# CS438: DissFuzz

Nikita Evsiukov
nikita.evsiukov@epfl.ch

Chau Ying Kot
chau.kot@epfl.ch

Qifan Pan
qifan.pan@epfl.ch

Milana Maric
milana.maric@epfl.ch

Derya Cögendez
derya.cogendez@epfl.ch

Vladimir Hanin
vladimir.hanin@epfl.ch

## I. Executive Summary

Fuzzing is a technique employed to identify vulnerabilities in a program, known as the program under test (PUT), by automatically generating and testing diverse inputs. Running a fuzzer locally on a single machine can be time-consuming, particularly with complex PUTs. Fuzzing is a parallelizable task that can be distributed across different machine cores. Further optimization can be achieved by distributing the workload among multiple machines.

Our project, DissFuzz, is the first work to support distributed and decentralised fuzzing based on untrusted nodes. The system incorporates different actors that use the blockchain for synchronisation, ensuring the security of our system. The blockchain contains diverse data essential for executing the fuzzing request with PUTs and also validating the legitimacy of bug reports. To encourage user participation in the system, we introduce a digital currency. Users contributing their resources to our system are rewarded. To maintain the integrity and security of our system, we implement a Proof of Work (PoW) mechanism. This mechanism is crucial in securing the system by requiring participants to demonstrate computational effort for adding blocks to the blockchain.

The system is composed of three key actors. The first actor, *Proposer* requests a fuzzing process for a given PUT with different customizable parameters. It secretly selects a random seed given to the fuzzer from a given range and calculates the *Execution Hash* [1] by running the PUT with that seed. This becomes a challenge to ensure the validity of the response from the other actors in the system. This fuzzing puzzle is called a *Fuzzle*. The fuzzing request is then handled by another actor in the system called *Executor*, whose role is to fuzz the PUT, searching for the secret seed that generated the given execution hash, and collecting any detected bugs in the process. Once the secret seed is found, the executor creates a *Solution* with the proof of the completion of the challenge. The fuzzles and solutions are sent to the last actor in the system, called *Validators*. They validate the fuzzles and the solutions, and incorporate them into blocks added with proof of work to the blockchain. Executors are rewarded by the proposer when they create a solution to a fuzzle, but also for each bug that it has found. *Validators* are also rewarded for taking part in the proof of work mechanism as they get a reward whenever they add a block on the blockchain.

The project is built on top of Peerster, and our contribution can be broadly categorized into three main areas; Cryptographic & Fuzzing Infrastructures, Blockchain and Validation. The cryptographic framework uses keys generated by the prime256v1 elliptic curve for the Elliptic Curve Digital Signature Algorithm (ECDSA). For the fuzzing, the engine is a modified version of LLVM based on the fuzzcoin project [2]. We have introduced a new blockchain storage design to handle forks in the blockchain. To manage fuzzles or fuzzle solutions awaiting addition to the blockchain, we have implemented a mempool. This structure facilitates the mining of blocks that have been moved to a fork. The mining difficulty is adjusted to maintain a consistent block mining time. The management of currency is handled by a balance manager, who is responsible for retrieving the current balance for a specific user based on the main chain. The fuzzle tracker is used by the executor to fetch the next unsolved fuzzle, and also aids the validator in confirming the uniqueness of a fuzzle or fuzzle solution and ensuring that a solution references an existing fuzzle. The executor and the validator need to verify the validity of the received data, and this task is handled by the validation module.

The system's implementation is highly compartmentalised into separate modules, which build on top of each other. Some of those modules are used by all roles, while others are reclusive to specific roles. When a user wishes to contribute to the system, they have the flexibility to choose which roles to undertake. A node can simultaneously assume multiple roles within the system.

The system was tested using unit tests for each module, with integration tests between a couple of modules and end-to-end tests. Benchmarks were conducted to measure the overhead of the proof of work and other implementation mechanisms, considering the system's main goal is to accelerate program fuzzing.

## II. Background and Related Work

Fuzzing is a software testing method to automatically generate and inject inputs into a software system to identify vulnerabilities. It can be easily parallelized, and there have been successful attempts to distribute the fuzzing workload across multiple machines. A notable example is Google's ClusterFuzz project [3], which offers a scalable fuzzing infrastructure for all Google products and some open-source projects.

While ClusterFuzz is built on trusted cluster nodes, Fuzzing@Home [4] extends distributed fuzzing to untrusted heterogeneous clients. However, it still relies on a trusted central control server. This server manages a fuzzing pool, distributing work and verifying fuzzing results with Proof-of-Fuzzing-Work (PoFW). Although this minimizes duplicate work, it does have the flaw that the system is centralised.

The objective of our project DissFuzz is to extend the distributed fuzzing further by eliminating the single point of trust in Fuzzing@Home. This is achieved through the implementation of permissionless consensus and zero-knowledge Proof-of-Fuzzing-Work (ZK-PoFW). DissFuzz enables nodes to submit Programs Under Tests (PUTs) to the network and receive fuzzing results. Furthermore, DissFuzz incentivises users to participate by rewarding them with cryptocurrency linked to the blockchain. This project would be the first to propose distributed and decentralised fuzzing.

## III. Design

Figure 1 illustrates the high-level architecture of the system, which involves three roles: *Proposer*, *Executor*, and *Validator*, which interact with each other through a blockchain. Each machine can take one or multiple roles from the three. A more detailed schema can be found in the Appendix 12.

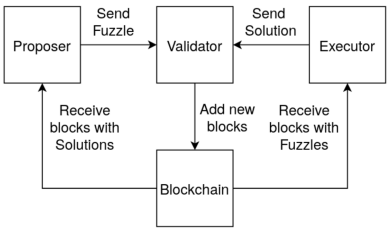


Fig. 1: High-level interaction of the different roles in the system

**The Proposer** initiates the fuzzing in the system by submitting the source code of the PUT (along with its corresponding fuzzer program), which gets uploaded to the peer's distributed file system. The proposer must pay some money, which will reward the other nodes participating in the fuzzing of its PUT. The amount of money is proportional to the amount of resources needed to fuzz the PUT. For instance, a large PUT that requires more time to fuzz one seed must be compensated with a highier reward. To ensure that other nodes have genuinely tested the seeds within the specific range, the proposer challenges other nodes to find a specific seed in the given range. This challenge is called a "fuzzle". The proposer eventually receives feedback indicating whether bugs were found in its PUT. If no other machine decides to solve its fuzzle, the proposer can still unlock the money it had paid to publish its PUT.

**The Executor** carries out the fuzzing of the proposed programs from a *Proposer* within the specified seed range. A reward is given to the *Executor* only if it can find the solution to the challenge presented by the *Proposer*. Additionally, the *Executor* can also include possible bugs discovered during the fuzzing process in the solution.

**The Validator** invests computational resources to maintain the system's state. The *Validator* validates the fuzzles and the solutions, and then adds them to the blockchain. In return, he receives rewards for it in the form of the system's currency.

### A. Zero-Knowledge Proof-of-Fuzzing-Work (ZK-PoFW)

Proof-of-Fuzzing-Work (PoFW) is first proposed in Fuzzing@Home [4] to avoid malicious nodes claiming the reward without doing the fuzzing work. In order to adapt PoFW to our decentralised setting, DissFuzz introduces zero knowledge Proof-of-Fuzzing-Work (ZK-PoFW) using public key cryptography. ZK-PoFW is used by the executor to prove that it has solved the fuzzing challenge set by the proposer. The *Proposer* wants to run the fuzzer program on a seed range. It chooses a random seed from the seed range and runs the fuzzer with the chosen seed to generate an execution hash, which is used as the private key, from which the public key is generated. Then the proposer includes the public key and the seed range in the fuzzle and publishes it. The *Executor* who is solving the fuzzle runs the fuzzer program which generates an execution hash for each seed in the seed range and tests whether each execution hash as a private key matches the public key included in the fuzzle. If the *Executor* finds the private key, it generates the *Proof* by signing *Hash of Fuzzle* and *Hash of Solution* using the discovered private key to prove that it has found the private key. Anyone can verify the *Proof* with the public key.

### B. Fuzzing Workload Distribution

To divide the fuzzing workload for a given PUT, the system uses the following methodology. The PUT can take any input values of any size, such as an integer or a whole file. The fuzzer chosen is the libFuzzer, capable of generating unique random values based on a seed. The PUT must then use a corresponding fuzzing program to adapt the random values from the fuzzer into its input values.

To distribute the work across multiple machines, the system divides the total seed range of the fuzzer into intervals. These intervals can be executed in parallel by different machines within the system, as illustrated in Fig. 2. In essence, each fuzzer seed interval will be executed by one or multiple machines, which will give each seed value within the given range to the fuzzer. This will create unique input values for the fuzzing program., which will convert the values to

interpretable input values, which will be tested against the PUT to see whether it generates a bug.

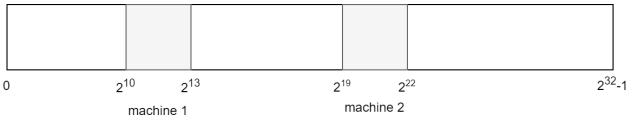


Fig. 2: Visualisation of the distribution of the fuzzing work between multiple machines.

A blockchain is used to confirm results published by machines, preventing malicious entities from falsely claiming they contributed to the fuzzing process or reporting inaccurate bug findings. Proof of work is used as a consensus algorithm. Each block in the blockchain contains either a PUT with a given seed interval or a solution to such a seed interval with potential bugs.

### C. Fuzzle Creation

As stated above, a fuzzle is a challenge that the proposer sets to executors. The fuzzle contains an interval of seeds. Within that interval, one seed will be chosen at random. The proposer then executes the fuzzer with that seed, and extracts an execution hash based on the code coverage of the PUT. Using cryptography, the execution hash is then used as a private key to generate a public key. This public key is then included in the fuzzle structure, which is sent to the validators, which will include this fuzzle in the blockchain. When the executors receive this new block, they extract the fuzzle and try to find the seed in the given range that will generate an execution hash that will generate a public key that matches the one in the fuzzle. The executor will walk randomly across the interval searching for the correct seed.

One important thing to note is that as soon as the executor finds the solution, it will stop testing other seeds and will report the solution to get the reward from the proposer. Hence, not all the seeds from a given range will be tested, which might cause some seeds with bugs never to be checked. The proposer can remedy this by specifying a target probability $p$ that a seed in the total seed range is tested during the whole fuzzing process. The proposer can then adjust two parameters, namely the size of the interval of the fuzzles it creates, and the number of intervals that will overlap each other, according to the equations below[1].

First, the probability $p_0$ that a seed was tested in a given fuzzle is given by :

$$p_0 = \frac{1}{x}\mathbb{E}[K] = \frac{1}{x}\sum_{k=0}^{x}\left(\frac{x-k}{x}\right)^n \tag{1}$$

where $x$ stands for the size of the interval, $n$ for the number of nodes working on this fuzzle, and $K$ stands for the number of seeds that the executor tests before finding the solution. From this probability, the size of the interval $x$ to get the requested probability $p$ is given by :

$$x = \frac{t_{cap}}{t_0 \cdot f(\frac{TN}{t_{cap}})}. \tag{2}$$

where $t_{cap}$ is the maximum time that can be spent solving a single fuzzle, $t_0$ is the time to run the PUT under one input value, $T$ is the time between blocks in the blockchain, and $N$ is the number of nodes in the system, and $f$ is a function that approximates 1 for valid $n$. Then, the overlapping factor $l$ can be calculated using the following:

$$l = \log_{1-f(\frac{TN}{t_{cap}})}(1-p). \tag{3}$$

This method is then used so that the proposer can set the required parameters to obtain its designed $p$, which it can increase until it has a desired confidence that a seed was tested.

### D. Proposer

The proposer's role is to request the execution of a fuzzer program, which takes as input a fuzzing seed number. The node randomly selects a fuzzing seed number within the target input range and executes the program with this value to generate an *Execution Hash*. This hash corresponds to the hash of the code coverage map generated during program execution, as described in Fuzzing@Home [4]. Then, the node creates a fuzzing puzzle, called "fuzzle", by generating a public key using the calculated Execution Hash as a private key. The challenge for the *Executors* is to solve the fuzzle by finding this private key. Since *Executor* does not need to execute the program on the entire seed range to find the *Fuzzle Solution*, *Proposer* can add multiple *Fuzzles* with overlapping input ranges to increase the probability of a seed being executed.

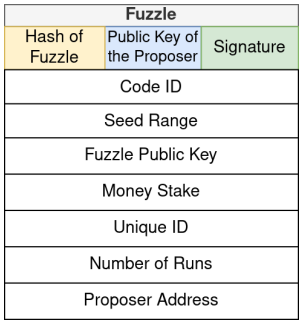


Fig. 3: Content of a fuzzle

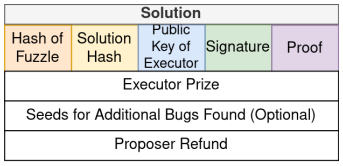


Fig. 4: Content of a solution

To propose a program, the *Proposer* must stake a certain amount of money. A portion of this stake serves as a reward for solving the given fuzzle, while the remaining will be returned to the proposer. Consequently, if a *Proposer* submits a fuzzle without a solution, he loses the right to retrieve the stacked money. However, the proposer can reclaim the stake by providing a valid solution itself in case no executors want to solve its fuzzle.

Fig. 3 outlines the content of a fuzzle. The *Public Key* of the proposer is its identity proved by the signature of the fuzzle.

[1]The details of the derivation can be found in our repository at the following link: https://github.com/cs438-epfl/project-dissfuzz/blob/main/fuzzle_probability.pdf

The *Code ID* is a metahash of the file in the Peerster data storage. The *Seed Range* denotes a range of values where one of them is used to generate the execution hash. *Number of Runs* is the number of times fuzzer should mutate each seed, this value also affects the generation of the execution hash. *Fuzzle Public Key* is a public key generated from the execution hash. The *Unique ID* is a randomly generated unique value used to distinguish fuzzles with otherwise identical fields.

### E. Executor

The executor plays the role of running the proposed fuzzer programs. The executor randomly chooses an unsolved fuzzle and tries to find the execution hash matching the *Fuzzle Public Key*. The *Code ID* allows the executor to retrieve the corresponding PUT from the Peerster's file storage. The executor runs it with input values from the *Seed Range*. When the node finds a matching execution hash, he generates a *Fuzzle Solution*, including any bugs detected during execution. The structure of the solution is detailed in Fig. 4.

The *Hash of Fuzzle* is used to bind the *Fuzzle Solution* to the corresponding *Fuzzle*. The *Proof* is a zero-knowledge proof attesting that the node has correctly solved the fuzzle. The proof is obtained by signing *Hash of Fuzzle* and *Hash Current* using the discovered execution hash as the private key. The verification of the proof can be done by anyone using the *Fuzzle Public Key*.

The *Public Key of the Executor* is the identity of the executor, which can be proven by the *Signature* of the fuzzle solution. The executor includes in the *Fuzzle Solution* possible identified bugs and rewards from the proposer for the solution. As the solution to a fuzzle confirms the validity of the fuzzle, it also releases the remaining reward from the *Stake* for the proposer to reclaim. The executor can also add any *Bug Seeds* that are found during the fuzzing process, this field is optional.

### F. Blockchain

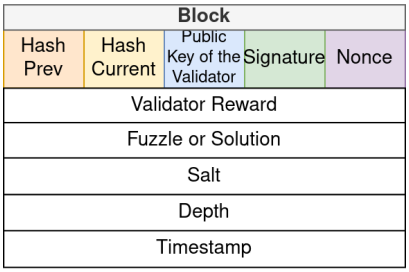


Fig. 5: Content of a block

A blockchain is a chain of blocks with a payload in the form of a *Fuzzle* or a *Solution*. The Blockchain is maintained using the Proof-of-Work (PoW) schema. Blocks from the blockchain are distributed using the gossiping mechanism of Peerster.

In our system, only validators can add blocks to the chain using the PoW. Proposers and Executors listen to blockchain to obtain relevant data for their respective tasks. The proposer and the executor broadcast the fuzzle or the solution to notify the validator to incorporate it into the blockchain. An illustration can be found in Fig. 6.

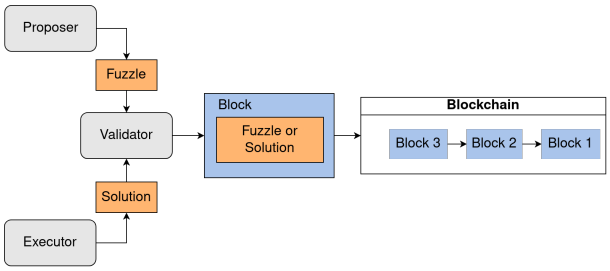


Fig. 6: Process to add a block in the blockchain

### G. Validator

The validator adds new blocks to the blockchain by mining nonces from the PoW schema and receive a reward from the system. The validator is responsible for validating fuzzles from proposers and fuzzles solutions from executors before including them in the main chain by mining the nonce. When the validator finds the correct nonce, it creates a block, as depicted in Fig. 5, and distributes it using the Peerster's gossiping mechanism.

The mining difficulty is adjusted to keep the mining time constant across the validators.

The *Public Key* of the validator is its identity proved by the *Signature* of the block. *Validator Reward* is the amount of reward from the system for maintaining the blockchain, this encourages nodes to become validators.

### H. Validation

To verify the *Fuzzle*, the validator must do the following steps: 1) validate the *Hash of Fuzzle*; 2) check that the fuzzle is not already proposed; 3) validate the *Signature*; 4) validate the *Code ID* and the corresponding program is the storage; 5) validate that the *Seed Range* contains at least one value; 6) validate that the *Money Stake* is not less than the stake threshold for proposing. 7) validate that the proposer has enough money.

To verify the *Fuzzle Solution*, the validator must do the following steps: 1) validate the *Hash Current*; 2) check the solution is not already proposed; 3) validate the *Signature*; 4) Validate the *Proof* using *Fuzzle Public Key* from the corresponding *Fuzzle* found using the *Hash of Fuzzle*; 5) validate reported bugs by running the program in the *Fuzzle* with the provided seeds; 6) validate the *Executor Prize* 7) validate the *Proposer Refund*

All proposers and executors must also verify both *Fuzzle Solutions* and *Fuzzles* using the schemas above. Additionally, everyone must validate the *Hash Current*, *Signature* and *Nonce* in the block from the blockchain.

### I. Threat Model and Assumptions

In our threat model, nodes can be uncooperative and malicious. For example, nodes can stash found bugs, drop/spam messages, upload malicious PUTs, etc.

In our system, we have the following assumptions:

1) The Program Under Test (PUT) is deterministic under its input and can generate execution hashes using our fuzzer.
2) All the executors use the same fuzzer, which requires a PUT and an integer as the seed number to generate the input to the PUT and an integer for the number of times the fuzzer will run the program.
3) Executors have the capability to sandbox malicious PUTs.
4) There are no normal direct transactions between peers in the network.
5) More than 50% of the network's hashrate (computational power) is controlled by honest nodes.
6) There are no blockchain network attacks, e.g., Distributed Denial of Service, or eclipse attacks.
7) There is a sufficient number of users to ensure that fuzzles and their solutions are proposed regularly.
8) The storage and the file sharing (from HW2) always work.

## IV. Implementation

Our project is built on top of *Peerster*. It uses *Peerster*'s gossiping protocol and data-sharing module with slight adaptations. We introduced local time to status messages to support Network-adjusted time and compute the median of the timestamps returned by all neighbouring nodes. Moreover, we adapted the search mechanism so that if it does not find the file with the corresponding metahash via expanding ring search it requests it from the node that uploaded the file directly.

### A. Cryptographic & Fuzzing Infrastructures

*1) Fuzzing Infrastructure:* The fuzzing infrastructure is a module that executes the fuzzer given a specific seed and outputs the execution hash of the seed, together with whether a bug was found for that seed.

*2) Cryptographic Infrastructure:* In the system, it is required that some roles generate a public key from a given 256-bit private key. This module uses the 'prime256v1' elliptic curve to perform ECDSA cryptography, from the OpenSSL library.

*3) Golang Wrappers:* Golang wrappers is a small module that streamlines the integration of the previous sections into the rest of the code base. As the Fuzzing Infrastructure and Cryptographic Infrastructure were made from bash scripts and C code, to enhance usability with the Peerster infrastructure, this module creates an interface to perform various operations from golang.

### B. Blockchain Infrastructures

*1) Fuzzer Blockchain Storage:* The fuzzer blockchain storage stores locally all the blocks of the blockchain. It builds upon the blockchain storage of Peerster. In the current version of the system, the blockchain storage is implemented in memory, though an in-file implementation has been designed too. A major addition to the storage is its ability to report when it detects a change in the main chain. This notifies the validator or the executor that they must potentially interrupt their current task and reevaluate the status of the main chain before starting their job again. Another feature of the storage is the capability to indicate whether a block is in the main chain or a fork. To archive this, an extra flag is maintained in a structure called *Fuzzer Storage Block*. This structure ensures efficient retrieval of this information. This flag is used by different components within our system to maintain consistency across the different modules.

Each block also stores the depth, a number indicating how far it is from the bootstrap block. This design choice aids in tracking the main chain and also detecting any reorganisation of the main chain. In scenarios where a fork has the same depth as the main chain, the decision was made to retrain the one that was created first.

*2) Mempool:* The mempool is a data structure that keeps track of the pending fuzzles and solutions, received from the proposers and executors respectively, that have not yet been added to the blockchain by the validators. When a validator finishes mining its previous block, it asks the mempool for the content to add to the next block it will mine. The mempool will always prioritise solutions over fuzzles. This means that even if there are a lot of proposers in the system, a proposer should ways receive its solution in the blockchain once an executor finds it. Also, the fuzzles are stored in the queue, so newer fuzzles or solutions should not impede on older ones. The mempool also allows it to be updated when the storage reports that the main chain has changed to a fork. It will update its content accordingly in case a pending fuzzle or solution is now included in the main chain.

*3) Network Time Service:* The network time service allows nodes to reach a consensus on the timestamp of a newly added block, which is required by the difficulty module. The timestamp has two properties:

- Lower bound: median of last 11 blocks, a monotonically growing number.
- Upper bound: network-adjusted time + 2 hours.

Network-adjusted time is the median of the timestamps returned by all nodes connected to the node. A timestamp is accepted as valid if it is greater than the median timestamp of the previous 11 blocks and less than the network-adjusted time + 2 hours.

*4) Difficulty module:* The difficulty module uses a complexity regulation algorithm adapted from Bitcoin [5], [6]. This allows us to keep the speed of the new block generation constant so that with the increased mining power, the system will still be stable.

The mining difficulty target in Bitcoin is the number of leading zero bits in the value `Hash(HashCurr||HashPrev||Nonce)` for the block to be accepted as valid.

The difficulty module uses the network time service so that nodes can agree on the time to mine the last $N$ blocks, which is used to update the mining difficulty target. This number changes every $N$ block. It is adjusted based on how efficient the miners were for the previous $N$ blocks according to this formula:

$$\textit{next difficulty} = \frac{\textit{previous difficulty} * N * 10\ min}{\textit{time to mine last N blocks}}$$

The effect of this formula is to scale the difficulty to maintain the property that blocks should be found by the network on average about once every ten minutes. This schema allows each node to adjust the complexity simultaneously. Nodes agree on the new difficulty because they agree on the time to mine the last $N$ blocks thanks to the network time service.

*5) PoW Miner:* The PoW miner changes the nonce in the block so that the number of leading zero bits in the value `Hash(HashCurr||HashPrev||Nonce)` matches the mining difficulty target. To avoid the case where the condition is not satisfied after the miner tries all possible nonces, we added a field Salt in the block that the miner can change to influence the `HashCurr`.

### C. Blockchain Services

Blockchain services run on top of the blockchain storage and provide functionalities to blockchain users to efficiently access the information stored in the blockchain, e.g., balance, unsolved fuzzles, etc.

*1) Balance Manager:* The balance manager keeps track of the balance for every peer at each block and supports constant time balance queries. This helps validators to check if a proposer has enough money to stake.

The balance manager works as an observer of the blockchain storage and updates its internal states whenever a new block is added to the chain. In order to scale to millions of blocks, the balance manager stores balance differences for each block (2-3 entries) and sets at an interval $T$, a configurable parameter, checkpoints that store a snapshot of user balances. On a balance query, the balance manager accumulates the balance differences by iterating from the queried block back to the last checkpoint.

Because most updates and queries happen at the tip of the main chain, we can segment the balance records by checkpoints into chunks and store them on disk, keep the most recent chunks in memory, and have a slow path for a chunk miss by fetching chunks from the disk. The current implementation is in memory, but it is open to extension to the chunk storage.

*2) Fuzzle Tracker:* The fuzzle tracker is a blockchain service that keeps track of unsolved fuzzles. It is used by executors to choose the next fuzzle to work on and validators to check if a solution is correct or if a fuzzle or a solution is duplicated (two solutions are seen as duplicated if they solve the same fuzzle). Similar to the balance manager, the fuzzle tracker is an observer over the blockchain storage. Different from the balance manager, the fuzzle tracker works only at the tip of the main chain so it needs to handle blockchain reorgs.

The fuzzle tracker keeps a map from fuzzle hashes to fuzzles and a set of unsolved fuzzles for sampling. The fuzzle map grows with the blockchain but can comfortably fit in the memory even at the scale of millions of blocks. The number of unsolved fuzzles is bounded because fuzzles can be solved by proposers themselves after a timeout.

### D. Validation Module

The validation module is a set of functions that helps the nodes verify the validity of different components of the system. When a validator receives a fuzzle or solution, it will use this module to verify the validity of the fuzzle before mining a nonce and putting a new block on the blockchain. Similarly, when any node receives a new block, it will validate each field as well as its content (fuzzle/solution included) of the block.

This module also uses the cryptographic module, which can be used for calculating hashes, generating key pairs, signing fuzzles, solutions and blocks etc. It also includes several helper functions for different formats of key pairs (struct, string, file)

### E. Proposer

The proposer is a node that possesses a program it intends to fuzz. To create *Fuzzles* user needs to provide: Code path, input range, overlapping parameter, max number of *Fuzzles*, timeout for *Fuzzle* retrying, timeout for money-unlocking, callback to be triggered upon receiving the *Fuzzle Solution*. The user must be aware of the difficulty of mining a nonce and fuzzing its solution in order to estimate the timeout for fuzzle retrying and timeout for money-unlocking.

When the user proposes code to fuzz, the proposer first uploads code to the *Peerster* storage by calling `Upload` and `Tag` functions of the data-sharing module. *Proposer* chooses a random seed and samples the execution time and treats it as an average time to execute the program. This average time is then used to slice the input range and calculate *Money Stake* that should be included in each *Fuzzle* so that *Executor* does not timeout when executing the program and discards the *Fuzzle*. *Proposer* slices the whole range on multiple *Fuzzles*, up to the maximum number of solved *Fuzzles*. The slicing is done with overlapping according to the overlapping parameter. For example, if the overlapping parameter is 0.5, then the $i$ *Fuzzle*'s input range starts from half of the $(i-1)$ *Fuzzle*'s input range. For each slice of input range, *Proposer* chooses a random seed, executes the program with this seed as input value, gathers the *ExecutionHash*, uses *ExecutionHash* to generate *Fuzzle Public Key*, creates a *Fuzzle* and its *Fuzzle solution* and adds *Fuzzle* to the proposing pool.

*Fuzzles* in the proposing pool are broadcasted to all *Validators* so that they can be included them in the blockchain. *Proposer* is subscribed to the new blocks being added to the blockchain. *Fuzzle* is removed from the proposing pool if it has been included in the blockchain. If *Fuzzle* was not included in the blockchain and the timeout for fuzzle retrying expires fuzzles broadcasted to *Validators* once again.

After *Fuzzle* has been included, the timer starts again and waits for *Fuzzle Solution*. If the solution is not received before the timeout *Proposer* adds the solution it computed earlier to the proposing pool in order to unlock *Money Stake*. Once the *Fuzzle Solution* is added to the blockchain, either if it was found by *Executor* or *Proposer* broadcasted it, a callback provided by the user is triggered.

### F. Executor

The *Executors* job is to monitor the blockchain for incoming *Fuzzle* blocks and to try to find the solution for an unsolved *Fuzzles*. In order to get a *Fuzzle* to fuzz solution, the *Executor* uses *Fuzzle Tracker*. When the *Executor* gets random unsolved *Fuzzle* from *Fuzzle Tracker* it first checks if it already processed this *Fuzzle* and if it has it does not try to solve it again it just adds the *Fuzzle Solution* to the proposer pool similar to one *Proposers* are using. If it never fuzzed the chosen *Fuzzle* before the *Executor* tries to find *Fuzzle Solution*.

When trying to solve a *Fuzzle*, the *Executor* first downloads the source code of the PUT using the *Peerster*'s data storage. The download procedure is slightly modified from the Peerster system, as it first uses an expanding ring to look for the file and if that search fails, it contacts the *Proposer* directly. Once the file with the source code is downloaded, each seed from the interval is executed against the execution engine. In order to guard itself from malicious *Proposers*, who might suggest code that doesn't terminate, *Executors* calculate the maximum time to run one seed based on the *Money Stake*. If the timeout for executing one seed is reached, the *Executor* marks the *Fuzzle* as invalid locally and never tries to find *Fuzzle Solution* for it again. If the seed terminates, the *Executor* uses *Execution Hash* to compute the private key and match it with *Fuzzle Private Key* to find the solution. Moreover, it checks if some error has occurred while executing the seed and adds the seeds to *Bug Seeds* if it has. If the *Executor* runs all seeds and does not manage to find the *Fuzzle Private Key* it marks the *Fuzzle* as invalid. If it found proof of work, it calculates the *ExecutorPrice* and *ProposerRefund* based on the number of bugs found and the *Money Stake*. Even if the *Executor* does not find any bugs, it still gets a price for solving the *Fuzzle*. When *ExecutorPrice* and *ProposerRefund* is calculated, *Executor* creates *Fuzzle Solution* with the *Execution Hash* corresponding proof of work, signs it so others cannot steal it and adds it to the proposer pool.

Proposer pool is similar to one *Proposer* uses. For each *Fuzzle Solution* it the pool, upon timeout, it is checked if it was included in the blockchain and if it was not the *Fuzzle Solution* is broadcasted to the *Validators*. Besides monitoring the blockchain to check if it should stop broadcasting *Fuzzle Solutions*, the *Executor* checks if the newly added block has the solution to a *Fuzzle* that is currently being fuzzed and stops the execution if it does.

### G. Validator

The primary job of the validator is to mine the correct nonce, by delegating the mining itself to the PoFW module. The selection of the target to mine is also simplified by the utilization of the mempool. In a background process, the validator queries the mempool for the next target to mine. Upon receiving a target, it proceeds to mine using the POFW module. Once the mining is complete, the target is removed from the mempool, and the new block is propagated through the system using the existing gossiping protocol.

To be added to the mempool, both a fuzzle or a fuzzle solution must be verified first. When a validator receives a fuzzle or a fuzzle solution, it checks the validity of the content using the validation module. All content present in the mempool is validated.

When the validator receives fuzzle, in addition to performing the validation, it also downloads the PUT referenced by the fuzzle. This approach is motivated by the initial condition where only the proposer possesses the PUT. Many executors attempting to download from the proposer may lead to congestion. Assuming a well-balanced network, the nodes possessing the program should be distributed across the system and mitigate potential congestion issues.

When the validator receives a block, the mining needs to be interrupted, as the state of the blockchain has been modified. It is also possible that the main chain changes with the newly arrived block, if this is the case, we also need to notify the mempool to reorganize his content. An overview of module interactions is depicted in Fig. 7.

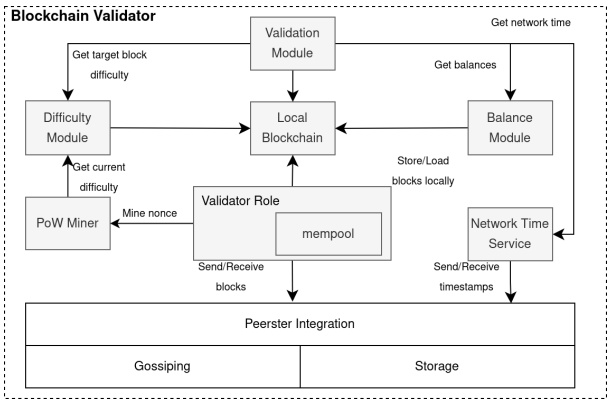


Fig. 7: Schema of the blockchain validator

### H. User Interface

In order for a user to interact with its node, the graphical user interface of Peerster was extended for this project. The graphical user interface now allows the configuration and the start of a fuzzing task, while also getting statistics from the node. For the backend, we extended the HTTP node to handle new POST requests for fuzzle proposing from the client and push statistics updates to the client. We collect statistics for the node, such as the balance, the number of bugs found for the executor, the number of blocks mined for the validator, etc. In addition to the raw statistics, the proposer can also visualise the progress of the current fuzzing task with a 1D heatmap, where the x-axis represents the seed number. We used colors to represent the status of a fuzzle, between it being created and it being solved by an executor. The bugs found will also be displayed on the heatmap, to indicate to the user where the bugs are located in the seed range. We use a color gradient to encode the probability of a seed number being covered if it is included in a solved fuzzle. The greener the heatmap, the more probable it is that the seed doesn't contain a bug.

## V. Evaluation

Our system was tested using first unit tests, then integration tests, end-to-end tests, and finally benchmarks were recorded. The code coverage of the system is 80.2%.

Our system has some tests that are non-deterministically passing. We use timeouts to emulate other components and we have mining in our system, which makes our tests probabilistic and hard to estimate the correct timeout. These tests include but might be exhaustive to Executor `Fuzz_Solution_Received_While_Solving_Block`, `Test_Fuzzing_E2E_One_Proposer_Multiple_Validators_Unlocking_Correct_Fuzzle` and `Test_Blockchain_Integration_Keep_Up_With_Difficulty` 1 and 2.

### A. Unit Tests

Unit tests were written for each separate module of our system, which were the following: the blockchain storage, the balance manager, the network time synchronization module, the PoW miner, the validation modules, the mempool, the fuzzle tracker, the difficulty module, the cryptographic module, the difficulty module, and the Peerster util classes such as a concurrent map. For the external util scripts (execution engine, ECDSA keygen, Docker) manually testing was done extensively.

### B. Integration Tests

There are a lot of modules in the system that depend on multiple smaller ones. To test their dependency, we implemented the following integration tests: the timestamp validator module with network time synchronization module and local blockchain storage, the PoW mining difficulty module with the local blockchain storage, the PoW miner and the difficulty module, the balance manager and local blockchain storage, the block validator module with the local blockchain storage and PoW difficulty module, the blockchain storage with the balance module and the execution engine, the components of the Validator role with the Peerster gossiping and storage modules, and finally the different components of the Proposer and Executor roles.

### C. End-to-End Tests

End-to-end tests were also performed to make sure the whole behavior from start to finish would work. The following scenarios were tested: a bunch of validators receiving correct and incorrect fuzzles (even if the main chain changes), a bunch of correct validators and a minority of malicious validators that try to publish invalid blocks, a valid proposer creating a valid fuzzle and it being added to the blockchain by the valid validators, a bunch of correct validators and a malicious proposer that creates an invalid fuzzle, a bunch of correct validator and a correct proposer publishing a valid fuzzle and it getting solved by a valid executor, a bunch of correct validator and a correct proposer publishing a valid fuzzle but with a malicious executor that sends an incorrect solution, a bunch of correct validator and an incorrect proposer which creates a fuzzle with an invalid PUT which doesn't get solved by an honest executor and doesn't get blocked by that malicious PUT, and lastly a scenario with a bunch of correct validators and executors and proposers which all work correctly to proposer fuzzles and solve them.

### D. Benchmarks

We ran some benchmarks to check the scalability and overhead of our system. To parallelize the execution of these benchmarks, we utilised different machines for each benchmark. However, it's important to note that within each benchmark section, all operations were executed on a single machine.

*1) System Overhead:* In this benchmark, we want to see the overhead our system introduces to the fuzzing process. We want to compare fuzzing on a single machine without our system and fuzzing using our system with one executor on one machine. For this benchmark, we try several difficulties and different numbers of validators with one executor and one proposer. We used a machine with AMD Ryzen 9 6900HS (16) @ 4.935GHz and 32GB of DDR5-4800 RAM with our docker container for this benchmark.

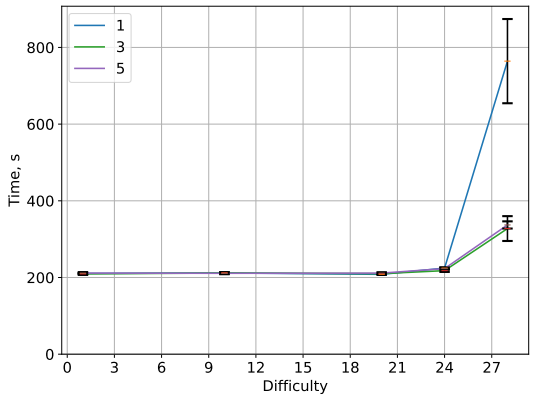


Fig. 8: System Overhead benchmark for different number of validators and difficulty parameter

Figure 8 depicts the relationship between the time of fuzzing of 5 proposals with 10 seeds each and the difficulty configuration of the system. It shows the exponential increase for the difficulty parameter from 20 to 30. This goes along with the fact that by increasing the difficulty parameter by 1 the cost of mining increases by 2 times. For the small values of the difficulty, the mining overhead is negligible.

*2) Validators Overhead:* In this benchmark, we want to check the scalability of our system with different numbers of validators. We compare fuzzing on a system with one executor, one proposer and different numbers of validators on one machine. We used a machine with Apple M1 (8 cores) @ 3.23GHz + 8 GB LPDDR4 SDRAM with our docker container for this benchmark.

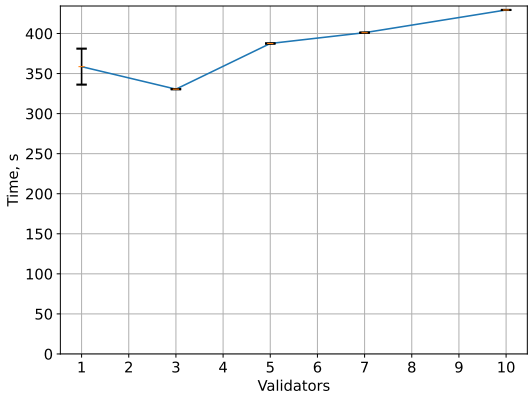


Fig. 9: Validator Overhead benchmark for different number of validators

Figure 9 depicts the relationship between the time of fuzzing of 5 proposals with 10 seeds each, with constant difficulty and increasing number of validators. It shows the overhead of adding validators only increasing gradually.

*3) Executors Speedup:* In this benchmark, we want to check the scalability of our system with different numbers of executors. We compare fuzzing on a system with five validators, one proposer and different numbers of executors on one machine. We used a machine with Intel Core i7-12700H (20) @ 4.70GHz and 16GB of DDR5-4800 RAM with Ubuntu 22.04.3 LTS for this benchmark

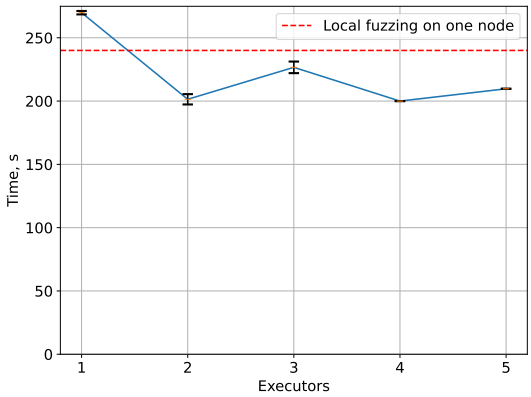


Fig. 10: Executor Speedup benchmark for different number of executors

Figure 10 depicts the relationship between the time of fuzzing of 5 proposals with 10 seeds each, constant difficulty and increasing number of executors. It shows a gradual decrease in time with more executors added to the system. After adding more executors, the fuzzing computational time decreases lower than the baseline of local fuzzing even with an additional overhead of the decentralized part of our system.

*4) Proposals Scalability:* In this benchmark, we want to check the scalability of our system with an increasing number of proposals of fuzzle from the proposer and different range sizes for each fuzzle. The range size of the fuzzle has an impact on the time of solving the fuzzle, on the other hand, the number of proposals affects the overhead on the validators since they need to mine a block for each fuzzle and solution. We used Google Cloud c2-standard-8 runner with Ubuntu 22.04.3 LTS for this benchmark.

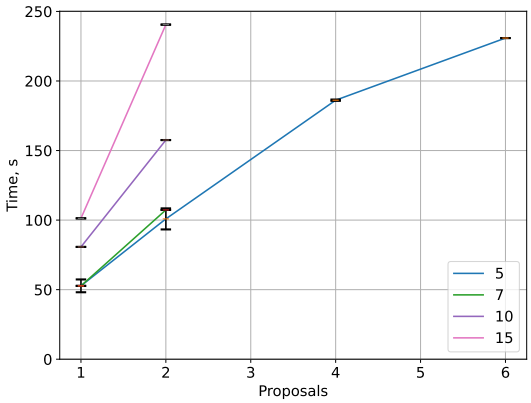


Fig. 11: Proposals Scalability benchmark for different number of range size

Figure 11 depicts the relationship between the time of fuzzing and the number of proposals. It depicts the fact that adding more proposals affects the system's performance more than the increasing range of each fuzzle. It shows the fact that the overhead of adding a new fuzzle is larger than the cost of increasing the range size. However, proposers can't increase it arbitrarily because of the time limit on the solving of fuzzles.

## VI. Conclusion

DissFuzz is the first system to support distributed and decentralised fuzzing based on untrusted nodes. Users can propose programs to fuzz, contribute resources to find bugs, and earn rewards. The system is proven to be robust against various attack vectors. The benchmark results show an encouraging result for the distributed and decentralized fuzzing. Our system has a better result compared to local fuzzing. The overhead added by our system is negligible compared to the achieved time speed-up.

## References

[1] E. Ahn, S. Kim, S. Park, J.-U. Hou, and D. Jang, "Efficient generation of program execution hash," vol. 10, 2022, pp. 61 707–61 720.

[2] D. Jang, "Fuzzcoin llvm," 2020. [Online]. Available: https://github.com/daehee87/fuzzcoin-llvm

[3] Google, "Clusterfuzz: a scalable fuzzing infrastructure," 2019. [Online]. Available: https://github.com/google/clusterfuzz

[4] D. Jang, A. Askar, I. Yun, S. Tong, Y. Cai, and T. Kim, "Fuzzing@home: Distributed fuzzing on untrusted heterogeneous clients," in Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses, ser. RAID '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–16. [Online]. Available: https://doi.org/10.1145/3545948.3545971

[5] S. Academy, "Bitcoin for developers," 2019. [Online]. Available: https://learn.saylor.org/course/view.php?id=500

[6] B. Wiki, “Difficulty,” 2023. [Online]. Available: https://en.bitcoin.it/wiki/Difficulty

## APPENDIX

### A. Detailed system schema

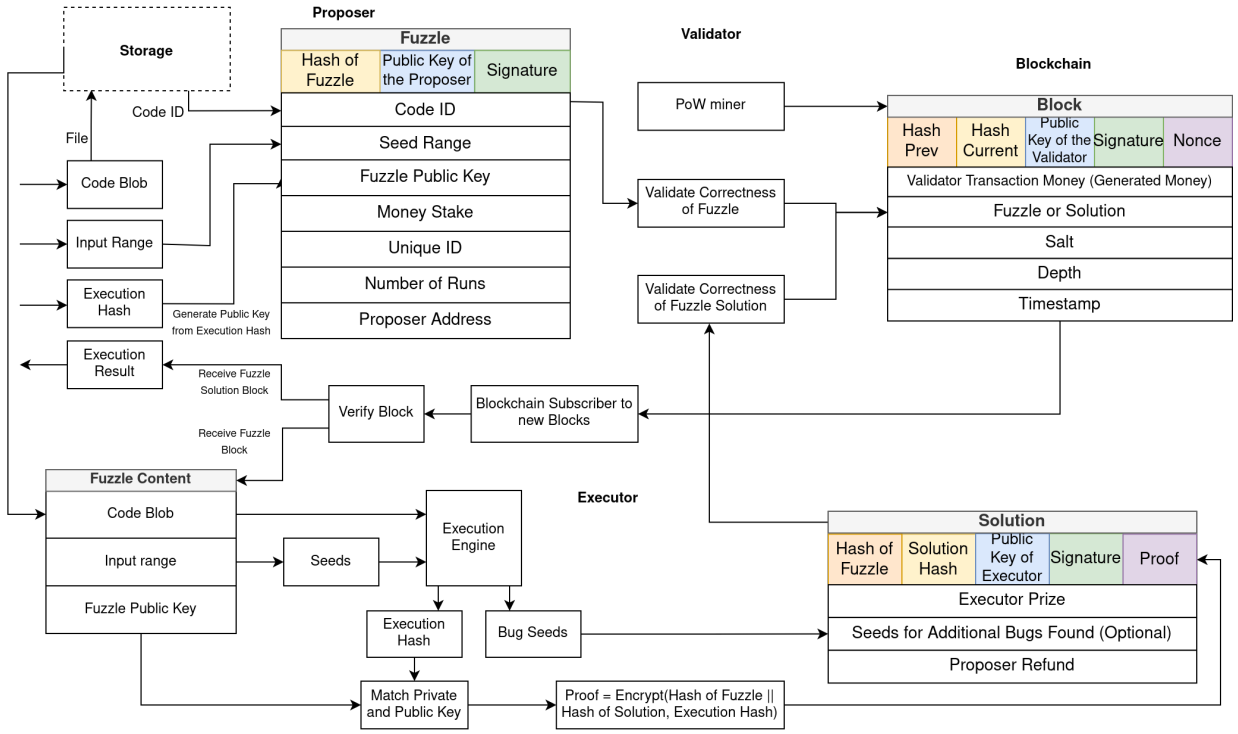


Fig. 12: Detailed system schema

### B. Future Improvements

This section contains features that were not yet implemented in the system.

*1) In-file blockchain storage:* For the in-file storage, each chain is stored in a separate file. The first element of the file is either the bootstrap block or either the first block of the fork.

To navigate through these files, we maintain a "master file" containing a mapping between the hash and the filename associated with the respective block.

*2) Bootstrapping:* Our system sets an initial balance for each user so that each user has enough balance to propose new fuzzles at the bootstrapping stage, which is essential to keeping the blockchain growing. This strategy is not safe because dishonest peers can leech by proposing each time with a new public key and never fuzz or mine, and a malicious node can propose multiple fuzzles with virtual public keys and solve them to get money easily.

A safer solution for bootstrapping is to allow empty block content so that nodes whose balance is below the threshold can mine blocks to get money. To motivate miners to include fuzzles/solutions, transaction fees should be introduced.

*3) Support for zip archives:* We need to add support for zip archives to handle PUTs that have more than one file (or depend on dynamic libraries that need to be included in the archive).