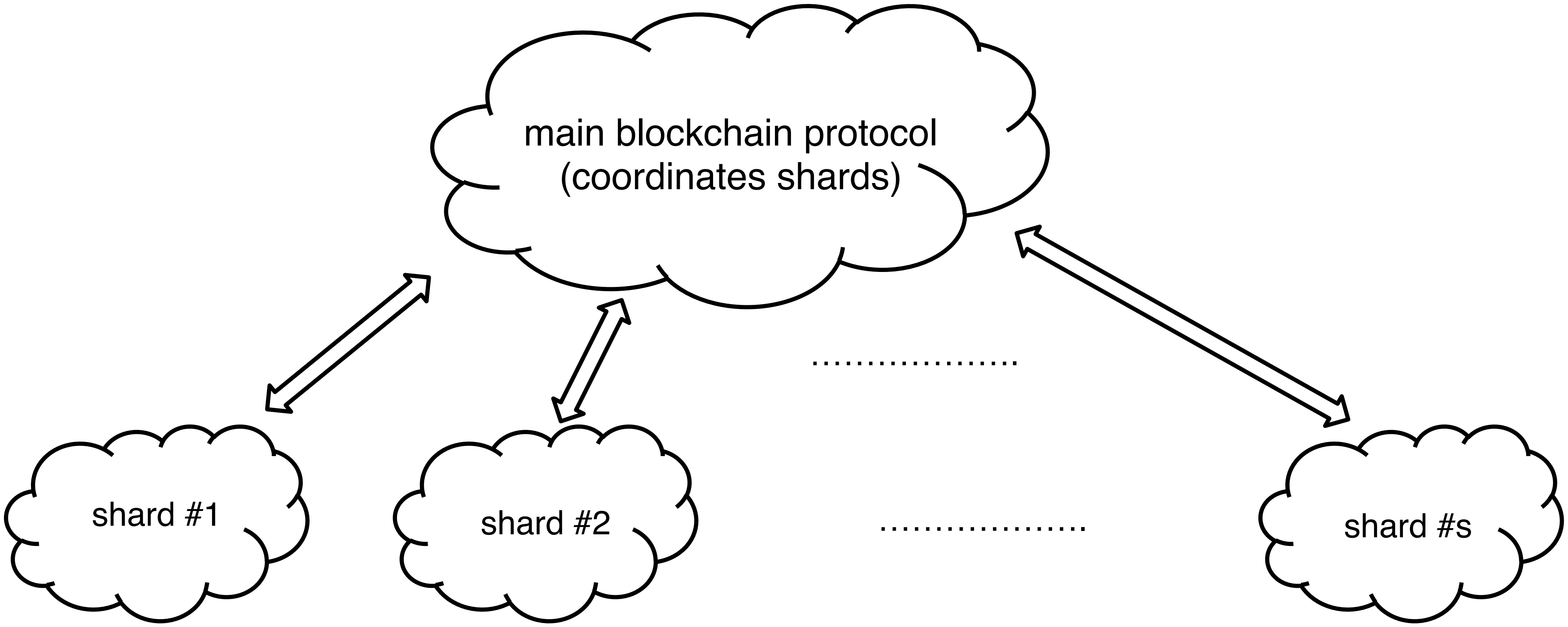
Lecture #18:
SNARKs

COMS 4995-001:
The Science of Blockchains

URL: https://timroughgarden.org/s25/

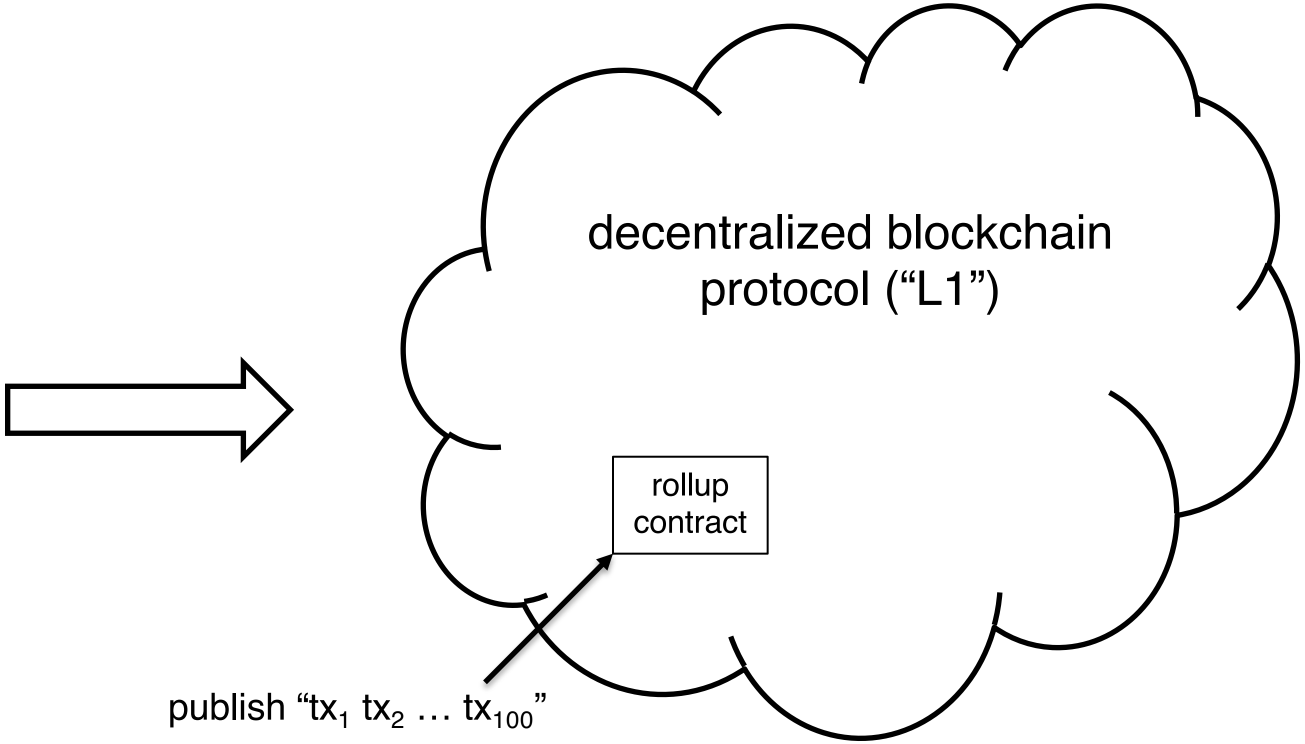
Tim Roughgarden

Scaling Execution via Rollups



L1 ⬄ Rollup Architecture

(possibly centralized) rollup



Goals for Lecture #18

1. Defining the state root verification problem.
   – "proactive proof of correctness" - the key problem for validity rollups

2. Witnesses and NP statements.
   – NP problem = easy to check correctness of purported solution

3. SNARKs.
   – short (<< witness length) & easy-to-verify proofs of an NP statement
   – suitable for posting to an L1 blockchain

4. General probabilistic verification and the PCP Theorem.
   – can derive SNARKS from one of the deepest results in theory CS

Rollups Review

Recall: in a “classic” rollup (optimistic or validity), periodically publish rollup txs to L1, along with new state commitment/root.

Rollups Review

Recall: in a “classic” rollup (optimistic or validity), periodically publish rollup txs to L1, along with new state commitment/root.

Hard part: ensure that L1 can programmatically verify correctness of each state root.

- without the L1 re-executing the rollup txs itself

Rollups Review

Recall: in a "classic" rollup (optimistic or validity), periodically publish rollup txs to L1, along with new state commitment/root.

Hard part: ensure that L1 can programmatically verify correctness of each state root.

- without the L1 re-executing the rollup txs itself
- possibly with assistance from 3rd parties like watchdogs (optimistic rollups) or provers (validity rollups)
  - generally require a "1 in N" trust assumption for these parties

Validity Rollups

Warning: often called “zk” rollups. (even though not zero-knowledge)

Validity Rollups

Warning: often called “zk” rollups. (even though not zero-knowledge)

High-level idea of validity rollups: guilty until proven innocent.

- L1 assumes by default that each state commitment is incorrect
- rely on “provers” to submit proofs of correctness to L1
  - if nothing else, rollup operator can run its own prover
- L1 verifies proof of correctness directly
  - state commitment rejected if accompanying proof fails verification

Validity Rollups

Warning: often called "zk" rollups. (even though not zero-knowledge)

High-level idea of validity rollups: guilty until proven innocent.

- L1 assumes by default that each state commitment is incorrect
- rely on "provers" to submit proofs of correctness to L1
  - if nothing else, rollup operator can run its own prover
- L1 verifies proof of correctness directly
  - state commitment rejected if accompanying proof fails verification

Hard part: verification of correctness proofs should be *much* easier than tx re-execution --- i.e., need "SNARKs."

State Root Verification (Attempt 1)

Question: what is the problem a validity rollup needs to solve?

- i.e., logic in L1 contract to proactively verify correctness of each state root

State Root Verification (Attempt 1)

Question: what is the problem a validity rollup needs to solve?

- i.e., logic in L1 contract to proactively verify correctness of each state root

Input: (i) previous state root r_0 (assumed correct); (ii) latest batch B = $t_1, t_2, \ldots, t_k$ of rollup txs published to L1; (iii) alleged new state root r_1.

State Root Verification (Attempt 1)

Question: what is the problem a validity rollup needs to solve?

– i.e., logic in L1 contract to proactively verify correctness of each state root

Input: (i) previous state root r_0 (assumed correct); (ii) latest batch B = $t_1, t_2, \ldots, t_k$ of rollup txs published to L1; (iii) alleged new state root r_1.

Output: "yes" if, starting from r_0, executing the txs of B (in order) results in r_1, and "no" otherwise.

State Root Verification (Attempt 1)

Question: what is the problem a validity rollup needs to solve?

- i.e., logic in L1 contract to proactively verify correctness of each state root

Input: (i) previous state root r_0 (assumed correct); (ii) latest batch B = $t_1,t_2,\ldots,t_k$ of rollup txs published to L1; (iii) alleged new state root r_1.

Output: "yes" if, starting from r_0, executing the txs of B (in order) results in r_1, and "no" otherwise.

Issue: execution of a rollup tx is defined with respect to the full rollup state, not just the (256-bit) state root.

State Root Verification (Attempt 2)

Input:

- previous rollup state root r_0 (assumed correct)
- latest batch B = $t_1, t_2, \ldots, t_k$ of rollup txs published to L1
- alleged new rollup state root r_1

State Root Verification (Attempt 2)

Input:

- previous rollup state root r_0 (assumed correct)
- latest batch $B = t_1, t_2, \ldots, t_k$ of rollup txs published to L1
- alleged new rollup state root r_1

Output: "yes" if *there exists a state* σ_0 with root(σ_0)= r_0 s.t. executing the txs of B results in state σ_1 with root(σ_1)= r_1, and "no" otherwise.

– assuming no hash function collisions, σ_1 must be correct new rollup state

State Root Verification (Attempt 2)

Input:

- previous rollup state root r_0 (assumed correct)
- latest batch B = $t_1, t_2, \ldots, t_k$ of rollup txs published to L1
- alleged new rollup state root r_1

Output: "yes" if *there exists a state* σ_0 with root(σ_0)= r_0 s.t. executing the txs of B results in state σ_1 with root(σ_1)= r_1, and "no" otherwise.

– assuming no hash function collisions, σ_1 must be correct new rollup state

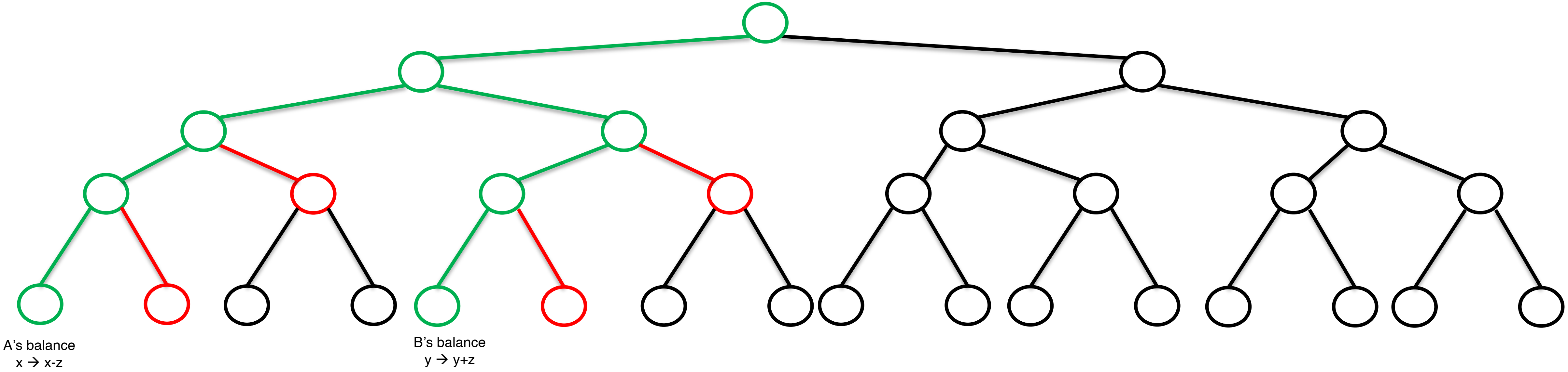
Question: is full rollup state necessary to verify correctness of r_1?

– cf., stateless validation

Example: Simple Transfers

Inputs to verification problem: initial state root r_0, list of txs, alleged new state root r_1.

Question: how much of the actual state is necessary to verify the correctness of r_1?



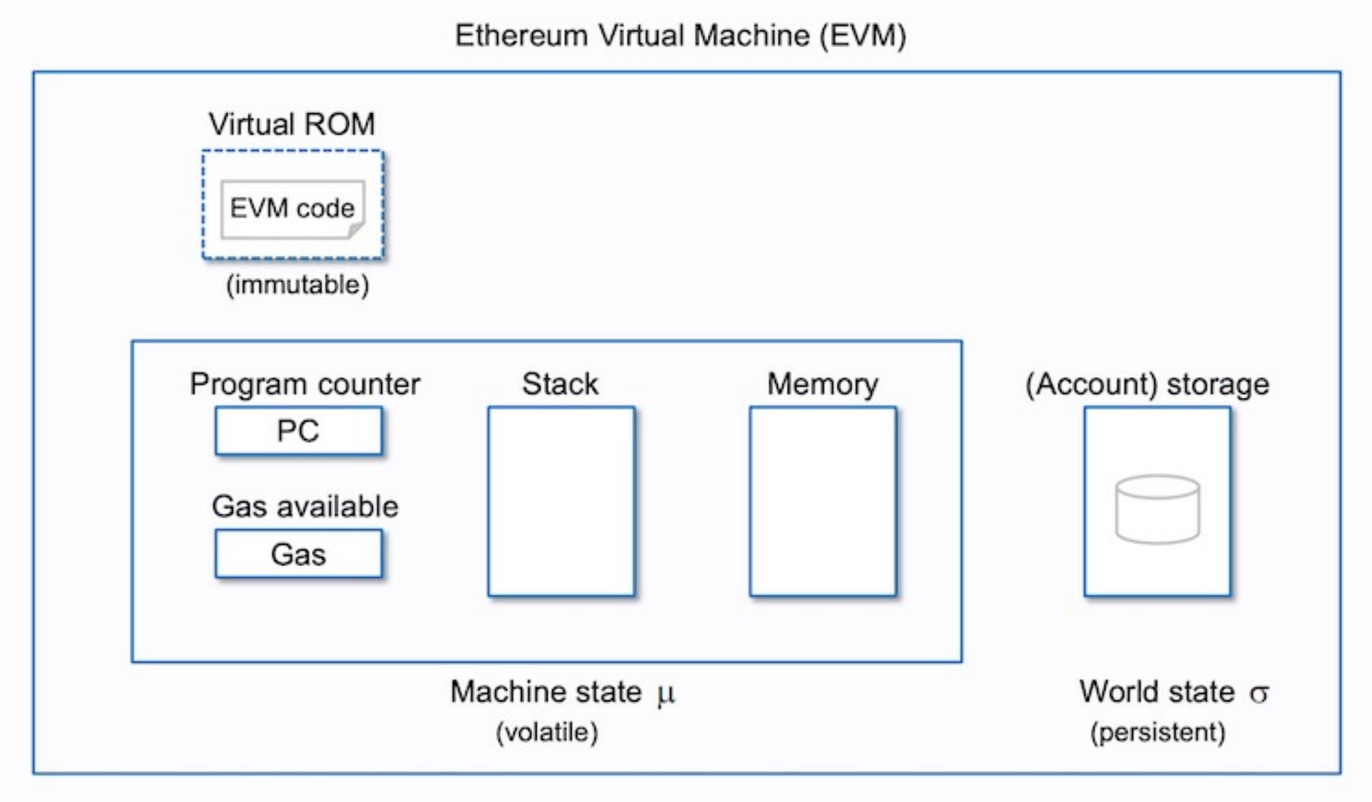
- sufficient to supply Merkle proofs for balances of A and B
  - this is enough info to compute new Merkle root

Verifying General Transactions

Inputs to verification problem: initial state root r_0, list of txs, alleged new state root r_1.

Question: how much of the actual state is necessary to verify the correctness of r_1?

Example: EVM State



[source: https://www.quicknode.com/guides/ethereum-development/smart-contracts/a-dive-into-evm-architecture-and-opcodes]

Verifying General Transactions

Inputs to verification problem: initial state root r_0, list of txs, alleged new state root r_1.

Question: how much of the actual state is necessary to verify the correctness of r_1?

- need Merkle pf for each part of rollup state accessed by some tx
  - Merkle proofs supply state info on need-to-know basis

Verifying General Transactions

Inputs to verification problem: initial state root r_0, list of txs, alleged new state root r_1.

Question: how much of the actual state is necessary to verify the correctness of r_1?

- need Merkle pf for each part of rollup state accessed by some tx
  - Merkle proofs supply state info on need-to-know basis
- after each update to state, recompute new state root
  - increment nonce, write new value to variable in contract storage, etc.

Verifying General Transactions

Inputs to verification problem: initial state root r_0, list of txs, alleged new state root r_1.

Question: how much of the actual state is necessary to verify the correctness of r_1?

- need Merkle pf for each part of rollup state accessed by some tx
  - Merkle proofs supply state info on need-to-know basis
- after each update to state, recompute new state root
  - increment nonce, write new value to variable in contract storage, etc.
- after processing all txs, can check if new state root = r'

State Root Verification (Final Attempt)

Input:

- previous rollup state root r_0 (assumed correct)
- latest batch $B = t_1, t_2, \ldots, t_k$ of rollup txs published to L1
- alleged new rollup state root r_1

State Root Verification (Final Attempt)

Input:

- previous rollup state root r_0 (assumed correct)
- latest batch B = $t_1, t_2, \ldots, t_k$ of rollup txs published to L1
- alleged new rollup state root r_1

Output: "yes" if *there exists Merkle proofs* $\pi_1, \pi_2, \ldots, \pi_k$ such that starting from r_0 and executing the txs of B (with relevant state supplied by the π_i's) results in r_1, and "no" otherwise.

- assuming no hash function collisions, r_1 must be the Merkle root of the correct new rollup state

How Can L1 Verify State Root Correctness?

Input: (i) previous state root r_0 (assumed correct); (ii) latest batch B = $t_1, t_2, \ldots, t_k$ of rollup txs published to L1; (iii) alleged new state root r_1.

Output: "yes" if *there exists Merkle proofs* $\pi_1, \pi_2, \ldots, \pi_k$ such that starting from r_0 and executing the txs of B (with relevant state supplied by the π_i's) results in r_1, and "no" otherwise.

How Can L1 Verify State Root Correctness?

Input: (i) previous state root r_0 (assumed correct); (ii) latest batch B = $t_1, t_2, \ldots, t_k$ of rollup txs published to L1; (iii) alleged new state root r_1.

Output: "yes" if *there exists Merkle proofs* $\pi_1, \pi_2, \ldots, \pi_k$ such that starting from r_0 and executing the txs of B (with relevant state supplied by the π_i's) results in r_1, and "no" otherwise.

Bad idea: sequencer posts relevant Merkle proofs to L1 (along with tx data and the new state root) so L1 can check correctness.

- turns the L1 contract into a stateless validator (for the rollup)
- impractical: Merkle proofs too big, re-execution of rollup txs too much work

How Can L1 Verify State Root Correctness?

Input: (i) previous state root r_0 (assumed correct); (ii) latest batch B = $t_1, t_2, \ldots, t_k$ of rollup txs published to L1; (iii) alleged new state root r_1.

Output: "yes" if *there exists Merkle proofs* $\pi_1, \pi_2, \ldots, \pi_k$ such that starting from r_0 and executing the txs of B (with relevant state supplied by the π_i's) results in r_1, and "no" otherwise.

Bad idea: sequencer posts relevant Merkle proofs to L1 (along with tx data and the new state root) so L1 can check correctness.

Revised idea: sequencer posts to L1 *a proof that it knows a solution* (i.e., relevant Merkle proofs) to the state root verification problem.

- Merkle proofs themselves not posted, only "proof of knowledge"
- L1 need only verify correctness of proof of knowledge (no tx re-execution)

Witness and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- example: Traveling Salesman Problem (TSP)

Witness and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- example: Traveling Salesman Problem (TSP)
- example: Satisfiability (SAT)

Witness and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- example: Traveling Salesman Problem (TSP)
- example: Satisfiability (SAT)
- example: state root verification (SRV)

Witness and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- example: Traveling Salesman Problem (TSP)
- example: Satisfiability (SAT)
- example: state root verification (SRV)

In general: an NP problem is defined by a poly-time algorithm C that, given an input x and purported solution/witness w, outputs 0 or 1.

- x is a "yes" instance if there exists a witness w with C(x,w)=1
- x is a "no" instance if C(x,w)=0 for every w

Witness and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- examples: TSP, SAT, SRV

In general: an NP problem is defined by a poly-time algorithm C that, given an input x and purported solution/witness w, outputs 0 or 1.

- x is a “yes” instance if there exists a witness w with C(x,w)=1
- x is a “no” instance if C(x,w)=0 for every w

What we need: (for SRV)

Witness and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- examples: TSP, SAT, SRV

In general: an NP problem is defined by a poly-time algorithm C that, given an input x and purported solution/witness w, outputs 0 or 1.

- x is a “yes” instance if there exists a witness w with C(x,w)=1
- x is a “no” instance if C(x,w)=0 for every w

What we need: (for SRV) convincing proof that a witness exists, with:

- proof length << witness length
- proof verification time << time to evaluate C

SNARKs

Definition: a SNARK for an NP problem (defined by C) is a way to generate short and easy-to-verify proofs π of existence of a witness.

SNARKs

Definition: a SNARK for an NP problem (defined by C) is a way to generate short and easy-to-verify proofs π of existence of a witness.

– verification algorithm V takes (x, π) as input, outputs "yes"/"no"

  - in validity rollups, π posted to L1, L1 smart contract runs V

SNARKs

Definition: a SNARK for an NP problem (defined by C) is a way to generate short and easy-to-verify proofs π of existence of a witness.

- verification algorithm V takes (x, π) as input, outputs "yes"/"no"
  - in validity rollups, π posted to L1, L1 smart contract runs V
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))
  - note: length of π will be at most running time of V (hence, short)

SNARKs

Definition: a SNARK for an NP problem (defined by C) is a way to generate short and easy-to-verify proofs π of existence of a witness.

- verification algorithm V takes (x, π) as input, outputs "yes"/"no"
  - in validity rollups, π posted to L1, L1 smart contract runs V
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))
  - note: length of π will be at most running time of V (hence, short)
- given w with C(x,w)=1, easy to generate π s.t. V(x, π)="yes"
  - "prover time," ideally O(RT of C)

SNARKs

Definition: a SNARK for an NP problem (defined by C) is a way to generate short and easy-to-verify proofs π of existence of a witness.

- verification algorithm V takes (x, π) as input, outputs "yes"/"no"
  - in validity rollups, π posted to L1, L1 smart contract runs V
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))
  - note: length of π will be at most running time of V (hence, short)
- given w with C(x,w)=1, easy to generate π s.t. V(x, π)="yes"
  - "prover time," ideally O(RT of C)
- if x a "no" instance, computationally infeasible to find π s.t. V(x, π)="yes"
  - i.e., practically impossible to convince verifier of a false statement

SNARKs in Validity Rollups

Definition: a SNARK for an NP problem (defined by C) is a way to generate short and easy-to-verify proofs π of existence of a witness.

- verification algorithm V takes (x, π) as input, outputs "yes"/"no"
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))
- given w with C(x,w)=1, easy to generate π s.t. V(x, π)="yes"
- if x a "no" instance, computationally infeasible to find π s.t. V(x, π)="yes"

SNARKs in Validity Rollups

Definition: a SNARK for an NP problem (defined by C) is a way to generate short and easy-to-verify proofs π of existence of a witness.

- verification algorithm V takes (x, π) as input, outputs "yes"/"no"
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))
- given w with C(x,w)=1, easy to generate π s.t. V(x, π)="yes"
- if x a "no" instance, computationally infeasible to find π s.t. V(x, π)="yes"

Application to validity rollups:

- each state root posted to L1 should be accompanied by SNARK proof π of existence of a witness to the corresponding state root verification input x
- L1 (smart contract) accepts state root ⇔ V(x, π) = "yes"

Measuring SNARK Performance

Application of SNARKs to validity rollups:

- each state root posted to L1 should be accompanied by SNARK proof π of existence of a witness to the corresponding state root verification input x
- L1 (smart contract) accepts state root ⇔ V(x, π) = "yes"

Measuring SNARK Performance

Application of SNARKs to validity rollups:

- each state root posted to L1 should be accompanied by SNARK proof π of existence of a witness to the corresponding state root verification input x
- L1 (smart contract) accepts state root ⇔ V(x, π) = “yes”
- note: will be feasible ⇔ π sufficiently short and V sufficient fast (and π not too hard for prover to generate, given a witness w)

Key SNARK metrics:

Measuring SNARK Performance

Application of SNARKs to validity rollups:

- each state root posted to L1 should be accompanied by SNARK proof π of existence of a witness to the corresponding state root verification input x
- L1 (smart contract) accepts state root ⇔ V(x, π) = "yes"
- note: will be feasible ⇔ π sufficiently short and V sufficient fast (and π not too hard for prover to generate, given a witness w)

Key SNARK metrics:

- proof length (<< running time of original computation C)

Measuring SNARK Performance

Application of SNARKs to validity rollups:

- each state root posted to L1 should be accompanied by SNARK proof π of existence of a witness to the corresponding state root verification input x
- L1 (smart contract) accepts state root ⇔ V(x, π) = "yes"
- note: will be feasible ⇔ π sufficiently short and V sufficient fast (and π not too hard for prover to generate, given a witness w)

Key SNARK metrics:

- proof length (<< running time of original computation C)

pay on L1
for this

Measuring SNARK Performance

Application of SNARKs to validity rollups:

- each state root posted to L1 should be accompanied by SNARK proof π of existence of a witness to the corresponding state root verification input x
- L1 (smart contract) accepts state root ⇔ V(x, π) = "yes"
- note: will be feasible ⇔ π sufficiently short and V sufficient fast (and π not too hard for prover to generate, given a witness w)

Key SNARK metrics:

- proof length (<< running time of original computation C)
- verifier time (<< running time of C) ⟵ pay on L1 for this

Measuring SNARK Performance

Application of SNARKs to validity rollups:

- each state root posted to L1 should be accompanied by SNARK proof π of existence of a witness to the corresponding state root verification input x
- L1 (smart contract) accepts state root ⇔ V(x, π) = "yes"
- note: will be feasible ⇔ π sufficiently short and V sufficient fast (and π not too hard for prover to generate, given a witness w)

Key SNARK metrics:

- proof length (<< running time of original computation C)
- verifier time (<< running time of C) ← pay on L1 for this
- prover time/memory (not too much worse than running C "natively")

Measuring SNARK Performance

Application of SNARKs to validity rollups:

- each state root posted to L1 should be accompanied by SNARK proof π of existence of a witness to the corresponding state root verification input x
- L1 (smart contract) accepts state root ⇔ V(x, π) = "yes"
- note: will be feasible ⇔ π sufficiently short and V sufficient fast (and π not too hard for prover to generate, given a witness w)

Key SNARK metrics:

- proof length (<< running time of original computation C)
- verifier time (<< running time of C) ← pay on L1 for this
- prover time/memory (not too much worse than running C "natively") ← off-chain

What Does “SNARK” Stand For?

What Does “SNARK” Stand For?

- “succinct”: length of proof of knowledge π << length of witness w and verifier time V(x, π) << running time of C(x,w)

What Does “SNARK” Stand For?

- “succinct”: length of proof of knowledge π << length of witness w and verifier time V(x, π) << running time of C(x,w)
- “noninteractive”: can just post π to L1 (“one and done”)
  - as opposed to interactive, e.g. repeated challenge-response

What Does “SNARK” Stand For?

- “succinct”: length of proof of knowledge π << length of witness w and verifier time V(x, π) << running time of C(x,w)
- “noninteractive”: can just post π to L1 (“one and done”)
  - as opposed to interactive, e.g. repeated challenge-response
- “argument”: (i.e., π really is a convincing proof)
  - completeness: if x a “yes” instance, there exists π with V(x, π)=“yes”
  - computational soundness: if x a “no” instance, computationally infeasible to find a π with V(x, π)=“yes” (cf., “proof” and “perfect soundness”)

What Does “SNARK” Stand For?

- “succinct”: length of proof of knowledge π << length of witness w and verifier time V(x, π) << running time of C(x,w)
- “noninteractive”: can just post π to L1 (“one and done”)
  - as opposed to interactive, e.g. repeated challenge-response
- “argument”: (i.e., π really is a convincing proof)
  - completeness: if x a “yes” instance, there exists π with V(x, π)=“yes”
  - computational soundness: if x a “no” instance, computationally infeasible to find a π with V(x, π)=“yes” (cf., “proof” and “perfect soundness”)
- “of knowledge”: π proves existence of a witness w, not just correctness of the computation C(x,w)
  - cf., verification of matrix multiplication

Do SNARKs Exist?

Do SNARKs Exist?

1990s (Killian, Micali): In principle, SNARKs exist.

Do SNARKs Exist?

1990s (Killian, Micali): In principle, SNARKs exist.

2020s (…): SNARKs can be made practical.

Do SNARKs Exist?

1990s (Killian, Micali): In principle, SNARKs exist.

2020s (…): SNARKs can be made practical.

Key ingredients: (cf., matrix multiplication)

- probabilistic verification (catches false claims e.g. 50% of the time)
- Fiat-Shamir heuristic "flattens" iterated checks into one-shot proof

Do SNARKs Exist?

1990s (Killian, Micali): In principle, SNARKs exist.

2020s (…): SNARKs can be made practical.

Key ingredients: (cf., matrix multiplication)

- probabilistic verification (catches false claims e.g. 50% of the time)
- Fiat-Shamir heuristic "flattens" iterated checks into one-shot proof

Question: how to probabilistically verify arbitrary computations?

The PCP Theorem

Amazing fact: *every* NP problem can be probabilistically verified.

The PCP Theorem

Amazing fact: *every* NP problem can be probabilistically verified.

PCP Theorem: (1992) for the satisfiability problem

The PCP Theorem

Amazing fact: *every* NP problem can be probabilistically verified.

PCP Theorem: (1992) for the satisfiability problem

Problem: 3-SAT

Input: A list of Boolean decision variables $x_1, x_2, \ldots, x_n$; and a list of constraints, each a disjunction of at most three literals.

Output: A truth assignment to $x_1, x_2, \ldots, x_n$ that satisfies every constraint, or a correct declaration that no such truth assignment exists.

For example, there's no way to satisfy all eight of the constraints

$$\begin{array}{cccc} x_1 \vee x_2 \vee x_3 & x_1 \vee \neg x_2 \vee x_3 & \neg x_1 \vee \neg x_2 \vee x_3 & x_1 \vee \neg x_2 \vee \neg x_3 \\ \neg x_1 \vee x_2 \vee x_3 & x_1 \vee x_2 \vee \neg x_3 & \neg x_1 \vee x_2 \vee \neg x_3 & \neg x_1 \vee \neg x_2 \vee \neg x_3, \end{array}$$

The PCP Theorem

Amazing fact: *every* NP problem can be probabilistically verified.

PCP Theorem: (1992) for the satisfiability problem (SAT), there is a format for purported proofs π of satisfiability and a verification algorithm V such that, for every SAT formula φ:

The PCP Theorem

Amazing fact: *every* NP problem can be probabilistically verified.

PCP Theorem: (1992) for the satisfiability problem (SAT), there is a format for purported proofs π of satisfiability and a verification algorithm V such that, for every SAT formula φ:

1. V makes O(1) random queries to learn bits of π, outputs “yes”/”no”.

The PCP Theorem

Amazing fact: *every* NP problem can be probabilistically verified.

PCP Theorem: (1992) for the satisfiability problem (SAT), there is a format for purported proofs π of satisfiability and a verification algorithm V such that, for every SAT formula φ:

1. V makes O(1) random queries to learn bits of π, outputs "yes"/"no".
2. If φ is satisfiable, there is a proof π s.t. Pr[V(φ, π)="yes"] = 1.

The PCP Theorem

Amazing fact: *every* NP problem can be probabilistically verified.

PCP Theorem: (1992) for the satisfiability problem (SAT), there is a format for purported proofs π of satisfiability and a verification algorithm V such that, for every SAT formula φ:

1. V makes O(1) random queries to learn bits of π, outputs "yes"/"no".
2. If φ is satisfiable, there is a proof π s.t. Pr[V(φ, π)="yes"] = 1.
3. If φ is not satisfiable, then for every alleged proof π, Pr[V(φ, π)="yes"] ≤ ½.

The PCP Theorem

Amazing fact: *every* NP problem can be probabilistically verified.

PCP Theorem: (1992) for the satisfiability problem (SAT), there is a format for purported proofs π of satisfiability and a verification algorithm V such that, for every SAT formula φ:

1. V makes O(1) random queries to learn bits of π, outputs "yes"/"no".
2. If φ is satisfiable, there is a proof π s.t. $\Pr[V(\varphi, \pi)=\text{"yes"}] = 1$.
3. If φ is not satisfiable, then for every alleged proof π, $\Pr[V(\varphi, \pi)=\text{"yes"}] \leq 1/2$. [repeat t times ➔ false positive probability $\leq 2^{-t}$]

The PCP Theorem

PCP Theorem: (1992) for the satisfiability problem (SAT), there is a format for purported proofs π of satisfiability and a verification algorithm V such that, for every SAT formula φ:

1. V makes O(1) random queries to learn bits of π, outputs "yes"/"no".
2. If φ is satisfiable, there is a proof π s.t. Pr[V(φ, π)="yes"] = 1.
3. If φ is not satisfiable, then for every alleged proof π, Pr[V(φ, π)="yes"] ≤ ½. [repeat t times ➔ false positive probability ≤ 2^{-t}]

Because SAT is NP-complete: *every* NP problem L can be likewise probabilistically verified. [Convert L to 3-SAT, use PCP theorem.]

PCP Theorem ➔ SNARKs

From the PCP theorem to a SNARK for state root verification:

PCP Theorem ➔ SNARKs

From the PCP theorem to a SNARK for state root verification:

- prover converts execution trace of SRV computation $C(x,\cdot)$ into an instance φ of 3-SAT [as in proof of Cook-Levin theorem]

PCP Theorem → SNARKs

From the PCP theorem to a SNARK for state root verification:

- prover converts execution trace of SRV computation C(x,·) into an instance φ of 3-SAT [as in proof of Cook-Levin theorem]
- prover knows witness w for x (i.e., correct Merkle proofs), computes PCP π for φ, forms Merkle tree T with leaves = bits of π

PCP Theorem ➔ SNARKs

From the PCP theorem to a SNARK for state root verification:

- prover converts execution trace of SRV computation C(x,·) into an instance φ of 3-SAT [as in proof of Cook-Levin theorem]
- prover knows witness w for x (i.e., correct Merkle proofs), computes PCP π for φ, forms Merkle tree T with leaves = bits of π
- SNARK proof = root of T + Merkle proofs for O(t) “random” bits of π
  - use Fiat-Shamir heuristic to derive which “random” bits of π to include
  - accept SNARK proof ⇔ corresponds to the transcript of an accepting computation for the PCP verifier