Converting from Chainlink VRF to Secret VRF in four steps

Got improvements or suggestions on how to convert your ChainlinkVRF contract to SecretVRF ? Please ask in the Secret NetworkTelegram or Discord. Converting from Chainlink VRF to Secret VRF is easier than you expect. Within four easy steps you can free your contract from bloat and use the lightweight and faster Secret VRF solution.

We start off with the example code from Chainlink fromhere :

```

Copy // SPDX-License-Identifier: MIT pragmasolidity^0.8.7;

import {VRFCoordinatorV2Interface} from "@chainlink/contracts@1.0.0/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol"; import{VRFConsumerBaseV2}from"@chainlink/contracts@1.0.0/src/v0.8/vrf/VRFConsumerBaseV2.sol";

contractVRFD20isVRFConsumerBaseV2{ uint256privateconstantROLL_IN_PROGRESS=42;

VRFCoordinatorV2Interface COORDINATOR;

// Your subscription ID. uint64s_subscriptionId;

// Sepolia coordinator. For other networks, // see https://docs.chain.link/docs/vrf-contracts/#configurations addressvrfCoordinator=0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625;

// The gas lane to use, which specifies the maximum gas price to bump to. // For a list of available gas lanes on each network, // see https://docs.chain.link/docs/vrf-contracts/#configurations bytes32s_keyHash= 0x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c;

// Depends on the number of requested values that you want sent to the // fulfillRandomWords() function. Storing each word costs about 20,000 gas, // so 40,000 is a safe default for this example contract. Test and adjust // this limit based on the network that you select, the size of the request, // and the processing of the callback request in the fulfillRandomWords() // function. uint32callbackGasLimit=40000;

// The default is 3, but you can set this higher. uint16requestConfirmations=3;

// For this example, retrieve 1 random value in one request. // Cannot exceed VRFCoordinatorV2.MAX_NUM_WORDS. uint32numWords=1; addresss_owner;

// map rollers to requestIds mapping(uint256=>address)privates_rollers; // map vrf results to rollers mapping(address=>uint256)privates_results;

eventDiceRolled(uint256indexedrequestId,addressindexedroller); eventDiceLanded(uint256indexedrequestId,uint256indexedresult);

constructor(uint64subscriptionId)VRFConsumerBaseV2(vrfCoordinator) { COORDINATOR=VRFCoordinatorV2Interface(vrfCoordinator); s_owner=msg.sender; s_subscriptionId=subscriptionId; }

functionrollDice( addressroller )publiconlyOwnerreturns(uint256requestId) { require(s_results[roller]==0,"Already rolled"); // Will revert if subscription is not set and funded. requestId=COORDINATOR.requestRandomWords( s_keyHash, s_subscriptionId, requestConfirmations, callbackGasLimit, numWords );

s_rollers[requestId]=roller; s_results[roller]=ROLL_IN_PROGRESS; emitDiceRolled(requestId,roller); }

functionfulfillRandomWords( uint256requestId, uint256[]memoryrandomWords )internaloverride{ uint256d20Value= (randomWords[0] %20)+1; s_results[s_rollers[requestId]]=d20Value; emitDiceLanded(requestId,d20Value); }

functionhouse(addressplayer)publicviewreturns(stringmemory) { require(s_results[player]!=0,"Dice not rolled"); require(s_results[player]!=ROLL_IN_PROGRESS,"Roll in progress"); returngetHouseName(s_results[player]); }

functiongetHouseName(uint256id)privatepurereturns(stringmemory) { string[20]memoryhouseNames=[ "Targaryen", "Lannister", "Stark", "Tyrell", "Baratheon", "Martell", "Tully", "Bolton", "Greyjoy", "Arryn", "Frey", "Mormont", "Tarley", "Dayne", "Umber", "Valeryon", "Manderly", "Clegane", "Glover", "Karstark" ]; returnhouseNames[id-1]; }

modifieronlyOwner() { require(msg.sender==s_owner); _; } }

```

Removing Chainlink bloat

In the first step, we remove the imports and the inheritance of the VRFConsumerBaseV2 and add our SecretVRF interface from this. There is no

```

Copy import {VRFCoordinatorV2Interface} from "@chainlink/contracts@1.0.0/src/v0.8/vrf/interfaces/VRFCoordinatorV2Interface.sol"; import{VRFConsumerBaseV2}from"@chainlink/contracts@1.0.0/src/v0.8/vrf/VRFConsumerBaseV2.sol";

contractVRFD20isVRFConsumerBaseV2{ uint256privateconstantROLL_IN_PROGRESS=42;

VRFCoordinatorV2Interface COORDINATOR;

// Your subscription ID. uint64s_subscriptionId;

// Sepolia coordinator. For other networks, // see https://docs.chain.link/docs/vrf-contracts/#configurations addressvrfCoordinator=0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625;

// The gas lane to use, which specifies the maximum gas price to bump to. // For a list of available gas lanes on each network, // see https://docs.chain.link/docs/vrf-contracts/#configurations bytes32s_keyHash= 0x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c;

// Depends on the number of requested values that you want sent to the // fulfillRandomWords() function. Storing each word costs about 20,000 gas, // so 40,000 is a safe default for this example contract. Test and adjust // this limit based on the network that you select, the size of the request, // and the processing of the callback request in the fulfillRandomWords() // function. uint32callbackGasLimit=40000; //... }

```

to get to this:

```

Copy interfaceISecretVRF{ function requestRandomness(uint32 _numWords, uint32 _callbackGasLimit) external payable returns (uint256 requestId); } contractVRFD20{ uint256privateconstantROLL_IN_PROGRESS=42; //...

// Depends on the number of requested values that you want sent to the // fulfillRandomWords() function. Storing each word costs about 20,000 gas, // so 40,000 is a safe default for this example contract. Test and adjust // this limit based on the network that you select, the size of the request, // and the processing of the callback request in the fulfillRandomWords() // function. uint32callbackGasLimit=40000;

// Sepolia Gateway. For other networks, // see https://docs.chain.link/docs/vrf-contracts/#configurations addresspublicVRFGateway=0x3879E146140b627a5C858a08e507B171D9E43139; }

```

Here, the behavior of thecallbackGasLimit is different than in ChainlinkVRF. The callback Gas limit is simply all of the gas that you need to pay in order to make callback, which includes the verification of the result as well. The callback gas is the amount of gas that you have to pay for the message coming on the way back. We recommend at least using100_000+ in Callback gas to ensure that enough gas is available. In case you did not pay enough gas, the contract callback execution will fail.

Simplifying the Constructor

Next, we can also simplify the constructor since we do not need to define any extra variables or subscriptionIds. Going from this:

```

Copy constructor(uint64subscriptionId)VRFConsumerBaseV2(vrfCoordinator) { COORDINATOR=VRFCoordinatorV2Interface(vrfCoordinator); s_owner=msg.sender; s_subscriptionId=subscriptionId; }

```

to this

```

Copy constructor() { s_owner=msg.sender; }

```

Change request randomness function

Next, we need to slightly adjust the behavior of the functionrollDice(address roller) function and how it calls the Request Randomness function within Secret VRF. Here, we need to use the Secet VRF gateway and call it directly instead.

Make sure to now mark this function aspayable ! ```

Copy functionrollDice(addressroller)publicpayableonlyOwnerreturns(uint256requestId) {
require(s_results[roller]==0,"Already rolled"); // Will revert if subscription is not set and funded.

// Get the VRFGateway contract interface ISecretVRF vrfContract=ISecretVRF(VRFGateway);

// Call the VRF contract to request random numbers. // Returns requestId of the VRF request. A contract can track a VRF call that way. uint256requestId=vrfContract.requestRandomness{value:msg.value}(_numWords,_callbackGasLimit);

s_rollers[requestId]=roller; s_results[roller]=ROLL_IN_PROGRESS; emitDiceRolled(requestId,roller); } }

```

Please make sure to actually prepay the right amount of callback gas directly as a value transfer into the contract. The callback gas is the amount of gas that you have to pay for the message coming on the way back. If you do pay less than the amount specified below, your Gateway TX will fail:

```

Copy ///@noticeIncrease the task_id to check for problems ///@param_callbackGasLimit the Callback Gas Limit

functionestimateRequestPrice(uint32_callbackGasLimit)privateviewreturns(uint256) {
uint256baseFee=_callbackGasLimit*block.basefee; returnbaseFee; }

```

Since this check is dependent on the currentblock.basefee of the block it is included in, it is recommended that you estimate the gas fee beforehand and add some extra overhead to it. An example of how this can be implemented in your frontend can be found in thisexample and here:

```

Copy //Then calculate how much gas you have to pay for the callback //Forumla: callbackGasLimit*block.basefee. //Use an appropriate overhead for the transaction, 1,5x = 3/2 is recommended since gasPrice fluctuates.

constgasFee=awaitprovider.getGasPrice(); constamountOfGas=gasFee.mul(callbackGasLimit).mul(3).div(2);

```

Add a check for the Secret VRF Gateway

Lastly, we'll add a small check to ensure that we actually got an incoming call from the SecretVRF gateway contract. For this, remove theinternal andoverride flags on the function and add the require:

```

Copy functionfulfillRandomWords(uint256requestId, uint256[]memoryrandomWords)external{
require(msg.sender==address(VRFGateway),"only Secret Gateway can fulfill");

uint256d20Value=(randomWords[0] %20)+1; s_results[s_rollers[requestId]]=d20Value;
emitDiceLanded(requestId,d20Value); }

```

Conclusion

That's all that you need to convert your contract from ChainlinkVRF to SecretVRF.

Last updated20 days ago On this page * Removing Chainlink bloat * Simplifying the Constructor * Change request randomness function * Add a check for the Secret VRF Gateway* Conclusion

Was this helpful? Edit on GitHub Export as PDF
```