Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing

Youngseok Yang, *Seoul National University;* Taesoo Kim, *Georgia Institute of Technology;* Byung-Gon Chun, *Seoul National University and FriendliAI*

https://www.usenix.org/conference/osdi21/presentation/yang

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing

Youngseok Yang[1] Taesoo Kim[2] Byung-Gon Chun[1,3*]
[1]*Seoul National University* [2]*Georgia Institute of Technology* [3]*FriendliAI*

Abstract

Ethereum is the second-largest blockchain platform next to Bitcoin. In the Ethereum network, decentralized Ethereum clients reach consensus through transitioning to the same blockchain states according to the Ethereum specification. Consensus bugs are bugs that make Ethereum clients transition to incorrect blockchain states and fail to reach consensus with other clients. Consensus bugs are extremely rare but can be exploited for network split and theft, which cause reliability and security-critical issues in the Ethereum ecosystem.

We describe Fluffy, a multi-transaction differential fuzzer for finding consensus bugs in Ethereum. First, Fluffy mutates and executes multi-transaction test cases to find consensus bugs which cannot be found using existing fuzzers for Ethereum. Second, Fluffy uses multiple existing Ethereum clients that independently implement the specification as cross-referencing oracles. Compared to a state-of-the-art fuzzer, Fluffy improves the fuzzing throughput by 510$\times$ and the code coverage by 2.7$\times$ with various optimizations: in-process fuzzing, fuzzing harnesses for Ethereum clients, and semantic-aware mutation that reduces erroneous test cases.

Fluffy found two new consensus bugs in the most popular Geth Ethereum client which were exploitable on the live Ethereum mainnet. Four months after we reported the bugs to Geth developers, one of the bugs was triggered on the mainnet, and caused nodes using a stale version of Geth to *hard fork* the Ethereum blockchain. The blockchain community considers this hard fork the greatest challenge since the infamous 2016 DAO hack. We have made Fluffy publicly available at https://github.com/snuspl/fluffy to contribute to the security of Ethereum.

1 Introduction

> The case is perhaps Ethereum's greatest challenge since the 2016 DAO fork, and it raises questions about Ethereum's oft-touted decentralization and the effectiveness of its developer coordination going into Ethereum 2.0.
>
> — *Coindesk, November 12th, 2020* [11]

*Corresponding author.

An extremely rare consensus bug[1], which makes Ethereum [17, 54] clients transition to incorrect blockchain states and fail to reach consensus with other clients, was triggered on the Ethereum mainnet on November 11th, 2020 [2, 11, 43, 45]. This was one of the two consensus bugs we found and reported to Ethereum developers four months before that date. The bug caused Ethereum nodes using a stale version of Geth [19], the most popular Ethereum client, to hard fork the Ethereum blockchain. One of the affected nodes was Infura [15], the largest infrastructure service that allows decentralized applications (DApps) to connect to the Ethereum network without having to run their own Ethereum nodes.

Consequently, Infura went down, and with it some of most popular Ethereum applications such as Metamask, MakerDAO, Uniswap, and Compound went down [11]. Shortly after, cryptocurrency exchanges around the world including Binance, the largest exchange, halted the trading of ETH, the cryptocurrency of Ethereum [2, 11]. Infura and others quickly upgraded their Geth clients to fix the bug. Nevertheless, around 30 Ethereum blocks from block 11234873 on the forked chain were lost [23], which transferred approximately 8.6 million USD worth of ETH[2]. The blockchain community considers this hard fork the greatest challenge since the infamous DAO hack of 2016 [11, 13].

This paper describes how we found such extremely rare, high-impact consensus bugs in the heavily-tested Ethereum network through *fuzzing* [9, 30, 38], an automated software testing technique that randomly mutates inputs and tests the target program on the resulting data.

Finding new consensus bugs in actively used Ethereum clients is challenging, because consensus bugs are extremely rare. Attackers can exploit consensus bugs for network split and theft, which cause reliability (e.g., delaying transactions) and security-critical issues (e.g., stealing ETH) in the Ethereum ecosystem. To prevent such issues, Ethereum developers make preventing consensus bugs a top priority, and invest heavily in auditing, testing, and fuzzing Ethereum

[1]We focus on consensus bugs in state management that make Ethereum clients transition to incorrect blockchain states. Bugs in blockchain consensus algorithms such as proof of work are not the focus of this paper.

[2](Total amount of ETH transferred by block 11234873 to 11234902 on the canonical chain) $\times$ (Closing price of ETH/USD on November 10th)

	EVMLab	**EVM libFuzzer**	**EVM Fuzzer**	**Fluffy**
Transactions per test	Single	Single	Single	**Multiple**
Contract language	Bytecode	Bytecode	Solidity	**Bytecode**
In-process fuzzing	-	✓	-	✓
Coverage-guided	-	✓	-	✓
Open source	✓	-	✓	✓
Created in	2017	2017	2019	**2020**
Impact of newly found bugs (Count)	High (1), Low (2)	Low (2)	N/A	**High (2)**

Table 1: A comparison of Fluffy and existing fuzzers for finding consensus bugs in Ethereum.

clients [18, 21, 22, 33]. Since the Ethereum network launched in July 2014, only 13 consensus bugs have been found in the most popular Geth [19] and OpenEthereum [41] clients, and only 6 of them would have been exploitable on the live Ethereum mainnet [34].

Ethereum fuzzers have found most of the consensus bugs [18, 22, 33, 34]. These existing fuzzers focus on the blockchain state, which is a set of Ethereum accounts that hold a balance of ETH. Specifically, the fuzzers model an Ethereum client as a blockchain state model, in which the blockchain state is transitioned by an Ethereum transaction. As a result, in each fuzzing iteration, they generate and test a pre-transaction blockchain state and a single transaction that transitions the blockchain state. Table 1 compares the existing fuzzers. EVMLab [18] generates random Ethereum Virtual Machine (EVM) bytecode of Ethereum smart contracts [54], and invokes the contracts with a single transaction. EVMLab has found in the most popular Geth and OpenEthereum clients one high-impact bug that was exploitable on the mainnet and two low-impact bugs that were not exploitable on a live network. EVM libFuzzer [33] is a closed source fuzzer whose details are unknown. EVM libFuzzer integrates with libFuzzer [30] and has found two low-impact bugs. EVMFuzzer [22] is a more recent fuzzer that generates Solidity [20] code of contracts.

However, the blockchain state model of existing fuzzers falls short to cover the full search space for finding consensus bugs. The full search space consists of the set of possible client program states, which are the values of program variables of Ethereum clients that can be reached after executing Ethereum transactions. For each pre-transaction blockchain state (e.g., Account A has 0 ETH), the blockchain state model can cover only a single pre-transaction program state (e.g., account_a = { ETH: 0, deleted: false}). Consequently, existing fuzzers fail to test other possible pre-transaction program states (e.g., account_a = { ETH: 3, deleted: true}) that represent the same blockchain state. This leads existing fuzzers to miss consensus bugs which are triggered only when a transaction is applied to such other pre-transaction program states.

To fully cover the search space for finding consensus bugs, we propose to model an Ethereum client as a client program state model, in which the client program state is transitioned by a transaction. Based on this model, in each fuzzing iteration, we generate and execute a sequence of multiple transactions that transition an initial client program state. This allows us to indirectly generate various intermediate pre-transaction program states, which can be reached after executing transactions and can lead to the discovery of new consensus bugs.

We embody our approach in Fluffy, a multi-transaction differential fuzzer for finding consensus bugs in Ethereum. Fluffy mutates and executes multi-transaction test cases to find consensus bugs which cannot be found using existing fuzzers for Ethereum. In addition, Fluffy uses multiple existing Ethereum clients that independently implement the specification as cross-referencing oracles, similar in concept to N-versioning [5]. This technique is known as differential fuzzing [9, 36] in the testing community.

Our Fluffy design employs several new fuzzing techniques. First, we modify existing Ethereum clients to provide an execution model that enables efficient multi-transaction fuzzing. Second, we use multi-transaction test cases that encode dependencies between multiple Ethereum transactions and enable mutating transaction contexts (i.e., sequence of transactions that are executed prior to the transaction) rather than pre-transaction blockchain states. Third, we introduce new semantic-aware mutation strategies for mutating transaction contexts, transaction parameters, and EVM bytecode.

We have implemented Fluffy in Rust and Go, and made it publicly available at https://github.com/snuspl/fluffy. Our current implementation supports fuzzing Geth and OpenEthereum, which are used by 98% of nodes in the Ethereum mainnet, as of August 2020 [6]. Our implementation adopts various optimizations including in-process fuzzing and optimized fuzzing harnesses for Ethereum clients, and also provides a debugger to analyze crashes due to consensus bugs. Our evaluation on a 12-core machine shows that Fluffy finds 10 out of 11 real-world consensus bugs in Geth and OpenEthereum within just 12 hours of fuzzing. Fluffy also improves the fuzzing throughput by 510× and the code coverage by 2.7× compared to EVMLab.

2 Background

We first provide an overview of Ethereum [17, 54], and then describe consensus bugs in Ethereum.

2.1 Ethereum

The Ethereum blockchain network consists of decentralized peer-to-peer Ethereum clients that implement the Ethereum Virtual Machine (EVM) specification [54]. EVM is a Turing-complete machine that specifies how the Ethereum blockchain state, a set of Ethereum accounts, is altered through transactions recorded on Ethereum blocks. Accounts in the blockchain state have an address and hold a balance of ETH, the cryptocurrency of Ethereum. There are two types of accounts: the externally-owned account (EOA) owned by an Ethereum user, and the smart contract account which is owned by code and has a key-value storage. In addition to the accounts, EVM provides precompiled contracts that perform specialized operations at fixed addresses. Ethereum transactions are either message call transactions that can invoke smart contracts, or contract creation transactions that create new smart contracts.

Blockchain state. EVM transitions blockchain states through processing Ethereum bytecode instructions invoked by transactions. EVM provides around 140 distinct instructions. Each instruction makes specific changes to EVM internal states such as the EVM stack and the EVM memory, as well as the blockchain state such as the ETH balance of EOAs and the storage of smart contracts. Each instruction also has a specific gas cost, a fee that the transaction sender pays to compensate for the computational effort that it takes to execute the instruction in the network. EVM throws an out-of-gas error when the sum of the gas cost exceeds the gas limit of a transaction.

Client program state. Ethereum clients implement EVM with a specific programming language such as Go and Rust [19, 41]. As a result, a client maintains its own client program state, which are the values of program variables (e.g., Rust or Go variables) that are reached after executing Ethereum transactions. There can be multiple different client program states that represent the same blockchain state, since the client determines the blockchain state it is in by interpreting the values of a subset of its program variables.

Example. Figure 1 shows how an Ethereum client executes a sequence of two transactions to transition an initial blockchain state (State 0). Transaction 1 first creates contract C by invoking the bytecode in its data field, which returns (`RETURN`) the bytecode that becomes the code of contract C (State 1). Transaction 2 then invokes the code of C with specific value and data parameters, such that a particular key-value pair is stored (`SSTORE`) in the storage of C (State 2).

Suppose we initialize an Ethereum client implementation in two different ways. First, we initialize the client directly with the blockchain state after Transaction 1 (State 1). Second, we initialize the client with the initial blockchain state (State 0), and then execute Transaction 1 to transition the blockchain state. Although the resulting blockchain state (State 1) is the same for the two clients, the resulting client program state may not be the same depending on the client implementation. As a result, the two clients can behave differently when executing the subsequent Transaction 2.

We show why it is important to fully test such different behaviors when testing whether clients transition to incorrect blockchain states, which we describe next.

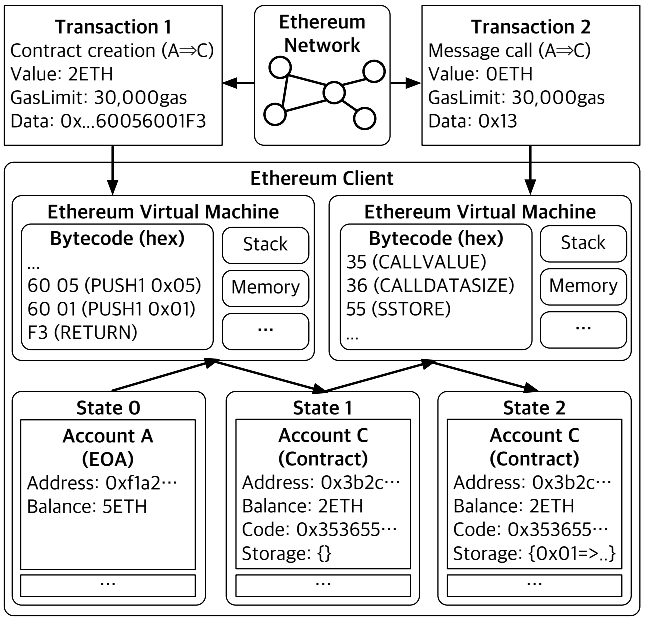


Figure 1: Multiple Ethereum transactions interact to determine the transitions of Ethereum blockchain states.

2.2 Consensus Bugs

Consensus bugs are implementation bugs that make Ethereum clients transition to incorrect blockchain states, and fail to reach consensus with other clients that transition to correct states according to the EVM specification [54]. The Ethereum community has fostered the development of diverse client implementations with a goal to verify the EVM specification and make the Ethereum network more secure. Nevertheless, only two Ethereum clients are used by 98% of nodes participating in the Ethereum mainnet, as of August 2020 [6]. Around 80% of nodes use Geth [19] written in Go, and 18% use OpenEthereum [41], previously called Parity, written in Rust. Therefore, consensus bugs that affect Geth and OpenEthereum have the most critical impacts on the Ethereum ecosystem.

Known bugs. Consensus bugs in actively used Ethereum clients are extremely rare. Table 2 shows the list of consensus bugs [34] reported to have been found in Geth, and OpenEthereum under the former name of Parity. Since the Ethereum network launched in July 2014 until 2019, only 13 consensus bugs were found. Only 6 of the 13 bugs are high-impact bugs that would have been exploitable on the live Ethereum mainnet. In addition to these 6 high-impact bugs, we were able to find 2 new high-impact consensus bugs in 2020, which we describe in detail later in the paper.

Consensus bugs are extremely rare for the following reasons. First, the Ethereum community makes preventing consensus bugs a top priority, and heavily invest in auditing, testing, and fuzzing Ethereum clients [18, 21, 22, 33]. Second, Ethereum clients are continuously being tested on the live

#	Client	Date	Consensus bug description	Tx	Impact	Finding method	Fluffy (Time)
1	Geth	Aug 2020	The balance of a deleted account is carried over to a new account	2	High	**Fluffy**	✓ (291m)
2	Geth	Jul 2020	The DataCopy precompile performs shallow rather than deep copy	1	High	**Fluffy**	✓ (386m)
3	Geth	Mar 2019	Block timestamps exceeding uint64 lead to a wrong block hash	1	High	Unknown	N/A
4	Parity	Oct 2018	The SSTORE gas refund counter does not go below zero when it should	1	Medium	Triggered-Testnet	✓ (57m)*
5	Parity	Jun 2018	Unsigned transactions are accepted and treated as valid	1	Medium	Triggered-Testnet	N/A
6	Geth	Feb 2018	Subgroups in elliptic curve pairings are not validated properly	1	High	Unknown	N/A
7	Parity	Oct 2017	CREATE in static context without enough balance throws a wrong error	1	High	EVMLab	✓ (41m)*
8	Geth	Oct 2017	CALL in static context with less than three stack elements crashes	1	Low	EVM libFuzzer	✓ (38m)
9	Parity	Oct 2017	The gas for the ModExp precompile overflows for certain inputs	1	Low	Manual auditing	Timeout-12h
10	Parity	Oct 2017	RETURNDATACOPY overflows during addition of offset and length	1	Low	EVM libFuzzer	✓ (14m)*
11	Parity	Oct 2017	The gas for the ModExp precompile overflows for large numbers	1	Low	EVMLab	✓ (15m)
12	Parity	Oct 2017	RETURNDATASIZE from a precompile returns a non-zero size	1	Low	EVMLab	✓ (2m)
13	Geth	Feb 2017	The EVM stack underflows for SWAP, DUP, and BALANCE	1	High	Unknown	✓ (6s)
14	Geth	Jan 2017	Undisclosed	-	High	Unknown	N/A
15	Geth	Nov 2016	Fails to revert the deletion of touched accounts on out of gas	1	High	Triggered-Mainnet	✓ (5m)

Table 2: The list of consensus bugs [34] found in Geth and Parity, which is the former name of OpenEthereum. The first two bugs are new bugs found by Fluffy. The number of transactions (Tx) indicates the minimum number of depending transactions required to trigger the bug. High-impact and medium-impact bugs are bugs that would have been exploitable on the live Ethereum mainnet and testnet respectively. Low-impact bugs are bugs that were fixed before they became exploitable on a live Ethereum network. The last column shows the time it takes Fluffy to find the bugs, which is explained in § 7.1.

mainnet for consensus bugs with real-world transactions. On the mainnet, multiple versions of multiple Ethereum clients have reached consensus on a total of more than 800 million transactions as of August 2020.

Ethereum fuzzers. All consensus bugs have been found with fuzzing [9,36], except for the bugs triggered on a live network, found with manual auditing, or whose finding method is unknown. Existing Ethereum fuzzers [18,22,33] focus on the blockchain state. Specifically, the fuzzers model an Ethereum client as a blockchain state model, in which the blockchain state is transitioned by an Ethereum transaction. As a result, in each fuzzing iteration, they generate and test a pre-transaction blockchain state and a single transaction that transitions the blockchain state.

However, the blockchain state model of existing fuzzers falls short to cover the full search space for finding consensus bugs. The full search space consists of the set of possible client program states that can be reached after executing Ethereum transactions. For each pre-transaction blockchain state (e.g., Account A has 0 ETH), the blockchain state model can cover only a single pre-transaction program state (e.g., account_a = { ETH: 0, deleted: false}). Consequently, existing fuzzers fail to test other possible pre-transaction program states (e.g., account_a = { ETH: 3, deleted: true}) that represent the same blockchain state. This leads existing fuzzers to miss consensus bugs which are triggered only when a transaction is applied to such other pre-transaction program states.

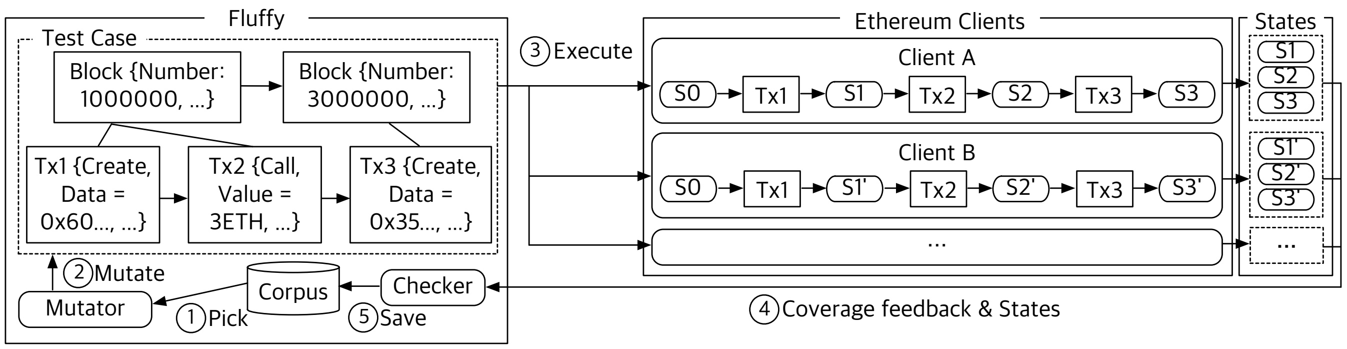
Bug impacts. Attackers can exploit consensus bugs for network split and theft. First, an attacker can trigger a consensus bug to make buggy client nodes create a fork of the blockchain, which only they agree on. The transactions recorded on the forked chain are eventually nullified, when the consensus bug is fixed and the fork is abandoned [23,51]. Second, an attacker can steal ETH from certain vulnerable smart contracts on buggy client nodes. Even if the business logic of a smart contract is perfectly secure, a consensus bug can alter how the buggy client executes the contract and allow the theft of ETH.

Attackers have strong incentives to find and exploit consensus bugs. Attackers can short ETH on cryptocurrency exchanges after triggering a consensus bug, with the expectation that the price of ETH will fall, when investors learn about the attack and lose trust in Ethereum. Attackers also can trade their ETH with an off-chain item (e.g., other cryptocurrency such as Bitcoin [39]) on the forked chain after triggering the bug, such that the traded ETH is given back to them when the bug is fixed and the forked chain is abandoned. Finally, attackers can steal ETH from vulnerable smart contracts.

3 Overview

We describe Fluffy, a multi-transaction differential fuzzer for finding consensus bugs in Ethereum. Unlike existing fuzzers which use the blockchain state model, Fluffy fully covers the search space for finding consensus bugs, by modelling an Ethereum client as a client program state model. Using this client program state model where the client program state is transitioned by a transaction, Fluffy tests a sequence of multiple transactions in each iteration. In addition, Fluffy uses different Ethereum clients as cross-referencing oracles.

Figure 2 illustrates an overview of Fluffy. First, Fluffy



Fluffy
Test Case
Block {Number: 1000000, ...}
Block {Number: 3000000, ...}
Tx1 {Create, Data = 0x60..., ...}
Tx2 {Call, Value = 3ETH, ...}
Tx3 {Create, Data = 0x35..., ...}
2 Mutate
Mutator
1 Pick
Corpus
Checker
5 Save
3 Execute
Ethereum Clients
Client A
S0
Tx1
S1
Tx2
S2
Tx3
S3
Client B
S1'
S2'
S3'
...
States
4 Coverage feedback & States

Figure 2: An overview of Fluffy. Fluffy first selects a test case from the corpus (①). Next, Fluffy mutates transactions in the test case using a semantic-aware mutation strategy (②). Fluffy then executes the new test case on multiple Ethereum clients (③). The clients transition the initial blockchain state to new states, while executing the transactions in the test case. When the execution completes, Fluffy collects the new states and coverage feedback (④). Fluffy saves the new test case if new code paths are discovered (⑤). Fluffy proceeds to the next iteration if the clients transitioned to the same states, and crashes otherwise.

selects a test case from the corpus of previously executed test cases (①). Each test case contains multiple transactions, and information about dependencies between the transactions. Fluffy then generates a new test case by mutating the transactions in the selected test case using a semantic-aware mutation strategy (②). Fluffy then executes the new test case on multiple Ethereum clients (③). The clients transition the initial blockchain state (S0) to new states (Client A: (S1, S2, S3), Client B: (S1′, S2′, S3′)), while executing the transactions (Tx1, Tx2, Tx3) in the test case. When the execution completes, Fluffy collects the new blockchain states and code coverage feedback from the clients (④). Fluffy saves the new test case in the corpus if new code paths are discovered (⑤). Fluffy cross-checks the blockchain states collected from different clients, and crashes if the clients transitioned to a different state during execution (S1 != S1′ || S2 != S2′ || S3 != S3′). Otherwise, Fluffy proceeds to the next fuzzing iteration.

4 Design

We describe the design of Fluffy, focusing on the execution model, the test case, and the mutation algorithm.

4.1 Execution Model

We modify existing Ethereum clients to provide an execution model that enables efficient multi-transaction fuzzing.

Genesis account. We modify clients to use an initial blockchain state that contains the genesis account, which we define as an EOA that has a large balance of ETH. The genesis account serves as the starting point for creating new smart contracts and invoking the code of the contracts (i.e., we set the sender of transactions to the genesis account).

Activated addresses. We modify clients to convert 20-byte Ethereum addresses to activated addresses, which we define as an address either owned by a precompiled contract, or owned by a contract created in previous EVM execution. The rationale is that it is extremely inefficient to explore the 20-byte Ethereum address space, especially given that we rewrite blockchain history and almost all of the addresses are not used by a contract. Moreover, we cannot know in advance which addresses will be activated and used by smart contracts in the future, since new contracts can be created dynamically during the execution of EVM bytecode (`CREATE`).

4.2 Test Case

We use test cases that are tailored to multi-transaction fuzzing. Figure 3 shows the data structure of test cases used in Fluffy. We execute test cases on Ethereum clients through iterating over the blocks and the transactions of each block in the list order, and applying each transaction to the blockchain state. Our design has several unique characteristics that enable efficient multi-transaction fuzzing.

First, we enable mutating the context of transactions, which we define as the ordered sequence of transactions that are executed prior to the transaction. Our approach is in contrast to existing approaches [18, 22, 33] that directly generate a pre-transaction blockchain state and test a single transaction. The blockchain state mutation strategy of existing approaches are limited to testing only a single pre-transaction client program state for each pre-transaction blockchain state. In contrast, our approach is able to generate and test various pre-transaction client program states (e.g., (account_a = { ETH: 0, deleted: false}), (account_a = { ETH: 3, deleted: true})) for each pre-transaction blockchain state (e.g., Account A has 0 ETH). This is because the values of program variables of Ethereum clients can change in various ways depending on which sequence of transactions are executed. This lets us find bugs like the transfer-after-destruct bug (§ 6.2), which requires testing particular pre-transaction client program states that the

```
class FluffyTestCase:
 Block[] blocks

class Block:
 Transaction[] transactions
 int versionNumber   // hard-fork upgrades
 int timestamp       // between prev/next block
 // Constants: author, gasLimit, ...

class Transaction:
 int gasLimit        // minimum to threshold
 int value           // 0, 1, or random
 byte[] data        // bytes
 // Constants: signature, gasPrice, ...

class CreateContract extends Transaction:
 byte[] constructor  // invoked bytecode
 byte[] codeToReturn // code of new contract

class MessageCall extends Transaction:
 int receiver        // activated address
```
Figure 3: The data structure of test cases used in Fluffy.

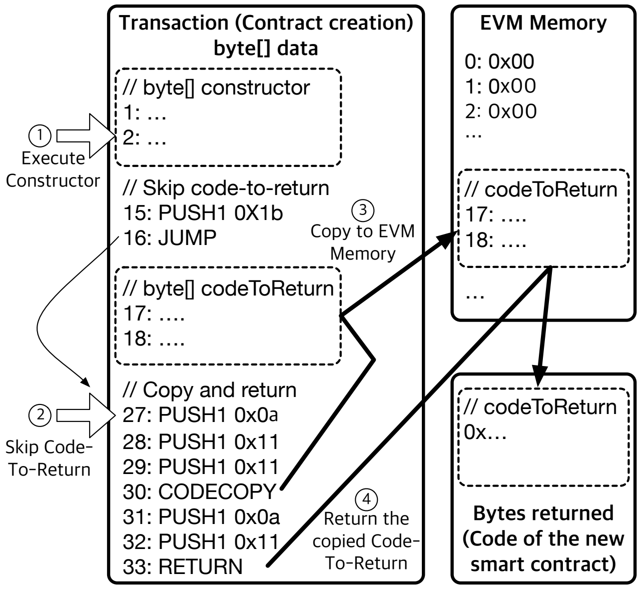


Figure 4: Fluffy mutates the data field of contract creation transaction by mutating the constructor field and the code-to-return field that become part of the data field along with a number of injected bytecode instructions. When the transactions are executed, the constructor is executed and the code-to-return is returned.

blockchain state mutation strategy is unable to generate.

Second, we introduce the constructor and the code-to-return fields for contract creation transactions. These fields, along with a number of injected instructions which we describe later, become part of the data field. Our approach enables directly mutating the code of the newly created smart contracts, which is set to the code-to-return when the transaction completes. Existing Ethereum fuzzers [18, 22, 33] do not consider this approach, since they execute a single transaction per fuzzing iteration and do not invoke the code of smart contracts created by transactions.

Third, we limit the possible values of transactions and block parameters to reduce wasting CPU cycles in meaningless mutations and executions. We use constant values for parameters that have limited effects on how clients execute transactions. Our approach reduces the overhead of mutating and executing multiple transactions.

4.3 Mutation

We use three mutation strategies to mutate test cases: context mutation, bytecode mutation, and parameter mutation. Our bytecode and context mutation strategies have not been employed in existing Ethereum fuzzers [18, 22, 33]. Minor parts of the parameter mutation strategy, such as setting the timestamp of a block within a certain range, share some similarities with existing fuzzers.

Context mutation. We randomly mutate the list of blocks and the list of transactions to mutate transaction contexts. We use four strategies: add, delete, clone, copy. We add a new block or a new transaction to the list, or delete an existing one. We also clone an existing block or a transaction, or copy its contents to another block or transaction.

Bytecode mutation. We mutate the constructor and the code-to-return fields of contract creation transactions to mutate bytecode. Specifically, we randomly add, delete, mutate, and copy bytecode instructions in the fields. Among the various EVM instructions, we do not add the `PUSH` instructions (`PUSH1`-`PUSH32`), which make EVM push a number of following bytes (1B-32B) in the fields onto the EVM stack, rather than interpreting the bytes as bytecode instructions and executing them. Our approach enables preserving the semantics of bytecode instructions that are not directly modified by Fluffy across mutations.

We then update the data field using the mutated constructor and code-to-return, as illustrated by Figure 4. We concatenate the constructor, instructions to skip the execution of the code-to-return (`JUMP`), the code-to-return, instructions to copy the code-to-return to the EVM memory (`CODECOPY`), and instructions to return the copied bytecode (`RETURN`). When the transaction is executed and the bytecode of the data field is invoked, the constructor is executed and the code-to-return is returned.

In contrast, it is difficult to generate smart contract constructors that return appropriate bytecode, if we treat the data field as a single sequence of instructions and mutate the data field as a whole. The reason is that the generated constructor is likely to prematurely terminate before invoking `RETURN` to return bytecode. Moreover, appropriate bytes should be stored in the right region of the EVM memory before invoking `RETURN`, and a small mutation is likely to completely alter the stored bytes.

Nevertheless, we do note that in Fluffy the code of a smart contract is not always equal to the code-to-return field of the transaction that creates the contract. This is because errors such as EVM stack underflow can still occur during the execution of the constructor field of the transaction, and prevent the following injected instructions to copy and return the code-to-return.

Parameter mutation. We mutate transaction parameters as follows. We simply set the transaction receiver address to a random integer, assuming that the target clients convert it to an activated address. We set the gas limit to the sum of the minimum gas required for EVM to not reject the transaction before invoking any bytecode, and a randomly generated number in the range between 0 to a threshold to avoid long sequences of meaningless instructions such as an infinite while loop. For example, we can set the threshold to 1.6 million gas that allows executing the `CREATE` instruction 50 times, which is the most expensive instruction that costs 3.2 thousand gas. 1.6 million gas costs only around 0.8 ETH on the Ethereum mainnet as of August 2020. In case of value, which determines the amount of ETH transferred by the transaction, we randomly choose 0, 1, or a random integer.

We also randomly mutate the parameters of blocks. Most notably, we mutate the block version number, which determines the version of EVM that executes the transaction. Since Ethereum launched in 2014, there has been around 10 non-backward compatible EVM hard-fork upgrades that came into effect at particular block version numbers. We use the version numbers that mark the start of a new EVM hard-fork upgrade, rather than covering all of the block version numbers used in mainnet, which are more than 10 million as of August 2020.

5 Implementation

We implement Fluffy on top of libFuzzer [30] using Rust and Go. We adopt the basic infrastructure of libFuzzer, including the code coverage bitmap and test case scheduling, but introduce several key components. We replace the default mutator of libFuzzer with our multi-transaction mutator. We also introduce fuzzing harnesses for OpenEthereum and Geth to enable in-process fuzzing and several other optimizations. Finally, we implement a crash debugger for analyzing crashes due to consensus bugs. The rest of the section describes notable implementation details.

5.1 Fuzzing Harnesses

We implement fuzzing harnesses as long-running processes that integrate with the transaction processing components of Ethereum clients.

In-process fuzzing. We reuse fuzzing harnesses across fuzzing iterations to avoid having to spawn new Ethereum client processes for every new test. We link the mutator with the OpenEthereum harness to mutate test cases and collect code coverage statistics in the same process, and run the Geth harness in a separate process. We use Linux FIFO files for exchanging test cases and execution results between the two processes.

Initial blockchain state. In addition to the genesis account, we add to the initial blockchain state accounts with a balance of 1 Wei (1×10^{-18} ETH) under the addresses of precompiled contracts. These non-zero balance accounts let us avoid triggering a false positive consensus bug in Geth related to a bug that was previously exploited in the live Ethereum mainnet (Bug #15 in Table 2) [51].

Activated addresses. We maintain a list of activated addresses in the harnesses. While executing transactions, we add addresses of newly created contracts to the list and convert Ethereum addresses to activated addresses (i.e., activatedList[bigInteger(address) % activatedList.length()]). To test deleted addresses, we do not remove addresses from the list when contracts invoke `SELFDESTRUCT` and destroy themselves.

Transaction verification. Ethereum clients use the secp256k1 ECDSA algorithm to verify the signature of transactions [54]. This requires signing and verifying each of the many transactions that Fluffy generates, which is costly considering that most of EVM bytecode instructions consume few CPU cycles. We skip these procedures in our harnesses, since we do not focus on signature verification.

Jump destinations. EVM throws an error if the destination of `JUMP` and `JUMPI` is not `JUMPDEST`, which marks a valid jump target. This increases the chance of Fluffy terminating prematurely due to an error when testing loops and conditional branches. To address this issue, we disable the checking of `JUMPDEST` and allow jumping to non-`JUMPDEST` instructions in our harnesses.

Number of transactions. Fluffy uses its mutator and the test case scheduler to determine the number of transactions it should use per test case. Fluffy randomly generates test cases with few transactions to build the initial corpus. If new code paths are not easily discovered with few transactions, Fluffy gradually generates test cases with more transactions through the transaction context mutations, adding the new test cases to the corpus if they discover new code paths. Fluffy provides an option to configure a libFuzzer parameter (-len_control), which determines whether to prefer to generate small test cases over large test cases. Fluffy also provides an option to set a hard limit on the number of transactions in a fuzzing iteration. We note that the search space of Fluffy, which is the Ethereum client program state model, is constant regardless of the number of transactions that Fluffy executes for each test case.

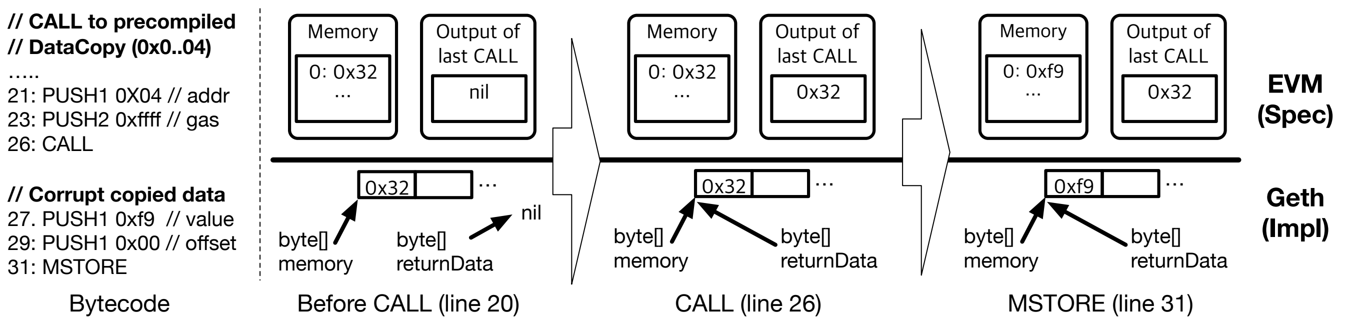


Figure 5: A minimal test case for the shallow copy bug in Geth. An attacker can exploit this bug to corrupt data copied through the precompiled DataCopy contract, making Geth deviate from the EVM specification.

5.2 Crash Debugger

The crash debugger enables analyzing crashes due to consensus bugs.

Analyzing the root cause. We find the first states that the clients output differently while processing the test case, which we call triggering states. We also find the last state that can reach the triggering states when used as the starting point of EVM execution, which we call the starting state. We then find which EVM bytecode instruction invoked during the execution of transactions between the starting state and the triggering states cause different behaviors in Ethereum clients. Finally, we use tools like Delve [14] on corresponding code in Ethereum clients to analyze the root cause.

Validating exploitability. We compare the latest blockchain state in the Ethereum mainnet with the starting state of the bug. We check whether an Ethereum user can transition the latest blockchain state to a new state that includes the accounts in the starting state. We also check whether the transactions between the starting state and the triggering states are processed by the latest version of EVM used in the mainnet. Finally, we convert the transactions into new transactions that can reproduce the bug on vanilla Ethereum clients. In particular, we convert active addresses to Ethereum addresses through examining EVM traces [19,41], and insert `JUMPDEST`s where appropriate.

6 New Consensus Bugs

Fluffy found two new consensus bugs in Geth which were exploitable on the live Ethereum mainnet: shallow copy bug and transfer-after-destruct bug. Existing ethereum fuzzers [18, 22, 33] that test only a single transaction per iteration are not able to find the transfer-after-destruct bug, because finding it requires testing particular pre-transaction client program states which the fuzzers are unable to generate. Although the shallow copy bug can be found by testing a single transaction, existing fuzzers failed to reach deep states of Ethereum clients and failed to find the bug during the time from when the bug became exploitable in the live Ethereum mainnet in November 2019 (Geth v1.9.7 release) to when we found and reported the bug in July 2020 [43]. In this section we describe the bugs using minimal test cases, and discuss the impact of the bugs. We also explain how the bugs were reported, fixed, and triggered, with a focus on vulnerability disclosure issues that occurred.

6.1 Shallow Copy Bug

The root cause of this bug is that the implementation of the precompiled DataCopy contract (address: 0x0..04) in Geth performs a shallow copy upon invocation, although the contract should perform a deep copy according to the EVM specification.

Figure 5 shows a minimal test case that triggers the bug. Suppose that a message call transaction is issued to a contract account that contains bytecode instructions shown in the figure. The figure shows the inner workings of the EVM specification (top) as well as the Geth implementation (bottom) when processing the bytecode of the contract invoked by the transaction. Geth implements the EVM memory and the output of the last `CALL` to external contracts with `byte[]memory` and `byte[]returnData` respectively, which are pointers to a byte buffer.

The following steps trigger the bug. Between line 1 to line 20, the contract stores a byte 0x32 in the EVM memory at offset 0. Geth carries out the execution by storing the byte 0x32 in the byte buffer that `byte[]memory` points to. At this point `byte[]returnData` is nil, since no `CALL` has been made from the contract yet.

Next, the contract `CALL`s the DataCopy contract at address 0x0..04, passing in the 1-byte data at the EVM memory offset 0 as the argument. This leads to the execution of the DataCopy implementation in Geth, which is a single line of code that simply returns the `byte[]` pointer that is given to it. In this case the pointer points to the 1-byte data in the byte buffer that `byte[]memory` is pointing to. Geth then sets `byte[]returnData` to this pointer.

The contract corrupts the copied data by simply storing

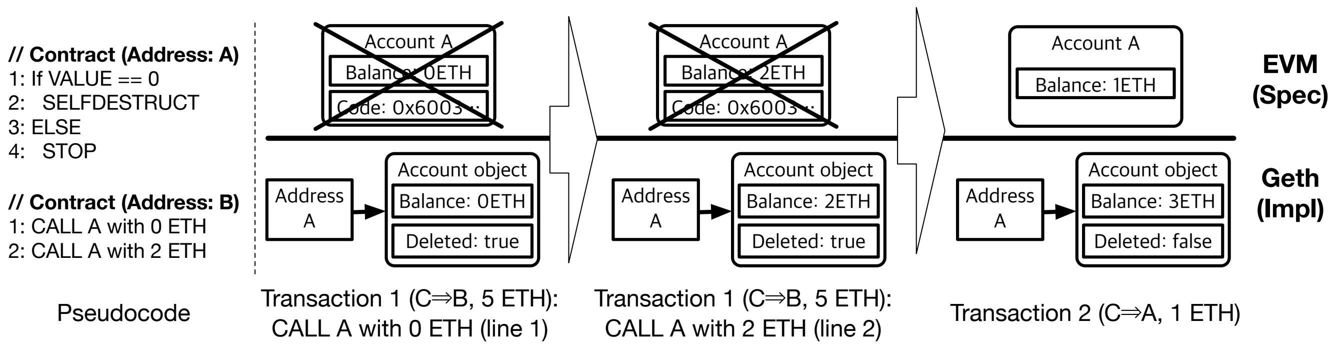


Figure 6: A minimal test case for the transfer-after-destruct bug in Geth. Transaction 1 invokes B, which leads to two `CALL`s to A. Transaction 2 invokes A. An attacker can exploit this bug to carry over the balance of a deleted account to a new account under the same address, making Geth deviate from the EVM specification.

new data, 0xf9, in the EVM memory at offset 0 (`MSTORE`). According to the specification, storing data in the EVM memory should never affect the data copied through DataCopy. However, in Geth, the contents of `byte[]returnData` changes from 0x32 to 0xf9.

Now, the contract can corrupt the blockchain state, for example through a sequence of bytecode instructions that stores the corrupted data in the storage of the contract (`RETURNDATACOPY`, `MLOAD`, `SSTORE`). After the transaction, Geth expects that 0xf9 is stored in the storage, whereas OpenEthereum and all other clients that comply with the specification expect 0x32.

Although this shallow copy bug is conceptually simple to understand, it is difficult to detect through code reviews, and tools such as static checkers which the Ethereum developers are actively using. The reason is that in the actual Go code of Geth it is not straightforward to track how pointers move across multiple files, classes, and functions.

Impact. When exploited, this bug can trigger a network split where Geth client nodes create a fork of the blockchain that stores 0xf9 in the storage. Furthermore, this bug can be exploited for smart contract theft. For example, suppose a contract invokes DataCopy by passing in an Ethereum address, overwrites the address with user input, and then transfers ETH to the copied address returned from DataCopy. Attackers can withdraw ETH from this contract to their account by specifying the address of their account as the user input, while others believe that the result of DataCopy should always be equal to the original address. Existing techniques for finding smart contract vulnerabilities [26, 32, 40, 49, 50, 57] are not capable of finding such vulnerability because they assume that the underlying Ethereum clients faithfully carry out the semantics of EVM bytecode instructions and precompiled contracts.

6.2 Transfer-After-Destruct Bug

The root cause of this bug is that Geth carries over the balance of a deleted account object to the newly created account object under the same Ethereum address, although it should not according to the EVM specification.

Figure 6 shows a minimal test case that triggers this bug. The initial blockchain state consists of two contracts. If the value of the transaction that invokes the contract is 0 ETH, the contract under address A destroys itself by invoking `SELFDESTRUCT`. If not, contract A simply terminates execution with `STOP`. The contract under address B issues two `CALL`s to A. To trigger the bug, we send a transaction to B, and then a transaction to A. The figure illustrates how the EVM specification (top) and the Geth implementation (bottom) handle the two transactions.

The first transaction (Transaction 1) is a message call transaction sent to B. The code of contract B is then executed as follows. First, we `CALL` A with 0 ETH (Contract B, line 1), which results in destroying contract A with `SELFDESTRUCT`. Geth carries out `SELFDESTRUCT` by marking the account object under address A as deleted, rather than destroying the account object as a whole. Geth also sets the balance of the account object to 0 ETH. Next, we `CALL` A with 2 ETH (Contract B, line 2). This makes Geth look up the account object under address A, and add 2 ETH to the balance of the object.

When the first transaction finishes, Geth transitions to a blockchain state where the account under address A is nil, through recognizing that the account object is marked as deleted and ignoring the balance of 2 ETH. This lets Geth comply with the EVM specification, which specifies that all information associated with the addresses of `SELFDESTRUCT`ed accounts should become nil after a transaction is processed [54].

However, Geth fails to comply with the EVM specification when processing the second transaction (Transaction 2). The second transaction is a message call transaction sent to A with 1 ETH. According to the specification, the balance of the account under address A should become 1 ETH after this transaction, since the balance of the account was nil, and thus was 0 ETH before the transaction. However, Geth mistakenly thinks that the account has 3 ETH after the transaction. When processing the transaction, Geth does recognize that the account object under address A is marked as deleted. Geth thus attempts to replace the old object with a new account object, but mistakenly carries over the balance of the old object to that of the new object during the process. This results in adding to the balance of the new account object 2 ETH from the old object, as well as 1 ETH from the transaction.

Impact. Similar to the shallow copy bug, this bug can be exploited for network split and theft. Moreover, this bug makes the total supply of ETH in circulation inconsistent between Geth and other Ethereum clients, which adds to the argument that the total supply of Ethereum is impossible to calculate [1, 53].

6.3 Responsible Vulnerability Disclosure

Responsible vulnerability disclosure in cryptocurrencies is hard because decentralized systems give no single party authority to push code updates [7]. To ensure that Ethereum mainnet nodes update securely, we privately reported the bugs to the Geth developers through the Ethereum bug bounty program [21]. Geth developers confirmed that the bugs are exploitable on the live Ethereum mainnet, and silently fixed the bugs in new versions of Geth to reduce the risk of an attacker exploiting the bugs. Ethereum mainnet nodes upgraded organically over time, thereby fixing the bugs.

Unfortunately, not all mainnet nodes upgraded, and this caused nodes using Geth v1.9.7 to v1.9.16 to hard fork the Ethereum block chain when the shallow copy bug was triggered four months later, on November 11th, 2020 [2, 11, 15, 43, 45]. Affected Ethereum infrastructure services and decentralized applications (DApps) went down, and cryptocurrency exchanges halted the trading of ETH. Around 30 Ethereum blocks from block 11234873 on the forked chain were lost [23], which transferred approximately 8.6 million USD worth of ETH. The blockchain community considers this hard fork the greatest challenge since the infamous DAO hack of 2016 [11, 13].

The hard fork sparked an active discussion on vulnerability disclosure protocols [11, 43, 52]. As a result, the Geth team created a public transparency policy for disclosing bugs [19]. The Geth team also revealed security advisories, including an advisory on the shallow copy bug (CVE-2020-26241) [35].

7 Evaluation

We evaluate Fluffy to answer the following questions.

- Does Fluffy effectively find real-world consensus bugs in Ethereum? (§ 7.1)
- Does Fluffy cover deep code paths that lead to consensus bugs in Ethereum clients? (§ 7.2)
- Does Fluffy efficiently test many instances of multiple transactions that rewrite blockchain history? (§ 7.3)
- Does Fluffy enable analyzing crashes triggered by consensus bugs? (§ 7.4)

We evaluate Fluffy on Intel(R) Xeon(R) CPU E5-2680 v3 (12 cores) with 128 GB memory. We compare Fluffy with EVMLab [18], which is an open-source, state-of-the-art differential fuzzer that is maintained by Ethereum developers. EVMLab is also the only existing fuzzer that found a high-impact consensus bug that would have been exploitable on the live Ethereum mainnet [34].

Unless noted otherwise, we configure the fuzzers as follows. For Fluffy, we configure libFuzzer parameters to run 24 parallel fuzzing instances (-fork=24), and prefer generating and executing small inputs rather than large inputs (-len_control=100). For EVMLab, we run 24 instances of the fuzzer in parallel. We run the fuzzers without any seed corpus for each experiment.

We use OpenEthereum v3.0.0 and Geth v1.9.14 as the target programs. EVMLab uses the vanilla version of the Ethereum clients. Fluffy uses the modified version that also fixes the two new bugs Fluffy found, since without the bug fixes Fluffy crashes due to the bugs during experiments.

7.1 Bug Finding Capability

We measure the time it takes for Fluffy to find the consensus bugs that occurred in Geth and OpenEthereum including the two new bugs Fluffy found, which are listed in Table 2. For each bug, we port the bug to OpenEthereum v3.0.0 or Geth v1.9.14, run Fluffy for 12 hours, and check if Fluffy finds the bug. We do not experiment with Bug #3 and Bug #5 which are associated with block mining and signature verification that Fluffy and existing fuzzers for Ethereum [18, 22, 33] do not focus on, Bug #6 which was fixed by switching to a different external library, and Bug #14 whose details are undisclosed [34].

Table 2 presents the result of the experiment. Fluffy finds 10 out of 11 real-world consensus bugs in Geth and OpenEthereum within just 12 hours of fuzzing. Among the 10 bugs, Fluffy finds Bug #7 and Bug #10 with a configuration that bounds the number of transactions and the length of the data of transactions, and finds Bug #4 with an earlier implementation of the multi-transaction mutator. The only bug Fluffy fails to find within 12 hours is Bug #9, which was originally found with manual auditing.

Result. Fluffy finds 10 out of 11 real-world consensus bugs in Geth and OpenEthereum within just 12 hours of fuzzing. The result shows that Fluffy is able to effectively find consensus bugs in Ethereum.

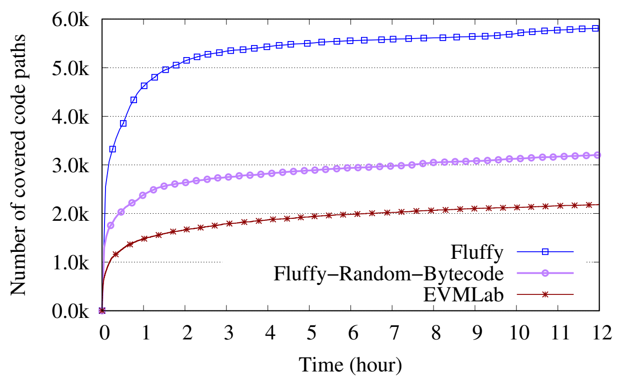


Figure 7: The number of code paths covered by Fluffy, a modified version of Fluffy called Fluffy-Random-Bytecode, and EVMLab over time.

7.2 Code Coverage

We measure the number of code paths covered by Fluffy, a modified version of Fluffy called Fluffy-Random-Bytecode, and EVMLab. Fluffy-Random-Bytecode simply generates random bytecode instructions rather than using the sophisticated bytecode mutation strategy (§ 4.3) of Fluffy. In case of Fluffy and Fluffy-Random-Bytecode, we measure the size of the corpus, which represents the number of code paths, over time. In case of EVMLab, which does not use and report code coverage, we replay the corpus it generates using libFuzzer on OpenEthereum.

Figure 7 shows the covered code paths over time. The result shows that Fluffy covers more code paths than EVMLab at all times. In the first 1 hour, Fluffy quickly covers more than 4,000 code paths, whereas EVMLab covers less than 2,000 paths. After 12 hours, Fluffy covers 5,809 code paths, which is 2.7 times as many code paths as 2,185 code paths that EVMLab covers.

Fluffy-Random-Bytecode performs better than EVMLab, but worse than Fluffy at all times. After 12 hours, Fluffy-Random-Bytecode covers 3,202 code paths, which is close to half of the paths covered by Fluffy and 1.5 times as many code paths as those covered by EVMLab. This show that the bytecode mutation strategy of Fluffy contributes significantly to the effectiveness of Fluffy.

Result. Fluffy explores 2.7 times as many code paths as EVMLab. The result shows that Fluffy is able to cover deep code paths in Ethereum clients that lead to consensus bugs.

7.3 Throughput

We evaluate whether Fluffy efficiently tests many instances of multiple transactions. For this evaluation we measure metrics such as the number of processed transactions, the number of executed fuzzing iterations, and the CPU usage.

Figure 8 shows the total number of Ethereum transactions

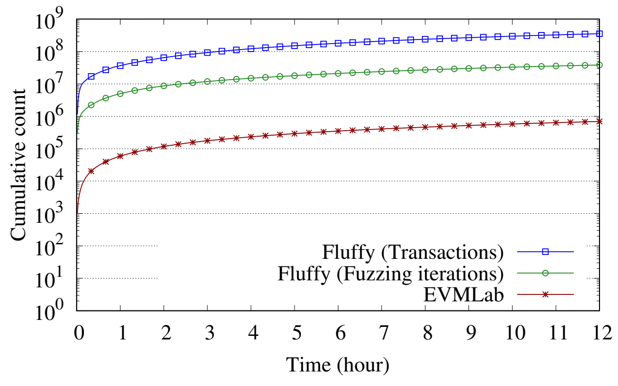


Figure 8: The total number of transactions and fuzzing iterations processed by Fluffy and EVMLab over time. Fluffy processes a varying number of transactions across iterations, whereas EVMLab processes 1 transaction per iteration.

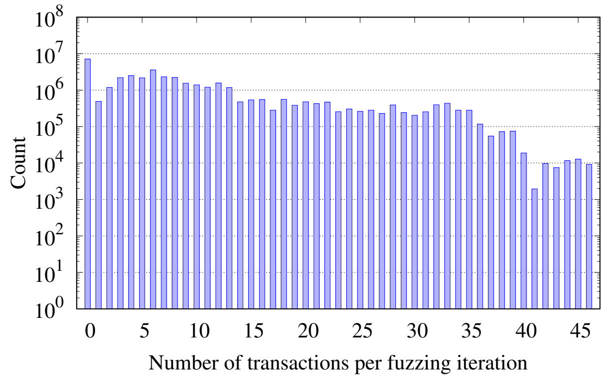


Figure 9: The distribution of the number of transactions that Fluffy processes per fuzzing iteration in 12 hours.

and fuzzing iterations processed by Fluffy and EVMLab over time. Fluffy processes a varying number of transactions across iterations, whereas EVMLab processes exactly 1 transaction in every iteration. After 12 hours, Fluffy processes more than 350 million transactions and more than 38 million fuzzing iterations. On average, Fluffy processes 9.2 transactions per fuzzing iteration. In contrast, EVMLab processes less than 700 thousand transactions and fuzzing iterations after 12 hours. In total, Fluffy processes 510 times as many transactions and 55 times as many iterations as EVMLab.

We then examine the number of transactions that Fluffy processes in each fuzzing iteration. Figure 9 shows the distribution of the number of transactions. 3 to 8 transactions are processed most frequently, except for the test cases with zero transaction. The largest number of transactions that Fluffy tests in an iteration for this specific 12-hour fuzzing experiment is 46. As a comparison, the number of transactions that have produced the blockchain state in the live Ethereum mainnet is more than 800 million transactions, as of August 2020. Therefore, this result shows that Fluffy tests Ethereum clients

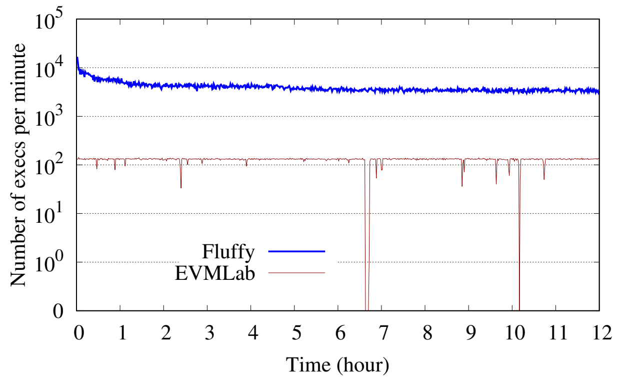


Figure 10: The number of fuzzing iterations executed per minute over time. We run the fuzzers in sequential mode without parallel fuzzing for this experiment.

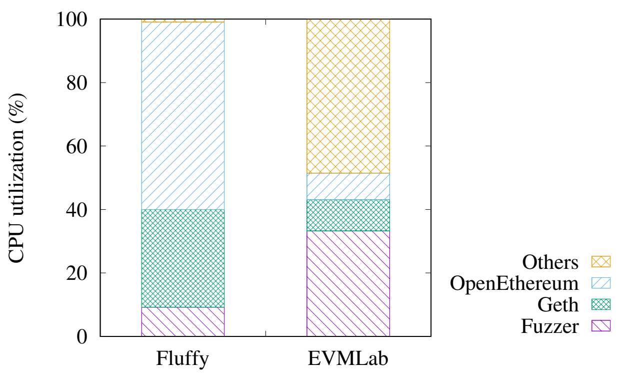


Figure 11: The CPU usage breakdown when running Fluffy and EVMLab.

using a small number of transactions that rewrite blockchain history.

Although the largest number of transactions is 46 for this experiment, Fluffy can test more or fewer number of transactions in a different experiment that runs for the same 12 hours, since how Fluffy determines the number of transactions is nondeterministic (§ 5.1). Furthermore, there can be consensus bugs that require more than 46 transactions to trigger.

Next, we examine the throughput of a single fuzzing instance through running Fluffy and EVMLab in sequential execution mode without parallel fuzzing for 12 hours. Figure 10 shows the number of fuzzing iterations executed per minute over time.

Fluffy achieves and sustains an order of magnitude higher throughput compared to EVMLab. In the beginning, Fluffy executes up to 16,805 fuzzing iterations per minute. We observe that in the beginning Fluffy generates and executes transactions that invoke a small number of bytecode instructions, which allows Fluffy to quickly execute more iterations. The throughput decreases gradually over time, as Fluffy discovers new inputs that invoke many bytecode instructions and executes mutations of those inputs. After 12 hours, Fluffy executes around 3,500 fuzzing iterations per minute.

In contrast, the throughput of EVMLab is overall flat, but fluctuates wildly at certain times during fuzzing. In particular, EVMLab fails to complete a fuzzing iteration for more than a minute between 6 and 7 hours, and 10 and 11 hours after fuzzing. We observe that such fluctuations occur when EVMLab is stuck in processing an excessively long sequence of bytecode instructions. Fluffy does not experience this, because Fluffy limits the number of bytecode instructions that are executed through limiting the gas limit of transactions.

We now measure the CPU usage to analyze why Fluffy achieves higher throughput, as both Fluffy and EVMLab are CPU-bound. Figure 11 shows the CPU usage breakdown when running Fluffy and EVMLab. We obtain the numbers through running the Linux perf command for one minute while running each fuzzer.

In case of Fluffy, most of the CPU time is spent in executing the code of OpenEthereum and Geth. We also observe that the majority of the CPU time spent in the clients is used to execute the core EVM interpreter logic, where most of the consensus bugs have been found.

In contrast, EVMLab suffers from the overhead of executing its own code, which is written in Python, and handling other tasks. We observe that much of the other tasks are associated with Docker [37], although EVMLab reuses Docker containers when spawning new OpenEthereum and Geth processes to test new inputs. We also observe that a large portion of the CPU time for executing the clients is spent in parsing test inputs and initializing the clients, where consensus bugs are unlikely to be found.

We emphasize that Fluffy does not outperform EVMLab by simply using a more efficient programming language and using specific configurations. In Figure 11, excluding the overhead of the fuzzer code (Fuzzer) and other tasks (Others), the CPU used for the Ethereum clients (OpenEthereum and Geth) is 18.2% for EVMLab and 89.9% for Fluffy, which is 4.9×. However, Fluffy processes 510× transactions and 55× fuzzing iterations compared to EVMLab, which is much larger than 4.9×. This shows that Fluffy executes Ethereum clients much more efficiently, even if the overhead external to Ethereum clients is the same as EVMLab.

Result. Fluffy processes 510 times and 55 times as many transactions and fuzzing iterations as EVMLab. The result shows that Fluffy efficiently tests many instances of multiple transactions that rewrite blockchain history.

7.4 Debugging

We used the crash debugger of Fluffy to analyze crashes. We encountered a false positive bug, when we set the initial blockchain state to a state that contains only the genesis account. We analyzed that the latest mainnet blockchain state cannot transition to a state that contains the accounts in the starting state of the bug, because the starting state contains a zero-balance account under the address of a precompiled contract. Creating such account was possible only before a previous EVM upgrade (related to Bug #15 in Table 2). To avoid triggering this false positive bug, we added to the initial blockchain state non-zero balance accounts under the addresses of precompiled contracts. We also used the crash debugger to analyze the shallow copy bug and the transfer-after-destruct bug, and create minimal test cases.

Result. Fluffy enables analyzing crashes triggered by consensus bugs.

8 Discussion and Limitations

We discuss future research directions, and limitations of the current design of Fluffy.

Smart contract blockchains. Our idea of multi-transaction differential fuzzing can be applied to other blockchains that provide smart contract capabilities like Ethereum. Smart contract blockchains have a total market capitalization of 95 billion USD, as of December 2020, and include popular blockchains such as Ethereum, Cardano, Stellar, EOS, Tron, Tezos, and Neo [12]. Like Ethereum, multiple depending transactions which create and invoke smart contracts determine the transitions of blockchain states in these blockchains. Therefore, techniques of Fluffy such as multi-transaction test cases and semantic-aware mutation strategy can be applied to find consensus bugs in these other blockchains.

Many-client fuzzing. Another research direction is to fuzz multiple versions of many Ethereum clients in addition to a single version of OpenEthereum (Rust) and Geth (Go). Although the two clients are used by 98% of nodes participating in the Ethereum mainnet, as of August 2020 [6], the Ethereum community is becoming more aware of the benefits of using multiple different clients since the shallow copy bug we reported was triggered in the live mainnet [15]. Examples of other Ethereum clients are Aleth (C++), Trinity (Python), Besu (Java), Nethermind (.NET), and EthereumJS (Javascript) [16]. Moreover, it is worthwhile to fuzz not only the latest version of the clients but also previous versions, since many of the decentralized Ethereum nodes in the mainnet do not immediately upgrade when new versions are released, and keep on using a previous version [6]. While it is straightforward to extend the current implementation of Fluffy to fuzz many clients, it remains a challenge to achieve high fuzzing throughput while executing multiple transactions on many clients.

Mutating client program states. Fluffy models an Ethereum client as a client program state model, rather than an EVM state model. Nevertheless, Fluffy mutates the Ethereum client program state indirectly through setting the initial program state to a corresponding initial EVM state and executing multiple transactions. This is because, like other fuzzers, Fluffy does not directly mutate internal states of the target programs. An alternative approach is to directly mutate client program states. However, it would be challenging to directly generate valid client program states that are reachable with transactions, such that the found bugs are exploitable on the Ethereum mainnet.

Limitations of differential fuzzing. Similar to existing differential fuzzers for Ethereum, Fluffy is unable to find bugs when the Etherum clients transition to the same incorrect blockchain state due to the same consensus bug. A practical solution to this limitation is to fuzz many different versions of different client implementations, since it is unlikely that the same bug exists in all of these different clients. A more fundamental solution is to utilize the EVM specification itself as an oracle, similar to how Hydra [27] implements an emulator along with a fuzzer. However, this approach reduces the number of input generation and testing as well as requires extensive engineering efforts unlike differential testing. We also note that, to our knowledge, there has been no previous case of the same bug occurring in multiple Ethereum client implementations.

Limitations of fuzzing. Like existing fuzzers for Ethereum, Fluffy inherits the limitations of fuzzing. Although fuzzing is good at exploring code paths with loose branch conditions (e.g., x > 0), fuzzing struggles to drive the target program into paths with tight branch conditions (e.g., x == 0xdeadbeef) [9, 48, 56]. This limitation is demonstrated by Fluffy failing to find Bug #9 in Table 2 within 12 hours, which requires specific inputs that satisfy tight branch conditions to trigger. We can address the limitation by combining fuzzing with concolic execution [9, 25, 48, 56], which interprets target program variables as symbolic variables and uses constraint solving to generate specific inputs that satisfy branch conditions.

9 Related Work

Fluffy is the first multi-transaction differential fuzzer for finding consensus bugs in Ethereum. In this section we describe existing works that are related to Fluffy.

Consensus in blockchains. Consensus in blockchains are increasingly becoming important as blockchains such as Bitcoin [39] and Ethereum [17] are becoming increasingly used. Researchers have proposed various techniques related to consensus in blockchains to improve the scalability, security, and usability of blockchains [4, 24, 28, 29, 31, 44, 47]. Our work complements these works by focusing on the implementation aspects of consensus in blockchains, and finding consensus bugs in Ethereum clients that lead to network split and theft.

Differential testing for consensus bugs. Differential testing is an effective software testing method that has been applied to various systems [3, 8, 10, 36, 42, 55]. Several fuzzers have been proposed to apply differential testing techniques to find consensus bugs in Ethereum [18, 22, 33]. These fuzzers generate a blockchain state and a transaction that transforms the state. Fluffy is also a differential fuzzer for finding consensus bugs, but Fluffy generates and runs multiple transactions that rewrite blockchain history and adopts various optimizations to improve the fuzzing throughput and the code coverage.

Coverage-guided fuzzing. Coverage-guided fuzzers such as libFuzzer [30] and AFL [38] leverage code path statistics to mutate test inputs. Fluffy extends such coverage-guided fuzzing mechanisms through extending libFuzzer. Leveraging more sophisticated mechanisms like gradient-guided techniques [46] is left as future work.

Smart contract vulnerabilities. Existing techniques for finding smart contract vulnerabilities [26, 32, 40, 49, 50, 57] focus on vulnerabilities in the business logic of smart contracts and transactions, whereas Fluffy focuses on vulnerabilities in the underlying Ethereum client implementations. For example, TxSpector [57] replays transaction history to extract logic relations, and applies user-specific logic rules to uncover vulnerabilities such as the re-entrancy vulnerability. In contrast, Fluffy generates and tests transactions which have never occurred in blockchain history to trigger consensus bugs in Ethereum clients that alter how the vulnerable clients execute the business logic of smart contracts.

10 Conclusion

Consensus bugs in Ethereum are extremely rare but can be exploited for network split and theft, which cause reliability and security-critical issues in the Ethereum ecosystem. Our fuzzer, called Fluffy, shows how to find consensus bugs hidden in deep states of Ethereum clients. Unlike existing fuzzers for Ethereum, Fluffy supports multi-transaction tests and uses different Ethereum clients as cross-referencing oracles. Fluffy also greatly improves the fuzzing throughput and the code coverage with various optimizations: in-process fuzzing, fuzzing harnesses for Ethereum clients, and semantic-aware multi-transaction mutation that reduces erroneous test cases. Fluffy found two new consensus bugs in the most popular Geth client which were exploitable on the live Ethereum mainnet. Fluffy is publicly available at https://github.com/snuspl/fluffy.

11 Acknowledgements

We thank our shepherd Ding Yuan and the anonymous reviewers for their insightful feedback. We thank the members of the Software Platform Lab at Seoul National University for their valuable input. This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

[1] Adriana Hamacher. So, what is the Ethereum (ETH) total supply?, August 2020. https://decrypt.co/38271/so-what-is-the-ethereum-eth-total-supply.

[2] Andrey Shevchenko. Binance briefly pauses Ethereum withdrawals as network suffers 'minor hard-fork', November 2020. https://cointelegraph.com/news/binance-pauses-ethereum-withdrawals-as-network-suffers-minor-hard-fork.

[3] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-box Differential Automata Learning. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.

[4] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A Device for Secure Transaction Approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[5] Algirdas Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 1985.

[6] Bitfly. Ethereum Mainnet Statistics. https://ethernodes.org, Accessed August 2020.

[7] Rainer Böhme, Lisa Eckey, Tyler Moore, Neha Narula, Tim Ruffing, and Aviv Zohar. Responsible Vulnerability Disclosure in Cryptocurrencies. *Commun. ACM*, 2020.

[8] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[9] George Candea and Patrice Godefroid. Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances. In *Computing and Software Science - State of the Art and Perspectives*. Springer, 2019.

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-Directed Differential Testing of JVM Implementations. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[11] Colin Harper. Ethereum's 'Unannounced Hard Fork' Was Trying to Prevent the Very Disruption It Caused, November 2020. https://coindesk.com/ethereums-hard-fork-disruption.

[12] CryptoSlate. Smart Contracts Coins: Protocols intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. https://cryptoslate.com/cryptos/smart-contracts, Accessed December 2020.

[13] David Siegel. Understanding The DAO Attack, June 2016. https://coindesk.com/understanding-dao-hack-journalists.

[14] Derek Parker. Delve: A Debugger for the Go Programming Language. https://github.com/go-delve/delve, Accessed August 2020.

[15] Eleazar Galano. Infura Mainnet Outage Post-Mortem 2020-11-11, November 2020. https://blog.infura.io/infura-mainnet-outage-post-mortem-2020-11-11.

[16] Ethereum. Clients, tools, dapp browsers, wallets and other projects. https://github.com/ethereum/wiki/wiki/Clients,-tools,-dapp-browsers,-wallets-and-other-projects, Accessed December 2020.

[17] Ethereum. Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/, Accessed August 2020.

[18] Ethereum. EVM lab utilities: Utilities for interacting with the Ethereum virtual machine. https://github.com/ethereum/evmlab, Accessed August 2020.

[19] Ethereum. Go Ethereum: Official Go implementation of the Ethereum protocol. https://geth.ethereum.org, Accessed August 2020.

[20] Ethereum. Solidity: An object-oriented, high-level language for implementing smart contracts. https://solidity.readthedocs.io/en/develop, Accessed August 2020.

[21] Ethereum. The Ethereum Bounty Program. https://bounty.ethereum.org, Accessed August 2020.

[22] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. EVMFuzzer: Detect EVM Vulnerabilities via Fuzz Testing. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[23] Geth team. Geth security release: Critical patch for CVE-2020-28362, November 2020. https://blog.ethereum.org/2020/11/12/geth_security_release.

[24] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[26] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *31st IEEE Computer Security Foundations Symposium (CSF)*, 2018.

[27] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[28] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A Decentralized Blockchain with High Throughput and Fast Confirmation. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[29] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[30] LLVM Project. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, Accessed August 2020.

[31] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and Secure Global Payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[32] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016.

[33] Martin Holst Swende. One year of Ethereum Security, November 2017. Devcon 3.

[34] Martin Holst Swende. Protecting The Baselayer - from Shanghai to Osaka, October 2019. Devcon 5.

[35] Martin Holst Swende. Shallow copy in the 0x4 precompile could lead to EVM memory corruption, November 2020. https://github.com/ethereum/go-ethereum/security/advisories/GHSA-69v6-xc2j-r2jf.

[36] William M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 1998.

[37] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, March 2014.

[38] Michał Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl, Accessed August 2020.

[39] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. https://bitcoin.org/bitcoin.pdf.

[40] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[41] OpenEthereum. OpenEthereum: Fast and feature-rich multi-network Ethereum client. https://github.com/openethereum/openethereum, Accessed August 2020.

[42] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[43] Péter Szilágyi. Geth v1.9.17 Post Mortem, November 2020. https://gist.github.com/karalabe/e1891c8a99fdc16c4e60d9713c35401f.

[44] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. Blockene: A High-throughput Blockchain Over Mobile Devices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[45] Scott Chipolina. How a Dormant Bug Briefly Split the Ethereum Blockchain, November 2020. https://decrypt.co/47891/how-a-dormant-bug-briefly-split-the-ethereum-blockchain.

[46] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[47] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. High Throughput Cryptocurrency Routing in Payment Channel Networks . In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[49] Christof Ferreira Torres, Mathis Steichen, et al. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *Proceedings of the 28th USENIX Security Symposium (Security)*, 2019.

[50] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[51] Vitalik Buterin. Security alert [11/24/2016]: Consensus bug in geth v1.4.19 and v1.5.2, November 2016. https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2.

[52] William Foxley. Developers Debate Disclosure Protocols After 'Accidental' Ethereum Hard Fork, November 2020. https://coindesk.com/developers-debate-disclosure-protocols-accidental-ethereum-hard-fork.

[53] William Foxley. How Much Ether Is Out There? Ethereum Developers Create New Scripts for Self-Verification, August 2020. https://coindesk.com/how-much-ether-is-out-there-ethereum-developers-create-new-scripts-for-self-verification.

[54] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. https://gavwood.com/paper.pdf.

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI), 2011.

[56] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, 2018.

[57] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In *Proceedings of the 29th USENIX Security Symposium (Security)*, 2020.