Bonus Lecture #5: SNARKs Under the Hood

COMS 4995-001: The Science of Blockchains

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

Tim Roughgarden

Goals for Bonus Lecture #5

1. Review of NP and SNARKs.
   - SNARK = succinct, noninteractive argument of knowledge
   - short (<< witness length) & easy-to-verify proofs of an NP statement
2. General probabilistic verification and the PCP Theorem.
   - every NP problem can be probabilistically verified
3. PCP Theorem ➔ SNARKs.
   - can derive SNARKS from one of the deepest results in theory CS
4. Bird's-eye view of modern SNARK constrictions.
   - front ends, back ends, polynomial commitments, polynomial IOPs

Review: Witnesses and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

Review: Witnesses and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- example: Sudoku

Review: Witnesses and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- example: Sudoku
- example: state root verification (SRV)

Review: Witnesses and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- example: Sudoku
- 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

Review: Witnesses and NP Statements

Recall: NP = problems that have efficiently verifiable solutions.

- example: Sudoku
- 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

Idea of a SNARK: proof that x a “yes” instance, with proof length << witness length and verification time << time to compute C(x,w).

Review: 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.

Review: 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”

Review: 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”
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))

Review: 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"
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))
- length of π << length of witness (ideally, length = O(log(RT of C)))

Review: 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”
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))
- length of π << length of witness (ideally, length = O(log(RT of C)))
- given w with C(x,w)=1, easy to generate π s.t. V(x, π)=“yes”
  - “prover time,” ideally O(RT of C) with non-astronomical hidden constant

Review: 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"
- running time of V << running time of C (ideally, RT of V = O(log(RT of C)))
- length of π << length of witness (ideally, length = O(log(RT of C)))
- given w with C(x,w)=1, easy to generate π s.t. V(x, π)="yes"
  - "prover time," ideally O(RT of C) with non-astronomical hidden constant
- 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

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)
  - ex: choose $x \in \{0,1\}^n$ at random, reject if $C \cdot x \neq A \cdot (B \cdot x)$

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)
  - ex: choose $x \in \{0,1\}^n$ at random, reject if $C \cdot x \neq A \cdot (B \cdot x)$
- Fiat-Shamir heuristic "flattens" iterated checks into one-shot proof
  - ex: set $x_i = h(C||i)$ for each i=1,2,…,t [h = cryptographic hash function]

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)
  - ex: choose $x \in \{0,1\}^n$ at random, reject if $C \cdot x \neq A \cdot (B \cdot x)$
- Fiat-Shamir heuristic "flattens" iterated checks into one-shot proof
  - ex: set $x_i = h(C||i)$ for each i=1,2,…,t [h = cryptographic hash function]

Question: how to probabilistically verify arbitrary computations?

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?

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

Recall: The 3-SAT Problem

Recall: The 3-SAT 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

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

Recall: The 3-SAT 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.

decision version: output "yes" if there exists a satisfying assignment and "no" otherwise

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

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

Recall: The 3-SAT 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.

decision version: output "yes" if there exists a satisfying assignment and "no" otherwise

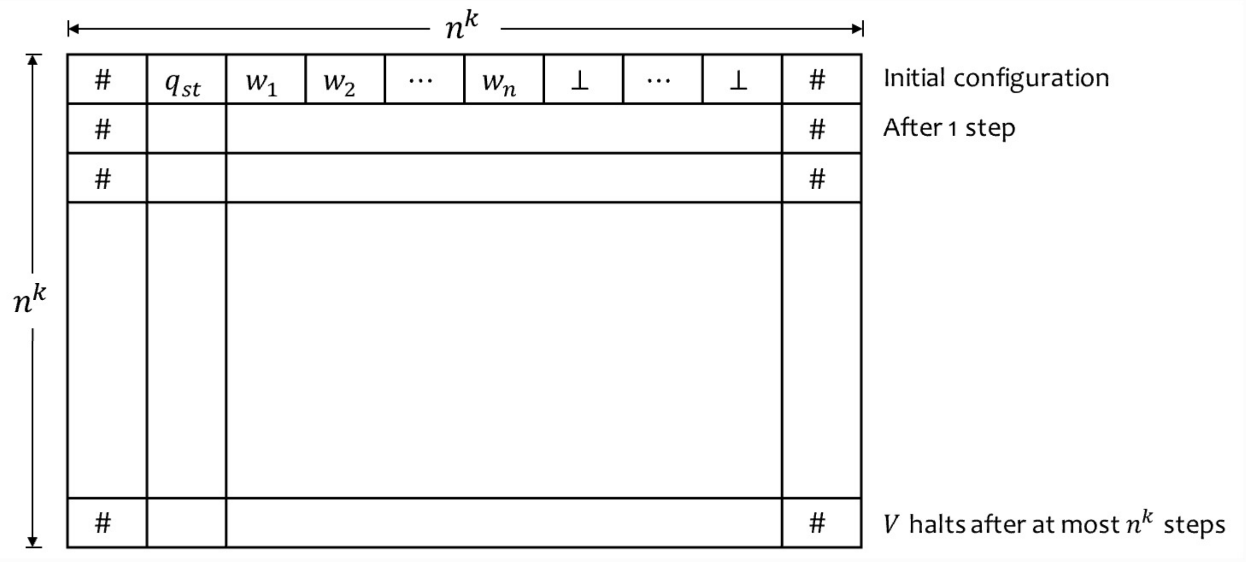
Cook-Levin Theorem: 3-SAT is NP-complete.

- i.e., *every* NP problem can be encoded as 3-SAT.

Proof of the Cook-Levin Theorem

Cook-Levin Theorem: 3-SAT is NP-complete.

- i.e., *every* NP problem can be encoded as 3-SAT
- proof idea: decision variables = state of memory at each time step of computation; constraints = computation proceeds according to C



Recall: The 3-SAT 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.

decision version: output "yes" if there exists a satisfying assignment and "no" otherwise

Cook-Levin Theorem: 3-SAT is NP-complete.

- i.e., *every* NP problem can be encoded as 3-SAT.

Recall: The 3-SAT 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.

decision version: output "yes" if there exists a satisfying assignment and "no" otherwise

Cook-Levin Theorem: 3-SAT is NP-complete.

- i.e., *every* NP problem can be encoded as 3-SAT.

Upshot: to probabilistically verify every NP problem, enough to probabilistically verify 3-SAT.

– verify an arbitrary NP problem by first converting it to 3-SAT

The PCP Theorem

PCP Theorem: (1992) there is a format for purported proofs y and a PCP verifier V such that, for every 3-SAT formula φ:

The PCP Theorem

PCP Theorem: (1992) there is a format for purported proofs y and a PCP verifier V such that, for every 3-SAT formula φ:

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

The PCP Theorem

PCP Theorem: (1992) there is a format for purported proofs y and a PCP verifier V such that, for every 3-SAT formula φ:

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

The PCP Theorem

PCP Theorem: (1992) there is a format for purported proofs y and a PCP verifier V such that, for every 3-SAT formula φ:

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

The PCP Theorem

PCP Theorem: (1992) there is a format for purported proofs y and a PCP verifier V such that, for every 3-SAT formula φ:

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

Fix: an NP problem (e.g., state root verification) with witness-checking algorithm C(·,·). [C a function of input x, alleged witness w]

PCP Theorem ➔ SNARKs

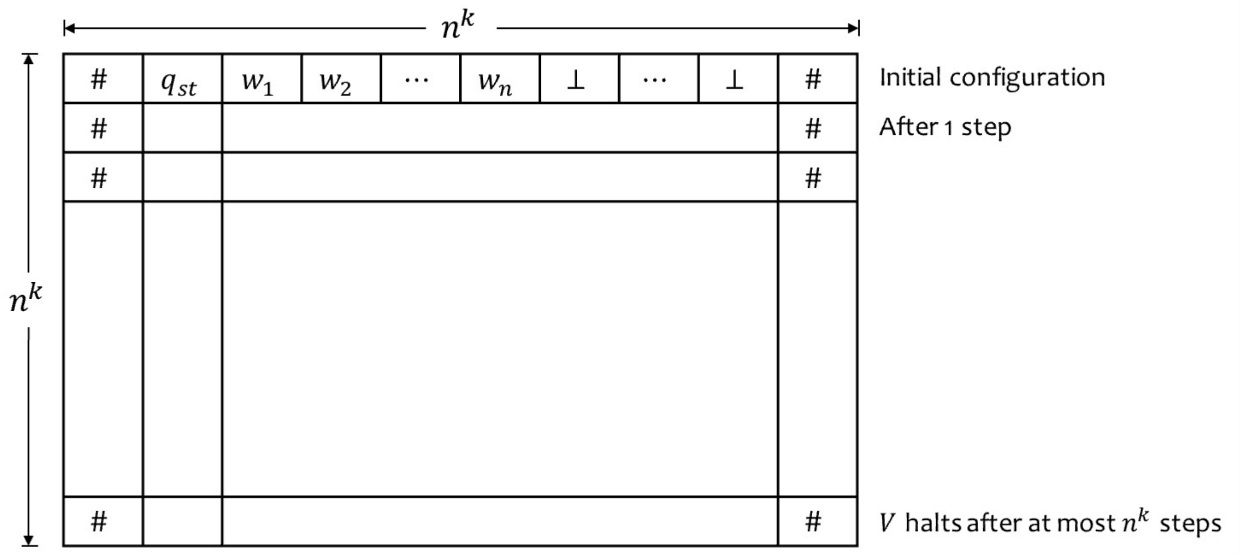
Fix: an NP problem (e.g., state root verification) with witness-checking algorithm $C(\cdot,\cdot)$. [C a function of input x, alleged witness w]

- Cook-Levin theorem ➔ there is a 3-SAT formula φ s.t. satisfying truth assignments of φ $\Leftrightarrow$ accepting computations of C

PCP Theorem ➔ SNARKs

Fix: an NP problem (e.g., state root verification) with witness-checking algorithm C(·,·). [C a function of input x, alleged witness w]

- Cook-Levin theorem ➔ there is a 3-SAT formula φ s.t. satisfying truth assignments of φ ⇔ accepting computations of C
  - decision variables of φ = bits of x, bits of w, various auxiliary variables



PCP Theorem ➔ SNARKs

Fix: an NP problem (e.g., state root verification) with witness-checking algorithm C(·,·). [C a function of input x, alleged witness w]

- Cook-Levin theorem ➔ there is a 3-SAT formula φ s.t. satisfying truth assignments of φ ⬄ accepting computations of C
  - decision variables of φ = bits of x, bits of w, various auxiliary variables

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

PCP Theorem ➔ SNARKs

Fix: an NP problem (e.g., state root verification) with witness-checking algorithm C(·,·). [C a function of input x, alleged witness w]

- Cook-Levin theorem ➔ there is a 3-SAT formula φ s.t. satisfying truth assignments of φ ⬄ accepting computations of C
  - decision variables of φ = bits of x, bits of w, various auxiliary variables

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

- assume matrices A,B known (e.g., already posted to blockchain)

PCP Theorem ➔ SNARKs

Fix: an NP problem (e.g., state root verification) with witness-checking algorithm C(·,·). [C a function of input x, alleged witness w]

- Cook-Levin theorem ➔ there is a 3-SAT formula φ s.t. satisfying truth assignments of φ ⇔ accepting computations of C
  - decision variables of φ = bits of x, bits of w, various auxiliary variables

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

- assume matrices A,B known (e.g., already posted to blockchain)
- prover posts C, allegedly the product of A and B

PCP Theorem ➔ SNARKs

Fix: an NP problem (e.g., state root verification) with witness-checking algorithm C(·,·). [C a function of input x, alleged witness w]

- Cook-Levin theorem ➔ there is a 3-SAT formula φ s.t. satisfying truth assignments of φ ⇔ accepting computations of C
  – decision variables of φ = bits of x, bits of w, various auxiliary variables

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

– assume matrices A,B known (e.g., already posted to blockchain)

– prover posts C, allegedly the product of A and B

– verifier (e.g., L1) derives x_i from h(A || B || C || i) for i=1,2,…,t [e.g., h=SHA-256]

PCP Theorem ➔ SNARKs

Fix: an NP problem (e.g., state root verification) with witness-checking algorithm C(·,·). [C a function of input x, alleged witness w]

- Cook-Levin theorem ➔ there is a 3-SAT formula φ s.t. satisfying truth assignments of φ ⇔ accepting computations of C
  - decision variables of φ = bits of x, bits of w, various auxiliary variables

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

- assume matrices A,B known (e.g., already posted to blockchain)
- prover posts C, allegedly the product of A and B
- verifier (e.g., L1) derives x_i from h(A || B || C || i) for i=1,2,…,t [e.g., h=SHA-256]
- verifier accepts C ⇔ $C \cdot x_i = A \cdot (B \cdot x_i)$ for each i=1,2,…,t

PCP Theorem ➔ SNARKs (con'd)

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

- assume matrices A,B known (e.g., already posted to blockchain)
- prover posts C, allegedly the product of A and B
- verifier (e.g., L1) derives x_i from h(A ll B ll C ll i) for i=1,2,…,t [e.g., h=SHA-256]
- verifier accepts C ⇔ $C \cdot x_i = A \cdot (B \cdot x_i)$ for each i=1,2,…,t

Arbitrary NP problem (first attempt):

PCP Theorem ➔ SNARKs (con’d)

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

- assume matrices A,B known (e.g., already posted to blockchain)
- prover posts C, allegedly the product of A and B
- verifier (e.g., L1) derives x_i from h(A II B II C II i) for i=1,2,…,t [e.g., h=SHA-256]
- verifier accepts C ⇔ $C \cdot x_i = A \cdot (B \cdot x_i)$ for each i=1,2,…,t

Arbitrary NP problem (first attempt):

- assume input x is known (e.g., state roots + txs posted to L1)

PCP Theorem ➔ SNARKs (con'd)

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

- assume matrices A,B known (e.g., already posted to blockchain)
- prover posts C, allegedly the product of A and B
- verifier (e.g., L1) derives x_i from h(A II B II C II i) for i=1,2,…,t [e.g., h=SHA-256]
- verifier accepts C ⇔ $C \cdot x_i = A \cdot (B \cdot x_i)$ for each i=1,2,…,t

Arbitrary NP problem (first attempt):

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)

PCP Theorem ➔ SNARKs (con'd)

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

- assume matrices A,B known (e.g., already posted to blockchain)
- prover posts C, allegedly the product of A and B
- verifier (e.g., L1) derives x_i from h(A || B || C || i) for i=1,2,…,t [e.g., h=SHA-256]
- verifier accepts C ⇔ $C \cdot x_i = A \cdot (B \cdot x_i)$ for each i=1,2,…,t

Arbitrary NP problem (first attempt):

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of "random" queries to y from h(x || y || i) [for i=1,2,…,t]

PCP Theorem ➔ SNARKs (con'd)

Recall: verifying matrix multiplication (in $O(n^2)$ rather than $O(n^3)$ time)

- assume matrices A,B known (e.g., already posted to blockchain)
- prover posts C, allegedly the product of A and B
- verifier (e.g., L1) derives x_i from h(A ll B ll C ll i) for i=1,2,…,t [e.g., h=SHA-256]
- verifier accepts C ⇔ $C \cdot x_i = A \cdot (B \cdot x_i)$ for each i=1,2,…,t

Arbitrary NP problem (first attempt):

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of "random" queries to y from h(x ll y ll i) [for i=1,2,…,t]
- verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

PCP Theorem ➔ SNARKs (con'd)

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of "random" queries to y from h(x ll y ll i) [for i=1,2,…,t]
- verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

Good news:

PCP Theorem ➔ SNARKs (con'd)

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of "random" queries to y from h(x || y || i) [for i=1,2,…,t]
- verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

Good news: (i) same correctness guarantees as matrix multiplication

PCP Theorem ➔ SNARKs (con'd)

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of "random" queries to y from h(x ll y ll i) [for i=1,2,…,t]
- verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

Good news: (i) same correctness guarantees as matrix multiplication

(ii) fast verification [O(t) evaluations of h, random accesses to y]

PCP Theorem ➔ SNARKs (con'd)

– assume input x is known (e.g., state roots + txs posted to L1)

– prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)

– verifier derives set S_i of "random" queries to y from h(x ll y ll i) [for i=1,2,…,t]

– verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

Good news: (i) same correctness guarantees as matrix multiplication (ii) fast verification [O(t) evaluations of h, random accesses to y]

Bad news: proof y is not succinct (at least as large as witness length).

– y is essentially a redundantly encoded satisfying truth assignment

PCP Theorem ➔ SNARKs (con'd)

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of "random" queries to y from h(x II y II i) [for i=1,2,…,t]
- verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

Revised attempt: instead of posting y:

PCP Theorem ➔ SNARKs (con'd)

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of "random" queries to y from h(x II y II i) [for i=1,2,…,t]
- verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

Revised attempt: instead of posting y:

- prover forms a Merkle tree T with leaves = bits of y, posts root r of T

PCP Theorem ➔ SNARKs (con’d)

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of “random” queries to y from h(x ll y ll i) [for i=1,2,…,t]
- verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

Revised attempt: instead of posting y:

- prover forms a Merkle tree T with leaves = bits of y, posts root r of T
- also posts Merkle proofs revealing the answer to each query in each S_i, where S_i derived from h(x ll y ll i) [or even h(x ll y ll Merkle pfs for $S_1,\ldots,S_{i-1}$)]
  - O(t) Merkle proofs in all

PCP Theorem ➔ SNARKs (con'd)

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts y, allegedly a PCP proof for a satisfying assignment of φ_x (i.e., φ with bits of x fixed accordingly)
- verifier derives set S_i of "random" queries to y from h(x ll y ll i) [for i=1,2,…,t]
- verifier accepts y ⇔ for each i, PCP verifier would accept y with queries S_i

Revised attempt: instead of posting y:

- prover forms a Merkle tree T with leaves = bits of y, posts root r of T
- also posts Merkle proofs revealing the answer to each query in each S_i, where S_i derived from h(x ll y ll i) [or even h(x ll y ll Merkle pfs for $S_1,\ldots,S_{i-1}$)]

Result: correctness same as before (assuming no hash fn collisions), proof size now O(log RT(C)). [assuming t = O(1)]

PCP Theorem ➔ SNARKs (con'd)

SNARK for an arbitrary NP problem:

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts r, allegedly the root of a Merkle whose leaves encode a PCP proof for a satisfying assignment of φ_x (i.e., φ w/bits of x fixed accordingly)
- prover posts t · q Merkle proofs (q = # of queries made by PCP verifier)
- verification: for i=1,2,…,t:
  - derive set S_i of queries to y from h(x ll y ll Merkle pfs for $S_1,\ldots,S_{i-1}$)
  - accept y ⇔ for each i, PCP verifier would accept y with queries S_i

Conclusion: SNARKs exist for arbitrary NP problems!

PCP Theorem ➔ SNARKs (con'd)

SNARK for an arbitrary NP problem:

- assume input x is known (e.g., state roots + txs posted to L1)
- prover posts r, allegedly the root of a Merkle whose leaves encode a PCP proof for a satisfying assignment of φ_x (i.e., φ w/bits of x fixed accordingly)
- prover posts t · q Merkle proofs (q = # of queries made by PCP verifier)
- verification: for i=1,2,…,t:
  - derive set S_i of queries to y from h(x ll y ll Merkle pfs for $S_1,\ldots,S_{i-1}$)
  - accept y ⇔ for each i, PCP verifier would accept y with queries S_i

Conclusion: SNARKs exist for arbitrary NP problems!

- not immediately practical (too much work to generate PCP proof y)
- but the conceptual basis for modern SNARK constructions

Modern SNARK Constructions

SNARK ingredients:

Modern SNARK Constructions

SNARK ingredients:

- front end: converts NP problem of interest (e.g., state root verification) into a form amenable to probabilistic verification

Modern SNARK Constructions

SNARK ingredients:

- front end: converts NP problem of interest (e.g., state root verification) into a form amenable to probabilistic verification
  - above: front end = proof of Cook-Levin theorem, output = 3-SAT formula
  - modern variants: output = arithmetic circuit, of a “constraint system” (e.g., R1CS)

Modern SNARK Constructions

SNARK ingredients:

- front end: converts NP problem of interest (e.g., state root verification) into a form amenable to probabilistic verification
  - above: front end = proof of Cook-Levin theorem, output = 3-SAT formula
  - modern variants: output = arithmetic circuit, of a “constraint system” (e.g., R1CS)
- back end: probabilistic verification + Fiat-Shamir heuristic

Modern SNARK Constructions

SNARK ingredients:

- front end: converts NP problem of interest (e.g., state root verification) into a form amenable to probabilistic verification
  - above: front end = proof of Cook-Levin theorem, output = 3-SAT formula
  - modern variants: output = arithmetic circuit, of a “constraint system” (e.g., R1CS)
- back end: probabilistic verification + Fiat-Shamir heuristic
  - above: use PCPs and Merkle trees

Modern SNARK Constructions

SNARK ingredients:

- front end: converts NP problem of interest (e.g., state root verification) into a form amenable to probabilistic verification
  - above: front end = proof of Cook-Levin theorem, output = 3-SAT formula
  - modern variants: output = arithmetic circuit, of a “constraint system” (e.g., R1CS)
- back end: probabilistic verification + Fiat-Shamir heuristic
  - above: use PCPs and Merkle trees
  - modern variants: use “interactive oracle proofs (IOPs)” rather than a one-shot PCP
    - allows prover to be more adaptive (pre-Fiat-Shamir) and do less work overall

Modern SNARK Constructions

SNARK ingredients:

- front end: converts NP problem of interest (e.g., state root verification) into a form amenable to probabilistic verification
  - above: front end = proof of Cook-Levin theorem, output = 3-SAT formula
  - modern variants: output = arithmetic circuit, of a “constraint system” (e.g., R1CS)
- back end: probabilistic verification + Fiat-Shamir heuristic
  - above: use PCPs and Merkle trees
  - modern variants: use “interactive oracle proofs (IOPs)” rather than a one-shot PCP
    - allows prover to be more adaptive (pre-Fiat-Shamir) and do less work overall
  - also: often use “polynomial commitments” (e.g., KZG) instead of Merkle trees

Modern SNARK Constructions

SNARK ingredients:

- front end: converts NP problem of interest (e.g., state root verification) into a form amenable to probabilistic verification
  - above: front end = proof of Cook-Levin theorem, output = 3-SAT formula
  - modern variants: output = arithmetic circuit, of a “constraint system” (e.g., R1CS)
- back end: probabilistic verification + Fiat-Shamir heuristic
  - above: use PCPs and Merkle trees
  - modern variants: use “interactive oracle proofs (IOPs)” rather than a one-shot PCP
  - also: often use “polynomial commitments” (e.g., KZG) instead of Merkle trees

Practical SNARKs: require careful joint optimization of the front end, the polynomial commitment scheme, and the polynomial IOP.