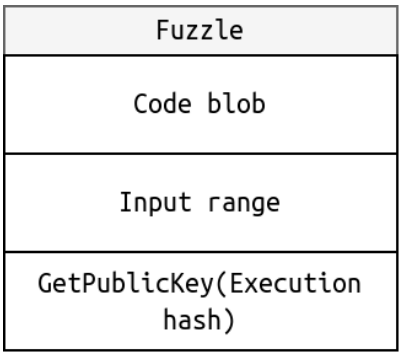
Decentralized concept

Summary:

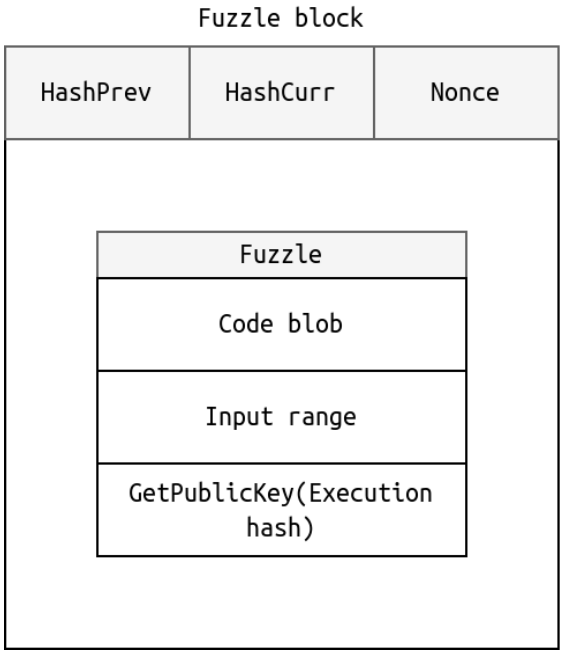
In this principal design, the challenges and requirements stated above will be addressed. The main idea is to use a blockchain to distribute fuzzing tasks and rewards for it, but instead of the classical Proof-of-Work scheme, we will use modified Proof-of-Fuzzing-Work. For simplicity, the possible optimisations of performance are not addressed in this design document. The high-level diagram will help with the big picture.

Core components:

- **Code blob.** A code blob is a binary executable file that has a single input argument from the fixed input range.
- **Execution hash**. Every code blob's execution is identified by a single unique value called execution hash. Different input arguments must produce different execution hashes. Everyone can reproduce the execution hash for the input argument by running the code blob with the specified input argument.
- **Fuzzle**. The fuzzing job (puzzle) that the Proposer wants to run on the Executors. It consists of three main parts:
    - Code blob of the program
    - Range of input arguments
    - Public key **which is derived** from the execution hash of some specific input from the range



- **Fuzzle block**. The block with the header and fuzzle. The header consists of:
    - Hash of the previous block
    - Hash of the payload of the current block
    - Nonce. The definition of it will be covered in the Blockchain component.



- **Fuzzle solution**. The solution for the Fuzzle is the specific input argument from the input range that was used to create the Fuzzle. The algorithm to determine that the input argument is in fact Fuzzle solution will be covered below.
- **Transaction**. The transaction is a transfer of money from the sender (possibly None) to the receiver. The transaction is valid either if the sender is specified and has enough money from the previous transactions or if the sender is not specified and the transaction is placed in the "right" block. (the second condition is needed to create money in the system, the condition for the transaction to be placed in the "right" block is under discussion)
- **Consolation prize transaction**. The consolation prize transaction is a transaction with the Executor which found the Fuzzle solution as a receiver. The sender could be either the Proposer which proposed the Fuzzle or None (if we want to make the system generates money). The validity of the transaction will be discussed in the Fuzzle solution block.

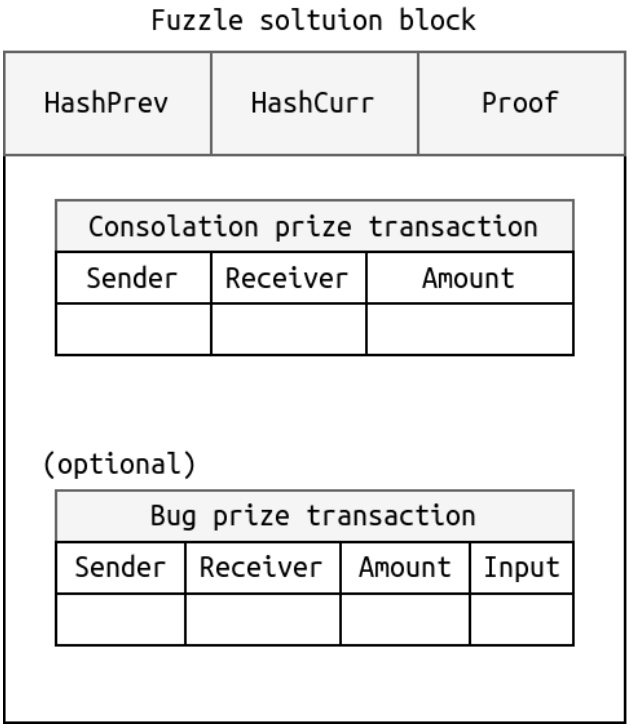
Consolation prize transaction		
Sender	Receiver	Amount

- **Bug prize transaction**. The bug prize transaction is a transaction with the input argument which leads to a bug and the Executor which found the bug as a receiver. The sender must be some Proposer with enough money remaining from the previous transactions. The validity of the transaction will be discussed in the Fuzzle solution block.

Bug prize transaction			
Sender	Receiver	Amount	Input

- **Fuzzle solution proof.** It is a zero-knowledge proof of solving the Fuzzle with the following requirements:
    - *Everyone* can verify that the author of the block solved the Fuzzle solution
    - *No one* other than the author of the block can use it to proof its Fuzzle solution
    - *No one* can use it to find its own proof of Fuzzle solution without code execution

- **Fuzzle solution block**. The header of the Fuzzle solution block must include apart from the HashPrev and HashCurr the Fuzzle solution proof. The block must include valid consolation prize transaction. The consolation prize transaction is valid if the zero-knowledge proof of solving the Fuzzle is verified and the amount of reward matches the rules of consolation prize rewards. Also, it might include valid bug rewards transactions. The bug rewards transaction is valid if the proof of solving is verified, the amount of reward matches the Proposer's proposed amount for the bug, and there is a connected bug seed which leads to the bug in the program.



- **Blockchain.** The Blockchain is a chain of valid Fuzzle solution blocks connected to the Fuzzle blocks using correct solution proofs. The Fuzzle blocks are connected to previous Fuzzle solution blocks using their nonces. Nonces are used to prevent spam consisted of Fuzzle blocks, so they should be relatively costly to compute. The one possible candidate for the nonce is the nonce from the Bitcoin (you need to find nonce such that Hash(HashPrev || HashCurr || nonce) = 0x00…0abc).

Fuzzle solution proof

In short: the idea is to use digital signature schema. The Proposer uses the execution hash of the target input argument as a private key, generates a public key from the private key, and include the public key in the Fuzzle. The Executor generates execution hashes from the input arguments and tests whether execution hash as a private key matches the public key of the Fuzzle. If it matches, then the Executor signs (|| HashCurr) and includes it as Fuzzle solution proof.
To use digital signature schema we need to have **private key** and **public key**. Then we can use private key to sign the content of the block. If the only way to effectively find the private key is to execute given code, then the signature will be a **proof of the code execution**. As the signature is bounded to their author block, no one can reuse it.

So the Fuzzle solution proof algorithm has three main parts:

- **Proposer phase**. Set up necessary information during the Fuzzle creation.
- **Executor phase**. Sign the block as a proof that the Fuzzle was solved.
- **Verification phase**. Use Fuzzle public key and verify the signature of the Fuzzle solution block.

Proposer phase

1. Choose the input argument range
2. Choose randomly the specific argument from the range
3. Execute the program with that argument and collect an execution hash from the run
4. Derive a public key from the execution hash (execution hash will be a private key)
5. Make a Fuzzle out of the code blob, input range and public key

Executor phase

1. For each argument from the input range:
    1.1. Execute the code blob with the input argument and collect an execution hash from the run
    1.2. Check if the execution hash matches the public key of the Fuzzle, finish the loop and save the execution hash if it matches
2. Sign HashCurr || HashPrev with the found execution hash and include it as the Fuzzer solution proof

Verification phase

1. Use public key of the Fuzzle and verify the signature of HashCurr || HashPrev of the Fuzzle solution block

Proof that the algorithm satisfies the Fuzzle solution proof properties:

- *Everyone* can verify that the author of the block solved the Fuzzle solution. This follows from the fact that the public key and signature are publicly available and the fact that it is move effective to find the private key solving the puzzle (otherwise you need to broutforce private key space to find the matching one).
- *No one* other than the author of the block can use it to proof its Fuzzle solution. The only public information is the public key and the signature of the original block. The signature can not be reused for other blocks from the properties of the digital signature.
- *No one* can use it to find its own proof of Fuzzle solution without code execution. Again, the only public information is the public key and the signature of the original block. The only information that can be derived from the good signature is the HashCurr || HashPrev which doesn't give any advantages for the Fuzzle solution.

Blockchain growth

Blockchain growth consists of two alternation rounds:

- **Proposers round**. Proposers propose next Fuzzle and publish Fuzzle block
- **Executors round**. Executors solve the Fuzzle and publish Fuzzle solution block

Proposers round

1. Every Proposer tries to include its own Fuzzle in the next block by computing nonce for the previous Fuzzle solution block

Executors round

1. Every Executor runs the code blob from the last Fuzzle using input arguments from the input range
2. Executor collects an execution hash from the run
3. Executor checks whether it matches the public key from the Fuzzle, if it is the case:
    a. Executor forms a consolation prize transaction and bugs prize transactions (if Executor found any bugs from executions)
    b. Executor includes these transactions to the Fuzzle solution block
    c. Executor signs the block using the found private key

Fuzzing

So, how the user can actually fuzz something in the network? The pipeline is as follow:

1. Prepare Fuzzle with the code blob, input ranges, and the public key
2. Find the nonce that matches the previous Fuzzle solution block
3. Wait for the blockchain to growth after your Fuzzle
4. If the Fuzzle is included in the blockchain, the next block will be a Fuzzle solution block, which contains possible bugs found in the range

Requirements

Let's look at the requirements stated before and see whether the proposed system satisfies them.

Code execution

Let's define the Proposer invalid if it sends invalid Fuzzle (for example, unsolvable). We do not cover Proposers which send "viruses", because we can assume that the code is executed in the restricted environment.

- **Safety.** If the Proposer is invalid, then it can't spam the system with invalid blocks (because of the nonce property). If the invalid Proposer find the correct nonce, it will send it to Executors, but Executors won't extend the incorrect blocks (they could drop the Fuzzle after some timeout).
- **Eventual termination.** This property is satisfied only under the assumption that the Proposer found the correct nonce faster than others and the Fuzzle is not very hard (so the Executors won't think that it is invalid Fuzzle).

Input/output data distribution

- **Deduplication**. This property **is not satisfied** because all Executors work on the same Fuzzle. However, after the Executor receives the Fuzzle solution block, it drops the current Fuzzle.
- **Adaptivity**. This property is satisfied if the Proposer adjust the input range according to the previous Fuzzle solutions.
- **Feedback**. If the Fuzzle is included in the blockchain, the Proposer will find possible bugs from the chained Fuzzle solution block.

Reward system

- **Trustworthy.** This property is satisfied because the Executor includes the reward transaction in the Fuzzle solution block. However, some Executor can be faster, so this property is satisfied only for the Fuzzle solution blocks which are included in the chain.
- **Consolation prize.** This property is satisfied because of the Consolation prize transaction in the Fuzzle solution block.
- **Fairness.** This is an open question. Should we add possibility for Proposer to adapt the cost of the Fuzzle?

Current design problems/attacks/open questions

- How to make money in the system? Should we include “out-of-thin-air” money for Executors who solve Fuzzles?
- How to batch Fuzzles? How to increase the throughput of the system? How to make Executors to not work on the same task? (and is it possible?)
- The Proposers do nothing during the Executors phase and vice versa
- With the “out-of-thin-air”: how to mitigate an issue that the Proposer creates a Fuzzle block and Fuzzle solution block itself to earn some money?
- How to solve the invalid Fuzzles problem? At what point should Executors drop the current Fuzzle?
- How to regulate the complexity of Fuzzles and reward proportionally? What if some Proposers create really hard Fuzzles, and some – really easy
- Should we allow Proposers to set up the pay for their Fuzzles, so the Executors can choice high-paid Fuzzles?
- How to regulate the complexity of the nonce finding for the Proposers? (should we use the same schema as the Bitcoin?)
- Proposers still waste resources to add a new Fuzzle. Can we solve this problem without making spam attack possible?

Design diagram

High level component diagram.