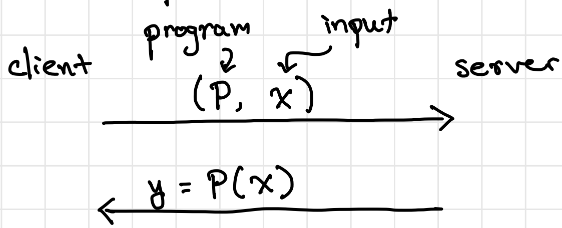
So far, we have focused on proving properties in a privacy-preserving manner. Next, we will look at achieving <u>short</u> proofs that are efficient to verify.

<u>Application</u>: verifiable computation



How do we know that the server computed the correct value?
↳ Can provide a proof $y = P(x)$. To be useful, checking the proof should be much faster than computing P.

<u>Main primitive</u>: aggregation scheme for proofs – batch argument for NP

<u>Setting</u>: given T statements $x_1, \dots, x_T$ and circuit C, show that all of the statements are true (i.e., $\exists w_i : C(x_i, w_i) = 1$ for all $i \in [T]$).

<u>Naively</u>: Give out T proofs, one for each statement x_i.

<u>Goal</u>: Can we do better. Namely, can we batch prove T statements with a proof of size $o(T)$?

For now, we will not worry about zero-knowledge. Turns out that succinctness can be used to achieve zero-knowledge.

Batch arguments provide a way to prove $(x_1, \ldots, x_T)$ are all true with a proof whose size scales with $|C|$ – the size of a <u>single</u> proof.

But this is <u>not</u> the setting of verifiable computation:

prove that $y = P(x)$ with a proof π that is much shorter and faster to check than computing the program P

Turns out to be sufficient. In the following, we will model P as a "RAM program" (random access machine).

<u>RAM model</u>: closer model of typical computer
- Program has access to M bits of memory and $O(1)$ bits of local state
- Initial contents of memory is the input
- Program consists of a sequence of instructions:
  - Read instruction: reads 1 bit of memory at any address and update local state
  - Write instruction: write 1 bit to any address and update local state
- Output is the contents of the memory

Model RAM program (with M bits of memory) execution on input x as follows:
- Initial state: memory contents is x $[M_0 = x, st_0]$ ← contents of local registers
- On each step of the computation

  $st_i, M_i \mapsto st_{i+1}, M_{i+1}$

Suppose there are T steps in the computation

$$\underbrace{(st_0, M_0)}_{\substack{\text{input state} \\ \text{(known to verifier)}}} \to (st_1, M_1) \to \cdots \to \underbrace{(st_T, M_T)}_{\substack{\text{purported output state} \\ \text{(known to verifier)}}}$$

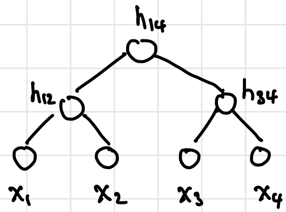
<u>Idea</u>: To prove correct evaluation of $P(x)$, give a BARG proof that each of the above steps is implemented correctly:
- Define $\mathrm{IsValid}((st_i, M_i), (st_{i+1}, M_{i+1}))$ to be predicate that checks correct execution
- Statements will be $((st_0, M_0), (st_1, M_1)), ((st_1, M_1), (st_2, M_2)), \ldots, ((st_{T-1}, M_{T-1}), (st_T, M_T))$
- Proof is BARG proof that IsValid holds for each statement

Many problems: 1. Verifier does not know intermediate states (st_i, M_i).
Communicating these defeats the whole point of delegation.

2. Size of BARG proof scales with size of circuit for checking one step of computation. This circuit would need to take the contents of memory as input. As such, BARG proof may not be succinct any more!

We need a compressed representation of the memory.

We can represent the contents of the memory by a Merkle hash:



Leaves are labeled by input values
Internal nodes are labeled by the hash of their children (under a collision-resistant hash function)
Hash value is value of the <u>root</u> of the tree

Let h be the root of the Merkle tree (on n values $x_1, \ldots, x_n$)
- Can open up the value x_i at position $i \in [n]$ by revealing sibling nodes along the path (i.e., can authenticate a value with $O(\lambda \log n)$ bits where λ is the output length of the hash function).
- Given a hash h, cannot open any index $i \in [n]$ to two different values $x_i \neq x_i'$ (otherwise, breaks collision resistance of hash function)

Hashing provides a succinct representation. Instead of setting the statements to be (st_i, M_i), we do the following:

1. Prover first computes $(st_0, M_0), \ldots, (st_T, M_T)$ and hashes all of these values to a h_{state}.
2. Prover gives the hashed value to the verifier. Now, the IsValid predicate checks the following:

This is now an NP relation!
- Statement is an index i
- Witness is an opening of h_{state} to (M_i, st_i) and (M_{i+1}, st_{i+1}) with respect to h_{state}.
- IsValid checks validity of the opening and that the state update implemented correctly.

No longer need to communicate intermediate states to the verifier, only the hash of <u>all</u> of them.

However, the IsValid circuit is still <u>large</u>: as large as the memory.

Need to compress memory!

Solution: hash <u>again</u>!

Each read/write operation is <u>local</u> (affects only <u>one</u> entry)
- Suppose h is a hash of the memory contents.
- Read operation: give an opening at index i to value x_i with respect to h
- Write operation: give an opening at index i to current value x_i
  opening suffices to compute updated hash with new value at index i

Updated protocol:
1. Prover computes $(st_0, M_0), \ldots, (st_T, M_T)$ as before. For each i, let h_i be the hash of the memory contents M_i.
2. Prover commits to $(st_0, h_0), \ldots, (st_T, h_T)$ with hash h_{state}.
3. Prover gives BARG proof for following IsValid relation:
   - Statement: i
   - Witness: Openings of h_{state} at indices i and $i+1$ to (st_i, h_i) and (st_{i+1}, h_{i+1}) and openings for h_i, h_{i+1} at associated index
   - IsValid checks validity of openings for h_{state} and depending on operation
     - Read: $h_i = h_{i+1}$
     - Write: h_{i+1} is correctly derived from h_i and bit written

   and st_{i+1} is correctly computed from st_i

Size of proof = size of BARG proof
- IsValid circuit only checks local openings for hash functions and validity of state transition
- If size of local state is constant (or poly(λ)), then $|\text{IsValid}| = \text{poly}(\lambda, \log T, \log M)$.

Total proof size is then poly$(\lambda, \log T, \log M)$, which is succinct, as required.

To prove security: we need that hash function be somewhere extractable (stronger than collision resistant), but can be built from similar techniques as for BARGs

Some extensions / applications:

- Functional commitment: commit to an input x and open to $f(x)$ where f is an arbitrary function [require commitment + openings be short]
  - ↳ follows similarly as above by viewing x as initial contents of memory and take BARG proof as the opening [different proof strategy needed to prove security since verifier does <u>not</u> know x in this case and <u>cannot</u> certify the initial hash]
  - ↳ can also realize algebraically with a construction where commitment size and opening size consists of a <u>constant</u> number of group elements

- Homomorphic signature: given a signature on a message m, derive signature on $f(m)$ for arbitrary f
  - ↳ Sign message m and the hash of m. Signature on $f(m)$ is the hash, the signature on the hash, the value $y = h(m)$ and a proof that $y = h(m)$ with respect to $h(m)$.
  - ↳ Observe that neither final signature nor verification time depend on original message size or on the running time of f (modulo log factors).