Adding Attributes to the Derivation Function

⚠ OP Stack Hacks are explicitly things that you can do with the OP Stack that arenot currently intended for production use.

OP Stack Hacks are not for the faint of heart. You will not be able to receive significant developer support for OP Stack Hacks — be prepared to get your hands dirty and to work without support.

Overview

In this tutorial, you'll modify the Bedrock Rollup. Although there are many ways to modify the OP Stack, you're going to spend this tutorial modifying the Derivation function. Specifically, you're going to update the Derivation function to track the amount of ETH being burned on L1! Who's gonna tellultrasound.money(opens in a new tab) that they should replace their backend with an OP Stack chain?

Getting the idea

Let's quickly recap what you're about to do. Theop-node is responsible for generating the Engine API payloads that triggerop-geth to produce blocks and transactions. Theop-node already generates a "system transaction"for every L1 block that relays information about the current L1 state to the L2 chain. You're going to modify theop-node to add a new system transaction that reports the total burn amount (the base fee multiplied by the gas used) in each block.

Although it might sound like a lot, the whole process only involves deploying a single smart contract, adding one new file toop-node , and modifying one existing file insideop-node . It'll be painless. Let's go!

Deploy the burn contract

You're going to use a smart contract on your Rollup to store the reports that theop-node makes about the L1 burn. Here's the code for your smart contract:

// SPDX-License-Identifier: MIT pragma

solidity ^0.8.0;

/ *** @title L1Burn * @notice L1Burn keeps track of the total amount of ETH burned on L1. */ contract L1Burn {** ** @notice Total amount of ETH burned on L1. */* uint256

public total;

/ * @notice Mapping of blocks numbers to total burn*/ mapping (uint64

=>

uint256) public reports;

/ * @notice Allows the system address to submit a report. * * @param _blocknum L1 block number the report corresponds to. * @param _burn Amount of ETH burned in the block.* / function

report (uint64

_blocknum ,

uint64

_burn) external { require (msg.sender ==

0xDeaDDEaDDeAdDeAdDEAdDEaddeAddEAdDEAd0001 , "L1Burn: reports can only be made from system address");

total += _burn; reports[_blocknum] = total; }

/ * @notice Tallies up the total burn since a given block number. * * @param _blocknum L1 block number to tally from. * * @return Total amount of ETH burned since the given block number;* / function

tally (uint64

_blocknum) external

view

returns (uint256) { return total - reports[_blocknum]; } } Deploy this smart contract to your L2 (using any tool you find convenient). Make a note of the address that the contract is deployed to because you'll need it in a minute. Simple!

Add the burn transaction

Now you need to add logic to theop-node to automatically submit a burn report whenever an L1 block is produced. Since this transaction is very similar to the system transaction that reports other L1 block info (found inl1_block_info.go(opens in a new tab)), you'll use that transaction as a jumping-off point.

Navigate to theop-node

package:

cd

~/optimism/op-node

Inside of the folderrollup/derive

, create a new file calledl1_burn_info.go :

touch

rollup/derive/l1_burn_info.go

Paste the following intol1_burn_info.go

, and make sure to replaceYOUR_BURN_CONTRACT_HERE with the address of theL1Burn contract you just deployed.

package derive

import ("bytes" "encoding/binary" "fmt" "math/big"

"github.com/ethereum/go-ethereum/common" "github.com/ethereum/go-ethereum/core/types" "github.com/ethereum/go-ethereum/crypto"

"github.com/ethereum-optimism/optimism/op-node/eth")

const (L1BurnFuncSignature =

"report(uint64,uint64)" L1BurnArguments =

2 L1BurnLen =

4

+

32 * L1BurnArguments)

var (L1BurnFuncBytes4 = crypto. Keccak256 ([] byte (L1BurnFuncSignature))[: 4] L1BurnAddress = common. HexToAddress ("YOUR_BURN_CONTRACT_HERE"))

type

L1BurnInfo

struct { Number uint64 Burn uint64 }

func (info * L1BurnInfo) MarshalBinary () ([] byte , error) { data :=

make ([] byte , L1BurnLen) offset :=

0 copy (data[offset: 4], L1BurnFuncBytes4) offset +=

4 binary.BigEndian. PutUint64 (data[offset + 24 :offset + 32], info.Number) offset +=

32 binary.BigEndian. PutUint64 (data[offset + 24 :offset + 32], info.Burn) return data, nil }

func (info * L1BurnInfo) UnmarshalBinary (data [] byte) error { if

len (data) != L1InfoLen { return fmt. Errorf ("data is unexpected length: %d " , len (data)) } var padding [24] byte offset :=

4 info.Number = binary.BigEndian. Uint64 (data[offset + 24 : offset + 32]) if

! bytes. Equal (data[offset:offset + 24], padding[:]) { return fmt. Errorf ("l1 burn tx number exceeds uint64 bounds: %x " , data[offset:offset + 32]) } offset +=

32 info.Burn = binary.BigEndian. Uint64 (data[offset + 24 : offset + 32]) if

! bytes. Equal (data[offset:offset + 24], padding[:]) { return fmt. Errorf ("l1 burn tx burn exceeds uint64 bounds: %x " , data[offset:offset + 32]) } return

nil }

func

L1BurnDepositTxData (data [] byte) (L1BurnInfo, error) { var info L1BurnInfo err := info. UnmarshalBinary (data) return info, err }

func

L1BurnDeposit (seqNumber uint64 , block eth.BlockInfo, sysCfg eth.SystemConfig) (* types.DepositTx, error) { infoDat := L1BurnInfo{ Number: block. NumberU64 (), Burn: block. BaseFee (). Uint64 () * block. GasUsed (), } data, err := infoDat. MarshalBinary () if err !=

nil { return

nil , err } source := L1InfoDepositSource{ L1BlockHash: block. Hash (), SeqNumber: seqNumber, } return

& types.DepositTx{ SourceHash: source. SourceHash (), From: L1InfoDepositerAddress, To: & L1BurnAddress, Mint: nil , Value: big. NewInt (0), Gas: 150 _ 000 _ 000 , IsSystemTransaction: true , Data: data, }, nil }

func

L1BurnDepositBytes (seqNumber uint64 , l1Info eth.BlockInfo, sysCfg eth.SystemConfig) ([] byte , error) { dep, err :=

L1BurnDeposit (seqNumber, l1Info, sysCfg) if err !=

nil { return

nil , fmt. Errorf ("failed to create L1 burn tx: %w " , err) } l1Tx := types. NewTx (dep) opaqueL1Tx, err := l1Tx. MarshalBinary () if err !=

nil { return

nil , fmt. Errorf ("failed to encode L1 burn tx: %w " , err) } return opaqueL1Tx, nil } Feel free to take a look at this file if you're interested. It's relatively simple, mainly just defining a new transaction type and describing how the transaction should be encoded.

Insert the burn transactions

Finally, you'll need to update~/optimism/op-node/rollup/derive/attributes.go to insert the new burn transaction into every block. You'll need to make the following changes:

Find these lines:

l1InfoTx, err :=

L1InfoDepositBytes (seqNumber, l1Info, sysConfig) if err !=

nil { return

nil , NewCriticalError (fmt. Errorf ("failed to create l1InfoTx: %w " , err)) }

After those lines, add this code fragment:

l1BurnTx, err :=

L1BurnDepositBytes (seqNumber, l1Info, sysConfig) if err !=

nil { return

nil , NewCriticalError (fmt. Errorf ("failed to create l1InfoTx: %w " , err)) }

Immediately following, change these lines:

txs :=

make ([]hexutil.Bytes, 0 , 1 + len (depositTxs)) txs =

append (txs, l1InfoTx) to

txs :=

make ([]hexutil.Bytes, 0 , 2 + len (depositTxs)) txs =

append (txs, l1InfoTx) txs =

append (txs, l1BurnTx) All you're doing here is creating a new burn transaction after everyl1InfoTx and inserting it into every block.

Rebuild your op-node

Before you can see this change take effect, you'll need to rebuild yourop-node :

cd

~/optimism/op-node make

op-node Now start yourop-node if it isn't running or restart yourop-node if it's already running. You should see the change immediately — new blocks will contain two system transactions instead of just one!

Checking the result

Query thetotal function of your contract, you should also start to see the total slowly increasing. Play around with thetally function to grab the amount of gas burned since a given L2 block. You could use this to implement a version ofultrasound.money(opens in a new tab) that keeps track of things with an OP Stack as a backend.

One way to get the total is to run these commands:

export ETH_RPC_URL = http://localhost:8545 cast

call

< YOUR_BURN_CONTRACT_HER E

"total()"

|

cast

--from-wei

Conclusion

With just a few tiny changes to theop-node , you were just able to implement a change to the OP Stack that allows you to keep track of the L1 ETH burn on L2. With a live Cannon fault proof system, you should not only be able to track the L1 burn on L2, you should be able toprove the burn to contracts back on L1. That's crazy!

The OP Stack is an extremely powerful platform that allows you to perform a large amount of computation trustlessly. It's a superpower for smart contracts. Tracking the L1 burn is just one of the many, many wild things you can do with the OP Stack. If you're looking for inspiration or you want to see what others are building on the OP Stack, check out the OP Stack Hacks page. Maybe you'll find a project you want to work on, or maybe you'll get the inspiration you need to build the next killer smart contract.

Using the Optimism SDK Adding a Precompile