How does Automata VRF work

The VRF system consists of:

- DRAND as an entropy source.
- Off-chain oracle.
- AutomataVRFCoordinator
- contract.
-

The flow for each component is the following:

- An off chain oracle is responsible for performing the following tasks in a trusted manner (e.g. execution in TEE)
-
    - Computes the SHA256 hash of the signature, to match with the randomness value given by DRAND.
-
    - Computeshash_to_field
-
    - operation on the message.
-
    - Generates SNARK proof for theBLS12-381
-
    - signature verification.
-
    - Generates a finaluint256
-
    - random number.
-
    - Producessecp256k1
-
    - signature over the random number.
-
    - Submits the final random number, thesecp256k1
-
    - signature, computedhash_to_field
-
    - value and SNARK proofs toVRFCoordinator
- *
- VRFCoordinator
- is responsible for the following:
-
    - After the Off Chain Oracle has published randomness on chain:
-
        -
            - Performsecrecover
-
        -
            - to ensure the randomness is signed by a trusted off chain oracle.
-
        -
            - Re-constructhash_to_field
-
        -
            - to match with the input. This step serves as a point of revert for mismatch of hash, to prevent spending unnecessary gas on the SNARK verification.
-
        -
            - Performs SNARK verification of theBLS12-381
-
        -
            - Signature.
-
        -
            - After passing all of the above verification steps, it updates the state of the program, allowing consumers to fetch the latest round of randomness.

-

- Consumers can invoke:
-
  - VRFCoordinator.getLatestRandomness()
-
  - VRFCoordinator.getLatestRandomWords(uint32 numWords)
-
  - VRFCoordinator.requestRandomWords()
-
  - Or query event logs, to get randomness from past rounds.

- *
-

Detailed Description of the Trusted Off Chain Oracle Workflow

1. Upon receiving new randomness from DRAND, it performs SHA256 of the current BLS signature to verify the correctness of randomness. Denote DRAND randomness as A.
2. Sends the DRAND parameters to an API service that generates the following:
3.
    1. 55x7 integer representation
4.
    1. of the BLS signature for the current round.
5.
    1. hash_to_field
6.
    1. value.
7.
    1. SNARK proof for the BLS signature verification.
8. 6.
9. Generates a randomness of its own, denoted as B.
10. Performs A XOR B, to generate C.
11. Generates a final randomness R, using C as the seed.
12. Signs over the hash of the concatenation of R and SNARK proofs withsecp256k1
13. ECDSA.
14. Submits the randomness toAutomataVRFCoordinator
15. , with the following parameters:
16.
    1. DRAND return values.
17.
    1. SNARK proof, consists of uint256[2] proof_a, uint256[2][2] proof_b and uint256[2] proof_c. Seehere
18.
    1. for reference of convertingproof.json
19.
    1. to the proper format.
20.
    1. hash_to_field
21.
    1. The final R value. (Randomness)
22.
    1. The ECDSA signature
23. 8.
24.

```

Copy structSnarkProof{ uint256[2] a; uint256[2][2] b; uint256[2] c; }

structSignedRandomness{ uint256randomness; uint8v; bytes32r; bytes32s; }

structDrandParam{ uint64currentRound; uint256[7][2][2] currentSig;// offload conversion from hex to 55x7 representation bytesprevSig; }

///@noticecaller must be granted with SUBMITTER_ROLE functionsubmitRandomness( DrandParamcalldatadrand, SnarkProofcalldataproof, uint256[7][2][2]calldatahash_to_field,// offload conversion from hex to 55x7 representation
```

SignedRandomnesscalldatarandomness)external;

```

Detailed Description ofAutomataVRFCoordinator

The consumer should be aware of the difference between arandomness value andrandom words :

- Randomness
- : A singleuint256
- verifiable random value generated by the oracle. It can be used as a seed to perform more complex random generation.
- Random words
- : This is a list ofuint256
- words generated by taking the hash of the randomness value concatenated by its index. It can generate at most2**32
- random words in a single call. Consumers are free to specify the number of random words to produce from the randomness generated at the latest round. The code below describes how they are computed:
-

```

Copy uint256randomness; uint256[]memoryrandomWords=newuint256;

for(uint256i=0; i<numOfWords; i++) { randomWords[i]=uint256(keccak256(abi.encodePacked(randomness,i))); }

```

There are three ways which a consumer can fetch the latest randomness or list of random words:

1. InvokeVRFCoordinator.getLatestRandomness()
2. . This method returns the final randomness R generated by the off chain oracle. This is most likely called by an EOA, or a smart contract that does not conform with the Chainlink VRF workflow.
3. InvokeVRFCoordinator.getLatestRandomWords(uint32 numWords)
4. . This method returns a list of random words, with the number of elements specified bynumWords
5. . It is most likely called by an EOA, or a smart contract that does not conform with the Chainlink VRF workflow.
6. To provide compatibility with existing contracts that consumes Chainlink VRF (e.g. the calling contract extends theVRFConsumerBaseV2
7. interface), consumers can invoke therequestRandomWords()
8. entrypoint.
9.

Additional note on #3: If the consumer contract were to indirectly invoke the entrypoint via the LINK token transfer callback, the Receiver handler function would forward the call to the entrypoint and refund LINK tokens back to the consumer contract. However, this also would cost more gas than directly calling the entrypoint.

Last but not least, an event is defined in this program to allow consumers to query for randomness produced for all past rounds.

```

Copy eventRandomnessPublished(uint256indexedroundId,uint256randomness);

```

Consumer-facing Interface forAutomataVRFCoordinator

```

Copy // this interface is loosely based on the Chainlink VRFCoordinatorV2 interface

///@devsince we do not charge consumers for their LINK tokens /// we need to overwrite the onTokenTransfer() method to forward calls to requestRandomWords() /// and refund LINK tokens to consumers
import{ERC677ReceiverInterface}from"@chainlink/contracts/src/v0.8/interfaces/ERC677ReceiverInterface.sol";

interfaceIAutomataVRFCoordinatorisERC677ReceiverInterface{

/ ***@noticethis method is callable only by contracts that * extends the VRFConsumerBaseV2 contract * @paramnumWords consumers specify the number of random words to be gemerated * * Other parameters are simply placeholders to keep the interface consistent * with Chainlink VRFCoordinatorV2. * @devrawFulfillRandomWords() callback is done at the end of this function. * This is because the off chain oracle actively publishes new randomness * within a defined period. *** @returnthe currentRoundId /

functionrequestRandomWords(bytes32, uint64, uint16, uint32, uint32numWords)externalreturns(uint256roundId);

/ ***@noticefetch the latest round of randomness generated by the oracle**/
functiongetLatestRandomness()externalviewreturns(uint256randomness);

/ ***@noticethis method can be called by any addresses to get a list of random words * generated from the latest round of randomness** /
functiongetLatestRandomWords(uint32numWords)externalviewreturns(uint256[]memoryrandomWords);

functiongetCurrentRound()externalviewreturns(uint256roundId); }

```

Previous Why Automata VRF Next Attestation Last updated24 days ago On this page Was this helpful?
```