzing," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 349–365.

[11] S. Bano, A. Sonnino, A. Chursin, D. Perelman, and D. Malkhi, "Twins: White-glove approach for bft testing," *arXiv preprint arXiv:2004.10617*, 2020.

[12] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," in *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[13] C. . documentation, "Address sanitizer," https://clang.llvm.org/docs/AddressSanitizer.html, 2021.

[14] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977.

[15] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, no. 10.1145. Portland, OR, 2000, pp. 343 477–343 502.

[16] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus in the lens of blockchain," *arXiv preprint arXiv:1803.05069*, 2018.

[17] AntGroup, https://antchain.net/, 2022.

[18] sard Chen, "Sigsegv: segmentation violation in orderer after recieving a message generated by fuzzer," https://jira.hyperledger.org/projects/FAB/issues/FAB-18529?filter=allissues, 2021.

[19] Hyperledger, "Hyperledger fabric," https://www.hyperledger.org/use/fabric, 2021, accessed at October 23, 2021.

[20] Diem, "Welcome to the diem project," https://www.diem.com/en-us/, 2021, accessed at October 23, 2021.

[21] FISCO, "Fisco bcos," https://github.com/FISCO-BCOS/FISCO-BCOS, 2021, accessed at October 23, 2021.

[22] Google, "Protocol buffers," https://developers.google.com/protocol-buffers, 2021, accessed at October 23, 2021.

[23] ——, "Protocol buffer structure encoding," https://developers.google.com/protocol-buffers/docs/encoding, 2021, accessed at October 23, 2021.

[24] ——, "American fuzzy lop," https://github.com/google/AFL, 2015.

[25] Wikipedia, https://en.wikipedia.org/wiki/GNU_Debugger, 2022.

[26] L. org, https://lldb.llvm.org, 2022.

[27] 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 *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 1186–1201. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00068

[28] E. org, https://github.com/ethereum/go-ethereum/tree/eb948962704397bb861fd4c0591b5056456edd4d, 2022.

[29] M. Diem, https://github.com/diem/diem/tree/testnet, 2021.

[30] I. Fabric, https://github.com/hyperledger/fabric/tree/cd88c6087081068c67f844182661b45ce250ad80, 2022.

[31] W. FISCO-BCOS, https://github.com/FISCO-BCOS/FISCO-BCOS/tree/09fb48e56bbc1b4da7bcdbd62d81927e489af110, 2022.

[32] FISCO-BCOS, https://fisco-bcos-doc.readthedocs.io/zh_CN/latest/docs/develop/stress_testing.html, 2022.

[33] Diem, https://github.com/diem/diem/tree/testnet/testsuite/cluster-test, 2022.

[34] ——, https://github.com/hyperledger/fabric-samples, 2022.

[35] Ethereum, https://github.com/drandreaskrueger/chainhammer, 2022.

[36] E. Wiki, "Rlp," https://eth.wiki/fundamentals/rlp, 2021.

[37] Fluffy, https://github.com/snuspl/fluffy#readme, 2022.

[38] Diem, https://github.com/diem/diem/blob/416fdd3b18d02b2a099917467910d5a6865f2fed/consensus/src/twins/basic_twins_test.rs, 2022.

[39] ConsensusFuzz, https://github.com/ConsensusFuzz/LOKI/blob/main/reproduce.md, 2022.

[40] J. Benet, "Ipfs - content addressed, versioned, p2p file system," *ArXiv*, vol. abs/1407.3561, 2014.

[41] IPFS, "Pinset orchestration for ipfs," https://github.com/ipfs/ipfs-cluster, 2020.

[42] CVE-2022-31121, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31121, 2022.

[43] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018.

[44] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, and Y. Alexandrov, "Smartcheck: static analysis of ethereum smart contracts," in *the 1st International Workshop*, 2018.

[45] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *ACM Conference on Computer and Communications Security*, 2018.

[46] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[47] F. Ma, M. Ren, L. Ouyang, Y. Chen, J. Zhu, T. Chen, Y. Zheng, X. Dai, Y. Jiang, and J. Sun, "Pied-piper: Revealing the backdoor threats in ethereum erc token contracts," *ACM Transactions on Software Engineering and Methodology*, 2022.

[48] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," *IACR Cryptology ePrint Archive*, p. 633, 2016.

[49] ConsenSys, "Mythril," https://github.com/ConsenSys/mythril-classic, 2018.

[50] C. F. Torres, M. Steichen *et al.*, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1591–1607.

[51] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[52] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[53] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[54] F. Ma, Z. Xu, M. Ren, Z. Yin, Y. Chen, L. Qiao, B. Gu, H. Li, Y. Jiang, and J. Sun, "Pluto: Exposing vulnerabilities in inter-contract scenarios," *IEEE Transactions on Software Engineering*, 2021.

[55] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*, 2018. [Online]. Available: http://dx.doi.org/10.1145/3238147.3238177

[56] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," *arXiv preprint arXiv:2004.08563*, 2020.

[57] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 65–68.

[58] F. Ma, M. Ren, F. Ying, W. Sun, H. Song, H. Shi, Y. Jiang, and H. Li, "V-gas: Generating high gas consumption inputs to avoid out-of-gas vulnerability," *ACM Transactions on Internet Technology (TOIT)*, 2018.

[59] M. Ren, F. Ma, Z. Yin, H. Li, Y. Fu, T. Chen, and Y. Jiang, "Scstudio: a secure and efficient integrated development environment for smart contracts," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 666–669.

[60] M. Ren, F. Ma, Z. Yin, Y. Fu, H. Li, W. Chang, and Y. Jiang, "Making smart contract development more secure and easier," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1360–1370.

[61] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: what is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 566–579.

[62] Ethereum, "Evmlab," https://github.com/ethereum/evmlab, 2020.

[63] 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 *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1110–1114.

[64] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi, "Evm*: From offline detection to online reinforcement for ethereum virtual machine," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 554–558.

[65] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[66] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 1–12.

[67] K. L. McMillan and L. D. Zuck, "Formal specification and testing of quic," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 227–240.

[68] J. Song, C. Cadar, and P. Pietzuch, "Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 695–709, 2014.

[69] S. Whalen, M. Bishop, and J. P. Crutchfield, "Hidden markov models for automated protocol learning," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2010, pp. 415–428.

[70] Fiware.org, "Libfuzzer," https://llvm.org/docs/LibFuzzer.html, 2015.

[71] P. Amini and A. Portnoy, "Sulley," https://github.com/OpenRCE/sulley, 2012.

[72] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "Ics protocol fuzzing: coverage guided packet crack and generation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[73] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ics protocol," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[74] F. Zuo, Z. Luo, J. Yu, Z. Liu, and Y. Jiang, "Pavfuzz: State-sensitive fuzz testing of protocols in autonomous vehicles," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 823–828.

[75] FISCO-BCOS, https://github.com/FISCO-BCOS/FISCO-BCOS/blob/b8b6c2a71bafde0b4b3fe2e0ce090678786be144/tools/BcosAirBuilder/build_chain.sh#L684, 2022.

[76] Hyperledger, "Hyperledger caliper," https://hyperledger.github.io/caliper/, 2021.

APPENDIX

A. LOKI under Various Node Scales

To test whether LOKI is effective under a large-scale network which is closer to the real-world scenario, we calculate the branch coverage of LOKI with 10, 20, 50 and 100 nodes. The results are shown in the Table V. We can find that the coverage does not vary much under different network scales (with only a variance of +/-1%). Besides, there are no new bugs found when the scale of the nodes increases. This indicates that LOKI's effectiveness remains the same when we increased the node scales.

TABLE V.   BRANCH COVERAGE UNDER DIFFERENT NODE SCALES.

	Go-Ethereum	Diem	Fabric	FISCO-BCOS
10-node	10058	31534	12117	14794
20-node	10103	31247	12157	14770
50-node	10037	31175	12188	14737
100-node	10112	31204	12145	14763

The reason is that a blockchain node only interacts with a certain number of its neighborhood nodes. Thus, the interaction logic is basically the same under various node scales. Specifically, in our definition, a state is defined as a triple $< p, R, S >$, where p is consensus phase, R means received message types and S represents sent message types. The size of the consensus protocol state space in the blockchain is affected by the interactive nodes (LOKI's neighborhoods) rather than the scale of the nodes.

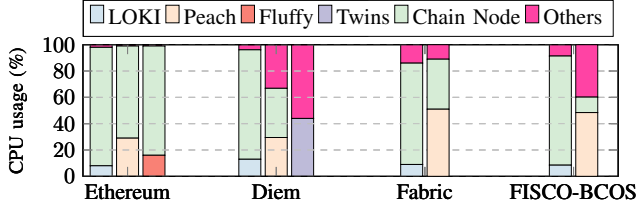### B. CPU Usage Breakdown of LOKI and Other Tools

Fig. 18.   The cpu usage of various tools. Each bar contains three parts: the corresponding tool, the chain node and other processes.

Moreover, we also calculate the CPU usage breakdown of all the tools. As figure 18 shows, LOKI takes only 10% of the CPU on average. In LOKI's case, most CPU time is spent on the blockchain nodes, where the consensus protocol bugs are found. State model construction process is combined with the fuzzing process. Thus state model construction still has a low overhead. In Twins' case, the CPU usage is around 44%. It needs to perform the settled byzantine strategies and costs plenty of CPU resources. Twins will not startup diem nodes to perform the test. Thus we cannot calculate the chain node's CPU usage in its case. As for Peach, it spent more CPU (39%) to execute its own code, including the data model extracting and packet producing. Fluffy has a similar CPU usage (16%) as LOKI. However, it cannot find the bugs detected by LOKI because it only targets the execution of EVM.

C. Bug Detection Time

In this section, we give the time of discovered bugs of LOKI, Peach and Fluffy (Twins did not detect any bug in our evaluation.). It should be noted that, we listed all the bugs in four platforms in the same plot.
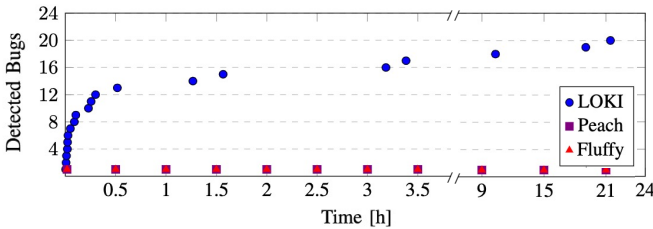
Fig. 19.   Number of bugs detected by LOKI and other tools over time. We omitted the time period from 3.5 to 9 hour because no bugs are found in this period. Each circle point in the figure represents a bug detected by LOKI. While the squares and triangles mean the bug detected by Peach and Fluffy.

The figure shows that LOKI found 13 bugs within the first hour. This indicates that many bugs in consensus protocols

can be effectively detected by LOKI's mechanism. During 1-2 hour, LOKI can still cover more branches(according to the results in Figure 15) and detect 2 new bugs. While after 2 hours, the coverage is converged, but by generating some boundary values, LOKI still found 5 bugs in this period.

As for Peach and Fluffy, each of them can detect 1 bug. Peach detected Bug#15 in Table II within 5 minutes (similar as LOKI). This is a bug which occurs in the very start phase of the consensus protocol. While Fluffy detected Bug#4 within 10 minutes. LOKI detected this bug in about 203 minutes. The reason is that this bug is due to an EVM error when executing a transaction based on the EIP-1283 contract. Fluffy focuses on generating multiple transactions for EVM test, thus it can trigger this bug quickly.

D. LOKI Deployment

In practice, LOKI's entire framework runs locally at each LOKI node without a central coordinator. There are basically 3 steps to deploy and start LOKI:

- **Compile LOKI nodes.** Designed as a plugin mode, LOKI can be compiled into several dynamic libraries. Developers need to link these libraries to the blockchain nodes and generate the binary file with LOKI's functions. Thus, LOKI nodes can be executed as normal nodes are.
- **Prepare a blockchain group with LOKI nodes.** Generally, each blockchain system provides a script to setup a group of nodes. This script always set the binary file of the blockchain node (For example, at line 684 in FISCO-BCOS's build_chain.sh [75] script.). Developers should change the binary of several nodes in the group to the compiled LOKI nodes. The number of LOKI nodes should be determined by the corresponding fault tolerant strategies. For BFT-based protocols (Diem's DiemBFT and FISCO BCOS's PBFT), there should be no more than 1/3 nodes being set as LOKI. While for Go-Ethereum's POW, there should be less than 50% LOKI nodes.
- **Start the blockchain group with LOKI nodes.** This step starts the prepared group. Each LOKI node collects the states of other nodes and generate messages under the direction of the state model builder and the message guider. Each LOKI nodes maintain a different state model and generate different messages. In this group, normal nodes are treated as black boxes by LOKI. While LOKI senses normal nodes' states by analyzing message sequences.
- **Start generating the workload.** This step generates transactions for the settled blockchain group to trigger the complete consensus process. For different blockchain systems, there are different ways to generate the workload. We have given the details of the workload generation in Section V.

E. An Example of LOKI Adaption

When adapting LOKI to a new protocol, four interfaces are required: wrap/unwrap interfaces and send/receive interfaces. Here we give an example of FISCO-BCOS's send interface. The complete adaption code can be found at our repository.

As the code in Figure 20 shows, the send interface will first create the node id. To satisfy the parameter format of the message sending function of the FRONTSERVICE, the interface then creates a vector to keep the ids of the target nodes.

```
1  extern "C" {
2  void send_packet(string target_id, unsigned
       char* _data){
3   auto keyFactory =
       std::make_shared<KeyFactoryImpl>();
4   unsigned char *u_target_id = new unsigned
       char[target_id.length()+1];
5   strcpy((char*) u_target_id, target_id.c_str());
6   // create the node id
7   auto node_id = keyFactory->createKey(
8     bytesConstRef((byte*)u_target_id,
9     target_id.length())));
10  // create the node id vector
11  auto bcosNodeIDs =
       std::make_shared<std::vector<NodeIDPtr>>();
12  bcosNodeIDs->reserve(1);
13  bcosNodeIDs->push_back(node_id);
14  // prepare the sent data
15  auto data = bytesConstRef((byte*)_data,
       strlen((char*)_data));
16  // send the PBFT messages to target nodes
17  FRONTSERVICE->asyncSendMessageByNodeIDs(
18    ModuleID::PBFT,*bcosNodeIDs,data
19  );
20  }
21  }
```

Fig. 20.   An example of the send iterface for FISCO-BCOS's PBFT. LOKI uses this interface to send generated messages to other nodes.

Afterwards, the interface prepares the sent data as shown at line 15. Finally, the interface utilizes the FRONTSERVICE's message sending function provided by FISCO-BCOS to send the data to target nodes.

F. More Bug Cases

Another bug case is the bug #14 listed in Table II. This bug is assigned with a CVE ID: CVE-2021-35041. The code snippet in Figure 21 describes the detailed information of the vulnerability. The function 'P2PMessageRC2::decode' is used

```
1  ssize_t P2PMessageRC2::decode(...){ ...
2    m_length =
         ntohl(*((uint32_t*)&buffer[offset]));
3    if (size < m_length) {
4  // the value of PACKET_INCOMPLETE is 0
5      return dev::network::PACKET_INCOMPLETE;
6    }
7    ...
8  }
9  // code for handling the decoding result
10 ssize_t result =
       message->decode(s->m_data.data(),
       s->m_data.size());
11 ...
12 else if (result == 0) {
13 // m_length size of memory is allocated
14    s->doRead();
15    break;
16 }
```

Fig. 21.   Code snippet that constantly allocate new memory. An attacker can sustain sending maliciously constructed packets to consume all the memory of the honest node's host and break it down.

to decode the received packets. The code at line 2 reads the first 4 bytes of data as the length of the received packet. If the current size of the packet is less than the target length, the node believes that the current packet has not been completely received and returns a signal: 'PACKET_INCOMPLETE' whose value is 0. This signal will be handled by the code at line 12, which will read more data from the session. The function 'doRead()' will allocate memory with size m_length for the incoming packet. If a malicious node sends a packet continuously, the node will consume the memory sustainably. During our experiment, almost 4 GB of memory was taken within about 80 seconds. As a result, the node will consume all the memory of the host machine and be killed by the operating system. An attacker can easily construct a packet with a large value of the field 'length' and keep sending it to the target node. The target will finally crash and lead to a DoS attack. This vulnerability has been fixed.

G. TPS Overhead of LOKI and ASAN

For the evaluation of the throughput, we calculate the value of 'transactions per second (TPS)' for all the consensus protocols of four blockchain systems. We use Hyperledger Caliper [76], a widely used blockchain performance benchmark, to give a report of TPS. Guided by Caliper, TPS is defined as the following equation.

$$TPS = \frac{trans\_num}{\lim_{n \to \infty}(|t_1^c - t^s|^n + |t_2^c - t^s|^n + \cdots)^{1/n}} \quad (1)$$

The numerator of the formula represents the number of all transactions. The denominator of the formula represents the maximum amount of time each node spends on confirming all transactions. The symbol t_i^c indicates the moment when node i confirmed all transactions, while t^s means the moment when the transactions are submitted. In a blockchain system, various nodes may have different confirmation time for a transaction. The whole system only considers a transaction as confirmed when all participants have confirmed it. So TPS is defined as the ratio of the total number of transactions to the maximum time it takes for a node to confirm these transactions. We calculated the TPS of each project under different node scales. The results are shown in Figure 22.

From the Figure 22 we can see that as the number of peers grows, the throughput of the blockchain network decreases. This is because the system complexity will grow with the number of nodes. However, as illustrated in figure (a), Go-Ethereum has a TPS of 371 with 10 peers while 394 with 20 peers. The reason is that Ethereum leverages POW to achieve consensus, whose throughput depends on the nodes computing power as well as the communication complexity. When the node size is increased from 10 to 20, the TPS is improved as the computing power has a greater effect on the gain in throughput. While after that, the impairment caused by the communication complexity has a greater impact on the TPS.

Generally, LOKI decreases the TPS by 2.2% to 6.5% compared with the original nodes. The negligible overhead is incurred mainly because LOKI's fuzz strategy causes the consensus nodes to process more packets, slowing down the original process of consensus. This indicates that one promising application of LOKI is to do large-scale security testing in a local test environment which is close to the practical scenario, before the release of a new version of the blockchain platform. It needs to be mentioned here that LOKI will not be used in
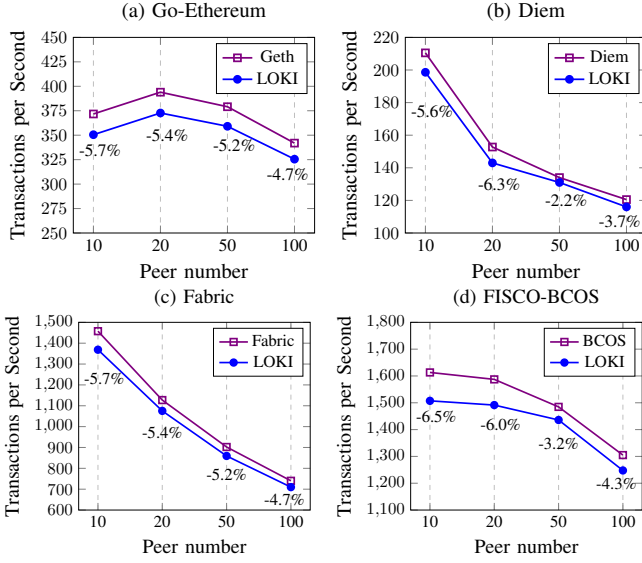
Fig. 22. TPS of the consensus protocols on Go-Ethereum, Diem, Fabric and FISCO-BCOS for LOKI and the original node. LOKI only decreases the TPS by 2.2% - 6.5% compared with the original nodes. The overhead of LOKI is similar under various node scales
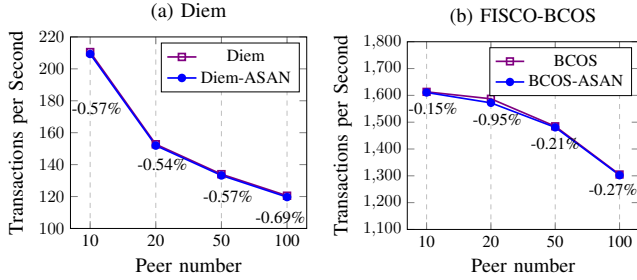

Fig. 23. TPS of the consensus protocols on Diem and FISCO-BCOS for the original nodes and nodes with ASAN. The result shows that TPS is only decreased by less than 1% with ASAN.

the official testchain which is used to test smart contracts like Ethereum testnet. Thus, the byzantine behaviors performed by LOKI will not affect the real online scenarios.

We also gave an experiment on how ASAN can affect the TPS of a blockchain system. Because Go is not supported with ASAN, we only evaluate Diem and FISCO-BCOS in this experiment. The results are shown at Figure 23.

As the figure indicates, a blockchain with ASAN nodes have only less than 1% overhead compared with the original chain. The reason may be that ASAN has little influence on block generation and confirmation. The main overhead caused by ASAN is memory-related operations. Though ASAN has also influenced the consensus process, the throughput is mainly affected by the network traffic and scale.

H. False Negatives of LOKI

To evaluate the accuracy of LOKI, we found 12 recent bugs from the 4 platforms. The bugs could be found in Table VI. LOKI can successfully reproduce 10 of them, while it cannot reproduce bug#3 and bug# because they are caused by data race conditions. This indicates that the false negative rate of LOKI is around 16.67%.

TABLE VI.   THE RECENT 12 BUGS FOR 4 PLATFORMS. LOKI CANNOT DETECT 2 OF THEM DUE TO DATA RACE.

#	Platform	Link
1	Go-Ethereum	https://github.com/ethereum/go-ethereum/issues/25953
2	Go-Ethereum	https://github.com/ethereum/go-ethereum/issues/25787
3	Go-Ethereum	https://github.com/ethereum/go-ethereum/issues/25868
4	Fabric	https://jira.hyperledger.org/projects/FAB/issues/FAB-18239
5	Fabric	https://jira.hyperledger.org/projects/FAB/issues/FAB-18535
6	Fabric	https://jira.hyperledger.org/projects/FAB/issues/FAB-14470
7	Diem	https://github.com/diem/diem/issues/8704
8	Diem	https://github.com/diem/diem/issues/8423
9	Diem	https://github.com/diem/diem/issues/7643
10	FISCO-BCOS	https://github.com/FISCO-BCOS/FISCO-BCOS/issues/2101
11	FISCO-BCOS	https://github.com/FISCO-BCOS/FISCO-BCOS/issues/2206
12	FISCO-BCOS	https://github.com/FISCO-BCOS/FISCO-BCOS/issues/2254