An Off-The-Chain Execution Environment for Scalable Testing and Profiling of Smart Contracts

Yeonsoo Kim and Seongho Jeong, *Yonsei University;* Kamil Jezek, *The University of Sydney;* Bernd Burgstaller, *Yonsei University;* Bernhard Scholz, *The University of Sydney*

https://www.usenix.org/conference/atc21/presentation/kim-yeonsoo

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

An Off-The-Chain Execution Environment for Scalable Testing and Profiling of Smart Contracts

Yeonsoo Kim[†], Seongho Jeong[†], Kamil Jezek[‡], Bernd Burgstaller[†], and Bernhard Scholz[‡]
[†]*Yonsei University* [‡]*The University of Sydney*

Abstract

Smart contracts in Ethereum are executable programs deployed on the blockchain, which require a client for their execution. When a client executes a smart contract, a world state containing contract storage and account details is changed in a consistent fashion. Hence, the execution of smart contracts must be sequential to ensure a deterministic representation of the world state. Due to recent growth, the world state has been bloated, making testing and profiling of Ethereum transactions at scale very difficult.

In this work, we introduce a novel off-the-chain execution environment for scalable testing and profiling of smart contracts. We disconnect transactions from the world state by using substates to execute the transactions in isolation and in parallel. Compared to an Ethereum client, our execution environment reduces the space required to replay the transactions of the initial 9 M blocks from 700.11 GB to 285.39 GB. We increased throughput from 620.62 tx/s to 2,817.98 tx/s (single-threaded) and 30,168.76 tx/s (scaled to 44 cores). We demonstrate the scalability of our off-the-chain execution environment for hard-fork testing, metric evaluations of smart contracts, and contract fuzzing.

1 Introduction

Ethereum is a permissionless blockchain capable of executing smart contracts. Since its inception in 2015, a total of 964 M transactions deployed in 11.6 M blocks have been processed. The number of unique Ethereum accounts soared from 20 M in 2018 to more than 131 M by January 2021. The rapid adoption has been fueled by applications in trade [34], banking [27, 37], governance, and supply chain. This trend has been amplified by a growing interest in decentralized finance[1], with a total value of 4 billion USD and a recent peak of 3.1 M contract calls in a single day [10].

Because smart contracts control large monetary values, blockchain designers and developers have a strong incentive in building tools that improve the correctness, efficiency, and security of their smart contracts. Developers typically write smart contracts in the Solidity language [17]. The Solidity compiler translates smart contracts into an immutable bytecode representation for the Ethereum virtual machine (EVM). Bytecode is persistently deployed on the blockchain where it can be invoked for execution on the EVM in a transaction.

The execution of smart contracts and Ethereum itself is a protocol defined formally in the Yellow paper [44] and practically implemented in Ethereum clients. Ethereum clients function as peers on the Ethereum network. They keep the ledger of the Ethereum blockchain consistent, fetch its updates, and integrate the EVM to execute smart contracts. The two most popular clients are Geth [18][2] and Parity [39][3]. When a client executes a smart contract, it constructs its input state from the blockchain's ledger and writes its state changes back onto the ledger. Hence, a smart contract's state evolves continuously on the ledger, where the history of all state changes of a smart contract is kept.

For testing, the state of a smart contract must be retrieved before it can be validated and analyzed. However, due to the ledger's cryptographically secured data structures and due to its sequential representation of the ledger as a blockchain, the retrieval of a smart contract's state is prohibitively slow [35, 46]. This problem is exacerbated for testing and debugging tools that process large quantities of blocks (or even all blocks) on the blockchain.

It has been acknowledged by the community that the testing of smart contracts at scale is a long-standing problem. It affects the development of the entire blockchain-oriented software (BOS) ecosystem, i.e., all software systems that work with a blockchain implementation. There have been calls for research into mocking blockchains [40] for enhancing testing

[1]Decentralized finance (DeFi) refers to an alternative, peer-to-peer financial infrastructure built on the blockchain.

[2]Geth is the most widespread open source Ethereum client officially maintained by the Ethereum community, with a market share of 80 %, according to `https://ethernodes.org/`.

[3]In Jan. 2020, Parity has been renamed to OpenEthereum.

and debugging in isolation. A recent survey [50] conducted among practitioners, GitHub developers, and industry professionals working on Ethereum smart contracts revealed the concern that the "*EVM is a single-threaded machine that cannot run transactions in parallel*", which may affect "*people who have a higher requirement on the timely reaction and verification of their transactions*". A survey [5] among 156 developers of popular blockchain projects on GitHub shows that backward compatibility (the ability to validate earlier transactions) and the difficulty of setting up testnets[4] are pressing issue for BOS developers, and that the distributed environment currently cannot be adequately simulated on development machines.

To mitigate the above problems and enable testing and profiling of smart contracts at scale, we introduce a *transaction record and replay* mechanism that enables the fast re-execution of individual transactions for testing purposes. Our mechanism records the historical Ethereum state that was used when a transaction was originally executed. The transaction can then be replayed in isolation for testing purposes. Our recorder reorganizes the Ethereum world state into transaction-relevant substates. Replay is conducted off-the-chain to avoid the overhead of the distributed system. The replayer mocks the distributed blockchain environment so that a transaction has the same execution behavior as if it was executed by an Ethereum client. Our approach facilitates the efficient replay of all transactions of the blockchain because transactions can be replayed in parallel (unlike testnets) and in isolation. We demonstrate that our novel *off-the-chain* execution environment enables applications such as hard-fork testing (i.e., a regression test checking whether the newly introduced policies of a hard fork do not hamper the execution of legacy smart contracts), metric evaluations of smart contracts (e.g., measuring the number of wasteful instructions in a smart contract), and contract fuzzing for detecting execution anomalies in smart contracts.

The contributions of this work are as follows:

- We present a new off-the-chain execution environment for blockchain transactions using a recorder and replayer so that transactions can run in isolation showing unprecedented performance improvements in comparison with state-of-the-art approaches.
- We introduce a new substate representation for transactions that captures the substate of the world state which is relevant for their execution.
- We show the efficiency and effectiveness of our approach using a regression tester for hard forks, a dead-code analysis, and a program fuzzer.

[4]A testnet is a blockchain that is used for testing of smart contracts in a production-like environment, but without spending cryptocurrency of real value.

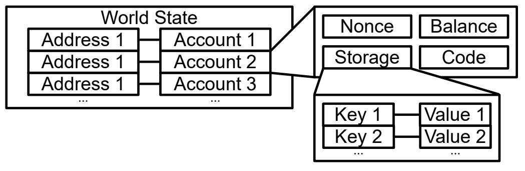
2 Background and Motivation

Ethereum Blockchain: A blockchain is a ledger shared among peers on a peer-to-peer (P2P) network. The Ethereum blockchain is fully described in its specification, the Yellow paper [44]. A *blockchain* is implemented as a chain of blocks that has been accepted by participants in the network following a consensus protocol. A *block* consists of a block header and a block body. The block header contains the reference to the parent block and metadata for the block verification. The block body holds a sequence of transactions created by participants and added in the block by miners. Transactions may either transfer funds or invoke smart contracts.

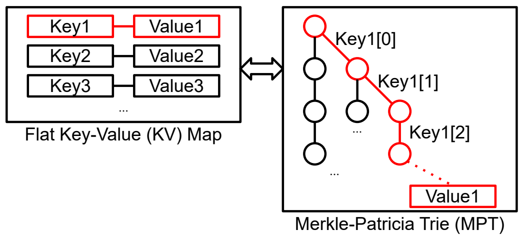
A *transaction* in the blockchain is signed by its sender and contains input data that are propagated into the global state. The execution of a transaction can be perceived as a state transition where a global state is transformed. The global state encompasses the state of smart contracts, accounts, and other aspects such as transaction receipts or smart-contract execution logs. Accounts are addressed by public keys generated by the private keys of the account owners. The sender issuing a transaction has to sign the transaction with the account's private key. Peers can verify the transaction by computing the sender's account address from the signatures. It is important to note that the state transition must be deterministic among all peers in the blockchain network for the sake of consistency. Without deterministic state transitions of a transaction, it is impossible to find consensus.

The Ethereum blockchain provides the following three types of transactions. A *transfer* transaction takes assets from a sender and transfers them to a recipient. A transaction for *contract creation* conveys the initial endowment of the newly created account and a smart contract in the form of EVM bytecode to initialize the created account. The created account is associated with contract code, and it has an account storage to maintain its state across contract invocations. A *contract invocation* transaction sends a message to an account to execute the associated contract code. The message contains input data and gas converted from cryptocurrency to fuel the EVM bytecode execution. The EVM will execute the contract code with the input data, consuming the gas metered based on executed instructions. If the gas consumption reaches the provided gas limit, the contract is terminated prematurely.

Ethereum World State: The Ethereum world state is the global state information of Ethereum that maps between account addresses and the data associated with an account. Each account has its own key-value storage in which only the owner has the authority to load and store values via `SLOAD` and `SSTORE` instructions of EVM bytecode. For state verification and as input to the next block, Ethereum clients are required to maintain a local copy of the entire global state after processing a particular block. As depicted in Figure 1a, Ethereum employs key-value (KV) maps to maintain (1) accounts and



(a) Ethereum world state and account storage.



(b) Flat KV map and corresponding MPT.

Figure 1: Ethereum world state and account storage (a), corresponding MPT as the underlying representation of a flat KV map (b).

(2) the data stored in each account. Data is encoded in recursive length prefix (RLP) format [44]. For cryptographic verification, the flat KV maps are encoded as Merkle Patricia tries (MPTs, [16]), depicted in Figure 1b. State verification in Ethereum relies on the property of MPTs that if two MPTs are *identical*, the hash values of their root nodes are identical [33].

The EVM bytecode interface provides the view of the flat KV map from Figure 1a to the application. Only after transaction execution will a client employ the RLP and MPT encoding depicted in Figure 1b, verify states, and store the MPT nodes in an on-disk key-value database (KVDB). A client may use a data representation different from MPTs to improve database performance [1]. Nevertheless, MPTs must still be constructed for state verification as mandated by the ledger protocol. The specification of the EVM and the world state are provided in the Ethereum Yellow paper [44].

3 Limitations of Smart Contract Testing

Testing smart contracts requires an Ethereum world state and a transaction execution environment. We refer to *transaction replay* as the execution of a transaction at the historical Ethereum state that the transaction was originally executed on at a given block height on the blockchain. Note that, in principle, it is not necessary to have the Ethereum network and blockchain available for testing. In this section, we consider different test environments currently available to blockchain designers and developers for replaying transactions. None of them facilitate replay on transaction granularity—only block granularity is supported. We highlight the impracticality of the test environments and conclude with the observed scalability problems. Unless otherwise specified, our evaluations were conducted on the server platform shown in Table 1.

Server	**Geth full node**	**Substate replayer**	**Geth archive node**
CPU	Intel Xeon E5-2699 v4, 2.2 GHz to 3.6 GHz, 22 cores × 2 sockets		
RAM	512 GB DDR4 RAM @ 2,400 MHz		
SSDs (PCIe)	Intel Optane 900P 480 GB Intel Optane DC P3700 800 GB		Samsung PM1725b 6.4 TB
OS	CentOS Linux release 7.9.2009 (core), kernel version 4.11.3-1.el7.elrepo.x86_64		
Filesystem	ZFS pool (1.2 TB total size)		XFS

Table 1: Evaluation platform specification.

3.1 Transaction Replay Delegation

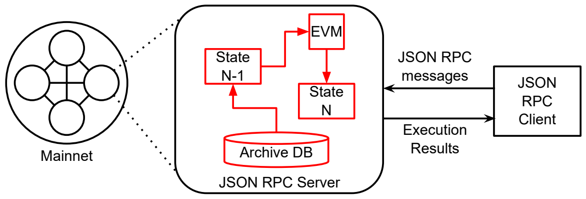


Figure 2: JSON RPC server (archive node) replaying transactions in block N on behalf of a client. State N is the state after executing all transactions in block N. The client sends request messages via the JSON RPC interface; messages include RPC method names and their arguments.

Ethereum clients can be configured as Ethereum *archive nodes*. They keep the entire history of world states in their database and can thus be used to retrieve historical state information. Geth provides a JSON RPC API[5] for this purpose. Via the JSON RPC API, a Geth client can delegate the replay of a transaction to an archive node. Figure 2 illustrates this scenario.

Unfortunately, delegation suffers from significant overhead incurred by the JSON RPC API and the fact that transaction throughput of archive nodes is seriously constrained by the considerable database size for storing all historical world states (6 TB of disk space for blocks up to 11 M [20]). An Ethereum client can be configured as an archive node that generates execution traces during transaction replay. In a small experiment, we observed that a query to replay traces of ten blocks at blocks after 9 M using `traceBlockByNumber` timed out running for an entire hour (at that stage we stopped the replay attempt). After disabling the EVM stack, memory, and storage tracing, the archive node achieved a replay throughput of 19.4 tx/s for 100 blocks at block 9 M and after. Hence, a test environment that uses transaction replay delegation is not a viable solution for testing—even for a small number of replay queries.

[5] https://geth.ethereum.org/docs/rpc/ns-debug

3.2 Transaction Replay in a Client

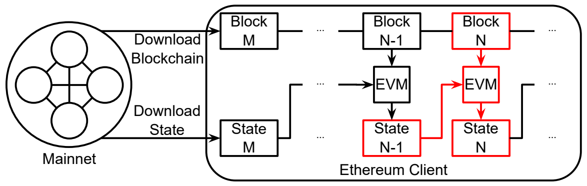


Figure 3: Full node to replay transactions in block N. State N is the state after block N has been imported, and block M is any block prior to block N on the blockchain.

When Ethereum clients are configured as Ethereum *full nodes*, they replay transactions for verification and synchronization with peers on the network. Figure 3 illustrates how Ethereum clients replay transactions during synchronization. To replay transactions in block N, a client requires the state at block $N-1$. A full node can download the state after block M directly from trusted peers using a fast synchronization protocol instead of processing all M blocks from the genesis block by itself. The fast synchronization protocol is a trade-off to save transaction processing time at the cost of the higher network bandwidth required for downloading the state. The community refers to fast synchronization as *fast-sync* mode and it represents the default configuration of Geth clients. At block 11.5 M, a Geth full node consumes more than 600 GB of disk space [21].

Transaction replay in full nodes suffers from severe drawbacks: If the desired block for replay is located far behind the tip of the chain, no peer will be available to provide the state for synchronizing in fast-sync mode. In the absence of a downloadable state, the client must process (import) all blocks itself, commencing from the genesis block, to produce state $N-1$ for replaying block N. In our experiment, the Geth client on an AWS `i3.2xlarge` instance took 8.13 h to fast-sync the state at block 11.5 M. When the Geth client imported blocks from the genesis block, it took 227 h (i.e., 9.5 d) to produce the state at block 8 M. Similarly to replay delegation, transaction replay throughput is very low: At block height 9 M the observed throughput of our archive node was 19.4 tx/s while our full node achieved 447.7 tx/s. Therefore, replaying millions of transactions with an archive node is not viable in practice, and a full node will take several weeks to accomplish such a testing task.

3.3 Testnets

The Ethereum main network (mainnet) is the only P2P network for real-world trading of Ethereum cryptocurrency. For testing purposes, the Ethereum community maintains public and private testnets where new protocols are activated and tested before they are deployed on the mainnet. Blockchain designers and developers deploy and test contracts on a testnet using clients that synchronize with recent blocks from the testnet like the nodes on the mainnet. The major limitation of testing with testnets is that its state can only be configured via side effects of smart contracts that are executed through transactions. This requires considerable effort from a developer to build a test fixture. In particular, the developer must deploy and execute the following transactions to test a smart contract on a testnet: (1) transactions to initialize the target contract and other participating accounts, (2) transactions to set up the complete world state and environment parameters for the input, and (3) a transaction to invoke the target contract with the prepared input state.

3.4 Lack of Scalability on Multicores

Testing smart contracts by replaying transactions on a full node does not scale on a multicore server because of two reasons: First, execution of transactions on the blockchain is inherently sequential. To replay transactions in blocks later than the current state, a node must import all intermediate blocks on the chain between the current block and the block to be replayed. Second, it takes a considerable amount of disk space to maintain a complete world state in MPT format. Running multiple Ethereum nodes with different ranges of blocks has the potential to increase the overall throughput on a multicore server. However, each node will require hundreds of GBs to maintain multiple world state instances. E.g., running nine full nodes for segments of 1 M blocks as stated in Table 2 requires 2.8 TB of disk space. One may expect an archive node to scale on multicores because it does not have to import blocks prior to replaying a transaction. But practically, as observed in our experiments, a single archive node requires 6 TB of disk space and exhibits very low transaction replay throughput. (Note that our evaluation platform from Table 1 uses PCIe SSDs and thus can be considered an advantageous case, performance-wise.) Testing smart contracts on a testnet has the same limitations as replaying of transactions on a full node because it executes transactions in order and has to encode a complete state in MPT format for verification.

4 Off-The-Chain Testing

Our new *off-the-chain* testing environment is based on a transaction record and replay mechanism. The record mechanism records the historical state from before and after a transaction on the chain. The recorder is an augmented Ethereum client that produces the historical state as a side effect while importing blocks from the blockchain. For a transaction, the recorder captures *substates*, which contain only relevant parts of the world-state and environment parameters that contain enough information to replay a transaction with no dependency on earlier transactions. The recorder stores the substates in the *substate database*, which is a flat key-value map.

The replay mechanism mocks the Ethereum protocol of a client such that transactions can be executed without the peer-to-peer network of Ethereum and the cryptographic verification tasks of a client. We use EVM binaries for building the replay mechanism, which are stand-alone clients and have been implemented for checking the correctness of the EVM only. Our replay mechanism operates directly on the substate database. It executes transactions off-the-chain, in parallel, and in isolation achieving unprecedented scalability for testing smart contracts.

4.1 Transaction Substate

Key to our record/replay mechanism is the substate, which contains a subset of the world-state trie to faithfully replay a transaction. The subset contains all the entries represented as a key-value pair (not as an MPT) for executing the transaction in full isolation. The substate is enhanced with meta information that is required for the execution of the transaction including (1) the values of the input arguments of the transaction, (2) the cryptographic hashes needed for the execution, and (3) the expected return values. In the following, we summarize the information stored in substates.

1. **Alloc**: information about each accessed account, i.e., account address, nonce, balance, code hash, and storage values from the world state accessed by the transaction.
2. **Block**: block number, block creation timestamp, block hashes, coinbase, difficulty, and block gas limit.
3. **Message**: nonce, gas price, provided gas, sender account address, recipient account address, input value, and input data of the transaction to initiate the message call.
4. **Result**: status code, transaction gas usage, logs and their Bloom filter from the output of the transaction execution.

The substate representing the historical Ethereum world state before executing a transaction is called the *input substate*. It consists of three parts: input alloc, block, and message. The substate representing the historical Ethereum world state after executing a transaction is called the *output substate*. It consists of two parts: output alloc and the transaction result. Together, the input and output substates contain all information required to faithfully replay and validate the respective transaction. Note that accounts and storage values that are not accessed by a transaction can be omitted from a substate because they have no effect on the transaction's execution and hence cannot affect the transaction's output.

4.2 Substate Recorder

Our substate recorder is an extended Ethereum client that collects and stores transactions' substates while importing blocks as depicted in Figure 4. Our recorder collects the substates from before and after a transaction on the chain. We refer to these substates as the input and output substates, respectively, and the recorder stores them as a substate record in the substate database.

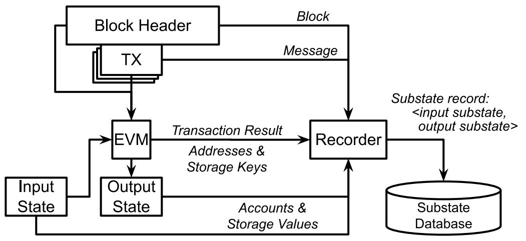


Figure 4: Substate recorder.

Throughout the execution of a transaction, the recorder collects the set of *indices* accessed in the world state. An index is represented by a tuple $\langle$`Address,Key`$\rangle$, where `Address` denotes an account/wallet address, and `Key` represents the storage key to the account's accessed storage location (as illustrated in Figure 1a). After the transaction's completion, the recorder collects for each index the corresponding value from the world state prior to the start of the transaction (for the input substate), and the value from the world state after the transaction terminated (for the output substate). The collected *input* and *output tuples* are of the shape $\langle\langle$`Address,Key`$\rangle$`,Value`$\rangle$. Collectively, the input/output tuples contain the complete set of storage locations and associated values read and written by a transaction.

As an example, we assume a *token transfer* of 10 units from account 1 at `address1` to account 2 at `address2`. Account 1 and account 2 are endowed with 25 units and 80 units before transaction commencement, and both accounts maintain their token values at storage key `key2`. Recording this transaction will result in the following input/output tuples, which signify the drop of funds in account 1 from 25 to 15 units, and the increase in account 2 from 80 to 90 units.

	Input tuples:	Output tuples:
(1)	$\langle\langle$`address1,key2`$\rangle$`,25`$\rangle$	$\langle\langle$`address1,key2`$\rangle$`,15`$\rangle$
	$\langle\langle$`address2,key2`$\rangle$`,80`$\rangle$	$\langle\langle$`address2,key2`$\rangle$`,90`$\rangle$

The recorded input/output tuples do not include storage locations where accounts keep information that has not been accessed as part of a given transaction. In practice, a transaction involves only a few accounts. The recorded input/output tuples thus constitute a small subset of the world state's account storage before and after a transaction, which facilitates space-efficient transaction replay.

We record input/output tuples for nested transactions, which occur with calls across smart contracts. Tuple collection includes reverted transactions, which are transactions that do not take effect on the world state, e.g., because the transaction runs out of gas. Even such invalid transaction termination requires the input/output tuples to ensure the faithful replay of the reverted transaction.

Consider as an example the aforementioned token-transfer contract. The contract has a runtime check for reverting a transaction if there are insufficient funds available. Assume that the initial endowment of account 1 is zero, hence, the runtime check will fail and make the transaction revert. In this case, the input substate will have the tuple $\langle\langle$`address1,key2`$\rangle$,`0`$\rangle$ only. This tuple is nevertheless necessary for reproducing the failed runtime check during replay.

As depicted in Figure 4, our recorder creates a *substate record* from the input/output substates (including the transaction's environment parameters, i.e., block, message, and transaction result), and stores the substate record in the substate database. Records in the substate database are indexed by the key $\langle$`block, tx`$\rangle$, where `block` is the block number and `tx` is the index of the transaction within the block. Each substate record can thereby be accessed by a single database lookup, which eliminates dependencies on earlier transactions and provides our replayer with instant access to the historical state of any recorded transaction.

Because our recorder collects substates during on-chain operations, it requires as much time and space as the synchronization of a Geth full node. However, the recording has to be performed only once. As soon as the substate database is created, all historical transactions can be replayed an arbitrary number of times with our replay mechanism.

4.3 Substate Replayer

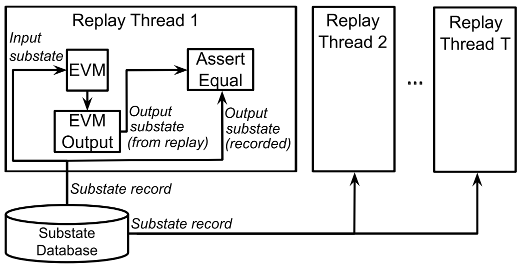


Figure 5: Substate replayer.

The replay mechanism is based on the EVM binaries that we extended for replaying transactions arbitrarily and in parallel from our substate database. Figure 5 illustrates the design of our multi-threaded replayer. It receives the number of replay threads T and an inclusive block range $[N_{\text{first}}, N_{\text{last}}]$ as input. The replayer creates a thread pool of T replay threads and pushes the designated block numbers from N_{first} to N_{last} into a channel on which all replay threads wait[6]. We employ a dynamic work partitioning scheme where replay threads repeatedly dequeue a block number, load the block's substate records, and replay its transactions.

[6]The EVM binaries and hence our extensions are implemented in the Go programming language.

For each transaction, a replay thread invokes a fresh EVM instance, for which it copies input alloc, block, and message from the input substate to an in-memory context that the EVM instance can read and modify off-the-chain. This in-memory context simulates on-chain features such as hard forks, gas costs, block environment, precompiled contracts, and database snapshots for reverted transactions.

The EVM instance executes the corresponding smart contract code in isolation on the in-memory context. The replay thread thereafter collects the output tuples and the transaction result from the modified context and validates them against the recorded output substate. If a thread successfully replayed all transactions in a block, it sends the block number back to the main thread. The main thread maintains the processed block total and triggers the termination of the replay threads once all blocks have been processed.

Replay threads collect the set of indices accessed in the world state in the same manner as the recorder. Because transaction replay must be deterministic (running the same transaction with the same input must produce the same output), the replayed transaction must access the same storage locations as the recorder, and the accessed indices must coincide.

For example, considering the token transfer example (1) on the previous page. If during replay the transaction accesses index `<address1, key1>` instead of `<address1, key2>`, the replayer will detect this index mismatch (cf. "Assert Equal" in Figure 5).

In general, if any of the accessed indices, output values, or the transaction result differ from the recorded historical information, the replayer will raise an exception, report the substate record's key and the difference between the EVM output and the expected output, and terminate.

4.3.1 Replay Performance and Accuracy

We evaluated the runtime and storage improvements of our replayer compared to a Geth full node, and we validated its replay accuracy. All experiments where conducted on the platform specified in Table 1. Tables 2 and 3 compare the disk space and execution-time requirements of a Geth full node and our substate replayer to replay all 590 million transactions of the initial 9 M blocks of the Ethereum mainnet. We imported blocks from files to replay transactions with the Geth full node. The size of Geth's database in Table 2 increases drastically as it contains all accounts existing in the previous blocks. The substate database requires less space than Geth because Geth must maintain a complete world state with all accounts and storage values in MPTs for on-chain synchronization, while our off-the-chain test environment selectively recorded accounts that are accessed during transaction execution.

Table 3 compares time and throughput of transaction replay between Geth and our substate replayer with a single thread. Both took more time in later blocks because the number of transactions per block increased. Blocks in range 2–

Blocks (M)	**Geth full node (GB)**	**Substate replayer (GB)**	**Space savings (%)**
0–1	0.96	0.68	29.17
1–2	3.00	1.83	39.00
2–3	29.30	16.91	42.29
3–4	37.76	6.58	82.57
4–5	124.57	39.55	68.25
5–6	272.06	51.58	81.04
6–7	407.48	50.30	87.66
7–8	551.04	55.96	89.84
8–9	700.11	62.00	91.14
0–9	700.11	285.39	59.24

Table 2: Disk space requirements and space savings of our substate replayer over a Geth full node. The space required by Geth is the size of its database at the last block of the stated range. Ranges 8–9 M and 0–9 M require the same space because the Geth full node maintains a complete world state at 9 M regardless of the number of blocks it replays.

Blocks (M)	**Geth full node**		**Substate replayer**		**Speed-up (×)**
	Time (s)	tx/s	**Time** (s)	tx/s	
0–1	1184	1414.07	526	3183.01	2.25
1–2	2879	2217.13	1517	4207.73	1.90
2–3	28906	252.73	24125	302.82	1.20
3–4	10775	1946.33	5222	4016.03	2.06
4–5	94868	1196.67	28873	3931.90	3.29
5–6	165673	748.71	35390	3504.97	4.68
6–7	173503	552.82	33672	2848.55	5.15
7–8	224426	485.51	38060	2862.87	5.90
8–9	248519	447.70	41999	2649.17	5.92
0–9	950733	620.62	209384	2817.98	4.54

Table 3: Total execution time (s) and throughput in transactions per second (tx/s) of Geth block import and substate replay with a single thread.

3 M showed lower performance because of denial-of-service (DoS) attacks described in [3]. In later blocks, the Geth full node was substantially slower than our substate replayer because Geth needs multiple LevelDB lookups to read MPT nodes leading to a single MPT leaf value. As the blockchain grows, the number of MPT nodes and the size of the Geth database increases, which increases both the total number of lookups and the time per lookup. In contrast, our substate recorder packages all values required to replay a transaction into a single substate record. Therefore, one LevelDB lookup is sufficient to load all data required to replay a transaction. Overall, our substate replayer is 4.54 times faster than the Geth full node. Because transaction replay from a substate record is an isolated execution on the EVM, it can be parallelized. Figure 6 shows the speedup when using multiple replay threads for the initial 9 M blocks of the Ethereum mainnet.

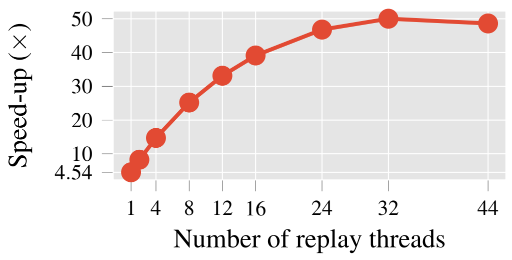


Figure 6: Speedup of parallel transaction replay by the multi-threaded substate replayer with 9 M blocks compared to a Geth full node.

As discussed in Section 4.3, our replayer will raise an exception if the execution of a replayed transaction deviates from the recorded historical information. When replaying the 590 million transactions on the Ethereum mainnet from the genesis block up to block 9 M, our replayer finished without raising an exception, which confirms that all transactions were replayed faithfully and that our off-the-chain testing environment achieved a replay accuracy [26] of 100 %.

5 Use Cases

In this section we introduce three use cases for our substate replayer: (1) a metric use case based on transaction replay, (2) a fuzzer use case based on testnets, (3) a method to assess hard forks using off-the-chain execution.

5.1 Metric Use Case

Program analysis techniques have been widely adopted for smart contracts. A 2019 survey of 27 Ethereum contract analysis tools [11] found that most tools concentrate on security issues and employ static analyses. But pressing efficiency and scalability limits of blockchains have become another major concern [13, 30, 41, 49]. We argue that because of the inherent limitations of static analyses, in particular precision and cost [7, 14], dynamic analysis techniques will gain further momentum. Specific metrics aimed at measuring the complexity, communication capability, gas-usage and performance have already been instigated in [40]. A set of metrics that includes contract execution time and the time to update the Ethereum world state has been proposed in [49]. Novel program metrics in smart contracts will be required to guide future design decisions for infrastructure and language implementation, e.g., to determine the profitability of speculative parallelization [12].

To demonstrate the effectiveness of our record/replay infrastructure for comprehensive dynamic analyses of transactions on the Ethereum mainnet, we implement a metric for *wasteful instructions*. A wasteful instruction is an instruction whose side effect does not lead to a lasting side effect on the

Nr.	Opcode	Nr.	Opcode
1	`PUSH 0x80`	13	`DUP`
2	`PUSH 0x40`	14	`REVERT`
3	`MSTORE`	15	`JUMPDEST`
4	`PUSH 0x00`	16	`POP`
5	`PUSH 0x01`	17	`PUSH 0x3797`
6	`SSTORE`	18	`DUP`
7	`CALLVALUE`	19	`PUSH 0x25`
8	`DUP`	20	`PUSH 0x00`
9	`ISZERO`	21	`CODECOPY`
10	`PUSH 0x15`	22	`PUSH 0x00`
11	`JUMPI`	23	`RETURN`
12	`PUSH 0x00`		

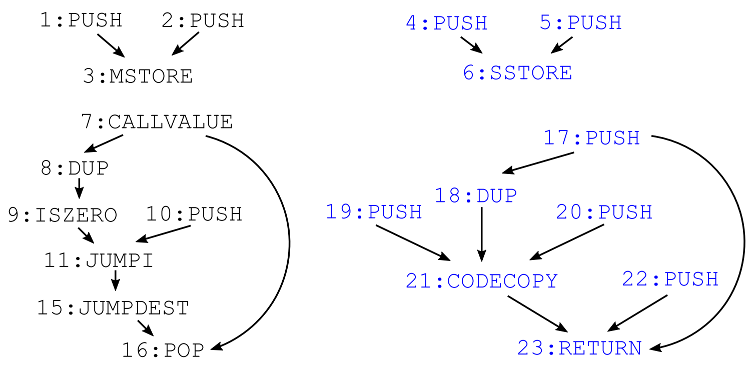


Figure 7: A bytecode of a smart contract and its value graph. Necessary instructions are colored in blue.

blockchain. Because each instruction needs to be paid for by gas, wasteful instructions are costly and should be avoided.

To determine wasteful instructions at runtime, we construct a value graph $G(V,E)$ for each smart contract execution. During the EVM execution, each instruction becomes a node in the value graph, and edges between nodes denote data flow among instructions. We define *necessary* instructions as instructions that contribute towards side effects on the blockchain or results to users. We categorize the list of necessary instructions as follows.

- `SSTORE`, `SELFDESTRUCT`, `CREATE`, and `CREATE2` cause side effects on the blockchain by changing the world state trie or the account storage trie.
- `RETURN` produces a result of the smart contract and delivers it to the user.
- `LOG0`, `LOG1`, `LOG2`, `LOG3`, and `LOG4` generate log records.
- `CALL`, `CALLCODE`, and `DELEGATECALL` execute another smart contract. Although these instructions do not have a side effect themselves, the callee contract might have a side effect. Investigating such a relationship is beyond this paper's scope, so we simplified and considered call-related instructions as necessary instructions. However, `STATICCALL`, which cannot have any side effect by its definition, is excluded.

A sample bytecode and its value graph are depicted in Figure 7. Each node in the graph represents an instruction with the attached number specifying the execution order. If instruction `x` depends on instruction `y`, we add a directed edge from instruction `y` to instruction `x`. For example, `3:MSTORE` receives two stack values as its arguments to store the value from `1:PUSH` at the memory address from `2:PUSH`. Dependencies are not restricted to values from the EVM stack but may extend to memory references. Instruction `23:RETURN` is such a case: it pops two arguments from the stack, which represent the address and length of data in memory to return. The third dependency (on `21:CODECOPY`) encodes the memory reference itself. Instructions 12–14 from the bytecode do not occur in the value graph because the underlying execution took the jump from instruction 11 to instruction 15 at runtime.

We built an algorithm that propagates the necessity of instructions in the value graph in a backward fashion, based on the introduced necessary instruction list. Necessary instructions are colored in blue in Figure 7. All instructions connected to `6.SSTORE` and `23.RETURN` are considered necessary. We obtain the set of wasteful instructions as the complement set of the necessary instructions.

To analyze the wastage characteristics of the Ethereum blockchain, we computed value graphs for the initial 9 M blocks on the replayer, which took 75 h to complete. A box plot and the average ratios of wasteful instructions are shown in Figure 8 and Figure 9.

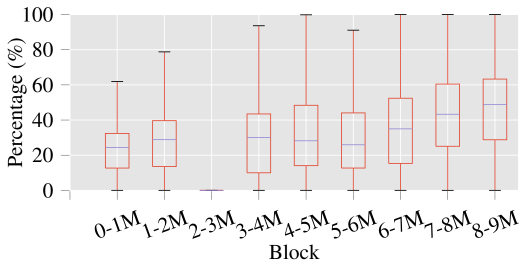


Figure 8: Box plots of wasteful instruction ratios for ranges of 1 M blocks.

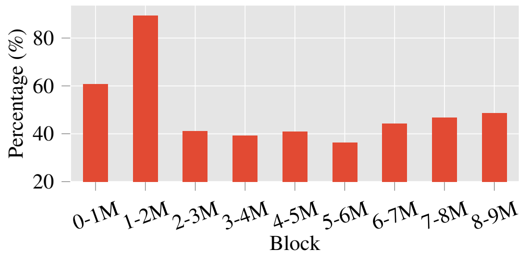


Figure 9: Average ratios of wasteful instructions for ranges of 1 M blocks.

The median and average of the wasteful instruction ratios are gradually increasing from block 5 M. As a consequence, wasteful instructions constitute nearly 50 % of the total instructions in range 8–9 M. The irregularity in range 2–3 M in Figure 8 is due to DoS attacks in the year 2016 (cf. [3]). As a result, the box and whiskers in this range stick to zero, while the average ratio is not affected. Another irregularity shows in range 1–2 M in Figure 9, because it contains smart contracts with a high total number of instructions but a negligible number of necessary instructions. This skewed the average ratio of wasteful instructions, although this was not caught in the box plot.

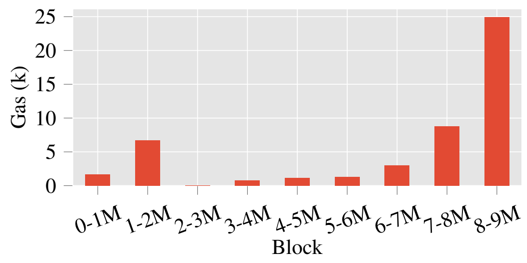


Figure 10: The average of wasted gas per transaction for ranges of 1 M blocks.

We calculated the gas consumed by wasteful instructions. As illustrated in Figure 10, recent transactions tend to waste more gas. We found a single transaction to waste nearly 25,000 gas in range 8–9 M.

5.2 Fuzzer Use Case

Fuzzing is a popular means to assess software quality [32] and exercise a program under test with many randomized inputs. A fuzzer aims to execute as many program paths as possible by varying input values, and, hence, fuzzers incur a large runtime overhead as they repeatedly execute the program under test. Some approaches reduce the environment overhead [47] or improve path coverage [36] for alleviating the fuzzing overheads. There is a variety of fuzzers available for Ethereum such as Echidna [24] and Harvey [45], which require user annotations in the Solidity source code for guiding the fuzzing process. Other fuzzing tools [31, 42] do not require annotations but still require the source code (not the EVM bytecode).

Low-performing fuzzers limit the effectiveness of a fuzzing campaign and may result in many false negatives [29]. Parallelizing fuzzers improve the performance but cannot be directly used for the blockchain testing due to its sequential execution. A parallel fuzzer for blockchains would require to replicate the blockchain for each parallel instance, which is not a viable approach.

We adapt our replay mechanism introduced in Section 4.3 for parallel fuzzing as a showcase. The aim of this showcase is to demonstrate the efficiency of the ContractFuzzer [29] using our parallel replay mechanism. The ContractFuzzer dynamically analyzes compiled bytecode, which is most suitable for third-party auditing of binaries that does not require user annotations for fuzzing. The ContractFuzzer cannot audit third-party contracts efficiently without our replay mechanism because contracts will be fuzzed on the testnet where all smart contracts undergoing the fuzzing campaign must be deployed (as described in Section 3.3). For each input variation of a fuzzing campaign, the contracts must be redeployed on the testnet slowing down the campaign.

For fuzzing, the ContractFuzzer tool needs an Ethereum smart contract application binary interface [19, ABI], which describes input parameters and the message type of a smart contract. Most re-usable contracts/services have their ABIs publicly available, e.g., via the Etherscan web service [22].

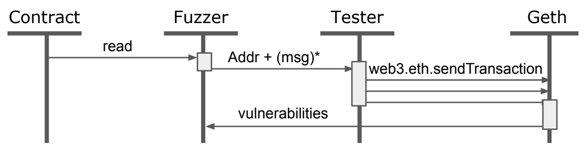


Figure 11: ContractFuzzer workflow.

Figure 11 depicts a sequence diagram of the Contract-Fuzzer workflow without our replay mechanism. The bytecode of a *contract* is read by the *fuzzer*, which analyzes the contract's ABI and generates a set of random messages that execute the contract and comply with its ABI. The *tester* sends the messages one-by-one to *Geth*, for execution on the EVM, which contains specific monitoring hooks for discovering vulnerabilities that occur during bytecode execution. The results from the bytecode execution are sent back to the *fuzzer*, which generates a vulnerability report.

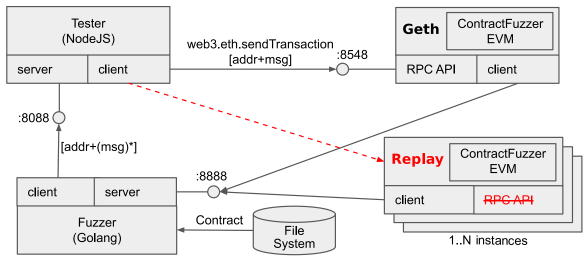


Figure 12: Extended ContractFuzzer architecture.

We extended the ContractFuzzer with our *replay* mechanism for parallel fuzzing as depicted in Figure 12. Our parallel fuzzing mechanism directly reads the input substate from the substate database (as indicated by the dashed arrow in Figure 12) such that fuzzing can be performed in parallel and without the need of deployment on the testnet.

The input variations generated by the ContractFuzzer in the form of random messages (see Figure 11) differ from the

Number of threads		**Throughput** (contracts/min)	**Speedup** (×) **over original**	**Speedup** (×) **replay**
Original	1	0.46	n/a	n/a
Replayer	1	3.28	7.07	n/a
	4	4.65	10.01	1.42
	8	6.25	13.46	1.90
	16	10.05	21.65	3.06
	32	15.99	34.44	4.87

Table 4: ContractFuzzer performance improvements.

recorded, historical *message* data of the input substate from Section 4.1. For each input variation, the replayer replaces the historical message data of the input substate by the information provided in the corresponding message from the ContractFuzzer. This modification of the transaction input obviates the need to validate the transaction ("Assert Equal" in Figure 5) because fuzzing does not attempt a faithful replay. Rather, the purpose of the input variation in the fuzzer is to execute untested program paths and uncover previously unobserved behavior in the smart contract under test. We used several of ContractFuzzer's sample contracts [28] to validate the integration of ContractFuzzer and its modified Geth EVM with our replayer. The integration produced the expected result, i.e., the same vulnerabilities as the original ContractFuzzer.

Our focus with this use case was to determine the achievable performance from off-the-chain fuzzing. For this experiment, we used our recording of the Ethereum mainnet. We created a simple NodeJS client for the Etherscan web service [22] to obtain the ABI of contracts. We used this client to get the first several hundred contracts from the mainnet that provide their ABI. Because ContractFuzzer supports contract analysis in batches, we divided this group into batches of ten contracts each. We have executed an analysis of this dataset using various combinations of batches running in parallel. We have measured the throughput of contracts (i.e, the number of analyzed contracts/min) as shown in Table 4. The first row lists the performance of the original ContractFuzzer, while the following rows list the throughput with increasing parallelism—i.e., the number of contract batches analyzed in parallel. The middle column provides the speedup over the original ContractFuzzer (w.r.t. the first row).

It follows from the second row that the ContractFuzzer with replay executing all batches in sequence is faster than the original ContractFuzzer. We attribute this to the fact that Geth adds more overhead compared to our light-weight substate. The speedup grows proportionally in the number of threads. The last column shows the relative speedup of multi-threaded over single-threaded replay.

Our fuzzing method has three key benefits: Firstly, our replayer provides smart contracts instantly and without prior deployment on a testnet, making the fuzzing configuration straightforward. Secondly, our database has no dependencies between substate records and enables parallel fuzzing. This allows fuzzing campaigns on smart contracts where an arbitrary number of threads can be utilized. Thirdly, the substate database always provides the same initial data, while fuzzing against the testnet requires repeated redeployment to start from the same initial state.

5.3 Hard Fork Assessment

The Ethereum specification has been updated over the years because of several reasons such as the introduction of new EVM instructions and the protection from DoS attacks. Peers willing to accept the update have participated by accepting a blockchain forked from the existing blockchain, or hard fork. A hard fork is first proposed via a meta Ethereum improvement proposal (meta EIP), which includes other EIPs. When the Ethereum community agrees on the hard fork, a certain block number is pre-determined after which the hard fork becomes effective.

At the time of writing there have been nine hard forks on the Ethereum blockchain that changed gas costs of existing instructions or introduced new instructions. Such updates of the EVM specification can cause problems on already-deployed smart contracts which—then—depend on an outdated specification. For example, Ethereum suffered from breaks of backward compatibility by EIP-1884 [15] during the Istanbul hard fork. The EIP changed the gas cost of several opcodes and it was expected that a few contracts will fail to operate. Indeed, Ethereum frameworks such as Aragon and Kyber found function calls in their smart contracts to fail with out-of-gas errors [6, 43]. Aragon expected that EIP-1884 would break 680 smart contracts in their framework [9], and had to release new smart contracts to replace them [43].

Therefore, it is essential to assess a hard fork on existing contracts before it is activated on the network. The Ethereum community maintains testnets where hard forks are activated and tested before being deployed on the mainnet. However, testnets have their own state databases which differ from the mainnet, and developers must execute transactions on a testnet to deploy contracts and predict possible impacts from a hard fork. This approach is merely a new execution of transactions and cannot reproduce the historical contexts with a new hard fork specification.

We propose the use of our replayer to assess new hard forks on already deployed smart contracts on the mainnet. Our replayer efficiently replays transactions in the same context except the protocols changed by the new hard fork. Hence, the effect of the hard fork on existing contracts can be observed, which helps decision making and hard-fork analysis.

For demonstration purposes we conducted an assessment of the historical hard forks on the Ethereum mainnet. Table 5 enumerates the history of the Ethereum hard forks at the time of writing. Out of nine hard forks, the *DAO Fork* and

Hard fork	Assessed transactions (M)	EVM runtime exception (%)			Output changed (%)	Gas usage changed (%)			Unaffected (%)
		Invalid JUMP	Invalid opcode	Reverted		Out-of-gas	Increased	Decreased	
Homestead	0.416	0.000	0.000	0.000	0.000	0.002	0.000	0.000	99.998
Tangerine Whistle	2.508	0.585	0.000	0.000	3.667	2.761	75.125	0.003	17.859
Spurious Dragon	3.055	0.530	0.000	0.000	9.407	2.549	71.182	0.001	16.331
Byzantium	26.014	0.062	0.000	0.000	2.560	0.299	8.359	0.001	88.718
Constantinople / Petersburg	193.668	0.008	0.000	0.000	0.344	0.040	1.123	0.000	98.485
Istanbul	303.300	0.064	0.198	1.009	3.804	7.884	68.494	18.547	0.000

Table 5: Hard fork assessment: percentage of affected and unaffected contract invocations and the share of each effect. Column "Assessed transactions" indicates the number of contract invocation transactions executed for the assessment. A transaction in "EVM runtime exception" was successful in the original invocation but raised an exception during the hard fork assessment. (1) "Invalid JUMP": destination of the `JUMP` instruction was not a `JUMPDEST` instruction. (2) "Invalid opcode": executed instruction was the `INVALID` instruction or not defined in the hard fork. (3) "Reverted": the `REVERT` instruction was executed. (4) "Output changed": side effect on replay output changed, disregarding gas usage and account balances. (5) "Out-of-gas": successful in the original invocation but raised out-of-gas exception in the hard fork assessment. (6) "Increased" & "Decreased": produced the same output as the original invocation except gas usage and account balances. (7) "Unaffected": produced the exactly same output as the original invocation.

Muir Glacier are not included because they do not affect the EVM specification for transaction execution. Hard fork *Constantinople / Petersburg* comprises two steps that took effect within the same block and hence were combined for this study. For the assessment of a hard fork on deployed contracts, all historic contract invocation transactions prior to the activation of the hard fork are relevant (cf. Section 2). The activation position of a hard fork on the chain thereby determines the number of transactions that need to be assessed, from the genesis block until the block where the hard fork became active. Column "Assessed transactions" in Table 5 depicts the number of contract invocation transactions that our replayer executed for each of the historical hard forks on the mainnet. Note that the *Istanbul* hard fork took place at block height 9.069 M, whereas the block range of our experiment was 0–9.0 M, but we do not regard the excluded 69 k blocks to be significant for this demonstration of our replayer. In total, our replayer executed 529 million contract invocation transactions as part of this assessment. The experiment took 15.15 h with an overall throughput of 9,694.30 tx/s.

The results, i.e., the effects of each hard fork on the contract invocation transactions prior to its activation on the mainnet, are depicted in Table 5. The columns below "EVM runtime exception" state the percentage of transactions for which the original invocation was successful but raised an exception in the hard fork assessment. `JUMP` instructions to invalid destinations and the `INVALID` instruction have been used as a pragmatic way to throw runtime exceptions. The *Byzantium* hard fork at block 4,370,000 introduced the `REVERT` instruction. The main difference is that `JUMP` and `INVALID` consume all remaining gas but `REVERT` refunds the remaining gas to the sender.

Effects on Execution Path The sum of column "EVM runtime exception" and column "Output changed" is the percentage of transactions for which the EVM specification change resulted in an execution path different from the original invocation. The hard fork with the highest ratio of sum of "EVM runtime exception" and "Output changed" is *Spurious Dragon* with 9.9 %, followed by *Istanbul*, and *Tangerine Whistle* with 5.1 % and 4.3 %. *Spurious Dragon* activated EIPs that increased the gas cost of EVM instructions, limited code size, and cleared empty accounts in the world state trie to protect the network from DoS attacks. The other two hard forks, *Istanbul*, and *Tangerine Whistle*, mainly updated gas costs of EVM instructions. This is an indication that the execution paths of those contracts are highly dependent on the EVM gas system. Hence, updating the gas system may cause such contracts to fail.

Effects on Gas Consumption The hard forks *Tangerine Whistle* and *Spurious Dragon* increased the gas costs of several EVM instructions, resulting in more than 70 % of contract invocations to consume more gas, and 2 % to raise an out-of-gas exception. The *Istanbul* hard fork increased the gas costs of several instructions that access tries and reduced the gas costs of loading call data input, which affected all contract invocations. As a result, 68.5 % of all contract invocations consumed more gas, 7.9 % raised an out-of-gas exception, and 18 % consumed less gas.

6 Related Work

Hartel and Staalduinen [26] proposed a tool to generate a replay script for historic transactions based on Truffle. It re-deploys the smart contract with historic data for replay on the blockchain. This approach cannot faithfully reproduce historic data because the execution environment may have changed on redeployment. Their re-execution fails to accurately replay 39 % of the 1120 sampled smart contracts [26]. In contrast, we identified the complete set of environmental parameters required to faithfully replay transactions. We achieved a replay accuracy of 100 % for the 590 million transactions on the Ethereum mainnet up to block 9 M. Our approach replays transactions off-the-chain and thereby eliminates the networking and consensus overhead. Our substate database provides direct access to the transaction-relevant substate of any recorded transaction, which allows transaction execution in isolation and at scale. In contrast, [26] can only execute transactions sequentially from the tip of the chain. Their method relies on the availability of the verified contract source code on Etherscan, but only 2.2 % of all Ethereum smart contracts deployed until Sept. 2018 have been made available as source code on Etherscan [38].

Transaction replay is often required to find vulnerabilities of smart contracts and verify the soundness of such methods. ContractFuzzer [29] and EVMFuzzer [23] adopted fuzzing techniques to spot weaknesses in smart contracts and EVMs. ContractFuzzer used 6,991 smart contracts for fuzzing and detected seven types of vulnerabilities. EVMFuzzer performed fuzzing for 253,153 contracts to expose vulnerabilities in different types of EVMs. Hartel and Schumi [25] used mutation testing to assess the quality of smart contracts. They injected smart contract specific mutations at a large scale and performed the experiment for more than 2 million transactions.

Replaying the whole blockchain can be conducted to understand the dynamic behavior of the Ethereum ecosystem. Yang et al. [48] and Baird et al. [4] measured transaction execution time for millions of blocks and discovered that time-per-gas ratios of instructions are not uniform, which potentially threatens the decentralization of the Ethereum network. Aldweesh et al. [2] observed a similar result by measuring CPU and gas usage of opcodes independently with their OpBench framework. TokenScope [8] replayed the blockchain to investigate inconsistent tokens in Ethereum. They took a trace-based approach to find inconsistencies by comparing the information about data structure, interfaces, and events. They reported 7,472 inconsistent tokens out of 57,411 tokens from 6 million examined blocks.

7 Conclusion

The Ethereum blockchain and its surrounding environment have been rapidly developed in recent years. This has led to a situation where "*the need for software engineers to devise specialized tools and techniques for blockchain-oriented software development*" [40] has arisen. However, there is a lack of tools that scale for tasks including third-party auditing, testing, debugging, and quality assurance.

Our work proposes a new infrastructure for the lightweight execution of smart contract transactions. Our framework can exercise smart contracts at scale, multi-threaded, and in isolation. We were able to execute all available smart contracts from the Ethereum mainnet 4.54 times faster in comparison to the standard Ethereum client, Geth. Moreover, we could scale the execution effectively, e.g., on 44 cores our framework runs 50.03 times faster.

Our infrastructure is highly suitable in scenarios that require the fast and repeated execution of smart contracts. We have demonstrated the application of our testing and profiling infrastructure in three use cases: (1) for bytecode metrics, (2) for smart contract fuzzing, and (3) for hard fork compatibility assessment. Our infrastructure scales for the whole blockchain, which has not been possible with prior approaches.

Acknowledgments

We thank our shepherd, Professor Liuba Shrira, and the anonymous reviewers for their insightful comments and suggestions.

This work was supported by Fantom Foundation, by the Australian Government through the ARC Discovery Project funding scheme (DP210101984), by the National Research Foundation of Korea (NRF) funded by the Korean government (MSIT) under Grant No. 2019R1F1A1062576, and by the Next-Generation Information Computing Development Program through the NRF, funded by the Ministry of Science, ICT & Future Planning under Grant No. NRF-2015M3C4A7065522.

Availability

The source code of our record-replay infrastructure is publicly available at https://github.com/verovm/usenix-atc21. The repository contains a download link to the substate database (285 GB) of the initial 9 M blocks of the Ethereum blockchain that has been recorded and replayed as part of this study.

References

[1] Alexey Akhunov. Turbo Geth. https://github.com/ledgerwatch/turbo-geth, accessed 2020-09-22.

[2] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad van Moorsel. The OpBench Ethereum opcode benchmark framework:

Design, implementation, validation and experiments. *Performance Evaluation*, 146:102168, 2021.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

[4] Kirk Baird, Seongho Jeong, Yeonsoo Kim, Bernd Burgstaller, and Bernhard Scholz. The economics of smart contracts, `arXiv:1910.11143 [cs.DC]`, 2019.

[5] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. Understanding the motivations, challenges and needs of Blockchain software developers: a survey. *Empirical Software Engineering*, 24(4):2636–2673, August 2019.

[6] ChainSecurity. Istanbul hardfork EIPs — changing gas costs and more. https://chainsecurity.com/istanbul-hardfork-eips-increasing-gas-costs-and-more/, 2019, accessed 2021-05-10.

[7] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ting Wang, Teng Hu, Xiuzhuo Xiao, Dong Wang, Jin Huang, and Xiaosong Zhang. A large-scale empirical study on control flow identification of smart contracts. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[8] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1503–1520, New York, NY, USA, 2019. Association for Computing Machinery.

[9] CoinDesk. Ethereum's Istanbul upgrade will break 680 smart contracts on Aragon. https://www.coindesk.com/ethereums-istanbul-upgrade-will-break-680-smart-contracts-on-aragon, 2019-10-01, accessed 2021-05-10.

[10] CoinDesk. Soaring DeFi usage drives Ethereum contract calls to new record. https://www.coindesk.com/soaring-defi-usage-drives-ethereum-contract-calls-to-new-record, 2020-07-29, accessed 2021-01-07.

[11] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing Ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78. IEEE, 2019.

[12] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Distributed Comput.*, 33(3-4):209–225, 2020.

[13] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*, pages 1085–1100, Chicago, Illinois, USA, 2017. ACM Press.

[14] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 530–541, 2020.

[15] Ethereum. EIP-1884: Repricing for trie-size-dependent opcodes. https://eips.ethereum.org/EIPS/eip-1884, 2019, accessed 2021-05-10.

[16] Ethereum. Merkle Patricia Tree. https://eth.wiki/en/fundamentals/patricia-tree, accessed 2020-12-10.

[17] Ethereum. Solidity language documentation. https://docs.soliditylang.org, accessed 2020-12-20.

[18] Ethereum. Go Ethereum (Geth) client. https://github.com/ethereum/go-ethereum, accessed 2021-01-05.

[19] Ethereum. Contract ABI specification. https://docs.soliditylang.org/en/develop/abi-spec.html, accessed 2021-05-07.

[20] Etherscan. Ethereum full node sync (archive) chart. https://etherscan.io/chartsync/chainarchive, accessed 2021-01-11.

[21] Etherscan. Ethereum full node sync (default) chart. https://etherscan.io/chartsync/chaindefault, accessed 2021-01-11.

[22] Etherscan. Etherscan API for smart contracts' ABIs. https://etherscan.io/apis, accessed 2021-05-05.

[23] 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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1110–1114, 2019.

[24] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020.

[25] Pieter Hartel and Richard Schumi. Mutation testing of smart contracts at scale. In *International Conference on Tests and Proofs*, pages 23–42. Springer, 2020.

[26] Pieter Hartel and Mark van Staalduinen. Truffle tests for free – replaying Ethereum smart contracts for transparency, arXiv:1907.09208 [cs.SE], 2019.

[27] Anna Irrera. Northern Trust uses blockchain for private equity record-keeping. https://www.reuters.com/article/nthern-trust-ibm-blockchain/northern-trust-uses-blockchain-for-private-equity-record-keeping-idUSL1N1G61TX, 2017, accessed 2021-01-08.

[28] Bo Jiang. Ethereum vulnerable smart contract benchmark. https://github.com/gongbell/ContractFuzzer/tree/master/examples, commit 4e69adb27b42213a045623d37bfe890a20b4ff05.

[29] Bo Jiang, Ye Liu, and WK Chan. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.

[30] John Kolb, Moustafa AbdelBaky, Randy H. Katz, and David E. Culler. Core concepts, challenges, and future directions in blockchain: A centralized tutorial. *ACM Comput. Surv.*, 53(1), February 2020.

[31] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.

[32] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[33] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[34] Ron Miller. IBM teams with Maersk on new blockchain shipping solution. https://techcrunch.com/2018/08/09/ibm-teams-with-maersk-on-new-blockchain-shipping-solution, 2018, accessed 2021-01-08.

[35] Gianmaria Del Monte, Diego Pennino, and Maurizio Pizzonia. Scaling blockchains without giving up decentralization and security: a solution to the blockchain scalability trilemma. In *Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, pages 71–76, 2020.

[36] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[37] Federal Reserve Bank of Boston. Beyond theory: Getting practical with blockchain. https://www.bostonfed.org/publications/fintech/beyond-theory-getting-practical-with-blockchain/building-an-ethereum-blockchain-proof-of-concept.aspx, white paper, 2019, accessed 2020-12-20.

[38] Gustavo Ansaldi Oliva, Ahmed E. Hassan, and Zhen Ming (Jack) Jiang. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empir. Softw. Eng.*, 25(3):1864–1904, 2020.

[39] OpenEthereum. Parity Ethereum client, version 2.4.0. https://github.com/paritytech/parity-ethereum/releases/tag/v2.4.0.

[40] Simone Porru, Andrea Pinna, Michele Marchesi, and Roberto Tonelli. Blockchain-oriented software engineering: Challenges and new directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 169–171, Buenos Aires, Argentina, May 2017. IEEE.

[41] Scott Ruoti, Ben Kaiser, Arkady Yerukhimovich, Jeremy Clark, and Robert Cunningham. Blockchain technology: What is it good for? *Commun. ACM*, 63(1):46–53, December 2019.

[42] Noama Fatima Samreen and Manar H Alalfi. Reentrancy vulnerability identification in Ethereum smart contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 22–29. IEEE, 2020.

[43] Brett Sun. Impact of the Istanbul hard fork on Aragon organizations. https://aragon.org/blog/istanbul-hard-fork-impact, 2019-11-30, accessed 2021-05-10.

[44] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, Petersburg version 3e2c089 – 2020-09-05. https://ethereum.github.io/yellowpaper/paper.pdf, accessed 2020-09-22.

[45] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.

[46] Junfeng Xie, F Richard Yu, Tao Huang, Renchao Xie, Jiang Liu, and Yunjie Liu. A survey on the scalability of blockchain systems. *IEEE Network*, 33(5):166–173, 2019.

[47] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328, 2017.

[48] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing Ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 310–319. IEEE, 2019.

[49] Peilin Zheng, Zibin Zheng, Xiapu Luo, Xiangping Chen, and Xuanzhe Liu. A detailed and real-time performance monitoring framework for blockchain systems. In *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18*, pages 134–143, Gothenburg, Sweden, 2018. ACM Press.

[50] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D. Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering*, 2019.