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

$$d \leftarrow Digest(A)$$

<u>Prover(A, d)</u>		<u>Verifier(d)</u>
$v \leftarrow A[i]$	i, v, π_r →	$Verify_{read}(d, i, v, \pi_r)$
$A[i_w] \leftarrow v_w$	d', i_w, v_w, π_w →	$Verify_{update}(d, i_w, v_w, d', \pi_w)$

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

Goal: Efficient Verification!

Outline

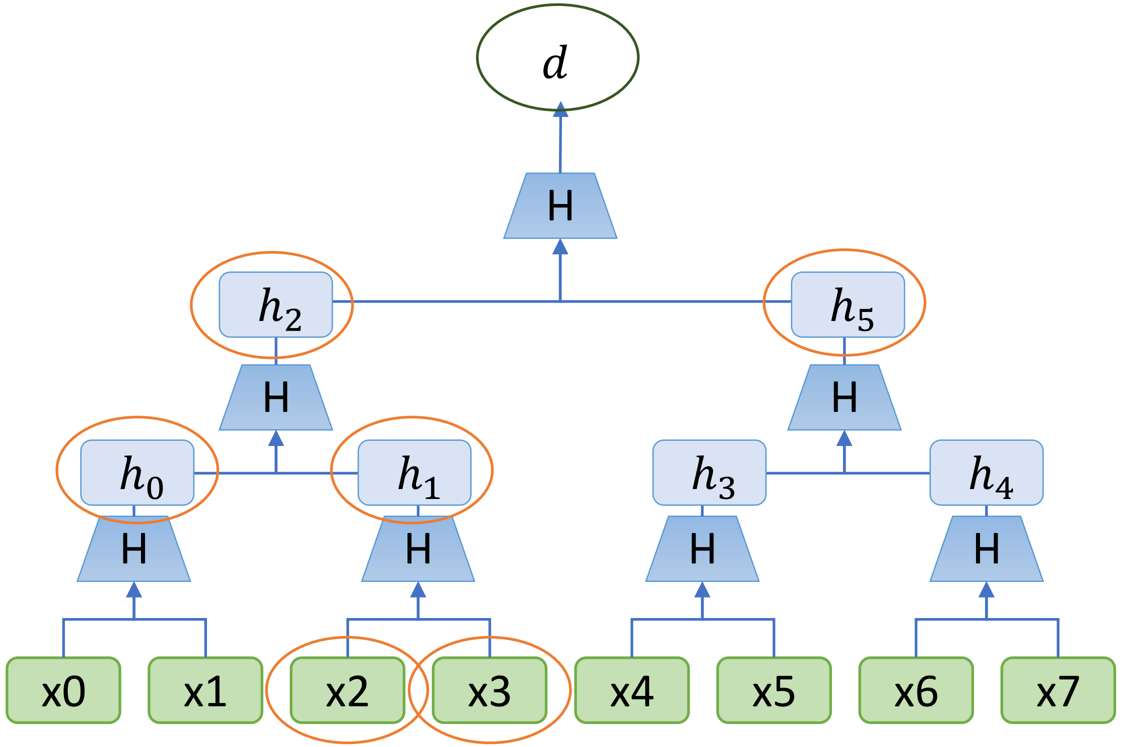
- Merkle Trees (existing approach)
- RSA Accumulators (proposed approach)
- Our Work:
  - Implementing RSA Accumulators
  - Demonstrating that they are cheaper in some situations

Computational Model

- Inherited from verifiable outsourcing
- The ***arithmetic constraint*** computational model ("constraints")
  - Data encoded in a large **finite field** (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: $\boldsymbol{k \log m}$ **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, $\sqrt[n]{x}$ 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 expontiaion

$$d' = d^{h(x)}$$

- 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)

Verification cost: $\boldsymbol{k}$ **hashes + 1 exponentiation**
for k updates and a storage of capacity m.

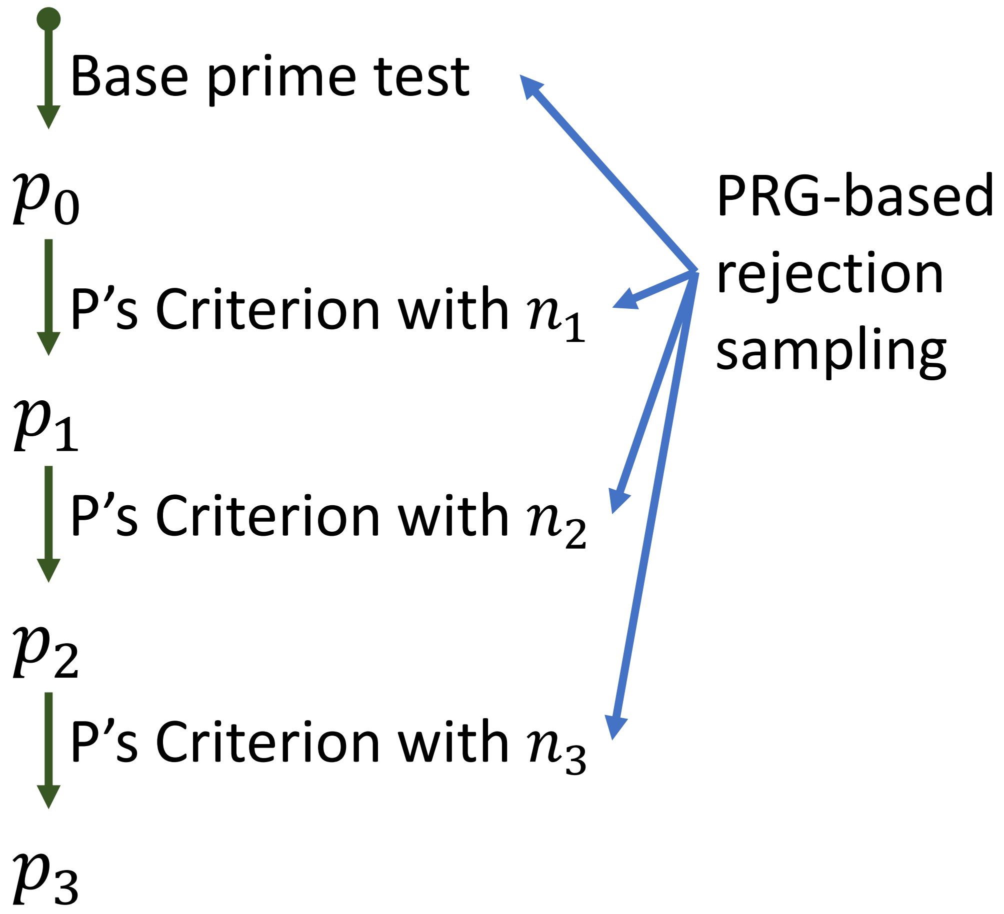
Traditional Hash-to-Prime

- 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 ← PRG(seed = x)
  while g.output() is composite:
        g.advance()
  Return g.output()
```

Pocklington Prime Generation

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



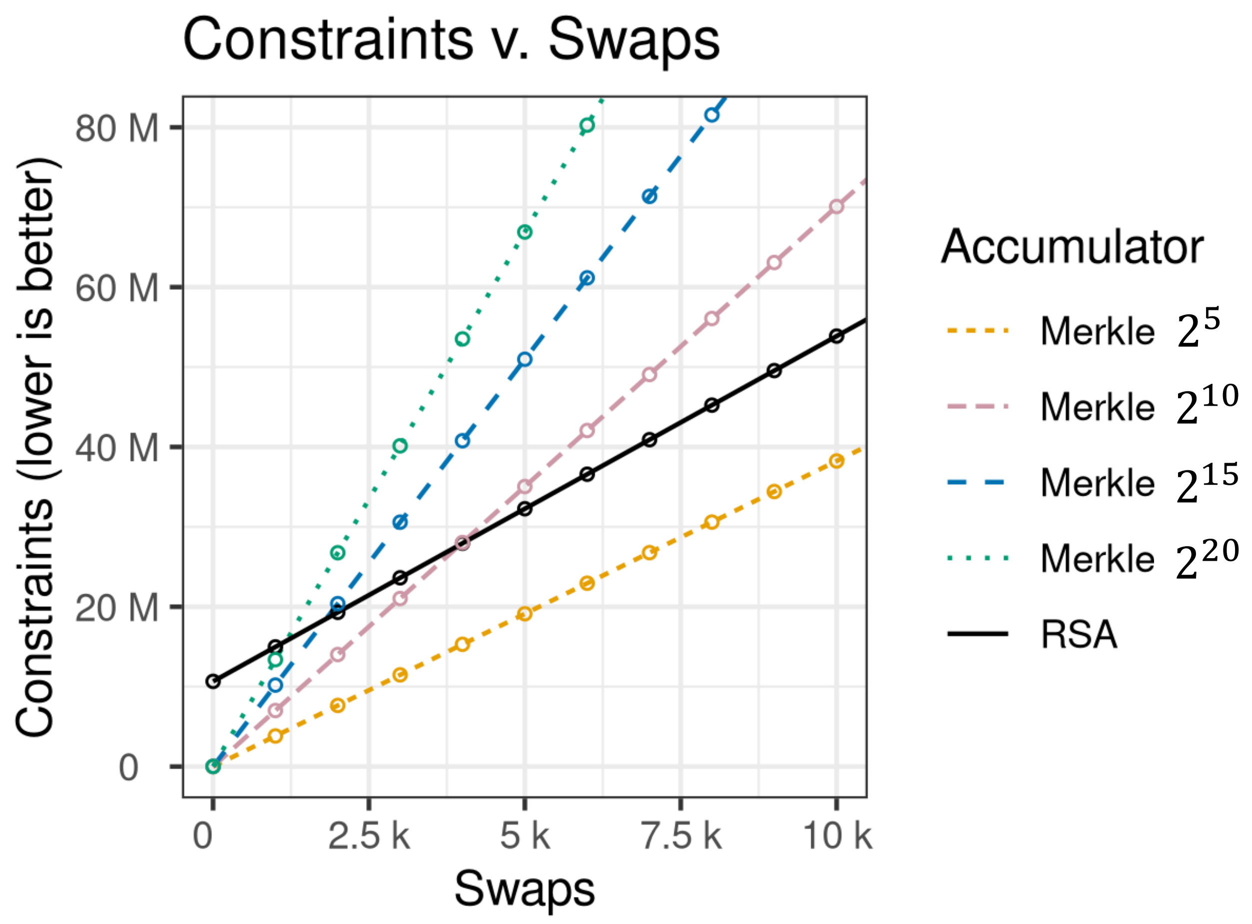
Many fewer constraints than Miller-Rabin, and provably prime

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

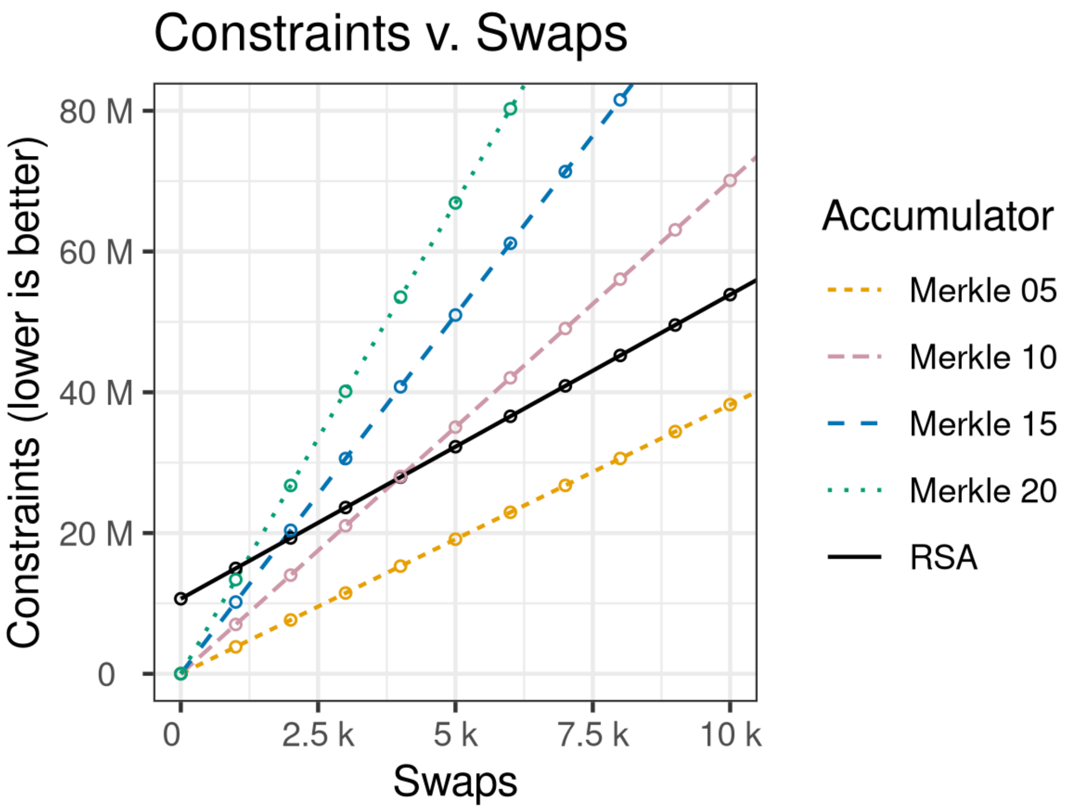
Research Question

Do RSA accumulators use fewer constraints than Merkle Trees?

Techniques

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

Conclusions



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