

FuzzyVM

Guided Differential Fuzzing of Ethereum Virtual Machines

Master thesis by Marius van der Wijden

Date of submission: March 30, 2021

1. Review: Prof. Dr. Max Mühlhäuser
 2. Review: Nikolaos Alexopoulos
- Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science Department
Telecooperation Lab

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Marius van der Wijden, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 30. März 2021

M. van der Wijden

Contents

| | |
|--|-----------|
| Zusammenfassung | 6 |
| Summary | 7 |
| 1 Introduction | 10 |
| 1.1 Challenges | 11 |
| 1.2 Goal | 12 |
| 2 Background | 16 |
| 2.1 Cryptography Basics | 17 |
| 2.1.1 Hash function | 17 |
| 2.1.2 Merkle tree | 19 |
| 2.1.3 Digital Signature Scheme | 20 |
| 2.1.4 Bilinear Pairings | 20 |
| 2.2 Ethereum Basics | 20 |
| 2.2.1 Gas | 23 |
| 2.2.2 Smart Contracts | 23 |
| 2.3 EVM Basics | 24 |
| 2.3.1 Opcodes | 24 |
| 2.3.2 Memory Layout | 25 |
| 2.3.3 Precompiles | 26 |
| 2.3.4 State tests | 27 |
| 2.3.5 Tracing | 28 |
| 2.4 Fuzzing Basics | 28 |
| 2.4.1 Building blocks | 28 |
| 2.4.2 Random Fuzzing | 30 |
| 2.4.3 Guided Fuzzing | 30 |
| 2.4.4 Structured Fuzzing | 30 |
| 2.4.5 Differential Fuzzing | 31 |

| | | |
|----------|---|-----------|
| 2.4.6 | go-fuzz | 32 |
| 3 | Related Work | 34 |
| 3.1 | Fuzzing in the Context of Blockchains | 35 |
| 3.1.1 | EVMFuzz | 37 |
| 3.1.2 | goevmlab | 38 |
| 4 | Generators | 39 |
| 4.1 | Filler | 39 |
| 4.2 | Basic functions | 41 |
| 4.2.1 | Opcodes | 41 |
| 4.2.2 | Storage generators | 42 |
| 4.2.3 | Create and Call | 43 |
| 4.3 | Precompiled Contracts | 45 |
| 4.3.1 | ECRecover | 45 |
| 4.3.2 | Hashes and DataCopy precompiles | 46 |
| 4.3.3 | Modular Exponentiation | 47 |
| 4.3.4 | BN256 Addition | 48 |
| 4.3.5 | BN256 Scalar Multiplication | 48 |
| 4.3.6 | BN256 Pairing | 49 |
| 4.3.7 | Blake2f | 49 |
| 4.4 | Jumps | 50 |
| 4.4.1 | Generation strategies | 51 |
| 5 | Corpus | 53 |
| 5.1 | Corpus size | 53 |
| 5.2 | Corpus elements | 55 |
| 5.3 | Corpus generation | 56 |
| 5.4 | Statistical variation of input length | 57 |
| 6 | Architecture | 58 |
| 6.1 | Strategy 1: Gen+Ex+Vrfy | 60 |
| 6.2 | Strategy 2: Gen/Ex/Vrfy | 61 |
| 6.3 | Splitting generation and execution | 62 |
| 6.3.1 | Strategy 3.1: Gen/Ex+Vrfy linear EVMs | 63 |
| 6.3.2 | Strategy 3.2: Gen/Ex+Vrfy parallel EVMs | 64 |
| 6.4 | Comparison | 65 |

| | | |
|----------|--------------------------------------|-----------|
| 7 | Evaluation | 69 |
| 7.1 | Throughput | 70 |
| 7.1.1 | Methodology | 70 |
| 7.1.2 | Batching | 71 |
| 7.1.3 | Docker | 73 |
| 7.2 | Code coverage | 74 |
| 7.3 | Ground Truth | 77 |
| 7.4 | Discussion | 78 |
| 8 | Conclusion | 80 |
| 8.1 | Research questions | 81 |
| 8.1.1 | Meaningful programs | 81 |
| 8.1.2 | Architectural choices | 82 |
| 8.1.3 | Corpus | 84 |
| 8.2 | EIP-3155 | 85 |
| 8.3 | Issues and Recommendations | 86 |
| 8.4 | Acknowledgements | 90 |
| 9 | Outlook | 91 |
| | Bibliography | 93 |
| | Appendix | 99 |

Zusammenfassung

Ethereum ist nach Bitcoin die zweitgrößte Kryptowährung nach Marktkapitalisierung. Ethereum wurde, anders als Bitcoin, in verschiedenen Programmiersprachen implementiert. Für die Sicherheit und Stabilität des Netzwerks ist es wichtig, dass sich alle Implementierungen des Ethereum-Protokolls auf einen gemeinsamen Zustand des Netzwerkes einigen, da sich sonst das Netzwerk spalten könnte, was es Angreifern ermöglichen würde Gelder doppelt auszugeben.

Ethereum ermöglichte als erstes Blockchain-Netzwerk die Verwendung von Turing-vollständigen Smart Contracts. Diese Smart Contracts sind kleine Programme, die auf der Ethereum-Blockchain ausgeführt werden können und nicht zensierbar sind. Sie werden auf einer Stapelmaschine (stack-machine), der sogenannten Ethereum Virtual Machine (EVM) ausgeführt. Neben normalen Opcodes wie ADD oder SUB sind in der EVM auch komplexere Anweisungen durch sogenannte vorkompilierte Verträge (precompiled contracts) verfügbar. Diese vorkompilierten Verträge implementieren Funktionen wie die Wiederherstellung eines öffentlichen ECDSA-Schlüssels aus einer Signatur oder Hashing mit der SHA3-256 Hashfunktion. Es ist wichtig, dass sich alle Implementierungen der EVM auf die Ausführung eines Vertrages einigen, da es sonst für das Netzwerk unmöglich ist sich auf einen gemeinsamen Zustand zu einigen. Daher sind Regressionen und Updates, die diesen konsenskritischen Teil des Codes beeinflussen, sicherheitskritisch.

Diese Arbeit konstruiert FuzzyVM, einen Code-Coverage basierten differentiellen Fuzzer, der Testfälle erzeugt welche möglichst alle Aspekte der Ethereum Virtual Machine abdecken. Da die vorkompilierten Verträge stark strukturierte Eingaben erfordern, verwendet FuzzyVM Generatoren um passende Eingaben zu erzeugen. Zusätzlich werden Generatoren für Verzweigungsanweisungen verwendet, um die Codeabdeckung zu erhöhen und Timeouts sowie enge Schleifen zu verhindern.

Moderne Fuzzer verwenden oft einen Satz von Testeingaben, den sogenannten Korpus, um interessantere Eingaben zu erzeugen. In dieser Arbeit wird ein Algorithmus zur automatischen Erzeugung von Korpuselementen, die bestimmte Längenanforderungen erfüllen,

vorgeschlagen. Des Weiteren werden verschiedene Architekturen zur Strukturierung von differentiellen Fuzzern, insbesondere mit dem Fokus auf Strategien zur Minimierung von Leistungsunterschieden zwischen Implementierungen, diskutiert. Diese Arbeit stellt verschiedene Evaluierungsmethoden vor mit denen entschieden werden kann ob ein Fuzzer “gut” ist, und evaluiert FuzzyVM anhand dieser Kriterien auf fünf verschiedenen EVM-Implementierungen.

In dieser Arbeit werden verschiedene Probleme diskutiert, die von FuzzyVM gefunden wurden. FuzzyVM fand einen kritischer Fehler in Nethermind, der dazu hätte genutzt werden können, alle Nethermind-Knoten vom Ethereum-Netzwerk zu trennen. Außerdem fand FuzzyVM einen Fehler in dem Besu Client der genutzt werden könnte um alle Besu Knoten im Ethereum Netzwerk gleichzeitig auszuschalten.

Summary

Ethereum is the second biggest cryptocurrency after Bitcoin and has been implemented in different programming languages. For the security and stability of the network, it is important that all implementations of the Ethereum protocol agree on a common state. Otherwise the network could split which would enable attackers to spend funds twice. Another issue is that all nodes that were on the "wrong" side of the split need to re-synchronize with the network, potentially losing hundreds of blocks.

Ethereum was the first blockchain network to include Turing complete smart contracts. These smart contracts are small programs that are executed in a trustless manner on top of the Ethereum blockchain. They are executed on a stack machine called the Ethereum Virtual Machine (EVM). Next to normal opcodes like ADD or SUB, more complex instructions are available in the EVM through so called precompiled contracts. These precompiled contracts implement functions like the recovery of a ECDSA public key from a signature or hashing using the SHA3-256 hashing algorithm. It is important that all implementations of the EVM to agree on the execution of a contract as it would otherwise be impossible for the network to agree on a common state. Thus regressions and updates that impact this consensus-critical part of the code are dangerous to the health of the network.

This work constructs FuzzyVM, a coverage-guided differential fuzzer that generates test cases that try to cover all aspects of the Ethereum Virtual Machine. FuzzyVM employs generators that create inputs for the precompiled contracts as they require highly structured inputs. Additionally generators for branch instructions are used to increase code coverage while preventing timeouts and tight loops.

Modern fuzzers often use a set of test inputs, called the corpus, to create more interesting inputs. In this thesis, a novel algorithm is proposed to create corpus elements that satisfy certain length requirements. It also proposes different architectures to structure differential fuzzers and discusses strategies to mitigate performance differences between implementations. This thesis introduces different evaluation methods used to decide

whether a fuzzer is “good” and evaluates FuzzyVM according to these criteria on five different EVM implementations.

This paper discusses several issues found by FuzzyVM, including a critical bug in Nethermind that could have been used to split all Nethermind nodes from the Ethereum network as well as a bug in the Besu client that could be used to crash all Besu nodes on the Ethereum network simultaneously.



1 Introduction

I don't believe a second, compatible implementation of Bitcoin will ever be a good idea - Satoshi Nakamoto

Since the introduction of Bitcoin and its underlying technology, blockchain, in 2009 by Satoshi Nakamoto, cryptocurrencies have gained significant prominence [45]. Today there are more than 6.000 active cryptocurrencies based on blockchain technology [12]. Blockchain technology allows every participant to validate the state of the network, thus enabling users to verify that no money is created outside of the rules of the system.

In 2013, Vitalik Buterin built on the ideas of Satoshi Nakamoto and proposed Ethereum [18][7]. Ethereum has the ability to execute arbitrary, Turing complete programs, called smart contracts, as well as normal transactions. These contracts are executed in a stack-based Virtual Machine called the Ethereum Virtual Machine (EVM). The EVM is the most critical part of the Ethereum node implementation as it computes the state of the network after a transaction is executed and implements the consensus rules described in Ethereum's yellowpaper [18].

One of the differences between Ethereum and Bitcoin is that there are multiple implementations of the Ethereum protocol [18] with significant user bases. While the Bitcoin network only consists of a single implementation [16], Ethereum has 5-10 competing implementations. Blockchain technologies, as most distributed systems, require all nodes in the system to agree about the current state of the network.

Differences in the EVM implementations could result in splits of the network so called "chain-splits". If a chain-split is resolved, the shorter (weaker) chain is discarded. Some nodes can not handle resolving long chain-splits and would need to resynchronize the network from the genesis block on. Chain-splits also enable malicious actors to perform so called "double-spends", where money is spent twice on both chains. Preventing double-spending is important as it impacts the soundness of the cryptocurrency.

Extensive testing is required to ensure that all EVM implementations perform exactly the same operations. This work proposes a new way to search for differences in these EVM implementations by implementing a generator-based differential fuzzer. A fuzzer creates “random” inputs to a program with the intent to provoke crashes of the program. Generators help to reach deeper states of the implementation as they do not pick the input fully at random, but adhere to certain patterns. Differential fuzzing focuses on finding differences in execution between implementations, not on creating faults or crashes in a single implementation, as these differences could have dire consequences in blockchain networks.

FuzzyVM solely focuses on the transaction execution and does not, for example, test the block propagation code. Thus not the whole node implementations are tested but separate binaries that encapsulate the transaction execution functionality as shown in Figure 1.1.

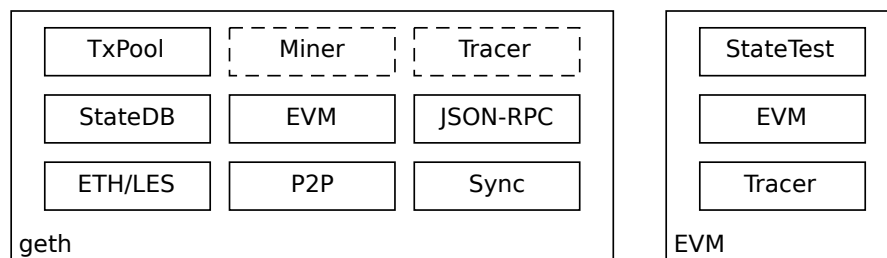


Figure 1.1: The EVM binary encapsulate the EVM functionality

Not testing the whole node implementation provides advantages in speed and efficiency as well as allowing for new clients to be tested without having to implement the block propagation or network synchronization functionalities.

1.1 Challenges

Testing different EVM implementations is important for the security of Ethereum. There have been multiple projects trying find crashes and differences in components of Ethereum protocol implementation and Ethereum Virtual Machines. Static test cases provide a good starting point for new implementations to test the required functionality. However, static test cases can not cover all aspects of the EVM.

Meaningful programs Creating programs for the EVM that try to cover most functionality is difficult. FuzzyVM is based on the learning’s from goevmlab [59] which provides non-coverage guided differential fuzzing for EVMs as well as EVMFuzz which constructed coverage guided differential fuzzing based on the generation of smart contracts in a higher level language called solidity [22][23].

FuzzyVM improves these approaches by using go-fuzz [67], a framework similar to American Fuzzy Lop [70], which provides coverage-guided fuzzing in order to create EVM-bytecode directly without having to go through the solidity compiler. Previous work also focused mostly on the opcodes (like ADD, MUL, SSTORE). The EVM provides, in addition to these, more complex cryptographic functions that need specific input to execute correctly.

Clients The clients under test are written in different programming languages and have different maturity levels. This resulted in varying execution times between implementation. A single test can run between one and five seconds, which makes testing and benchmarking a sufficient number of tests time-consuming.

Testing clients in different languages also imposes limitations on FuzzyVM as the EVM functionality can not be called directly. Fortunately most clients provide separate binaries that encapsulate the EVM functionality and allow the execution of tests. They output a trace for every opcode that is executed, thus providing a way to find differences in execution between the EVM implementations.

1.2 Goal

This thesis tries to answer the research questions posed in Figure 1.2 by constructing a guided differential fuzzer for EVM implementations. The findings of this thesis can be applied to fuzzing efforts of other smart contract platforms as well as non blockchain-related fuzzers.


- 
-
1. How can a fuzzer create meaningful programs to test Turing-complete Virtual Machines and cryptographic functions?
 2. How can the quality of a differential fuzzing framework be evaluated, and how do architectural choices impact it?
 3. How much benefit can an existing corpus bring and how can the quality of an initial corpus be improved?

Figure 1.2: Research questions

How can a fuzzer create meaningful programs to test Turing-complete Virtual Machines and cryptographic functions?

The Ethereum Virtual Machine enables execution of Turing-complete programs. This thesis proposes different strategies to create programs that test as much functionality of the EVM as possible.

Infinite loops The EVM supports loops with the opcodes `JUMP` and `JUMPI`. While it has a mechanism to prevent infinite loops called gas, the output of a program with a tight loop can grow to several megabytes. Some EVM implementations can not handle such large outputs and hang indefinitely. Additionally these infinite loops often execute very little code which reduces the efficiency of the fuzzer.

Previous fuzzers for the EVM either banned the use of `JUMP` instructions, imposed an upper limit on loop executions in a higher level language that gets compiled to EVM bytecode [22] or only jumped forwards[59]. Section 4.4 introduces generators that generate code with jump instructions to a certain extent, but disallow or minimize the occurrence of infinite loops. These generation techniques can be used in the future for all fuzzers that are used to test Turing-complete stack machines.

Precompiled contracts In addition to opcodes like `ADD` or `POP` the EVM also provides complex cryptographic functions to the program. These functions are called precompiled contracts or precompiles. A list of precompiles in Ethereum can be found in Figure 2.8.

A generator for a cryptographic function has to take the particularities of the function into account, otherwise the input would be marked invalid early on and most code would not be executed. A fully random mutator would create a valid ECDSA signature, that can be recovered by the ECRrecover precompile, only with negligible probability for example.

FuzzyVM employs fully random mutators for the precompiles to test the preconditions as well as more complex generators to cover as much code as possible. These generators can be used in the future to fuzz the cryptographic functions in standalone implementations as well as to generate an unlimited amount of integration tests for projects that want to re-implement these cryptographic functions.

How can the quality of a differential fuzzing framework be evaluated, and how do architectural choices impact it?

It is important to evaluate architectural choices of differential fuzzing frameworks due to their complexity. Section 7 discusses three metrics (execution speed, code coverage and ground truth) to evaluate differential fuzzing frameworks and evaluates FuzzyVM accordingly.

Architectural choices Due to the different languages in which the EVMs are implemented, execution time of the same test varies significantly between implementations. This thesis focuses on the five major implementation that are currently able to verify the full Ethereum network. Figure 1.3 shows the execution time of the five major implementations.

| Implementation | Executed in (ms) | User time (ms) |
|----------------|------------------|----------------|
| go-ethereum | 31 | 21 |
| OpenEthereum | 17 | 11 |
| Nethermind | 926 | 463 |
| Besu | 1540 | 5790 |
| Turbogeth | 200 | 184 |

Figure 1.3: Execution time of a typical state test

Besu is written in Java, which means a Java Runtime Environment has to be spun up for every single test which impacts performance significantly. In a naive architecture where all clients are started on each test, this varying execution time results in the test being very slow since all clients need to wait for the slowest client to finish. Besu implements

a batching mode which allows executing multiple tests on the same instance. Chapter 7.1.2 explores different strategies to reuse the same instance of the EVM for multiple tests which increases the throughput of FuzzyVM.

Executing batches instead of single test cases poses challenges to the overall architecture as tests have to be grouped together and execution results have to be handled differently. Additionally the structure of the fuzzing framework impacts how good go-fuzz can “learn” the test target. Chapter 6 explores multiple architectures to support these batching strategies to accommodate different clients with varying execution times. The findings from Chapter 6 are applicable to most differential fuzzing efforts, especially when the difference in runtime between the test targets is significant.

How much benefit can an existing corpus bring and how can the quality of an initial corpus be maximized?

Modern fuzzing frameworks like AFL and go-fuzz maintain a set of known “interesting” inputs called corpus. Generated elements are added to the corpus if they increase the code coverage when executed by the test target. Section 5 explores how choosing a good initial corpus can help the fuzzer to create more interesting inputs.

Since FuzzyVM creates programs from the provided input, the input has to have a certain size which depends on the input itself. Hand crafting corpus elements that fit the needs of FuzzyVM is difficult. Thus Section 5 proposes an algorithm to automatically produce valid corpus elements. The algorithm can be applied to all fuzzing efforts where the optimal size of the input is dependent on the input itself.

2 Background

Ethereum and the Ethereum Virtual Machine are specified in the Ethereum Yellowpaper [18]. It is a description of all features needed to build a working implementation of Ethereum and has been updated continuously. This description allows teams to implement Ethereum nodes in different programming languages. There are EVM implementations for almost every major programming language. An inconclusive list can be found in table 2.1.

The variety of implementations creates a more resilient network, as faults in one implementation might not affect the whole network. In 2016, the Ethereum network could withstand a bug in go-ethereum because Parity managed to produce valid blocks, so the network did not stop while the bug was fixed [53]. Having different implementations of the same protocol also allows for testing them against each other to find deviations from the specification.

| name | language | timeframe | marketshare [24] |
|-----------------|----------|-------------|------------------|
| go-ethereum | golang | 2013 | 81.9 |
| OpenEthereum | rust | 2019 | 11.1 |
| parity-ethereum | rust | 2013 - 2019 | 3.6 |
| Nethermind | .NET | 2017 | 1.5 |
| TurboGeth | golang | 2020 | 0.9 |
| Besu | Java | 2018 | 0.6 |
| EthereumJ | Java | 2014 - 2019 | 0.1 |
| teth | rust | 2019 - 2019 | 0.1 |
| cpp-ethereum | C++ | 2014 - 2020 | 0 |
| Trinity | Python | 2017 | 0 |
| CoreGeth | golang | 2015 | 0 |
| MultiGeth | golang | 2018 | 0 |

Figure 2.1: Table of client implementations

While the Ethereum Foundation maintains a list of tests-cases for implementations that should ensure a bug free implementation of the yellow paper [20], this list cannot cover all edge cases that might occur in an implementation nor can it cover language specific or implementation specific details. There have been multiple bugs in EVM implementations in the past that resulted in chain splits. In 2016, a critical bug in go-ethereum caused a rollback of several days of transactions as well as multiple days of uncertainty on the network [8]. Another example is a bug in go-ethereum where a newly introduced library panicked on divisions by zero [57]. These bugs could have easily been prevented by a differential fuzzer that is continuously run on every release of the clients.

Differential fuzzing is also important to test future upgrades. For example, non-guided differential fuzzing was used to identify multiple problems in implementations of BLS-Signatures (Boneh–Lynn–Shacham) [5] for a future upgrade of Ethereum [68].

2.1 Cryptography Basics

Ethereum, like all cryptocurrencies, relies heavily on cryptography. Since we want to use fuzz-testing to test some of these aspects, we need to have a basic understanding of the schemes and algorithms used in Ethereum. This section presents cryptographic primitives used in blockchain-based systems.

2.1.1 Hash function

A cryptographic hash function is a compressing function that maps arbitrarily large inputs to a fixed size output [35].

Cryptographic hash functions have several important features:

- The function should be deterministic, always returning the same output for an input.
- Calculating the input from the output should be hard.
- Calculating two different inputs that result in the same output should be hard.
- A small change in the input should result in a big change in the output.

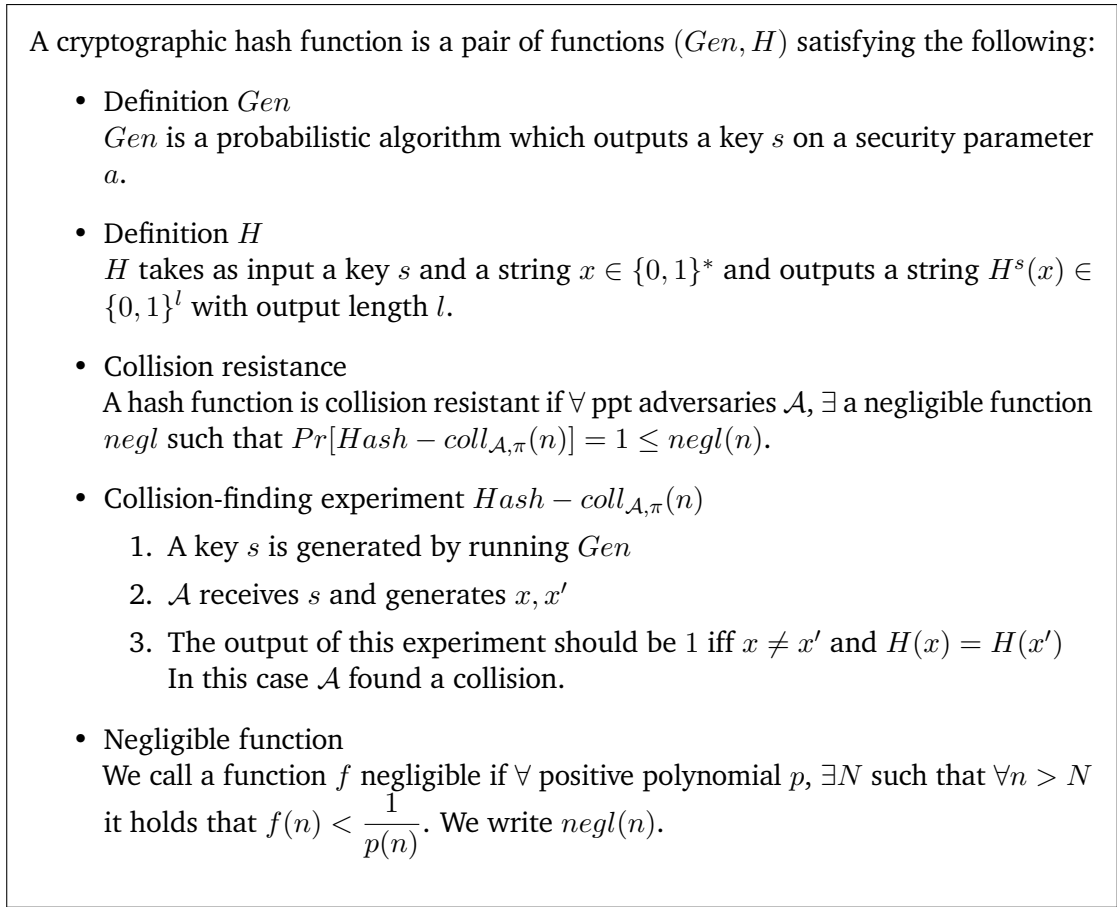


Figure 2.2: Cryptographic hash function scheme [35]

The definitions in Figure 2.2 are based on “Introduction to Modern Cryptography” by Katz and Lindell [35]. Some of the definitions have been slightly altered to ensure better readability.

Ethereum uses the hash function Keccak256. A slightly altered version of Keccak256 was later standardized as SHA3-256. However Ethereum uses the original version making it incompatible with the standardized version. The EVM provides the Keccak256 hash function as an opcode. It also provides a precompiled contract for SHA3-256 as well as the Blake2 compression function F which can be used to implement different schemes of

the BLAKE2 family of hash functions, and the RIPEMD160 hash function.

2.1.2 Merkle tree

A merkle tree is a data structure which is heavily used in blockchain technology because it is easy to verify the integrity of a dataset as well as prove membership of a value. It is a binary tree in which the leaves of the merkle tree contain hashes of the datasets, e.g. $H(data(k))$. The non-leaf nodes of the tree contain the hash of the left child and the right child concatenated and hashed, e.g. $H(left||right)$ with $left = H(x)$ and $right = H(y)$. Since cryptographic hash functions with the properties discussed in Figure 2.2 are used, merkle tree's as shown in Figure 2.3 have the following properties.

- The root changes whenever a single bit of one of the datasets in one of the leaves changes.
- If a leaf changes of the merkle tree changes, only parts of the tree have to be recalculated, which is faster than recalculating the hash of all the nodes.
- A change in the set can be proven by recalculating the merkle root.
- It can be proven that an element is in the set, by providing the path inside the tree that connects the element to the merkle root. Since it is hard to compute a different path that results in the same root, we can be sure that the element is in the set. This path is called a merkle path.

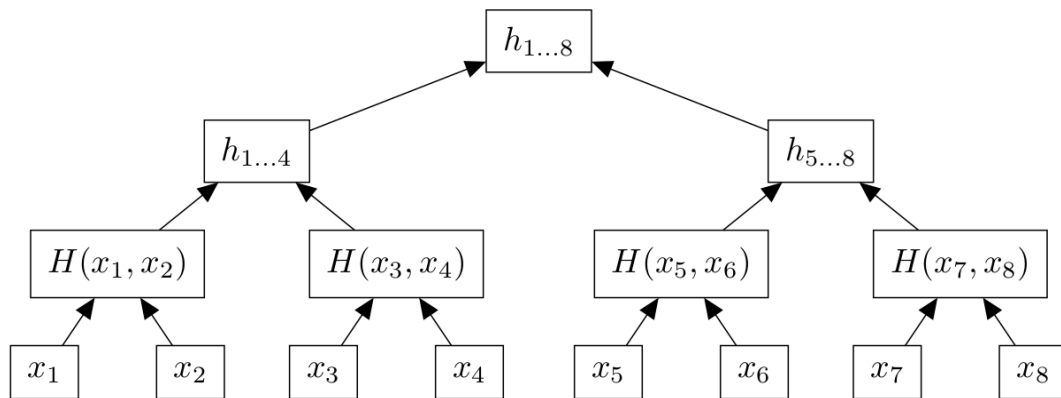


Figure 2.3: A merkle tree for eight elements [35]

Ethereum uses a similar scheme, merkle-patricia trees (hexary merkle-trees with special extension nodes), to compute the state root which is the root of the current state of the network. All changes to the state of the network also change the state root.

2.1.3 Digital Signature Scheme

Digital Signature Schemes are used to verify the authenticity of data. It is a primitive similar to normal signatures used in the analog world. Users can sign documents and verify signatures of others on a document. A digital signature scheme consists of three algorithms (Gen , $Sign$, $Vrfy$) that run in polynomial time. The definitions in Figure 2.4 are based on “Introduction to Modern Cryptography” by Katz and Lindell [35]. Some of the definitions have been slightly altered to ensure better readability.

Ethereum uses ECDSA-Signatures on a secp256k1 curve for signing transactions and provides signature verification in the ECRrecover precompile.

2.1.4 Bilinear Pairings

Bilinear Pairings are extensively used in cryptography to reduce problems to easier subgroups. A pairing function e maps elements of one group to another group. They can be used to map hard problems to another group where the problems are easier to compute. Figure 2.5 provides a definition for bilinear pairings which is loosely based on “An Introduction to Pairing-Based Cryptography” [43].

In Ethereum the BN256 pairing precompile implements an Optimal-Ate pairing over a Barreto-Naehrig curve. This algorithm was proposed by Naehrig, Niederhagen and Schwabe [44] however Ethereum uses different parameters compared to the original scheme. Additionally a Tate pairing over the Boneh–Lynn–Shacham curve BLS-12381 [4] is considered to be included in Ethereum with the next update [65]. These pairings allow for verification of aggregated signature schemes in Ethereum.

2.2 Ethereum Basics

A basic understanding of the components and schemes involved in the EVM is needed to implement a fuzzer for Ethereum Virtual Machines. This section introduces the basic

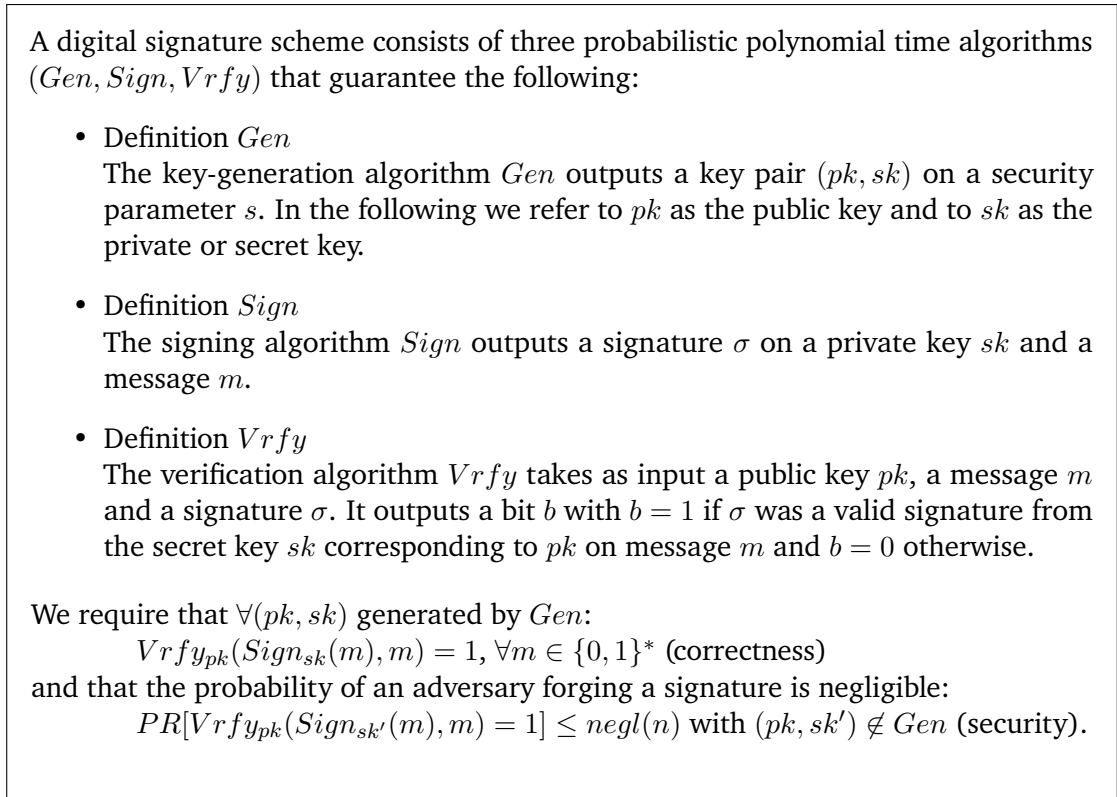


Figure 2.4: Digital signature scheme [35]

building blocks of Ethereum which are necessary to understand the need for differential fuzzing.

Consensus All nodes of a blockchain network need to agree on the current state of the network. Ethereum currently uses a Proof-of-Work consensus algorithm similar to Bitcoin to achieve consensus between the nodes. Hereby cryptographic puzzles are solved to decide who can propose a new block of transactions. If a block reaches a node, all transactions in the block have to be executed locally to check if the block was created correctly. It is crucial for the consensus that all nodes calculate the same state as they might otherwise discard a block as invalid.

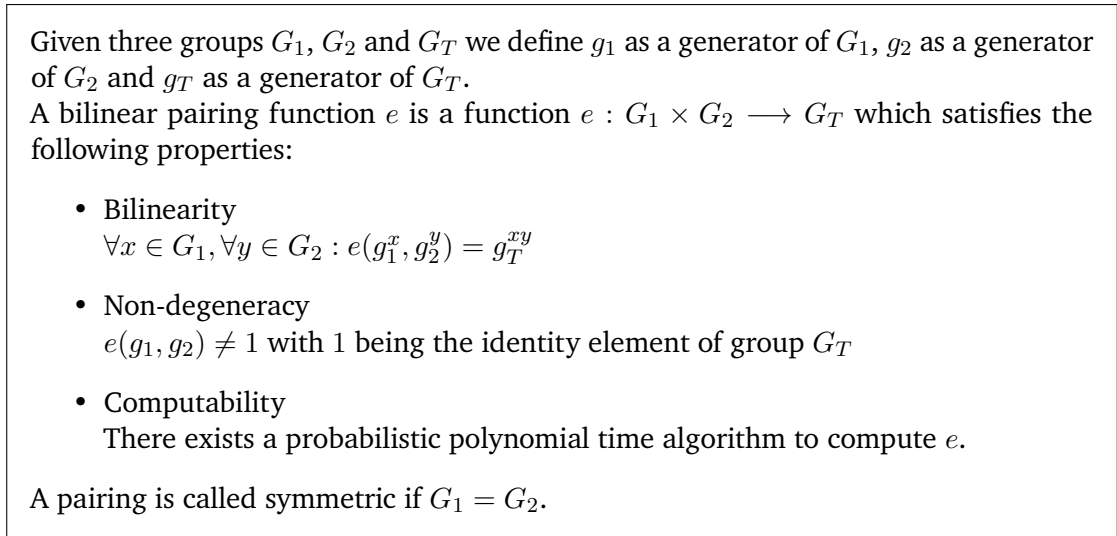


Figure 2.5: Bilinear pairing function

Ether The inherent currency of Ethereum is Ether. It is used to pay for sending transactions and executing Smart Contracts. Ether is denominated in eth (or ETH for the currency used in the official Ethereum network) which is equal to 10^{19} Wei, the smallest unit of the Ethereum network. Gas prices are usually between 20 and 100 Giga-Wei or GWei on the Ethereum mainnet [56].

Block format A block in Ethereum consists of a block header, the transactions and optionally uncle headers. The format of the block headers is described in the Appendix, Figure 9.4. For EVM-testing the StateRoot is the most important part of the block header. The StateRoot is the merkle root of the state after all transactions in the block have been applied. The state is stored in a tree format called the state trie. Being a merkle tree, all state changes influence the StateRoot. If two EVM's calculate the StateRoot differently, a split in the blockchain occurs since one node will not accept the blocks of the other node anymore.

2.2.1 Gas

Ethereum has a property called gas in order to prevent denial-of-service (DoS) attacks. A user has to send gas with every transaction to prevent them from sending too many transactions. Every instruction executed in the EVM costs gas (`GasCost`). This prevents users from creating programs that run indefinitely as they will run out of gas eventually. A simple transaction that only transfer Ether and does not call a contract costs 21.000 gas.

Gas is also closely connected to the transaction cost. A user has to specify how much they are willing to pay for a single unit of gas (`GasPrice`). The total cost of a transaction can be calculated as $\text{TotalCost} = \text{GasCost} * \text{GasPrice}$. Since the sum of all gas used in a block is also part of the block header, as shown in Figure 9.4, the gas cost and gas usage are consensus critical.

The user can also specify how much gas they are willing to spend which is called the `GasLimit`. If a transaction runs out of gas during execution ($\text{GasCost} > \text{GasLimit}$) the execution is stopped and all changes are reverted. The miner can include the transaction even if it ran out of gas and get $\text{GasLimit} * \text{GasPrice}$ as compensation.

A block also has a `BlockGasLimit` which specifies the amount of gas that can be included into a block. This prevents malicious miners from mining huge blocks that can not be verified in the 13 second blocktime of Ethereum. The `BlockGasLimit` can only be increased slowly with each block.

2.2.2 Smart Contracts

Smart Contracts have first been introduced by Nick Szabo in 1997 [60]. They are small programs that provide functions that can be called via transactions and are executed by all nodes in the network. While Bitcoin provides a small Virtual Machine that can execute code, Bitcoin contracts are not Turing complete by design. Thus, a lot of problems cannot be solved by Bitcoin contracts.

Ethereum enables Turing complete Smart Contracts. They are usually written in a high level programming language called Solidity which is then compiled to EVM bytecode and can be executed by the Ethereum Virtual Machine (EVM). A user can deploy a contract by sending a transaction with a special code and the bytecode of the program to the zero address. The contract bytecode is saved under a newly created address. A user can interact with the contract by sending a transaction to the address of the contract. Contracts can

send and receive Ether, write and read from their storage and call other contracts. Smart Contracts can not, however, initiate transactions.

2.3 EVM Basics

The Ethereum Virtual Machine is a stack-based machine that can execute Ethereum bytecode. This bytecode contains both operands, the so called “opcodes”, as well as data. If a contract gets deployed, the bytecode of the contract is stored in the databases of the Ethereum nodes. A user can trigger a bytecode by creating a transaction with the function name that they want to call as well as arguments to the call. The EVM loads the bytecode from the database and executes the called function. The EVM has a word-size of 256 bits.

The user has to specify a maximum amount of gas that a transaction can use with every transaction. Every opcode strictly decreases the amount of gas left during execution¹. Once the amount of provided gas reaches zero, the transaction runs “out of gas” and is canceled. If a transaction fails, all state changes of the transaction get reverted. The miner that included the transaction still receives the transaction fees that the user sent, even though the transaction failed.

2.3.1 Opcodes

Opcodes are the operands of the EVM. Table 9.2 in the Appendix gives an overview of all opcodes as well as their respective gas cost. Some opcodes require a formula to compute their gas cost. This is often the case if an opcode is dependent on the amount of data that is provided to the function. For example the KECCAK256 opcode takes 30 gas + 6 gas for every 256-bit word that should get hashed with the keccak256 hash function.

Opcode ordering Calls from a smart contract to other smart contracts and precompiles can fail without reverting the whole transaction. This allows contracts to call another contract and continue execution even if the call fails. Since the gas can be limited and run out between every step, the ordering of the execution of opcodes is consensus critical too. For example a contract that does some computation and then calls another contract which fails, the function might still execute correctly. If an EVM reorders the opcodes

¹Except for the STOP and RETURN opcodes that halt execution.



```
\texttt{SSTORE} // Executed
CALL // Fails and burns 63/64ths of all available gas
// OR
CALL // Fails and burns 63/64ths of all available gas
\texttt{SSTORE} // Fails because the remaining gas is too little to
execute
```

Figure 2.6: Example: Opcode reordering impacts consensus

to call the contract first and then do the computation, the function will most likely fail. Figure 2.6 visualizes this example.

2.3.2 Memory Layout

The Ethereum Virtual Machine has access to several different storage segments [18] which are visualized in Figure 2.7.

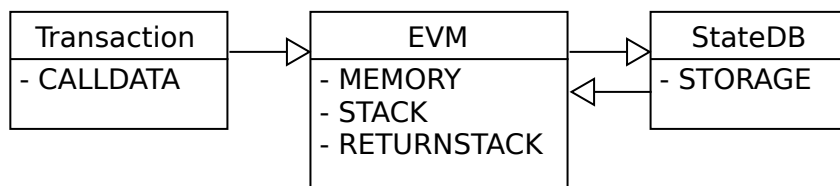


Figure 2.7: The different storage segments available to the EVM

CALLDATA Users can send arbitrary data with their transaction that may start functions on a smart contract. This data is called the CALLDATA segment. Data from the CALLDATA segment can be moved to the MEMORY segment by either executing the MSTORE, MSTORE8 opcodes or calling the identity precompile as described in Section 4.3. The CALLDATA segment is read-only for the EVM.

Stack The EVM is a stack-based Virtual Machine. The stack has a maximum size of 1024 elements. Elements can be added to the stack using the PUSHX opcodes with $X \in \mathbb{Z}_{32}$ whereby X denotes the amount of items to be added. Elements can be removed from the

stack with the POP opcode. Additionally the DUPX opcodes with $X \in \mathbb{Z}_{16}$ can be used to duplicate the X 'th item of the stack. Finally the SWAPX opcodes with $X \in \mathbb{Z}_{16}$ can be used to swap the first and the $X + 1$ 'th item of the stack.

MEMORY The MEMORY segment is a part of the EVM that allows for storing and loading data during execution. Data can be written to the MEMORY segment with the MSTORE and MSTORE8 opcodes and read with the MLOAD opcode. The MEMORY segment is non-persistent, therefore everything stored in the MEMORY segment is deleted after every transaction.

STORAGE The STORAGE segment allows for persistent storage. It can be written with the SSTORE opcode and read with the SLOAD opcode. Since this data has to be stored by all nodes the SSTORE opcode is the one of the most expensive opcodes. Additionally setting a used storage slot to zero gives the user a refund as zero slots do not need to be saved anymore. The STORAGE segment is saved in the state trie after successful execution thus the StateRoot changes whenever a transaction changes the STORAGE segment. The root of the state trie (StateRoot) is part of the block header thus SSTORE opcodes influence the consensus directly.

RETURNSTACK A new Ethereum improvement proposal EIP-2315 introduces a subroutine mechanism which introduces another storage segment called RETURNSTACK. It stores the current ProgramCounter when executing the new JUMPSUB opcode. This allows a subroutine to return to the ProgramCounter when a subroutine is finished. The RETURNSTACK can store up to 1024 elements. EIP-2315 was scheduled for the next hardfork in Ethereum called Berlin. However it was removed from the Berlin hardfork due to concerns of its usefulness for higher level smart contract languages (like solidity).

2.3.3 Precompiles

All opcodes are 1 byte in size which limits the amount of different opcodes that can be assigned. This historic limitation encouraged the core developers to create the so-called precompiled contracts of precompiles. The precompiles are contracts that execute more complex functions. They are located under the addresses $0x00..01 - 0x00..09$ and can be called similar to normal smart contracts.

Figure 2.8 lists all currently available precompiles on Ethereum as well as their gas usage. In order to call a precompile the contract has to put parameters for the precompile as well as the address on the stack and then use the CALL or STATICCALL opcode to execute the precompile. The result of the execution as well as a success flag are pushed on top of the stack by the precompile.

| Address | Name | Gas formula |
|---------|----------------|----------------------------------|
| 1 | ECRecover | 3000 |
| 2 | SHA256 | $ input * 12 + 60$ |
| 3 | RipeMD160Hash | $ input * 120 + 600$ |
| 4 | DataCopy | $ input * 3 + 15$ |
| 5 | ModExp | complex formula ² |
| 6 | BN256Add | 150 |
| 7 | BN256ScalarMul | 6000 |
| 8 | BNPairing | $34000 * k + 45000$ ³ |
| 9 | Blake2F | $Rounds * 1$ |

Figure 2.8: List of EVM precompiles

2.3.4 State tests

The Ethereum Foundation manages a set of test cases that can be used for new implementations to test proper execution of the chain rules [20]. These tests are written in a special JSON format which is described in the official documentation [14]. There are different categories of state tests for different areas of the code.

The tests relevant to this thesis are the “State Transitions Tests” which test the execution of a transaction. They contain all relevant information needed to execute and verify a transaction. The prestate contains the code and balance of all available accounts before execution, a transaction that should be executed and multiple other fields like the current gas limit. The poststate contains the hash of the resulting logs and the StateRoot which

² $\text{floor}(\text{multcplx}(\max(|mod|, |base|)) * \max(ADJEXP, 1) / 20)$
with $\text{multcplx}(x)$: if $x \leq 64$: return $x ** 2$; elif $x \leq 1024$: return $4 + 96 * x - 3072$; else: return $16 + 480 * x - 199680$;
and $ADJEXP$: if $\text{exponent} = 0$; return 0; elif $|\text{exponent}| \leq 32$; return q ; else return $8 * (|\text{exponent}| - 32) + q$
with q as the index of highest bit of exponent

³ k denotes the number of pairings being computed

is the merkle hash of the state trie. Figure 9.7 in the Appendix provides an example for the format of a state test. FuzzyVM creates tests in the state test format as tooling for this format is readily available.

2.3.5 Tracing

In order to debug differences in execution between EVMs faster, most implementations provide a way to trace transactions. Tracing reports a summary about the current state of the execution for every opcode that is executed by the EVM. It contains, amongst other things, the program counter, the remaining gas and the gas cost of the next transaction. The tracing formats differ between implementations. In order to compare them, the fields `GasCost`, `MemorySize`, `RefundCounter`, `ReturnStack`, `ReturnData` and `Memory` have to be dropped from the output. As a consequence FuzzyVM cannot detect differences in these fields.

Therefore, Martin Holst Swende and I proposed an EIP (Ethereum Improvement Proposal) [40] which encourages the clients to consolidate their tracing formats to a common format. This common format allows for simpler testing infrastructure and increases the surface that can be covered by FuzzyVM. It is described in Section 8.2. An example for the format of a trace is provided by Figure 9.8 in the Appendix.

2.4 Fuzzing Basics

Fuzz-testing or fuzzing is the act of executing a program on random inputs and observing its behaviour [39]. It can provide better coverage than manual tests. Therefore, fuzzing is extensively used to test libraries and cryptographic function. There are different strategies to maximize the coverage of a fuzzer. This section describes building blocks and strategies commonly used by fuzzers.

2.4.1 Building blocks

Fuzzers often consist of different building blocks. These building blocks often define the strategy of the fuzzer.

Test target The program to be fuzzed is called the test target or client under test. Test targets can be either simple functionalities or standalone programs.

Source of randomness Fuzzers need a source of randomness to create test cases. Early fuzzers used the built-in randomness provided by the operating system, like `/dev/random`. These sources of randomness can exhaust and provide bad randomness or even block if too much data is read. Additionally these sources are non-deterministic which makes it hard to calculate the randomness used in a test. Most modern fuzzers use a pseudo-random number generator to generate deterministic random inputs. By doing so users can specify seeds to better control the fuzzer during benchmarks and create large scale fuzzing operations.

Mutators The naive approach of using purely random data as input to the the test targets often proves inadequate. Most modern fuzzers have mutators built in that mutate random data to have certain patterns. This increases the chance of triggering interesting code paths. On a byte-level, multiple consecutive `0x00` and `0xFF` bytes might be interesting inputs or non-null terminated strings. Mutators can also take into account previous inputs and mutate them with operations like bit flips or shifts. A list of mutators used in go-fuzz is provided in Figure 9.1 in the Appendix.

Generators Most complex functions need inputs with very specific structures. Generators can help reach deeper states by taking the input from the mutator and putting them in the structure needed by the complex function. Generators are specific to the algorithm under test while mutators mutate elements after generic mutation rules.

Generators are primarily used in structured fuzzers. In literature the terms generators and mutators are sometimes used synonymous. This work will discriminate between generators and mutators.

Sometimes overfitting the inputs to the needed structure might also hinder a fuzzer from generating good edge cases. FuzzyVM provides, next to specialized generators, also generators that create unstructured calls to test the code that verifies the structure.

Corpus A corpus is a set of exemplary inputs. These inputs help the mutator to generate interesting inputs as they can be used as a basis for new inputs. Often mutators will also merge inputs from the corpus together to create new inputs.

There are two different types of corpi; static and dynamic corpi. A static corpus does not change over the runtime of a fuzzer. A fuzzer with a dynamic corpus can choose to add interesting inputs back into the corpus, creating a feedback loop which helps the fuzzer create better test inputs as shown in Figure 2.9.

It is important for the corpus to be as small as possible as duplicated items generally decrease the efficiency of the fuzzing effort. Providing an initial corpus can help the fuzzer to create interesting inputs quicker than if no initial corpus is provided.

2.4.2 Random Fuzzing

Early fuzzers used a source of randomness directly, to fuzz the test target. This technique yields decent results on simple targets as it can create tests extremely fast. However it can not distinguish between good and bad test cases. Thus, purely random fuzzers can not “learn” to create better inputs.

2.4.3 Guided Fuzzing

Most modern fuzzing frameworks use guided fuzzers in order to increase coverage. These fuzzers instrument the test target with instructions to measure code coverage. Using the code coverage, guided fuzzers try to improve the coverage by mutating the inputs. Additionally, most guided fuzzers use a dynamic corpus and store test cases that increase coverage. This creates a feedback loop which helps improve the quality of test inputs as shown in Figure 2.9. Guided fuzzing is usually more efficient than purely random fuzzing as the fuzzer “learns” which inputs yield better results consequently increasing the probability of creating interesting inputs in the future.

2.4.4 Structured Fuzzing

Structured fuzzing, also called generator-based fuzzing takes into account the structure of the input to create test cases that reach deeper states. Often implementations need a certain structure. These structures are usually checked in the preprocessing steps before

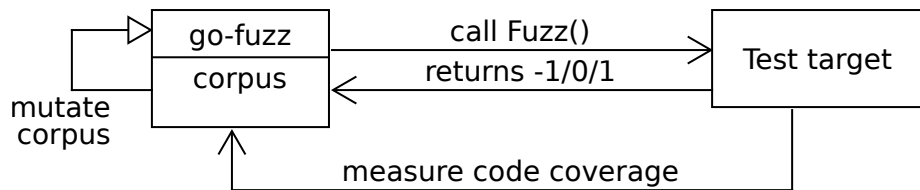


Figure 2.9: A feedback loop allows go-fuzz to learn about interesting mutations

executing the actual algorithm. Non-structured fuzzers, like AFL or LibFuzzer, often create inputs that do not pass these initial checks, thus the actual algorithm is not properly tested [32]. One of the go-ethereum fuzzers, for example, that used non-structured fuzzers to test the BN256 precompiles had a chance of creating a valid test of roughly 1 in 80 million [58]. It was replaced by a structured fuzzer that creates a valid test on every execution.

2.4.5 Differential Fuzzing

Most fuzzers try to find exceptions or critical flaws like unsafe memory accesses. Differential fuzzing is a technique to test different implementations of the same protocol against each other to find differences in behaviour. It is a powerful technique as it can not only find panics or critical flaws but it can also find differences in execution which points to differences between one of the implementations and the specification.

Differential fuzzing can also find unspecified behaviour if two implementations implement the behaviour differently [69]. As a result, they are often used to test different implementations of cryptographic functions and network protocols [52][50].

Differential fuzzers are also important for the consensus-critical code in blockchains as differences can result in consensus splits. This type of fuzzing comes with the drawback that it is more complex than other approaches and slower as the test cases need to be executed on different targets. Contrary to other fuzzers, differential fuzzers work best when the targets produce output data. This data can be compared against each other to find differences.

Language differences The targets for a differential fuzzer are often written in different programming languages and are standalone programs. Therefore, a shared input and

output format is needed to test the implementations. This format should define the data types so that new implementations can be added to the differential fuzzers if they implement the format. The specification of the input and output formats should be unambiguous. This reduces the logic needed by the fuzzing framework to bring the output of the implementation to a common format before verifying it.

The EVM implementations consume the state test format as discussed in Section 2.3.4. They output a JSONI object for every executed operation in a special tracing format ⁴. An example for the tracing format can be found in Figure 9.8 in the Appendix.

Another problem when fuzzing implementations against each other are execution times and resource consumption. The same logic written in different programming languages can result in different execution times. Compiled languages are typically faster than interpreted and require less resources to execute. Different algorithms with different characteristics can be used to implement the same logic and the standard libraries differ significantly between languages [49].

Section 6 discusses architectures for the fuzzers that can alleviate drawbacks originating from these differences.

2.4.6 go-fuzz

Go-fuzz is a fuzzing framework specially designed to test implementations written in golang [67]. It implements a coverage-guided fuzzing framework and implements similar mutators as American Fuzzy Lop [70]. These mutators are for example bit flips, splicing two test cases together or replacing a byte with 0xFF. A list of all mutations performed by go-fuzz can be found in Figure 9.1 in the Appendix.

In order to use go-fuzz, the fuzzing target has to implement a function `Fuzz()` with the signature shown in Figure 2.10 which defines the entrypoint for go-fuzz. During execution, this function is called with the data generated and mutated by go-fuzz as shown in Figure 2.9. Go-fuzz uses a feedback mechanism in order to “learn” which mutations are more interesting than others. Developers can help go-fuzz create better inputs by specifying return values as described in Figure 2.10.

⁴Documentation for the JSONI format can be found at <https://jsonlines.org/>

The entrypoint for go-fuzz has the following function signature:

```
func Fuzz(data []byte) int
```

The developer of the client under test can help go-fuzz by returning an integer from the *Fuzz()* function.

- If *Fuzz()* returns -1, the test cases is not added to the corpus.
- If *Fuzz()* returns 0, the test case is added to the corpus if it increased code coverage.
- If *Fuzz()* returns 1, the test cases is likely to be added to the corpus.

Figure 2.10: Entrypoint signature and return values

3 Related Work

The topic fuzz-testing has been discussed extensively in recent years as an addition to unit and integration testing [52] [29]. This chapter provides an overview of related works with regards to fuzzing especially in the context of blockchain technology.

The Art, Science, and Engineering of Fuzzing: A Survey [39] proposes a taxonomy for fuzzers and differentiates between black-box, white-box and gray-box fuzzing. White-box fuzzers analyze the source code of the program under test and generate test cases based on their knowledge while black-box fuzzers can only observe the behaviour of the program. A gray-box fuzzer leverages some information about the program. They do not analyze the source code directly, but usually instrument the binary to collect information about branch usage and code coverage.

A lot of gray-box fuzzing frameworks have been developed to create intelligent “random” inputs for programs. The widely used gray-boxfuzzing frameworks today are American Fuzzy Lop(AFL) [70], libFuzzer [38], CryptoFuzz [66] and go-fuzz [67]. They start with a set of inputs called corpus and mutate the inputs using operations like bit flips to find interesting test cases. They provide the test cases to the target under test and measure the resulting statements that have been executed. If a test case increases the code coverage, it is added to the corpus.

Differential Fuzzing The term “differential testing” was first introduced in 1998 [42] to describe the act of testing two implementations against each other. It has been used for example to uncover side channels in security critical software [46]. Differential fuzzing is also heavily used in the industry to find disparities between different implementations of cryptographic primitives [34].

Continuous Fuzzing Continuous fuzzing is the technique of running a fuzzer continuously and parallel to your normal continuous integration pipeline. This method can find bugs and regressions as soon as they are introduced [54]. Running a fuzzer continuously can also increase the test coverage significantly. Google created an internal continuous fuzzing infrastructure for the chrome project which was later made available to the public as OSS-Fuzz [31]. OSS-Fuzz has found over 20.000 bugs in 300 Open Source projects [31] by continuously fuzzing every new version. It also provides an optional feature to fuzz incoming pull request to provide regression testing before a change is merged into the codebase [30].

Ensemble Fuzzing Another interesting approach to fuzzing is ensemble fuzzing. Hereby multiple fuzzers with different mutators are run against the same implementation. Different fuzzers might implement different mutation strategies, like depth-first or breadth-first searching [11]. Ensemble fuzzing also increases the chance to reach deeper states in the test target thus finding bugs faster than with individual fuzzers [10]. Ensemble fuzzing is especially useful if the fuzzers share a corpus of test inputs.

Guided fuzzing in go In the last years, go-fuzz [67] has established itself as the fuzzing framework for go code. Go-fuzz is a coverage-guided fuzzer that instruments the test target and tries to maximize the reachable code by mutating its inputs. It implements a lot of mutation strategies from AFL and provides additional strategies. A corpus of inputs can be provided to the mutator to increase the chance of it generating useful inputs. If go-fuzz finds an interesting input, it will also add it to the corpus.

There are proposals to integrate fuzzing as a first-order function into the go programming language itself [29].

3.1 Fuzzing in the Context of Blockchains

Blockchain infrastructure, as most critical infrastructure, should employ fuzzing to find bugs and regressions. This section presents related work on testing in the context of blockchain and specifically in relation to Ethereum.

Non-differential Fuzzers Fuzzers are used to test most of the encoding/decoding and networking code in Bitcoin [17] [37]. There are also fuzzers available to test different functionalities of the go-ethereum code base that run continuously on oss-fuzz [2]. These fuzzers are (except for some that fuzz cryptographic libraries in go-ethereum) not differential and only provide coverage against crashes.

State Tests In order to allow for comprehensive tests between clients, the Ethereum Foundation created a suite of state tests that should be passed by all clients [20]. These tests are created by hand and should cover the basic functions of the EVM as well as newly found consensus bugs. The format of these tests is explained in Section 2.3.4.

Hive Hive is a set of tests for Ethereum clients. Most tests are related to syncing a blockchain from one client to another. Additionally these tests cover the networking side and the graphql interface. The tests are run continuously to ensure bug free clients and prevent regressions between releases [41].

Smart Contract Fuzzing There are over 14.5 million smart contracts currently deployed on Ethereum [48]. Contracts can be written and deployed by anyone which means they are prone to bugs. There have been many instances of money getting lost or stolen due to mistakes in smart contracts [26] [15]. In order to combat these hacks, Jiang et al. introduced ContractFuzzer, a fuzzing tool specialized on smart contracts [33]. The tool generates inputs based on the interface of a smart contract and executes the contract on a local EVM. It is very efficient in finding faults in smart contracts, but it operates on the assumption that the EVM works as described.

Differential Fuzzing for ETH2 ETH2 is an upcoming network upgrade for Ethereum. It plans to move the Ethereum network away from the energy intensive Proof-of-Work to a Proof-of-Stake. A lot of effort was put into creating differential fuzzers for these new node implementations [50]. However, the created fuzzers only fuzz on a network and decoding/encoding level as the early implementations rely on the same EVM as the current Ethereum network.

3.1.1 EVMFuzz

Fu et al. created EVMFuzz (later EVMFuzzer) which creates smart contracts based on custom mutators to find differences in EVM implementations [22] [23]. EVMFuzz has a set of 8 mutators that mutate a given smart contract in Solidity. Solidity is a high-level programming language for smart contracts. The mutated smart contract is compiled to EVM bytecode and executed on different EVM's. EVMFuzz has several shortcomings that limit the usefulness of their approach.

Generally EVM fuzzing in Solidity is not as fruitful, as the EVM operates on raw EVM-bytecode which means the mutated Solidity contracts have to be compiled to EVM-bytecode to deploy them. Therefore the Solidity compiler influences which bytecode gets generated. This is generally not desirable as it means only a subset of opcodes are executed as the Solidity compiler might not generate all opcodes (due to performance reasons).

The corpus for EVMFuzz was based on a suite of deployed smart contracts. This does provide a good starting point, however, it might also limit the fuzzer as it will try to create similar code to the one it already knows. Creating a fully random corpus or a corpus based on code coverage of an actual EVM implementation could yield better results and improve the chance to hit deep states in the EVM.

EVMFuzz employed the metrics `GasUsed` and `OpCodeOrder` to find differences in EVM executions. FuzzyVM employs the same metrics in addition to `StackItems`, `StateRoot` and `MemoryItems` in order to find more subtle differences in the execution.

Another aspect EVMFuzz overlooked are the precompiles (see Table 2.8). These programs are part of the EVM and should therefore also be subject to the fuzzers. In order to properly fuzz them, generators are needed that can create structured output that can pass a lot of initial checks.

EVMFuzz executed 253,153 test cases. The slowest goevmlab fuzzer generates and executes a similar amount of test cases in 8 hours while faster fuzzers (for example the SSTORE-fuzzer) can generate and execute them in 45 minutes. FuzzyVM executed roughly 1.8 million test cases in 7 days.

3.1.2 goevmlab

Goevmlab is a collection of tools to create state tests and find differences in the output of EVM implementations, created by Martin Holst Swende [59]. Goevmlab has also been used to create rudimentary, non-guided fuzzers for the Blake precompile, the SSTORE opcode and a fuzzer for BLS-precompiles which might be introduced in the next upgrade of Ethereum. FuzzyVM uses goevmlab as a library as it provides functions to create state tests and to read and verify the traces that the EVM implementations generate.

4 Generators

The first research question posed in Figure 1.2 asked “How can a fuzzer create meaningful programs to test Turing-complete Virtual Machines and cryptographic functions”. This chapter describes the generators used for generating valid programs to test EVM implementations. Generators take the output of the mutator (go-fuzz) and structure it into valid programs so it has a higher probability of reaching deeper states of the implementation.

It also introduces a “filler” class to easily fill objects with the output of the mutator. This class can be used by fuzzers that use a modern fuzzing framework (go-fuzz, AFL) to create meaningful tests.

4.1 Filler

FuzzyVM uses a filler class internally which is initialized with the input generated by go-fuzz. Whenever the generator needs some random data during the generation process, it can ask the filler to provide random data in a specified format. The filler internally stores the input data and a pointer on the next byte to be read. On a request, the filler returns the requested amount of bytes and increments the pointer to the next byte to be read. If the pointer reaches the end of the input data, the filler class resets the pointer to the beginning of the input data. This allows the generator to request more data than provided by go-fuzz. Figure 4.1 visualizes this process. In order to notify go-fuzz that this input was not perfect, an indicator is set when the pointer reaches the end of the input data.

The filler provides the following convenient functions to create basic types:

- `BigInt()` returns a new big integer in $[0, 2^{32})$.
- `BigInt16()` returns a new big int in $[0, 2^{16})$.

- `Bool()` returns a new boolean.
- `Byte()` returns a new byte.
- `ByteSlice(items int)` returns a byte array with `items` values.
- `ByteSlice256()` returns a byte array with 0..255 values.
- `GasInt()` returns a big integer in $[0, 2^{32})$ with probability $\frac{1}{255}$ and in $[0, 20.000.000]$ with probability $\frac{254}{255}$.
- `Read(b []byte)` reads `len(b)` bytes into `b`.
- `Uint16()` creates an unsigned integer in $[0, 2^{16})$.
- `Uint32()` creates an unsigned integer in $[0, 2^{32})$.
- `Uint64()` creates an unsigned integer in $[0, 2^{64})$.

FuzzyVM computes most internal states with input from go-fuzz in order to increase the amount of states that can be reached. Thus, the length of the bytes needed by FuzzyVM is highly dependent on the data itself. The filler, for example, determines the size of a new array by reading a byte from the input data. Therefore, the amount of data needed for filling this array directly depends on the first byte of the data.

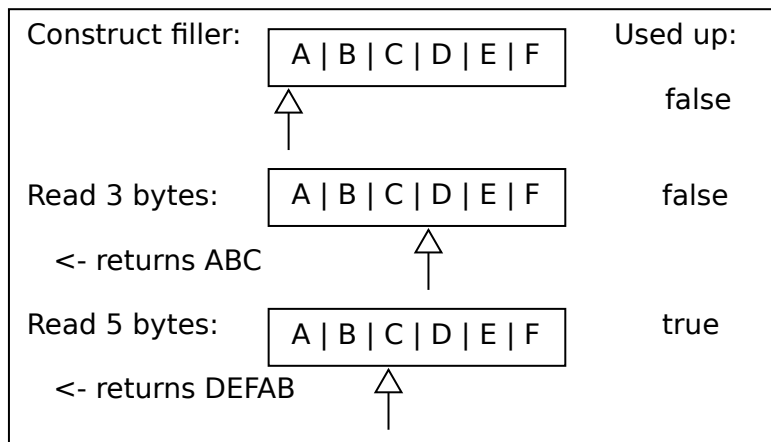


Figure 4.1: Requesting data from the filler class

4.2 Basic functions

Figure 4.2 visualizes the test generation algorithm. The first byte of the input determines how many rounds of the generators are executed. In each round, a byte is read from the filler. This byte determines which generation strategy is chosen in this round. The generators use the filler to read bytes from the input. This section discusses 13 different strategies used by FuzzyVM to create interesting tests.

```
CREATE_BYTECODE(filler)
  program = []
  rounds = filler.Byte()
  for rounds
    strategy = filler.Byte()
    switch strategy
      case 0:
        code = OpcodeGenerator(filler)
        program += code
      case 1:
        code = PushGenerator(filler)
        program += code
      ...
    end switch
  end for
  return program
```

Figure 4.2: Test generation algorithm

4.2.1 Opcodes

The simplest generator is the opcodes generator. It reads a single byte from the filler and converts it into an opcode. The opcode is added to the program.

Some opcodes require data to be stored in memory or on the stack. Executing these opcodes might not succeed if the memory or the stack are empty. Thus more complex generators are needed to cover the EVM.

The opcode generator is useful to test simple opcode operations. It, for example, found a bug in Nethermind where the `SIGNEXTEND` opcode did not pop 2 values from the stack on invalid input in a previous fuzzer [55].

BLOCKHASH The `BLOCKHASH` opcode can be used to obtain the hash of a previous block in the EVM. It allows for querying the last 256 block hashes during normal execution. During state test execution, the EVM does not have access to blocks as this would significantly increase the complexity of the state tests. Nethermind currently computes the `BLOCKHASH` opcode in tests as `keccak256(blocknumber)` while go-ethereum, Besu and OpenEthereum simply return zero. Considering all tests that execute the `BLOCKHASH` opcode fail immediately, the opcode generator was modified not to generate `BLOCKHASH` opcodes.

COPY opcodes The opcodes generator, in conjunction with other generators, found a new class of errors in Nethermind's implementation of the `CODECOPY` and `CALLDATACOPY` opcodes. The Nethermind client halted execution when the length was zero for these opcodes. Upon manual inspection, the `EXTCODECOPY` and `RETURNDATACOPY` opcodes had the same problem in Nethermind.

4.2.2 Storage generators

The EVM has access to different storage segments as introduced in Section 2.3.2. This Section discusses generators to store and load data to and from these segments.

Push The Push generator generates code that writes an array to the stack. It creates an array with 0...32 bytes and appends it to the bytecode. The Push generator calculates the length of the array in bytes as `X`. It generates a `PUSHX` opcode which is appended to the bytecode.

MemToStorage The MemToStorage generator generates a load operation which loads data from the `MEMORY` segment into the `STORAGE` segment. It generates a sequence of opcodes and data that loads data from the `MEMORY` segment using the `MLOAD` opcode. Then `SSTORE` opcodes are appended to the bytecode to store the loaded memory in the `STORAGE` segment. The generator creates three integers `memStart`, `memSize` and

`startSlot`. It generates code to load the memory in chunks of 32 bytes starting with `memStart` until `memSize` bytes have been read. The code saves the loaded memory in the `STORAGE` segment starting from the `startSlot`.

MSTORE The `MSTORE` generator generates code to write an array into the `MEMORY` segment. It generates an array of 0..256 bytes and an integer `memStart`. The resulting code stores the array in chunks of 32 bytes starting with `memStart` using the `MSTORE` opcode. All remaining bytes that are not part of a 32 byte chunk are stored using the `MSTORE8` opcode.

Sstore The `Sstore` generator generates code that stores an array in the `STORAGE` segment. An array of 0...32 bytes and an integer `startSlot` is generated using the randomness provided by `go-fuzz`. The generated code stores the array in the `STORAGE` section starting at the `startSlot` using the `SSTORE` opcode.

Return The `Return` generator generates two integers `offset` and `length`. It appends the byte code with `offset` `length` and the `RETURN` opcode. This creates a code that returns memory starting from `offset` with `length` bytes. The generated code might fail during execution if no memory is available at `offset` or if `length` is too big. The `RETURN` opcode halts the execution.

ReturnData The `ReturnData` generator generates an array of 0...32 bytes. It creates code that stores the array using the `MSTORE` opcode and returns the data using the `RETURN` opcode. This ensures that enough data is stored for the `RETURN` opcode to function properly. The `RETURN` opcode halts the execution.

4.2.3 Create and Call

A contract can call functions on other contracts. This is often used to create library smart contracts that can be used by other contracts. `FuzzyVM` contains a generator that can call other contracts. These calls can be nested up to 1024 calls deep. The generator will limit the call depth to 1025 calls in order to generate test cases that hit the limit while not spending too many resources on the generation.

Contracts can also create other contracts and call them in the same transaction. This increases the complexity of transactions significantly. FuzzyVM has two different generators to test the create and call paradigm. It will either generate a new contract or create and call a purely random contract. Including both increases the chance of the generator to create interesting test cases.

CALL opcodes Contracts can be called with three different opcodes: `CALL`, `STATICCALL` and `DELEGATECALL`. These opcodes set the `msg.sender` field to the calling contract and create a new execution environment for the callee. The `STATICCALL` opcode is used for calling contracts in a way that disallows modifications of the state during execution of the callee[64]. The `DELEGATECALL` opcode executes the called contract in the same execution environment as the caller. This means the called contract can read and change the memory and storage of the caller. The `CALL` opcode executes the called contract in its own execution environment and allows for `STORAGE` and `MEMORY` modifications in the context of the callee.

Call random address The `RandomCallAddress` generator creates code to call a random address. The code uses data stored in the `MEMORY` segment as `CALLDATA` for the call and stores the output from the call in memory. It executes the call with one of the three call opcodes. Since most addresses do not have any contract stored, it is unlikely that the generated code will ever call an address with deployed code.

Create and call random contract The `CreateAndCallRandom` generator creates a contract consisting of random bytecode. It then creates byte code that stores the contract code in the `MEMORY` segment and creates the contract with the `CREATE` or the `CREATE2` opcode. Once the contract is created, it is called using one of the three `CALL` opcodes. Once the contract execution is finished, the return value and the address of the contract is removed from the stack using the `POP` opcode. Generating random programs will most likely not result in programs that reach a big call depth as they often contain undefined opcodes that terminate execution.

Create and call generated contract The second create and call generator, `CreateAndCallGenerated`, initializes a new filler with a subset of the data from the original filler. It recursively calls another generator loop that will generate a valid program, similar to the current one being generated. Only initializing the filler with a subset of the data ensures

that the generated program is different to the current program. The generated contract is stored and called in the same way a purely random contract is created and called.

4.3 Precompiled Contracts

The EVM provides several precompiled contracts as discussed in Section 2.3.3. They mostly implement cryptographic functions which require their input to have specific structures. Thus generating valid inputs for the precompiles is complicated. Randomly generated input is unlikely to meet the specific requirements needed for the cryptographic functions.

In order to test deeper states of the precompiles, the generators need to take the required structure of the precompiles into account. Additionally a fully randomized generator, that does not take the structure into account, is used in order to hit interesting code paths in the validation code.

Precompiles can be called via the CALL, STATICCALL or DELEGATECALL opcode. STATICCALL is the intended way to call a precompile as the precompiled contracts do not modify any state and only interact with the caller over the input and output parameters. The generators will generate code that calls the precompiles with all three opcodes as the gas calculation formulas are different between them.

Random precompile The random precompile generator calls a random precompile with data from the MEMORY segment. This generator is very similar to the random call generator described in Section 4.2.3. It will not setup any memory, therefore it might fail depending on the state of the MEMORY segment before the call is executed.

4.3.1 ECRrecover

ECRrecover implements the ECDSA (Elliptic Curve Digital Signature Algorithm) signature recovery algorithm on the SECP256k1 curve [6]. The input for the precompile is defined as $h, v, r, s \in P_{256}$ [18], whereby h is the hash of the signed message (usually $\text{keccak256}(\text{msg})$). The components r and s describe together the X-coordinate on the SECP256k1 curve of the signature. Since the X-coordinate describes two points on the curve (y and $-y$), another bit is needed to differentiate between the two points. The value v is set to 27 or 28 for valid signatures to differentiate between the points. The precompile

returns $o \in P_{256}$ on a valid signature. The upper 12 bytes of o are set to zero and the lower 20 bytes are set to the lower 20 bytes of $\text{keccak}(\text{ECDSARecover}(h, v, r, s))$. This is exactly the Ethereum address of the recovered signer.

Generation strategy The ECRrecover generator generates a random key pair (sk, pk) on the SECP256k1 curve. It also generates a message $m \in P_{256}$ which is used instead of the hash h .

In order to give the mutator more control over the message, the message is not hashed before signing. Generating a message and hashing it (as usual with ECRrecover), would require the mutator to learn the structure of keccak in order to produce interesting inputs to the precompile. Consequently, the mutator would be able to predict the output of keccak which would break the security assumptions of keccak.

The ECRrecover generator now signs the message m with the generated private key sk which returns a valid signature in the $[R || S || V]$ format. The generator now splits the components to produce r , s and v and extends them to 256-bit. It generates code to store the message m as well as the components v , r , s in the MEMORY segment using the MSTORE opcode and call the precompile. Please note the changed order of the components as the precompile expects them as $(h, v, r, s) \in P_{256}$.

4.3.2 Hashes and DataCopy precompiles

The EVM contains three precompiles that consume byte arrays of variable lengths. These precompiles implement the SHA256 and RipeMD hash functions and a DataCopy function. They need no specific structure of input, thus the generation strategy for them is similar. The following proposes a generation strategy for all three precompiles as well as provides some insights into their outputs.

Generation strategy All generators for these three functions create a byte array in with a variable length. They then create code to store the in array in the MEMORY segment and call the precompile with one of the three CALL opcodes. The result of the call is not removed from the STACK after execution.

SHA256 The SHA256 precompile implements the SHA256 hashing algorithm [18]. This precompile provides the standardized SHA3-256 algorithm, not the proposed KECCAK256 which is available as an opcode. The precompile returns the hash $o \in P_{256}$ on any byte array input in .

RipeMD160 The RipeMD160 precompile implements the RipeMD (RACE Integrity Primitives Evaluation Message Digest) hashing algorithm [18]. Bitcoin uses RipeMD to calculate addresses from private keys. Through the addition of this precompile, smart contracts are able to generate and verify Bitcoin addresses which enables smart contracts to verify bitcoin transactions on the Ethereum blockchain. The precompile returns $o \in P_{256}$ on any byte array input. The upper 12 bytes of o are set to zero and the lower 20 bytes are set to the 160 bits of the hash.

DataCopy The DataCopy precompile (also called identity precompile) returns the input byte array as output [18]. It can be used to copy data from the MEMORY segment to the MEMORY segment without having to call the MSTORE opcode in a loop. This can significantly reduce the gas cost for copying data in the MEMORY segment. The output of the DataCopy precompile is as long as the provided input data.

4.3.3 Modular Exponentiation

The modular exponentiation precompile executes $b^e \bmod m$ and returns the result [18]. It takes as input six parameters $l_b, l_e, l_m \in P_{256}$ and byte arrays b, e, m . The first three parameters denote the length of the latter three parameters. This generator found a critical crasher in Besu that could be used to crash all Besu nodes. When called with l_b and l_e set to zero and l_m set to a high number, Besu would read the modulo m into a number that could overflow which crashed the client.

Generation strategy The generator creates three byte arrays $base$, exp and mod . The lengths of the arrays have to be extended to the word size of 32 bytes. The generator creates code to store the lengths and the arrays in the MEMORY segment using the MSTORE opcode before calling the precompile with one of the three call opcodes.

4.3.4 BN256 Addition

The BN256Add precompile implements addition on the alt_bn Barreto-Naehrig curve as used by Zcash [18]. It takes as input $a_x, a_y, b_x, b_y \in P_{256}$ which describe two points on the curve a and b . The precompile returns $c_x, c_y \in P_{256}$ which describe the point $c = a + b$ on the curve.

Generation strategy The generator creates two large numbers k and k' in order to create two valid points on the alt_bn curve. It then computes $a := g^k$ and $b := g^{k'}$ with g being the generator of the group¹. Out of the generator property of g , it follows that g^x is a valid element of the group for every $x \in \mathbb{N}$. Additionally, it follows that all valid points of the group can be generated by computing g^x . The points a and b are marshaled into their canonical form (32 bytes each for the x and y coordinate). The generator then generates code that stores the marshaled points a and b in the MEMORY segment using the MSTORE opcode and calls the precompile.

4.3.5 BN256 Scalar Multiplication

The BN256ScalarMul precompile implements a scalar multiplication on the alt_bn Barreto-Naehrig curve as used by Zcash [18]. It takes as input $a_x, a_y, s \in P_{256}$ which describe a point on the curve a and a scalar s . If a is a valid point on the curve, the precompile returns $c_x, c_y \in P_{256}$ which describe the point $c = a * s$ on the curve.

Generation strategy The generator creates two large numbers k and s . It computes $a := g^k$ with g being the generator of the alt_bn curve. The point a is marshaled into canonical form (32 bytes each for the x and y coordinate). Then the scalar s is extended to the EVM word size of 32 bytes. The generator now generates code that stores the marshaled point and the extended scalar in the MEMORY segment using MSTORE opcodes and calls the precompile.

¹The g^x operation here is defined as $g * g * \dots * g \bmod p$ with p the prime order of the group

4.3.6 BN256 Pairing

The BN256Pairing precompile implements a pairing check on the alt_bn Barreto-Naehrig curve as used by Zcash[18]. It takes as input a list of n tuples $(a_x, a_y, b_{xx}, b_{xy}, b_{yx}, b_{yy})$ with $a_x, a_y, b_{xx}, b_{xy}, b_{yx}, b_{yy} \in P_{256}$. The elements of a tuple define points a on the G_1 curve and b on the G_2 curve. The pairing precompile calculates an Optimal Ate [44] pairing on the n pairs of points. The precompile outputs a bit 0/1 (extended to 32 bytes word size) if the pairing is valid.

Generation strategy In order to test the BN256 pairing precompile, the generator will create a list of n random tuples (a, b) . It will then compute

$$e(a_1 * g_1, b_1 * g_2) * e(a_2 * g_1, b_2 * g_2) * \dots * e(a_n * g_1, b_n * g_2) == e(g_1, g_2)^s$$

with $s = \sum_{i=1}^n (a_i * b_i)$, g_1 and g_2 being the generators of the G_1 and G_2 curve respectively, which forms a valid pairing.

The generator inserts, with probability 1/2, random tuples $(a, 0)$ or $(0, b)$ into the left hand side of the equation. These tuples represent points at infinity that shall be ignored by the precompile when computing the pairing.

The tuples are marshaled into canonical form (32 bytes each for the two coordinates of the G_1 points and for the four coordinates of the G_2 points, resulting in 196 bytes per tuple). The generator generates code to store the tuples in the MEMORY segment and to call the BN256Pairing precompile.

4.3.7 Blake2f

The Blake2f precompile implements the Blake2 “F” compression algorithm [28][18]. This compression algorithm is used in the BLAKE2b cryptographic hashing algorithm. Multiple different algorithms in the BLAKE2 family use the “F” compression algorithm. Including the compression algorithm instead of one of the BLAKE2 variants (like BLAKE2b) has the advantage that other algorithms can use it as well. The precompile takes as input a tuple $(rounds, h_0 \dots h_7, m_0 \dots m_{15}, t_0, t_1, f)$ with $rounds \in P_{32}$, $h_x, m_x, t_0, t_1 \in P_{64}$ and $f \in P_8$. The element $rounds$ denotes the number of rounds that the “F” function shall be executed. The element $h_0 \dots h_7$ is a state vector denoting a previous state of the “F” function. The element $m_0 \dots m_{15}$ is the message to be compressed. Elements t_0 and t_1 are offset counters

and f is a flag which indicates the final block. It outputs $h'_0 \dots h'_7$ a new state vector which is either the final state or the input for the next round of the “F” function.

Generation strategy The Blake2f generator generates a tuple $(rounds, h_0 \dots h_7, m_0 \dots m_{15}, t_0, t_1, f)$ as described previously. The generator sets f to a random byte with probability $1/2$, to 0 with probability $1/4$ and to 1 with probability $1/4$. The precompile will only accept inputs with f set to 0 or 1.

The elements of the tuple are marshalled into a byte array with 213 elements. The generator creates code to store the marshalled elements in the MEMORY segment using MSTORE opcodes and to call the Blake2f precompile.

4.4 Jumps

The EVM allows for jumps using the JUMP and JUMPI opcodes. There are several peculiarities concerning the use of JUMP opcodes in the EVM. This section will discuss some of the peculiarities of the EVM and strategies how to create valid code to test the jump instructions.

JUMPDEST The EVM contrary to most other virtual machines does not allow to jump into arbitrary destinations. All jump destinations have to be set up with the JUMPDEST opcode. This is a safety measure intended to prevent obscure side effects during execution. Generators therefore need to set up jump destinations in the code to which they can later jump to maximize code coverage.

Infinite loops Generators for arbitrary programs have to be careful to not generate infinite loops [25]. Ethereum's inherent mechanism of each executed opcode costing gas prevents programs from executing indefinitely. Thus infinite loops are impossible in Smart Contracts. However, very small loops can cause problems with the amount of output created. One issue during testing was that OpenEthereum did not provide a feature to disable printing the storage. FuzzyVM created tests which called SSTORE in a loop with a big array consisting of ones. This test created so much output that the OpenEthereum binary could not finish executing the test in more than one hour. The issue was filed as a result of this thesis and later fixed [51]. Creating tests where an opcode is executed in a

tight loop, while being beneficial for finding problems like this, could severely impact the performance of the fuzzer.

4.4.1 Generation strategies

Several different strategies were tested in order to create interesting tests with jump instructions, without creating too many loops and especially without creating too many small loops.

Setting up jump destinations Jump destinations can be set up by adding a JUMPDEST opcode to the program. The EVM executes a jump instruction only if the destination contains the JUMPDEST opcode. If a JUMPDEST is executed during normal control flow it is executed as a no-op that costs 1 gas. Most clients will run a jump destination analysis before executing a transaction to store the valid jump destinations in memory. The JumpDest generator adds a JUMPDEST operation to the bytecode and stores its location, so it can be used in other generators as a destination.

Disallow backward jumps This strategy disallows backward jumps, meaning jumps where the jump destination is lower than the current program counter. This strategy prevents all loops as forward jumps can never create a loop. Therefore, it is the optimal strategy in preventing infinite loops. A drawback of this strategy is that it limits the amount of testing that can be done as no loops are possible anymore. FuzzyVM does not use this strategy.

Allow long backward jumps This strategy improves the previous strategy of disallowing all backward jumps by allowing jumps that are “long”. Different metrics can be utilized to measure “long” in this context. The most obvious metrics are the amount of gas used by one iteration and the number of executed opcodes.

Due to the way the bytecode is generated, the generator does not know how much gas is spent during execution. This could be solved by executing the bytecode and measuring the gas of each opcode. This approach would increase the execution time of the generation phase significantly while providing little benefit.

The second metric - number of executed opcodes - can be used by FuzzyVM as it knows both the program counter of the jump destination as well as the jump instruction. Thus the

number of executed opcodes in one iteration can be easily calculated. FuzzyVM prevents the generator from creating loops that execute less than 10 opcodes.

Forward jumps Forward jumps are more interesting than backward jumps as they do not execute code that has already been executed. Since the EVM cannot jump arbitrarily though, the program needs to know the jump destination when executing the jump instruction. Also it means that potentially interesting instructions might be skipped.

Since the EVM needs to know the jump destinations beforehand, FuzzyVM creates a placeholder (8 times the byte 0xFF) instead of the jump destination next to the JUMP or JUMPI opcode. The program counter of the jump instruction is stored. FuzzyVM also stores the program counters of all generated JUMPDEST opcodes during generation. After the program is successfully generated, all placeholders are replaced with valid jump destinations. If no valid jump destinations are found, a random value is inserted instead.

Jump in data segments Since a random jump destination is inserted, if no other jump destinations are available, the program might jump into a data segment. The byte 0x5B is the byte value for the JUMPDEST opcode. If the destination in the data segment by chance contains a 0x5B byte, the code in the data segment might get executed. This should not be disallowed as it can lead to code paths that otherwise would not be executed.

Conditional jumps The JUMPI instruction is a conditional instruction meaning the jump is only executed if a certain condition is met. It expects two elements on the stack, the condition and the jump destination. If the condition is non-zero the JUMPI instruction will jump to the destination. The generator for conditional jumps will create code that sets the condition to zero with probability $1/2$ and to a value in $[0, 2^{32})$ with probability $1/2$ before executing the JUMPI instruction.

5 Corpus

The corpus is a set of test inputs maintained by the mutator that produce interesting EVM programs. These inputs can be used by the mutator to generate other interesting inputs. This chapter aims to answer the third research question posed in Figure 1.2: “How much benefit can an existing corpus bring and how can the quality of an initial corpus be improved?”.

If go-fuzz receives no corpus, it will try to generate its own corpus. This corpus however is very unlikely to generate sufficiently large test inputs. Section 5.1 explores how a good initial corpus impacts the code coverage a fuzzer can achieve.

Creating corpus elements that are too large or too short reduces the efficiency of the fuzzer or even decreases the possibility of certain programs being created. Section 5.2 discusses problems related to the size of the corpus elements. Creating good corpus elements by hand is complicated as it would require the user to reverse engineer the generation algorithm. Section 5.3 proposes an algorithm that produces valid corpus elements that are neither too short nor too long for the generators. Finally Section 5.4 discusses the statistical variance in length of generated elements and their impact on the performance of a fuzzer.

5.1 Corpus size

This section discusses the effect of a good existing corpus on the success of a fuzzing framework. Theoretically a fuzzer without existing corpus is able to create the same tests as one with existing corpus. However providing initial corpus increases the probability of creating interesting tests. Figure 5.1 shows the impact of existing corpus on the code coverage achieved by the fuzzer. FuzzyVM was executed in the following three different configurations.

- In the first configuration the fuzzer is executed without existing corpus
- In the second configuration the fuzzer is executed with 100 existing corpus elements generated by the algorithm discussed in Section 5.3
- For the third configuration 100 corpus elements were created and a run that produced 1 million tests was executed. Then the fuzzer was executed with the resulting 294 corpus elements to generate 100.000 tests.

All configuration were run to generate 100.000 tests.

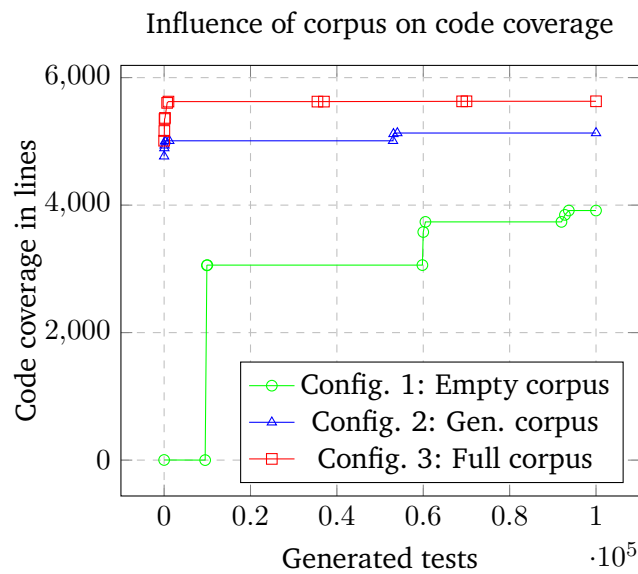


Figure 5.1: Influence of corpus on code coverage

Figure 5.1 shows that fuzzers with a generated corpus cover more code than without any generated corpus. Thus generating some corpus increases the efficiency of a fuzzer greatly. Configuration 3, which used a corpus refined over one million tests provided the highest code coverage and smallest improvements.

Comparing configuration 2 and 3 also shows the impact of generating one million tests has on code coverage. Configuration 2 provided a code coverage of 5009 lines after the initial setup phase while configuration 3 covered 5624 lines. This difference is significant as improvements in code coverage are usually between 3-20 lines.

Figure 5.1 also shows that the fuzzer often runs into local minima. If it escapes a minima, the fuzzer usually finds multiple interesting inputs before ending up in another local minima.

It has to be noted that all configurations will produce the same total coverage given a long enough time frame. However, executing 100.000 tests takes roughly 10 hours on a decent sized machine.

5.2 Corpus elements

It is important to generate sufficiently large test inputs. The length of input needed is highly dependent on the input itself as described in Section 4.1. If the data is too short, the same data is used in different parts of the generator. This means that a change in the data will change the behaviour in different parts of the program. Hence, the mutator can not mutate the inputs properly. Figure 5.2 visualizes what can happen if the input is too short.

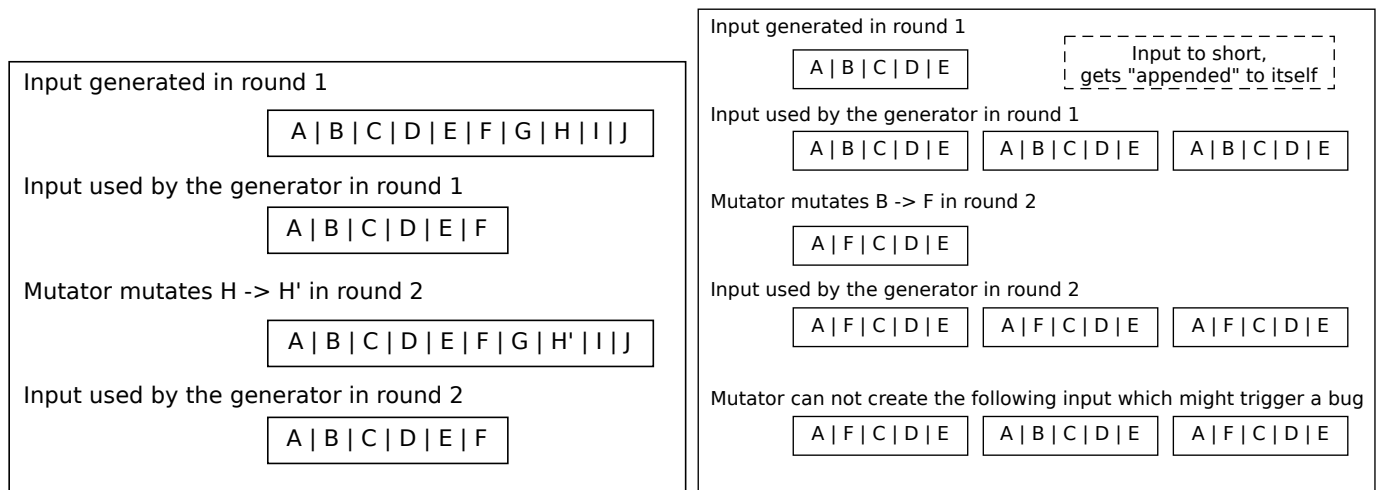


Figure 5.2: Inputs that are too long or too short confuse the mutator

On the other hand, creating inputs that are too large will reduce efficiency. The created contracts will be the same if the mutator only switched bits in the data that is not read during test generation. Figure 5.2 visualizes what can happen if the input is too long.

Therefore, it makes more sense to create inputs that are slightly too long as they will only mean a loss of efficiency while inputs that are too short might not trigger bugs.

5.3 Corpus generation

Generating valid test inputs is important for the efficiency of corpus-based fuzzers. FuzzyVM provides a command to generate valid test inputs for the corpus. These test inputs can then be added to the corpus. Figure 5.3 describes the algorithm used to generate valid inputs.

```
GENERATE_CORPUS_ELEMENT()
  Sample a random byte array r <- rnd().
  Set array b to r[0..len(b)/2].
  Set right to len(r)
  Set left to 0
  Repeat:
    Execute the Fuzz() function on b
    If the input was sufficiently large:
      Set right to len(b)
      Set mid to left + (right-left)/2
      Set b to r[0..mid]
    Else:
      Set left to len(b)
      Set mid to left + (right-left)/2
      Set b to r[0..mid]
  While right-left > 1
  return r[0..right]
```

Figure 5.3: Corpus generation algorithm

The algorithm uses a binary search over the length of an array. The algorithm samples a long byte array at random. Now the generation algorithm `Fuzz()` is executed on half of the byte array. If the input was sufficiently large, the input size gets halved and the generation algorithm is executed again. Otherwise, the input gets extended from the original array until the input is sufficiently large. This algorithm generates a valid input in $O(\log(n))$ steps.

5.4 Statistical variation of input length

During the implementation of the binary search algorithm, it became apparent that the length of the required input data varies greatly. The smallest generated input was 53 bytes while the biggest input was roughly 13.000 bytes. This discrepancy warrants a critical analysis.

Figure 5.4 shows the statistical variation of input lengths when randomly sampling 10.000 tests. In order to reduce noise, the inputs were grouped in groups of 500 bytes. Thus a test that needs 40 bytes and one needing 350 bytes would be grouped into the $[0, 499]$ group. It shows that most tests need less than 6000 bytes of input. The median tests needs 3384 bytes of input in order to not reuse or waste inputs. Computing the standard deviation per tests results in 806 bytes.

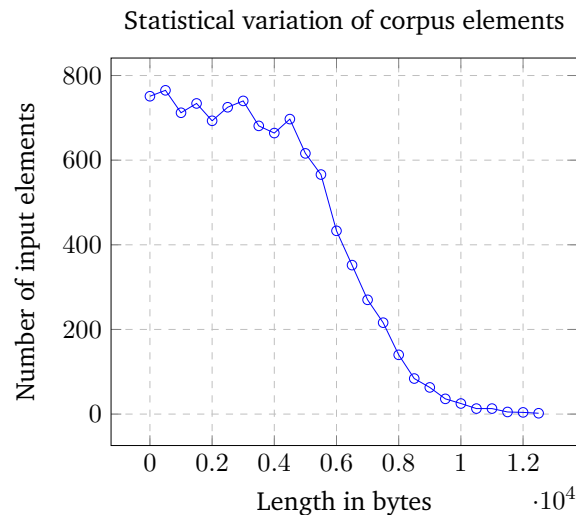


Figure 5.4: Statistical variation of corpus elements

As previously discussed, inputs that are either too short or too long impact the performance of the fuzzer. Thus, a high standard deviation of the input length points to a suboptimal scheme. Unfortunately, it is unavoidable to have this variance since limiting the input length also decreases potential code coverage in the EVM's. Go-ethereum's standalone fuzzing of their modExp precompile for example found a crasher in the golang math library that only occurred when a large modulo (larger than 100 bytes) was executed [62].

6 Architecture

The tasks performed by a differential fuzzing frameworks can be categorized in three phases: generation, execution and verification. This chapter discusses different architectures to combine these phases into processes in order to answer the second research question posed in Figure 1.2 which asked “How can the quality of a differential fuzzing framework be evaluated, and how do architectural choices impact it?”.

These phases require different amount of work. In FuzzyVM the verification phase is the fastest, taking around 20 microsecond per test. The generation phase takes around 2 milliseconds to complete while the execution phase for a single test takes around 2 seconds to execute as shown in Table 6.1.

This chapter discusses four different approaches to structure these phases. Section 6.1 discusses the naive structure where generation, execution and verification are run in the same process. Section 6.2 discusses a structure where generation, execution and verification are executed in three different processes and intermediate results have to be stored on disk. Section 6.3.1 and Section 6.3.2 discuss structures where execution and verification are executed in the same process.

Phases In order to decide on a fitting architecture it is important to benchmark the three different phases separately. FuzzyVM provides a `-bench N` flag which benchmarks the different phases with N tests each. Figure 6.1 provides an overview of the results. All phases were constricted to a single thread with no advanced techniques like batching or docker enabled. It shows that the execution phase is the most computational intensive. The execution phase takes over 54 minutes on 1000 tests while the generation phase only takes 2 seconds to generate the same number of tests. Thus optimizing the execution phase is more important than optimizing generation and verification. It also shows that splitting execution and verification into different processes and requiring communication between them would result in an unnecessary overhead.

| Phase | 1 | 10 | 100 | 1000 |
|---------------------|-------|--------|--------|-------|
| <i>generation</i> | 80ms | 232ms | 437ms | 2.52s |
| <i>execution</i> | 4.62s | 32.54s | 5m25s | 54m6s |
| <i>verification</i> | 15µs | 101µs | 1.85ms | 20ms |

Table 6.1: Timing of different phases with 1, 10, 100 and 1000 tests

Generation The *generation* phase takes the input of go-fuzz and creates a new state test using the generators discussed in Chapter 4. The generated tests have to be executed once on go-ethereum to calculate the correct state root, otherwise the clients will declare every tests as failed. This execution allows go-fuzz to measure the code coverage that the state test achieved inside go-ethereum’s EVM implementation. Thus FuzzyVM aims to maximize the code coverage of the *generation* phase. The state tests are then written to disk so they can be read in the *execution* phase by the EVMs.

Execution In the *execution* phase the EVMs are started. They read the generated state tests from disk, execute them and produce the traces as shown in Figure 9.8 in the Appendix. They might also produce errors which have to be caught by FuzzyVM. Since the EVMs are executed in different processes it is impossible for FuzzyVM to measure the code coverage that a test achieved within an EVM in the *execution* phase.

The EVMs can either be executed sequentially or in parallel. Section 6.3.1 discusses executing the EVMs in the *execution* phase sequentially and Section 6.3.2 discusses executing them in parallel within the overall architecture.

Verification The traces produced by the different EVM implementations are compared in the *verification* phase. If a difference is found all traces and the state test that produced the traces are saved for manual review. If no difference is found in the *verification* phase, the state test is deleted.

Processes and Threads This section distinguishes between processes and threads. Processes are standalone program executions with their own resources while threads share some resources with other threads. A process can start multiple threads. These threads can efficiently share data via shared memory. Processes can interact with each other using inter-process communication or by reading and writing to disk. When using IPC the sender and receiver process have to be synchronized.

EVMs have to be executed in their own process as they are provided as standalone applications. An exception to this is the go-ethereum EVM implementation which can be called by go-fuzz in the same thread as it is written in golang. Go-ethereum's EVM functionality is also used to measure the code coverage of the fuzzer.

FuzzyVM uses threads to implement parallel executions as much as possible as they are more lightweight than processes.

6.1 Strategy 1: Gen+Ex+Vrfy

Figure 6.1 visualizes a basic structure where the generation, execution and verification phases are run in the same process. This is the structure the basic fuzzers provided by goevmlab [59] use. This naive structure is the easiest to implement and good for normal fuzzers however it has some disadvantages for differential fuzzers.

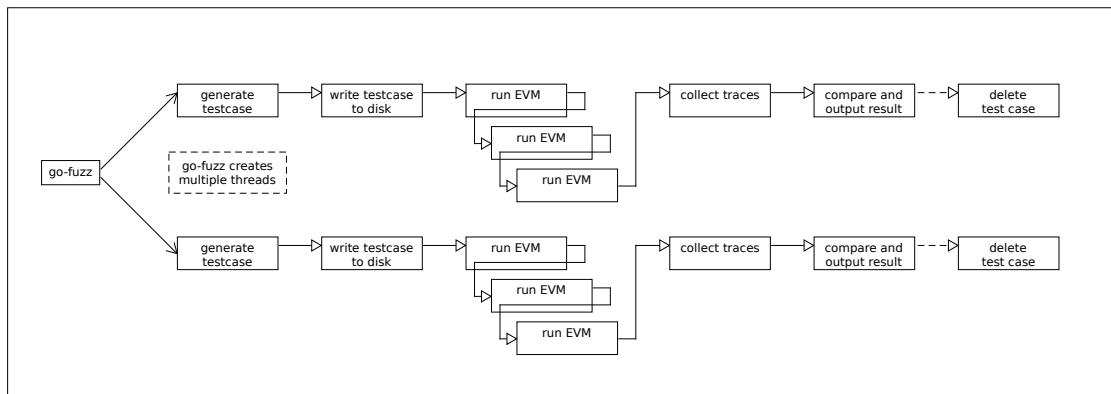


Figure 6.1: Basic structure of the goevmlab fuzzers

Combining execution and generation means that go-fuzz has to wait for all EVMs to finish until it can mutate the input. Since go-fuzz creates different threads for each execution it is not possible to batch tests together with this structure. Additionally since go-fuzz measures the code coverage of all code that is executed, the code for execution and verification is also taken into consideration when mutating. This can negatively influence the mutator as it might mutate the input to increase coverage of execution and verification instead of generation.

6.2 Strategy 2: Gen/Ex/Vrfy

Another strategy is splitting generation, execution and verification in three different processes. Hereby the phases are executed in parallel and will wait in a loop for the intermediate results of the previous state. After the `generation` phase, the state tests are stored to disk.

The `execution` process then executes the test cases and saves the results back to disk. This results in 1 write and n reads of the test (one for each EVM) and n writes of the traces. Afterwards the execution results are read by the `verification` process and verified against each other, resulting in another n reads of the traces.

The biggest disadvantage of this structure is the increased disk requirement. Every test result of every EVM implementation has to be written to disk and read afterwards resulting in an additional n writes and n compared to the naive approach. In conclusion this strategy has the highest disk requirement but lowest memory requirements since the results can be written to the file as soon as they are available. Figure 6.2 shows the test generation for this strategy, Figure 6.3 shows the test execution in this strategy and Figure 6.3 shows how test verification works in this strategy.

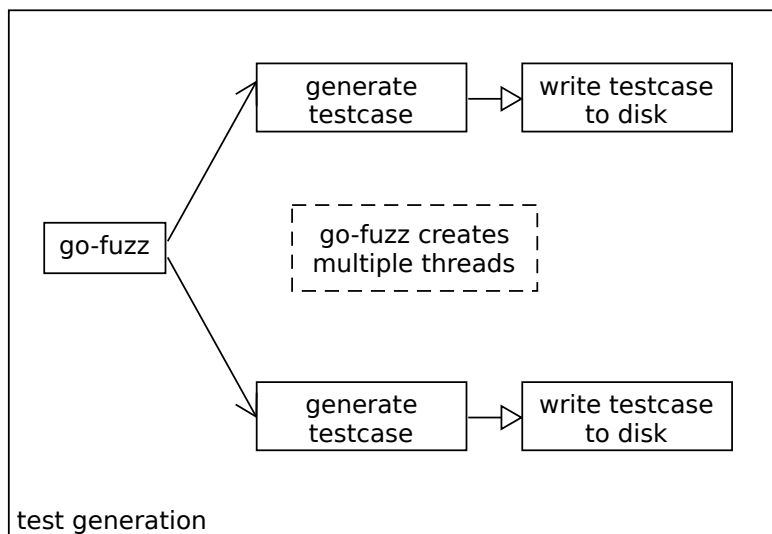


Figure 6.2: Test generation using `go-fuzz` for strategies 2, 3.1 and 3.2

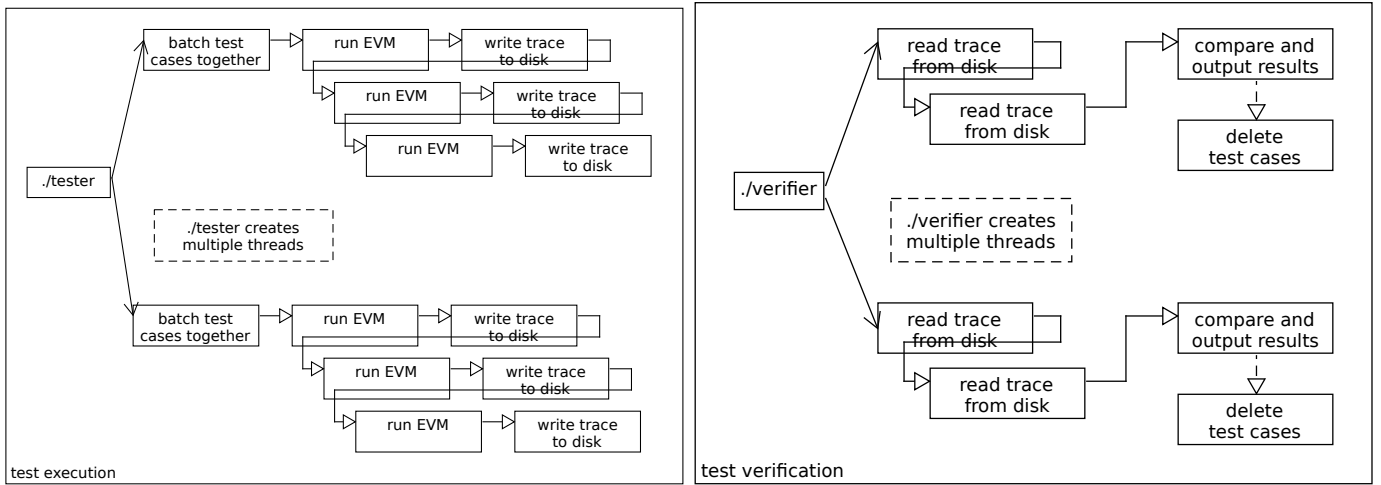


Figure 6.3: Test execution and verification in strategy 2

6.3 Splitting generation and execution

The strategy used by FuzzyVM combines execution and verification while executing the generation phase in separate processes. Hereby go-fuzz creates test cases and writes them to disk similar to Strategy 2. Another process `tester` reads the tests, batches them and executes them on the EVMs. This provides several benefits over the basic structure.

Splitting generation and execution allows FuzzyVM to execute several tests in a batch which greatly improves throughput. Batching techniques are discussed in Section 7. Additionally it permits for finer granularity between the amount of threads generating and executing the state test. Test generation is significantly faster than execution, therefore a single generation thread can generate enough tests for multiple execution threads. The test generation for Strategy 3.1 and Strategy 3.2 is the same as for strategy 2 as shown in Figure 6.2.

During the execution phase the tests on the EVMs can be executed one after another or in parallel. Section 6.3.1 describes the linear approach while Section 6.3.2 describes the parallel approach.

6.3.1 Strategy 3.1: Gen/Ex+Vrfy linear EVMs

In this structure the EVMs are executed on the same tests one after another. The tests results are stored in memory and compared to each other after all tests finished. Figure 6.4 shows the linear execution structure.

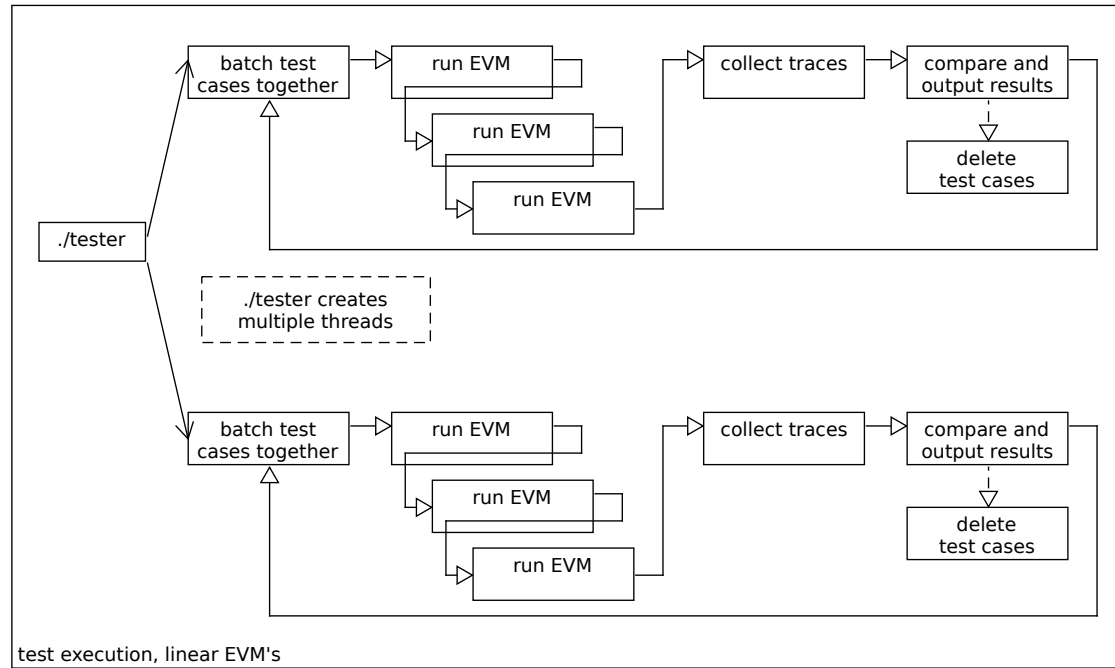


Figure 6.4: Linear test execution

Since the EVMs are executed in sequence its easier to estimate how many threads are needed to fully utilize a processor.

A disadvantage of this approach is that the execution results have to be stored for the whole time of the execution resulting in memory required during execution of $M_E + n * M_R$ with n being the number of EVMs, M_E the memory required for the execution of a single EVM and M_R the memory required for loading a test result. This means the memory needed for the execution results ($n * M_R$) has to be held longer. A strategy to mitigate this is to run two tests, compare the results and drop one of the results if equal before running the next EVM. This would result in a maximum memory requirement of $M_E + 2 * M_R$

per state test. Since a total of p tests can be scheduled concurrently on a machine with p cores, the overall memory requirement is bounded by $O(p * (M_E + 2M_R))$.

6.3.2 Strategy 3.2: Gen/Ex+Vrfy parallel EVMs

In this strategy all EVMs are started concurrently in the generation phase as shown in Figure 6.5. All execution results and errors are collected. After the execution finished on all EVMs the test results are compared to each other. If an error occurred during the execution or the test results are different all traces of the EVMs are written to files.

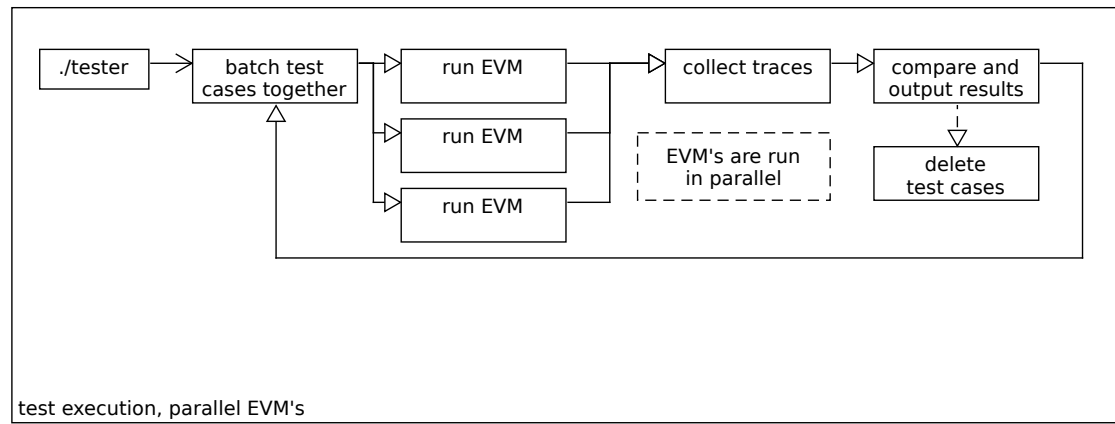


Figure 6.5: Parallel execution

This strategy is more difficult to implement compared to the previous strategies as the results and potential errors of the execution have to be collected concurrently. This approach makes it harder to compute the parallel threads needed to fully saturate a machine. Instead of scheduling p threads executing each all EVMs sequentially, we need to schedule $\frac{p}{n}$ threads, with p being the number of processor cores available and n the number of EVMs. This might result in a loss of efficiency if one EVM takes significantly longer than others, resulting in a core being idle until all results are collected.

Running the EVMs concurrently increases the memory consumption to $n * M_E + n * M_R$ per test execution compared to Strategy 3.1. However since less tests can be executed concurrently this does not result in increased memory consumption, even though the memory saving technique of dropping results right after being compared does not work in

this architecture. The overall memory requirement of this technique is $O(\frac{p}{n} * n * (M_E + M_R))$ which can be simplified to $O(p * (M_E + M_R))$.

6.4 Comparison

This chapter introduced four different strategies of structuring the three phases `generation`, `execution` and `verification`. Table 6.2 provides an overview of the different strategies and their advantages and disadvantages.

| Strategy | complexity | disk access | | batching | Efficiency |
|---------------------------|------------|-------------|------|----------|------------|
| | | write | read | | |
| 1: Gen+Ex+Vrfy | simple | 1 | n | no | medium |
| 2: Gen/Ex/Vrfy | medium | $1 + n$ | $2n$ | yes | high |
| 3.1: Gen/Ex+Vrfy linear | medium | 1 | n | yes | high |
| 3.2: Gen/Ex+Vrfy parallel | hard | 1 | n | yes | low |

Table 6.2: Overview of different strategies

Implementation complexity The four structures differ in implementation complexity. Strategy 1 is the easiest to implement since everything is structured around `go-fuzz`. The other strategies require a split between the binaries and therefore logic to write the generated tests to a file. Strategy 2 is slightly harder to implement than strategy 3.1 as it needs additional logic to read and write the test results from files. Strategy 3.2 is the most complex as running the EVMs in parallel introduces locking as well as a consumer/producer mechanism.

Mutator compatibility Since `go-fuzz` only supports total code coverage, limiting the amount of code which is fuzzed provides better results. All strategies that separate the generation phase from the other phases are optimal in this respect, since as little code as possible is considered by the mutator. Consequently strategy 1 is sub-optimal regarding this dimension as the mutator will take code from the `execution` and `verification` phase into account when computing the code coverage of the test target.

Batching Executing test cases in batches can significantly increase the throughput of FuzzyVM. However, batching in strategy 1 is infeasible as the multiple test would have to be generated on the same input. Hence the mutator would run only $\frac{t}{k}$ times compared to t times in the other cases with t being the number of parallel executions, k the number of tests per batch. Additionally the input needs to be $k * b_t$ instead of b_t bytes long for the mutator to function perfectly with b_t being the bytes needed per test. Consequently strategy 1 is sub-optimal regarding this dimension as well.

Strategy 2 would allow for batching of the verification phase. However since verification is a strictly linear process that is very fast in practice, batching in the verification phase does not provide benefits. A more detailed look on batching strategies can be found in Chapter 7.1.2.

Disk usage Another important aspect is disk usage. Google’s guide for fuzzing suggests to restrict I/O usage as it significantly impacts the performance of a fuzzer [19]. Since the EVM functionality is provided as standalone applications that read the test cases from disk, the test cases need to be written to disk in the generation phase. Hence 1 write per test occurs in the generation phase regardless of structure.

In the execution phase, each EVM reads the test separately, therefore n reads occur regardless of structure. Since reading files from disk typically only needs a read lock [21], the tests can be executed in parallel. On operating systems that do not support shared read locks, strategy 3.2 would not be feasible as the EVM implementations typically close the file after finishing the test not before. Thus parallel access would be serialized.

In strategy 2 the test results have to be either written to a file in the execution phase or passed to the verification process over a pipe. If they are written to a file, they have to be read again by the verification process. Thus another n files are written and n files are read. This creates unnecessary I/O which slows down FuzzyVM. Therefore splitting the execution and verification phases into different processes reduces the efficiency of FuzzyVM.

Strategy 2 could be useful in cases where extremely large traces are generated that would not fully fit into memory. Splitting the verification phase would allow to read traces line by line from files which needs even less memory than Strategy 3.1 with the memory decreasing optimization.

Efficiency It is crucial to calculate the amount of threads needed by the application in order to match it to the number of processors in the system. If too many threads are started at the same time, cache invalidation and branch prediction can severely impact performance. If too little threads are started, the processor runs idle. All structures that do not start multiple threads in parallel within the *execute* and *verification* phases make predicting the required number of threads straightforward. For these structures the needed parallelization can be easily calculated as $t = p$ with p being the number of processors available in the system.

Only Strategy 3.2 starts multiple threads in parallel during the *execution* phase. Theoretically the threads needed for strategy 3.2 can be calculated as follows: $t = \left\lceil \frac{n}{p} \right\rceil$ with n being the number of EVMs. This has two different drawbacks, since EVMs have vastly different runtimes (as shown in Table 1.3) all EVMs need to wait for all others to finish until starting the verification. This reduces the efficiency since cores stay idle, waiting for the execution on one to finish. The second drawback is that if $\frac{n}{p} \notin \mathbb{N}$ either some cores will always idle or more work is scheduled resulting in cache pollution.

Since the *generation* phase is faster than *executing*, a single *generation* thread can supply multiple *execution* threads. Strategy 1 needs to write the newly generated test to disk before *executing* the EVMs. Strategy 2 and 3.1 can *execute* tests slightly faster, since the tests are already generated and written to disk by another process. Additionally the instrumentation of go-fuzz decreases the efficiency of Strategy 1 slightly, as more code with the instrumentation is *executed*.

Conclusion Figure 1.2 posed the question “How can the quality of a differential fuzzing framework be evaluated, and how do architectural choices impact it?”. Section 6.4 compared the four different structures on different dimensions. Strategy 1 describes a structure that, while very easy to implement, is bad for batching tests and slightly worse than others in workload efficiency and confuses the mutators by having more code under fuzzing than other approaches. Strategy 2 disqualifies through the pricey overhead of writing the test results to a file. This strategy would be suitable for large test cases with heavy computation and little output. It can also enable execution if the test results are large or many EVMs are under test, since the test results can be streamed to the files which requires less memory than other strategies.

Strategies 3.1 and 3.2 are virtually equal, but 3.1 has the advantage that threads do not have to wait for the longest running EVM to finish. Additionally it is easier to calculate the number of processes needed to fully utilize a CPU. The performance difference between



| Strategy | plain | batch | plain docker | docker batch |
|---------------------------|-------|-------|--------------|--------------|
| 3.1: Gen/Ex+Vrfy linear | 7m17 | 4m14 | 19m46 | 13m55 |
| 3.2: Gen/Ex+Vrfy parallel | 19m25 | 18m51 | 24m7 | 25m18 |

Table 6.3: Execution of 1.000 tests with different structures

these strategies can be seen in Figure 6.3. It shows that Structure 3.1. is significantly faster than 3.2 in every configuration. The plain execution just executes one test after the other while the batch execution executes multiple tests together on the same instance as discussed in Section 7.1.2.

This chapter answered our research question by showing that the architectural choices when designing a differential fuzzing framework impact different aspects of the execution. Strategy 3.1, which splits the generation phase into a different process from the other two phases provides the optimal architecture regarding compatibility with the mutator and the amount of necessary disk accesses. It allows for batching test cases together which increases efficiency and it allows for fully utilizing the given resources. This architecture can be used by other differential fuzzing frameworks to increase their throughput when testing standalone programs.

7 Evaluation

Evaluating the performance of a fuzzing framework is complicated and has been topic of discussion in literature [36]. This chapter aims to explore the second research question posed in Figure 1.2 “How can the quality of a differential fuzzing framework be evaluated, and how do architectural choices impact it?”. Fuzzers can be evaluated according to different metrics. The most commonly used metrics are throughput, code coverage and ground truth [36].

The easiest metric to test is throughput, often measured in executions per second. Section 7.1 provides some insight and potential improvements regarding the throughput of FuzzyVM.

FuzzyVM uses a feedback loop mechanism between go-fuzz and the generator as discussed in Section 2.4.6. Hereby go-fuzz generates inputs and measures the code coverage they achieve. Thus measuring code coverage is a good metric when evaluating guided fuzzers. Section 7.2 evaluates FuzzyVM regarding the code coverage metric.

The third metric discussed is ground truth which describes the time needed to reproduce known bugs. We distinguish between intentionally placing a bug and trying to reproduce an existing bug in an old version. When placing a bug an intentional defect is introduced into the code and the time is measured until the fuzzer hits the bug. When trying to reproduce an existing bug, an older version of the test target is fuzzed to measure how long it would have taken the fuzzer to find it [3]. Section 7.3 evaluates FuzzyVM regarding this metric.

Phases Chapter 6 introduced three different phases which a differential fuzzing framework implements; the generation, execution and verification phase. There is a big difference in execution time between these phases. The execution phase takes over 54 minutes to execute 1000 tests while the generation phase only takes 2 seconds to generate the same number of tests. The verification phase is the fastest phase with

roughly 20ms to verify output from 1000 tests. Thus optimizing the execution phase is more important than optimizing generation and verification. It also shows that splitting execution and verification into different processes and requiring communication between them would result in an unnecessary overhead as previously shown.

7.1 Throughput

This section aims to provide more insights into the performance of FuzzyVM and potential bottlenecks. Section 7.1.1 discusses the methodology and environment used to test the runtime and throughput of FuzzyVM. Section 7.1.2 discusses improvements that can be used to increase the throughput of a fuzzing framework.

7.1.1 Methodology

FuzzyVM provides the `-bench N` flag which executes N tests with the different strategies and techniques in order to produce consistent and reproducible results. The benchmarks always use the same randomness which ensures that the generated tests are always the same between runs.

FuzzyVM version The performance of the tests is highly dependent on which tests are executed since they represent different programs that are executed on the EVM. Tests that fail after a few instructions are less resource intensive than tests that execute a loop with a lot of memory accesses. Since the generated tests are highly dependent on the algorithms used in the generator, it is crucial to test a specific version to ensure comparability of the results. FuzzyVM version 0.1 with git commit hash `c1ad276fa37a39d519bb2b4ee74987092f52c79a` was used to generate all performance numbers employed in this thesis.

Machine All tests were executed on a AMD EPYC 7702P 64-Core processor with 64 cores and Hyperthreading enabled. The machine has 128GB RAM and a NVME SSD with up to 355k IOPS. The computer ran linux 4.19.0-13-amd64.

EVMs The following versions of the EVMs have been used for all tests:

- go-ethereum: 61469cfeaf2a6d0b1598afbaf35dd2d1872604ce
- nethermind: 58751d3a487735cff6d759073ba7ce1c58b55332
- openethereum: 3f8e0cfec4e0691f62a4bd2baf234ddd9d8da016
- besu: dcc45aa407ef38e5a38ff2d5429613bdfe56e412
- turbogeth: c6c82b75698fb29e4c6af2f483e065e84f806356

All EVMs except for Besu's evmtool were disabled for the batching and piping and in Section 7.1.2 and the docker benchmarks in Section 7.1.3. This allows for a better comparison of the raw numbers between the different techniques as most clients neither implement the piping technique nor provided proper Docker images that could allow for batching.

7.1.2 Batching

Section 6 provided some insights into the architecture that could be used to execute tests. Profiling the slowest Ethereum implementation - Besu - revealed that most time was spent during setup and teardown of the Java Runtime Environment. This environment has to be created on each test.

There are several techniques to reduce the time spent on setting up costly execution environments. FuzzyVM implements two approaches: "Simple batching" and "Piping". Figure 7.1 shows the differences between the naive approach and the two advanced techniques. The numbers displayed in Figure 7.1 have been obtained by executing only the Besu EVM implementation in the three modes with a single execution thread each. It shows that the advanced techniques are significantly faster than the naive approach. Besu executes 1000 tests in less than 10 seconds in piping mode, compared to more than 25 minutes when executing the tests one after another.

Figure 9.5 in the Appendix shows that once the Besu implementation switches to the batching approach, it becomes almost three times faster than Nethermind which was previously faster than Besu.

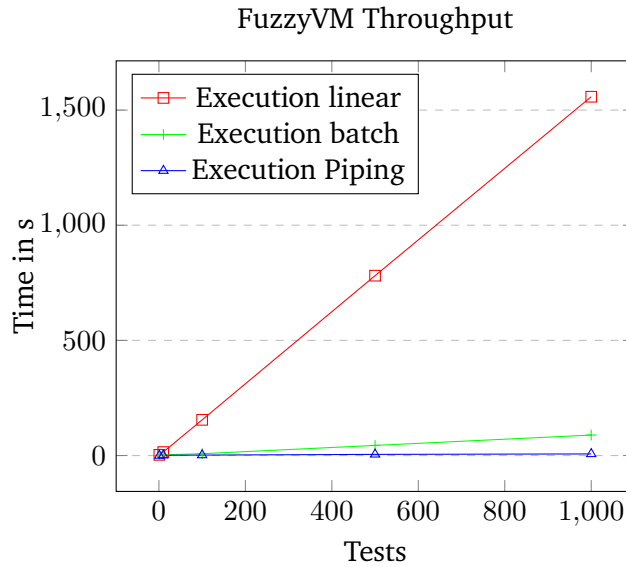


Figure 7.1: FuzzyVM throughput: Normal vs. Simple batching vs. Piping

Simple batching In the simple batching technique test cases are executed in batches. Once enough test cases are generated, a Besu environment is spun up taking as input a list of test cases. The test cases are executed one after the other and their traces are collected. Once the execution of a batch is finished the results of the batch are compared to the results of the other implementations.

This has the disadvantage that all execution results of all EVM implementations have to be cached until all tests of a batch are executed. This results in increased memory requirements of roughly $\text{Mem} = \#EVM * \text{BatchSize} * \text{TraceSize}$ instead of $\#EVM * \text{TraceSize}$. A single trace can easily reach a size of 500KB. With 5 EVM implementations and a batch size of 40 a single batch might consume 100MB memory. Running 16 batches in parallel might result in a memory usage of roughly 1.6GB. Allocating and de-allocating this memory can easily become the bottleneck in a large deployment of FuzzyVM.

Another disadvantage of this approach is that if a single test in the batch failed, the whole batch has to be tested again one by one in order to make sure that not more than one test failed and that a broken environment did not influence the results of the other tests.

Piping In the piping technique an environment is spun up on startup. This environment lives for the duration of the fuzzer. The executor can send filenames of tests over the STDIN pipe to the environment which reads and executes the tests. This approach has the advantage of reduced memory consumption. Additionally verification of the test outputs is simpler compared to the simple batching technique. The piping approach is roughly 155 times faster than the naive approach and 8 times faster than the batching approach.

A disadvantage of the piping approach is that if a test manages to break the execution environment all subsequent calls are stalled or fail and need to be rescheduled. Additionally the executor needs to check that execution environment is still running and restart it if needed. Another problem with this approach is that multi-threading becomes harder to implement. Either parallel requests to the execution environment have to be serialized or multiple execution environments have to be started. The speed of this strategy however outweighs these disadvantages.

7.1.3 Docker

FuzzyVM is supposed to run continuously on the newest build to prevent regressions. These regression can, if not found early enough, be exploited on a large scale. There have been several incidents where regressions in go-ethereum were exploited to take a number of nodes offline [61].

Fuzzing the newest version poses some challenges to the maintainers as they have to continuously update the binaries as soon as new releases are published. Running the clients inside Docker containers enables the maintainers to update the images automatically.

Docker does have an overhead of roughly 1s per execution if a container is created for each test. This overhead quickly adds up as shown in Figure 7.2. The data for Figure 7.2 was obtained by using Docker images from Besu since the other clients either did not provide Docker images for their EVM implementations or did not implement a batching mode.

Figure 7.2 additionally shows that reusing the same Docker instance provides a reasonable compromise between performance and ease of use. This technique is very similar to the batching technique described in Section 7.1.2 For the batching approach the generator and the Docker instances share a volume to which the generated tests are written.

In the future the piping approach can be used to significantly speed up the executing inside a Docker environment. This can be achieved either by implementing the piping

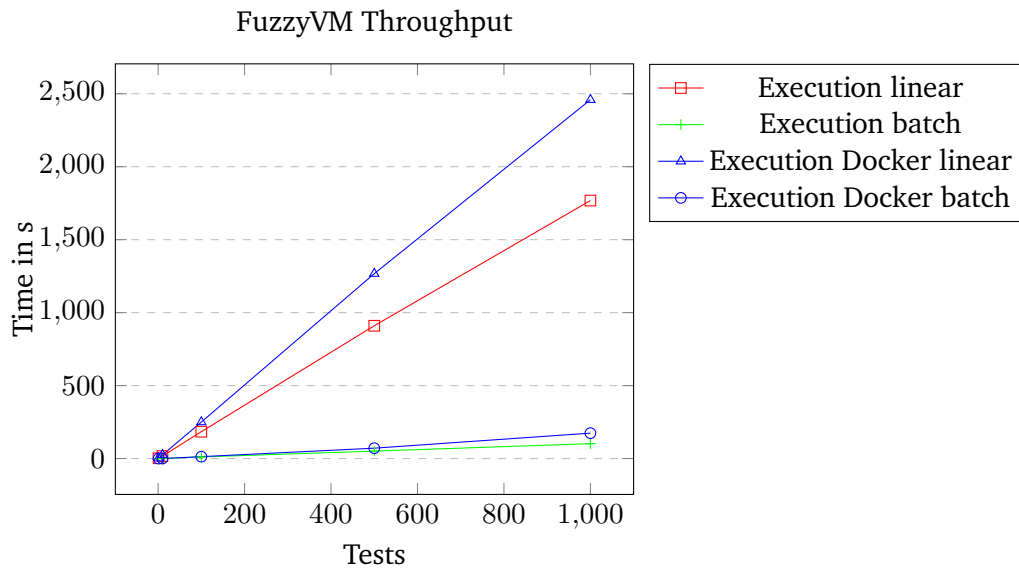


Figure 7.2: FuzzyVM throughput: Normal vs. Docker

approach in the test targets or by creating a small program that runs inside the Docker environment, listens to incoming tests and starts the EVM implementation inside the Docker. Both approaches would keep the same Docker environment running continuously, thus increasing the throughput of FuzzyVM.

7.2 Code coverage

Another metric used to evaluate fuzzing frameworks is how much code is actually executed during the tests. While Klees et al. postulate that “there is no fundamental reason that maximizing code coverage is directly connected to finding bugs” [36], they acknowledge that code coverage is a good secondary measure.

Go-fuzz measures how many lines of code are executed by instrumenting the test target. This type of code coverage is called line coverage. There are other types of code coverage like modified condition and decision coverage which tries to evaluate each decision to true and false independently of other parts of the decision [47]. Calculating more

sophisticated types of code coverage is computationally intensive, thus slowing down the fuzzing process. Therefore line coverage is typically used in fuzzing frameworks instead of modified condition and decision coverage.

Line coverage FuzzyVM achieves a code coverage of approximately 5600 lines of code after executing one million tests. While it is relatively easy to determine how many lines are covered by a fuzzer it is hard to determine how many lines of code could be covered as deciding whether a line can be covered is NP-hard. This makes it difficult to evaluate a fuzzer. Thus we propose a metric $FuzzCov$ as described by Equation 7.1 in Figure 7.3. This metric can be used to evaluate a fuzzer against an existing set of test cases. If the $FuzzCov$ metric is greater than one, the fuzzer is useful, as it covers more ground than the set of test cases. If the $FuzzCov$ metric is slightly lower than one, the fuzzer might still be useful as it might test the interaction between different parts of the code better. However if the $FuzzCov$ metric is significantly below 0.5, the generators should be rewritten to reach deeper states in the implementation.

Test coverage Due to the chosen strategy of splitting the generation and execution phase, FuzzyVM is able to generate test cases without executing them. These generated test cases can be used to test and improve code coverage by executing them manually and measuring the resulting code coverage. This approximates the code coverage that the fuzzer can achieve.

Looking at the code coverage increases understanding of the limitations that a fuzzer has during test generation. This can be used to improve the generators in the fuzzer to create better structured input in order to reach deeper states of the implementation.

Evaluation Equation 7.2 in Figure 7.3 proposes a formula to evaluate line coverage of a fuzzer. Instead of comparing the line coverage of a fuzzer against a set of test cases, it compares it against the total lines in the code. This metric, while very useful in theory, is hard to compute in practice. In order to compute the $LinesCoverable$ we need to compute the total lines of the codebase. This is difficult for large codebases as $TotalLines$ encompasses the code under test as well as all libraries the code depends on.

As a quick example, the go-ethereum codebase has around 350.000 lines of go code. More than half of it being unit tests to tests the code and code not related to EVM. Additionally it depends on roughly 70 projects which themselves have dependencies. Finally the codebase

$$FuzzCov = \frac{Coverage_{Fuzz}}{Coverage_{Test}} \quad (7.1)$$

$$TotalCov = \frac{Coverage_{Fuzz}}{LinesCoverable} \quad (7.2)$$

$$LinesCoverable = TotalLines - UnusedLines \quad (7.3)$$

Figure 7.3: Formulas to calculate $FuzzCov$ and $TotalCov$

also heavily depends on the go standard library. This makes computing the $TotalLines$ accurately a huge task.

Additionally $UnusedLines$ which are lines that can never be covered by the fuzzer, are impossible to compute for a large codebase. All lines not directly related to the EVM functionality, the state test execution or tracing are impossible to be covered by FuzzyVM. The EVM package (core/evm) in Go-ethereum for example contains roughly 5600 lines. State tests can only cover approximately half of these.

An upper bound of $TotalLines$ can be calculated as about 110,000 lines, as shown in Figure 9.6 in the Appendix. These estimates of course include test code as well as code that can not be executed during fuzzing nor during transaction processing. An estimate for $LinesCoverable$ that could theoretically be achieved by FuzzyVM is 10.000 lines of code¹.

FuzzyVM reaches a code coverage $Coverage_{Fuzz}$ of roughly 5600 lines. With an estimated $LinesCoverable$ of 10.000 lines of code, a $TotalCov$ of 0.56 or 56% can be computed using Equation 7.2 of Figure 7.3. Therefore we can conclude that the Fuzzer should be improved further to reach deeper states in the implementation.

In order to compute the more meaningful metric $FuzzCov$, the coverage of a set of test cases; $Coverage_{Test}$ has to be computed. Fortunately the Ethereum Foundation provides a set of test cases that can be used [20]. Figure 9.12 in the Appendix shows commands to compute this metric. The “ethereum/tests” test cases reach a $Coverage_{Test}$ of roughly 3.000 lines of code. Using Equation 7.1 of Figure 7.3 $FuzzCov$ can be computed to 1.86

¹Estimated as 1300 LoC in FuzzyVM, 600 LoC in goevmlab, 8100 LoC in go-ethereum packages (100 LoC tests, 2000 LoC crypto, 6000 LoC core and subpackages)

or 186%. A $FuzzCov$ greater than 1, as discussed previously, indicates that the fuzzer covers more ground than the test cases.

7.3 Ground Truth

A third metric used to evaluate fuzzers is measuring against so called ground truths [36], [27]. Hereby previously discovered bugs are reintroduced into the client under test in order to measure if and how quickly a fuzzer finds these bugs. This technique has been used for example to reproduce the Heartbleed bug in OpenSSL in around 6 hours [3].

BLOCKHASH Due to a misunderstanding between the different clients, the BLOCKHASH opcode is calculated differently in tests. In normal execution the BLOCKHASH opcode returns on the input of a blocknumber the hash of the block, if the block is within the last 256 blocks. This does not work during testing as the tests do not define previous blocks. Nethermind therefore calculates the BLOCKHASH opcode on any input as `keccak(blocknumber)` while all other clients compute it as 0. Therefore all executions of the BLOCKHASH opcode cause a test to fail.

The BLOCKHASH opcode has been deactivated in the Opcode generator as this difference was usually found by the fuzzer in less than a minute. For example, on a generated corpus with roughly 200 elements it took the generator 855 tests to create a test that executes the BLOCKHASH opcode. Even after deactivating the BLOCKHASH opcode in the generator, FuzzyVM was able to produce tests that would execute the BLOCKHASH opcode, as FuzzyVM contains an option to create fully random programs. It took FuzzyVM around 14.000 tests to produce a test that failed on the BLOCKHASH opcode when the opcode was deactivated in the generator.

Missing precompile For a second test the blake2f precompile was removed from the go-ethereum EVM². Thus every successful call to the blake2f precompile would create different return values.

The introduced bug was found after roughly 1 minute 30 seconds. FuzzyVM was started with a generated corpus of 210 elements and the bug was found between test 260 and

²<https://github.com/MariusVanDerWijden/go-ethereum/tree/istanbul-no-precompile>

280. Since FuzzyVM uses generators that generate valid calls to all precompiles, it was able to find this bug quickly.

SIGNEXTEND bug We reintroduced a crasher in Nethermind where the SIGNEXTEND opcode did not pop 2 values from the stack if the opcode fails, resulting in an invalid stack [55]. An old version of Nethermind was used to test how long FuzzyVM would take to create a test that triggers the SIGNEXTEND opcode in a way that creates an invalid stack.

The bug was found after roughly 14 hours of continuous fuzzing. FuzzyVM was started with a generated corpus of 220 and the bug was found after executing roughly 120.000 tests. FuzzyVM does not have a generator specifically designed to test this opcode, thus the failing code snippet was generated by the basic opcode generator.

7.4 Discussion

This chapter introduced three different metrics that can be used to evaluate fuzzing frameworks; throughput, code coverage and ground truths. It showed two techniques (batching and piping) that can be used to significantly speed up differential fuzzing efforts whenever one of the test target requires a long time for setup and teardown. The EVM implementation of the Besu client requires a significant time to setup the Java Runtime Environment. By executing tests in batches of 20, the execution time was reduced 17 fold compared to normal execution. Switching to the piping strategy reduced the execution time another 8 times from batching and 155 times from the initial execution.

Section 7.1.3 discussed the advantages and disadvantages of Docker. Using test targets deployed via Docker reduces the throughput of the fuzzing framework, but allows the test targets to be continuously updated. This provides a good basis for continuous fuzzing, which enables the fuzzer to find regressions in the code quickly before they are widely deployed.

This chapter also discussed the code coverage metric and proposed equations that can be used to evaluate the fuzzing efforts. Given the equations, we estimated the *TotalCov* metric as 0.56, meaning FuzzyVM covers roughly 56% of all code under test. We also computed the metric *FuzzCov* as 1.86, thus FuzzyVM covers roughly 86% more than the 2.600 integration tests in the “ethereum/tests” repository.

The third metric covered is fuzzing against ground truths. We measured the time FuzzyVM needed to find known flaws. The first flaw is a known difference in Nethermind which executes the BLOCKHASH opcode differently than other clients. This flaw was found after 855 tests with the normal opcode generator. When preventing the opcode generator from generating BLOCKHASH opcodes, it took FuzzyVM roughly 14.000 test to find the flaw with the random CreateAndCall generator. For the second flaw a precompile was removed simulating a faulty configuration. This flaw was found after roughly 260 tests. The third flaw, a bug that was previously found through fuzzing in an old version of Nethermind was found after roughly 120.000 tests.

During these times two new critical issues were found as discussed in Section 8.3.

Therefore we conclude that FuzzyVM provides a benefit for the security of the client implementation. It should be improved further to reach deeper states in the implementations and execute tests quicker once either the piping or batching strategy is implemented by Nethermind.

8 Conclusion

This thesis constructed a fuzzing framework for Ethereum Virtual Machines and optimized it in order to answer five research questions. During the work on FuzzyVM several issues in different implementations surfaced including two critical issues. This section aims to summarize our findings regarding the research questions, explain these issues and introduce an Ethereum improvement proposal called EIP-3155 that aims to make EVM fuzzing more effective.

Challenges FuzzyVM faced several challenges. A single test could take up to 5 seconds to be executed on all EVM implementations. Thus collecting benchmarks and test results is time consuming. Additionally due to pre-existing problems some false positives had to be weeded out whenever FuzzyVM found a problem.

FuzzyVM uses a fuzzing framework called go-fuzz internally. Go-fuzz needs to instrument the go-lang code of go-ethereum and FuzzyVM, this resulted in challenges since several incompatibilities between go-lang versions and go-fuzz existed ¹.

Different implementations produce slightly different outputs that need to be “massaged” into a common format in order to properly compare the results. We introduced an Ethereum Improvement Proposal (EIP) to consolidate these output formats as discussed in Section 8.2.

Additionally a variety of problems exist when the EVM implementations produce large traces. OpenEthereum for example stopped indefinitely when a test used too much storage, since the storage was printed with every executed opcode. A similar issue was also uncovered by FuzzyVM with the ReturnData field. If a call returned too much data, the go-lang JSON parser broke down, which meant that FuzzyVM stalled indefinitely. These issues were overcome by limiting the scope of the fuzzer as well as notifying the

¹<https://github.com/dvyukov/go-fuzz/issues/294>

OpenEthereum team which implemented two command line flags to limit the amount of data printed in every call ².

Responsible Disclosure During the creation of this thesis, two critical flaws were discovered as well as multiple minor issues. For minor issues, issues on the respective github repositories were opened. The maintainers of the Ethereum clients were notified of the critical issues privately in order to give them time to fix the issues and release new versions. Nethermind disguised their fix to one issue as speed improvements due to its critical nature.

Documentation Another challenge this thesis faced was that documentation of the implementations are scarce and often outdated. Most clients did not have instructions how to compile their EVM implementations into standalone binaries that can be differential fuzzed.

The documentation of the precompiled contracts in Ethereum’s yellowpaper [18] is mathematical and can not be translated to code easily. Thus most generators where created by reading the source code of go-ethereum.

8.1 Research questions

The following sections collect the findings regarding the three research questions posed in Figure 1.2.

8.1.1 Meaningful programs

The first research question posed in Figure 1.2 asked “How can a fuzzer create meaningful programs to test Turing-complete Virtual Machines and cryptographic functions?”. Section 4 showed how generators can be used to create meaningful input from randomness provided by the mutation engine. It also discussed techniques to minimize infinite loops in the generated programs.

²<https://github.com/openethereum/openethereum/issues/97>

Generators Section 4 discussed several strategies to create generators for specific cryptographic functions. These generators are highly specific to the functions in the context of the EVM, however they might be used to differential fuzz other implementations of these cryptographic functions.

Section 4 also constructed a filler class which helps to use the input mutated by go-fuzz in the generators. It allows generators to consume the input as common data types as well as provide an indication to the mutator whether the input was too short for the generator. This filler paradigm can be used by most types of generators and fuzzers based on go-fuzz or AFL and is most useful with generators that need input data of a minimal size to function.

Infinite loops Since FuzzyVM creates programs to test Turing complete Virtual Machines there have to be mechanisms to prevent generating infinite loops. Even though infinite loops are impossible due to the gas mechanism in the EVM, long running loops will still decrease the performance of FuzzyVM.

Section 4.4 discussed different strategies to prevent infinite loops. Disallowing all backward jumps is the only strategy that successfully prevents infinite loops. However this strategy has the drawback that it greatly decreases the coverage of the tests. Therefore, backward jumps are allowed in FuzzyVM if more than 10 opcodes are executed between the jump instruction and the jump destination, preventing tight loops.

Forward jumps, while not being able to create infinite loops, can end up in data segments as code and data is not separated in the EVM. FuzzyVM allows for arbitrary forward jumps as they are only valid if they end up on a 0x5b byte. Therefore the probability of an arbitrary forward jump to land in a data segment is 1/255, thus not impacting the performance of the fuzzer. Differential fuzzers for other virtual machines that allow for arbitrary jump destinations, might need to restrict the usage of arbitrary forward jumps further.

8.1.2 Architectural choices

The second research question posed in Figure 1.2 asked “How can the quality of a differential fuzzing framework be evaluated, and how do architectural choices impact it?”. Section 6 discussed different architectures for differential fuzzing frameworks and their advantages and disadvantages. Section 7 discussed different metrics to evaluate fuzzing frameworks and evaluated FuzzyVM accordingly.

Architecture Section 6 introduced the categorization of the tasks of differential fuzzing frameworks into three distinct phases. The `generation` phase in which test cases are generated, the `execution` phase in which the test cases are executed on the different implementations and the `verification` phase in which the outputs of the implementations are compared to each other.

Section 6.4 concluded that architectures that splitting `generation`, and `execution` and `verification` into two different processes (Strategy 3.1 and Strategy 3.2) provide the optimal architecture for FuzzyVM. This result holds for most differential fuzzers. Only in cases where the `verification` phase takes significant amounts of memory, a full split of the phases into three different processes might make sense.

Section 6.4 then compares strategies 3.1 and 3.2 against each other, concluding that running the implementation in sequence within the `execution` phase yields a slight advantage against running them in parallel. This can be explained due to the increased ability to set the correct amount of threads needed to fully saturate a processor and caching effects.

Evaluation Section 7 introduced three metrics that can be used to evaluate fuzzers; execution speed, code coverage and fuzzing against a ground truth.

Execution speed or throughput is usually measured in tests per second. It is important for this metric that the tests executed are identical between architectures as differences in the executed tests can have great impact on the throughput, especially when testing Virtual Machines. FuzzyVM therefore created tests from the same randomness during benchmarks. Section 7.1.2 introduced two techniques, batching and piping, that can increase the throughput of a differential fuzzing framework by decreasing the amount of time spent on creating the Java Runtime Environment. These techniques can be used by all differential fuzzing efforts if there is a significant difference in runtime between test targets.

Section 7.2 discussed topics related to code coverage. It establishes formulas that can be used to evaluate fuzzers by calculating the code coverage achieved relative to the coverage provided by a static set of test cases. The section computes that FuzzyVM covers around 86% more code than the reference test cases provided in the “ethereum/tests” repository [20]. The established formulas can be used to evaluate fuzzing efforts whenever a set of integration tests is present.

The third metric - ground truth - is discussed in Section 7.3. Hereby an intentional bug is introduced to measure how long it takes the fuzzing framework to uncover the bug. The section introduces three different bugs in the clients under test.

- A difference that is currently in the testing code (BLOCKHASH)
- An intentionally placed bug in go-ethereum (missing blake2f precompile)
- A bug that was recently fixed in Nethermind (SIGNEXTEND)

These three bugs represent different problems that existed or might exist in the code and that could cause a split of the Ethereum blockchain.

FuzzyVM found the first bug after executing 855 tests. Even when preventing the opcode generator from generating BLOCKHASH opcodes, the bug was encountered after roughly 14.000 tests by the generator that creates and calls a random precompile. The second bug was found after executing 260 tests. FuzzyVM uses generators that can generate valid calls to all precompiles. Therefore, it was expected that this bug was found quickly. The third error was found after executing 140.000 tests since it needs a very specific setup to trigger it.

During the fuzzing for ground truths, FuzzyVM found critical bugs in Nethermind and Besu which are discussed in Section 8.3. From this we can conclude that FuzzyVM is able to reliably reproduce old errors and also find new, previously unknown errors.

8.1.3 Corpus

The third research question posed in Figure 1.2 asked “How much benefit can an existing corpus bring and how can the quality of an initial corpus be improved?”. Section 5 explored the benefit of an existing corpus in decreasing the time needed to create “interesting” inputs for the test target. It also explored the problems of corpus elements that are shorter or larger than needed. While corpus elements that are too large reduce the efficiency of the fuzzer, elements that are too short will confuse the mutator and might harm the fuzzing efforts.

Section 5 also introduces a novel algorithm to generate valid corpus elements. It uses a binary search over the length of the input to calculate the minimal size needed of a corpus element.

The section also discusses the variation of the minimal length of corpus elements. In order to do so, 10.000 random valid corpus elements are sampled using aforementioned

algorithm and their size is plotted in Figure 5.4. It showed that most corpus elements need 5000 bytes or less to produce valid state tests. However there are corpus elements that need more than 12.000 bytes to create a valid test. This variance between the length of the corpus elements is generally harmful as it means slight variations to the corpus elements might result in elements that are too short or too long.

This type of evaluation can be used by other projects that use a filler-based approach to convert the randomness of the mutator into datatypes. Generally the variance between the data needed should be as small as possible with high variance being an indicator that the fuzzing target is too large. However, decreasing the variance in input lengths can also negatively impact the performance of the fuzzer as it might not be able to create certain tests.

8.2 EIP-3155

The EVM implementations output lines of JSON objects. The format of these lines differed significantly between implementations. In order to compare them some fields had to be dropped. Parity (later OpenEthereum) does not support the fields `GasCost`, `MemorySize` and `RefundCounter` while Nethermind does not support `ReturnStack` and `ReturnData`. Additionally the `Storage` and `Memory` fields have to be dropped due to performance reasons. Dropping fields decreases the effectiveness of the differential fuzzer as less differences can be found. A shared format for all clients would increase the effectiveness, force implementations to add the missing fields and convince other EVM implementations to adopt the format to become part of the differential fuzzing efforts.

During this work Martin Holst Swende and I proposed an EIP (Ethereum Improvement Proposal) to standardize the tracing formats named EIP-3155[40]. It specifies the datatypes used for the different fields which can be found in Figure 9.9 in the Appendix. During the standardization process it became apparent that this format can be used for other tracing related tasks too, such as the RPC methods “`debug_traceBadBlockToFile`” or “`debug_traceBlockToFile`”. Thus the output fields are split into required and optional fields as shown in Figure 9.10 in the Appendix. The optional fields like *opName* and *error* can be ignored during fuzzing as they contain no consensus critical information.

After the test is executed, the EVM implementations print a summary of the execution which contains the state root. This summary varies widely between the different imple-

mentations. The EIP also describes a common format for these summaries which can be seen in Figure 9.11 in the Appendix.

EIP-3155 is still in the DRAFT state. Both the go-ethereum and Besu teams have signaled willingness to adapt the proposed standard. EthereumJS and Aleth(formerly cpp-ethereum) have signaled interest in the proposal. We hope that other implementations will adopt EIP-3155 too, which would improve the ability to differential test EVM implementations significantly.

8.3 Issues and Recommendations

While working on FuzzyVM, several issues surfaced. Most of the issues are related to the tools that execute the tests and are not inherent to the EVM functionality. Figure 8.1 provides an overview of the issues found. Their criticality is categorized in five categories: none, low, medium, high and critical.

- None: These are issues that are related to efficiency or compatibility. They have no impact on the fuzzing efforts or the correctness of the network. Feature requests are also marked none.
- Low: These issues have some impact on the execution of the fuzzer since they may cause the fuzzer to create false positives.
- Medium: Issues marked medium impact the fuzzer severely by creating a large number of false positives
- High: Issues marked high impact consensus in a way that does not produce differing state roots.
- Critical: Issues marked critical impact the consensus and can be used to produce differing state roots.

1 and 2: Execution of non-existent or empty tests Issues one and two impact the nethtest binary of Nethermind. The program panicked when called with a filename that did not exist ³ and when called with an empty file⁴ instead of shutting down. Both issues

³<https://github.com/NethermindEth/nethermind/issues/2344>

⁴<https://github.com/NethermindEth/nethermind/issues/2344>

| No. | Criticality | Client | Explanation |
|-----|-------------|-----------------|---|
| 1 | low | Nethermind | Execution panics on non-existent test |
| 2 | low | Nethermind | Execution panics on empty test |
| 3 | medium | Geth & Besu | Execution of broken state test without error |
| 4 | medium | Nethermind | Blockhash executed as keccak(blockno) |
| 5 | medium | All | Gas reporting difference in summary |
| 6 | none | Besu | Ignoring the <code>-nomemory</code> flag |
| 7 | none | Geth | ReturnData field encoded in base64 |
| 8 | none | Besu | Ignoring the <code>-version</code> flag |
| 9 | none | Besu | No support for piping multiple tests into test tool |
| 10 | medium | OpenEthereum | Deadlock on too many SSTORE opcodes |
| 11 | none | OE & Nethermind | Do not provide docker images |
| 12 | medium | All | large ReturnData size breaks test execution |
| 13 | critical | Nethermind | COPY opcodes error on zero length |
| 14 | critical | Besu | ModeExp errors on zero length, big modulus |

Figure 8.1: Issues

do not impact the test execution of FuzzyVM as it guarantees the generation of state tests in the correct state test format.

3: Execution of broken state tests FuzzyVM generated a state test that was not executed by both Nethermind and OpenEthereum due to insufficient balance of the sender⁵. While OpenEthereum returned an error, Nethermind panicked. Go-ethereum and Besu on the other hand neither execute any operations nor report that the sender did not have sufficient balance to execute the test. The suggestion here would be for Nethermind to return an error instead of panicking and for go-ethereum and Besu to return an error instead of executing the test as if it would succeed.

4: Blockhash opcode Nethermind is the only client that executes the BLOCKHASH opcode during state tests as `keccak256(blocknumber)`⁶. All other clients return 0. The state test documentation does define the BLOCKHASH opcode as `keccak256(blocknumber)` in state tests which means the other clients should adhere to the spec. This difference resulted

⁵<https://gist.github.com/MariusVanDerWijden/008b91a61de4b0fb831b72c24600ef59>

⁶<https://gist.github.com/MariusVanDerWijden/97fe9eb1aac074f7ccf6aef169aaadaa>

in most tests failing, thus the opcode generator was modified to not generate BLOCKHASH opcodes.

5: Gas reporting differences in summary All clients provide a short summary after executing a transaction. These summaries differ slightly as discussed in Section 8.2. One of the major differences is that the clients report the total gas used differently when a transaction runs out of gas. Nethermind and go-ethereum report the total available gas that was used in their summary, while OpenEthereum and Nethermind report the total available gas plus the gas that would be needed for executing the next operation. This makes it currently impossible to compare the gas used section of the summaries against one another⁷.

6: Ignoring the `–nomemory` flag The `–nomemory` flag is provided by the clients to limit the size of the traces. Besu used to provide the flag, but ignored it during execution and printed the memory regardless⁸. This issue was fixed after reporting it to the Besu team.

7: Go-ethereum encodes the `ReturnData` field in base64 The go-ethereum client used to encode the `ReturnData` field in base64 while others encoded it as hexadecimal strings⁹. This meant that FuzzyVM needed special logic to decode the `ReturnData` field from geth. This issue was fixed after reporting it to the go-ethereum team.

8: Ignoring the `–version` flag The `–version` flag is provided by the clients to print out their current version. Besu ignored the `–version` flag which made it difficult to tell the version of the evmtool with which a certain issue occurred¹⁰. This issue was fixed after reporting it to the Besu team.

9: No support for piping multiple tests into test tool This was a feature request to implement the piping paradigm in Besu’s EVM implementation¹¹. Besu supported batching by calling the evmtool with multiple filenames as discussed in Section 7.1.2. However, it

⁷<https://gist.github.com/MariusVanDerWijden/b162e5f58e198402dc1d92652e728489>

⁸<https://gist.github.com/MariusVanDerWijden/31b67a54ba7cc09d4aea605d7283f0fd>

⁹<https://github.com/ethereum/go-ethereum/issues/21709>

¹⁰<https://github.com/hyperledger/besu/issues/1471>

¹¹<https://github.com/hyperledger/besu/issues/1470>

did not support the piping paradigm. The piping paradigm was later implemented and tested to produce the numbers used in Section 7.1.2.

10: OpenEthereum deadlocks on SSTORE opcodes OpenEthereum did output the whole storage for every opcode in the traces. This resulted in OpenEthereum deadlocking on tests that execute many SSTORE opcodes or touch a lot of storage^{12 13}. This issue was fixed after reporting it to the OpenEthereum team. OpenEthereum now provides the option to disable memory and storage output during tracing.

11: Docker images Both Nethermind and OpenEthereum did not provide a docker image for their EVM implementations which made continuous fuzzing using docker impossible^{14 15}. Nethermind provides docker images for their EVM implementation now, while the issue is still open in OpenEthereum.

12: Large ReturnData Most clients provide flags to limit the amount of storage and memory that is printed per opcode. This allows us to keep traces as small as possible as FuzzyVM often breaks when reading large traces. Currently no client implements a way to limit the amount of ReturnData that is returned by a call. FuzzyVM creates tests that store large arrays in memory and call the identity precompile. The call returns with ReturnData of multiple kilobytes in a single JSONl line which breaks the fuzzer¹⁶.

13: Copy Methods FuzzyVM found a critical flaw in Nethermind that caused Nethermind to halt execution and produce different state roots than other clients. The CALLDATACOPY, CODECOPY, EXTCODECOPY and RETURNDATACOPY opcodes consume three items from the stack; the destination, source and length of the data that should be copied. The Nethermind client halted execution when the length was zero for these four opcodes. Thus calling them with length set to zero would alter the execution and provide an invalid state root¹⁷.

¹²<https://github.com/openethereum/openethereum/issues/97>

¹³<https://gist.github.com/MariusVanDerWijden/a6d0239e0dbe87e9d0bdd062bce75e42>

¹⁴<https://github.com/NethermindEth/nethermind/issues/2599>

¹⁵<https://github.com/openethereum/openethereum/issues/190>

¹⁶<https://gist.github.com/MariusVanDerWijden/3023a2397e68f12d48b66284cb24f335>

¹⁷<https://gist.github.com/MariusVanDerWijden/c5e1f3e39bdfae6d4dd2c901626f9350>

Upon notifying the Nethermind team of these issues, they were promptly fixed and a new version of Nethermind was released¹⁸.

FuzzyVM found the CALLDATACOPY and CODECOPY errors after roughly 150.000 test executions. The EXTCODECOPY and RETURNDATACOPY errors were found on manual inspection of the source code. FuzzyVM contains no generator that explicitly searches for these kind of flaws. Finding this error is a good validation that FuzzyVM works as intended.

14: ModExp The second critical flaw found by FuzzyVM concerns the ModExp precompile in the Besu client. The ModExp precompile consumes six parameters from the stack; the base length, modulus length, exponent length, base, modulus and exponent. If the base length and the modulus length are zero, Besu would read the exponent length as well as all other parameters anyway. If the exponent was set to a high number it could overflow causing the client to crash.

This critical flaw was even more critical than Nethermind's copy method error as it could crash the node meaning all Besu nodes that would receive a block with a transaction like this would crash.

FuzzyVM found the ModExp error in Besu after roughly 200.000 test executions. The error was reported to the Besu team and quickly fixed¹⁹.

8.4 Acknowledgements

I would like to thank the maintainers of the different Ethereum clients for resolving most of the issues and recommendations I submitted in a timely fashion and for allowing me to publish my findings. I would especially like to thank Martin Holst Swende from the go-ethereum team for his support on EIP-3155, Danno Ferrin and Ratan (Rai) Sur from the Besu team, and Tomasz Kajetan Stańczak and Sebastian Dremo from Nethermind. As well as Rene Lubov from go-ethereum and Veronika Kaletta.

¹⁸<https://github.com/NethermindEth/nethermind/pull/2880/files>

¹⁹<https://github.com/hyperledger/besu/pull/2014>

9 Outlook

This chapter discusses further topics that should be implemented as part of FuzzyVM or should be researched in the future.

Continuous fuzzing The Ethereum Foundation has signaled willingness to provide server capacity to run FuzzyVM continuously on new client releases of the major clients. In order to continuously fuzz multiple test targets, the reporting capabilities of FuzzyVM need to be improved. Additionally all clients should provide docker images for their EVM binaries in order to make updating the clients easier.

Reuse docker instances Using docker to provide the test targets decreases the throughput of FuzzyVM significantly as shown in Section 7.1.3. Batching tests together does increase the throughput when using docker as less time has to be spent on setting up the instances. However, reusing the same docker instance, by either implementing a piping mode in all clients or creating a separate program that runs inside the docker image and calls the EVM implementation from within the docker image, can significantly increase the throughput of FuzzyVM.

Ensemble fuzzing Chen et. al. showed that using different fuzzing engines with the same corpus can improve fuzzing success significantly[11]. Go-fuzz provides an interface for ensemble fuzzing, which allows test target instrumented by go-fuzz to be tested with inputs from libfuzzer. Since libfuzzer implements different mutation algorithms from go-fuzz this combination of libfuzzer and go-fuzz might create more interesting state tests than go-fuzz alone.

Another idea worth pursuing in the future would be to write a small wrapper around FuzzyVM so it can be fuzzed by American Fuzzy Lop.

Generators for upcoming forks The Ethereum ecosystem evolves through incremental updates called hard-forks. These hard-forks introduce new operations and precompiles into the EVM. In order to properly test this forks, FuzzyVM will be extended to support generators for these new schemes. In the upcoming Berlin hard-fork three new schemes are discussed. EIP-2537 proposes 8 new precompiles for cryptographic primitives on the BLS-12381 curve [65]. EIP-2315 proposes a mechanism to create subroutines inside the EVM [13]. EIP-2930 proposes optional access lists that can be used to tell the EVM which storage slots are going to be touched in order to reduce gas usage and enable prefetching of the storage slots [9]. These proposals should be fuzzed on their own as well as together with all other elements of the EVM inside FuzzyVM to provide good integration tests.

Code coverage Currently FuzzyVM only takes the code coverage within the EVM implementation of go-ethereum into account. In the future, it would be beneficial to also measure the code coverage of tests in the other EVM implementations as language-specific behaviour might lead to differences. Measuring the code coverage of implementations in multiple different languages is exceedingly hard and is topic of research.

Duplicate tests Currently, it is possible for FuzzyVM to generate multiple identical tests. Duplicated tests decrease the efficiency of FuzzyVM since the same functionality is tested multiple times. FuzzyVM could save the generated state tests under their hash which would eliminate duplicate tests as long as the duplicate has not been executed.

Corpus variance The length of the corpus elements currently display a very high variance. This variance reduces the effectiveness of go-fuzz. It is very difficult to reduce this variance as reducing the length of the corpus elements also decreases the paths that are taken in the EVM. More research should be done on decreasing the variance while keeping the possibilities of programs that can be created as high as possible.

Reduce dependence on generator versioning Currently the corpus is highly dependent on the version of the generator. If the generators are updated, the old corpus does not fit to it anymore as it is highly dependent on the consumption patterns of the generators. More research should be done into reducing the dependence of the corpus on the version of the generator.

Corpus from existing contracts Another interesting problem for further research would be to find a way to turn existing smart contracts into corpus elements. This would allow for the use of interesting contracts, that for example call a precompile and verify the result, to be used as a basis for new tests. The resulting program could be supplemental to FuzzyVM and allow for “hand crafted” corpus elements. It might also be used to transfer existing corpus elements from one generator version to another.

Bibliography

- [1] The go-ethereum Authors. *Go-Ethereum Block Format*. github.com/ethereum/go-ethereum/core/types/block.go. Accessed: 2020-11-30. 2020.
- [2] The go-ethereum authors. *Integration script for oss-fuzz*. <https://github.com/ethereum/go-ethereum/blob/8f03e3b107c0f7a39de31a9e7deb658431a937ac/oss-fuzz.sh>. Accessed: 2020-11-30. 2020.
- [3] Hanno Böck. *How Heartbleed could've been found*. <https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html>. Accessed: 2020-11-30. 2015.
- [4] D. Boneh et al. *draft-irtf-cfrg-bls-signature-02*. <https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02>. Accessed: 2020-09-21. 2020.
- [5] Dan Boneh and Matt Franklin. "Identity-Based Encryption from the Weil Pairing". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 213–229. ISBN: 978-3-540-44647-7.
- [6] Daniel R. L. Brown. *Standards for Efficient Cryptography 2 (SEC 2)*. <https://www.secg.org/sec2-v2.pdf>. Accessed: 2020-12-30. 2010.
- [7] Vitalik Buterin. *Ethereum Whitepaper*. <https://ethereum.org/en/whitepaper/>. Accessed: 2020-11-30. 2013.
- [8] Vitalik Buterin. *Security alert [11/24/2016]: Consensus bug in geth v1.4.19 and v1.5.2*. <https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2/>. Accessed: 2020-11-30. 2016.
- [9] Vitalik Buterin and Martin Holst Swende. *EIP-2930: Optional access lists*. <https://eips.ethereum.org/EIPS/eip-2930>. Accessed: 2020-09-21. 2020.
- [10] Alan Cao. *DeepState Now Supports Ensemble Fuzzing*. <https://blog.trailofbits.com/2019/09/03/deepstate-now-supports-ensemble-fuzzing/>. Accessed: 2020-11-30. 2019.

-
-
- [11] Yuanliang Chen et al. “EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1967–1983. ISBN: 978-1-939133-06-9. URL: <https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang>.
 - [12] CoinMarketCap. *Top 100 Cryptocurrencies by Market Capitalization*. <https://coinmarketcap.com/>. Accessed: 2020-11-30. 2020.
 - [13] Greg Colvin and Martin Holst Swende. *EIP-2315: Simple Subroutines for the EVM*. <https://eips.ethereum.org/EIPS/eip-2315>. Accessed: 2020-09-21. 2019.
 - [14] Ethereum Community. *Ethereum Tests*. <https://ethereum-tests.readthedocs.io/en/latest/>. Accessed: 2020-09-21. 2020.
 - [15] G. Destefanis et al. “Smart contracts vulnerabilities: a call for blockchain software engineering?” In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 2018, pp. 19–25.
 - [16] Bitcoin Core Developers. *Bitcoin Core integration/staging tree*. <https://github.com/bitcoin/bitcoin>. Accessed: 2020-11-30. 2020.
 - [17] The Bitcoin Core developers. *Fuzzing Bitcoin Core*. <https://github.com/bitcoin/bitcoin/blob/master/doc/fuzzing.md>. Accessed: 2020-11-30. 2020.
 - [18] Vitalik Buterin Dr. Gavin Wood. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. <https://github.com/ethereum/wiki/wiki/White-Paper>. Accessed: 2020-11-30. 2013.
 - [19] Google Fuzzing Forum. *What makes a good fuzz target*. <https://github.com/google/fuzzing/blob/master/docs/good-fuzz-target.md>. Accessed: 2020-12-06. 2020.
 - [20] Ethereum Foundation. *Ethereum Consensus Tests*. <https://github.com/ethereum/tests>. Accessed: 2020-09-21. 2020.
 - [21] Inc. Free Software Foundation. *13.16 File Locks*. https://www.gnu.org/software/libc/manual/html_node/File-Locks.html. Accessed: 2020-09-21. 2020.
 - [22] Ying Fu et al. “EVMFuzz: Differential Fuzz Testing of Ethereum Virtual Machine”. In: *CoRR* abs/1903.08483 (2019). arXiv: 1903.08483. URL: <http://arxiv.org/abs/1903.08483>.

-
- [23] Ying Fu et al. “EVMFuzzer: Detect EVM Vulnerabilities via Fuzz Testing”. In: ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 1110–1114. ISBN: 9781450355728. DOI: 10.1145/3338906.3341175. URL: <https://doi.org/10.1145/3338906.3341175>.
- [24] bitfly GmbH. *Ethereum Mainnet Statistics*. <https://www.ethernodes.org/>. Accessed: 2020-11-30. 2020.
- [25] Samuel Groß. *FuzzIL: Coverage Guided Fuzzing for JavaScript Engines*. <https://saelo.github.io/papers/thesis.pdf>. Accessed: 2020-11-30. 2018.
- [26] Osman Gazi Güçlütürk. *The DAO Hack Explained: Unfortunate Take-off of Smart Contracts*. <https://medium.com/@oguccluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562>. Accessed: 2020-11-30. 2018.
- [27] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. *Magma: A Ground-Truth Fuzzing Benchmark*. <https://arxiv.org/pdf/2009.01120.pdf>. Accessed: 2020-11-30. 2020.
- [28] Tjaden Hess et al. *EIP-152: Add BLAKE2 compression function ‘F’ precompile*. <https://eips.ethereum.org/EIPS/eip-152>. Accessed: 2020-09-21. 2016.
- [29] Katie Hockman. *Design Draft: First Class Fuzzing*. <https://go.googlesource.com/proposal/+master/design/draft-fuzzing.md>. Accessed: 2020-11-30. 2020.
- [30] Google Inc. *Continuous Integration*. <https://google.github.io/oss-fuzz/getting-started/continuous-integration/>. Accessed: 2020-12-30. 2020.
- [31] Google Inc. *OSS-Fuzz: Continuous Fuzzing for Open Source Software*. <https://github.com/google/oss-fuzz>. Accessed: 2020-11-30. 2020.
- [32] Google Inc. *Structure-Aware Fuzzing with libFuzzer*. <https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md>. Accessed: 2021-02-13. 2020.
- [33] Bo Jiang, Ye Liu, and W. K. Chan. “ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: Association for Computing Machinery, 2018, pp. 259–269. ISBN: 9781450359375. DOI: 10.1145/3238147.3238177. URL: <https://doi.org/10.1145/3238147.3238177>.

-
-
- [34] Yolan Romailier JP Aumasson. *Automated Testing of Crypto Software Using Differential Fuzzing*. <https://www.blackhat.com/docs/us-17/wednesday/us-17-Aumasson-Automated-Testing-Of-Crypto-Software-Using-Differential-Fuzzing.pdf>. Accessed: 2020-11-30. 2017.
- [35] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. Chapman and hall/crc cryptography and network security series. CRC Press/Taylor and Francis, Boca Raton, second edition edition. 2015.
- [36] George Klees et al. *Evaluating Fuzz Testing*. <https://arxiv.org/pdf/1808.09700.pdf>. Accessed: 2020-11-30. 2018.
- [37] Jonathan Knudsen. *Fuzzing Bitcoin with the Defensics SDK part 1: Create your network*. <https://www.synopsys.com/blogs/software-security/defensics-sdk-fuzzing-bitcoin/>. Accessed: 2020-11-30. 2018.
- [38] LLVM Maintainer. *libFuzzer – a library for coverage-guided fuzz testing*. <https://llvm.org/docs/LibFuzzer.html>. Accessed: 2020-11-30. 2020.
- [39] Valentin J. M. Manès et al. “Fuzzing: Art, Science, and Engineering”. In: *CoRR* abs/1812.00140 (2018). arXiv: 1812.00140. URL: <http://arxiv.org/abs/1812.00140>.
- [40] Marius van der Wijden Martin Holst Swende. *EIP-3155: EVM trace specification [DRAFT]*. <https://eips.ethereum.org/EIPS/eip-3155>. Accessed: 2020-12-19. 2020.
- [41] Péter Szilágyi Martin Holst Swende. *Hive*. <https://hivetests.ethdevops.io/>. Accessed: 2020-11-30. 2020.
- [42] William M. McKeeman. “Differential Testing for Software”. In: *DIGITAL TECHNICAL JOURNAL* 10.1 (1998).
- [43] Alfred Menezes. *An Introduction to Pairing-Based Cryptography*. <https://www.math.uwaterloo.ca/~ajmeneze/publications/pairings.pdf>. Accessed: 2020-09-21. 1991.
- [44] Peter Schwabe Michael Naehrig Ruben Niederhagen. *New software speed records for cryptographic pairings*. <https://cryptojedi.org/papers/dclxvi-20100714.pdf>. Accessed: 2020-09-21. 2010.
- [45] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. <https://bitcoin.org/bitcoin.pdf>. Accessed: 2020-11-30. 2008. URL: <https://bitcoin.org/bitcoin.pdf>.

-
- [46] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. “DifFuzz: Differential Fuzzing for Side-Channel Analysis”. In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE ’19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 176–187. DOI: 10.1109/ICSE.2019.00034. URL: <https://doi.org/10.1109/ICSE.2019.00034>.
- [47] Tanay Kanti Paul and Man Fai Lau. “A Systematic Literature Review on Modified Condition and Decision Coverage”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC ’14. Gyeongju, Republic of Korea: Association for Computing Machinery, 2014, pp. 1301–1308. ISBN: 9781450324694. DOI: 10.1145/2554850.2555004. URL: <https://doi.org/10.1145/2554850.2555004>.
- [48] Andrea Pinna et al. “A Massive Analysis of Ethereum Smart Contracts. Empirical study and code metrics”. In: *IEEE Access* (June 2019). DOI: 10.1109/ACCESS.2019.2921936.
- [49] L. Prechelt. “An empirical comparison of seven programming languages”. In: *Computer* 33.10 (2000), pp. 23–29. DOI: 10.1109/2.876288.
- [50] Sigma Prime. *beacon-fuzz*. <https://github.com/sigp/beacon-fuzz>. Accessed: 2020-11-30. 2020.
- [51] Ratika. *Add flag to disable storage output in openethereum-evm tool*. <https://github.com/openethereum/openethereum/pull/115>. Accessed: 2021-01-30. 2020.
- [52] Gaganjeet Singh Reen and Christian Rossow. *DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC*. <https://publications.cispa-saarland/3220/1/DPIFuzz.pdf>. Accessed: 2020-12-30. 2020.
- [53] Camilla Russo. *The Infinite Machine: How an Army of Crypto-Hackers Is Building the Next Internet with Ethereum*. HarperCollins Publishers, first edition. 2020.
- [54] K. Serebryany. “Continuous Fuzzing with libFuzzer and AddressSanitizer”. In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016, pp. 157–157.
- [55] Tomasz Kajetan Stańczak. *State tracer changes*. <https://github.com/NethermindEth/nethermind/commit/03aa0828cd79d2bb23d5bcef022582f49ae95250>. Accessed: 2020-12-06. 2020.
- [56] ETH gas station. *ETH Gas-Time-Price Estimator*. <https://ethgasstation.info>. Accessed: 2020-11-30. 2018.
- [57] Dima Stebaev. *index out of range during full node sync*. <https://github.com/ethereum/go-ethereum/issues/21367>. Accessed: 2020-11-30. 2020.

-
-
- [58] Martin Holst Swende. *crypto/bn256: improve bn256 fuzzer*. <https://github.com/ethereum/go-ethereum/pull/21815>. Accessed: 2020-09-21. 2020.
- [59] Martin Holst Swende. *Go evmlab*. github.com/holiman/goevmlab. Accessed: 2020-11-30. 2020.
- [60] Nick Szabo. “Formalizing and Securing Relationships on Public Networks”. In: *First Monday* 2.9 (1997). Accessed: 2020-11-30.
- [61] Péter Szilágyi. *Geth v1.9.17 Post Mortem*. <https://gist.github.com/karalabe/e1891c8a99fdc16c4e60d9713c35401f>. Accessed: 2020-12-06. 2020.
- [62] Geth team. *Geth security release: Critical patch for CVE-2020-28362*. https://blog.ethereum.org/2020/11/12/geth_security_release/. Accessed: 2021-01-30. 2020.
- [63] Unknown. *Gas Costs from Yellow Paper – EIP-150 Revision (1e18248 - 2017-04-12)*. https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWc0lRem_m009GtSKEKraSf07Frgx18pNU. Accessed: 2020-11-30. 2017.
- [64] Christian Reitwiessner Vitalik Buterin. *EIP-214: New opcode STATICCALL*. <https://eips.ethereum.org/EIPS/eip-214>. Accessed: 2020-09-21. 2017.
- [65] Alex Vlasov. *EIP-2537: Precompile for BLS12-381 curve operations*. <https://eips.ethereum.org/EIPS/eip-2537>. Accessed: 2020-09-21. 2020.
- [66] Guido Vranken. *Cryptofuzz - Differential cryptography fuzzing*. <https://github.com/guidovranken/cryptofuzz>. Accessed: 2020-11-30. 2020.
- [67] Dmitry Vyukov. *go-fuzz: randomized testing for Go*. <https://github.com/dvyukov/go-fuzz>. Accessed: 2020-11-30. 2020.
- [68] Marius van der Wijden. *Fuzzing the BLS-Precompiles*. <https://medium.com/@m.vanderwijden1/fuzzing-the-bls-precompiles-ba3728dec622>. Accessed: 2020-09-21. 2020.
- [69] Xuejun Yang et al. *Finding and Understanding Bugs in C Compilers*. <https://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf>. Accessed: 2020-12-06. 2011.
- [70] Michał Zalewski. *american fuzzy lop*. <https://lcamtuf.coredump.cx/afl/>. Accessed: 2020-11-30. 2020.



Appendix

- Remove/Insert/Duplicate/Copy a random range of bytes
- Flip a random bit
- Set a byte to a random value
- Swap two bytes
- Add/subtract from a byte/uint16/uint32/uint64
- Replace a byte/uint16/uint32 with a interesting value (0, Max_Value-1)
- Replace a single/multibyte ASCII digit with another digit
- Splice/Insert parts of another corpus element
- Insert/Replace a literal

Figure 9.1: Mutations implemented by go-fuzz [67]

| Value | Mnemonic | Gas Used |
|-------|--------------|----------|
| 0x00 | STOP | 0 |
| 0x01 | ADD | 3 |
| 0x02 | MUL | 5 |
| 0x03 | SUB | 3 |
| 0x04 | DIV | 5 |
| 0x05 | SDIV | 5 |
| 0x06 | MOD | 5 |
| 0x07 | SMOD | 5 |
| 0x08 | ADDMOD | 8 |
| 0x09 | MULMOD | 8 |
| 0x0a | EXP | FORMULA |
| 0x0b | SIGNEXTEND | 5 |
| 0x10 | LT | 3 |
| 0x11 | GT | 3 |
| 0x12 | SLT | 3 |
| 0x13 | SGT | 3 |
| 0x14 | EQ | 3 |
| 0x15 | ISZERO | 3 |
| 0x16 | AND | 3 |
| 0x17 | OR | 3 |
| 0x18 | XOR | 3 |
| 0x19 | NOT | 3 |
| 0x1a | BYTE | 3 |
| 0x20 | SHA3 | FORMULA |
| 0x30 | ADDRESS | 2 |
| 0x31 | BALANCE | 400 |
| 0x32 | ORIGIN | 2 |
| 0x33 | CALLER | 2 |
| 0x34 | CALLVALUE | 2 |
| 0x35 | CALLDATALOAD | 3 |
| 0x36 | CALLDATASIZE | 2 |
| 0x37 | CALLDATACOPY | FORMULA |
| 0x38 | CODESIZE | 2 |
| 0x39 | CODECOPY | FORMULA |
| 0x3a | GASPRICE | 2 |
| 0x3b | EXTCODESIZE | 700 |
| 0x3c | EXTCODECOPY | FORMULA |

Figure 9.2: Table of OpCodes [63][18]

| Value | Mnemonic | Gas Used |
|-------------|--------------|----------|
| 0x40 | BLOCKHASH | 20 |
| 0x41 | COINBASE | 2 |
| 0x42 | TIMESTAMP | 2 |
| 0x43 | NUMBER | 2 |
| 0x44 | DIFFICULTY | 2 |
| 0x45 | GASLIMIT | 2 |
| 0x50 | POP | 2 |
| 0x51 | MLOAD | 3 |
| 0x52 | MSTORE | 3 |
| 0x53 | MSTORE8 | 3 |
| 0x54 | SLOAD | 200 |
| 0x55 | SSTORE | FORMULA |
| 0x56 | JUMP | 8 |
| 0x57 | JUMPI | 10 |
| 0x58 | PC | 2 |
| 0x59 | MSIZE | 2 |
| 0x5a | GAS | 2 |
| 0x5b | JUMPDEST | 1 |
| 0x60 – 0x7f | PUSH* | 3 |
| 0x80 – 0x8f | DUP* | 3 |
| 0x90 – 0x9f | SWAP* | 3 |
| 0xa0 | LOG0 | FORMULA |
| 0xa1 | LOG1 | FORMULA |
| 0xa2 | LOG2 | FORMULA |
| 0xa3 | LOG3 | FORMULA |
| 0xa4 | LOG4 | FORMULA |
| 0xf0 | CREATE | 32000 |
| 0xf1 | CALL | FORMULA |
| 0xf2 | CALLCODE | FORMULA |
| 0xf3 | RETURN | 0 |
| 0xf4 | DELEGATECALL | FORMULA |
| 0xfe | INVALID | NA |
| 0xff | SELFDESTRUCT | FORMULA |

Figure 9.3: Table of OpCodes [63][18] cont.



| Name | Datatype | Description |
|-------------|----------------|--|
| ParentHash | common.Hash | Hash of the previous block |
| UncleHash | common.Hash | Hash of an included uncle |
| Coinbase | common.Address | Address of the miner who mined the block |
| Root | common.Hash | Root of the world state |
| TxHash | common.Hash | Root hash of the transaction trie |
| ReceiptHash | common.Hash | Root hash of the receipts |
| Bloom | Bloom | Bloom filter over the transaction logs |
| Difficulty | *big.Int | Difficulty that the block had to reach |
| Number | *big.Int | Number of the block |
| GasLimit | uint64 | Gas limit of the block |
| GasUsed | uint64 | Gas used by all transactions |
| Time | uint64 | Timestamp of the block |
| Extra | []byte | Extra data that can be set by the miner |
| MixDigest | common.Hash | Part of the PoW, makes verification easier |
| Nonce | BlockNonce | Nonce used in the PoW |

Figure 9.4: Block Header format [1], [18]

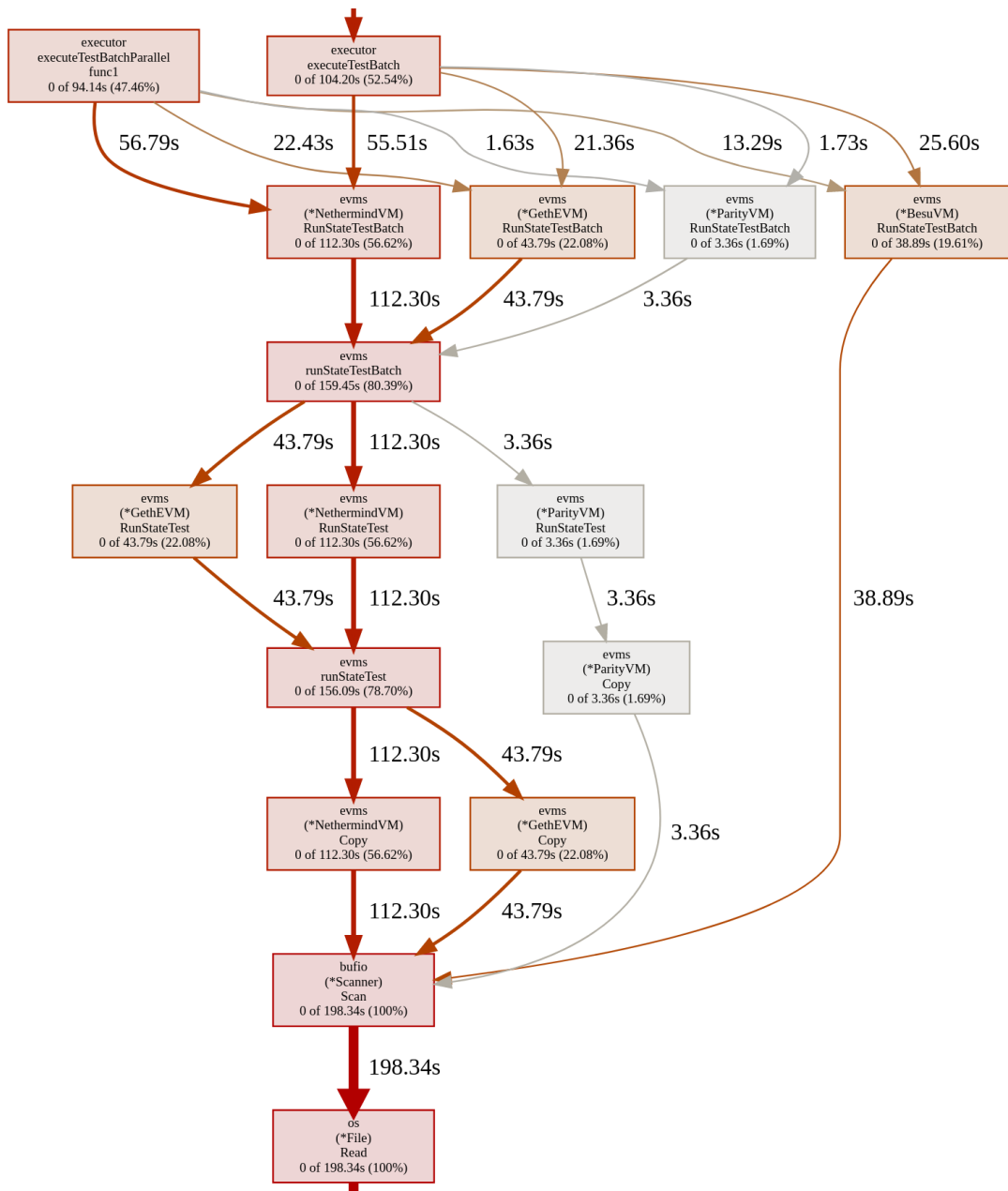


Figure 9.5: Performance traces


```
~/go/bin/godepq -from . -show-loc
Packages:
. (256)
github.com/MariusVanDerWijden/FuzzyVM/benchmark (328)
github.com/MariusVanDerWijden/FuzzyVM/fuzzer (182)
github.com/MariusVanDerWijden/FuzzyVM/generator (276)
github.com/MariusVanDerWijden/FuzzyVM/executor (433)
github.com/holiman/goevmlab/evms (935)
github.com/holiman/goevmlab/fuzzing (1523)
github.com/MariusVanDerWijden/FuzzyVM/generator/precompiles (621)
github.com/MariusVanDerWijden/FuzzyVM/filler (152)
github.com/ethereum/go-ethereum/tests (1761)
github.com/holiman/goevmlab/program (323)
github.com/ethereum/go-ethereum/consensus/ethash (3031)
github.com/holiman/goevmlab/ops (552)
github.com/ethereum/go-ethereum/core (8375)
github.com/ethereum/go-ethereum/core/vm (6790)
github.com/ethereum/go-ethereum/consensus (168)
github.com/ethereum/go-ethereum/core/rawdb (3655)
github.com/ethereum/go-ethereum/core/types (2249)
github.com/ethereum/go-ethereum/common/prque (389)
github.com/ethereum/go-ethereum/ethdb/leveldb (491)
github.com/ethereum/go-ethereum/ethdb/memorydb (315)
github.com/ethereum/go-ethereum/metrics (3807)
github.com/syndtr/goleveldb/leveldb (7288)
github.com/ethereum/go-ethereum/crypto (359)
github.com/ethereum/go-ethereum/crypto/secp256k1 (526)
github.com/ethereum/go-ethereum/common (831)
[...]

Total Lines Of Code: 117604 in 83 packages
```

Figure 9.6: Upper bound for total lines of code included by FuzzyVM

```

{ "TraceTest": {
  "env": {
    "currentCoinbase": "b94f5374fce5edbc8e2a8697c15331677e6ebf0b",
    "currentDifficulty": "0x20000",
    "currentGasLimit": "0x26e1f476fe1e22",
    "currentNumber": "0x1",
    "currentTimestamp": "0x3e8",
    "previousHash": "0x00000000000000000000000000000000..."
  },
  "pre": {
    "0x00000000000000000000000000000000ca1100b1a7e": {
      "code": "0x6040604001604002",
      "storage": {},
      "balance": "0x0",
      "nonce": "0x0" },
    "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b": {
      "code": "0x",
      "storage": {},
      "balance": "0xffffffff",
      "nonce": "0x0" }
  },
  "transaction": {
    "gasPrice": "0x1",
    "nonce": "0x0",
    "to": "0x00000000000000000000000000000000Ca1100b1A7E",
    "data": [ "0x80e3193e421154d1de1cd3a0b425cc21ea" ],
    "gasLimit": [ "0x7a1200" ],
    "value": [ "0x4862" ],
    "secretKey": "0x45a915e4d060149eb4365960e6a7a45f33439309306..."
  },
  "out": "0x",
  "post": {
    "Istanbul": [{
      "hash": "f9a857ac4ab3ebd098dfb06326d7a852f75273bb47116...",
      "logs": "1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7...",
      "indexes": {
        "data": 0,
        "gas": 0,
        "value": 0 }
    ]
  }
}
}
}
}

```

Figure 9.7: Example state tests

```

{"pc":0,"op":96,"gas":"0x79be38","gasCost":"0x3","memory":"0x","memSize":0,"stack":[],"returnStack":[],"returnData":"0x","depth":1,"refund":0,"opName":"PUSH1","error":""}
{"pc":2,"op":96,"gas":"0x79be35","gasCost":"0x3","memory":"0x","memSize":0,"stack":["0x40"],"returnStack":[],"returnData":"0x","depth":1,"refund":0,"opName":"PUSH1","error":""}
{"pc":4,"op":1,"gas":"0x79be32","gasCost":"0x3","memory":"0x","memSize":0,"stack":["0x40","0x40"],"returnStack":[],"returnData":"0x","depth":1,"refund":0,"opName":"ADD","error":""}
{"pc":5,"op":96,"gas":"0x79be2f","gasCost":"0x3","memory":"0x","memSize":0,"stack":["0x80"],"returnStack":[],"returnData":"0x","depth":1,"refund":0,"opName":"PUSH1","error":""}
{"pc":7,"op":2,"gas":"0x79be2c","gasCost":"0x5","memory":"0x","memSize":0,"stack":["0x80","0x40"],"returnStack":[],"returnData":"0x","depth":1,"refund":0,"opName":"MUL","error":""}
{"pc":8,"op":0,"gas":"0x79be27","gasCost":"0x0","memory":"0x","memSize":0,"stack":["0x2000"],"returnStack":[],"returnData":"0x","depth":1,"refund":0,"opName":"STOP","error":""}
{"output":"","gasUsed":"0x11","time":285282}
{"stateRoot": "
  f9a857ac4ab3ebd098dfb06326d7a852f75273bb47116275a81eb56331ffefef"}
[
  {
    "name": "TraceTest",
    "pass": true,
    "fork": "Istanbul"
  }
]

```

Figure 9.8: Example trace output



| Type | Explanation | Example |
|------------|--|-------------------------|
| Number | Plain json number | "pc":0 |
| Hex-Number | Hex-encoded number | "gas":"0x2540be400" |
| String | Plain string | "opName":"PUSH1" |
| Hex-String | Hex-encoded string | "returnData":"0xABAC" |
| Array of x | Array of x encoded values | "stack":["0x40","0x40"] |
| Key-Value | Key-Value structure encoded as hex strings | "0x40":"0x80" |
| Boolean | Json bool can either be true or false | "pass": true |

Figure 9.9: Datatypes used in EIP-3155, [40]

| Required fields: | | |
|-------------------|---------------------------|--|
| Name | Type | Explanation |
| <i>pc</i> | Number | Program Counter |
| <i>op</i> | Number | OpCode |
| <i>gas</i> | Hex-Number | Gas left before executing this operation |
| <i>gasCost</i> | Hex-Number | Gas cost of this operation |
| <i>stack</i> | Array of Hex-Numbers | Array of all values on the stack |
| <i>depth</i> | Number | Depth of the call stack |
| <i>returnData</i> | Hex-String | Data returned by function call |
| <i>refund</i> | Hex-Number of Hex-Numbers | Amount of global gas refunded |
| <i>memSize</i> | Number of Hex-Numbers | Size of memory array |

| Optional fields: | | |
|--------------------|----------------------|--|
| Name | Type | Explanation |
| <i>opName</i> | String | Name of the operation |
| <i>error</i> | Hex-String | Description of an error (should contain revert reason) |
| <i>memory</i> | Array of Hex-Strings | Array of all allocated values |
| <i>storage</i> | Key-Value | Array of all stored values |
| <i>returnStack</i> | Array of Hex-Numbers | Array of values, Stack of the called function |

Figure 9.10: Trace fields specified by EIP-3155, [40]

| Required Fields: | | |
|------------------|------------|--|
| Name | Type | Explanation |
| <i>stateRoot</i> | Hex-String | Root of the state trie after executing the transaction |
| <i>output</i> | Hex-String | Return values of the function |
| <i>gasUsed</i> | Hex-Number | All gas used by the transaction |
| <i>pass</i> | Boolean | Bool whether transaction was executed successfully |

| Optional Fields: | | |
|------------------|--------|---|
| Name | Type | Explanation |
| <i>time</i> | Number | Time in nanoseconds needed to execute the transaction |
| <i>fork</i> | String | Name of the fork rules used for execution |

Figure 9.11: Summary fields specified by EIP-3155, [40]

```

go test -v -coverpkg=./core/... -coverprofile=profile.cov ./tests/...
wc -l profile.cov
11261 profile.cov
grep -v -e " 1$" profile.cov | wc -l
9082
-> ~2179 lines in core package covered by state test

wc -l profilevm.cov
2243 profilevm.cov
grep -v -e " 1$" profilevm.cov | wc -l
1535
-> ~708 lines in vm package covered

```

Figure 9.12: Commands used to calculate the code coverage of the ethereum/tests

```
Benchmark BenchmarkTestGeneration took 80.265845ms
Benchmark BenchmarkExecution took 4.622382488s
Benchmark BenchmarkVerification took 15.19µs
Benchmark BenchmarkSingle took 3.25696237s
Benchmark BenchmarkSingleBatch took 3.227804458s
Benchmark BenchmarkSingleBatchDocke took 5.479475938s
Benchmark BenchmarkSingleBatchDocke took 5.476752064s
Benchmark BenchmarkLinear took 3.25902396s
Benchmark BenchmarkLinearBatch took 45.081µs
Benchmark BenchmarkLinearBatchDocke took 5.502472588s
Benchmark BenchmarkLinearBatchDocke took 34.9µs
Benchmark BenchmarkParallel took 1.154059446s
Benchmark BenchmarkParallelBatch took 28.67µs
Benchmark BenchmarkParallelBatchDocke took 1.364674968s
Benchmark BenchmarkParallelBatchDocke took 37.481µs
```

Figure 9.13: Benchmark results for 1 test

```
Benchmark BenchmarkTestGeneration took 232.940304ms
Benchmark BenchmarkExecution took 32.540975244s
Benchmark BenchmarkVerification took 101.21µs
Benchmark BenchmarkSingle took 32.569908208s
Benchmark BenchmarkSingleBatch took 18.618049302s
Benchmark BenchmarkSingleBatchDocke took 54.64752233s
Benchmark BenchmarkSingleBatchDocke took 32.178211823s
Benchmark BenchmarkLinear took 6.809133357s
Benchmark BenchmarkLinearBatch took 80.71µs
Benchmark BenchmarkLinearBatchDocke took 15.881415788s
Benchmark BenchmarkLinearBatchDocke took 83.941µs
Benchmark BenchmarkParallel took 11.547027188s
Benchmark BenchmarkParallelBatch took 76.441µs
Benchmark BenchmarkParallelBatchDocke took 14.161032697s
Benchmark BenchmarkParallelBatchDocke took 79.22µs
```

Figure 9.14: Benchmark results for 10 tests

```
Benchmark BenchmarkTestGeneration took 437.11682ms
Benchmark BenchmarkExecution took 5m25.083891367s
Benchmark BenchmarkVerification took 1.853987ms
Benchmark BenchmarkSingle took 5m25.21682416s
Benchmark BenchmarkSingleBatch took 2m51.225577141s
Benchmark BenchmarkSingleBatchDocke took 9m7.004574366s
Benchmark BenchmarkSingleBatchDocke took 4m56.517466875s
Benchmark BenchmarkLinear took 45.668400653s
Benchmark BenchmarkLinearBatch took 36.428571019s
Benchmark BenchmarkLinearBatchDocke took 2m3.068607566s
Benchmark BenchmarkLinearBatchDocke took 1m33.359965363s
Benchmark BenchmarkParallel took 1m55.461644019s
Benchmark BenchmarkParallelBatch took 1m52.840345177s
Benchmark BenchmarkParallelBatchDocke took 2m21.252965374s
Benchmark BenchmarkParallelBatchDocke took 2m30.846663799s
```

Figure 9.15: Benchmark results for 100 tests

```
Benchmark BenchmarkTestGeneration took 1.24701042s
Benchmark BenchmarkExecution took 27m20.229633735s
Benchmark BenchmarkVerification took 18.913412ms
Benchmark BenchmarkSingle took 27m17.553398835s
Benchmark BenchmarkSingleBatch took 14m13.545376699s
Benchmark BenchmarkSingleBatchDocke took 46m28.908600139s
Benchmark BenchmarkSingleBatchDocke took 24m57.728994569s
Benchmark BenchmarkLinear took 2m10.335371886s
Benchmark BenchmarkLinearBatch took 1m12.536428292s
Benchmark BenchmarkLinearBatchDocke took 8m47.053512241s
Benchmark BenchmarkLinearBatchDocke took 6m1.474793305s
Benchmark BenchmarkParallel took 9m44.429427379s
Benchmark BenchmarkParallelBatch took 9m3.388994374s
Benchmark BenchmarkParallelBatchDocke took 12m12.39878288s
Benchmark BenchmarkParallelBatchDocke took 12m14.362542239s
```

Figure 9.16: Benchmark results for 500 tests

```
Benchmark BenchmarkTestGeneration took 2.526458463s
Benchmark BenchmarkExecution took 54m6.551028309s
Benchmark BenchmarkVerification took 20.336717ms
Benchmark BenchmarkSingle took 54m11.556962542s
Benchmark BenchmarkSingleBatch took 28m17.053024675s
Benchmark BenchmarkSingleBatchDocked took 1h31m20.396178405s
Benchmark BenchmarkSingleBatchBatchDocked took 49m6.075392857s
Benchmark BenchmarkLinear took 7m17.515634165s
Benchmark BenchmarkLinearBatch took 4m14.277742443s
Benchmark BenchmarkLinearBatchDocked took 19m46.981875368s
Benchmark BenchmarkLinearBatchBatchDocked took 13m55.023040088s
Benchmark BenchmarkParallel took 19m25.38774544s
Benchmark BenchmarkParallelBatch took 18m51.094307848s
Benchmark BenchmarkParallelBatchDocked took 24m7.44921456s
Benchmark BenchmarkParallelBatchBatchDocked took 25m18.575576436s
```

Figure 9.17: Benchmark results for 1.000 tests