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

**Application:** verifiable computations

```
client    program       input
             (P, x)     →

server

y = P(x)
```

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

**Main primitive:** aggregation scheme for proofs - batch argument for NP

**Setting:** given $T$ statements $X_1, ..., X_T$ and circuit $C$, show that all of the statements are true (i.e., $\exists w: C(x_i, w) = 1$ for all $i \in \{1, ..., T\}$).

**Naively:** Give out $T$ proofs, one for each statement $X_i$.

**Goal:** 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 \(1/\epsilon\) - the size of a single proof.

But this is not the setting of verifiable computation:

prove that \(y = P(x)\) with a proof \(\pi\) 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).

**RAM model**: 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 \(x\) \([M_0 = x, \ st_0]\)
- On each step of the computation \(\text{contents of local registers}\)
  \(st_i, M_i \rightarrow st_{i+1}, M_{i+1}\)

Suppose there are \(T\) steps in the computation

\[
\begin{align*}
(St_0, M_0) \rightarrow (st_1, M_1) \rightarrow \cdots \rightarrow (st_T, M_T)
\end{align*}
\]

\(\text{input state (known to verifier)}\)
\(\text{purported output state (known to verifier)}\)

Idea: To prove correct evaluation of \(P(x)\), give a BARG proof that each of the above steps is implemented correctly:

- Define \(\text{IsValid}((st_i, M_i), (st_{i+1}, M_{i+1}))\) to be predicate that checks correct execution
- Statements will be \((St_0, M_0), (st_1, M_1), (st_2, M_2), \ldots, (st_T, M_T)\)
- Proof is BARG proof that \(\text{IsValid}\) holds for each statement
Many problems: 1. Verifier does not know intermediate states \((S_i, M_i)\).
   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:

\[
\begin{align*}
    h_{14} & \mid h_{15} \\
    h_{15} & \mid h_{16} \\
    X_1 & \mid X_2 \\
    X_3 & \mid X_4
\end{align*}
\]

- 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 root 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 \( O(x \log n) \) bits where \( x \) 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 \((S_i, M_i)\), we do the following:

1. Prover first computes \((S_0, M_0), \ldots, (S_{i-1}, M_{i-1})\) and hashes all of these values to a state.
2. Prover gives the hashed value to the verifier. Now, the IsValid predicate checks the following:
   - Statement is an index \( i \)
   - Witness is an opening of state to \((M_i, S_i)\) and \((M_{i+1}, S_{i+1})\) with respect to 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 all of them.

However, the IsValid circuit is still large: as large as the memory.
Need to compress memory!

Solution: hash again!

Each read/write operation is local (affects only one 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 \((s_0, M_0), ..., (s_T, M_T)\) as before. For each \( i \), let \( h_i \) be the hash of the memory contents \( M_i \).
2. Prover commits to \((s_0, h_0), ..., (s_T, h_T)\) with hash \( h \) state.
3. Prover gives BARG proof for following \( \text{IsValid} \) relation:
   - Statement: \( i \)
   - Witness: Openings of \( h \) state at indices \( i \) and \( i+1 \) to \((s_i, h_i)\) and \((s_{i+1}, h_{i+1})\)
     and openings for \( h_i, h_{i+1} \) at associated index
   - \( \text{IsValid} \) 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 \( s_i \) written
       and \( s_{i+1} \) is correctly computed from \( s_i \)

Size of proof = size of BARG proof
- \( \text{IsValid} \) circuit only checks local openings for hash functions and validity of state transition
- If size of local state is constant (or poly \((\lambda)\)), then \( |\text{IsValid}| = \text{poly}(\lambda, \log T, \log M) \).
Total proof size is then \( \text{poly}(\lambda, \log T, \log M) \), 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 not know $x$ in this case and cannot certify the initial hash.]
  - Can also realize algebraically with a construction where commitment size and opening size consists of a constant 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).