Zero-Knowledge Proofs III
Beyond zk-SNARKs
& Accumulators

Oct. 23, 2019

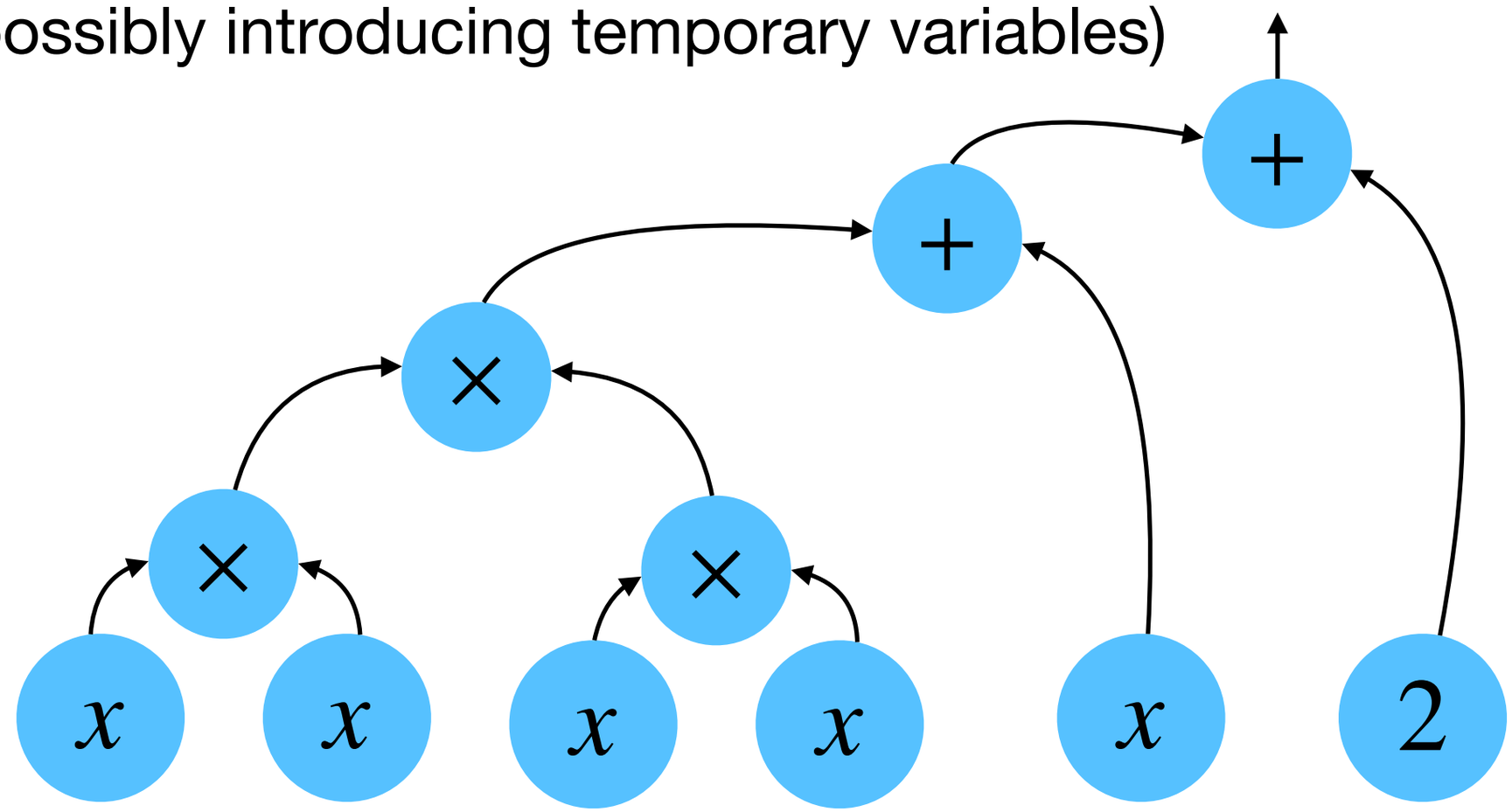
Recap zk-SNARKs

Flattening the computation

Proof: We know x so that $x^4 + x + 2 = 86$ (hint $x = 3$)

- We can verify all basic operations (+,-,*,assignment)
  - We need to represent the computation as a sequence of basic steps (possibly introducing temporary variables)

1. $a = x \cdot x$
2. $b = a \cdot a$
3. $c = b + x$
4. $\text{out} = c + 2$



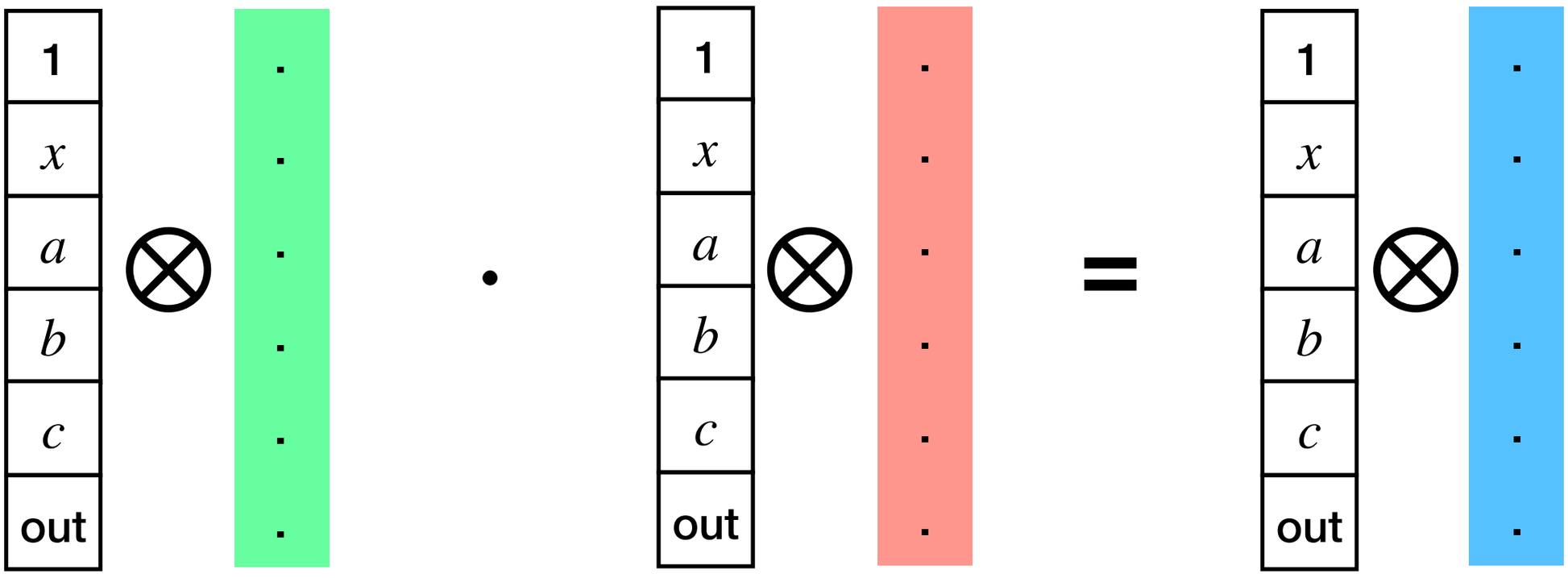
Each operation as vector

L **operator** $(\cdot, +, -)$ R = O

List of all variables:

$1, x, a, b, c$, out

We can generalize all operations using 3 vectors:



Summarized Constraints

Proof: We know x so that $x^4 + x + 2 = 86$ (hint $x = 3$)

1	0	0	1	1	0	0	0	2	0	0	0	0
x	1	0	0	0	1	0	1	0	0	0	0	0
a	0	1	0	0	0	1	0	0	1	0	0	0
b	0	0	0	0	0	0	1	0	0	1	0	0
c	0	0	0	0	0	0	0	1	0	0	1	0
out	0	0	0	0	0	0	0	0	0	0	0	1

1st constraint $x \cdot x = a$

3rd constraint $b + x = c$

Quadratic Assignment Problem

1
x
a
b
c
out

$L_1(t)$
$L_x(t)$
$L_a(t)$
$L_b(t)$
$L_c(t)$
$L_{\text{out}}(t)$

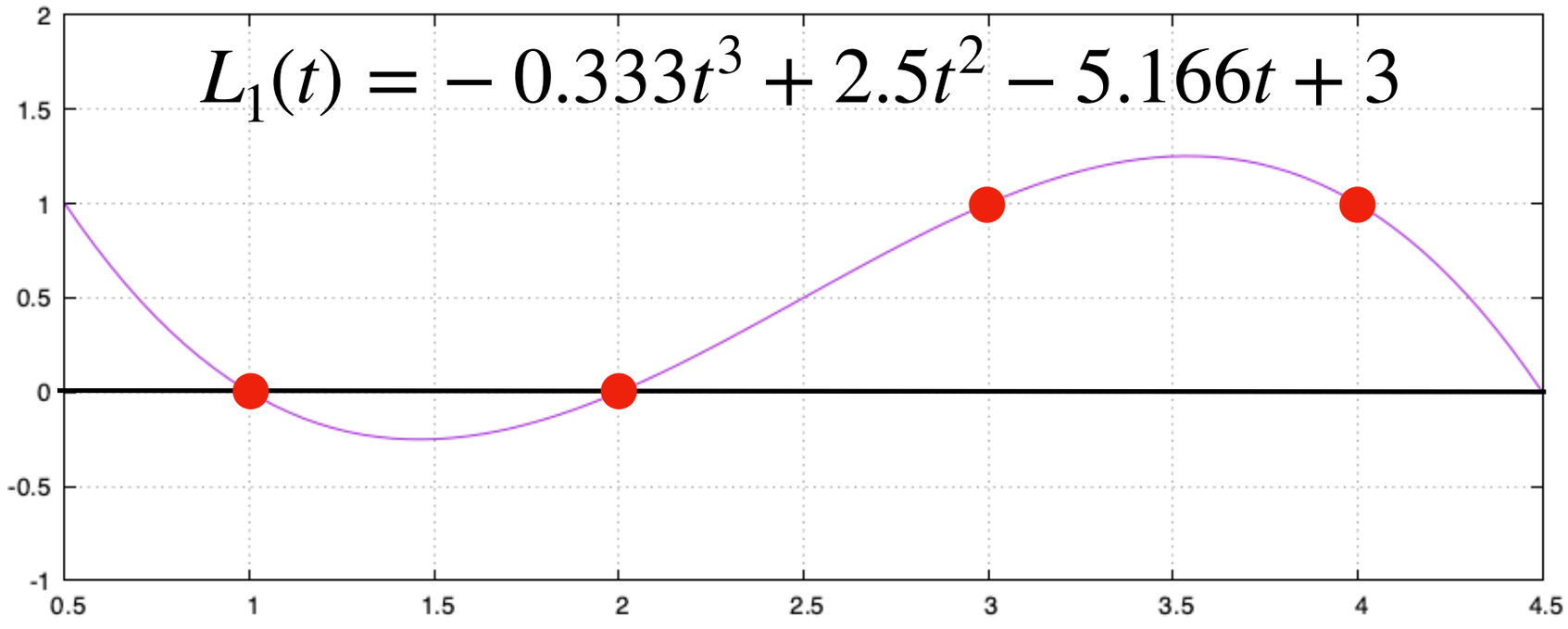
0	0	1	1
1	0	0	0
0	1	0	0
0	0	0	0
0	0	0	0
0	0	0	0

$$L_1(1) = 0$$

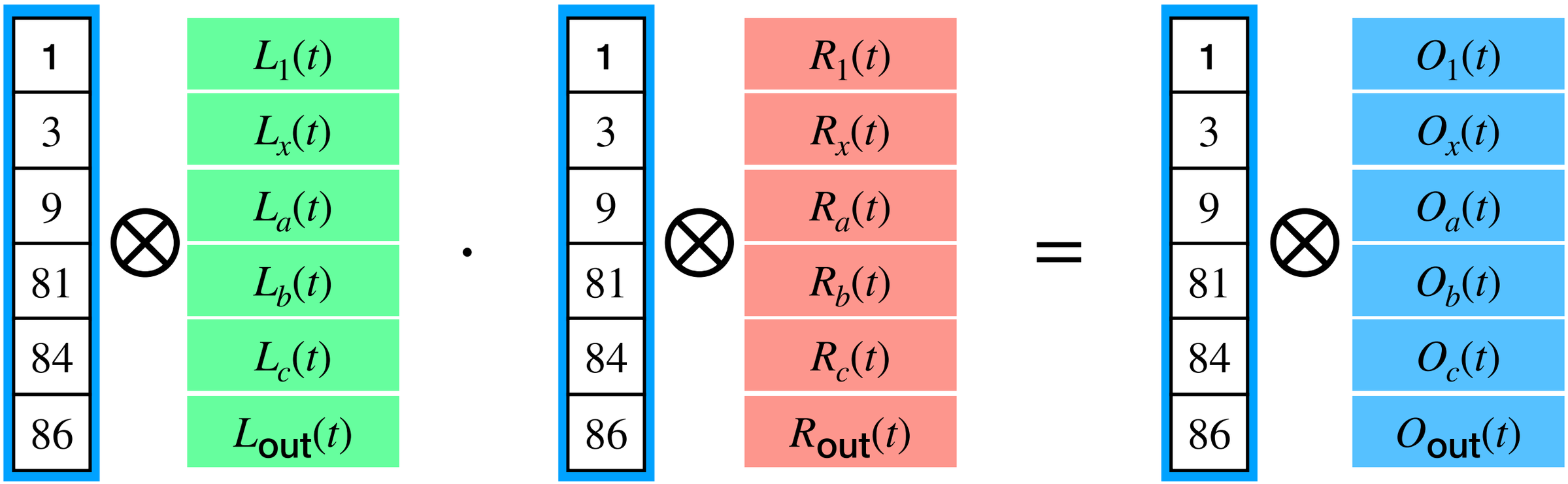
$$L_1(2) = 0$$

$$L_1(3) = 1$$

$$L_1(4) = 1$$



Check all Constraints



$$L(t) \cdot R(t) = O(t)$$

For $t = 1,2,3,4$

Polynomials with same roots

- We compute $X(t) = L(t)R(t) - O(t)$
- We show $Z(t) = (t-1)(t-2)(t-3)(t-4)$ is a divisor of $X(t)$
  - $X(t)$ is 0 at $t = 1,2,3,4$
    - Thus $L(t) \cdot R(t) = O(t)$ at $t = 1,2,3,4$
- Compute $H(t) = \frac{X(t)}{Z(t)}$
  - If the witness were fake, this division leaves a residue
- All that's left to prove is $H(t)Z(t) = X(t)$

To check all constraints

$$X(t) = L(t)R(t) - O(t)$$

$$Z(t) = (t-1)(t-2)(t-3)(t-4)$$

$$H(t) = \frac{X(t)}{Z(t)}$$

- Instead of $H(t)Z(t) = X(t)$, show $H(t)Z(t) - X(t) = 0$ everywhere

- Instead of $H(t)Z(t) - X(t) = 0$ everywhere, pick a secret t and evaluate the 3 functions there (with ECC math)

Summary

- Alice:
  - List an arbitrary computation as a set of basic operations
  - Create $L\_(t), R\_(t), O\_(t)$ polynomials for each input, temporary variables, output and the constant 1
- Bob:
  - Creates the witness vector
  - Computes $L(t) = W \otimes L\_(t), R(t) = \ldots, O(t) = \ldots$
  - Divides $H(t) = X(t)/Z(t)$
- Alice:
  - Evaluates the equation $H(t)Z(t) = X(t)$ at a point of her choosing, accepts if 0

Trusted Setup

- This is done non-interactively if Alice encrypts the point t as $T = tG$, and Bob proves that $H(T)Z(T) - X(T) = 0$
- If Bob can break the encryption (or if he breaks into Alices computer), he can find t
  - knowing at which point Alice evaluates $H(t)Z(t) = X(t)$, he can fake a solution
- Coda, Zerocoin, Zerocash, and others use zk-SNARKS
  - We need to trust that the creators do not collaborate with some users and share the secret value t

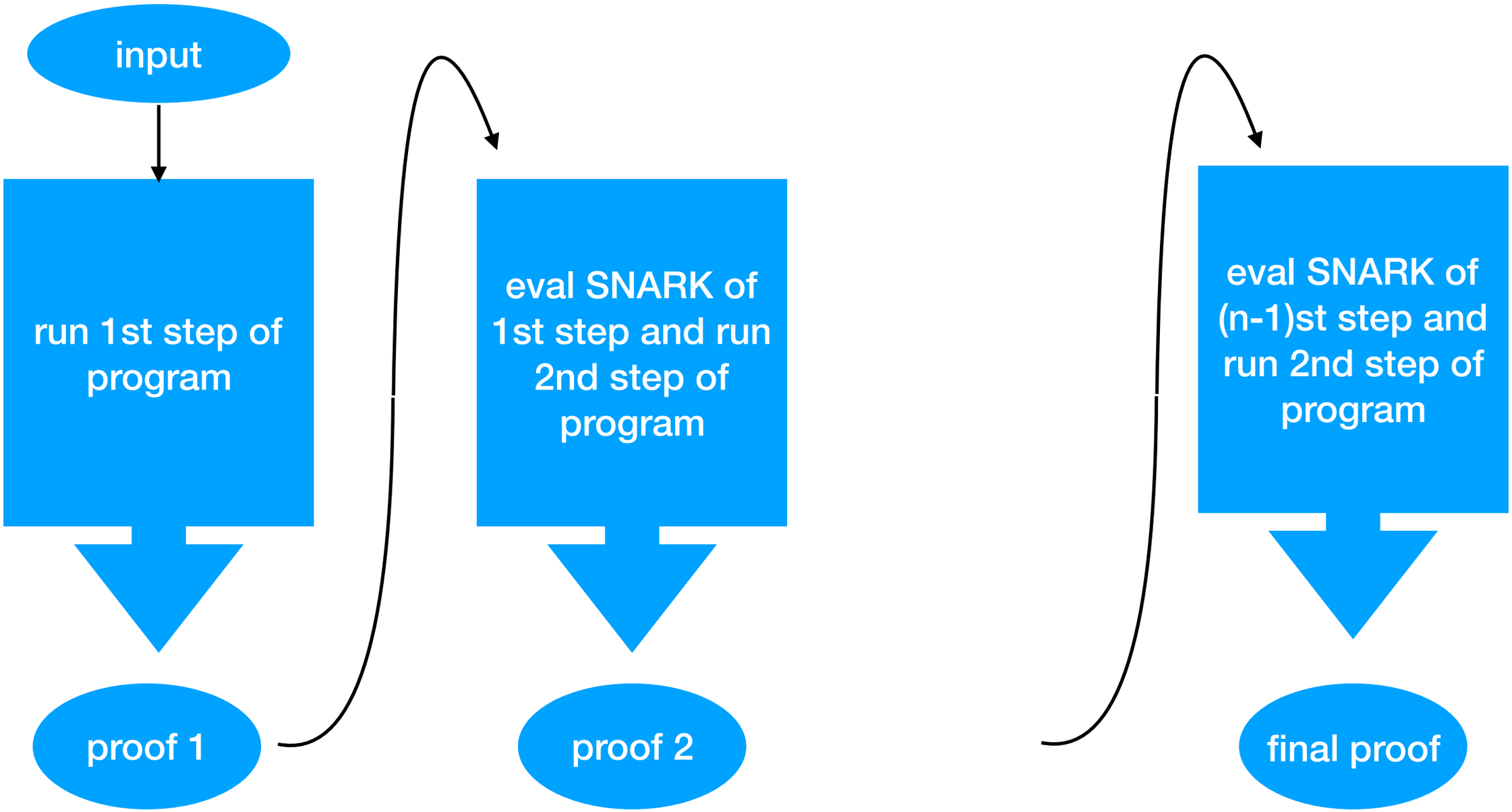
Arbitrary computation

- A zk-SNARK needs to know the computational steps $t = 1,2,3,\ldots,n$ beforehand
- No loops (you need to unravel loops)
- Not Turing complete
- Not well suited for long/complex operations
  - How can we still enable arbitrary computations?

Evaluate a SNARK

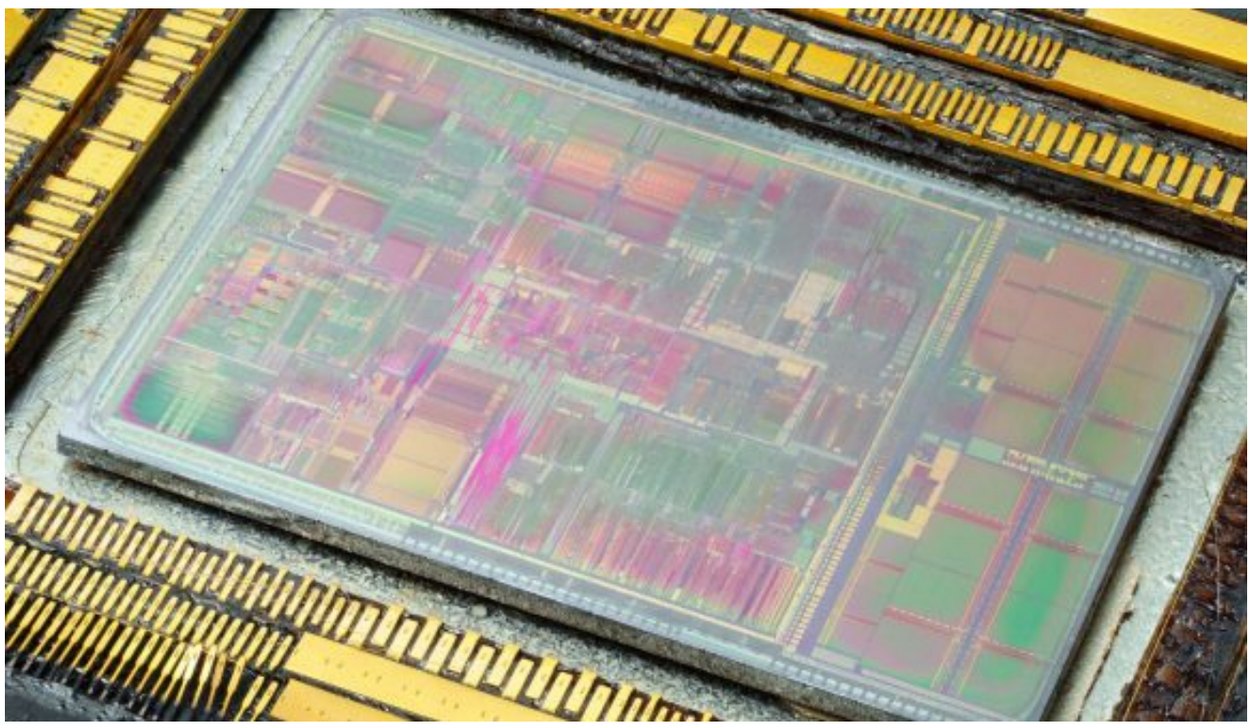
- How do you verify a zk-SNARK?
  - you check whether $H(t)Z(t) - X(t) = 0$ at a random/ secret point
  - This in itself is also a computation I can run in a SNARK

Chaining zk-SNARKs



A universal program

- Any program runs on a CPU
  - The CPU itself (each cycle) is a fixed set of instructions
- why not simulate a CPU as a program?



vnTinyRAM

- Simulate CPU cycle with 3 proofs, namely a proof that

    1. the fetched instruction was executed correctly

    2. the right instruction was fetched from memory

    3. each load from memory retrieves the last value stored there (no one tampered with the memory)

- Side note: Memory consistency is done via Merkel-Trees

vnTinyRAM

“The generated vnTinyRAM circuit implements exactly one cycle of the CPU. It takes as input a previous CPU state, along with a proof that the prior state was valid. It also takes the supposed next state. Because the circuit checks the prior proof and that the transition is valid, feeding the circuit through the SNARK algorithms spits out an updated proof that can then be fed back into the universal circuit again to run the next clock cycle.

You keep doing this, feeding proofs back into the same circuit again to prove the next step, until the program you’re running eventually answers YES (if it wouldn’t answer YES then doing all this is pointless, you’re just burning CPU time). As the exact point at which the program accepts might be sensitive, for privacy reasons you can keep iterating the CPU beyond that time, it just won’t change the answer.”

— Mike Hearn

vnTinyRAM

				128 bits of security $W = 32, K = 16$				
				$\ell = 2$K	$\ell = 4$K	$\ell = 6$K	$\ell = 8$K	$\ell = 10$K
KeyGen	time/T	$n = 100$	$T = 4$K	206.8 ms	232.1 ms	257.5 ms	275.9 ms	306.4 ms
			$T = 8$K	190.9 ms	205.9 ms	216.1 ms	228.9 ms	238.8 ms
			$T = 16$K	195.4 ms	198.1 ms	204.2 ms	213.6 ms	218.3 ms
			$T = 32$K	206.0 ms	208.4 ms	211.2 ms	213.5 ms	223.7 ms
	|pk|/T		$T = 4$K	584.2 KB	653.6 KB	727.1 KB	784.0 KB	876.8 KB
			$T = 8$K	552.4 KB	585.2 KB	618.1 KB	655.1 KB	683.7 KB
			$T = 16$K	539.4 KB	553.9 KB	570.4 KB	586.9 KB	605.5 KB
			$T = 32$K	533.8 KB	541.1 KB	548.3 KB	555.6 KB	563.4 KB
	|vk|		$T = *$	17.0 KB	33.1 KB	49.2 KB	65.3 KB	81.5 KB
Prove	time/T	$n = 100$	$T = 4$K	75.7 ms	86.7 ms	103.4 ms	104.8 ms	133.7 ms
			$T = 8$K	69.2 ms	79.7 ms	97.0 ms	110.4 ms	113.0 ms
			$T = 16$K	89.0 ms	89.1 ms	98.4 ms	99.6 ms	103.3 ms
			$T = 32$K	98.9 ms	98.6 ms	102.3 ms	102.1 ms	114.2 ms
Verify	time	(indep. of T)	$n = 0$	19.0 ms	30.0 ms	40.6 ms	51.2 ms	61.3 ms
			$n = 10$	19.1 ms	30.2 ms	40.7 ms	51.2 ms	61.4 ms
			$n = 10^2$	19.6 ms	30.7 ms	41.3 ms	51.8 ms	61.9 ms
			$n = 10^3$	23.0 ms	34.1 ms	44.7 ms	55.2 ms	65.4 ms
			$n = 10^4$	48.9 ms	60.0 ms	70.6 ms	81.1 ms	91.3 ms

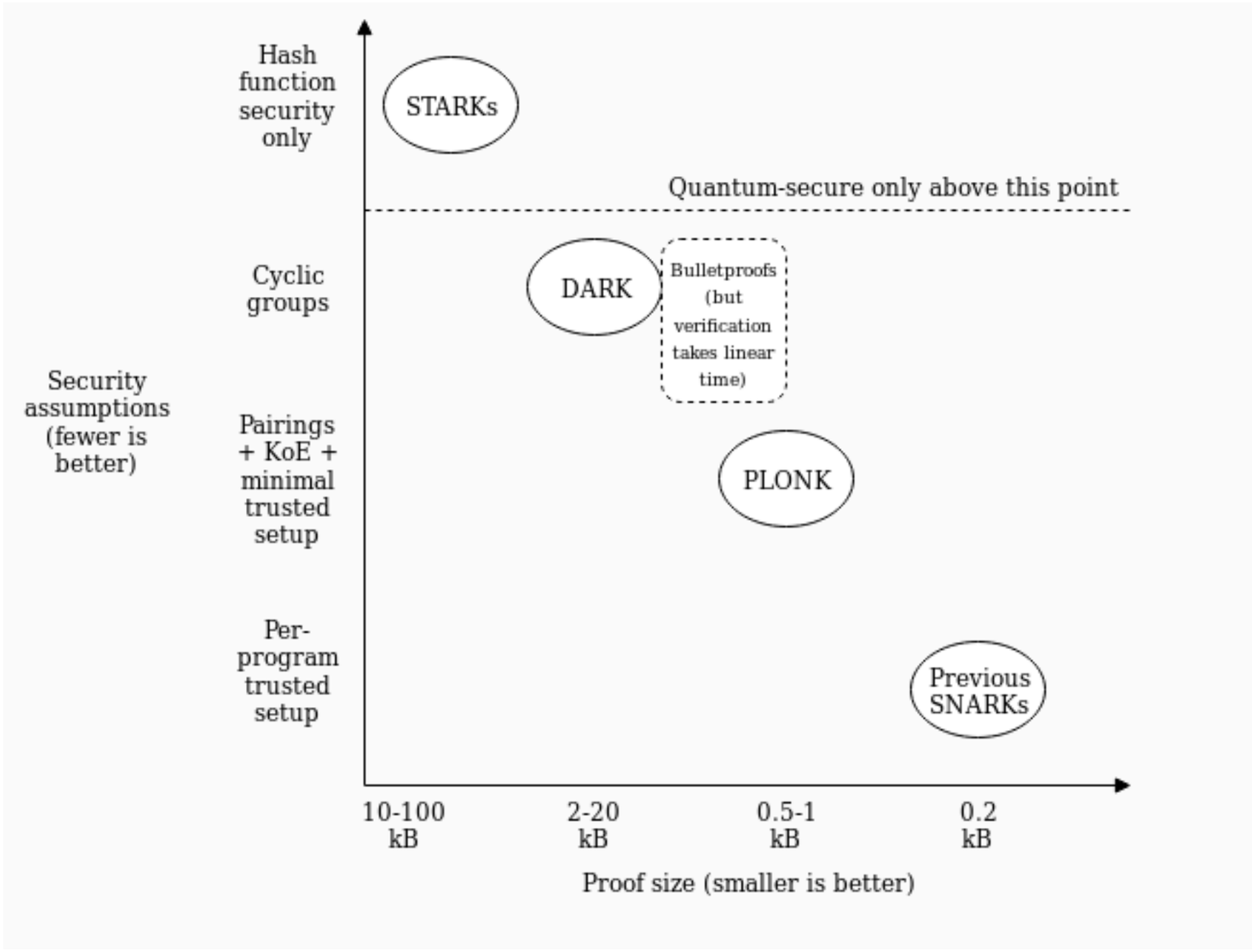
Verification time / CPU cycle: program size l, input size n

vnTinyRAM

- If it can be run on a CPU (anything) it can be run as zk-SNARK
- Verification of any arbitrary computation possible
- Performance is very slow, ~10 sec. for each simulated CPU cycle

Alternatives to zk-SNARKS

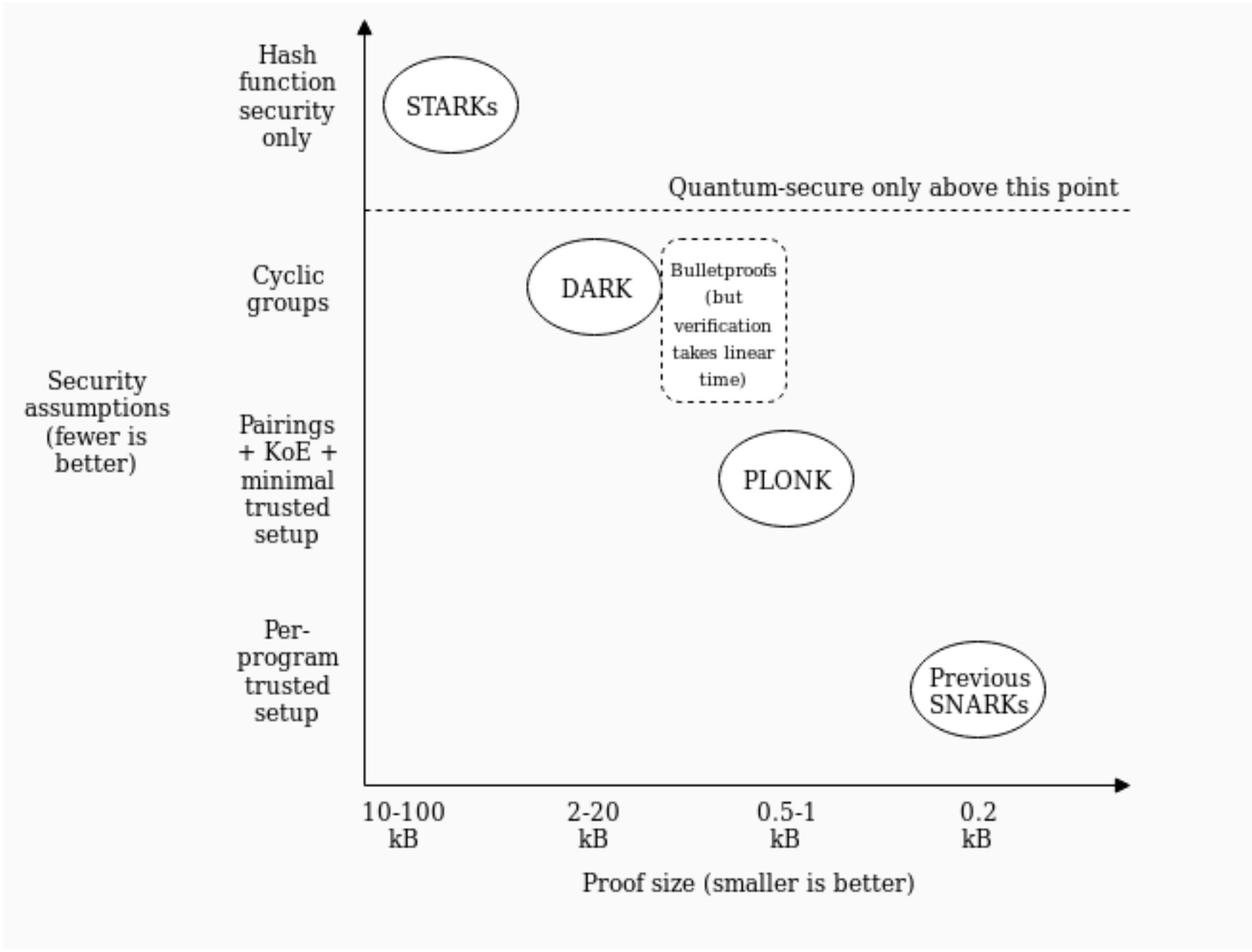
- STARKS
- DARK
- SHARK
- Sonic
- PLONK
- Bulletproofs
- Supersonic
- Aurora



https://vitalik.ca/general/2019/09/22/plonk.html

Alternatives to zk-SNARKS

- STARKS
- DARK
- SHARK
- Sonic
- PLONK
- Bulletproofs
- Supersonic
- Aurora



https://vitalik.ca/general/2019/09/22/plonk.html

STARKS

- Relies on Hash functions only
- quantum resistant
- larger proofs
    - few hundred kilobytes versus the 288 bytes in zk-SNARKs

Bulletproof

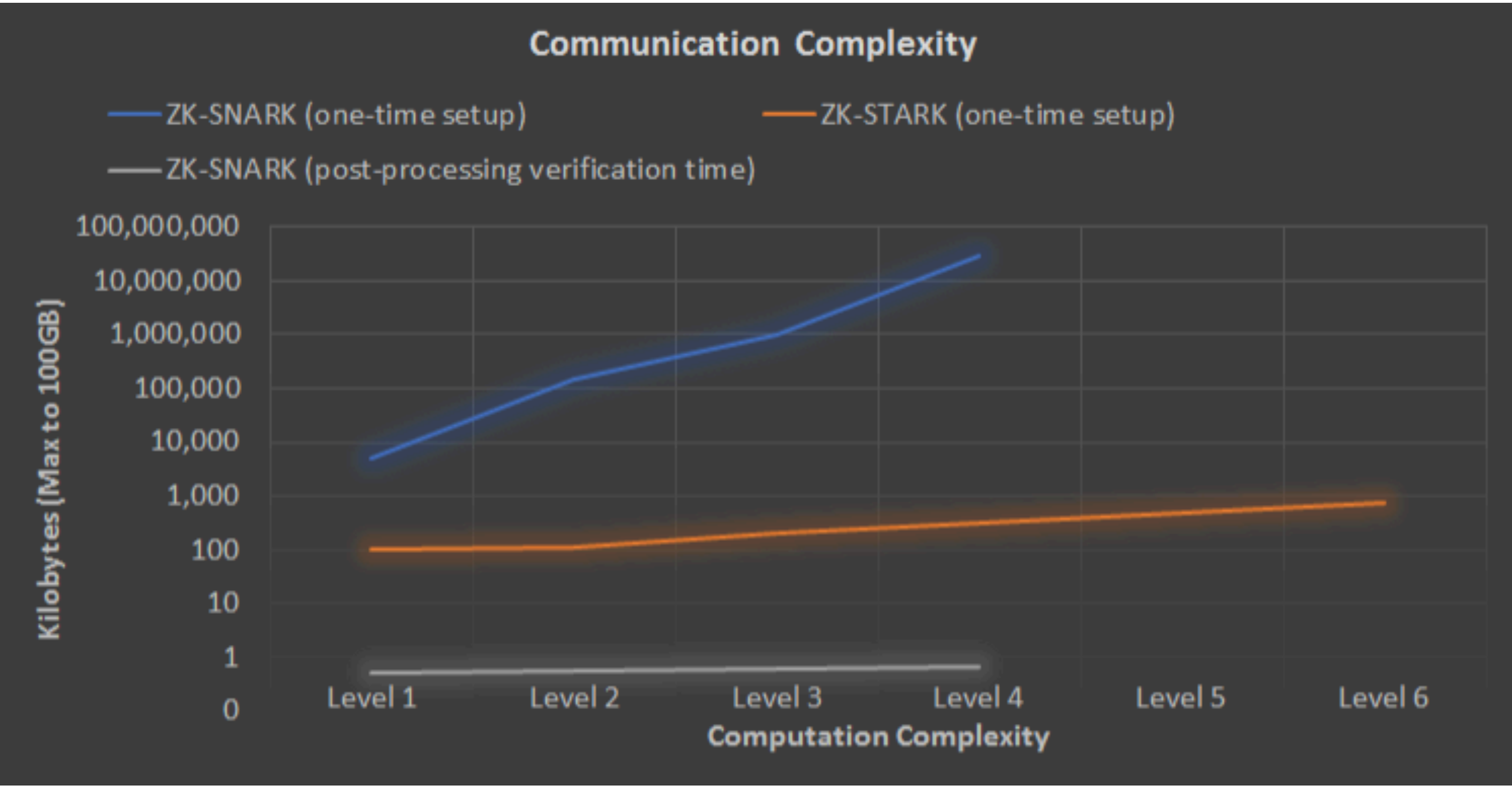
- Represent the computation as Pedersen Commitments
- Everything done in ECC math
- Currently used for range proofs (e.g. MimbleWimble proof that in $vG + rH\ v > 0$

Comparison

(secret evaluation point t)

	Toxic-waste free	Proof time	Verify time	Proof size
SNARKs ⓩ	No	2.3 s	10 ms	~200 B
STARKs	Yes	~1.6 s	~16 ms	**~45,000 B** 2 orders of magnitude too big
Bulletproofs	Yes	**~30 s** 1 order of magnitude too big	**~1100 ms** 2 orders of magnitude too big	~1300 B
Aurora	Yes	**~10 seconds** 1 order of magnitude too big	**~100 milliseconds** 1 order of magnitude too big	**~100,000 B** 2 orders of magnitude too big

Comparison


Communication Complexity
ZK-SNARK (one-time setup)
ZK-STARK (one-time setup)
ZK-SNARK (post-processing verification time)
Kilobytes (Max to 100GB)
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
Level 1
Level 2
Level 3
Level 4
Level 5
Level 6
Computation Complexity

Comparison

	Proof Size	Prover Time	Verification Time
SNARKs (has trusted setup)	288 bytes	2.3s	10ms
STARKs	45KB-200KB	1.6s	16ms
Bulletproofs	~1.3KB	30s	1100ms

— Elena Nadilinski, Devcon4

https://docs.google.com/presentation/d/1gfB6WZMvM9mmDKofFibIgsyYShdf0RV_Y8TLz3k1Ls0

Trusted setup needed: Yes 🔴 / No 🟢

zk-STARK 🟢

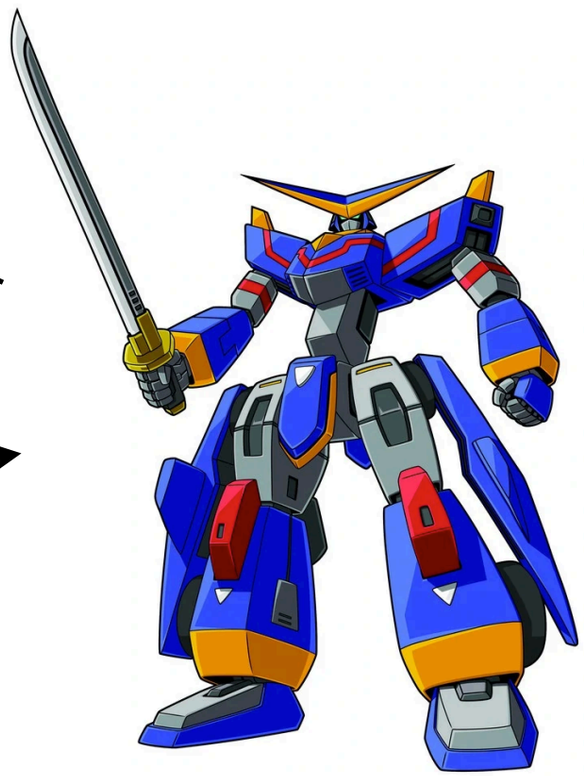
faster

🔴

zk-SNARK

More efficient setup (shorter)

faster



🟢

Bulletproof

More efficient setup (shorter)

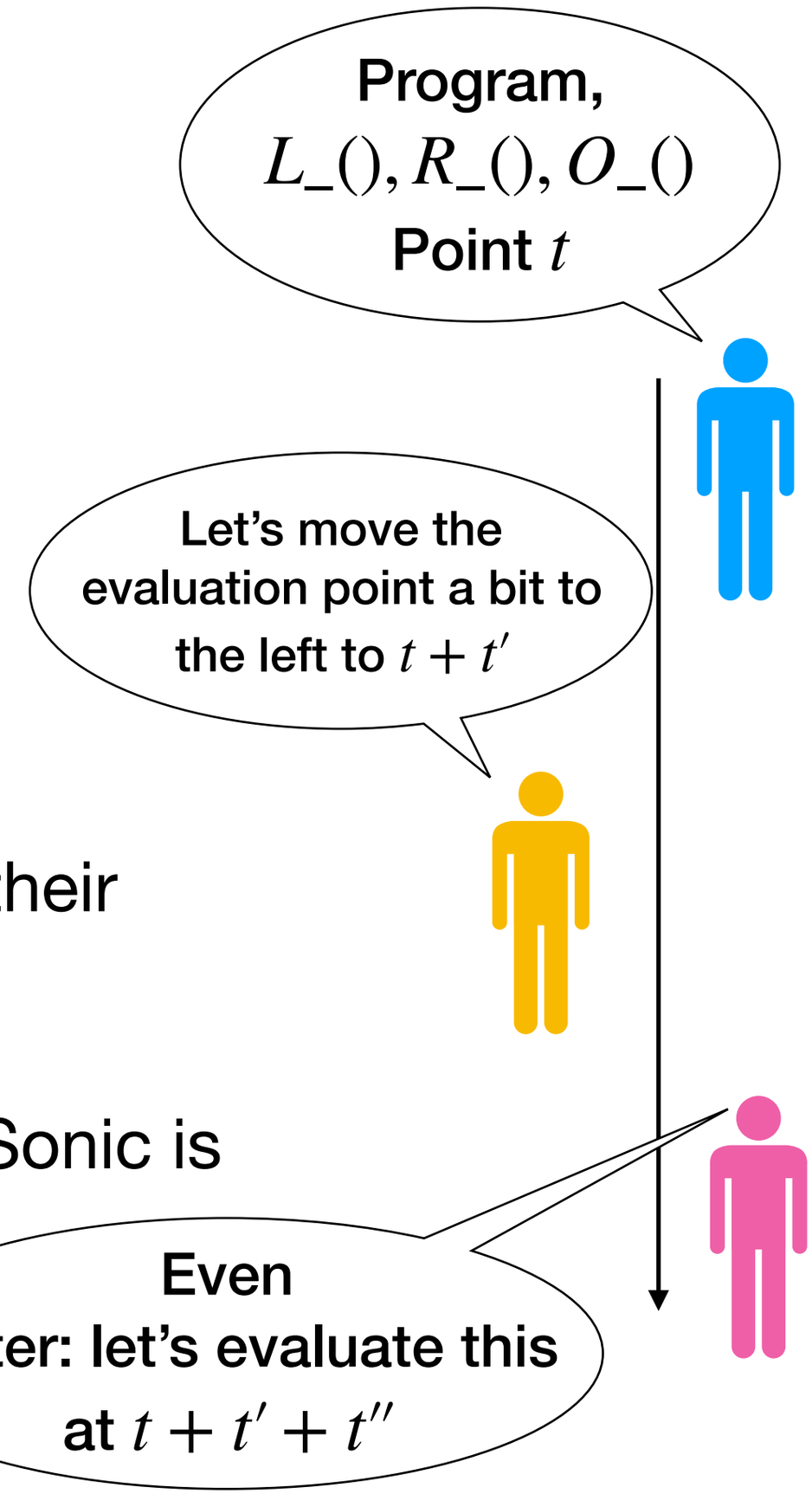
Alternatives to zk-SNARKS

- STARKS
- DARK
- SHARK
- Sonic
- PLONK
- Bulletproofs
- Supersonic
- Aurora

Sonic

- Continuous trusted setup ceremony
- Everybody can chime in and add their (secret) input
- As long as one person is honest, Sonic is secure

Sonic

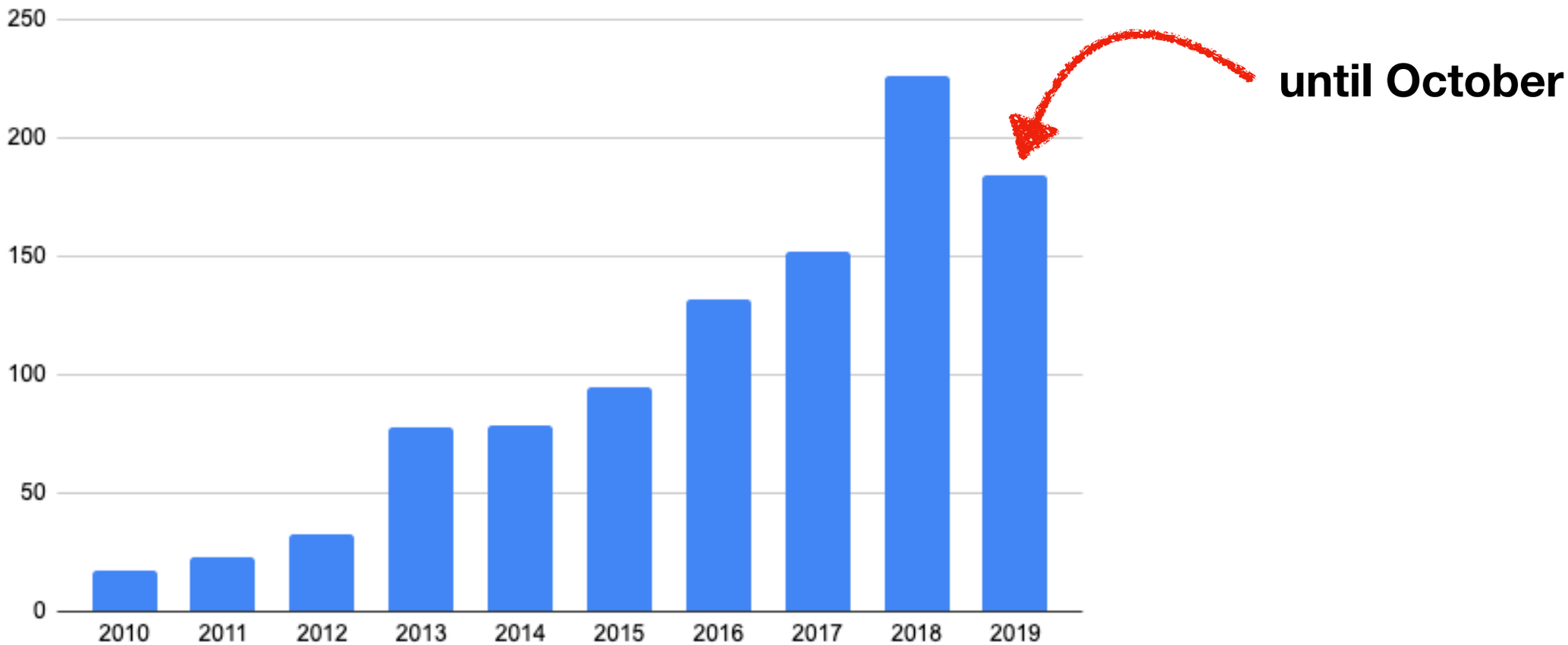


- Continuous trusted setup ceremony
  - Everybody can chime in and add their (secret) input
  - As long as one person is honest, Sonic is secure

This picture is only conceptually correct, in reality SONIC has more differences to zk-SNARK

Summary *zk-Something*

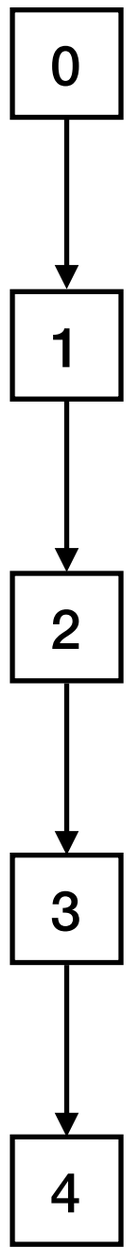
- It is possible to verify the correct execution of arbitrary code
- zk-SNARKs sparked a revolution in Zero-Knowledge Proofs
- More to come in the near future …

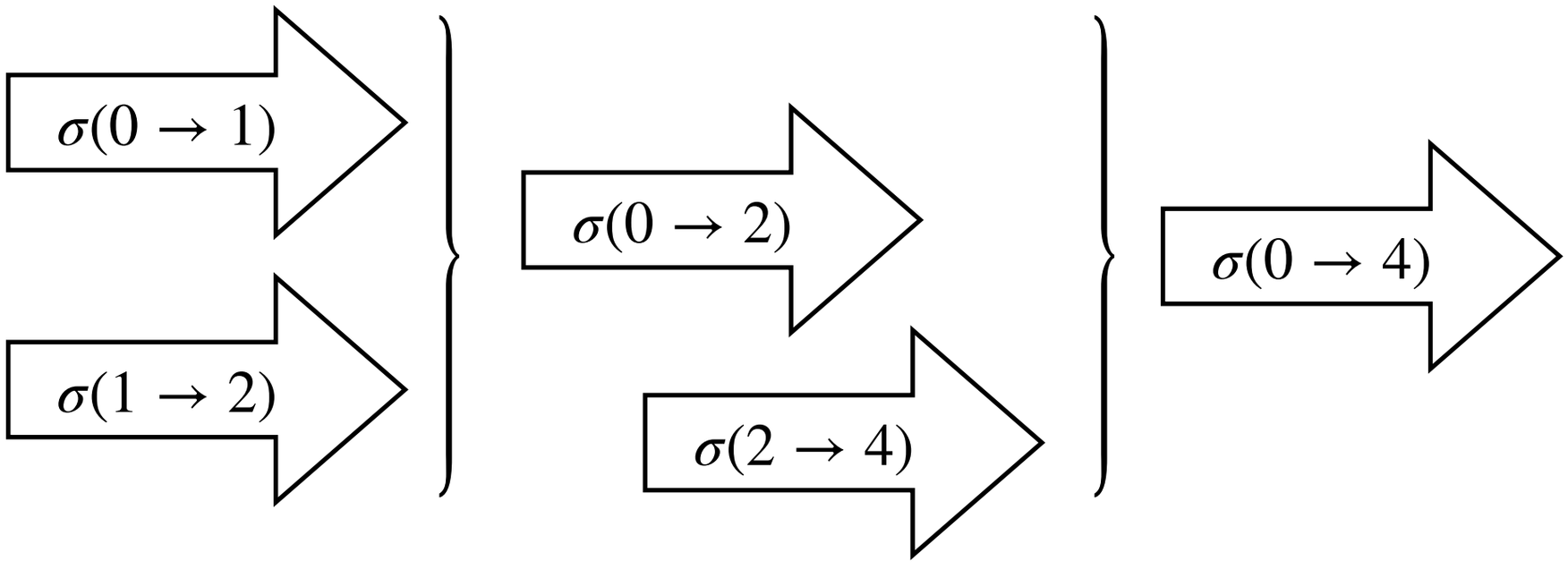


Papers found for “zero knowledge" "succinct" "argument"

Coda

- A blockchain completely in zk-SNARK
  - Verification of a transactions:
    1. A (recursive) SNARK that verifies a block was generated starting at the genesis block
    2. A SNARK verifying that the inputs are a leaf node in a Merkle Tree
- Snarks are recursively build up. If block 1 is a correct successor of block 0 $\sigma(0 \to 1)$ and block 2 is a successor of block 1 $\sigma(1 \to 2)$, then we can build a SNARK that evaluates both transitions to get a proof that 2 is a successor of 0 $\sigma(0 \to 2)$





Coda

- A blockchain completely in zk-SNARK
- Consensus, block building, zk-SNARK construction is done by powerful nodes
- Verification can be done by any user
    - Data “fits into a couple of tweets”
    - Verification time is ~100ms
    - no ‘delegation of trust’ to the miners (because in other protocols, the blockchain grows and becomes infeasible to verify for normal users)
- Constant verification size/time in Coda

End of Zero Knowledge

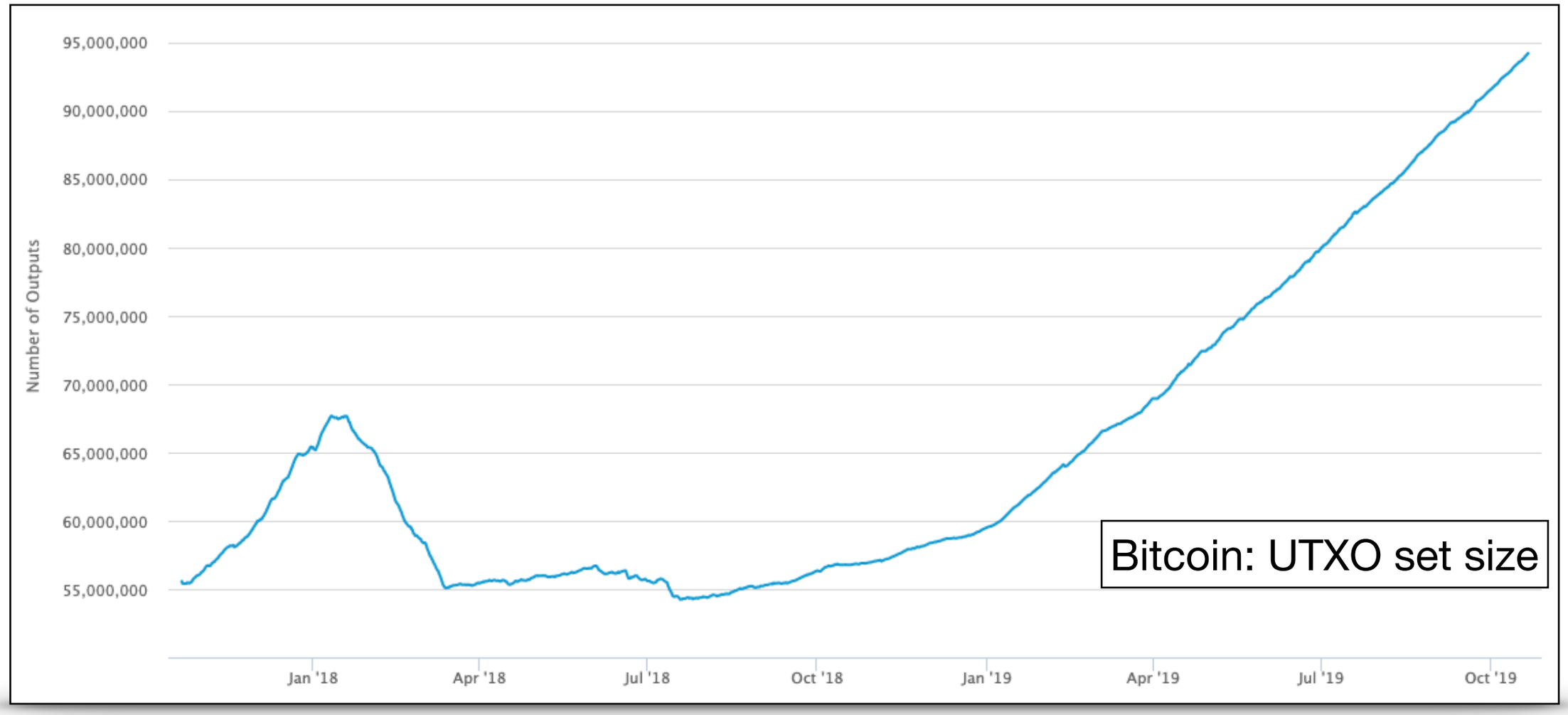
Questions?

Accumulators

UTXO replacement

Problem statement

- Currently the UTXO set in Bitcoin is a simple list
  - UTXO: Unspend transition outputs (coins in circulation)
- The miners need to keep track of this list



https://www.blockchain.com/charts/utxo-count?timespan=2years

Problem statement

- What if we shift the burden of proving that I own a transaction to the user
- blockchain miner create blocks (Hashes of parent blocks, Merkle Trees)
    - Can proof that an element is part of the blockchain
- users could maybe proof to the miners that the transaction output is not yet spend (in UTXO set)

Accumulators

- An accumulator is a short element (a number, a hash, etc.) that *contains* a proof about set membership
- Example: A Merkle Tree root is an accumulator
    - Set membership can be proven by showing a path from the root to the leaf containing the data
        - If the element is not in the Merkle Tree, such a path cannot be created.

Merkle Tree as Accumulator

- Merkle Trees are not very flexible
    - To add/delete an element, we need to rebuild the entire tree, i.e. $O(n)$ runtime to add delete an element

Accumulator

- What exactly do we need for an accumulator?
    - Base value (i.e. Merkle Tree root)
    - Either
        - the set of inputs (to generate a membership proof *on-the-fly* when needed)
        - or the set of membership proofs for each element

Accumulator

- What exactly do we need for an accumulator?
  - Base value (i.e. Merkle Tree root) **= Accumulator**
  - Either
    - the set of inputs (to generate a membership proof *on-the-fly* when needed)
    - or the set of membership proofs for each element **= Witness**

Accumulator

- Operations
  - Initialize: Generate an empty accumulator
  - Add element: re-compute the accumulator
  - Delete element: re-compute the accumulator
  - Witness update: re-compute the witness for an element

More specific terminology

- **Accumulator**: “A cryptographic accumulator is a primitive that produces a short binding commitment to a set of elements together with short membership/non-membership proofs for any element in the set.”

- **Dynamic Accumulator**: “Accumulator which supports addition/deletion of elements with O(1) cost, independent of the number of accumulated elements”

- **Universal Accumulator**: “Dynamic Accumulator which supports membership and non-membership proofs”

— D. Boneh, B. Bünz, B. Fisch,
“Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains”, 2018

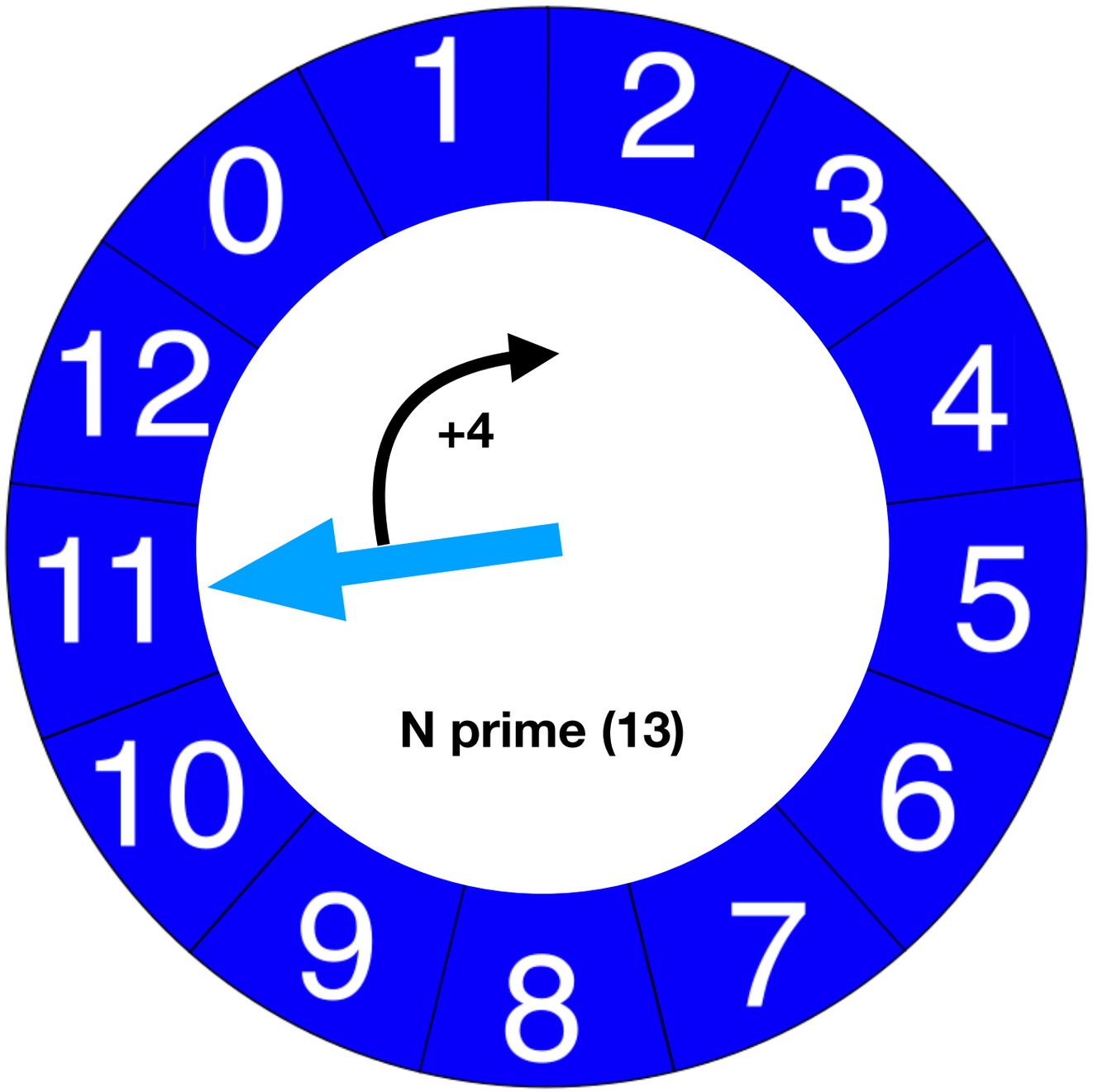
RSA Accumulator

- Using modulo math, assume we have a number $A \in \mathbb{Z}_N$

- Assume we have a hash function that creates a prime number as output $\mathscr{H}_P(\ldots)$

- Then $A' = A^{\mathscr{H}_p(\text{document})}$ is a RSA-Accumulator

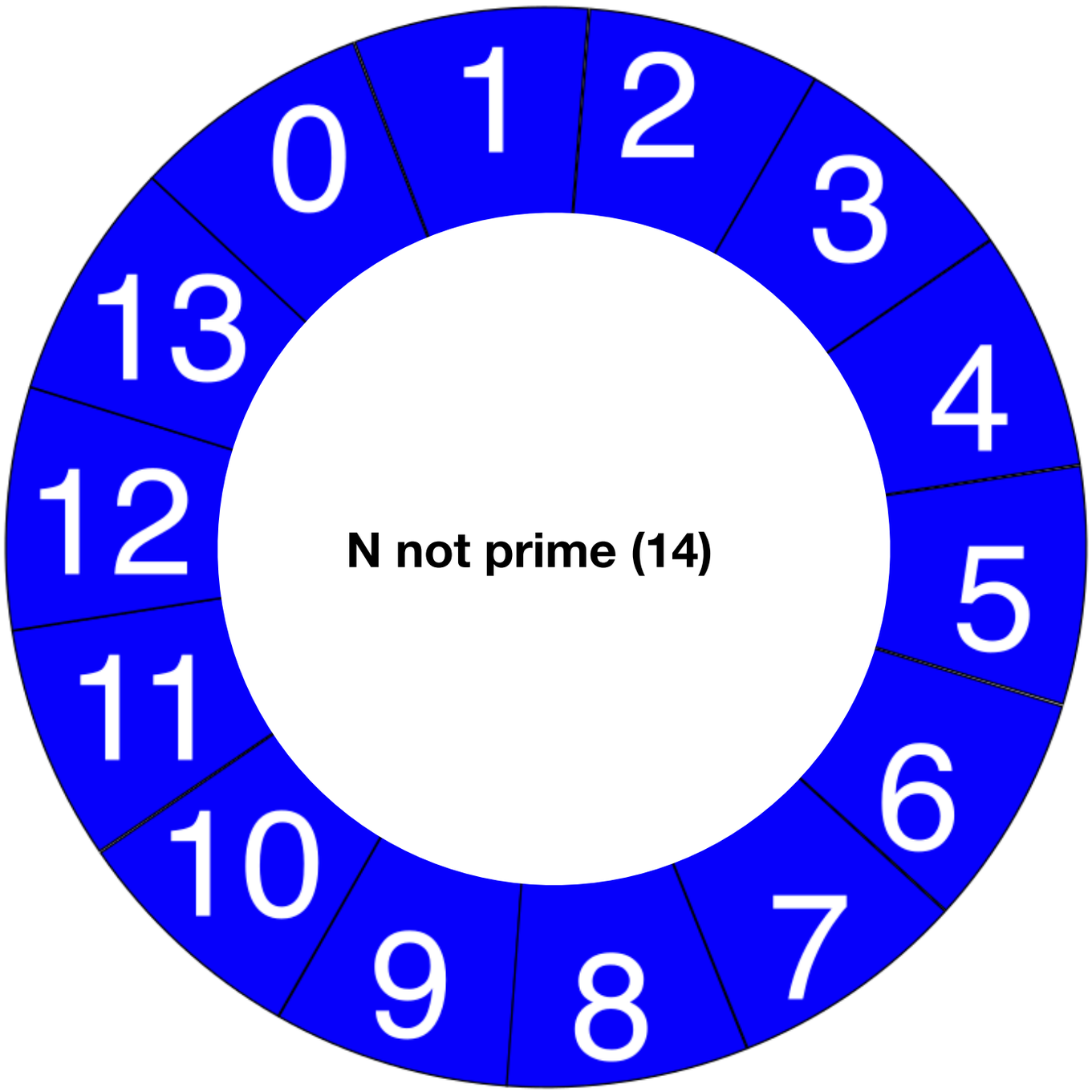
Module Math



Addition, Multiplication, etc. all well defined

$$(a+b) \mod N = \left((a \mod N) + (b \mod N) \mod N\right)$$

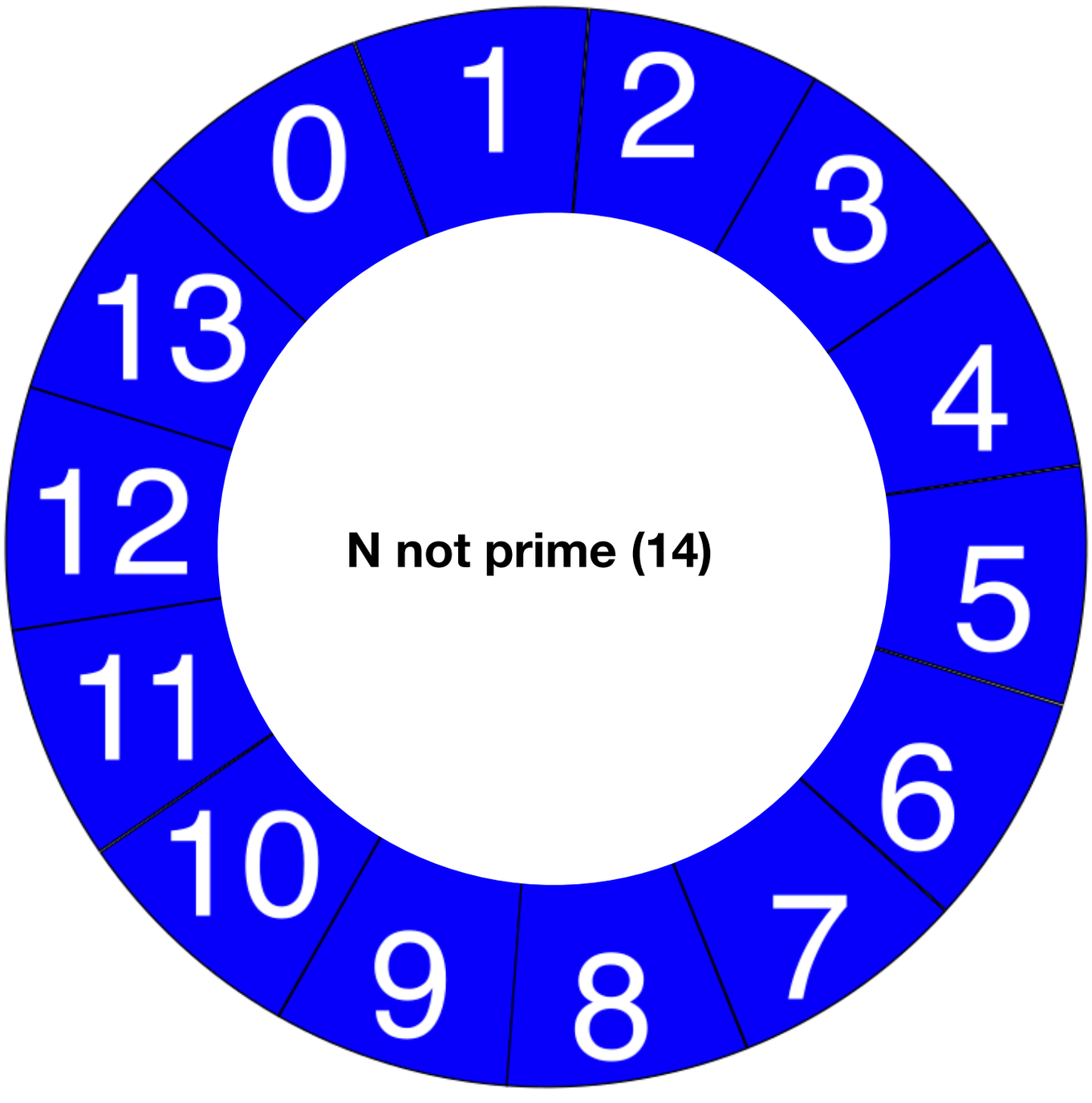
Module Math



Now we have

$$7 \cdot 2 = 14 = 0 \mod 14$$

Module Math



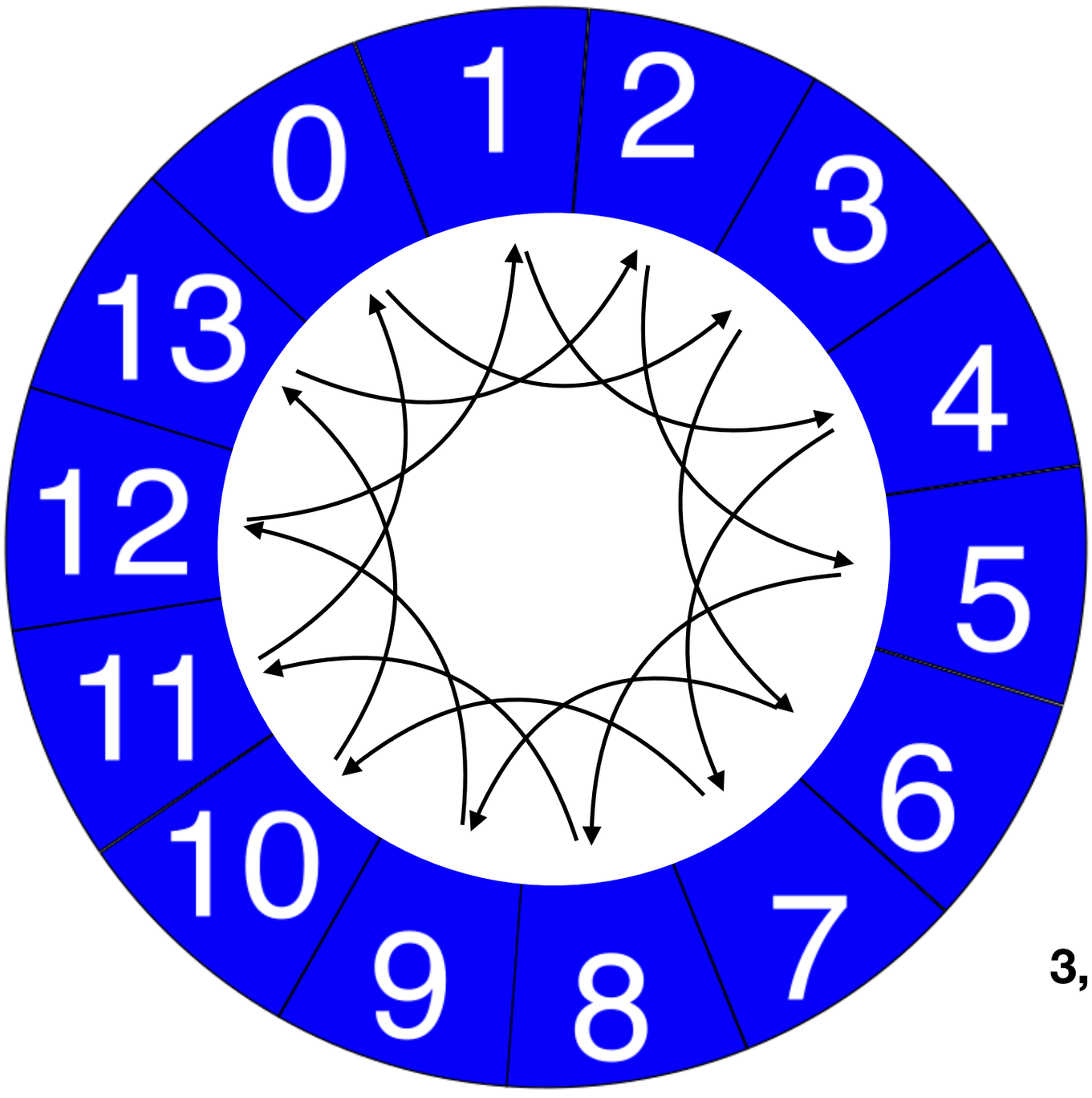
Now we have

$$7 \cdot 2 = 14 = 0 \mod 14$$

$$\frac{0}{7} = 2 \mod 14$$

Divisors of 0!!!!!

Module Math



N not prime (14)

Group order:

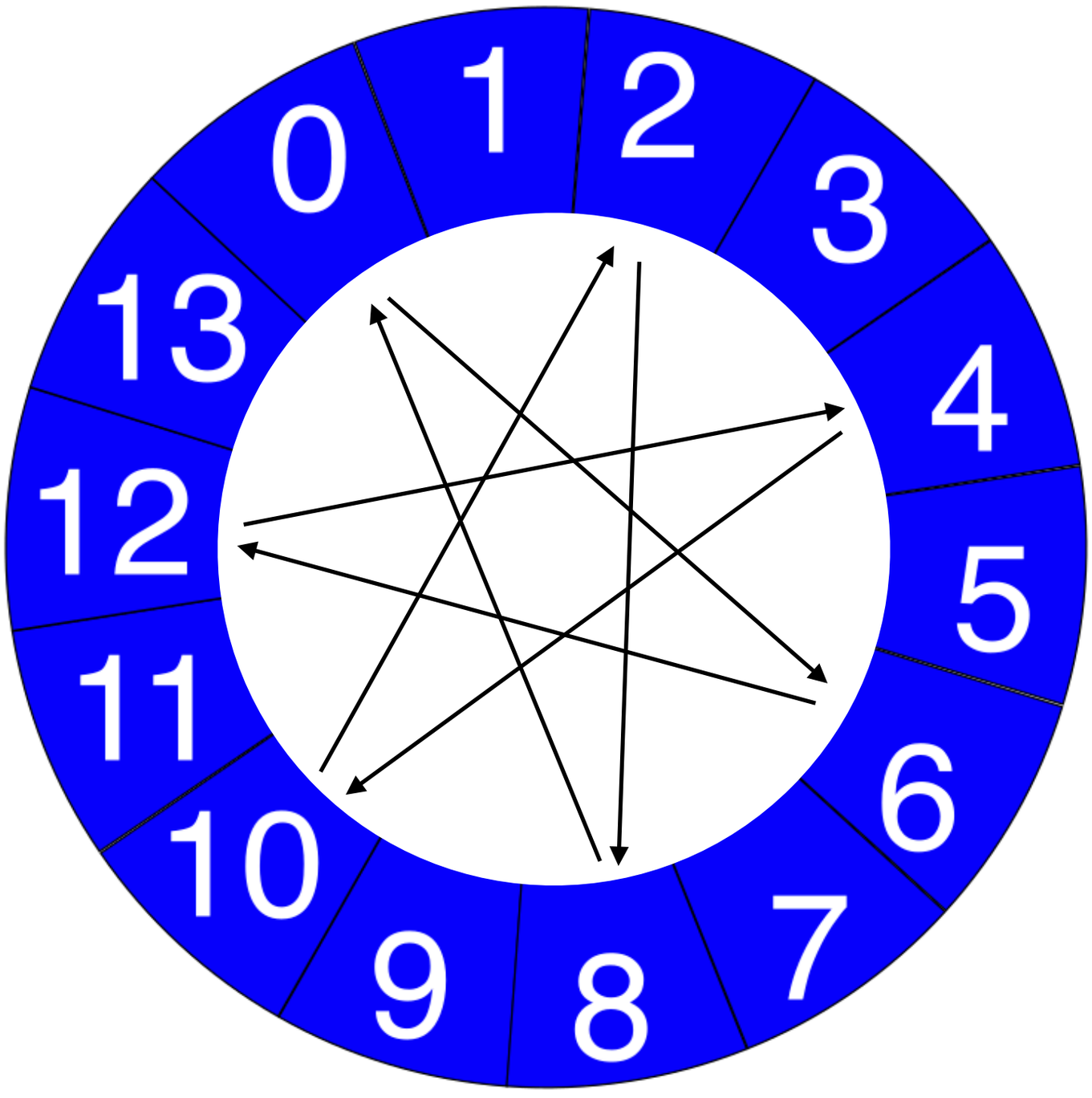
How often can I multiply an element before I get back to the beginning

Example: base element 3

3, 6, 9, 12, 1, 4, 7, 10, 13, 2, 5, 8, 11, 0

order 14

Module Math



N not prime (14)

Group order:

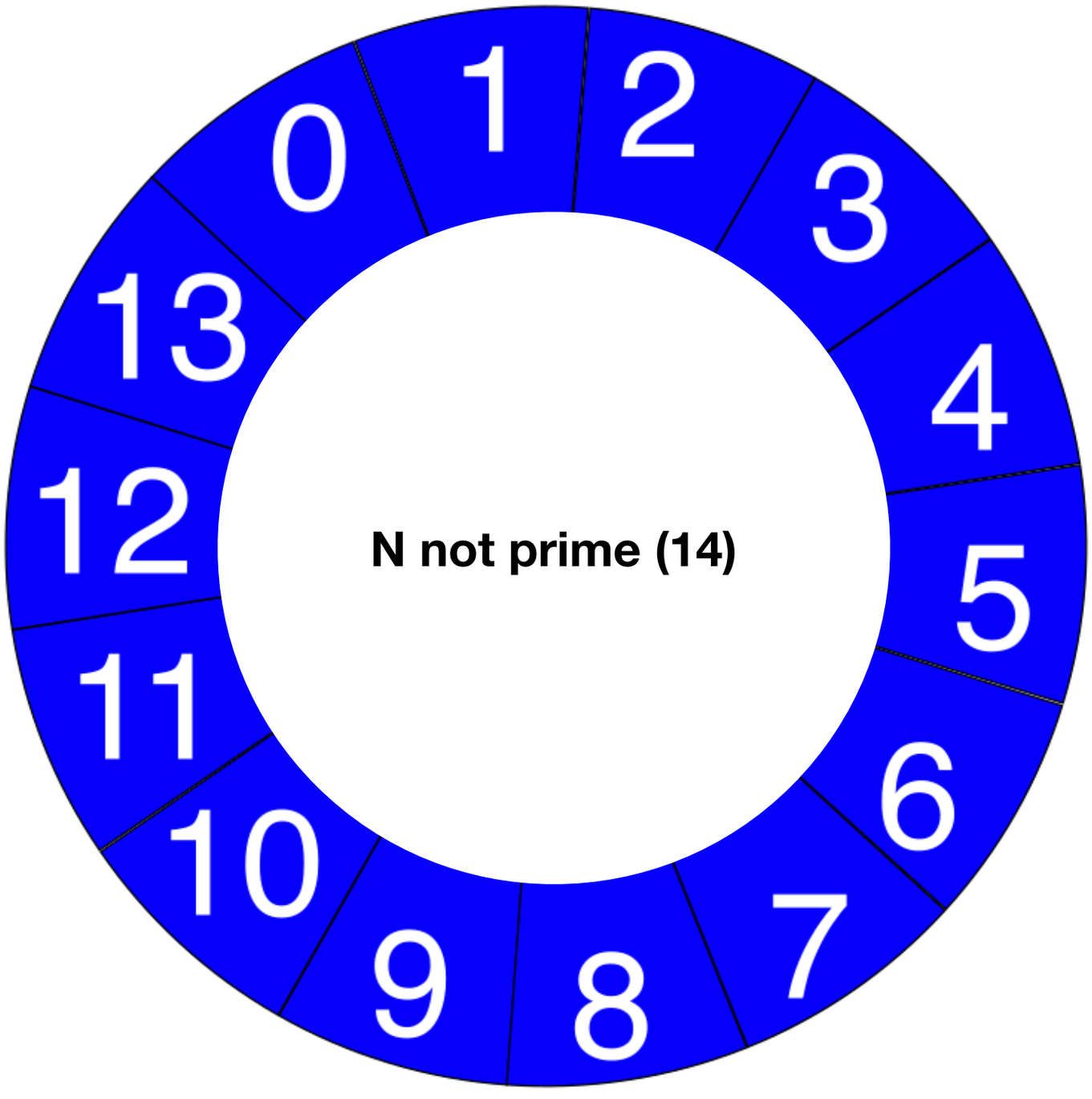
How often can I multiply an element before I get back to the beginning

Example: base element 6

6, 12, 4, 10, 2, 8, 0

order 7

Module Math



Group order:

a*b	3	5	6	7
0	0	0	0	0
1	3	5	6	7
2	6	10	12	0
3	9	1	4	7
4	12	6	10	0
5	1	11	2	7
6	4	2	8	0
7	7	7	0	7
8	10	12	6	0
9	13	3	12	7
10	2	8	4	0
11	5	13	10	7
12	8	4	2	0
13	11	9	8	7

Module Math
Group with unknown order

- Assume 2 large prime numbers p, q and $n = pq$
  - It is impossible to compute p and q given n
- Do all math $\mod n$
- Given a random value A, its order is not known

RSA Acumulator

- Init: Empty accumulator $A \xleftarrow{\text{random}} \mathbb{Z}_n$
- Add an element $A_{\text{new}} = A^e$ (if e is prime)
- Witness: $A^{\frac{1}{e}}$, because $\left(A^{\frac{1}{e}}\right)^e = A$
  - The accumulator without the element is the witness
  - Verify by adding the element and check for equality

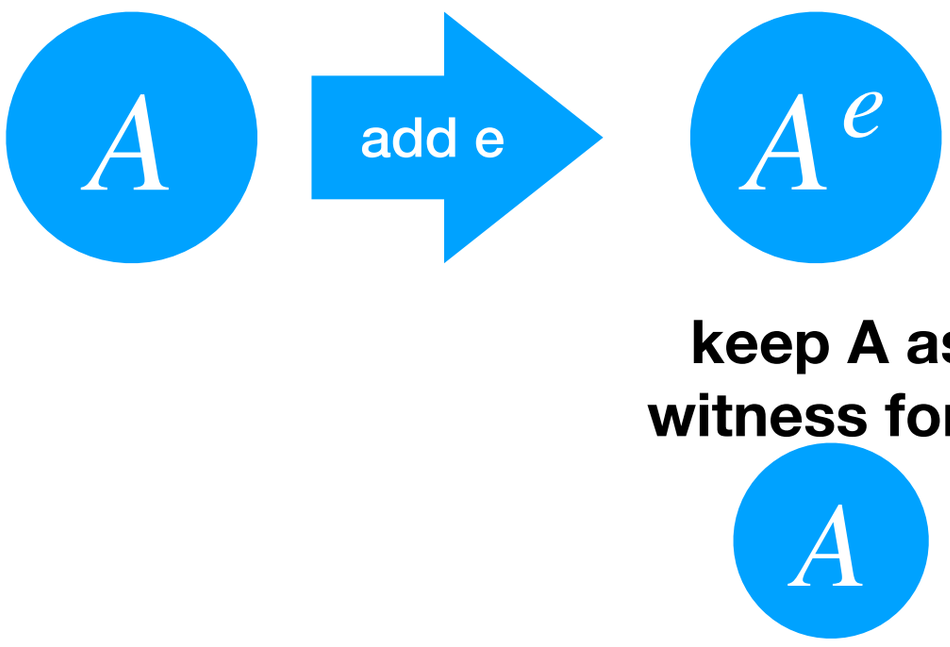
RSA Accumulator

- If the order is unknown, $A^{\frac{1}{e}}$ can not be computed for a new e

- When adding an element, keep the accumulator from before as a witness

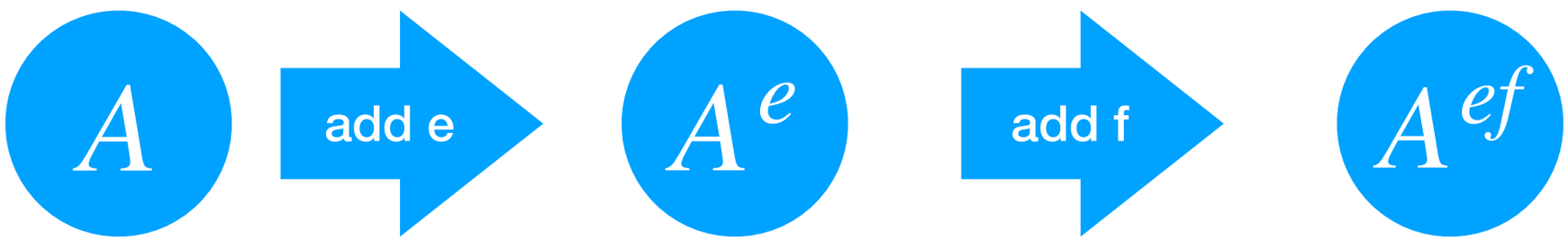
Witness

Adding element e to accumulator A

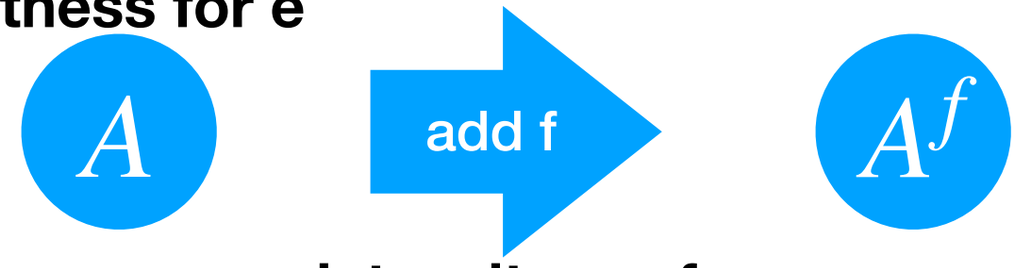


Witness

Adding element f to accumulator A'



keep A as witness for e

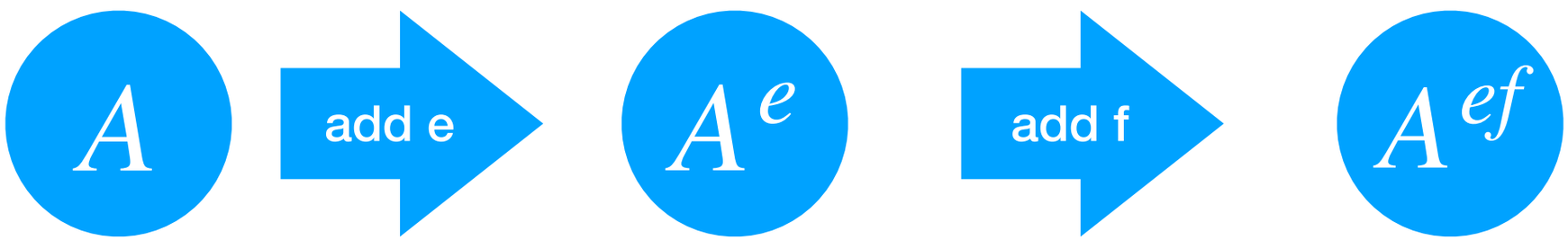


update witness for e

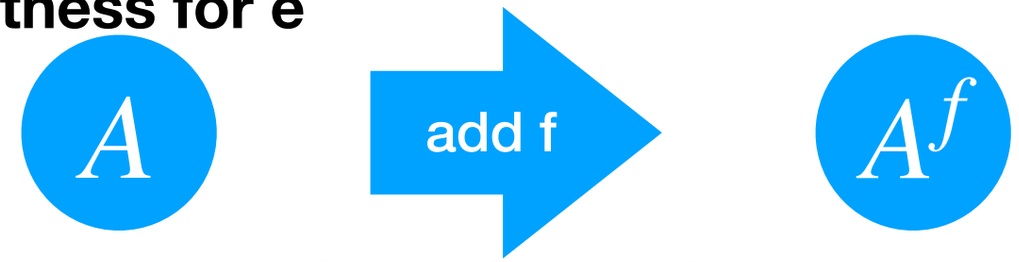
keep A^e as witness for f

Witness

Adding element f to accumulator A'



keep A as witness for e



update witness for e

keep A^e as witness for f

Verify:

$$\left(A^f\right)^e = A^{ef}$$

$$\left(A^e\right)^f = A^{ef}$$

Witnesses

- Accumulator $B = A^{e_1 \cdot e_2 \cdots e_n}$
  - B has accumulated the set $\mathcal{S} = \{e_1, e_2, \ldots, e_n\}$
  - B is a single number (2048 bits), independent of the size of the set $\mathcal{S}$
- A witness W_{e_i} for an element e_i is simply $A^{e_1 \cdots e_{i-1} e_{i+1} \cdots e_n}$
  - a single number
  - Verification via one exponentiation $\left(W_{e_i}\right)^{e_i} \stackrel{?}{=} B$

Hash to prime

- Currently we treated all elements e_i as prime numbers

- We need a hash function that produces primes

Hash to prime

- Currently we treated all elements e_i as prime numbers
- We need a hash function that produces primes
  - The output of a hash is a number
    1. Test for primality.
       - if yes → done
       - if no → hash the output once more. GOTO 1

$\mathscr{H}(e) \rightarrow \mathscr{H}(\mathscr{H}(e)) \rightarrow \mathscr{H}(\mathscr{H}(\mathscr{H}(e))) \rightarrow \ldots$ until prime

UTXO Replacement

- Theoretically, bitcoin could replace the UTXO set with an RSA Accumulator
- Adding the output of a new transaction: $A^{\mathcal{H}(\text{tx output})}$
- Spending: Prove membership via witness W_{txo}
  - Elements are removed, when output is spend
  - Witness itself is accumulator with the value

Summary

- Accumulators are can be used to squeeze a large set into a single element
    - Merkle Tree root can be seen as an accumulator
    - Even a blockchain is an accumulator
- Dynamic accumulators: adding and deleting elements
- Efficient accumulators perform adding/deleting in $O(1)$
- As a UTXO replacement, they shift the burden of tracking the UTXO set to the individual users

Remarks

- Several algorithms exist to deal with large numbers of elements
  - Naively, updating M elements requires $O(M)$ steps
    - Intelligently done, only $O(\log M)$ steps are needed
- If the prime values p, q are known, a new witness can be *invented*, since $A^{1/x}$ can be computed easily for any x
  - p, q are called *toxic waste* (trusted setup)