New Blobstream X deployments

This document will go over the instructions to deploy BlobstreamX to a new chain.

Deploying the contracts

To deploy a Blobstream X to a new chain, where a Gateway contract does not exist yet, the following steps need to be followed.

If any of the components already exist in the target chain, feel free to skip the corresponding step.

Deploy a new SuccinctGateway

TheSuccinctGateway is a contract that acts as a registry for onchain circuit verifiers and manages their access control. It is the entrypoint for proof verification and does the following:

1. Receive a PLONK proof from the prover
2. Fetch the address of the target function verifier
3. Verify if the prover is whitelisted, if whitelisting is enabled
4. Forward the proof to the function verifier to be verified
5. If the proof is valid, it calls back theBlobstreamX
6. contract to update its state

TheBlobstreamX requires the update to be provided through theSuccinctGateway . Otherwise, the contract can't be updated.

To deploy aSuccinctGateway contract, you need to havefoundry installed. If not, refer tofoundry documentation .

Then, clone the succinctx repo:

shell git

clone

https://github.com/succinctlabs/succinctx cd

succinctx git

clone

https://github.com/succinctlabs/succinctx cd

succinctx Next, build the contracts:

shell cd

contracts forge

build cd

contracts forge

build Then, setup the.env containing the required information for deployment. An example.env.example is provided to be inspired from.

Assume we're deploying to a chain which chainID is 12345. The.env should look like:

shell

The salt to be used for CREATE2 deployments. It's a 32-byte string in hex format.

Example: 0x0000000000000000000000000000000000000000000000000000000000000001

CREATE2_SALT

The 'owner' of the contracts, recommended to be an EOA

GUARDIAN

The default prover to fulfill requests for Functions that have not opted for a different prover

PROVER

RPC URLs for each chain you want to deploy to

RPC_12345

Etherscan API keys for each chain you want to deploy to

ETHERSCAN_API_KEY_12345

Wallet type is the type of the wallet.

If you are using a ledger to sign the transaction,

set WALLET_TYPE=LEDGER and specify the MNEMONIC_INDEX

for which signer you want to use.

If you are using a private key,

set WALLET_TYPE=PRIVATE_KEY and specify the PRIVATE_KEY.

In this example, we're using private key.

WALLET_TYPE

PRIVATE_KEY

The private key of the deployer account

PRIVATE_KEY

The address of the succinct fee vault contract. Set it to an example address

if you don't want to use a vault like:
0x0000000000000000000000000000000000000001

SUCCINCT_FEE_VAULT_12345

The salt to be used for CREATE2 deployments. It's a 32-byte string in hex format.

Example:
0x0000000000000000000000000000000000000000000000000000000000000001

CREATE2_SALT

The 'owner' of the contracts, recommended to be an EOA

GUARDIAN

The default prover to fulfill requests for Functions that have not opted for a different prover

PROVER

RPC URLs for each chain you want to deploy to

RPC_12345

Etherscan API keys for each chain you want to deploy to

ETHERSCAN_API_KEY_12345

Wallet type is the type of the wallet.

If you are using a ledger to sign the transaction,

set WALLET_TYPE=LEDGER and specify the MNEMONIC_INDEX

for which signer you want to use.

If you are using a private key,

set WALLET_TYPE=PRIVATE_KEY and specify the PRIVATE_KEY.

In this example, we're using private key.

WALLET_TYPE

PRIVATE_KEY

The private key of the deployer account

PRIVATE_KEY

The address of the succinct fee vault contract. Set it to an example address

if you don't want to use a vault like: 0x0000000000000000000000000000000000000001

SUCCINCT_FEE_VAULT_12345

- PRIVATE_KEY
- : the private key of the account used to send the transaction.
- CREATE2_SALT
- : the salt used to generate the gateway address. Using the same salt between deployments in different chains will make the gateways be on the same address.
- GUARDIAN
- : the owner of theSuccinctGateway
- contract. It should be set to an account that will have full access to the gateway.
- PROVER
- : the address of the account used to submit the proofs to the gateway. It is enabled by default for all the registered function verifiers.
- RPC_12345
- : the RPC endpoint for the EVM chain whose chain ID is 12345. If the chain ID is different, make sure to change it in the environment variable name as well.
- ETHERSCAN_API_KEY_12345
- : the Etherscan API key corresponding to the chain whose chain ID is 12345. Similar toRPC_12345
- , make sure to change the chain ID in the environment variable name if it's a different chain.

Then, save the environment to a.env file and run the following:

shell ./script/deploy.sh

"SuccinctGateway"

"12345 1234" ./script/deploy.sh

"SuccinctGateway"

"12345 1234" with12345 and1234 being chainID that we want to deploy to, given that the corresponding environment variables, as specified above, are setup correctly.

Now the SuccinctGateway address should be printed on the terminal.

Deploy the function verifiers

The function verifiers are the onchain circuit verifiers that take PLONK proofs as an input and verify them onchain.

The function verifiers can be either downloaded from the succinct platform or regenerated.

To download the function verifiers, along with the circuits binaries, check the list in theSuccinct documentation and download the binaries and function verifiers corresponding to the circuits you want to verify.

Alternatively, you can generate them locally as specified in the regenerating the downloaded artifacts section.

After getting theFunctionVerifier.sol , either via downloading it or generating it locally, you can choose your favourite way to deploy it.

A simple way would be to copy the contract in the BlobstreamX repo and deploy it:

shell cd HOME git

clone

https://github.com/succinctlabs/blobstreamx cd

blobstreamx/contracts

forge

install

copy the function verifier contract to the blobstream X contracts

cp

< path_to_function_verifie r

/FunctionVerifier.sol

src

forge

build

forge

create

FunctionVerifier

--rpc-url

< rpc_ur l

--private-key

< private_ke y

--verify

--verifier

etherscan

--etherscan-api-key

< etherscan_api_ke y

^ will return the address of the function verifier

cd HOME git

clone

https://github.com/succinctlabs/blobstreamx cd

blobstreamx/contracts

forge

install

copy the function verifier contract to the blobstream X contracts

cp

< path_to_function_verifie r

/FunctionVerifier.sol

src

forge

build

forge

create

FunctionVerifier

--rpc-url

< rpc_ur l

--private-key

< private_ke y

--verify

--verifier

etherscan

--etherscan-api-key

< etherscan_api_ke y

^ will return the address of the function verifier

Register the function verifier in the deployed SuccinctGateway

After deploying a function verifier for your target circuit, we should register it in aSuccinctGateway contract, either deployed by you in a previous step or pre-existing on the chain, to get afunctionID and be able to use it to verify circuits.

A simple way to do it is to usecast tool from the Foundry toolset:

First, we will get the expected functionID by callingcast call which simulates the transaction execution and returns the return value:

shell cast

call

< succinct_gateway_contract_addres s

"registerFunction(address,address,bytes32)(bytes32)"

--rpc-url

< rpc_ur l

< owner_of_the_function_verifie r

< address_of_the_function_verifie r

< 32_bytes_of_sal t

--private-key

< private_ke y

cast

call

< succinct_gateway_contract_addres s

"registerFunction(address,address,bytes32)(bytes32)"

--rpc-url

< rpc_ur l

< owner_of_the_function_verifie r

< address_of_the_function_verifie r

< 32_bytes_of_sal t

--private-key

< private_ke y

This will return abytes32 which is the function ID.

Now, we will execute the transaction to register that returned functionID in the gateway, using the same parameters:

shell cast

send

< succinct_gateway_contract_addres s

"registerFunction(address,address,bytes32)(bytes32)"

--rpc-url

< rpc_ur l

< owner_of_the_function_verifie r

< address_of_the_function_verifie r

< 32_bytes_of_sal t

--private-key

< private_ke y

cast

send

< succinct_gateway_contract_addres s

"registerFunction(address,address,bytes32)(bytes32)"

--rpc-url

< rpc_ur l

< owner_of_the_function_verifie r

< address_of_the_function_verifie r

< 32_bytes_of_sal t

--private-key

< private_ke y

To verify that the registration was successful, run:

shell cast

call

< succinct_gateway_contract_addres s

"verifiers(bytes32)(address)"

--rpc-url

< rpc_ur l

< returned_function_I D

cast

call

< succinct_gateway_contract_addres s

"verifiers(bytes32)(address)"

--rpc-url

< rpc_ur l

< returned_function_I D

which will return the address of the function verifier that was deployed in the previous section.

NOTE: For BlobstreamX, there are always two function verifiers, corresponding to the two circuits:header_range andnext_header . Make sure to register both the function verifiers corresponding to those circuits as we will use those function IDs to deploy the BlobstreamX contract in the next section.

Enable prover whitelisting

Now that the function verifier's contract is deployed and registered in the succinct gateway, we can define whitelisting rules for the proof submission.

by default, the whitelist status is set toDefault . This means that only the default verifier, which was setup whendeploying the SuccinctGateway . And if you want to restrict the list of provers that can submit proofs to your registered function verifier, you can set the whitelisting status of the function verifier and then add a custom prover. Or even allow for permissionless submission.

Set Whitelist Status

Set the whitelist status of a functionID to:

- 0
- : Default status, which only allows the default prover to submit proofs
- 1
- : Custom status, which only allows a custom list of provers to submit the proofs. The custom list can be setup by calling theaddCustomProver(bytes32 _functionId, address _prover)
- function.
- 2
- : Disabled status, which allow for permissionless proof submission, i.e., any relayer can submit proofs to be verified by that function verifier.

shell cast

calldata

"setWhitelistStatus(bytes32,uint8)"

< YOUR_FUNCTION_I D

< WHITELIST_STATU S

cast

calldata

"setWhitelistStatus(bytes32,uint8)"

< YOUR_FUNCTION_I D

< WHITELIST_STATU S

Add Custom Prover

Add a custom prover for a specific functionID.

shell cast

calldata

"addCustomProver(bytes32,address)"

< FUNCTION_I D

< CUSTOM_PROVER_ADDRES S

cast

calldata

"addCustomProver(bytes32,address)"

< FUNCTION_I D

< CUSTOM_PROVER_ADDRES S

Deploy the BlobstreamX contract

To deploy aBlobstreamX contract, you need to havefoundry installed. If not, refer tofoundry documentation .

Then, clone the repository:

shell cd HOME git

clone

https://github.com/succinctlabs/blobstreamx cd

blobstreamx cd HOME git

clone

https://github.com/succinctlabs/blobstreamx cd

blobstreamx Deploying the BlobstreamX contract requires initializing it with a trusted header.

In case we want to run a local prover, as is documented in therun a local prover section, the trusted header can be any header in the last three weeks of blocks, aka the unbonding period. Using an older header is also possible, but in case of a malicious RPC provider, the malicious validators won't be slashed.

Otherwise, we can run a proof replayer that will read the proofs from an existing deployment and submit them into the new one. This is documented under therun a proof replayer from an existing deployment section. In this case, the trusted header should correspond to a height already submitted to the existing deployment, and was the beginning of a batch. These heights can be found via querying theDataCommitmentStored events, or navigating to theevents tab in Etherscan for the existing deployment, and choosing astartBlock from a certain event.

Querying the trusted hash

Given that we have a trusted height, we can query its corresponding trusted hash using:

shell TENDERMINT_RPC_URL =< celestia_rpc_addres s

cargo

run

--bin

genesis

--

--block

< trusted_heigh t

TENDERMINT_RPC_URL =< celestia_rpc_addres s

cargo

run

--bin

genesis

--

--block

< trusted_heigh t

Alternatively, the trusted hash can be obtained via querying theblock?height= and getting theresult.block_id.hash field.

Deployment instructions

Once we have the trusted hash, we can deploy the BlobstreamX contract. To do so, create a.env file containing the necessary variables. An example one can be found undercontracts/.env.example :

shell cd

contracts cp

.env.example

.env vim

.env

and fill the necessary environment variables

cd

contracts cp

.env.example

.env vim

.env

and fill the necessary environment variables

The needed environment variables are:

- PRIVATE_KEY
- : the EVM private key to use to deploy the contract.
- RPC_URL
- : the RPC endpoint of the chain where we want to deploy the BlobstreamX contract.
- ETHERSCAN_API_KEY
- : the Etherscan API for the target chain.
- CREATE2_SALT
- : salt for theCREATE2
- method. Example:0xaa
- .
- GUARDIAN_ADDRESS
- : the BlobstreamX contract's owner. It allows it to update the functionIDs of the deployment, freeze the contract, upgrade it, and others.
- GATEWAY_ADDRESS
- : the address of the SuccinctGateway deployment that will update the contract's state. As stated in thesection above
- , the SuccinctGateway is the entrypoint that routes the proof to the function verifier, and if the proof is valid, it updates the BlobstreamX contract.
- NEXT_HEADER_FUNCTION_ID
- : the functionID of the next header circuit verifier. It is obtained after deploying a function verifier, corresponding to the next header circuit, and registering it in the SuccinctGateway. More information in thefunction verifier registration section
- .
- HEADER_RANGE_FUNCTION_ID
- : similar toNEXT_HEADER_FUNCTION_ID
- but for the header range circuit.
- GENESIS_HEIGHT
- : is the height of the trusted header that we queried its trusted hash in thequerying the trusted hash section
- . If the value starts with0x
- , it will be parsed as a hex value. Otherwise, it will be treated as a decimal number.
- GENESIS_HEADER
- : the queried trusted hash in thequerying the trusted hash section
- .
- DEPLOY
- : set to true since we're deploying the contract.
- UPGRADE
- : set to false since we're not upgrading an existing deployment.
- UPDATE_GENESIS_STATE
- : set to false since we're not updating the genesis state.
- UPDATE_FUNCTION_IDS
- : set to false since we're not updating the function IDs.
- UPDATE_GATEWAY
- : set to false since we're not updating the address of the gateway.
- CONTRACT_ADDRESS
- : set to an empty value since we're not upgrading the deployment.

Save the.env file then run:

shell forge

install

source

.env

forge

script

script/Deploy.s.sol

--rpc-url RPC_URL --private-key PRIVATE_KEY --broadcast

--verify

--verifier

etherscan

--etherscan-api-key ETHERSCAN_API_KEY forge

install

source

.env

forge

script

script/Deploy.s.sol

--rpc-url RPC_URL --private-key PRIVATE_KEY --broadcast

--verify

--verifier

etherscan

--etherscan-api-key ETHERSCAN_API_KEY And you should see the address printed in the logs.

Run a local prover

Now that the BlobstreamX contract is deployed, we can either run a local prover or opt for proofs replaying .

For running a local prover, make sure to have a beefy machine. It is recommended to run the prover in at least a 32CPU 256GB RAM machine to be able to generate proofs in time for a 1hr proofs submission frequency.

To run the prover:

shell cd HOME

in case you still did not clone the repository:

git clone https://github.com/succinctlabs/blobstreamx

cd

blobstreamx

create the environment for the prover

cp

.env.example

.env

edit the environment file

vim

.env cd HOME

in case you still did not clone the repository:

git clone https://github.com/succinctlabs/blobstreamx

cd

blobstreamx

create the environment for the prover

cp

.env.example

.env

edit the environment file

vim

.env The following is the required environment for the prover:

- PRIVATE_KEY
- : the EVM private key to use to submit the proofs.
- RPC_URL
- : the RPC endpoint of the chain where the BlobstreamX contract is deployed.
- TENDERMINT_RPC_URL
- : the Celestia chain RPC endpoint. Accepts a comma-separated list of RPC URLs for fail over.
- CHAIN_ID
- : the target EVM chain ID.
- CONTRACT_ADDRESS
- : the target BlobstreamX contract address.
- NEXT_HEADER_FUNCTION_ID
- : the function ID of the next header function registered in the succinct gateway.
- HEADER_RANGE_FUNCTION_ID
- : the function ID of the header range function registered in the succinct gateway.
- LOOP_DELAY_MINS
- : the time to wait before sending the proofs in minutes. For example, for having a proof every 1hr, set it to60
- .
- LOCAL_PROVE_MODE

- : set to true to enable local proving, which is what these instructions are about.
- LOCAL_RELAY_MODE
- : set to true to enable submitting the proofs onchain, which is what these instructions are about.
- GATEWAY_ADDRESS
- : the address of the succinct gateway contract in the target chain.

For the circuits binaries, you can either generate them as specified in the build the circuits section , or download them from the Succinct platform . And then set the following environment variables:

- PROVE_BINARY_0xFILL_IN_NEXT_HEADER_FUNCTION_ID
- : the next header circuit binary path. Make sure to also change the0xFILL_IN_NEXT_HEADER_FUNCTION_ID
- in the environment variable name to the functionID of the corresponding function verifier.
- PROVE_BINARY_0xFILL_IN_HEADER_RANGE_FUNCTION_ID
- : the header range circuit binary path. Make sure to also change the0xFILL_IN_HEADER_RANGE_FUNCTION_ID
- in the environment variable name to the functionID of the corresponding function verifier.

Similarly, the verifier build can be generated as specified in the regeneration of the verifier build section , or downloaded from the succinct platform . And then set the following environment variable:

- WRAPPER_BINARY
- : the path the wrapper verifier

After setting the environment, run:

shell source

.env cargo

run

--bin

blobstreamx

--release source

.env cargo

run

--bin

blobstreamx

--release And you should see the operator's logs.

Run a proof replayer from an existing deployment

A cheaper alternative to running the prover locally is to deploy the BlobstreamX contract at a height that is followed by an existing BlobstreamX contract. Then, keep replaying the proofs from the existing deployment to the new one.

Check the blobstream-ops repository for more documentation.

Optional: Regenerating the downloaded artifacts

In case you want to generateverifier-build , which is the template solidity contract used to verify the PLONK proofs onchain, or the circuits build, which are used to generate the proofs, follow this section.

Otherwise, use the downloaded ones as specified in the previous section.

Regenerate theverifier-build

Theverifier-build is the gnark wrapper circuit used by the plonky2x circuits for cheap on-chain verification. It outputs a PLONK proof.

It can be built using the following instructions:

shell cd HOME git

clone

https://github.com/succinctlabs/succinctx cd

succinctx cargo

test

test_wrapper

^ This should generate a data folder under plonky2x/verifier/data

Compile circuit

cd

plonky2x/verifier mkdir

verifier-build go

run

.

-compile

-data

verifier-build

-circuit

data/dummy

Compile executable

CGO_ENABLED

0

go

build

-o

verifier-build/verifier

-ldflags

"-s -w"

.

ls

verifier-build/

^ should have: pk.bin r1cs.bin verifier Verifier.sol vk.bin

cd HOME git

clone

https://github.com/succinctlabs/succinctx cd

succinctx cargo

test

test_wrapper

^ This should generate a data folder under plonky2x/verifier/data

Compile circuit

cd

plonky2x/verifier mkdir

verifier-build go

run

.

-compile

-data

verifier-build

-circuit

data/dummy

Compile executable

CGO_ENABLED

0

go

build

-o

verifier-build/verifier

-ldflags

"-s -w"

.

ls

verifier-build/

^ should have: pk.bin r1cs.bin verifier Verifier.sol vk.bin

This should generate theVerifier.sol contract which is a template contract used to generate the final function verifiers based on the plonky2x circuits.

In the generated contract, only the circuit digest is specific to each circuit; the rest of the logic is universal. This is why a random circuit was used to create the template, indicated by the flag-circuit data/dummy . When we build a specific circuit, theVerifier.sol contract is fed to the circuit builder, which then replaces the circuit digest with the one corresponding to the actual circuit being built.

Also, you will see theverifier binary which is the PLONK wrapper that takes a plonky2x proof and wraps it into a PLONK proof.

During the build, the CRS string gets downloaded:

bash 2024/04/15

19 :35:00

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript04.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript04.dat 2024/04/15

19 :35:05

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript05.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript05.dat 2024/04/15

19 :35:11

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript06.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript06.dat 2024/04/15

19 :35:41

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript11.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript11.dat 2024/04/15

19 :35:47

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript12.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript12.dat 2024/04/15

19 :35:53

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript13.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript13.dat 2024/04/15

19 :35:58

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript14.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript14.dat 2024/04/15

19 :36:03

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript15.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript15.dat 2024/04/15

19 :35:00

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript04.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript04.dat 2024/04/15

19 :35:05

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript05.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript05.dat 2024/04/15

19 :35:11

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript06.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript06.dat 2024/04/15

19 :35:41

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript11.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript11.dat 2024/04/15

19 :35:47

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript12.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript12.dat 2024/04/15

19 :35:53

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript13.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript13.dat 2024/04/15

19 :35:58

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript14.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript14.dat 2024/04/15

19 :36:03

download

https://aztec-ignition.s3.amazonaws.com/MAIN%20IGNITION/196_0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript15.dat

196 _0x9c3f372a2aacc0d7272f56a4664311b7647b0031/transcript15.dat This randomness is taken from the Aztec trusted setup .

NOTE: In a scaleway instance of 64CPU and 504G RAM, the build takes ~20 minutes to complete. Make sure to run it on a beefy machine.

Build the circuits and function verifiers

To build the circuits:

bash cd HOME git

clone

https://github.com/succinctlabs/blobstreamx cd

blobstreamx

if a specific version is needed: git checkout

this assumes the verifier-build was generated in the previous section.

if not, it can be downloaded as specified in the first section.

cp

-r

../succinctx/plonky2x/verifier/verifier-build

.

^ This copies the verifier build to the blobstreamx folder

mkdir

-p

build && RUST_LOG = debug

cargo

run

--bin

header_range_1024

--release

build

--wrapper-path

verifier-build/ && mv

./target/release/header_range_1024

./build/header_range_1024 ls

build

^ should return something like: 0x007d0b2a2e2b013612e8.circuit 0x9039e58b2089e5f9abbb.circuit 0xce1636cfaf2bd5497c11.circuit FunctionVerifier.sol main.circuit 0x8e1ede4ce0865b41d714.circuit 0xa2140c9bde000dc5e21e.circuit 0xf6759ff933786ddacb92.circuit header_range_1024

cd HOME git

clone

https://github.com/succinctlabs/blobstreamx cd

blobstreamx

if a specific version is needed: git checkout

this assumes the verifier-build was generated in the previous section.

if not, it can be downloaded as specified in the first section.

cp

-r

../succinctx/plonky2x/verifier/verifier-build

.

^ This copies the verifier build to the blobstreamx folder

mkdir

-p

build && RUST_LOG = debug

cargo

run

--bin

header_range_1024

--release

build

--wrapper-path

verifier-build/ && mv

./target/release/header_range_1024

./build/header_range_1024 ls

build

^ should return something like: 0x007d0b2a2e2b013612e8.circuit 0x9039e58b2089e5f9abbb.circuit 0xce1636cfaf2bd5497c11.circuit

FunctionVerifier.sol main.circuit 0x8e1ede4ce0865b41d714.circuit 0xa2140c9bde000dc5e21e.circuit 0xf6759ff933786ddacb92.circuit header_range_1024

Theheader_range_1024 is a specific circuit. Other circuit names can be used there. The current circuits that we have for BlobstreamX:

- header_range_1024
- : skip function circuit for batches that are <= 1024 block.
- header_range_2048
- : skip function circuit for batches that are <= 2048 block.
- next_header
- : step function circuit.

All the deployments currently rely on two circuits: a header range circuit, either the 1024 or the 2048, depending on the frequency of the batches; The 1024 is mainly for batches that are at a ~1hr frequency and the 2048 for batches that are at a ~3-4hr frequency; and a next header circuit. So, if you're re-building the circuits, make sure to build the correct two circuits for your target deployment.

Now, if you check thebuild folder, you will find a file calledFunctionVerifier.sol , which is the function verifier contract for your circuit that you can deploy on-chain.

Also, you will find theheader_range_1024 underbuild folder which is the binary used to generate the proofs. It is mainly used by the operator to generate the plonky2x proofs that will be PLONK wrapped later using the generatedverifier-build .

At this level, you can deploy theFunctionVerifier.sol onchain, then register it in the SuccinctGateway, then use the generatedheader_range_1024 circuit and theverifier to generate the proofs and submit them onchain. These steps are detailed in the previous section.

NOTE: In a scaleway instance of 64CPU and 504G RAM, the build takes ~10 minutes to complete. Make sure to run it on a beefy machine. [][Edit this page on GitHub] Last updated: Previous page Requesting data commitment ranges Next page Celestia-node key []