Supra, VRF

Community member contribution Shoutout to@ksdumont for contributing the followingthird-party document ! Supra's VRF can provide the exact properties required for a random number generator (RNG) to be fair with tamper-proof, unbiased, and cryptographically verifiable random numbers to be employed by smart contracts.

Integrating with Supras' VRF is quick and easy. Supra currently supports several Solidity/EVM-based networks, like Arbitrum, and non-EVM networks like Sui, Aptos.

To get started, you will want to visitSupras' docs site and review the documentation or continue to follow this guide for a quick start.

Latest version of Supra VRF requires a customer controlled wallet address to act as the main reference for access permissions and call back(response) transaction gas fee payments. Therefore, users planning to consume Supra VRF should get in touch with our team to get your wallet registered with Supra.

Once your wallet is registered by the Supra team, you could use it to whitelist any number of VRF requester smart contracts and pre-pay/top up the deposit balance maintained with Supra in order to pay for the gas fees of callback(response) transactions.

You will be interacting with two main contracts:

- Supra Deposit Contract:
- To whitelist smart contracts under the registered wallet address, pre-pay/top up the callback gas fee deposit maintained with Supra.
- Supra Router Contract:
- To request and receive random numbers.

Step 1: Create the Supra router contract interface

Add the following code to the requester contract i.e, the contract which uses VRF as a service. You can also add the code in a separate interface and inherit the interface in the requester contract.

interface ISupraRouterContract { function generateRequest(string memory _functionSig, uint8 _rngCount, uint256 _numConfirmations, uint256 _clientSeed, address _clientWalletAddress) external returns(uint256); function generateRequest(string memory _functionSig, uint8 _rngCount, uint256 _numConfirmations, address _clientWalletAddress) external returns(uint256); } This interface will help the requester contract interact with the Supra router contract and through which the requester contract can use the VRF service.

Step 2: Configure the Supra router contract address

Contracts that need random numbers should utilize the Supra router contract. In order to do that, they need to create an interface and bind it to the on-chain address of the Supra router contract.

contract ExampleContract { ISupraRouter internal supraRouter;

constructor(address routerAddress) { supraRouter = ISupraRouter(0x7d86fbfc0701d0bf273fd550eb65be1002ed304e); } }

Step 3: Use the VRF service and request a random number

In this step, we will use the "generateRequest" function of the Supra Router Contract to create a request for random numbers. There are two modes for the "generateRequest" function. The only difference between them is that you can optionally provide a client-side input, which will also be part of the payload being threshold signed to provide randomness.

_functionSig- a string parameter, here the requester contract will have to pass the function signature which will receive the callback i.e., a random number from the Supra Router Contract. The function signature should be in the form of the function name following the parameters it accepts. We will see an example later in the document. _rngCount - an integer parameter, it is for the number of random numbers a particular requester wants to generate. Currently, we can generate a maximum of 255 random numbers per request. _numConfirmations - an integer parameter that specifies the number of block confirmations needed before supra VRF can generate the random number. _clientSeed (optional) - an optional integer parameter that could be provided by the client (defaults to 0). This is for additional unpredictability. The source of the seed can be a UUID of 256 bits. This can also be from a centralized source. _clientWalletAddress - an "address" type parameter, which takes the client wallet address which is already registered with the Supra Team, as input. Supra's VRF process requires splitting the contract logic into two functions. The request function - the signature of this function is up to the developer The callback function - the signature must be of the form "uint256 nonce, uint256[] calldata rngList"

function exampleRNG() external { //Function validation and logic // requesting 10 random numbers uint8 rngCount = 10;

// we want to wait for 1 confirmation before the request is considered complete/final uint256 numConfirmations = 1; address _clientWalletAddress = //Add the whitelisted wallet address here uint256 generated_nonce = supraRouter.generateRequest(“exampleCallback(uint256,uint256[])”, rngCount, numConfirmations, _clientWalletAddress);

// store generated_nonce if necessary (eg: in a hashmap) // this can be used to track parameters related to the request, such as user address, nft address etc in a lookup table // these can be accessed inside the callback since the response from supra will include the nonce }

Step 4 - Add the validation in the callback function of requester contract

Inside the callback function where the requester contract wants the random number (in this example the callback function is exampleCallback), the requester contract will have to add the validation such that only the Supra router contract can call the function. The validation is necessary to protect against malicious contracts/users executing the callback with fake data. For example, if the callback function is pickWinner in the requester contract, the snippet can be as follows.

function exampleCallback(uint256 _nonce ,uint256[] _rngList) external { require(msg.sender == address(SupraRouter)); // Following the required logic of the function }

Step 5 : Whitelist your requester contract with Supra deposit contract and deposit funds

It is important to note that your wallet address must be registered with Supra before this step. If that is completed, then you need to whitelist your requester smart contract under your wallet address and deposit funds to be paid for your call back transactions gas fees. The simplest way to interact with the deposit contract will be through Remix IDE. Go to Remix IDE & create a file with name IDepositContract.sol Paste the following code in the file:

interface IDepositUserContract { function depositFundClient() external payable; function addContractToWhitelist(address _contractAddress) external; function removeContractFromWhitelist(address _contractAddress) external; function setMinBalanceClient(uint256 _limit) external; function withdrawFundClient(uint256 _amount) external; function checkClientFund(address _clientAddress) external view returns (uint256); function checkEffectiveBalance(address _clientAddress) external view returns (uint256); function checkMinBalanceSupra() external view returns (uint256); function checkMinBalance(address _clientAddress) external view returns(uint256); function countTotalWhitelistedContractByClient(address _clientAddress) external view returns (uint256); function getSubscriptionInfoByClient(address _clientAddress) external view returns (uint256, uint256, bool); function isMinimumBalanceReached(address _clientAddress) external view returns (bool); function listAllWhitelistedContractByClient(address _clientAddress) external view returns (address[] memory);

} Navigate to the “Navigate & run Transactions” tab in remix, and paste the deposit contract address into the text box besides the “At Address” button & press the at address button. You will find the instance for the deposit contract created using which a user can interact and use the features provided by the deposit contract. Following functions will facilitate whitelisting your requester smart contracts and fund deposits. “addContracttoWhitelist(address)” - The whitelisted users will have to whitelist their contract which they will be using to request for random numbers. The parameter this function takes is the User’s contract address. This function will be called after the user deploys the requester contract post development and makes it ready for interacting with the Supra Contracts. “depositFundClient()” - is another mandatory function for a user to use once, before the user starts requesting from that contract. This is a function which will deposit funds in the deposit contract from the users for the response/callback transaction. The funds for a specific user should remain higher than the minimum amount set by the Supra(0.1 Eth for Arbitrum testnet) for the new request transactions to be accepted. Basically the gist here is that the user will have to interact with the Deposit contract and add funds for their accounts, which will be utilized for the response transaction gas fee. There will be a script from Supra which will be monitoring the funds and will alert the user if a refill is required.

Example implementation

// SPDX-License-Identifier: MIT pragma solidity ^0.8.0; interface ISupraRouter { function generateRequest(string memory _functionSig , uint8 _rngCount, uint256 _numConfirmations, uint256 _clientSeed,address _clientWalletAddress) external returns(uint256); function generateRequest(string memory _functionSig , uint8 _rngCount, uint256 _numConfirmations,address _clientWalletAddress) external returns(uint256); } contract Interaction { address supraAddr; constructor(address supraSC) { supraAddr = supraSC; } mapping (uint256 => string) result; mapping (string => uint256[]) rngForUser; function getRNGForUser(uint8 rngCount, string memory username) external { uint256 nonce = ISupraRouter(supraAddr).generateRequest("myCallbackUsername(uint256,uint256[])", rngCount, 1, 123, msg.sender); //Can pass "msg.sender" when calling from the whitelisted wallet address result[nonce] = username; } function myCallbackUsername(uint256 nonce, uint256[] calldata rngList) external { require(msg.sender == supraAddr, "only supra router can call this function"); uint8 i = 0; uint256[] memory x = new uint256; rngForUser[result[nonce]] = x; for(i=0; i<rngList.length;i++){ rngForUser[result[nonce]][i] = rngList[i] % 100; } } function viewUserName(string memory username) external view returns (uint256[] memory) { return rngForUser[username]; } }

Going further with Supra

If you want to take the next step, consider registering for theSupra Network Activate Program (SNAP) .

The Supra Network Activate Program (SNAP) offers companies discounted oracle credits, technical documentation, and customer support to embed much-needed oracles and VRF/RNG. SNAP supports Web3 scaling and growth to buffer costs which could typically inhibit a company's success.

The SNAP program is partnered with some of Web3's most prolific names who are helping with project selection and qualification.

Connect with us

Still looking for answers? We got them! Check out all the ways you can reach us:

- Visit us atsupraoracles.com
- Read ourDocs
- Chat with us onTelegram
- Follow us onTwitter
- Join ourDiscord
- Check us out onYoutube