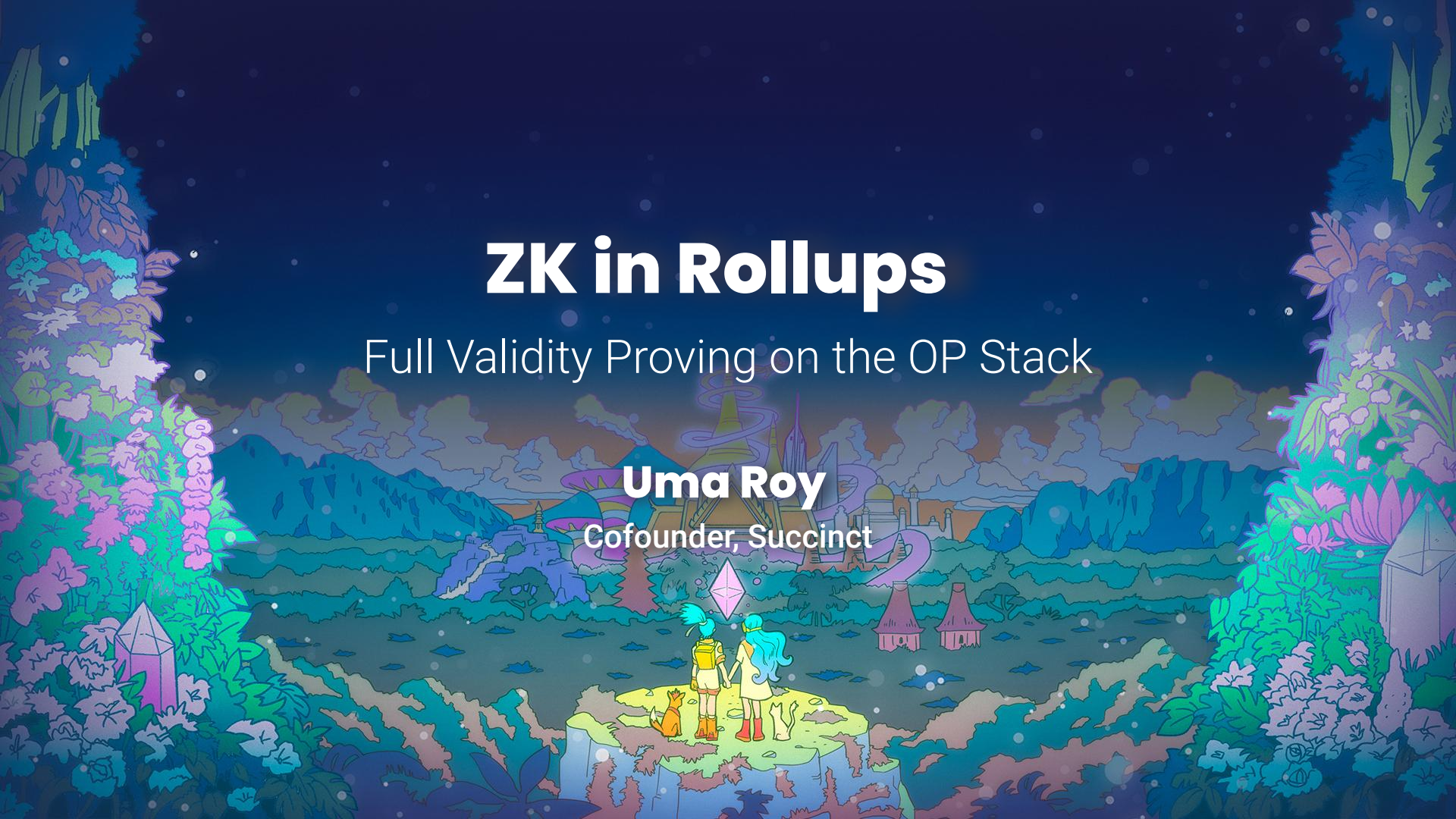


# ZK in Rollups

Full Validity Proving on the OP Stack

**Uma Roy**

Cofounder, Succinct



# ZK Rollups are the endgame

ZK is the only way that \_\_\_\_\_ will get solved.

- Fast finality for rollups
- Interoperability across Ethereum
- Unified liquidity for users
- Bridging across ecosystems
- Good UX

Decentralization imposes overhead by requiring redundant computation. ZKPs fix this.



# Until recently, ZK rollups have been challenging

zk\_evm / evm\_arithmetization / src / cpu / kernel / asm / main.asm

Code Blame 109 lines (93 loc) · 4.28 KB · 0

```

56 global start_txn:
57 // stack: (empty)
58 %load_global_metadata(@GLOBAL_METADATA_TXN_NUMBER_BEFORE)
59 // stack: txn_nb
60 DUP1 %scalar_to_rlp
61 // stack: txn_counter, txn_nb
62 DUP1 %num_bytes %mul_const(2)
63 // stack: num_nibbles, txn_counter, txn_nb
64 %increment_bounded_rlp
65 // stack: txn_counter, num_nibbles, next_txn_counter, next_num_nibbles, txn_nb
66 %load_global_metadata(@GLOBAL_METADATA_BLOCK_GAS_USED_BEFORE)
67
68 // stack: init_gas_used, txn_counter, num_nibbles, next_txn_counter, next_num_nibbles
69
70 // If the prover has no txn for us to process, halt.
71 PROVER_INPUT(no_txn)
72 %jump_if(execute_withdrawals)
73
74 // Call route_txn. When we return, we will process the txn receipt.
75 PUSH txn_after
76 // stack: retdest, prev_gas_used, txn_counter, num_nibbles, next_txn_counter, txn_after
77 DUP4 DUP4
78
79 %jump(route_txn)
80
81 global txn_after:
82 // stack: success, leftover_gas, cur_cum_gas, prev_txn_counter, prev_num_nibbles, txn_after
83 %process_receipt
84 // stack: new_cum_gas, txn_counter, num_nibbles, txn_after
85 SWAP3 %increment SWAP3
86 %jump_if(execute_withdrawals_post_stack_op)
87
88 global execute_withdrawals:
89 // stack: cum_gas, txn_counter, num_nibbles, next_txn_counter, next_num_nibbles
90 %stack (cum_gas, txn_counter, num_nibbles, next_txn_counter, next_num_nibbles)
91 execute_withdrawals_post_stack_op:
92 %withdrawals
93

```

•  $\chi$ -step:

$$A[x, y] = B[x, y] \oplus ((\text{NOT } B[x + 1, y]) \text{ AND } D[x])$$

•  $\iota$ -step:

$$A[0, 0] = A[0, 0] \oplus D[0]$$

In the above all the operations on the indices are done on a permutation state array and  $A[x, y]$  denotes a particular element in the state array. The symbol  $\oplus$  denotes the bitwise complement and AND the bitwise AND operation, and  $\circ$  denotes the bitwise cyclic (right) shift operation, modulo the lane size.

Cyclic shift offset constants  $r[x, y]$  are picked according to the following table:

$r(x, y)$	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	25	39	3	10	43
$y = 1$	55	20	36	44	6
$y = 0$	28	27	0	1	62
$y = 4$	56	14	18	2	61
$y = 3$	21	8	41	45	15

Compatibility

1

- \* Can verify an environment that looks exactly like Ethereum, and even the Ethereum chain itself
- \* Can scale the Ethereum L1, and not just rollups
- \* Maximally easy for rollups because you can share infrastructure (incl execution clients)
- \* Takes a very long time to generate proofs

2

- \* Can verify an environment that looks exactly like Ethereum, but with minor changes (eg. state tree) that don't touch the application layer
- \* Fully compatible with almost all Ethereum apps
- \* Can share most infrastructure
- \* Takes a long time to generate proofs

3

- \* Can verify an environment that is similar to Ethereum, but with minor changes (eg. hash function, no precompiles) that do touch the application layer
- \* Fully compatible with most Ethereum apps
- \* Can share a lot of infrastructure
- \* Faster to generate proofs

2.5

- \* Modify the EVM only by changing gas costs
- \* Makes it faster to generate proofs
- \* Introduces a few incompatibilities

4

- \* Compiles contracts written in Solidity, Vyper or other high-level langs to a specialized VM, and proves that
- \* Not compatible with some Ethereum apps
- \* Can't share a lot of infrastructure
- \* Fastest proof generation time, saves costs and reduces centralization risks

Performance



# What is SP1?

**SP1**: a blazing fast zkVM that enables any developer to create real-world ZKP applications by simply writing Rust.

SP1 Program

```
fn main() {  
  let n: u32 = zkvm::read();  
  let mut a = 0;  
  let mut b = 1;  
  for i in 0...n {  
    let c = a + b;  
    ...  
  }  
}
```

RISC-V (ELF)

```
addi    sp, sp, -16  
sw      ra, 12(sp)  
addi    s0, sp, 16  
sw      a0, -12(s0)  
mul      a0, a0, a0
```

SP1

```
sp1.prove(elf, stdin)  
  
* STARKs  
* FRI  
* Log Derivative  
* BabyBear Prime Field
```



Proof

```
program_hash("vkey hash")  
* input_bytes  
* output_bytes
```



## SP1 makes ZK rollups great (again)

	Before SP1	With SP1
Required expertise	✗ Specialized	✓ Little
Customization & maintenance burden & upgradeability	✗ Poor	✓ Excellent "cargo update reth"
Security surface area	✗ Large	✓ Leverages existing codebases + audits
EVM compatibility	✗ No	✓ Type 1
Expensive	✓ Not really	✓ Not really



**Step 1 to building a ZK rollup: ZKP execution of a block**



# Proving Ethereum blocks with SP1 + Reth

We wrote an SP1 program using Reth to execute individual Ethereum blocks.

1. **Execute block with “RPC DB” to fetch all relevant merkle proofs in “host”.**
2. Construct “ClientInput” with current block, previous block + merkle proofs
3. Execute inside client program
4. Generate proof in SP1



# Proving Ethereum blocks with SP1 + Reth

We write an SP1 program using Reth to execute individual Ethereum blocks

1. Execute block with “RPC DB” to fetch all relevant merkle proofs in “host”.
- 2. Construct “ClientInput” with current block, previous block + merkle proofs**
3. Execute inside client program
4. Generate proof in SP1





# Proving Ethereum blocks with SP1 + Reth

We write an SP1 program using Reth to execute individual Ethereum blocks

1. Execute block with “RPC DB” to fetch all relevant merkle proofs in “host”.
2. Construct “ClientInput” with current block, previous block + merkle proofs
- 3. Execute inside client program**
4. Generate proof in SP1



# Proving Ethereum blocks with SP1 + Reth

We write an SP1 program using Reth to execute individual Ethereum blocks

1. Execute block with “RPC DB” to fetch all relevant merkle proofs in “host”.
2. Construct “ClientInput” with current block, previous block + merkle proofs
3. Execute inside client program
4. **Generate proof in SP1**

**1139 LOC total:** <https://github.com/succinctlabs/rsp>



# SP1 + Reth Cost Estimates

From benchmarking data on cloud GPU

Block Number	Gas Used	Transaction Count	Number of Cycles	Cycles Per Transaction	Cost Per Transaction
20528720	13831834	150	682,513,352	4,550,089	\$0.0033
20528721	13182083	139	562,562,943	4,047,215	\$0.0029
20528722	25483756	349	1,004,427,115	2,878,014	\$0.0021
20528723	12057640	217	580,214,305	2,673,798	\$0.0020
20528724	12641380	167	610,973,878	3,658,526	\$0.0027
20528725	15256584	182	722,433,945	3,969,417	\$0.0029



# **SP1's novel ZK innovations + hardcore perf engineering enables low costs**

## **Precompile-centric architecture**

- keccak + secp256k1 + sha256 precompiles reduce cycle count by 6-10x
- bn254 and bls12-381 precompiles help with pairings + KZG point eval

## **Optimized GPU prover**

- Improves cost + latency by ~5x over CPU prover

## **Other algorithmic optimizations**

- "Memory in the head" argument that doesn't pay for merkleized memory
- Smaller blowup factor + multi-table batch FRI



# Takeaways

- Costs are **cheap** (currently \$0.001-0.003 proving costs per transaction)
- Easily customizable
- Minimal LOC
- Minimal maintenance surface area
- Easily upgradeable



**Step 2 to building a ZK rollup: everything else**



## How to draw an owl

1.



1. Draw some circles

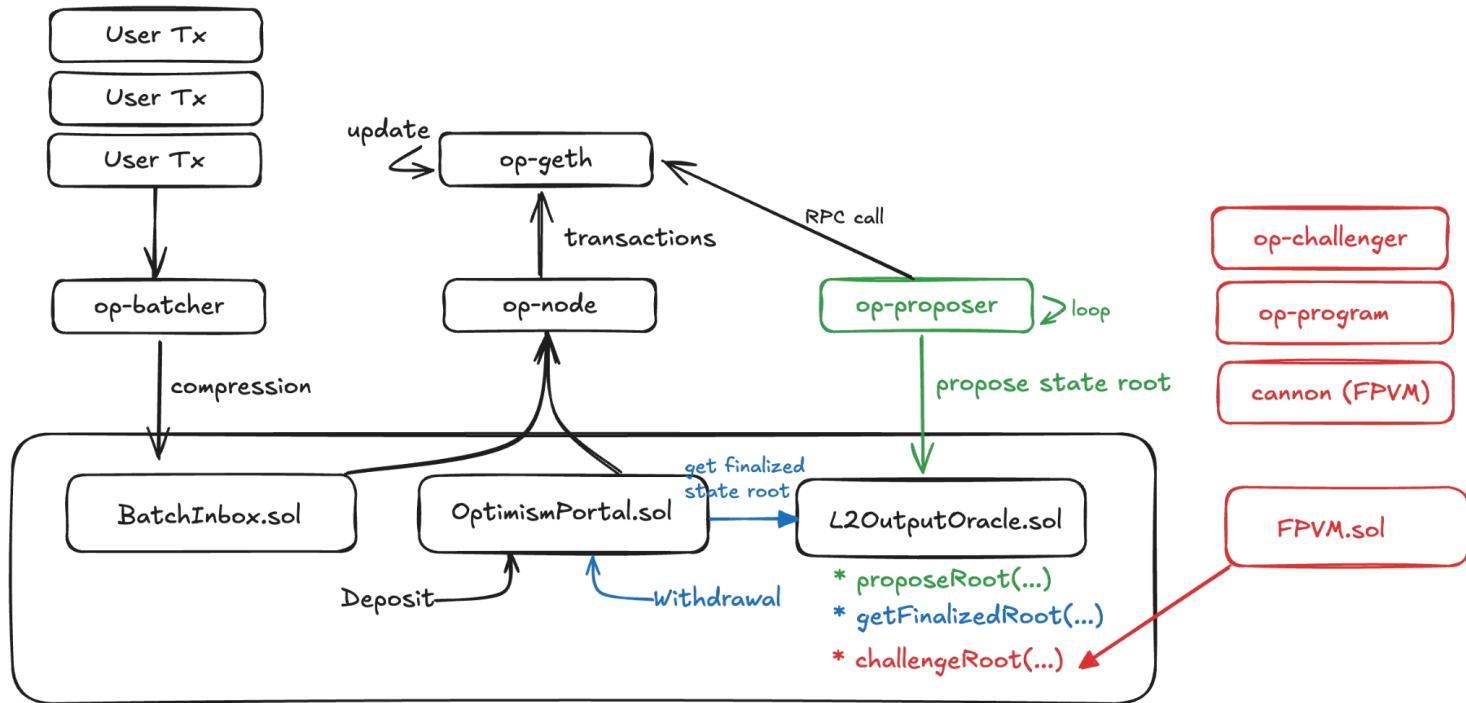
2.



2. Draw the rest of the fucking owl

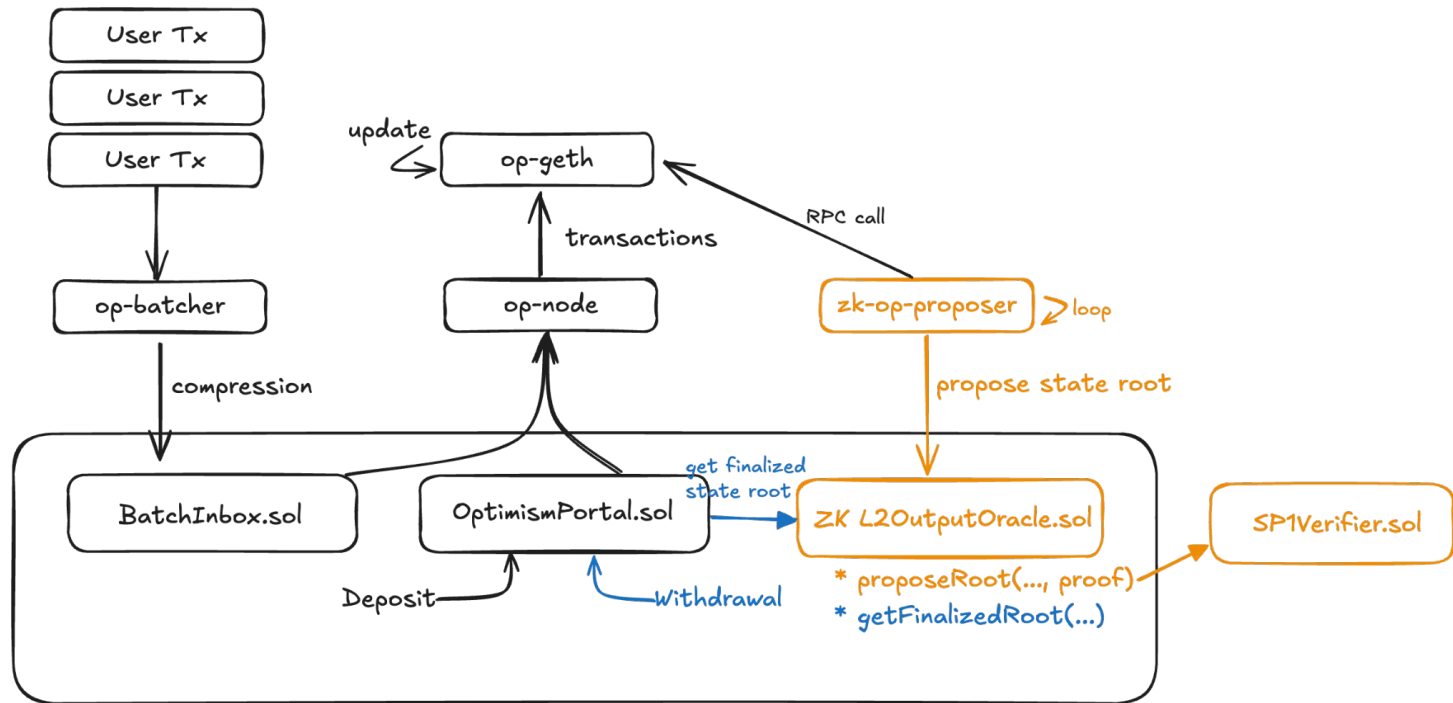


# OP Stack to the Rescue: How it Works

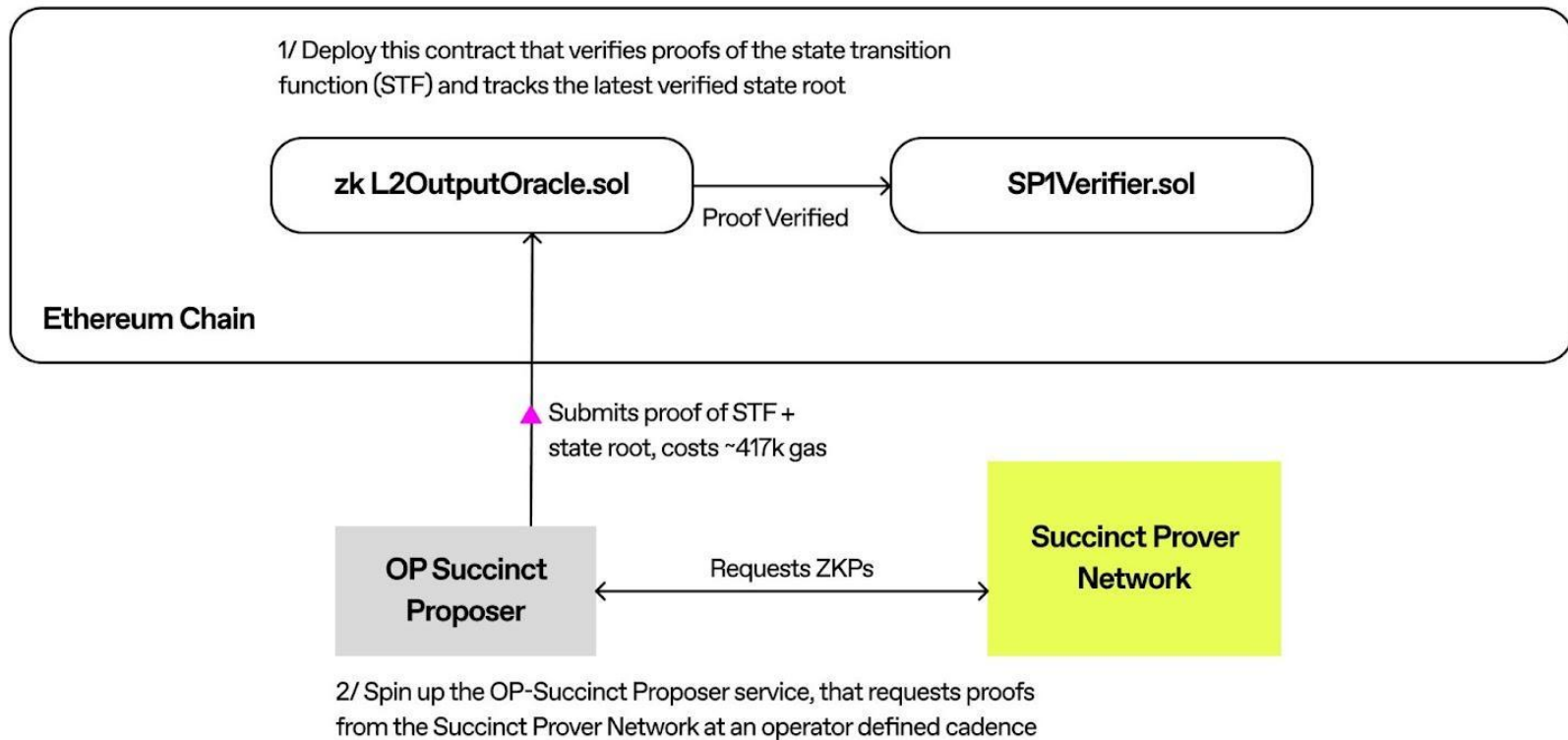




# OP Stack to the Rescue: How it Works



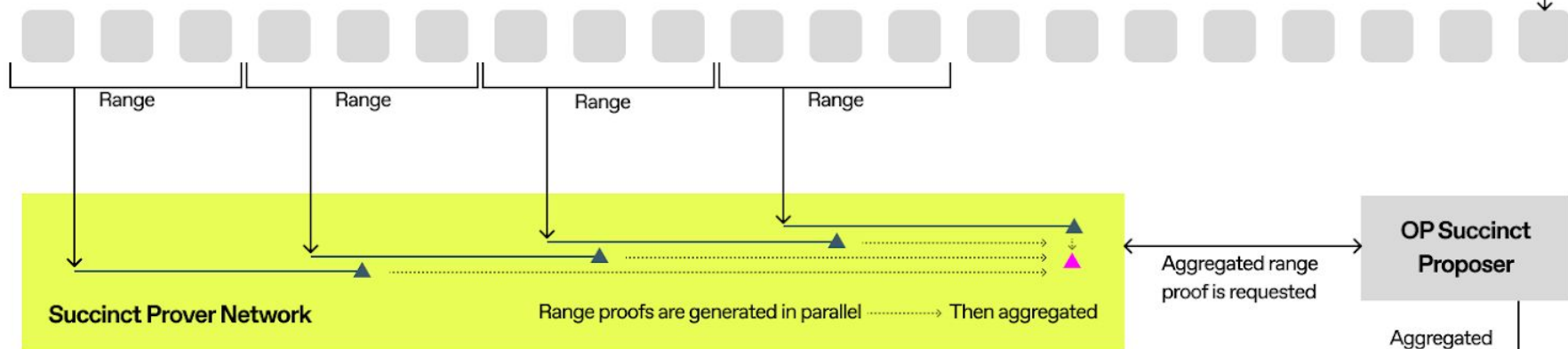
# Upgrading Your OP Stack to use SP1 in 2 Steps



# OP Succinct Proposer Service Architecture

1. Proofs are generated for a range of N blocks
2. These "range proofs" are generated in parallel
3. Range proofs are aggregated and verified onchain at an operator defined cadence

## Rollup Blocks



## L1 Blocks



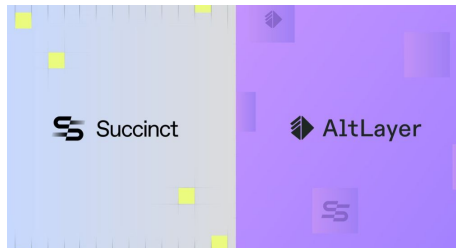
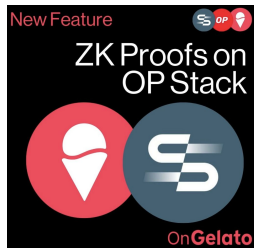
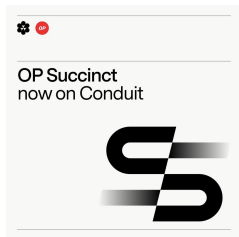
## What **<1 hour Finality** with ZK Validity Proofs **Enables**

- More liquidity on the chain
  - Large users are no longer concerned about long withdrawal windows
- More capital efficient rebalancing for cross chain intent / interop protocols
  - Reduced capital costs → reduces fees for users
- <1 hr interop today!
- Live validity proofs (stage 1) today!



# Practicality

Transaction Type	ETH Gas	Proving Cost
ETH Transfer	21000	\$0.001
ERC-20 Transfer	65000	\$0.0025
DEX Swap	185000	\$0.0070



# Security

Can be layered with other mechanisms for practical security assurances.

- Permissioned proof submitter (similar to whitelisted fault proof participants)
- “Mini challenge period” after proof submission

Security surface is re-used with production system



# Next Steps & Reach Out

- **5-10x cost reductions on the horizon:**
  - SP1 improvements
  - Protocol and software optimizations to Rust implementation of OP Stack's state transition function
- **Reach out:** @pumatheuma, <https://partner.succinct.xyz/>
- **Repos**
  - <https://github.com/succinctlabs/rsp>
  - <https://github.com/succinctlabs/op-succinct>

