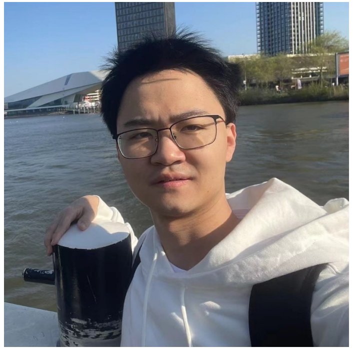
zkEVM design, optimization and applications

Guest Lecturer: Ye Zhang

Scroll



Zero Knowledge Proofs

Instructors: Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang





Stanford University

Berkeley UNIVERSITY OF CALIFORNIA

GEORGETOWN UNIVERSITY

TEXAS A&M UNIVERSITY

What is Scroll?

A scaling solution for Ethereum

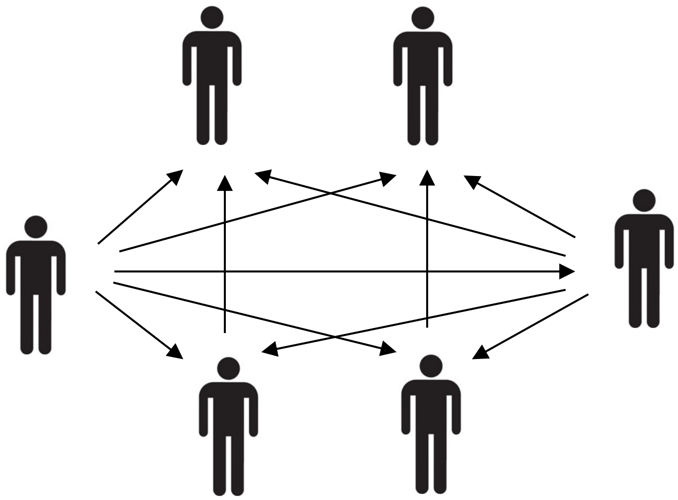
What is Scroll?

An EVM-equivalent zk-Rollup

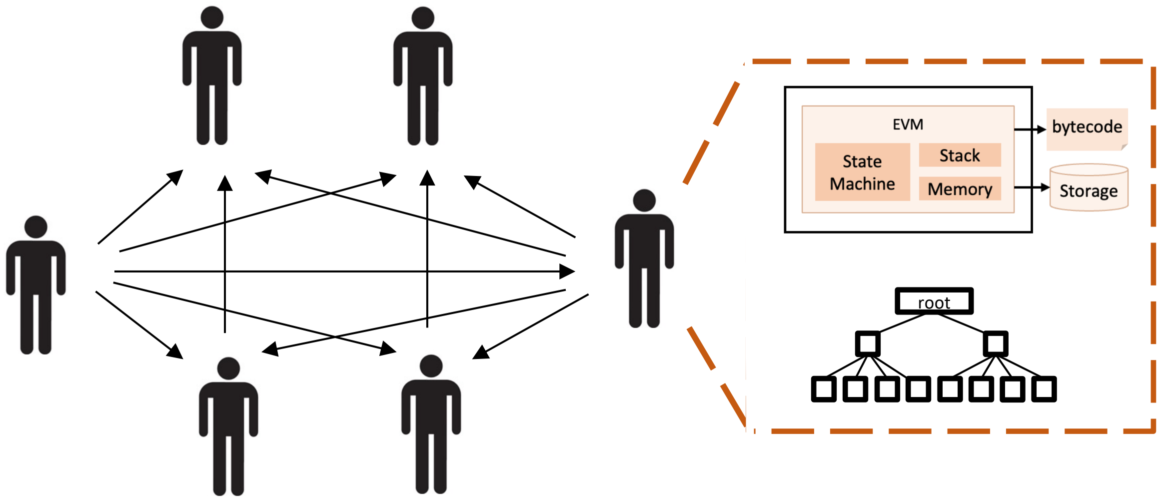
Outline

- Background & motivation
- Build a zkEVM from scratch
- Interesting research problems
- Other applications using zkEVM

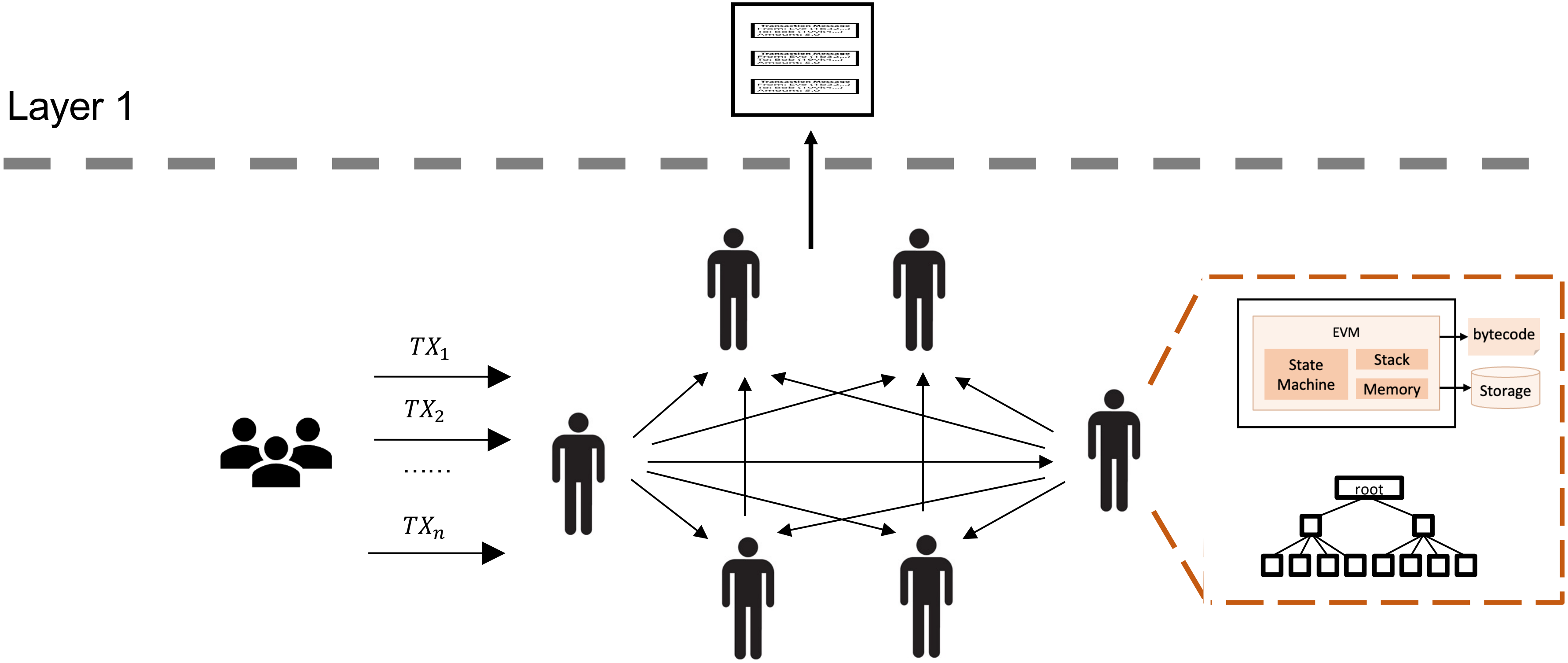
The diagram of Layer 1 blockchain



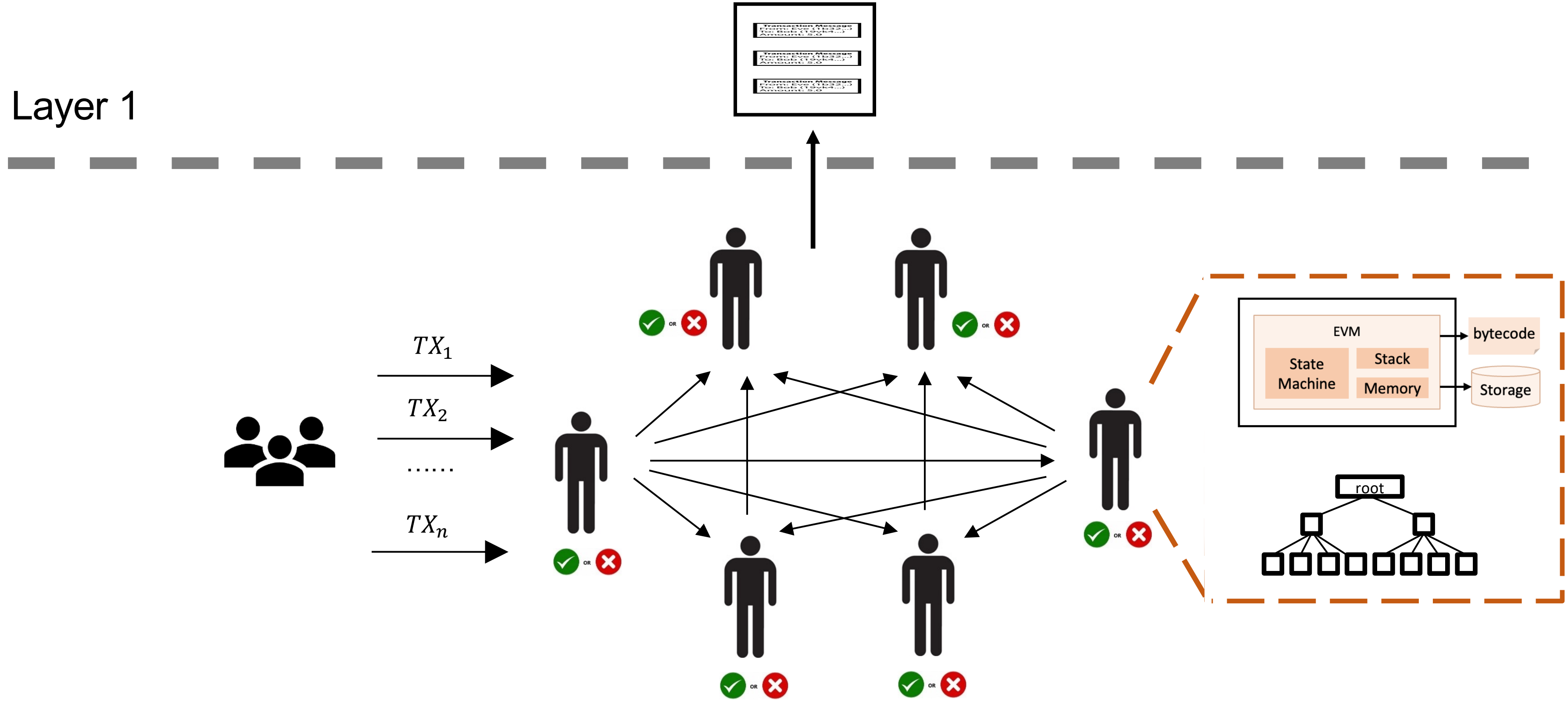
The diagram of Layer 1 blockchain



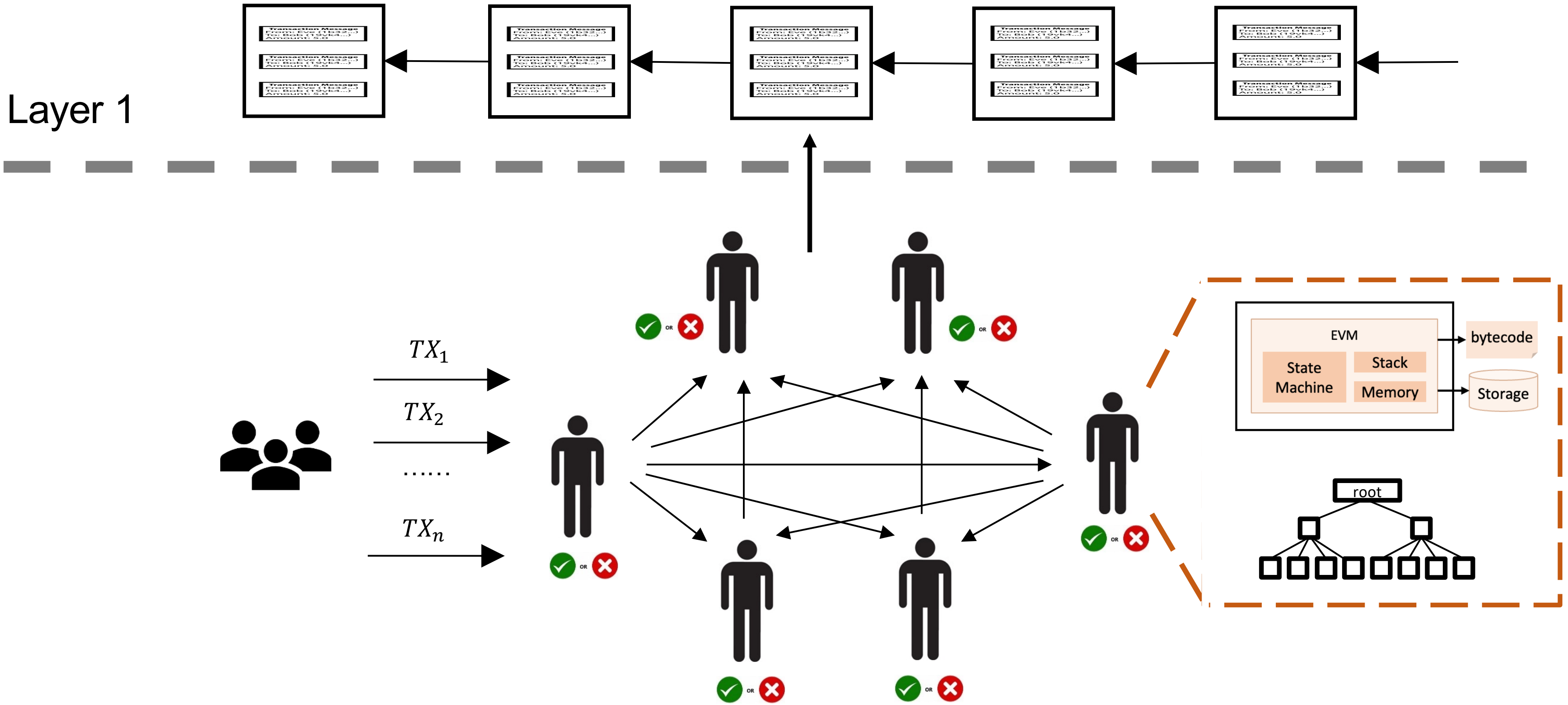
The diagram of Layer 1 blockchain



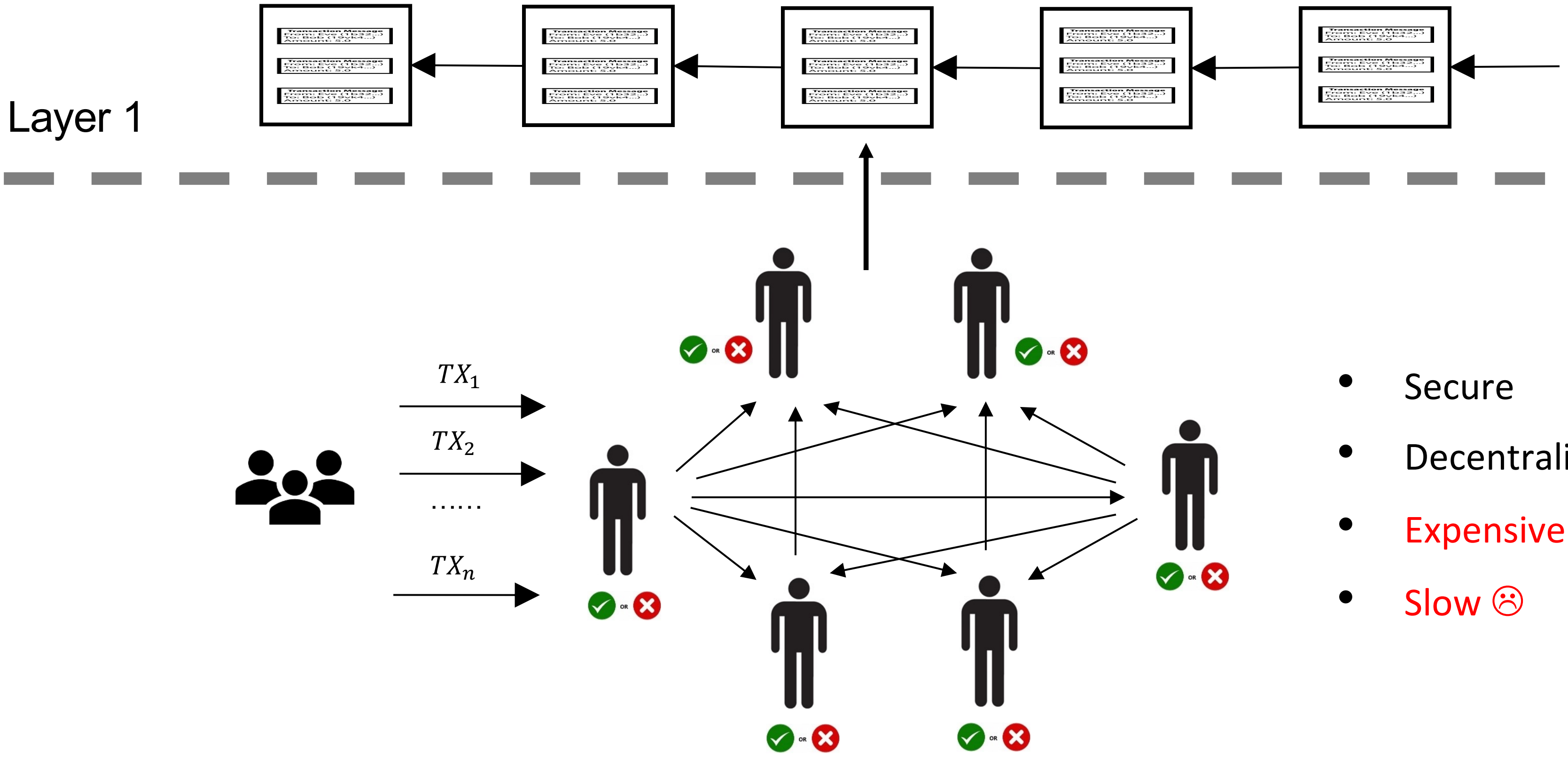
The diagram of Layer 1 blockchain



The diagram of Layer 1 blockchain

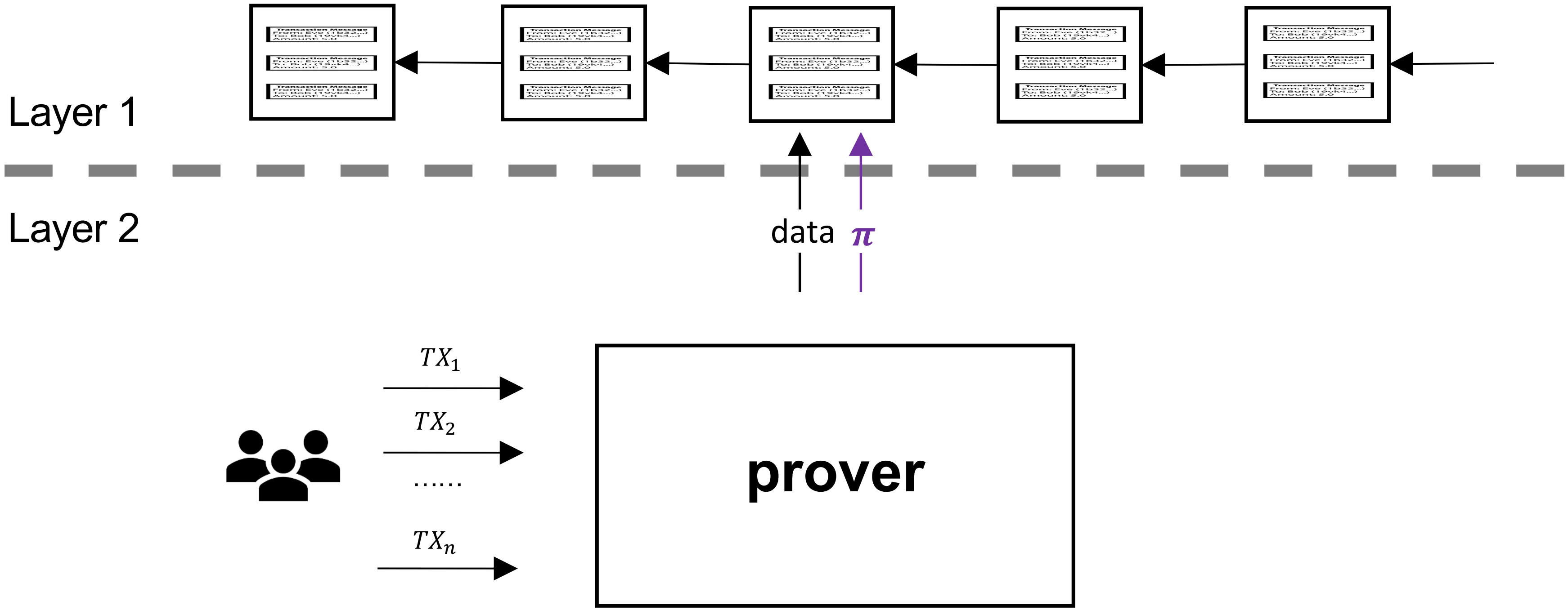


The diagram of Layer 1 blockchain



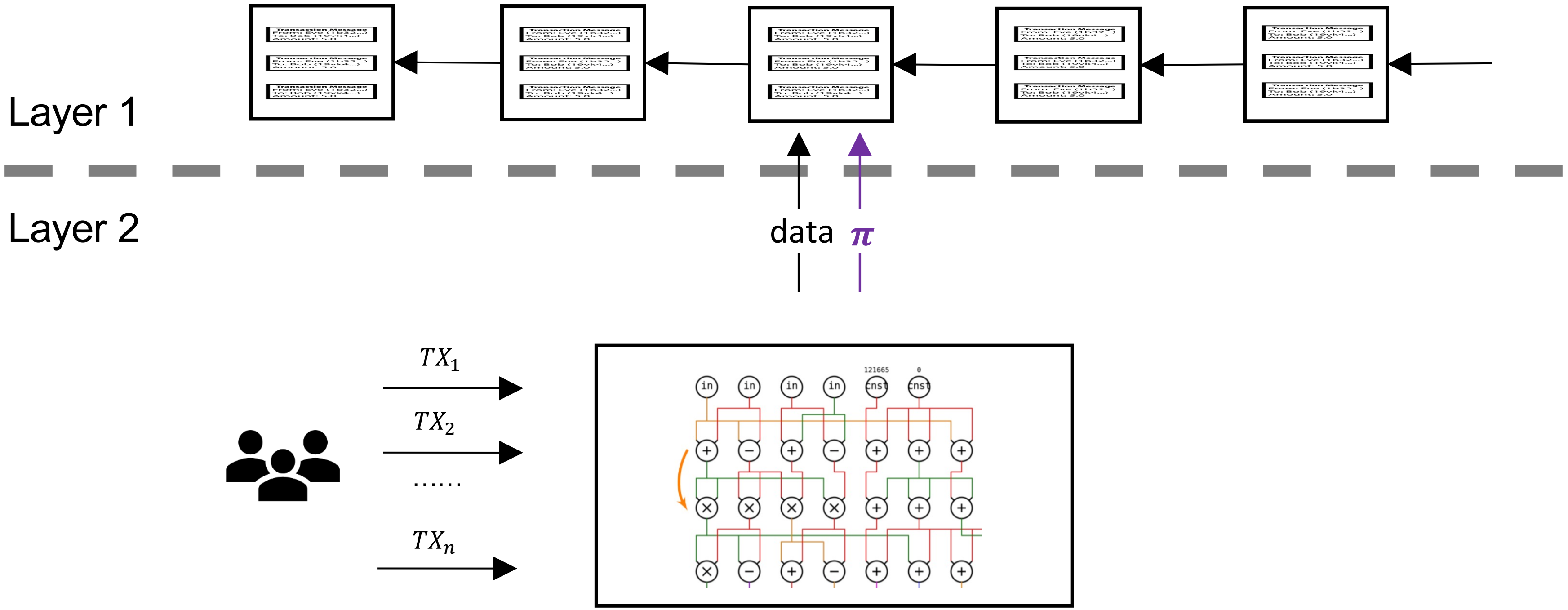
- Secure
- Decentralized
- Expensive ☹
- Slow ☹

Zk-Rollup

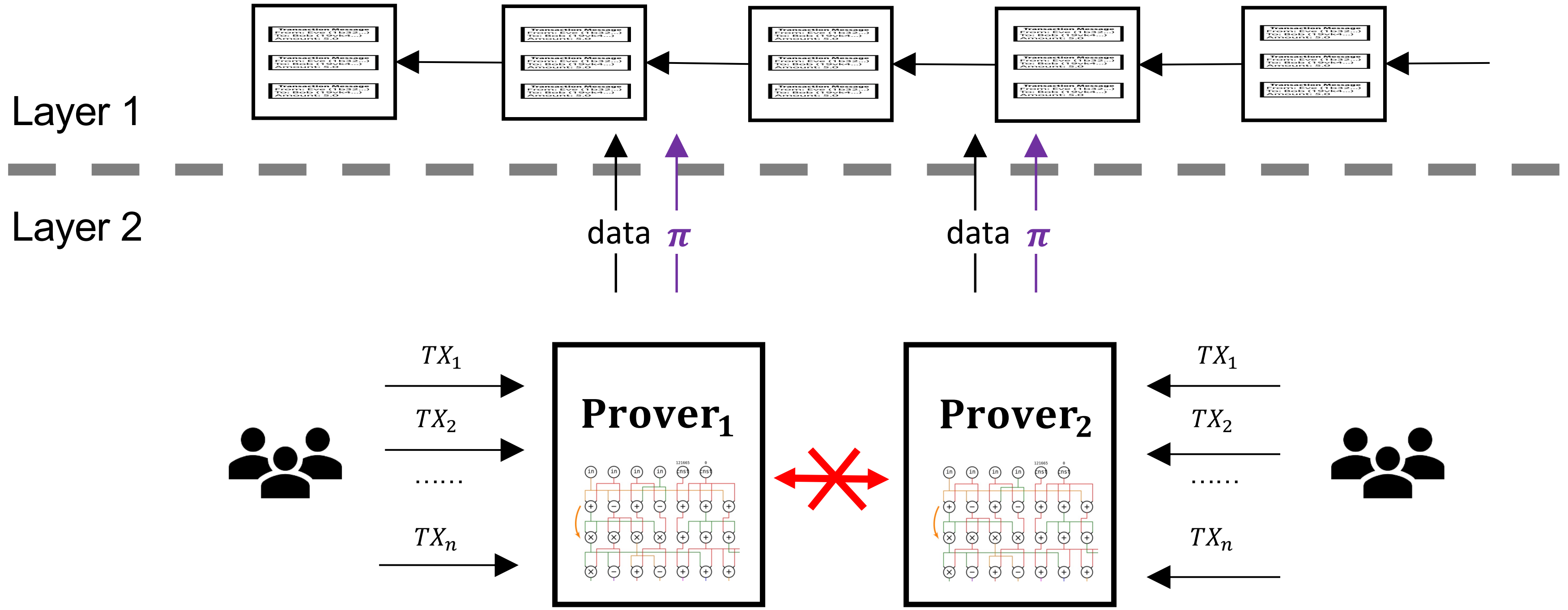


However, …

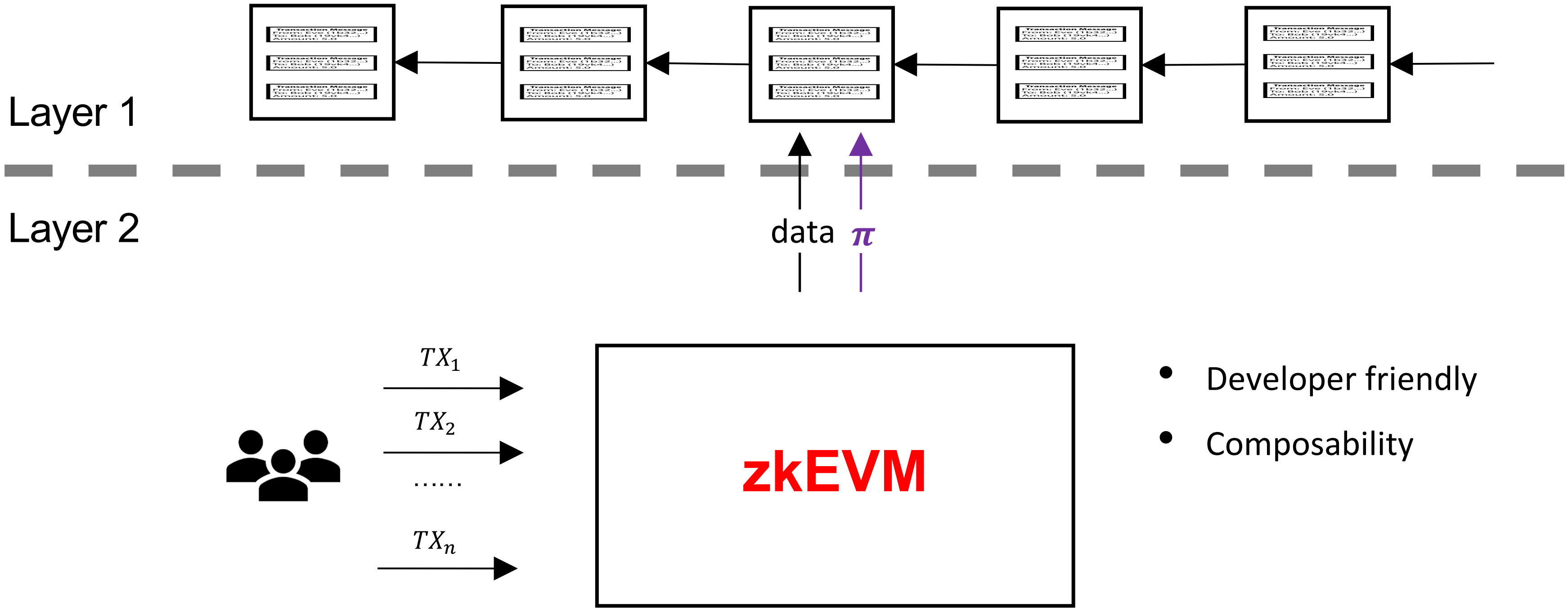
Scroll



However, …

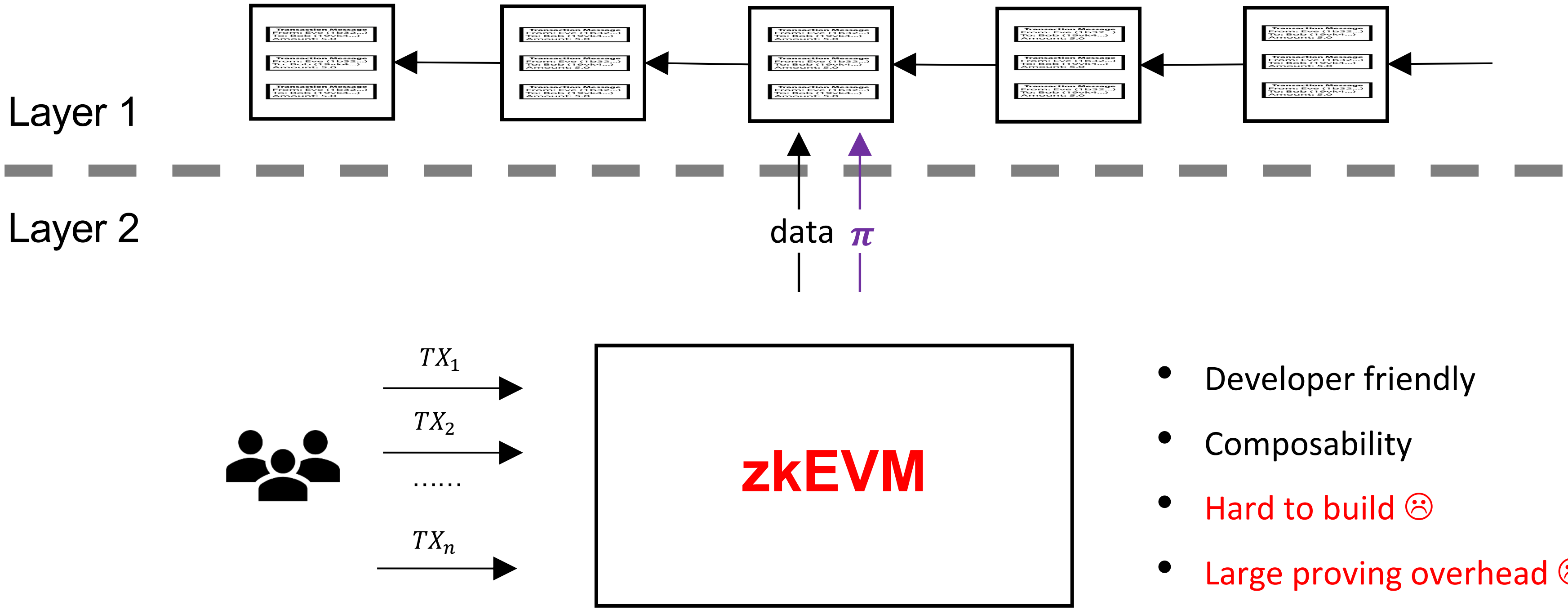

Layer 1
Layer 2
data
π
data
π
TX_1
TX_2
……
TX_n
Prover_1
Prover_2
TX_1
TX_2
……
TX_n

Scroll: a native zkEVM solution



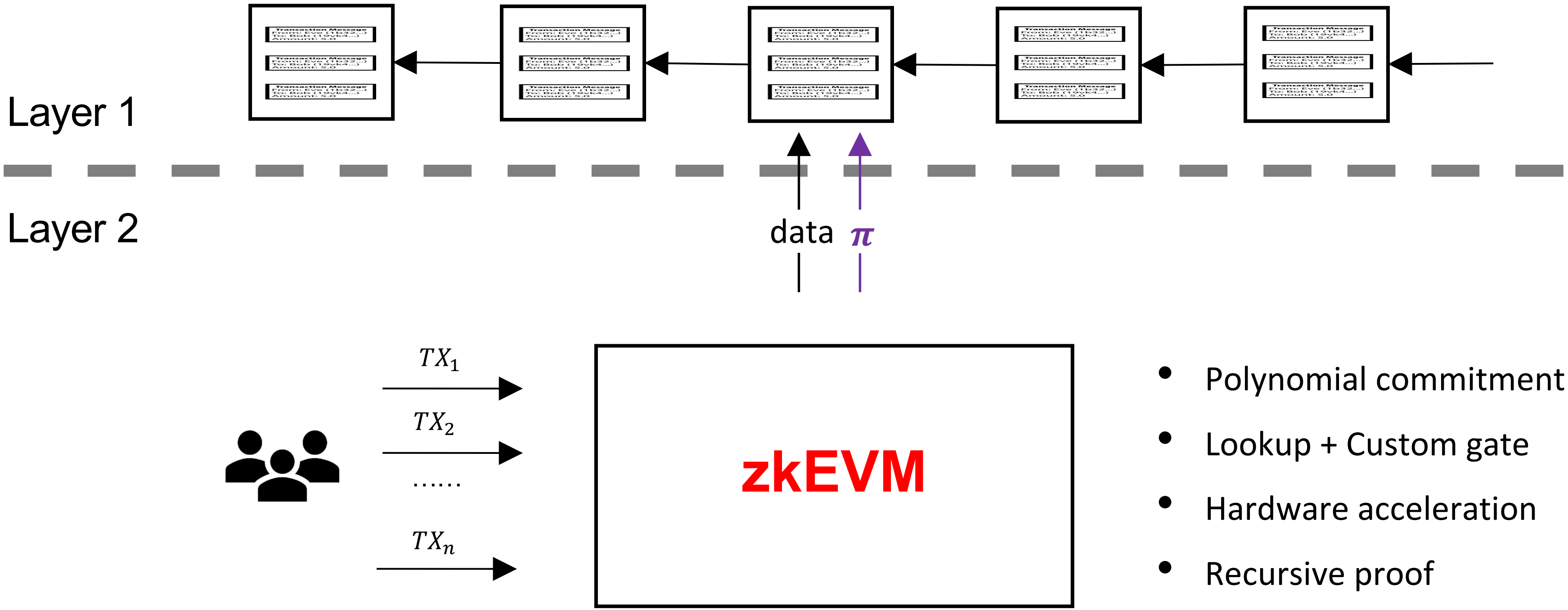
- Developer friendly
- Composability

Scroll: a native zkEVM solution



- Developer friendly
- Composability
- Hard to build ☹
- Large proving overhead ☹

Scroll: a native zkEVM solution



- Polynomial commitment
- Lookup + Custom gate
- Hardware acceleration
- Recursive proof

zkEVM flavors (by Justin Drake)

- **Language level**
  Transpile an EVM-friendly language (Solidity or Yul) to a SNARK-friendly VM which differs from the EVM. This is the approach of Matter Labs and Starkware.

- **Bytecode level**
  Interpret EVM bytecode directly, though potentially producing different state roots than the EVM, e.g. if certain implementation-level data structures are replaced with SNARK-friendly alternatives. This is the approach taken by Scroll, Hermez, and Consensys.

- **Consensus level**
  Target full equivalence with EVM as used by Ethereum L1 consensus. That is, it proves validity of L1 Ethereum state roots. This is part of the "zk-SNARK everything" roadmap for Ethereum.

zkEVM flavors (by Justin Drake)

- **Language level**
  Transpile an EVM-friendly language (Solidity or Yul) to a SNARK-friendly VM which differs from the EVM. This is the approach of Matter Labs and Starkware.
- **Bytecode level**
  Interpret EVM bytecode directly, though potentially producing different state roots than the EVM, e.g. if certain implementation-level data structures are replaced with SNARK-friendly alternatives. This is the approach taken by Scroll, Hermez, and Consensys.
- **Consensus level**
  Target full equivalence with EVM as used by Ethereum L1 consensus. That is, it proves validity of L1 Ethereum state roots. This is part of the "zk-SNARK everything" roadmap for Ethereum.

Outline

- Background & motivation
- Build a zkEVM from scratch
- Interesting research problems
- Other applications using zkEVM

The workflow of zero-knowledge proof

Scroll

Program

```python
def hcf(x, y):
    if x > y:
        smaller = y
    else:
        smaller = x

    for i in range(1,smaller + 1):
        if((x % i == 0) and (y % i == 0)):
            hcf = i

    return hcf
```

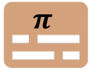
R1CS
Plonkish
AIR

Constraints

```
        x * x  ==  var1
     var1 * x  ==  y
    (y+x) * 1  ==  var2
 (var2+5) * 1  ==  out
```

Polynomial IOP
+
PCS

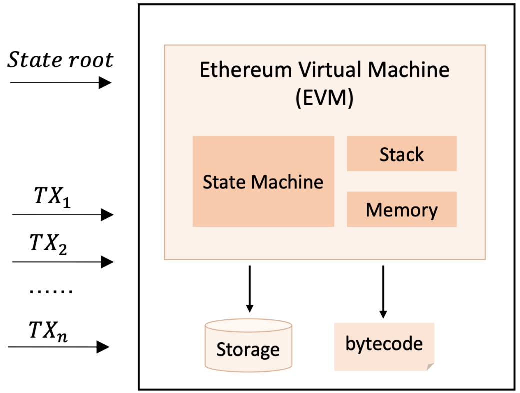
Proof



The workflow of zero-knowledge proof

Scroll

Program



R1CS
Plonkish
AIR

Constraints

x * x == var1
var1 * x == y
(y+x) * 1 == var2
(var2+5) * 1 == out

Polynomial IOP
+
PCS

Proof

π

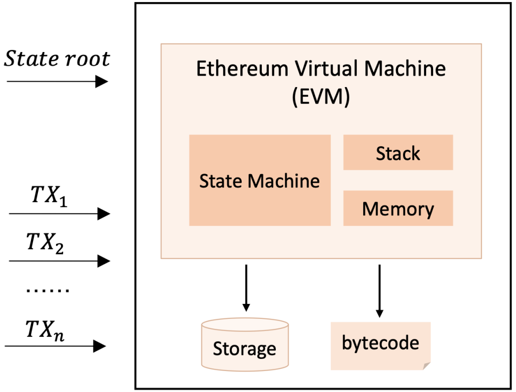
The workflow of zero-knowledge proof

Scroll

Program

Constraints

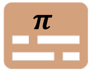
Proof



R1CS
Plonkish
AIR

```
x * x == var1
var1 * x == y
(y+x) * 1 == var2
(var2+5) * 1 == out
```

Plonk IOP
+
KZG



Let’s start with R1CS

w_1	w_2	w_3	w_4	w_5	……	w_{n-1}	w_n

Let's start with R1CS

w_1	w_2	w_3	w_4	w_5	……	w_{n-1}	w_n

$$(a_1 w_1 + \cdots + a_n w_n) * (b_1 w_1 + \cdots + b_n w_n) == (c_1 w_1 + \cdots + c_n w_n)$$

Let's start with R1CS

w_1	w_2	w_3	w_4	w_5	……	w_{n-1}	w_n

$$(a_1 w_1 + \cdots + a_n w_n) * (b_1 w_1 + \cdots + b_n w_n) == (c_1 w_1 + \cdots + c_n w_n)$$

$$(2w_1 + 1) * (3w_1 + 4w_2) == (w_{n-2} + 2)$$

$$(w_3 + 2) * (w_4) == (w_n + 1)$$

… …

… …

Let's start with R1CS

Scroll

w_1	w_2	w_3	w_4	w_5	……	w_{n-1}	w_n
$input_0$	$input_1$	$input_2$	va_1	vb_1	……	vc_1	vd_1

$$(a_1 w_1 + \cdots + a_n w_n) * (b_1 w_1 + \cdots + b_n w_n) == (c_1 w_1 + \cdots + c_n w_n)$$

$$(2w_1 + 1) * (3w_1 + 4w_2) == (w_{n-2} + 2)$$
$$(w_3 + 2) * (w_4) == (w_n + 1)$$

… …

… …

I know a vector ***{input, va, vb, vc, …}*** that satisfies all those constraints

Plonkish Arithmetization

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4

witness (a_0–a_4) **Table 1** (T_0–T_2) **Table 2** (T_3–T_4)

Plonkish Arithmetization

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	......	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness (a_0–a_4) — Table 1 (T_0–T_2) — Table 2 (T_3–T_4)

Plonkish Arithmetization – Custom gate

Scroll

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	……	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness (a_0–a_4) · Table 1 (T_0–T_2) · Table 2 (T_3–T_4)

$$va_3 * vb_3 * vc_3 - vb_4 = 0$$

Plonkish Arithmetization – Custom gate

Scroll

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	……	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness (columns a_0–a_4) | Table 1 (columns T_0–T_2) | Table 2 (columns T_3–T_4)

$$va_3 * vb_3 * vc_3 - vb_4 = 0$$

- High degree
- More customized

Plonkish Arithmetization – Custom gate

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	……	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness (a_0–a_4) | Table 1 (T_0–T_2) | Table 2 (T_3–T_4)

$$vb_1 * vc_1 + vc_2 - vc_3 = 0$$

- High degree
- More customized

Plonkish Arithmetization – Custom gate

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	……	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness (columns a_0–a_4) — Table 1 (columns T_0–T_2) — Table 2 (columns T_3–T_4)

$$vc_1 + va_2 * vb_4 - vc_4 = 0$$

- High degree
- More customized

Plonkish Arithmetization – Permutation

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	……	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness (columns a_0–a_4) | Table 1 (columns T_0–T_2) | Table 2 (columns T_3–T_4)

$$vb_4 = vc_6 = vb_6 = va_6$$

Plonkish Arithmetization – Lookup argument

Scroll

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	……	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7	Lookup						
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness — Table 1 — Table 2

$$(va_7, vb_7, vc_7) \in (T_0, T_1, T_2)$$

Plonkish Arithmetization – Lookup argument

Scroll

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$	0000				
va_1	vb_1	vc_1		vd_1	0001				
va_2	vb_2	vc_2		vd_2	0010				
va_3	vb_3	vc_3		vd_3	0011				
va_4	vb_4	vc_4	……	vd_4	0100				
va_5	vb_5	vc_5		vd_5	0101				
va_6	vb_6	vc_6		vd_6	……				
va_7	vb_7	vc_7	Lookup		1101				
va_6	vb_6	vc_6		vd_6	1110				
va_7	vb_7	vc_7		vd_7	1111				

witness (a_0–a_4) · Table 1 (T_0–T_2) · Table 2 (T_3–T_4)

$$vc_7 \in [0, 15]$$

Plonkish Arithmetization – Lookup argument

Scroll

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$	0000	0000	0000		
va_1	vb_1	vc_1		vd_1	0000	0001	0001		
va_2	vb_2	vc_2		vd_2	0000	0010	0010		
va_3	vb_3	vc_3		vd_3	0000	0011	0011		
va_4	vb_4	vc_4	……	vd_4	0000	0100	0100		
va_5	vb_5	vc_5		vd_5	0000	0101	0101		
va_6	vb_6	vc_6		vd_6	……	……	……		
va_7	vb_7	vc_7	Lookup		1111	1101	0010		
va_6	vb_6	vc_6		vd_6	1111	1110	0001		
va_7	vb_7	vc_7		vd_7	1111	1111	0000		

witness (a_0–a_4) | Table 1 (T_0–T_2) | Table 2 (T_3–T_4)

$$vc_7 \in [0, 15]$$

$$va_7 \oplus vb_7 = vc_7$$

Plonkish Arithmetization – Lookup argument

Scroll

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$	0000	0000	0000		
va_1	vb_1	vc_1		vd_1	0000	0001	0001		
va_2	vb_2	vc_2		vd_2	0000	0010	0010		
va_3	vb_3	vc_3		vd_3	0000	0011	0011		
va_4	vb_4	vc_4	……	vd_4	0000	0100	0100		
va_5	vb_5	vc_5		vd_5	0000	0101	0101		
va_6	vb_6	vc_6		vd_6	……	……	……		
va_7	vb_7	vc_7	Lookup		1111	1101	0010		
va_6	vb_6	vc_6		vd_6	1111	1110	0001		
va_7	vb_7	vc_7		vd_7	1111	1111	0000		

witness (a_0–a_4) | Table 1 (T_0–T_2) | Table 2 (T_3–T_4)

$$vc_7 \in [0, 15]$$

$$va_7 \oplus vb_7 = vc_7$$

RAM operation

Plonkish Arithmetization – Constraints

Scroll

a_0	a_1	a_2	a_3	a_4	T_0	T_1	T_2	T_3	T_4
$input_0$	$input_1$	$input_2$		$output$					
va_1	vb_1	vc_1		vd_1					
va_2	vb_2	vc_2		vd_2					
va_3	vb_3	vc_3		vd_3					
va_4	vb_4	vc_4	……	vd_4					
va_5	vb_5	vc_5		vd_5					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					
va_6	vb_6	vc_6		vd_6					
va_7	vb_7	vc_7		vd_7					

witness (a_0–a_4) | Table 1 (T_0–T_2) | Table 2 (T_3–T_4)

$$vb_1 * vc_1 + vc_2 - vc_3 = 0$$
$$va_3 * vb_3 * vc_3 - vb_4 = 0$$
$$vb_4 + vc_6 * vb_6 - va_6 = 0$$

……

$$vb_4 = vc_6 = vb_6 = va_6$$

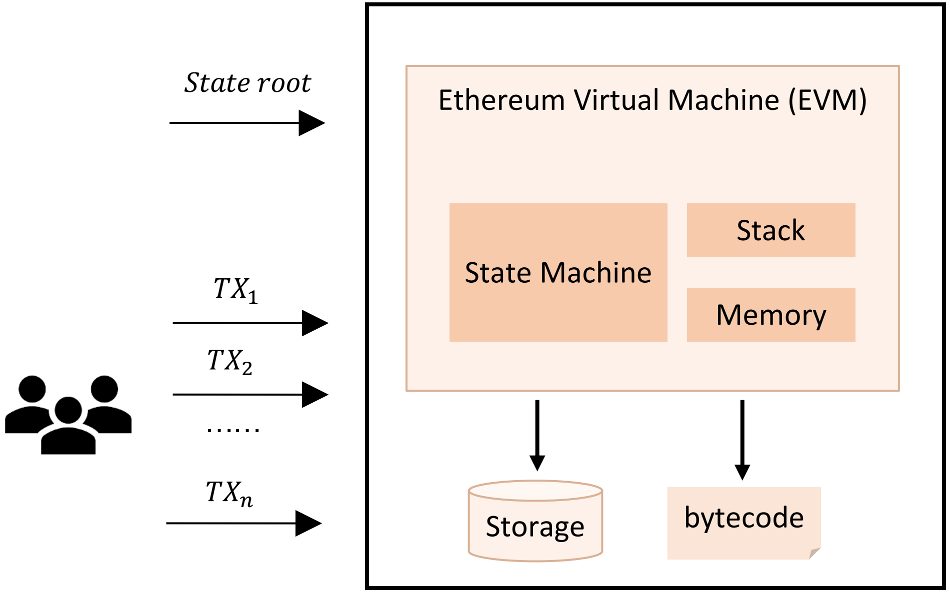
……

$$(va_7, vb_7, vc_7) \in (T_0, T_1, T_2)$$

How should we choose "front-end"?

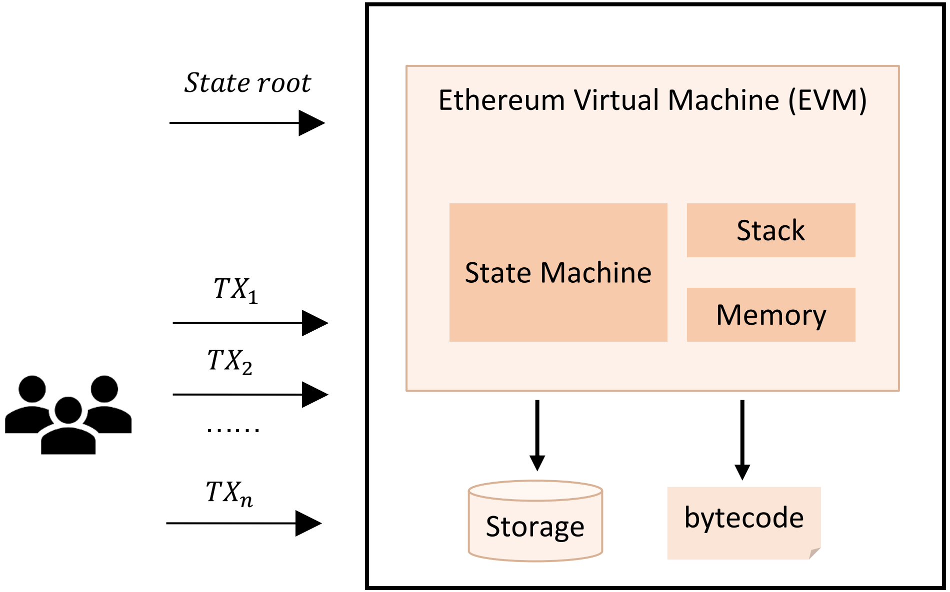
Scroll

Computation



How should we choose "front-end"?

Computation

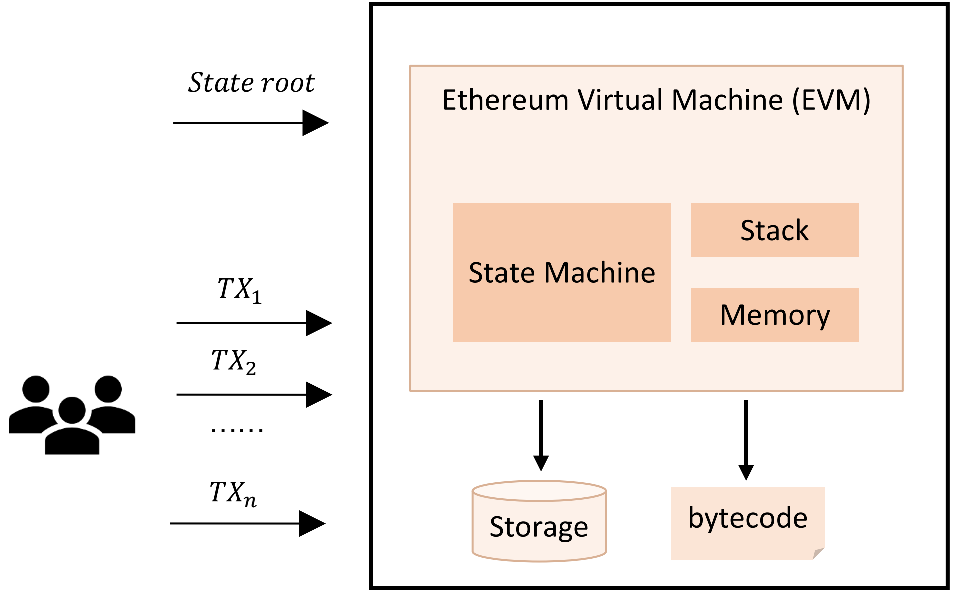


- EVM word size is 256bit
  - Efficient range proof
- EVM has zk-unfriendly opcodes
  - Efficient way to connect circuits
- Read & Write consistency
  - Efficient mapping
- EVM has a dynamic execution trace
  - Efficient on/off selectors

How should we choose "front-end"?

Scroll

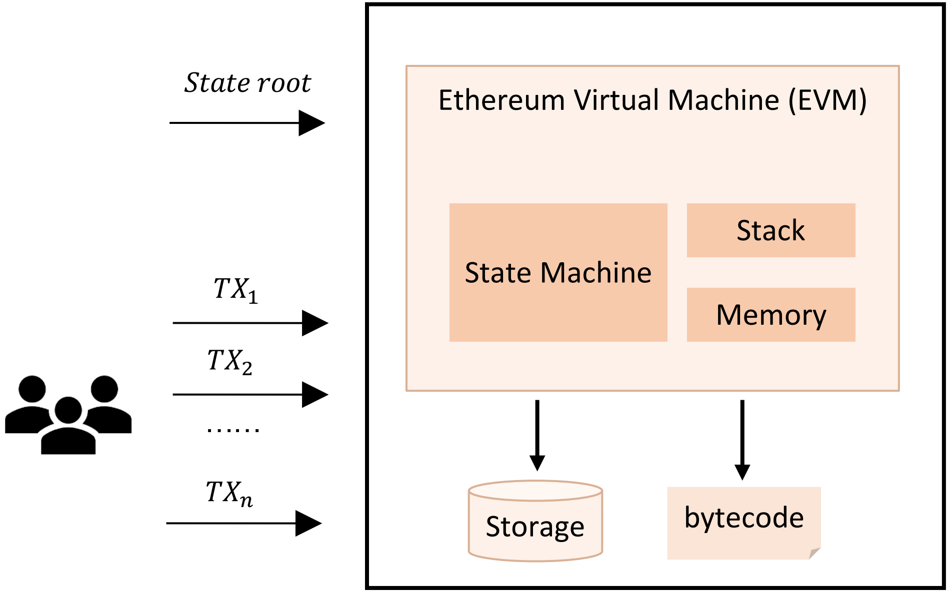
Computation



- EVM word size is 256bit
  - Efficient range proof
- EVM has zk-unfriendly opcodes
  - Efficient way to connect circuits
- Read & Write consistency
  - Efficient mapping
- EVM has a dynamic execution trace
  - Efficient on/off selectors

How should we choose “front-end”?

Computation

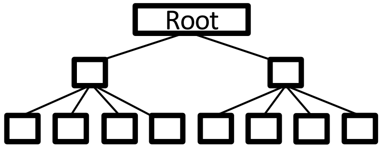


- EVM word size is 256bit
    - Efficient range proof
- EVM has zk-unfriendly opcodes
    - Efficient way to connect circuits
- Read & Write consistency
    - Efficient mapping

- EVM has a dynamic execution trace
    - Efficient on/off selectors

What you need to prove

World State (t)

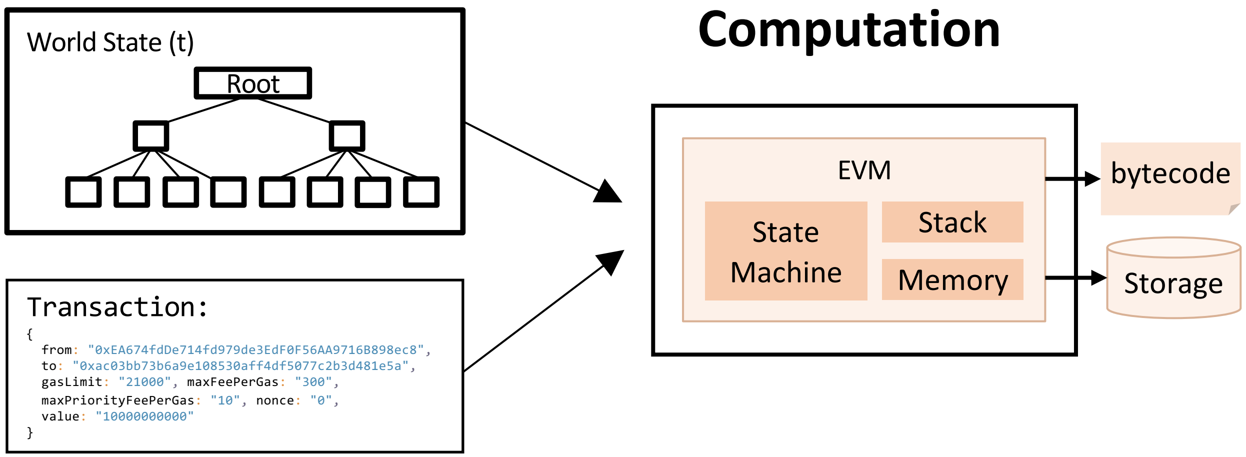


Transaction:

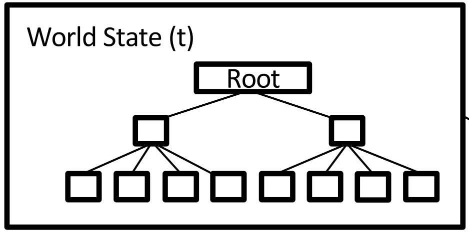
```
{
  from: "0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8",
  to: "0xac03bb73b6a9e108530aff4df5077c2b3d481e5a",
  gasLimit: "21000", maxFeePerGas: "300",
  maxPriorityFeePerGas: "10", nonce: "0",
  value: "10000000000"
}
```

What you need to prove

Scroll

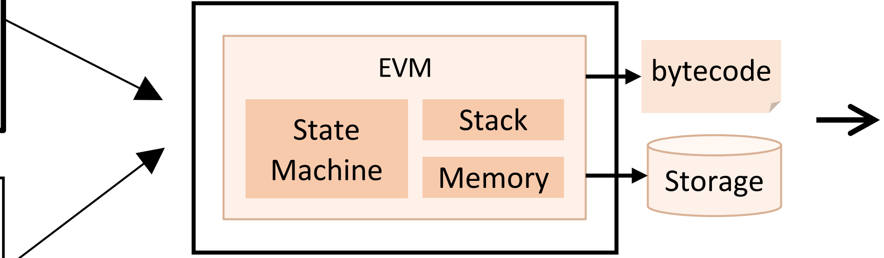


What you need to prove



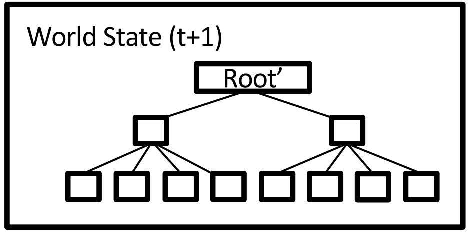

```
Transaction:
{
  from: "0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8",
  to: "0xac03bb73b6a9e108530aff4df5077c2b3d481e5a",
  gasLimit: "21000", maxFeePerGas: "300",
  maxPriorityFeePerGas: "10", nonce: "0",
  value: "10000000000"
}
```

Computation

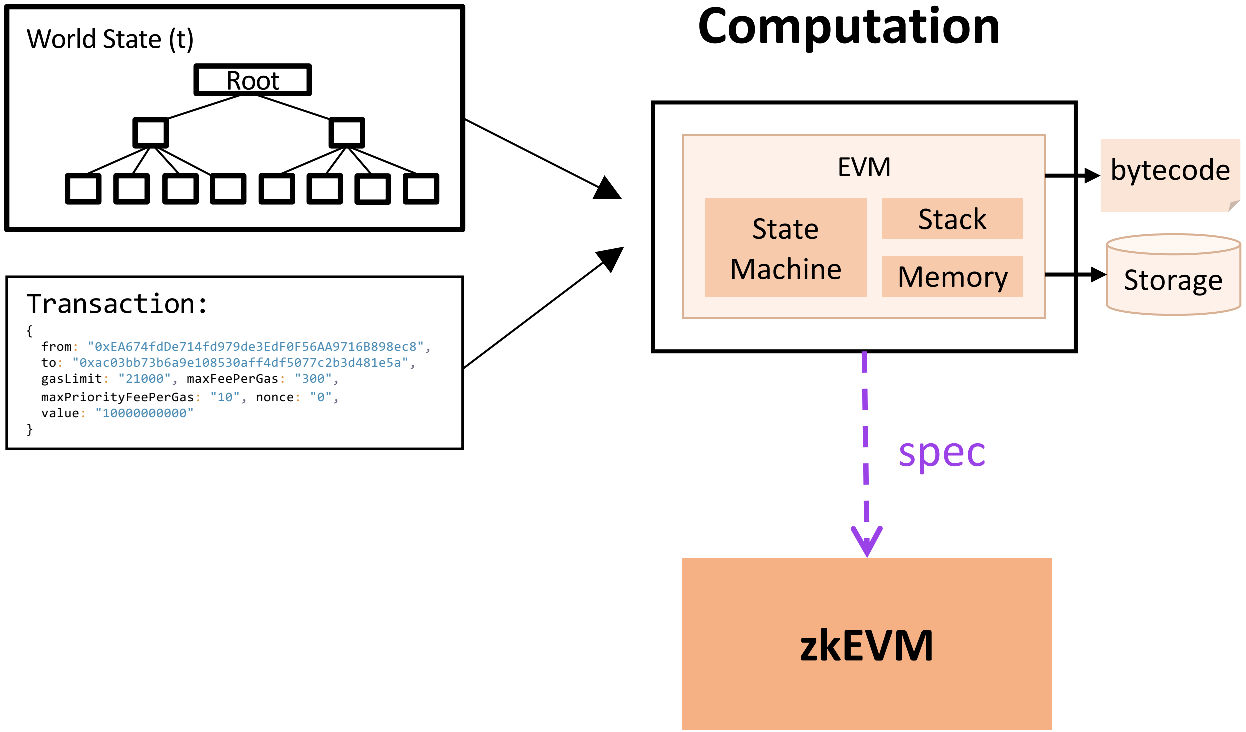


Execution trace

Step	Opcode
0	PUSH1 80
1	PUSH1 40
2	MSTORE
3	CALLVALUE
...	...
n	RETURN

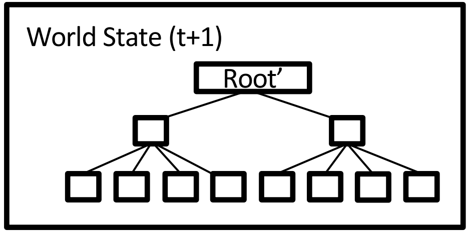


What you need to prove

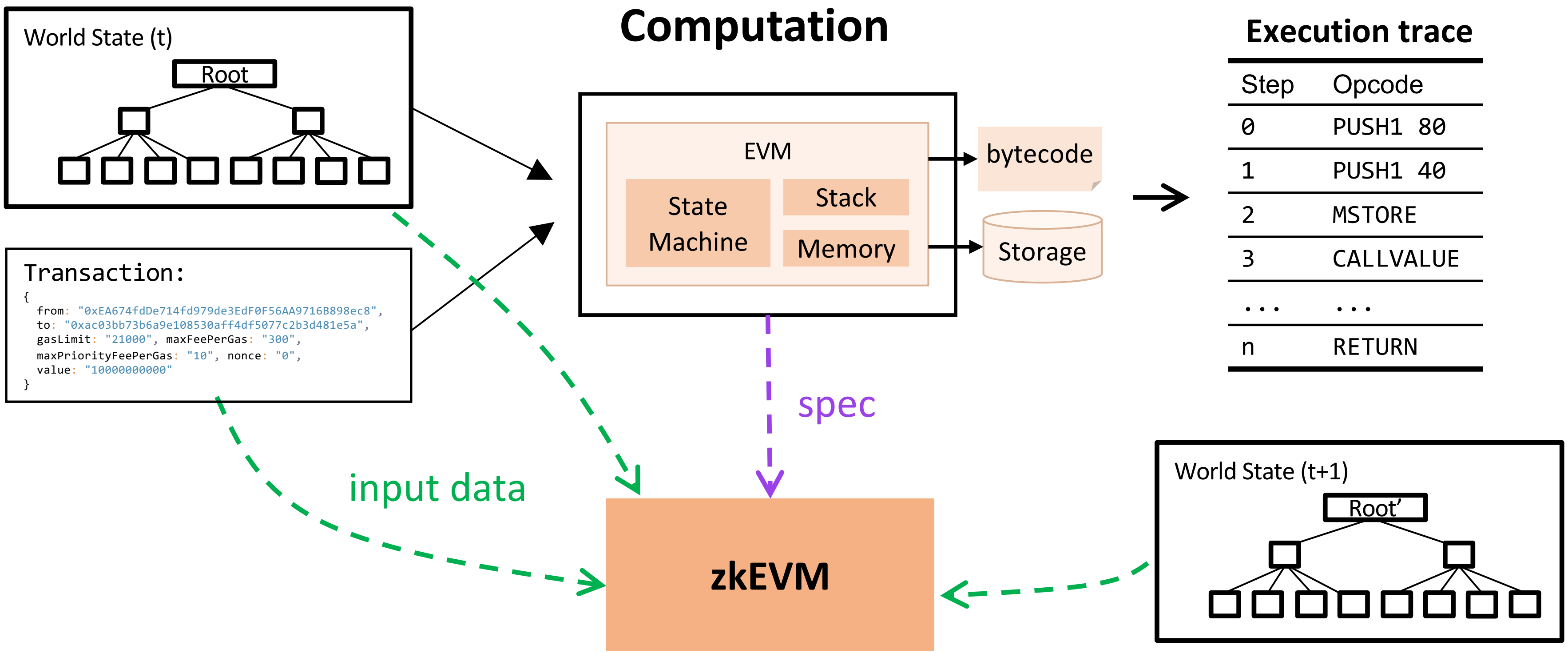


Execution trace

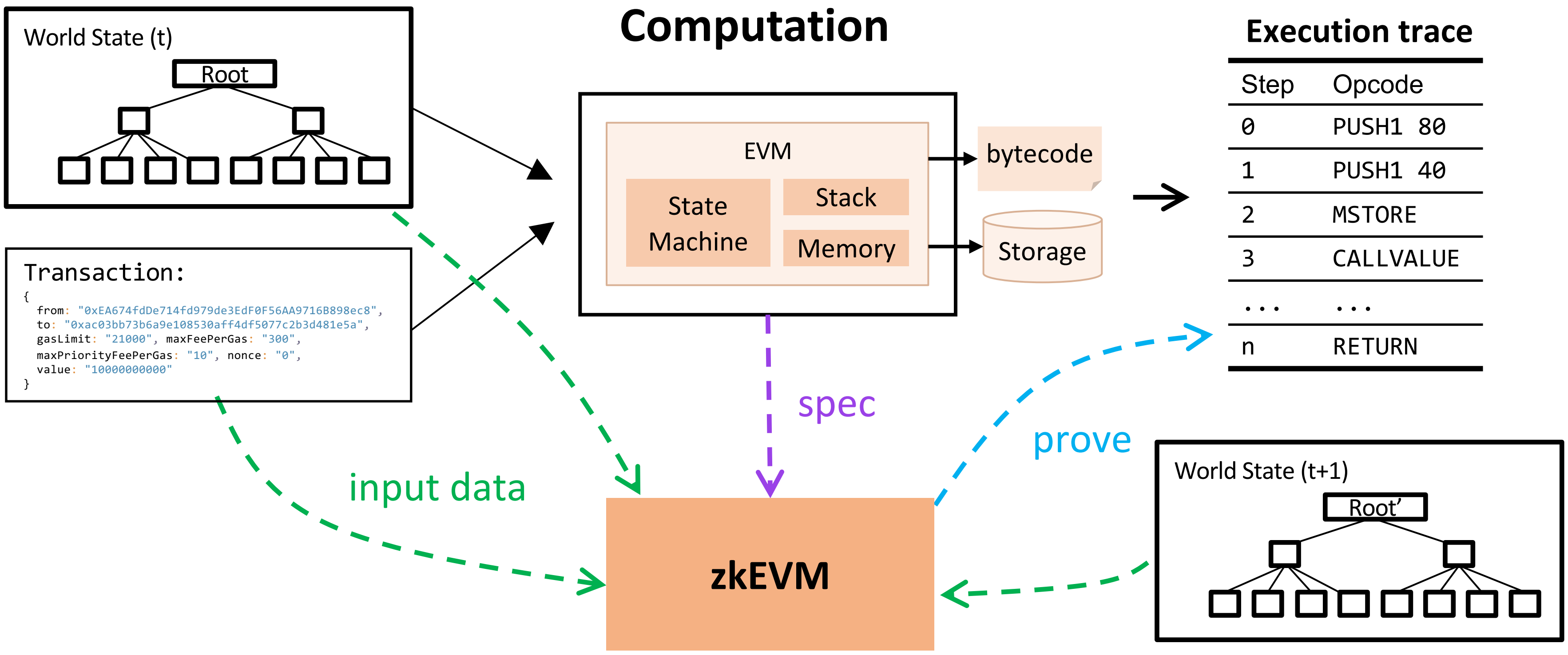
Step	Opcode
0	PUSH1 80
1	PUSH1 40
2	MSTORE
3	CALLVALUE
...	...
n	RETURN



What you need to prove



What you need to prove

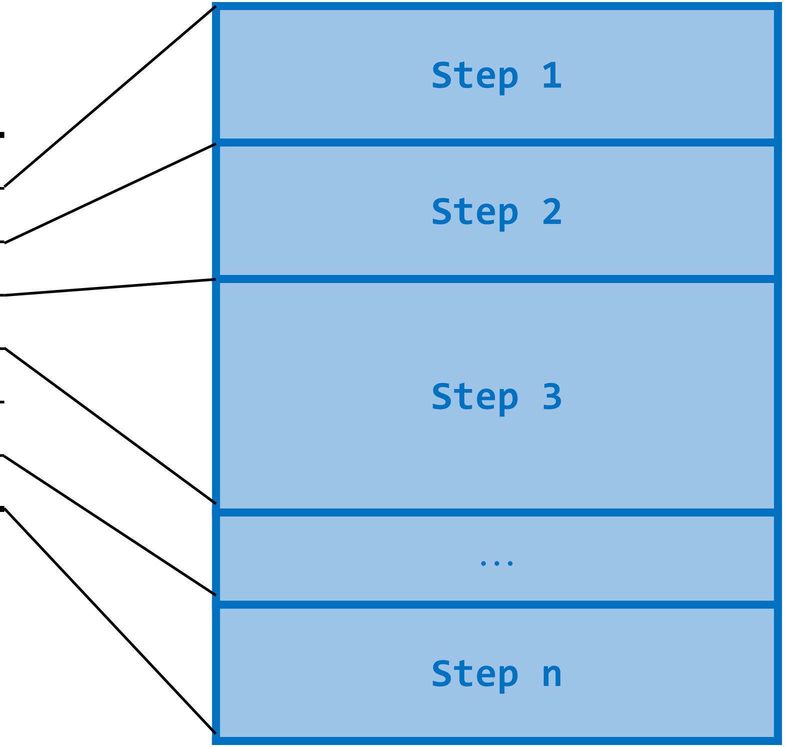


EVM circuit

Scroll

Execution trace

step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN

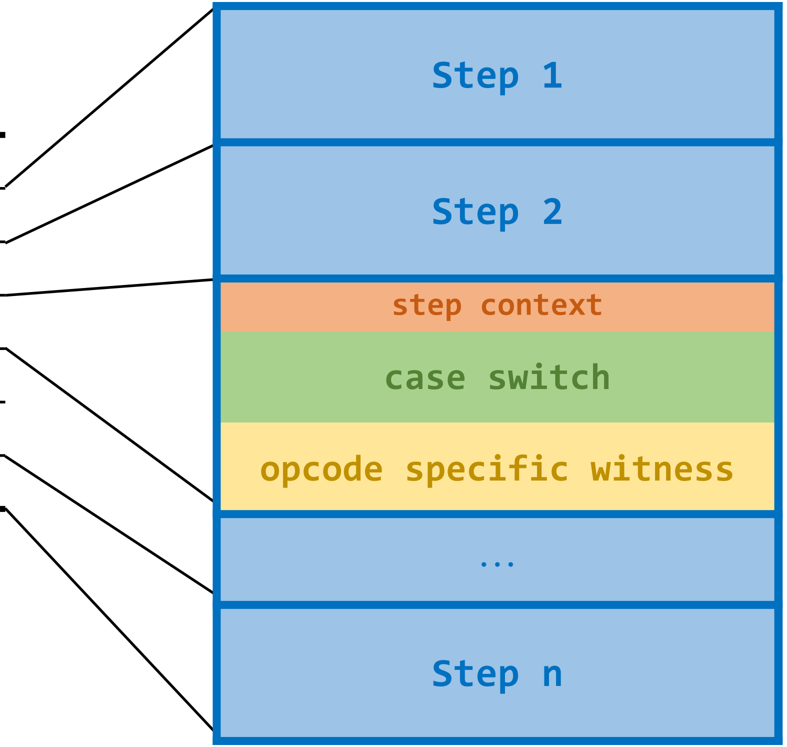


EVM circuit

Scroll

Execution trace

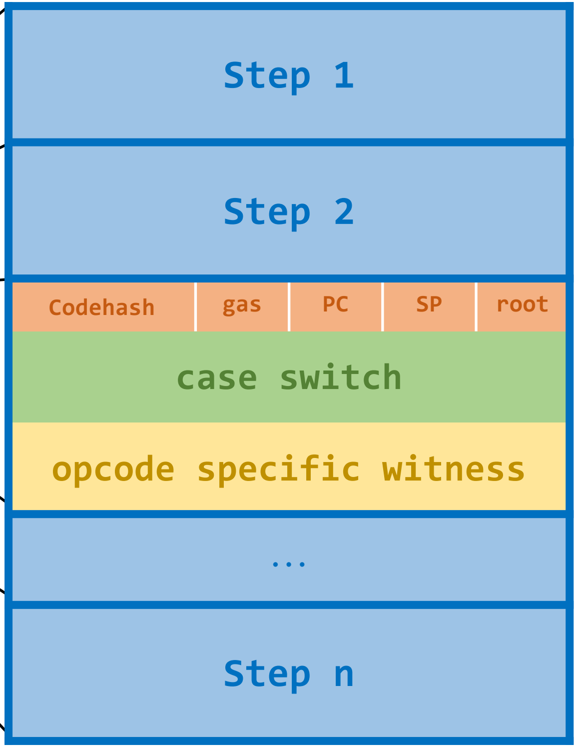
step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN



EVM circuit

Execution trace

step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN

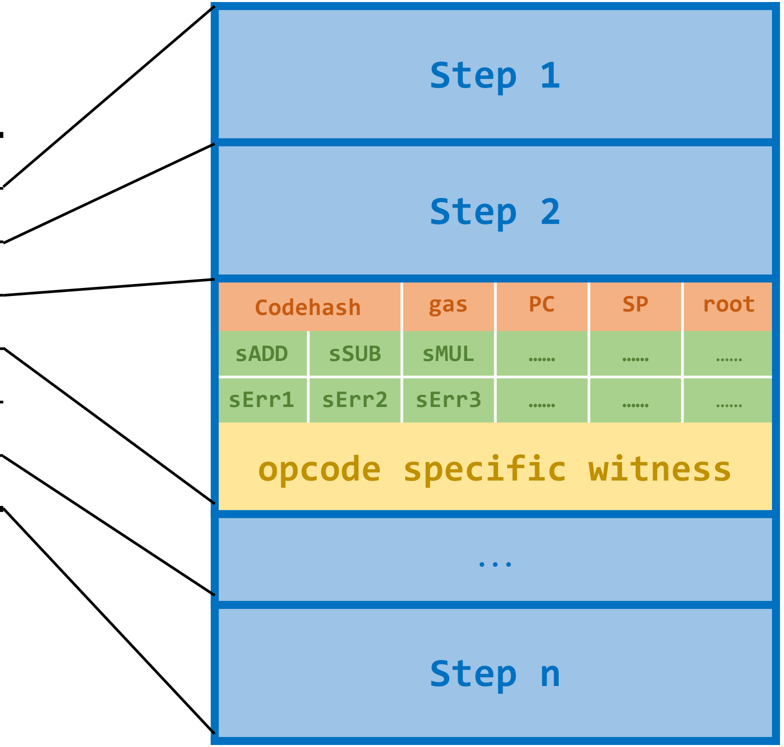


- **Step context**
  - Codehash
  - Gas left
  - Program counter, Stack pointer

EVM circuit

Execution trace

step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN



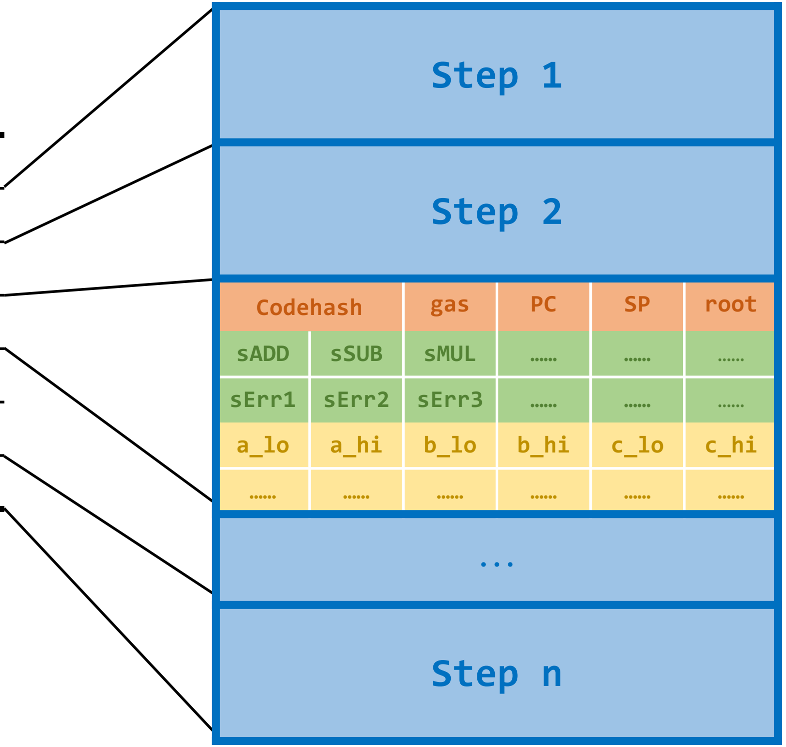
- **Step context**
  - Codehash
  - Gas left
  - Program counter, Stack pointer
- **Case switch**
  - Select opcodes & error cases
  - Exactly one is switched on

EVM circuit

Scroll

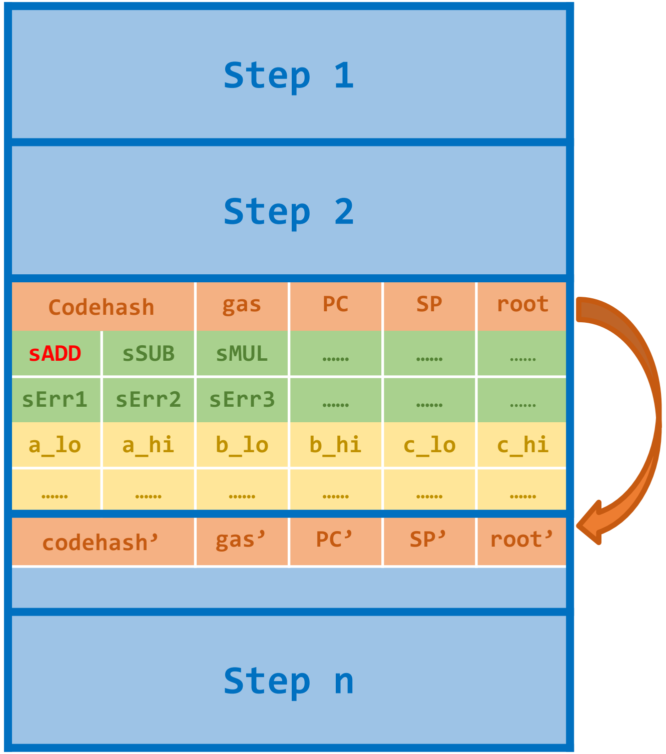
Execution trace

step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN



- **Step context**
  - Codehash
  - Gas left
  - Program counter, Stack pointer
- **Case switch**
  - Select opcodes & error cases
  - Exactly one is switched on
- **Opcode specific witness**
  - Extra witness used for opcodes
  - i.e. operands, carry, limbs, …

EVM circuit - ADD



- **Step context**

```
sADD * (pc’ – pc – 1) == 0
sADD * (sp’ – sp – 1) == 0
sADD * (gas’ – gas – 3) == 0
```

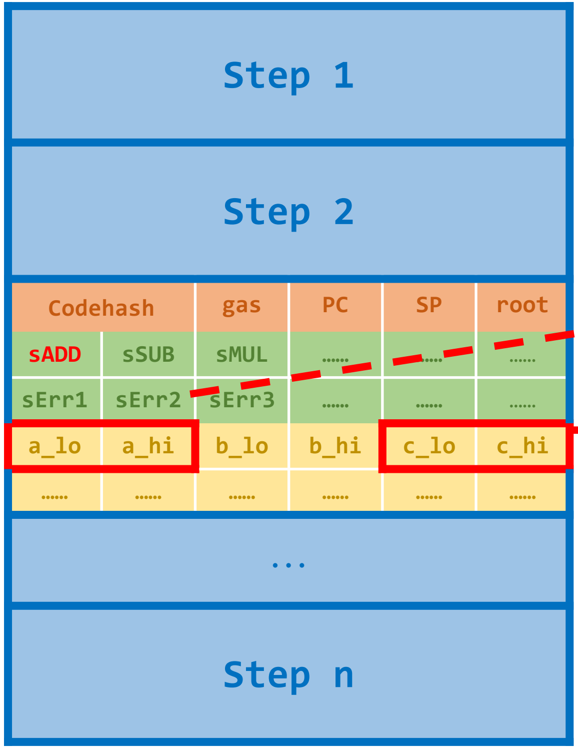
- **Case switch**

```
sADD * (1 – sADD) == 0
sMUL * (1 – sMUL) == 0
        ...
sADD + sMUL + ... + sERRk == 1
```

- **Opcode specific witness**

```
sADD*(a_lo+b_lo-c_lo – carry0* 2^128)== 0
sADD*(a_hi+b_hi+carry0-c_hi – carry1*2^128)== 0
```

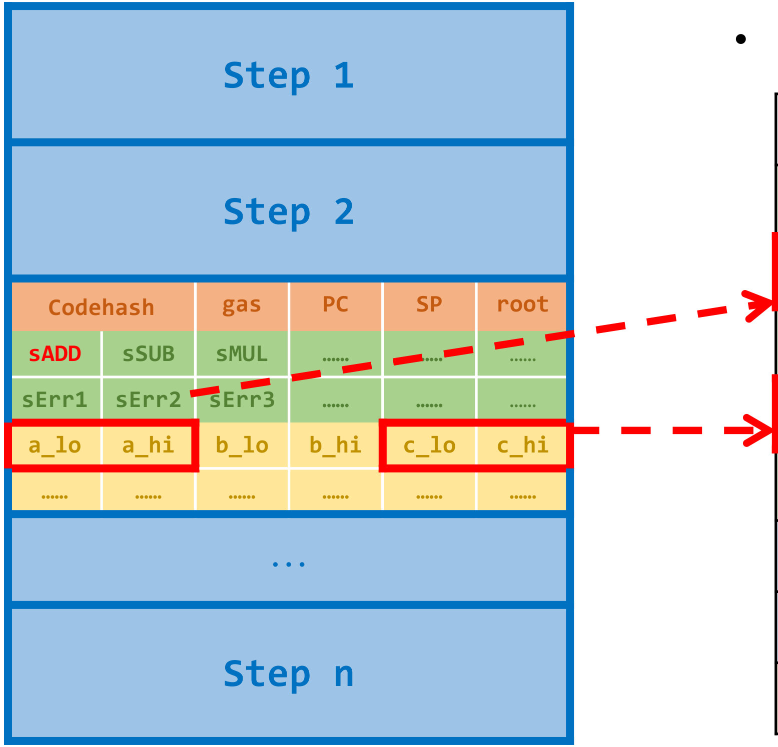
EVM circuit - ADD



- **Opcode specific witness**

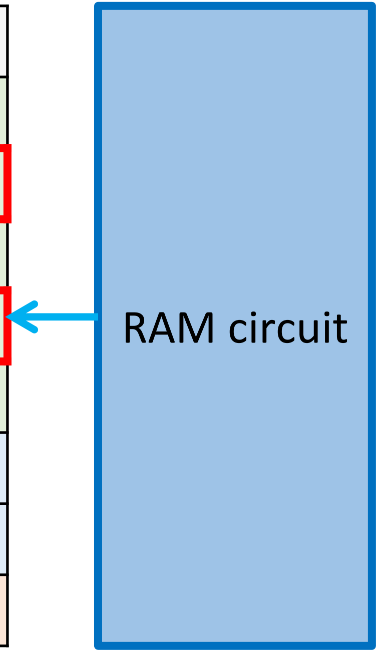
idx	tag	addr	R/W	value
1	STACK	1023	1	…
5	STACK	1022	0	word_a
6	STACK	1023	0	word_b
7	STACK	1023	1	word_c
…	STACK	…	…	…
…	MEMORY	0x40	1	…
…	MEMORY	…	…	…
…	STORAGE	…	…	…

EVM circuit - ADD



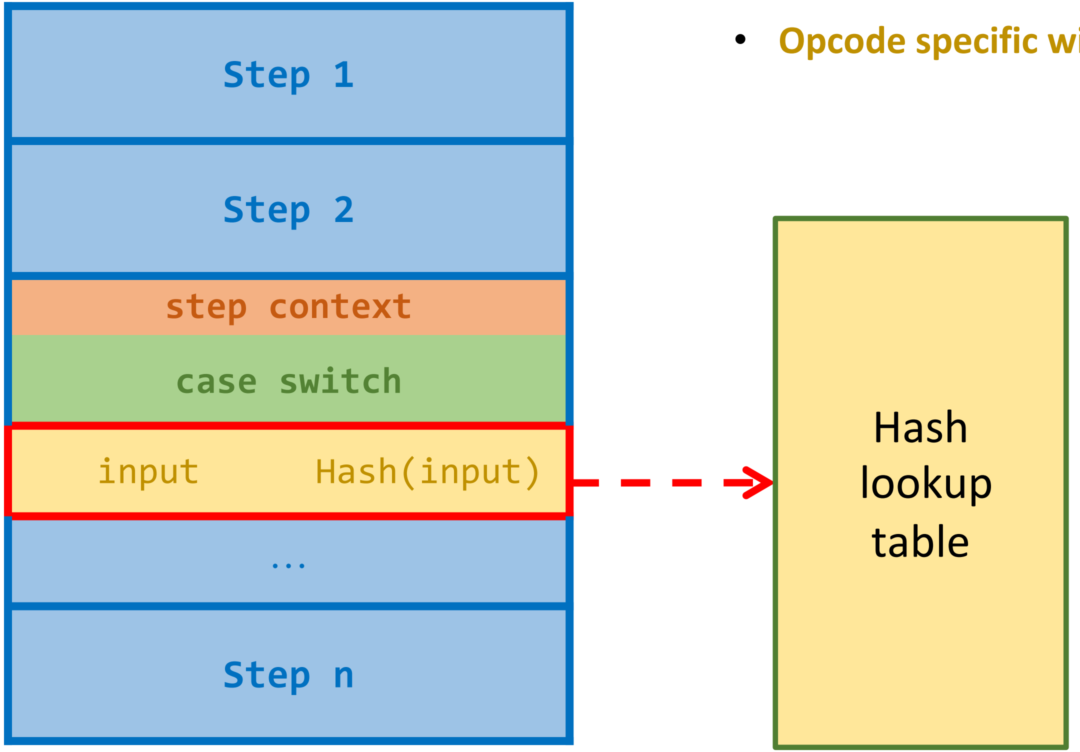
- **Opcode specific witness**

idx	tag	addr	R/W	value
1	STACK	1023	1	…
5	STACK	1022	0	word_a
6	STACK	1023	0	word_b
7	STACK	1023	1	word_c
…	STACK	…	…	…
…	MEMORY	0x40	1	…
…	MEMORY	…	…	…
…	STORAGE	…	…	…



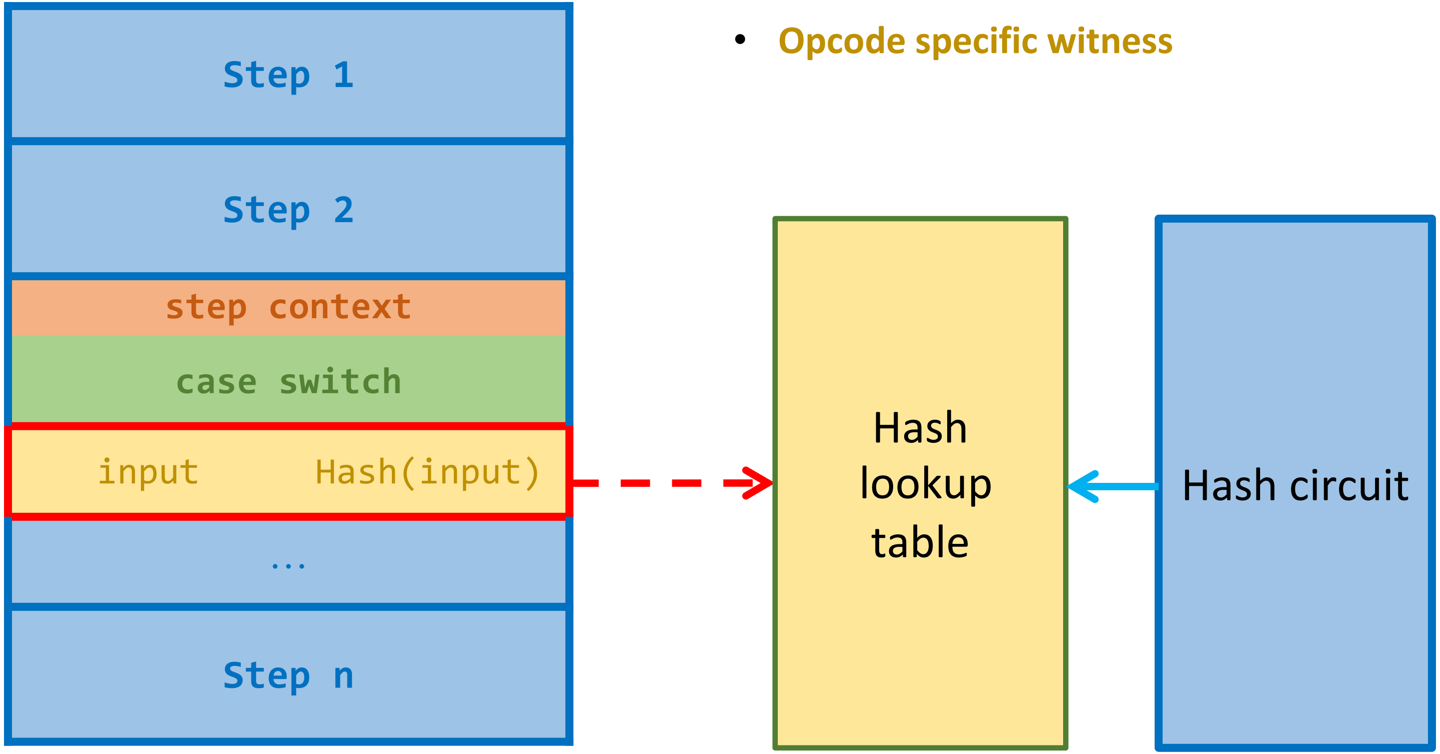
EVM circuit - Hash

- **Opcode specific witness**

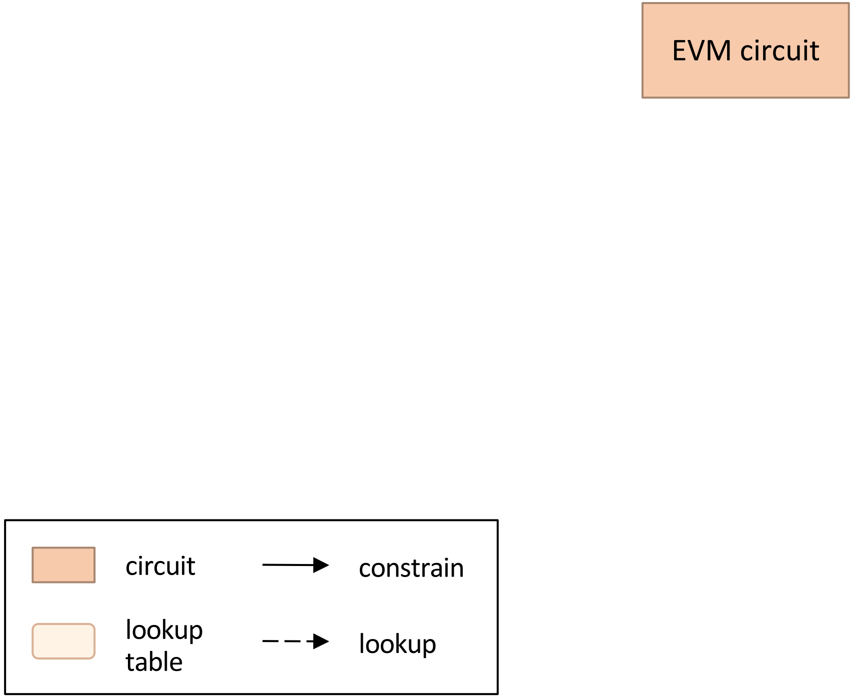


EVM circuit - Hash

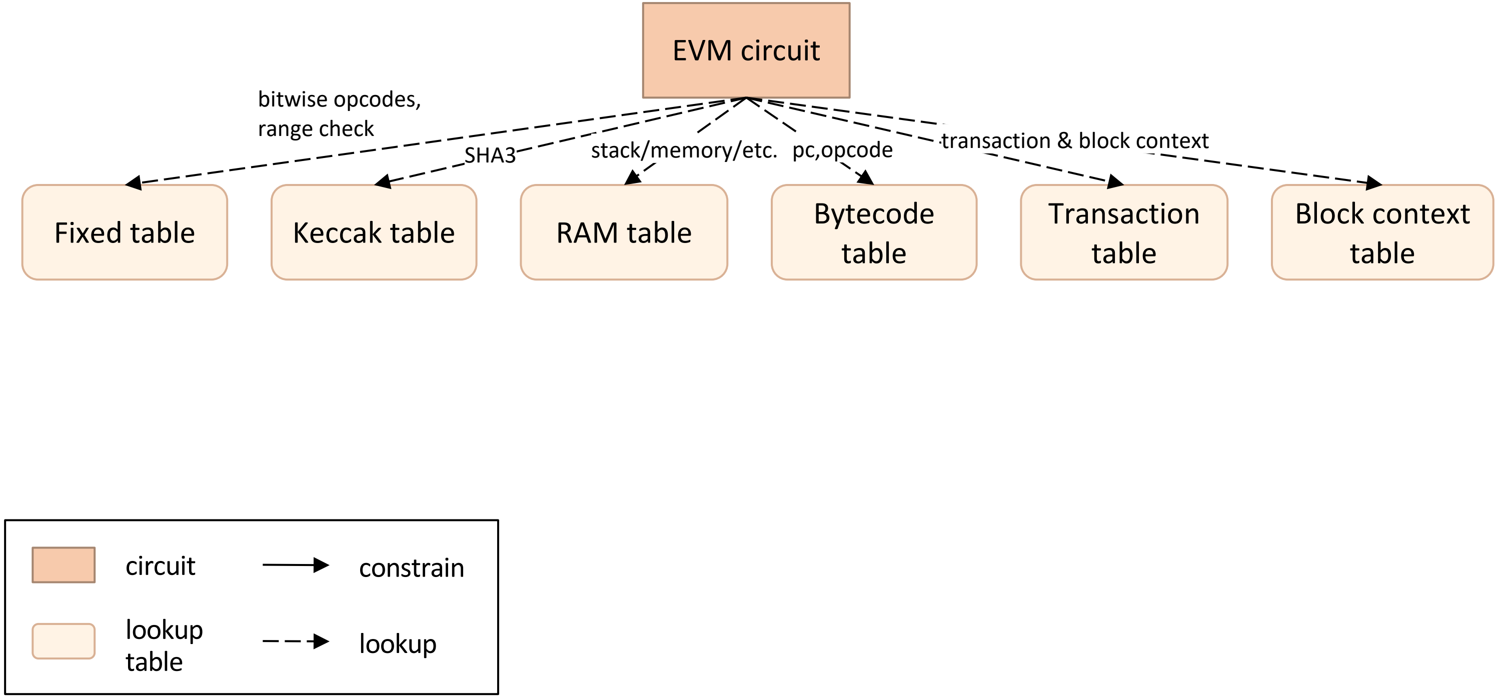
- **Opcode specific witness**



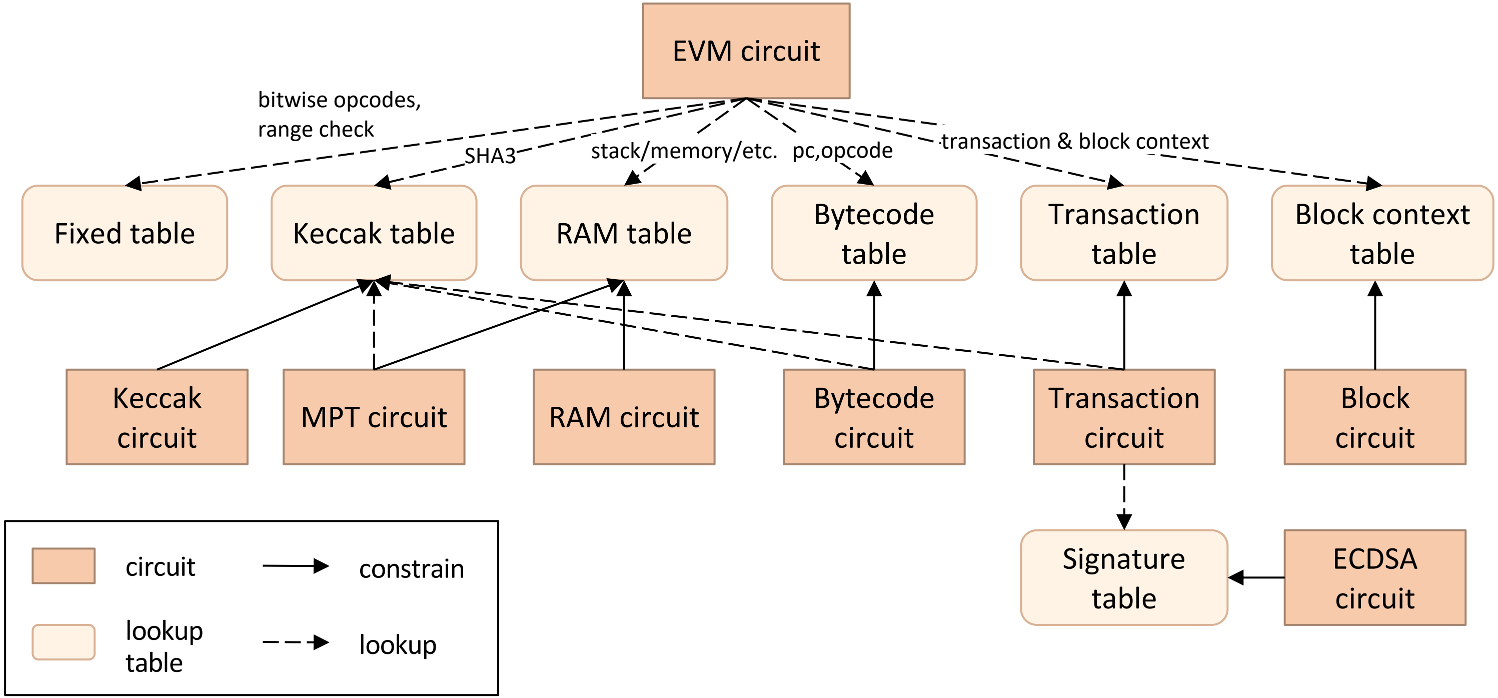
The architecture of zkEVM circuits



The architecture of zkEVM circuits

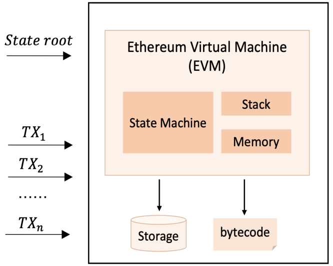


The architecture of zkEVM circuits



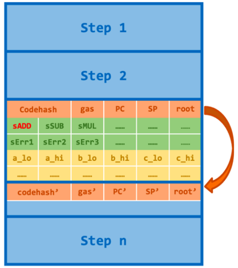
The workflow of zero-knowledge proof

Program



R1CS
Plonkish
AIR

Constraints



- **Step context**

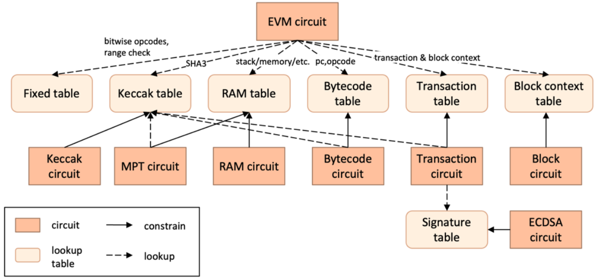
```
sADD * (pc' – pc – 1) == 0
sADD * (sp' – sp – 1) == 0
sADD * (gas' – gas – 3) == 0
```

- **Case switch**

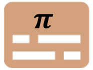
```
sADD * (1 – sADD) == 0
sMUL * (1 – sMUL) == 0
      ...
sADD + sMUL + ... + sERRk == 1
```

- **Opcode specific witness**

```
sADD*(a_lo+b_lo-c_lo – carry0* 2^128)== 0
sADD*(a_hi+b_hi+carry0-c_hi – carry1*2^128)== 0
```

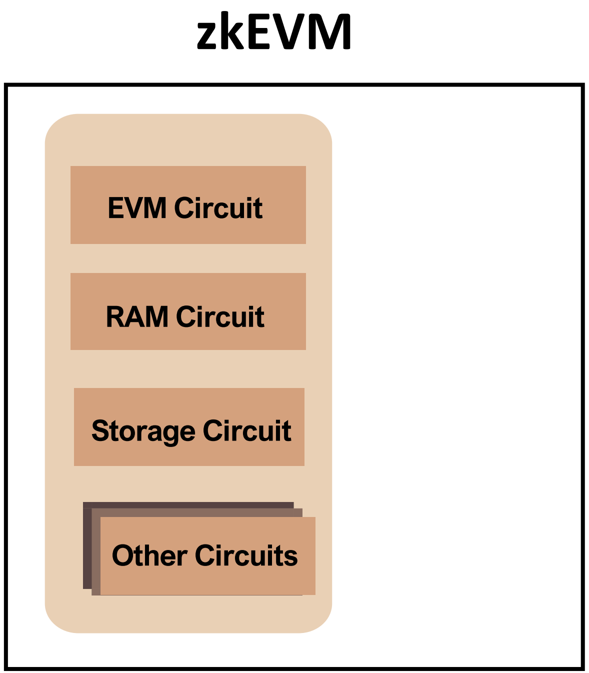


Proof



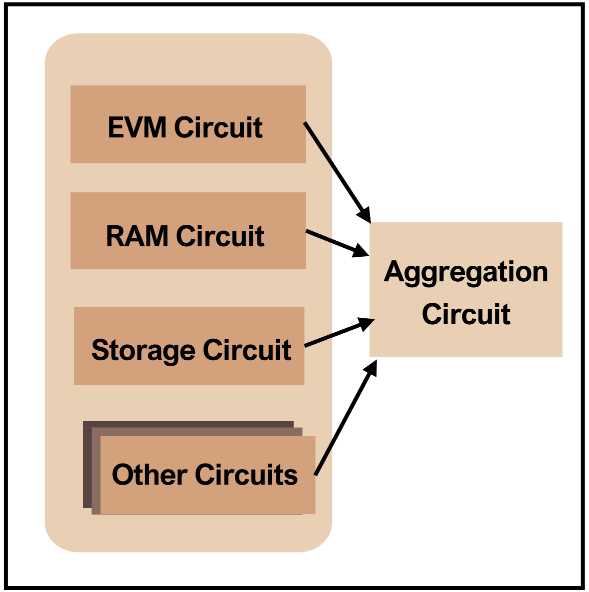
Plonk IOP
+
KZG

The proof system for zkEVM



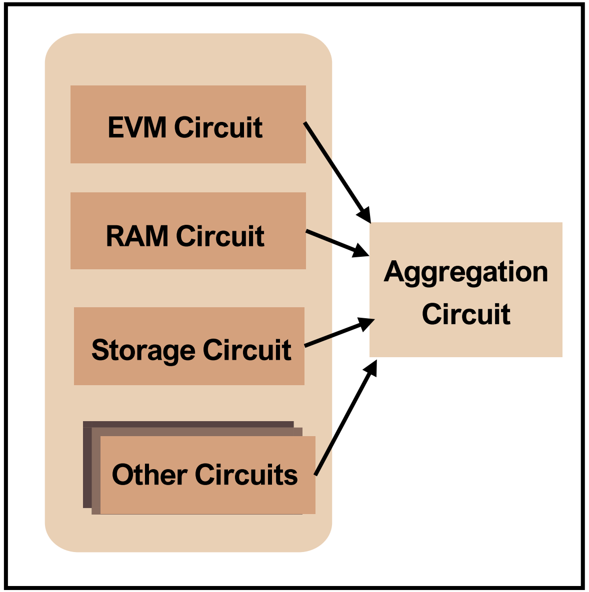
The proof system for zkEVM

zkEVM



Two-layer architecture

zkEVM

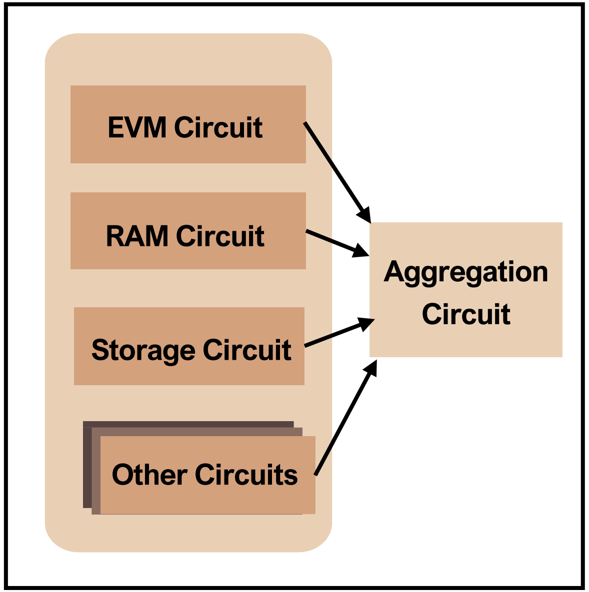


- The first layer needs to handle large computation
  - Custom gate, Lookup support ("expressive", customized)
  - Hardware friendly prover (parallelizable, low peak memory)
  - The verification circuit is small
  - Transparent or Universal trusted setup
- Some promising candidates
  - Plonky2/Starky /eSTARK
  - Halo2/Halo2-KZG
  - New IOP without FFTs (i.e. HyperPlonk, Plonk without FFT)
  - If Spartan/Virgo/… (sumcheck based) or Nova can support Plonkish

Two-layer architecture

Scroll

zkEVM

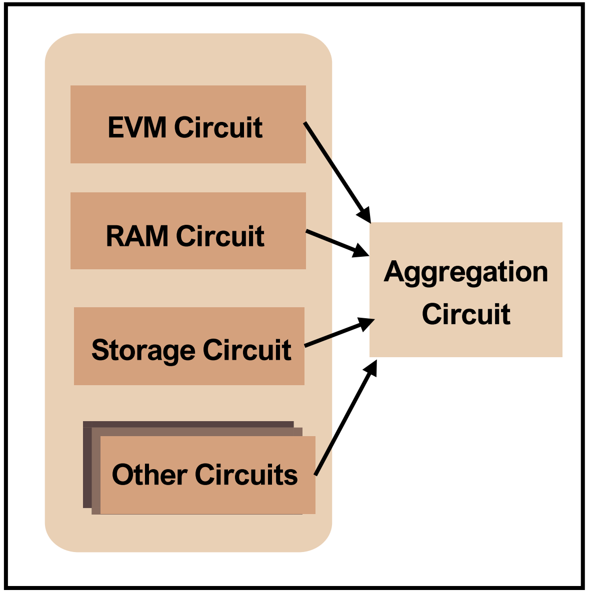


- The second layer needs to be verifier efficient (in EVM)
  - Proof is efficiently verifiable on EVM (small proof, low gas cost)
  - Prove the verification circuit of the former layer efficiently
  - Ideally, hardware friendly prover
  - Ideally, transparent or universal trusted setup
- Some promising candidates
  - Groth16
  - Plonk with very few columns
    - KZG/Fflonk/Keccak FRI (larger code rate)

Two-layer architecture

Scroll

zkEVM

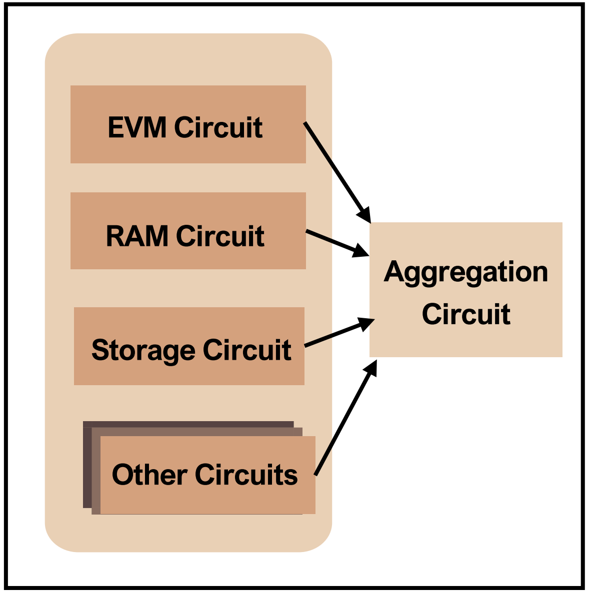


- **The first layer is Halo2-KZG** (Poseidon hash transcript)
  - Custom gate, Lookup support
  - Good enough prover performance (GPU prover)
  - The verification circuit is “small”
  - Universal trusted setup
- **The second layer is Halo2-KZG** (Keccak hash transcript)
  - Custom gate, Lookup support (express non-native efficiently)
  - Good enough prover performance (GPU prover)
  - The final verification cost can be configured to be really small

The layout

Scroll

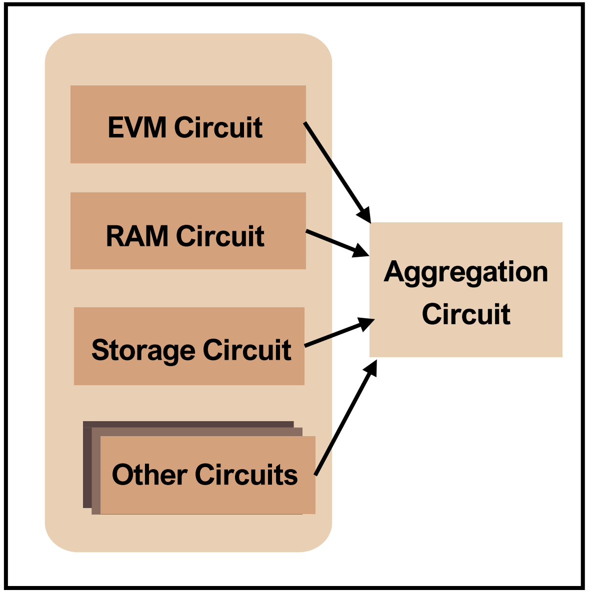
zkEVM



- The first layer needs to be "expressive"
  - EVM circuit has **116 columns, 2496 custom gates, 50 lookups**
  - Highest custom gate degree: 9
  - For 1M gas, EVM circuit needs **2^18 rows** (more gas, more rows)
- The second layer needs to aggregate proofs into one proof
  - Aggregation circuit has **23 columns, 1 custom gate, 7 lookups**
  - Highest custom gate degree: 5
  - For aggregating EVM, RAM, Storage circuits, it needs **2^25 rows**

The performance

zkEVM

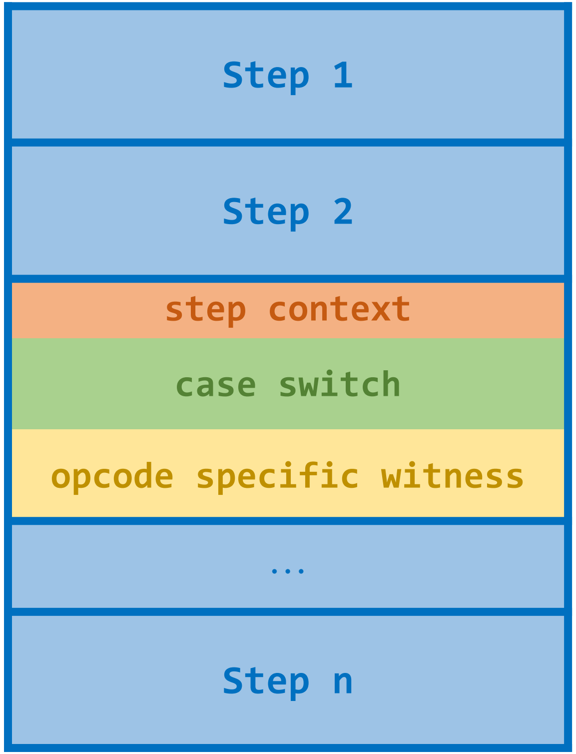


- Our GPU prover optimization
  - MSM, NTT and quotient kernel
  - Pipeline and overlap CPU and GPU computation
  - Multi-card implementation, memory optimization
- The Performance
  - For EVM circuit
    - CPU prover takes 270.5s, GPU prover takes **30s (9x speedup!)**
  - For Aggregation circuit
    - CPU prover takes 2265s, GPU prover takes **149s (15x speedup!)**
  - For 1M gas, first layer takes 2 minutes, second layer takes 3 minutes

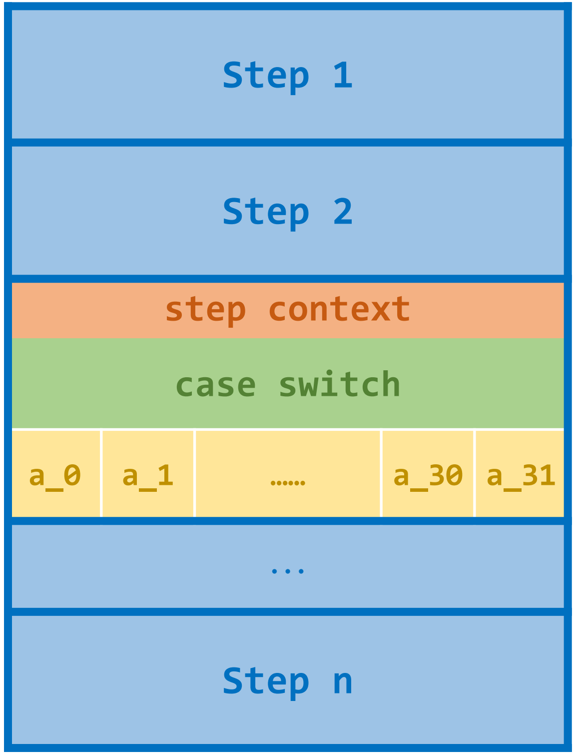
Outline

- Background & motivation
- Build a zkEVM from scratch
- Interesting research problems
- Other applications using zkEVM

Circuit - Randomness



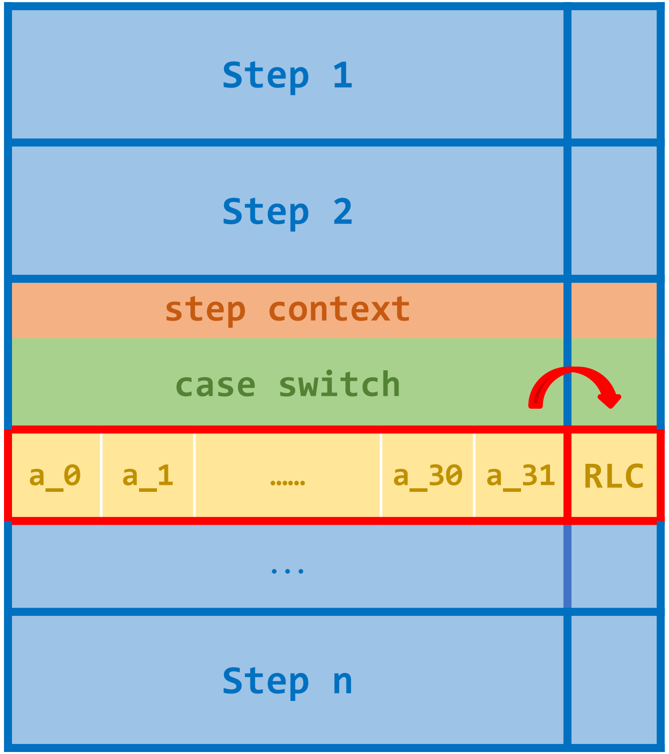
Circuit - Randomness



- Break down 256-bit word into 32 8-bit limbs.

$$A = a_0 + a_1 * 256 + a_2 * 256^2 + \cdots + a_{31} * 256^{31}$$

Circuit - Randomness



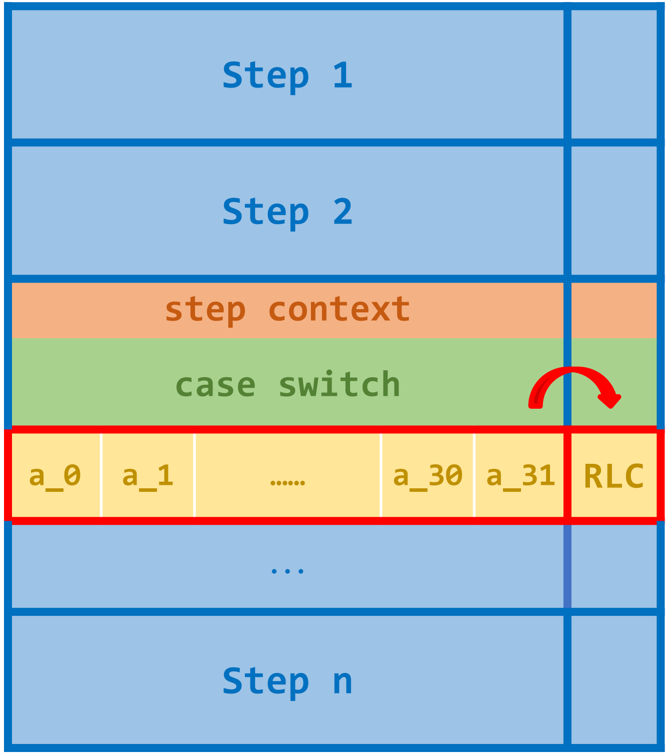
- Break down 256-bit word into 32 8-bit limbs.

$$A = a_0 + a_1 * 256 + a_2 * 256^2 + \cdots + a_{31} * 256^{31}$$

- Encode EVM word using RLC (Random Linear Combination)

$$A_{RLC} \equiv a_0 + a_1 * \theta + a_2 * \theta^2 + \cdots + a_{31} * \theta^{31} \pmod{F_p}$$

Circuit - Randomness



- Break down 256-bit word into 32 8-bit limbs.

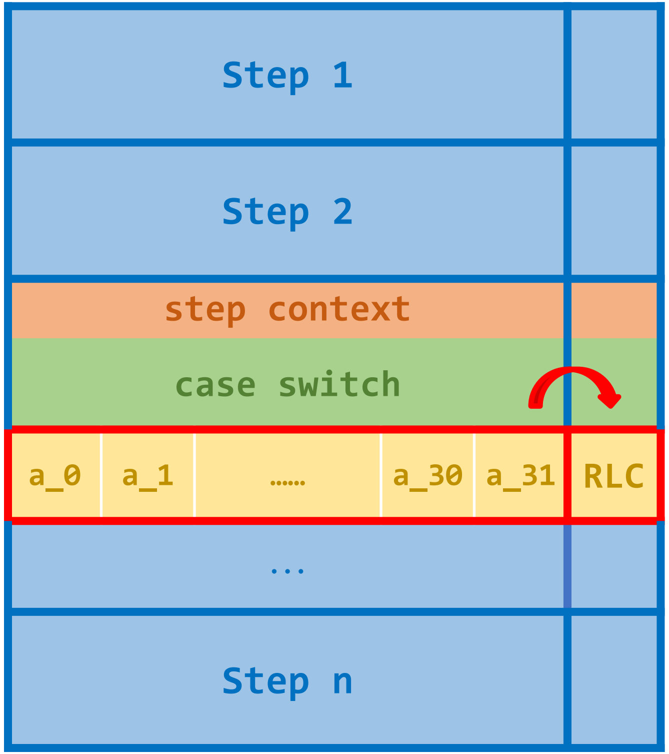
$$A = a_0 + a_1 * 256 + a_2 * 256^2 + \cdots + a_{31} * 256^{31}$$

- Encode EVM word using RLC (Random Linear Combination)

$$A_{RLC} \equiv a_0 + a_1 * \theta + a_2 * \theta^2 + \cdots + a_{31} * \theta^{31} \pmod{F_p}$$

- θ should be computed after $a_0, \ldots, a_{31}$ are fixed
  - Multi-phase prover: synthesis part of witness, derive witness

Circuit - Randomness



- Break down 256-bit word into 32 8-bit limbs.

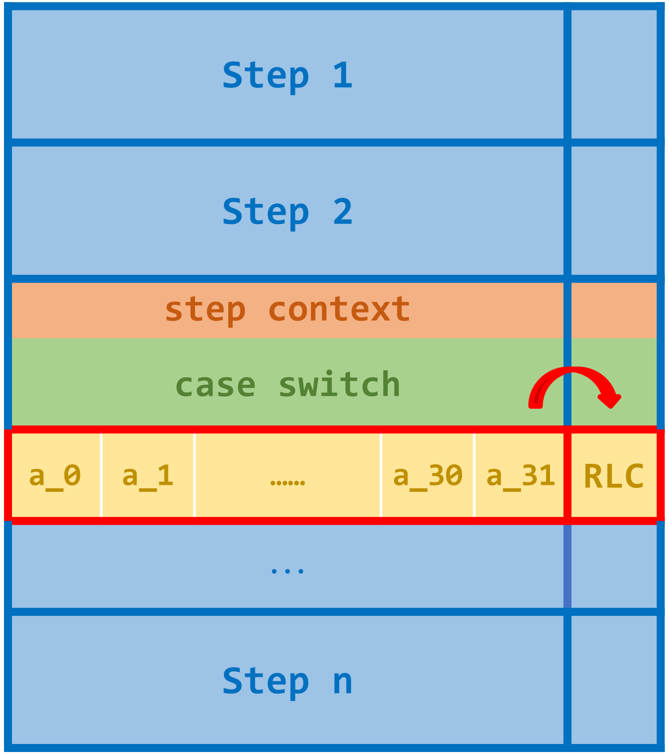
$$A = a_0 + a_1 * 256 + a_2 * 256^2 + \cdots + a_{31} * 256^{31}$$

- Encode EVM word using RLC (Random Linear Combination)

$$A_{RLC} \equiv a_0 + a_1 * \theta + a_2 * \theta^2 + \cdots + a_{31} * \theta^{31} \pmod{F_p}$$

- θ should be computed after $a_0, \ldots, a_{31}$ are fixed
  - Multi-phase prover: synthesis part of witness, derive witness
- RLC is useful in many places
  - Compress EVM word into one value
  - Encode dynamic length input
  - Lookup layout optimization

Circuit - Randomness



- Break down 256-bit word into 32 8-bit limbs.

$$A = a_0 + a_1 * 256 + a_2 * 256^2 + \cdots + a_{31} * 256^{31}$$

- Encode EVM word using RLC (Random Linear Combination)

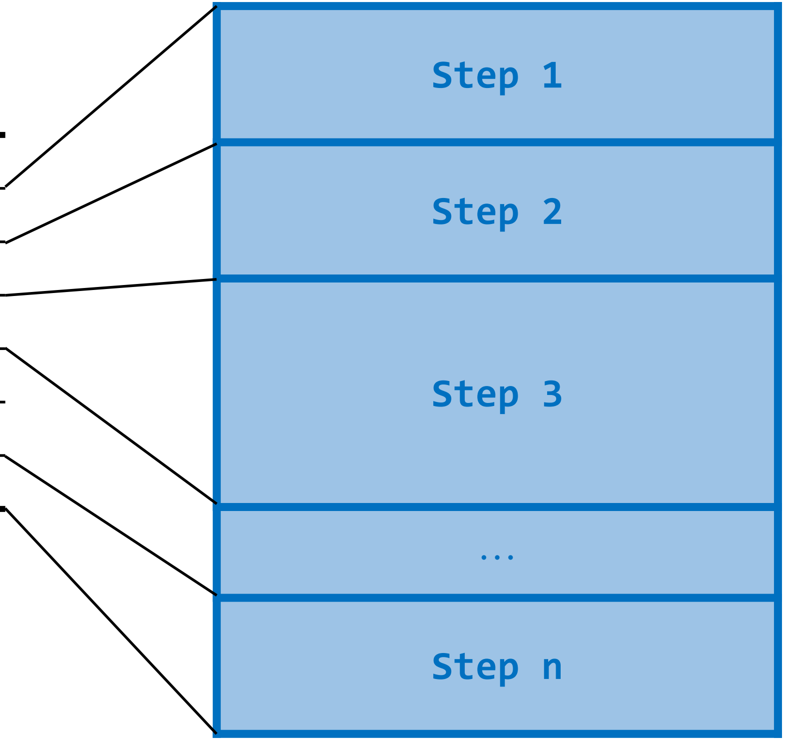
$$A_{RLC} \equiv a_0 + a_1 * \theta + a_2 * \theta^2 + \cdots + a_{31} * \theta^{31} \pmod{F_p}$$

- θ should be computed after $a_0, \ldots, a_{31}$ are fixed
  - Multi-phase prover: synthesis part of witness, derive witness
- **RLC is useful in many places, remove it?**
  - Compress EVM word into one value → high, low for EVM word
  - Encode dynamic length input → fixed chunk, dynamic times
  - Lookup layout optimization

Circuit - Layout

Execution trace

step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN



Circuit - Layout

Execution trace

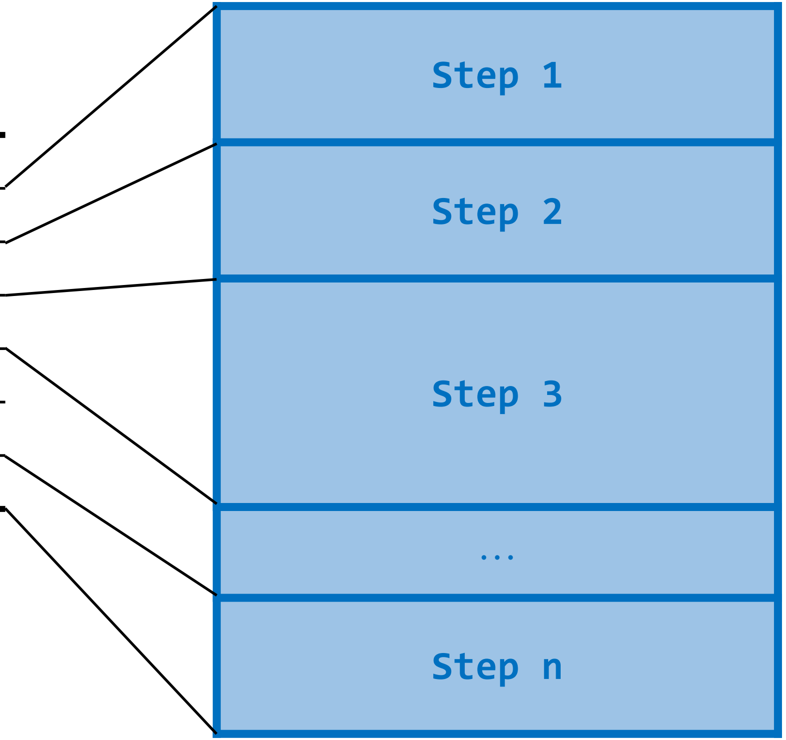
step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN

- The execution trace is dynamic
  - → enable different constraints
  - → permutation is not fixed
  - → hard to use standard gates

Circuit - Layout

Execution trace

step	**opcode**
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN

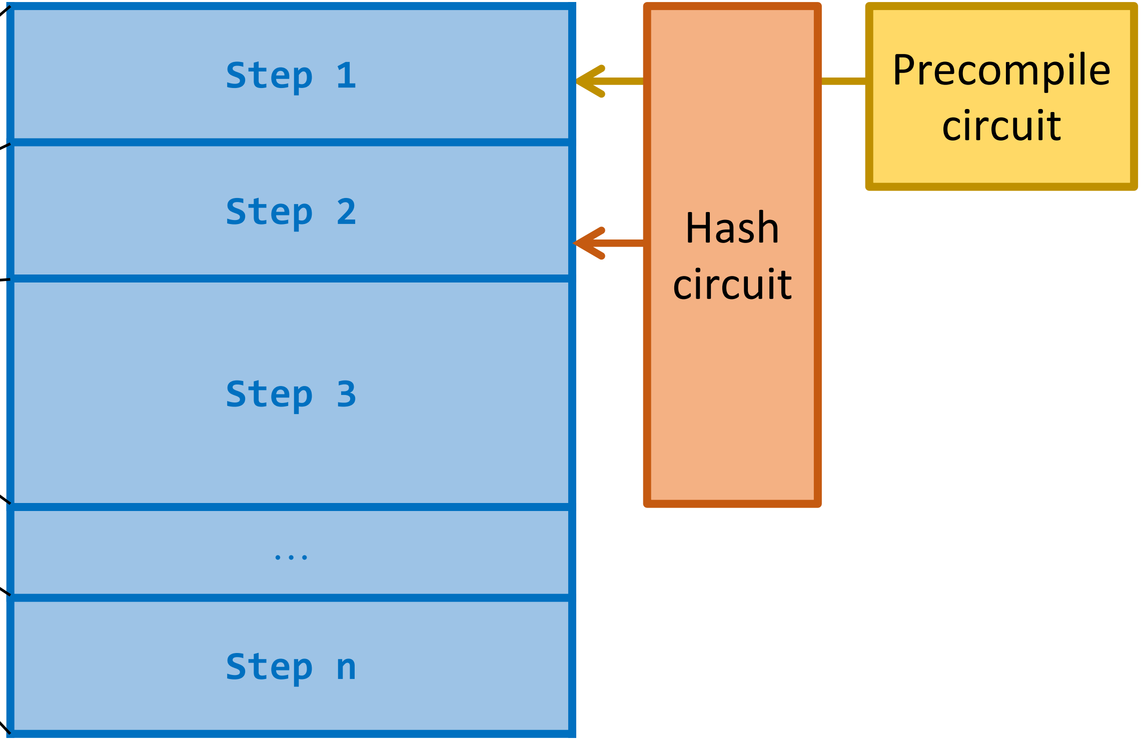


- The execution trace is dynamic
  - → enable different constraints
  - → permutation is not fixed
  - → hard to use standard gates
- **Better way to layout?**
  - We have 2000+ custom gates
  - Different rotation to access cells

Circuit - Dynamic size

Execution trace

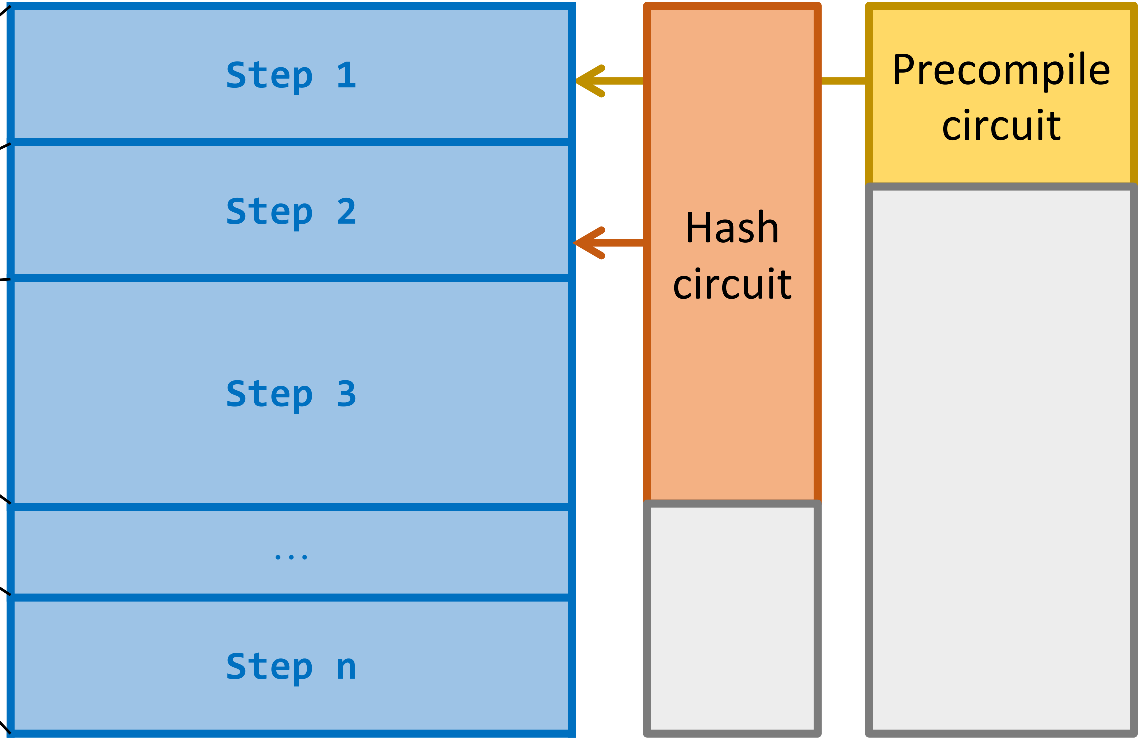
step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN



Circuit - Dynamic size

Execution trace

step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN



Circuit - Dynamic size

Execution trace

step	opcode
1	PUSH1 80
2	PUSH1 40
3	ADD
...	...
...	...
n	RETURN

- Some bad influences

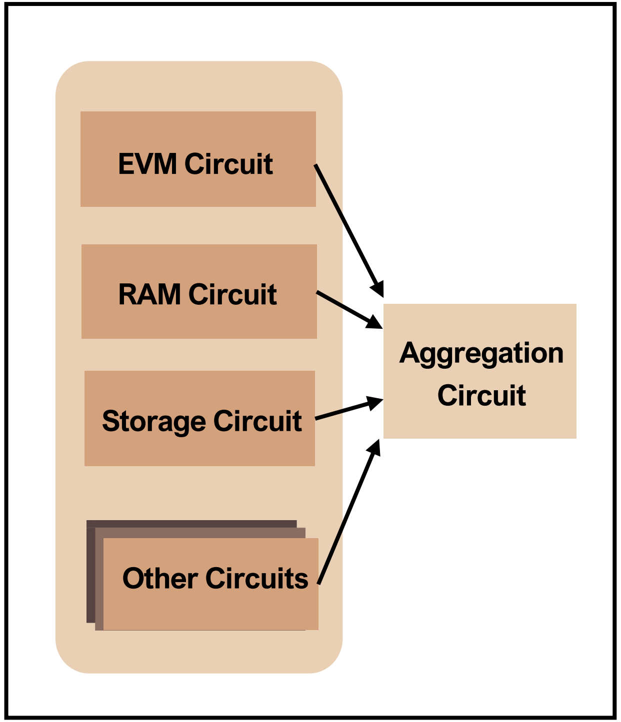
  i.e. Maximum number of Keccaks

- i.e. Mload is more costly (more rows)

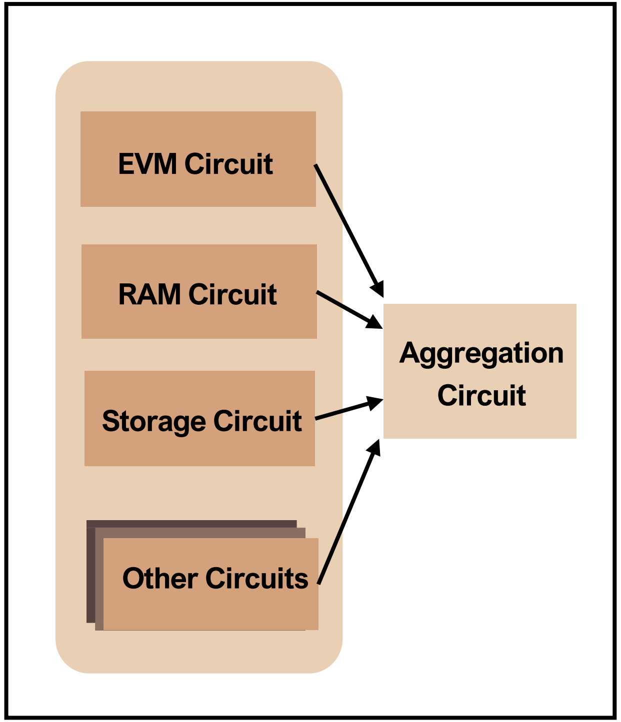
  i.e. Pay larger proving cost for padding

- **Can we make zkEVM dynamic?**

Prover – Hardware & Algorithm



Prover – Hardware & Algorithm

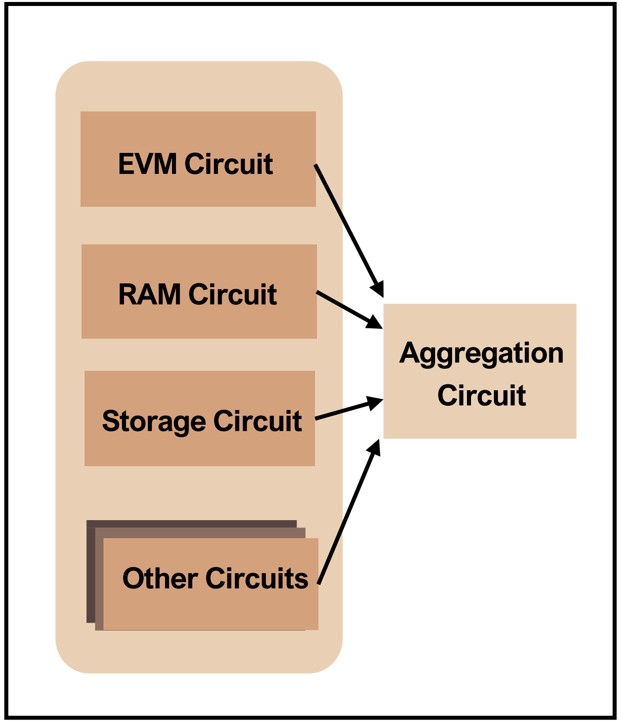


- Our prover
  - GPU can make MSM & NTT really fast

    Bottleneck moves to witness generation & data copy
  - Need large CPU memory (1TB -> 300GB+)

- **Hardware friendly prover?**
  - Parallelizable & Low peak memory
  - Don't ignore the witness generation
  - Run on cheap machines, more decentralized

Prover – Hardware & Algorithm

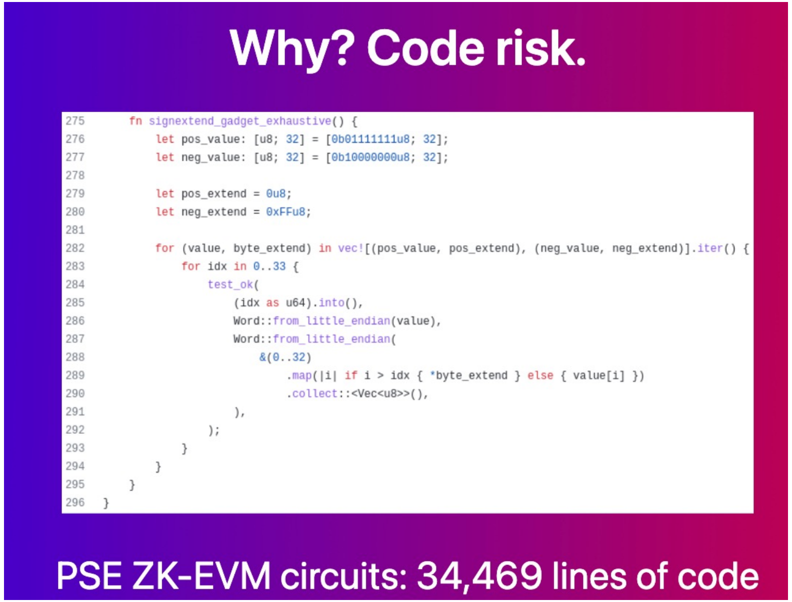


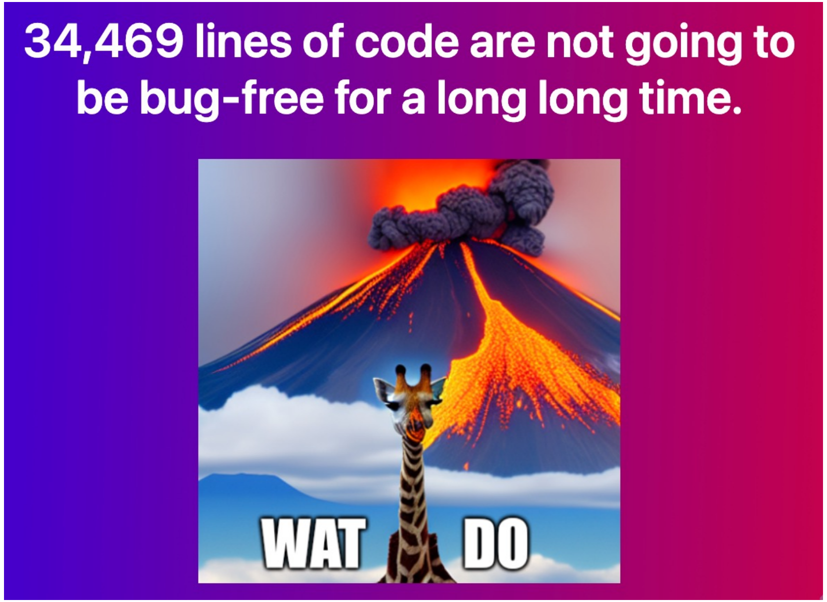
- **Best way to compose different proof system?**
  - The first layer needs to be “expressive”
  - The second layer needs to be verifier efficient (in EVM)
  - **Should we move to smaller field?**

    (Breakdown/FRI with Goldilocks, Mersenne prime)
  - **Should we stick to EC-based constructions?**

    (SuperNova, Cyclic elliptic curve with fast MSM)
  - More options waiting for you → Reach out to us!

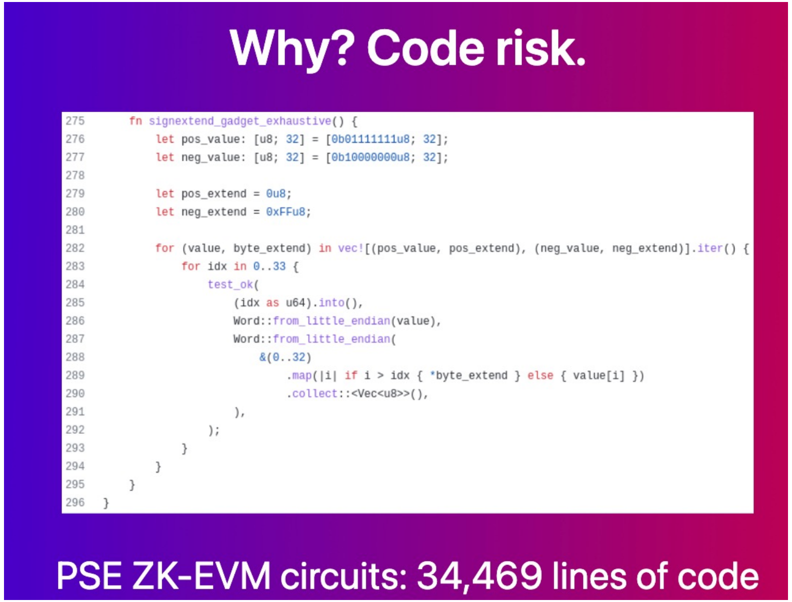
Security – Audit & Soundness





Screenshot From Vitalik

Security – Audit & Soundness



- **The best way to audit zkEVM circuit?**

  (In general, VM circuit based on IR)

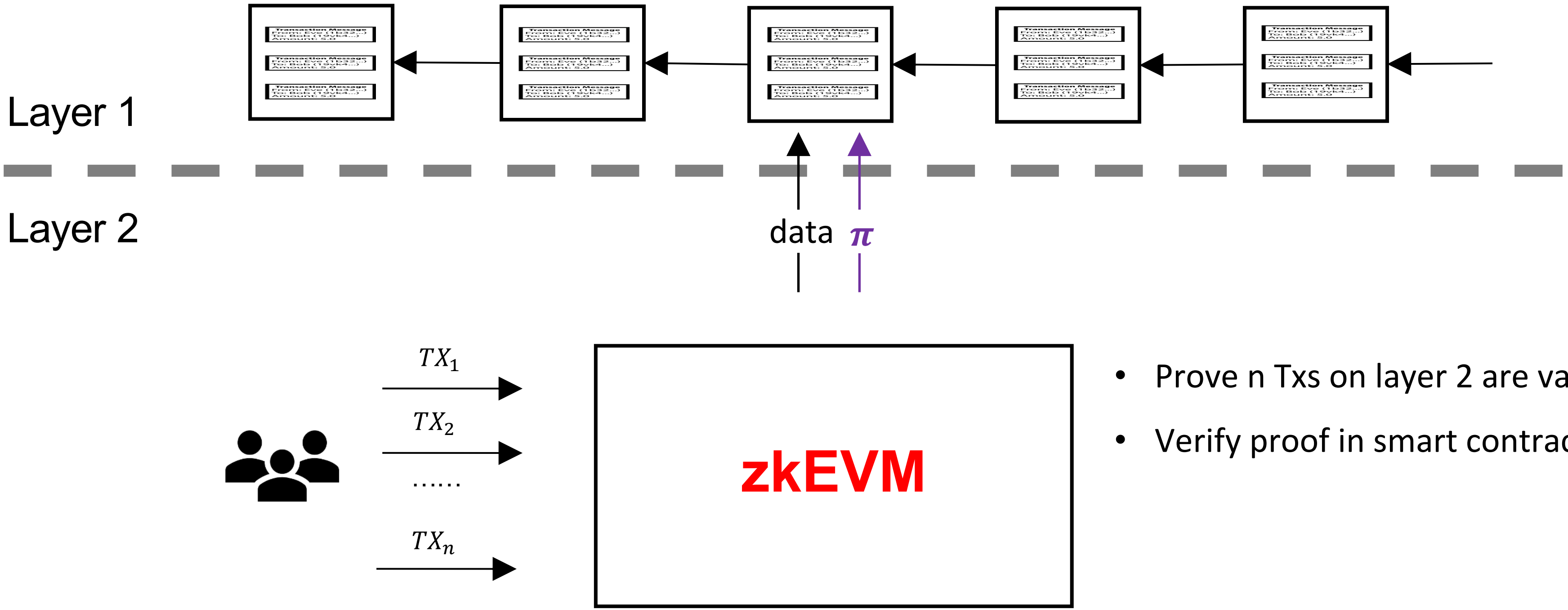
  - Audit Manually
  - Formal verification for some opcodes

Outline

Scroll

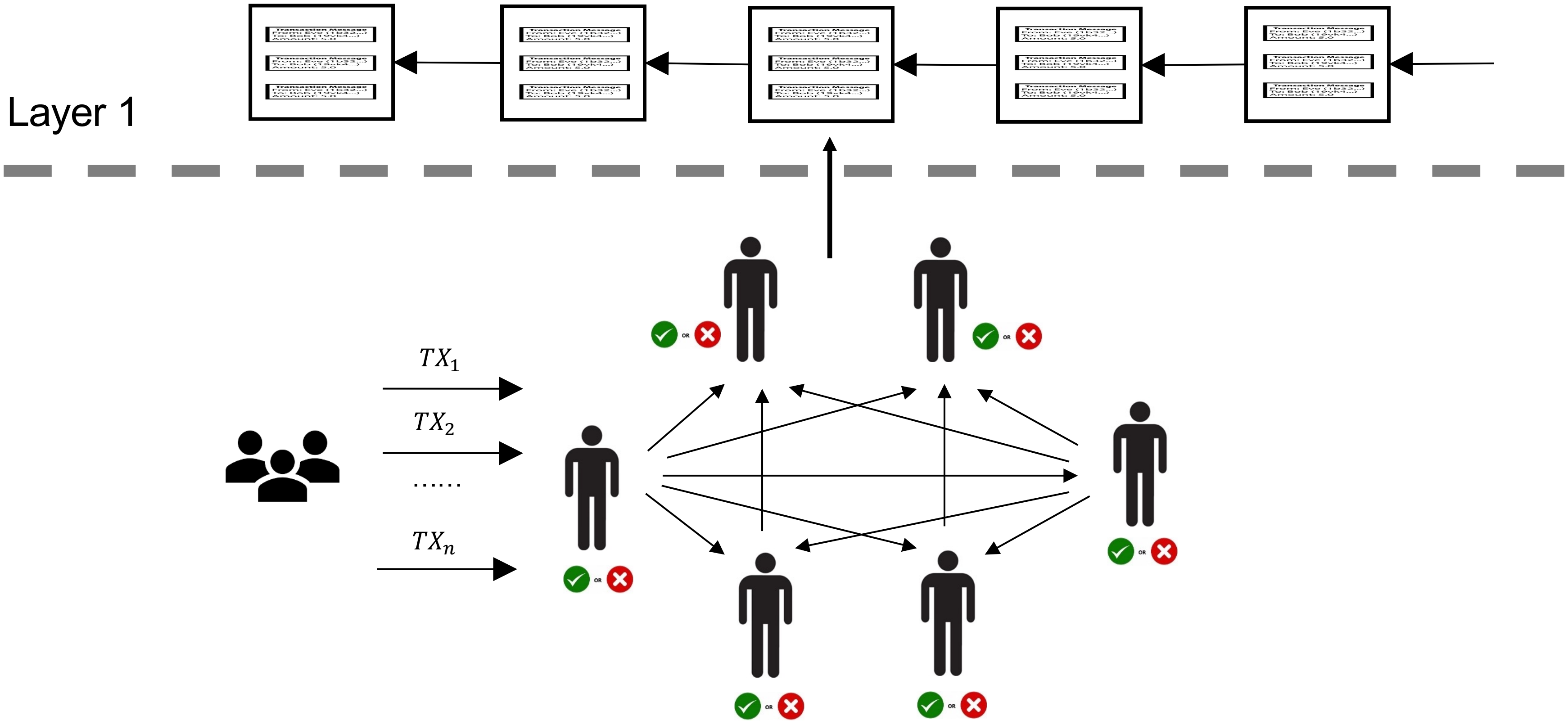
- Background & motivation
- Build a zkEVM from scratch
- Interesting research problems
- Other applications using zkEVM

Applications – zkRollup

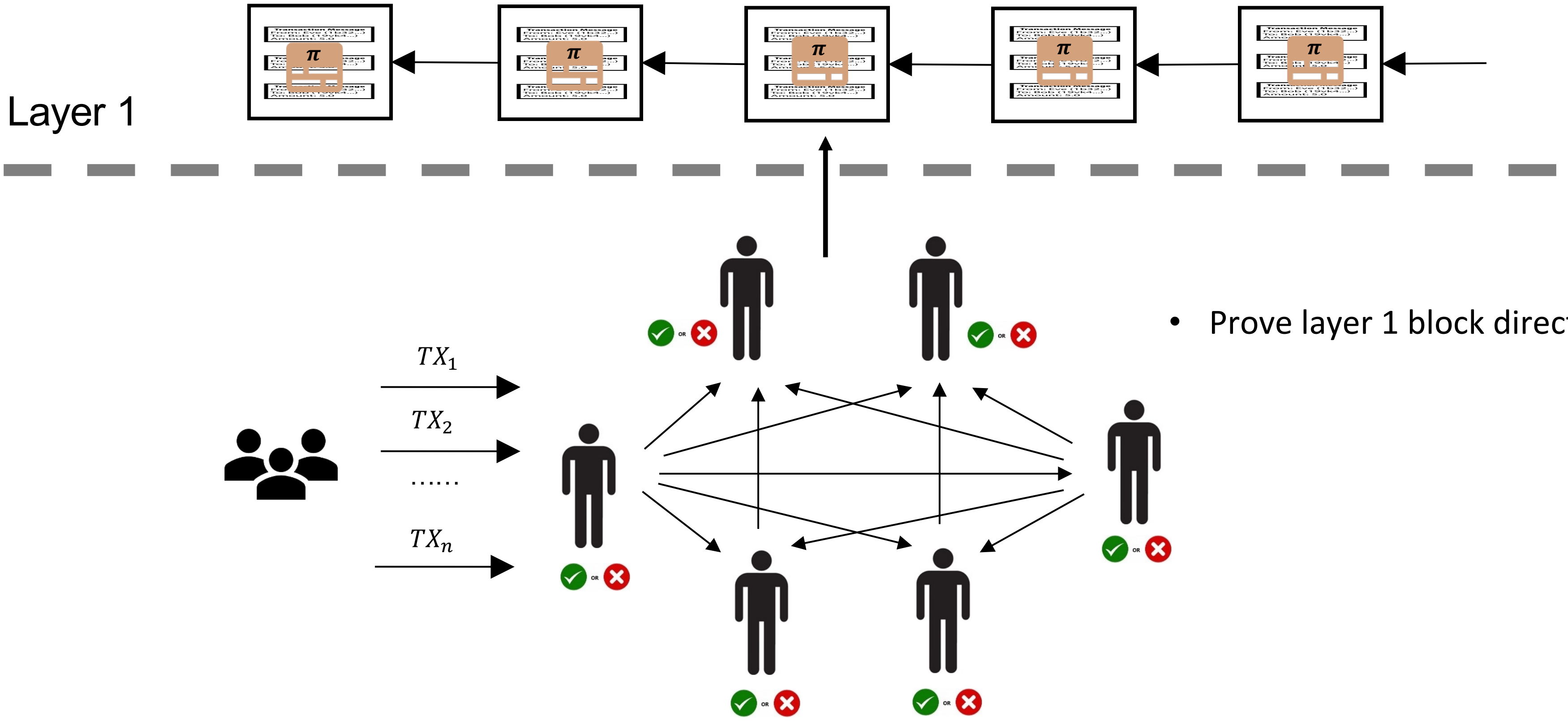


- Prove n Txs on layer 2 are valid
- Verify proof in smart contract

Application - Enshrine blockchain



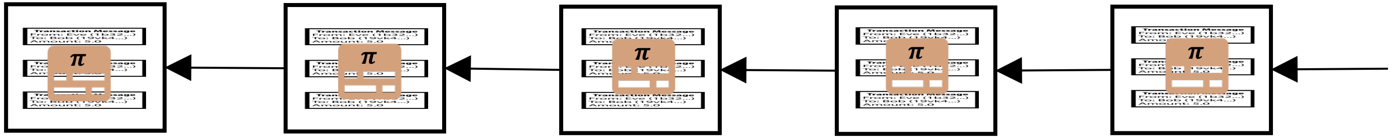
Application - Enshrine blockchain



- Prove layer 1 block directly

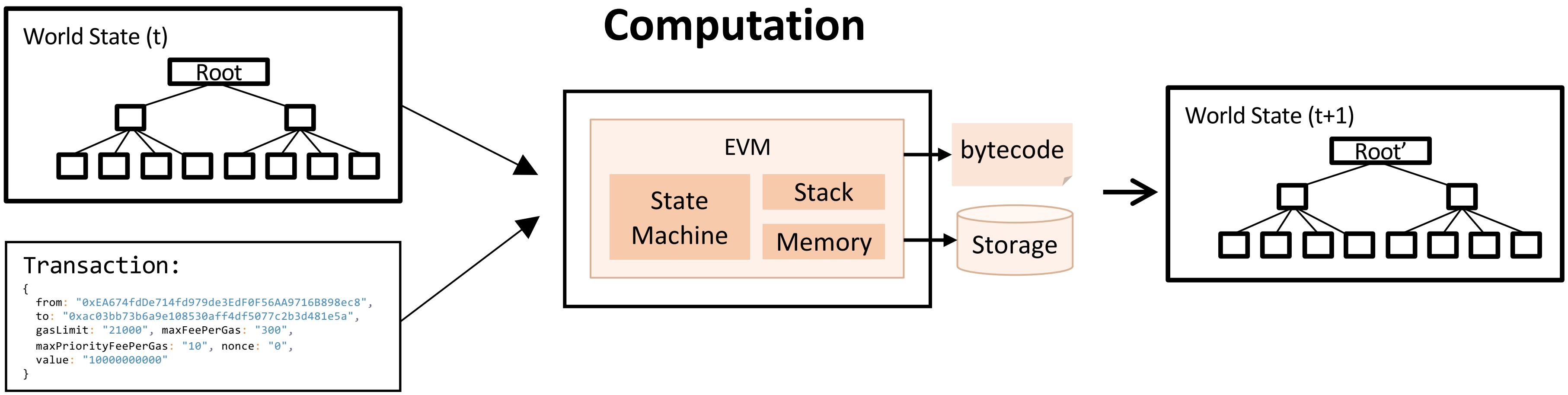
Application - Enshrine blockchain

Layer 1



- Prove layer 1 block directly
- Recursive proof
- One proof for blockchain

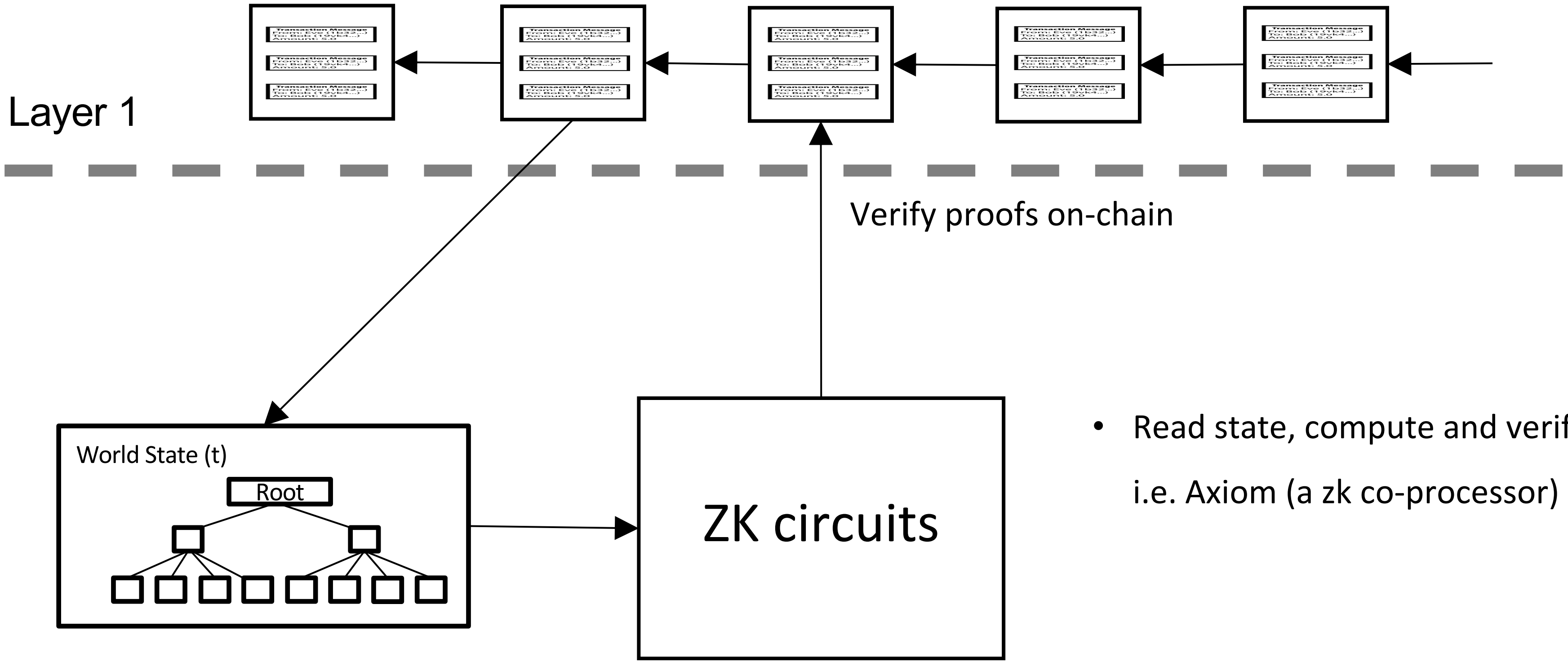
Applications – Proof of exploit



- Prove I know a Tx that can change the state root to state root'

  (Prove I know a bug that can change your balance, etc)

Applications – Attestation (”zk oracle”)



- Read state, compute and verify

  i.e. Axiom (a zk co-processor)

Trustlessly read historic on-chain data
(Need state proof of zkEVM)

Finally, …

- **We are building cool things at Scroll!**
  - Scroll is a general purpose scaling solution for Ethereum based on zkRollup
  - Building a native zkEVM using very advanced circuit arithmetization + proof system
  - Building fast prover through hardware acceleration (GPU in production) + proof recursion
  - We are live on the testnet with a production-level robust infrastructure

- **There are a bunch of interesting problems to be solved!**
  - Protocol design and mechanism design
  - Zk engineer & research for practical efficiency

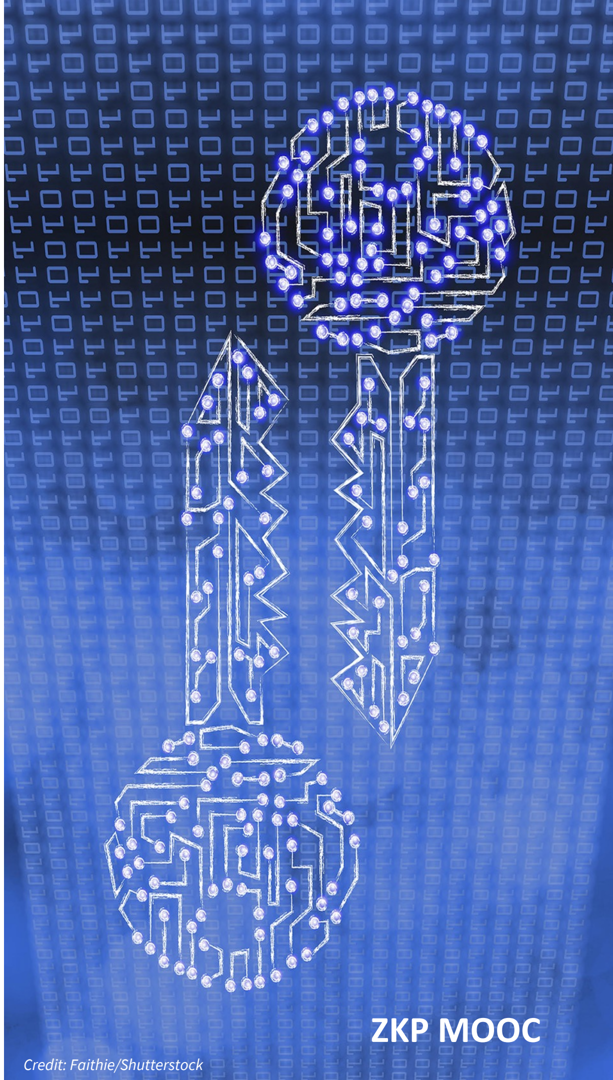
Thank you!

@yezhang1998

Testnet

Discord

Hiring



Credit: Faithie/Shutterstock