VRF integration

In this section, we will discuss how to integrate your smart contracts with the Band VRF. This section is separated into two sub-sections:Requesting andResolving .

Typically, when building on-chain applications that rely on an unpredictable outcome, such as lottery apps or games, the system requires a reliable source of randomness, and that is when the Band VRF comes into play.

VRF Flow

As shown in the VRF workflow, a request and callback model is used to obtain new random data. We will summarize the steps shown in the figure above into 3 main steps to make it easier to understand; steps 1 and 3 are on the EVM side, and step 2 is on the Bandchain side.

1. The process begins with a transaction that contains a consumer's request for random data from Band's VRFProvider contract.
2. After the transaction in step 1 is confirmed, an off-chain entity will grab parameters from its log to make a request transaction on Bandchain, producing the VRF result.
3. The last step is grabbing the result in step 2 with the proof of availability on Bandchain for making a relay transaction on the EVM side to resolve the request in the first step.

Contracts integration

Requesting

Assume that you are building an on-chain application that uses the Band VRF as a reliable source of randomness. Your contract(s) should contain a reference to Band's VRFProvider contract to be able to request random values.

First, let's define an interface for the VRFProvider contract.

interface

IVRFProvider

{ /// @dev The function for consumers who want random data. /// Consumers can simply make requests to get random data back later. /// @param seed Any string that used to initialize the randomizer. function

requestRandomData (string

calldata seed)

external

payable ; } Then, the consumer only needs to callrequestRandomData with a string parameter calledseed .

caution Please note that the seed is a generated string on the consumer side. Any two different consumers may be using the same seed. However, every consumer must use a unique seed for all their requests, as the VRFProvider has a mapping to check this condition. contract

VRFProvider

{ . . .

// Mapping that enforces the client to provide a unique seed for each request mapping (address

=>

mapping (string

=>

bool))

public hasClientSeed ;

. . . } After including theIVRFProvider , the consumer can now make a request-call to the VRFProvider contract, as shown in the example implementation below.

contract

MockVRFConsumer

{ IVRFProvider public provider ;

constructor (IVRFProvider _provider)

{ provider = _provider ; }

function

requestRandomDataFromProvider (string

calldata seed)

external

payable

{ provider . requestRandomData { value : msg . value } (seed) ; } } When callingrequestRandomData(seed) , the consumer can specifymsg.value to incentivize others to resolve the random data request. However, consumers can choose not to provide any incentive and resolve the request themselves.

After the consumer knows how to request from theVRFProvider , the consumer must also implement a callback function for the VRFProvider contract to call after the corresponding VRF result has been relayed.

The implementation below takes the previous step's code and adds theconsume function.

contract

MockVRFConsumer

{ IVRFProvider public provider ;

constructor (IVRFProvider _provider)

{ provider = _provider ; }

function

requestRandomDataFromProvider (string

calldata seed)

external

payable

{ provider . requestRandomData { value : msg . value } (seed) ; }

// ==========

string

public latestSeed ; uint64

public latestTime ; bytes32

public latestResult ;

function

consume (string

calldata seed ,

uint64 time ,

bytes32 result)

external override { require (msg . sender ==

address (provider) ,

"Caller is not the provider") ;

latestSeed

seed ; latestTime = time ; latestResult = result ; } } As shown above, theconsume function implements a logic that verifies whether the caller is the VRFProvider contract or not. This is to ensure that no one can call this function except the VRFProvider contract. With regards to the remaining logic in the example, the callback function only saves the callback data from the VRFProvider contract to its state.

Finally, we got our completeMockVRFConsumer contract that can make a request called theVRFProvider and then consume the VRF result once that request is resolved from theVRFProvider side.

Resolving

Anyone can get any existing requests in the VRFProvider contract by tracking the mapping called tasks. A task(VRF request) was designed to be resolved in a decentralized way. Therefore, there is no permission to resolve any task. The only thing that needs to resolve any unresolved tasks is the Merkle proof of the VRF result available on Bandchain.

The code below shows what atask looks like and the data structure(mapping) that helps track the tasks.

contract

VRFProvider

{

. . .

struct

Task

{ bool isResolved ; uint64 time ; address caller ; uint256 taskFee ; bytes32 seed ; bytes32 result ; string clientSeed ; }

// A global variable representing the number of all tasks in this contract uint64

public taskNonce ;

// Mapping from nonce => task mapping (uint64

=> Task)

public tasks ;

. . .

function

relayProof (bytes

calldata _proof ,

uint64 _taskNonce)

external nonReentrant { . . . }

. . .

} When a resolver(a self-implemented worker, a bot, a bounty hunter, etc.) finds an unresolved request, the resolver can resolve it by requesting the VRF randomness on the BandChain. After the VRF result is finalized on the BandChain, the resolver can retrieve the Merkle proof of availability of the result and then relay the proof via arelayProof function on the VRFProvider contract. The resolver also needs to specify the nonce of the task it wants to resolve.

The next step will demonstrate how to manually request and resolve the VRF randomness using the user interfaces of Cosmoscan(Band) and Etherscan(EVM).

Manually request and resolve

This section will demonstrate how to request random data from theVRFProvider and then resolve the request manually usingGoerli andLaozi-Testnet6 UI.

Firstly, go to theVRFProvider contract on Goerli to view some of its global variables.

To request the VRF randomness on Bandchain, we need to know theoracleScriptID . The VRF oracle script is the ID335 on the laozi-Testnet6.

Now, let's move to theMockVRFConsumer contract to begin the VRF flow started by calling a functionrequestRandomDataFromProvider .

The video below shows the flow of theMockConsumer that requests the VRF random value from theVRFProvider .

Recommended Oracle Script Request Settings

Parameter Value Prepare Gas 100000 Execute Gas 400000 Gas Limit (Leave Blank) Ask Count 4 Min Count 3

Implement your own resolver

An alternative to manually resolve the request is to use a resolver bot. Anyone can implement their own version of resolver bot to automate the resolving process. We provide an open-source version of Band's VRF worker bot, which is available atVRFWorkerV1 repository . Previous Getting Started Next Example Use Cases