Scaling Verifiable Computation Using Efficient Set Accumulators

USENIX Security, 2020

Alex Ozdemir*, Riad Wahby*, Barry Whitehat^, Dan Boneh*

*Stanford   ^Unaffiliated
Verifiable Storage

- Represent a large storage (e.g. array) with a small digest
- Verifiably read and update the digest

\[ d \leftarrow \text{Digest}(A) \]

**Prover** \((A, d)\)
- \(v \leftarrow A[i]\)
- \(A[i_w] \leftarrow v_w\)

**Verifier** \((d)\)
- \(\text{Verify}_{\text{read}}(d, i, v, \pi_r)\)
- \(\text{Verify}_{\text{update}}(d, i_w, v_w, d', \pi_w)\)

Application: Verifiable Outsourcing (e.g. smart contracts)

Goal: Efficient Verification!
Outline

• Merkle Trees (existing approach)
• RSA Accumulators (proposed approach)
• Our Work:
  • Implementing RSA Accumulators
  • Demonstrating that they are cheaper in some situations
Computational Model

• Inherited from verifiable outsourcing

• The *arithmetic constraint* computational model (“constraints”)
  • Data encoded in a large *finite field* \((\text{integers mod } p, p \approx 2^{256})\)
  • Constraints are expressed as equations of sums & products in the field
    • One multiplication per constraint!
    • Goal: minimize the number of constraints

• The prover can provide *advice*
  • E.g. the inverse of a field element.
    • Computable using Fermat’s little theorem (many constraints)
    • Checkable using 1 constraint.
Merkle Trees

• Based on a hash function $H : F \times F \rightarrow F$
  • Collision-Resistant
• Reduce the array to a single value with a hash-tree
• Proofs based on paths in the tree

Verification cost: $k \log m$ hashes
for $k$ updates and a storage of capacity $m$. 
RSA Accumulators

• Based on RSA groups
  • The integers modulo $pq$: the produce of two unknown primes.
  • Hard to compute roots.
    • $x^n$ is easy, $n\sqrt{x}$ is hard.

• The digest of an RSA Accumulator is

$$d = g^\prod_i h(x_i)$$

• The stored elements
  - Fixed generator
  - A (special) hash function
RSA Accumulator Proofs

- Insertion proof:
  - Verifier checks an exponentiation

- Removal proof:
  - Insertion in reverse

- Membership proof:
  - A removal proof, but the new digest is forgotten
  - Sound because computing roots is hard!

- Batches require a single exponentiation [BBF 18]/[Wes 18]
  - Requires a hash function to prime numbers (for non-interactivity)

\[ d' = d^{h(x)} \]

Verification cost: **k hashes + 1 exponentiation**
for \( k \) updates and a storage of capacity \( m \).
Traditional Hash-to-Prime

- Rejection sampling of primes
- Miller Rabin primality test
  - Probabilistic!
  - $2^{-\lambda}$ soundness uses $O(\lambda)$, $\tilde{O}(\lambda)$-bit exponentiations
  - Many constraints

procedure `HashToPrime(x)`: 
  
  $g \leftarrow PRG(\text{seed} = x)$
  
  while $g$.output() is composite:
    
    $g$.advance()
  
  Return $g$.output()
Pocklington Prime Generation

- **Pocklington's criterion:**
  - If
    - $p$ is prime
    - $n < p$
    - $\exists a. a^{np} \equiv_{np+1} 1 \land \gcd(a^n - 1, np + 1) = 1$
  - Then $np + 1$ is prime
- **Basis for a recursive primality certificate**
  - Idea: Rejection sampling of prime certificates

Many fewer constraints than Miller-Rabin, and provably prime
Other Techniques and Tricks

• Multiprecision arithmetic in constraints
  • Based on xjSnark [KPS 18]

• A new hash function, conjectured to be division-intractable

• Precise semantics for batching dependent accesses.
Evaluation

• Implementation in Bellman, using Groth16.
• Consider storage of varying size
• Perform varying numbers of swaps (remove x, add y)
• Measure constraints
• Crossover occurs at a few thousand operations
Summary

Research Question
Do RSA accumulators use fewer constraints than Merkle Trees?

Techniques
• Multiprecision arithmetic
• Division-intractable hashing
• Hashing to prime numbers
• Semantics of dependent accesses

Implementation: github.com/alex-ozdemir/bellman-bignat