

Example Use Cases

Band VRF provides deterministic pre-commitments for low entropy inputs, which must resist brute-force pre-image attacks. In addition, the VRF can be used as defense against offline enumeration attacks (such as dictionary attacks) on data stored in hash-based data structures. Therefore, it can be used as a secure source of on-chain randomness. The Band VRF use cases are outlined below.

Use cases

We separate the use cases into four categories based on the behaviors of the VRF users/consumers.

1. One-time-use:
2. Consumers only request random data once in their product lifetime, typically when they initiate their contracts.
3.
 - NFT minting - in the case where a single random seed is used to generate the entire collection.
4. Batch-use:
5. Consumers request random data multiple times but with a countable number of requests for the entire life of their product.
6.
 - NFT minting - in the case where every single ID will be minted one by one. Therefore, whenever the end-user tries to mint an NFT, the VRF request is created to resolve the minting. This process will continue until the entire collection is minted.
7. Interval-use:
8. Consumers set their specific intervals to request random data from our VRF protocol. This process will continue indefinitely since some parts of their products rely on a trusted source of randomness.
9.
 - Lottery dApps (predetermined start-end, calculatable start-end)
10.
 - NFT minting with a specific minting interval
11. Continuous-use:
12. Consumers use randomness as parts of their product with no specific interval. Therefore, they can request random data at any time based on the internal logic of their contracts/system.
13.
 - Lottery dApps (no predetermined interval, unable to calculate future start-end)
14.
 - On-chain games
15.
 - NFT on-demand minting

Lottery Example (Continuous-use)

This on-chain lottery dApp is an example of a continuous-use VRF, and it satisfies the requirements below.

Requirements:

- Only the owner can set the minimum price and the round's duration.
- Only the owner can start a new round.
- The owner determined the seed of the started round.
- Anyone can buy lotteries during a started round.
- Anyone can request to resolve the current round if it has ended.
- Only the VRF Provider contract can resolve the resolving round.

pragma

solidity

^ 0.8.17 ;

interface

IVRFConsumer

```
{ /// @dev The function is called by the VRF provider in order to deliver results to the consumer. /// @param seed Any string that used to initialize the randomizer. /// @param time Timestamp where the random data was created. /// @param result A random bytes for given seed and time. function
```

```

consume ( string

calldata seed , uint64 time , bytes32 result )

external ; }

interface

IVRFPProvider

{ /// @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 ; }

contract

SimpleLottery

is IVRFConsumer {

event

Buy ( address buyer ,

uint256 buyerIndex ,

uint256 roundNumber ,

uint256 buyPrice ) ; event

StartRound ( uint256 roundNumber ,

string seed ) ; event

ResolvingRound ( uint256 roundNumber ,

string seed ) ; event

RoundResolved ( uint256 roundNumber ,

string seed ,

bytes32 result ) ;

address

public owner ;

uint256

public minLotteryPrice ; uint256

public roundDuration ; uint256

public roundCount ; bool

public isResolvingCurrentRound ;

IVRFPProvider public provider ;

struct

Round

{ uint256 startBlock ; uint256 endBlock ; string seedOfRound ; address [ ] buyers ; }

mapping ( uint256

```

=> Round)

public rounds ;

constructor (IVRFProvider _provider ,

uint256 _minLotteryPrice ,

uint256 _roundDuration)

{ provider = _provider ; minLotteryPrice = _minLotteryPrice ; roundDuration = _roundDuration ; owner = msg . sender ; }

function

isCurrentRoundStart ()

public

view

returns (bool)

{ return rounds [roundCount] . startBlock

0 ; }

function

currentRoundBlocksRemaining ()

public

view

returns (uint256)

{ Round memory currentRound = rounds [roundCount] ; if

(block . number

currentRound . endBlock)

{ return

0 ; } return currentRound . endBlock - block . number ; }

function

setMinLotteryPrice (uint256 _minLotteryPrice)

external

{ require (msg . sender == owner ,

"SimpleLottery: not the owner") ; require (! isCurrentRoundStart () ,

"SimpleLottery: this round is in progress") ;

minLotteryPrice

_minLotteryPrice ; }

function

setRoundDuration (uint256 _roundDuration)

external

{ require (msg . sender == owner ,

"SimpleLottery: not the owner") ; require (! isCurrentRoundStart () ,

"SimpleLottery: this round is in progress") ;

roundDuration

```
_roundDuration ; }

function

startANewRound ( string
memory roundSeed )

external

{ require ( msg . sender == owner ,
"SimpleLottery: not the owner" ) ; require ( ! isCurrentRoundStart ( ) ,
"SimpleLottery: this round is in progress" ) ;

Round memory currentRound = rounds [ roundCount ] ;

currentRound . seedOfRound = roundSeed ; currentRound . startBlock = block . number ; currentRound . endBlock =
currentRound . startBlock + roundDuration ;

rounds [ roundCount ]
= currentRound ; emit

StartRound ( roundCount , roundSeed ) ; }

function

buy ( )

external

payable

{ require ( currentRoundBlocksRemaining ( )
0 ,
"SimpleLottery: this round is not in progress" ) ; require ( msg . value
= minLotteryPrice ,
"SimpleLottery: given price is too low" ) ;

uint256 currentBuyerIndex = rounds [ roundCount ] . buyers . length ; emit

Buy ( msg . sender , currentBuyerIndex , roundCount , msg . value ) ;

rounds [ roundCount ] . buyers . push ( msg . sender ) ; }

function

resolveCurrentRound ( )

external

{ require ( isCurrentRoundStart ( ) ,
"SimpleLottery: this round is not started yet" ) ; require ( currentRoundBlocksRemaining ( )
==
0 ,
"SimpleLottery: this round has not ended yet" ) ; require ( ! isResolvingCurrentRound ,
"SimpleLottery: round is resolving" ) ;

Round memory currentRound = rounds [ roundCount ] ; if
```

```

( currentRound . buyers . length
0 )
{ isResolvingCurrentRound =
true ;
provider . requestRandomData { value :
0 } ( currentRound . seedOfRound ) ; emit
ResolvingRound ( roundCount , currentRound . seedOfRound ) ; }
else
{ emit
RoundResolved ( roundCount , currentRound . seedOfRound ,
bytes32 ( 0 ) ) ;
roundCount +=
1 ; isResolvingCurrentRound =
false ; } }
function
consume ( string
calldata seed ,
uint64 time ,
bytes32 result )
external override { require ( msg . sender ==
address ( provider ) ,
"Caller is not the provider" ) ; require ( isResolvingCurrentRound ,
"SimpleLottery: round is not resolving" ) ;
Round memory currentRound = rounds [ roundCount ] ; address winner = currentRound . buyers [ uint256 ( result )
% currentRound . buyers . length ] ;
emit
RoundResolved ( roundCount , seed , result ) ;
roundCount +=
1 ; isResolvingCurrentRound =
false ;
winner . call { value :
address ( this ) . balance } ( "" ) ; } } We have deployed the reference contracts to the Goerli testnet here.
Contract Address Bridge 0xD291A502e3ca4Bb13E09892e57d8Ff0271Bd198A VRFProvider
0xF1F3554b6f46D8f172c89836FBeD1ea8551eabad VRFLens 0x6e876b4Ed458af275Eb049a3f89BF0909618d154
SimpleLottery 0xCD3528283aA330003E50350134a48d1920BA70A0

```

NFT Minting Example (Batch-use)

This NFT is an example of a batch-use VRF, and it satisfies the requirements below.

Requirements:

- The max supply is set once at the time the contract is deployed.
- Anyone can call mintWithVRF
- to start minting an NFT for themselves.
- An actual minting is done when the VRF provider resolves the token id for the minter/receiver.

```
pragma
solidity
^ 0.8.17 ;
import
{ ERC721Enumerable }
from
"@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol" ;
/* * @dev String operations. / library
Strings
{ bytes16
private
constant _HEX_SYMBOLS =
"0123456789abcdef" ;
/* * @dev Converts a uint256 to its ASCII string decimal representation. / function
toString ( uint256 value )
internal
pure
returns
( string
memory )
{ // Inspired by OraclizeAPI's implementation - MIT licence // https://github.com/oraclize/ethereum-
api/blob/b42146b063c7d6ee1358846c198246239e9360e8/oraclizeAPI_0.4.25.sol
if
( value ==
0 )
{ return
"0" ; } uint256 temp = value ; uint256 digits ; while
( temp !=
0 )
{ digits ++ ; temp /=
10 ; } bytes
memory buffer =
new
bytes ( digits ) ; while
( value !=
```

```

0 )
{ digits -=
1 ; buffer [ digits ]
=
bytes1 ( uint8 ( 48
+
uint256 ( value %
10 ) ) ) ; value /=
10 ; } return
string ( buffer ) ; } }

interface
IVRFConsumer

{ /// @dev The function is called by the VRF provider in order to deliver results to the consumer. /// @param seed Any string
that used to initialize the randomizer. /// @param time Timestamp where the random data was created. /// @param result A
random bytes for given seed and time. function

consume ( string
calldata seed , uint64 time , bytes32 result )

external ; }

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 ; }

contract
ExampleNFT
is ERC721Enumerable , IVRFConsumer { using
Strings
for
uint256 ;

IVRFProvider public immutable provider ; uint256
public immutable maxSupply ;

uint256
public mintRequestCount =
0 ; uint256
public mintResolveCount =
0 ;

```

```

mapping ( uint256
=>
uint256 )
public tokenMintingLogs ; mapping ( string
=>
address )
public tokenSeedToMinter ;
constructor ( IVRFProvider _provider ,
uint256 _maxSupply )
ERC721 ( "ExampleNFT" ,
"ENFT" )
{ provider = _provider ; maxSupply = _maxSupply ; }
function
mintWithVRF ( )
external
{ require ( mintRequestCount < maxSupply ,
"Reach max supply" ) ; string
memory clientSeed =
string ( abi . encodePacked ( "ExampleNFT-" , mintRequestCount . toString ( ) ) ) ; tokenSeedToMinter [ clientSeed ]
= msg . sender ;
mintRequestCount ++ ;
provider . requestRandomData { value :
0 } ( clientSeed ) ; }
function
consume ( string
calldata seed ,
uint64 time ,
bytes32 result )
external override { require ( msg . sender ==
address ( provider ) ,
"Caller is not the provider" ) ;
address _receiver = tokenSeedToMinter [ seed ] ; uint256 index =
uint256 ( result )
%
( maxSupply - mintResolveCount ) ;
uint256 tokenID = tokenMintingLogs [ index ] ; if
( tokenID ==

```



```
0 )  
{ tokenID = index ; }  
mintResolveCount ++ ; tokenMintingLogs [ index ]  
= maxSupply - mintResolveCount ;  
_safeMint ( _receiver , tokenID ) ; } } We have deployed the reference contracts to the Goerli testnet here.
```

Contract Address Bridge [0xD291A502e3ca4Bb13E09892e57d8Ff0271Bd198A](#) VRFProvider
[0xF1F3554b6f46D8f172c89836FBeD1ea8551eabad](#) VRFLens [0x6e876b4Ed458af275Eb049a3f89BF0909618d154](#)
NFTBatchMinting [0x0b590C537608d121F8e46c2b366f5d22EC942c0f](#) [Previous VRF integration](#) [Next VRF Supported Blockchains](#)