Scaling Verifiable Computation Using Efficient Set Accumulators

USENIX Security, 2020

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

*Stanford  ^Unaffiliated
Problem: Verifiable Storage

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

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

<table>
<thead>
<tr>
<th>Prover(A, d)</th>
<th>Verifier(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v \leftarrow A[i])</td>
<td>(\text{Verify}_{\text{read}}(d, i, v, \pi_r))</td>
</tr>
<tr>
<td>(A[i_w] \leftarrow v_w)</td>
<td>(\text{Verify}_{\text{update}}(d, i_w, v_w, d', \pi_w))</td>
</tr>
</tbody>
</table>

Context: Verifiable outsourcing/cryptographic proof systems

Our Work: Concretely cheaper verifiable storage using RSA accumulators
Cryptographic Proof Systems

Programming Them

RSA Accumulators
Cryptographic Proof Systems

Programming Them

RSA Accumulators
NP Proof Systems

L ∈ NP
(x ∈ L)?

Properties

• |w| ∈ poly(|x|)
• T_{V_L} ∈ poly(|x|)
• Aladdin learns w
Cryptographic Proof Systems: Abstract

\[ L \in NP \quad (x \in L) ? \]

Extra Properties

- \(|\pi| \in O(1)\)
- \(T_{\text{Verify}} \in O(|x|)\)
- (Aladdin doesn’t learn \(w\))
- \(T_{\text{Prove}} \in \text{poly}(T_{V_L})\)

Using PCPs + Cryptography

\[ \pi \leftarrow \text{Prove}_{V_L}(x, w) \]

\[ w \leftarrow ? \quad \text{s.t.} \quad V_L(x, w) = T \]
Cryptographic Proof Systems: Concrete

$L$ must be verifiable by an arithmetic constraint system (arithmetic circuit)

\[ V_L(x, w) \]
Rank-1 Constraint Systems (R1CS)

• Constraints have the form
  \[ A \times B = C \]
  where \( A, B, C \) are linear combinations of variables

• Prover time proportional to constraint count.

\[
\begin{align*}
x_0(1 - x_0) &= 0 \\
0 &= w_0 + 2w_1 + 4w_2 - x \\
x_0x_1 &= w \\
x_0x_1x_2 &= w \quad \times
\end{align*}
\]
Cryptographic Proof Systems

Programming Them

RSA Accumulators
What Does Programming in R1CS Mean?

Abstract Constraint

Variables encoded as field variables
Predicates encoded as constraints

*Constraints may use witness variables*

```
z < 16
```

“Programming”

Rank-1 Constraints

```
A_1 \times B_1 = C_1
A_2 \times B_2 = C_2
A_3 \times B_3 = C_3
\vdots
A_n \times B_n = C_n
```
Inequality in R1CS

Abstract Constraint

\[ z < 16 \]

Encoded as the field variable \( z \)

Rank-1 Constraints

\[
\begin{align*}
    w_0 \times (1 - w_0) &= 0 \\
    w_1 \times (1 - w_1) &= 0 \\
    w_2 \times (1 - w_2) &= 0 \\
    w_3 \times (1 - w_3) &= 0 \\
    0 &= w_0 + 2w_1 + 4w_2 + 8w_3 - z
\end{align*}
\]
Polynomial Multiplication

Abstract Constraint

\[ f(x) \cdot g(x) = h(x) \]

Each coefficient is a field variable:

- \( f(x) = f_0 + f_1 x + f_2 x^2 \)
- \( g(x) = g_0 + g_1 x + g_2 x^2 \)
- \( h(x) = h_0 + h_1 x + h_2 x^2 + h_3 x^3 + h_4 x^4 \)

Rank-1 Constraints

\[
\begin{align*}
(f_0 + f_1 + f_2)(g_0 + g_1 + g_2) &= h_0 + h_1 + h_2 + h_3 + h_4 \\
(f_0 + 2f_1 + 4f_2)(g_0 + 2g_1 + 4g_2) &= h_0 + 2h_1 + 4h_2 + 8h_3 + 16h_4 \\
(f_0 + 3f_1 + 9f_2)(g_0 + 3g_1 + 9g_2) &= h_0 + 3h_1 + 9h_2 + 27h_3 + 81h_4 \\
(f_0 + 4f_1 + 16f_2)(g_0 + 4g_1 + 16g_2) &= h_0 + 4h_1 + 16h_2 + 64h_3 + 256h_4 \\
(f_0 + 5f_1 + 25f_2)(g_0 + 5g_1 + 25g_2) &= h_0 + 5h_1 + 25h_2 + 125h_3 + 625h_4
\end{align*}
\]

Check \( f(a) \cdot g(a) = h(a) \) for different \( a \)
Big Natural Multiplication

Abstract Constraint

\[ x \cdot y = z \]

Represent naturals with limbs, base \( b \). Each limb is a field element.
- \( x = x_0 + x_1b + x_2b^2 \)
- \( y = y_0 + y_1b + y_2b^2 \)
- \( z = z_0 + z_1b + z_2b^2 + z_3b^3 + z_4b^4 + z_5b^5 \)

Rank-1 Constraints Sketch

\[ \text{carry} \left( \text{nat}(\text{poly}(x) \times \text{poly}(y)) \right) = z \]

~ a ripple-carry adder from digital architecture (range checks!)
Abstract Constraint

\[ \lfloor y/x \rfloor = q \]

Represent naturals with limbs, base \( b \). Each limb is a field element.

- \( x = x_0 + x_1 b + x_2 b^2 \)
- \( y = y_0 + y_1 b + y_2 b^2 \)
- \( q = q_0 + q_1 b + q_2 b^2 \)

Rank-1 Constraints Sketch

\[ \exists r. \quad y = xq + r \]
Cryptographic Proof Systems

Programming Them

RSA Accumulators
The Competition: Merkle Trees

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

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

- Based on RSA groups
  - The integers modulo \( pq \): the product of two unknown primes.
  - Hard to compute roots.
    - \( x^n \) is easy, \( n\sqrt{x} \) is hard.
- The digest of an RSA Accumulator is

\[
d = g^\prod_i H_\Delta(y_i)
\]

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!

\[ d' = d^{H_\Delta(y)} \]
Batched RSA Accumulator Proofs

- Batches require two small exponentiations \([BBF 18]/[Wes 18]\)
- Requires a hash function to prime numbers (for non-interactivity)

\[d' = d \prod_i H_\Delta(y_i)\]

\[Q \leftarrow d \left[\prod_i H_\Delta(y_i)/\ell\right]\]

\[\ell \leftrightarrow \text{Primes}\]

\[Q\]

\[d' = Q^\ell \cdot d \prod_i H_\Delta(y_i)^\%\ell\]

Verification cost: \(k \text{ (hashes & modular } \times) + 2 \text{ exponentiations}\)

for \(k\) updates and a storage of capacity \(m\).
RSA Accumulator Circuit Overview

\[ \ell \leftarrow H_p(\ldots) \]

\[ d' = Q^\ell \cdot d^{\prod_i H_\Delta(y_i) \% \ell} \]
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 \textbf{HashToPrime}(x):
\begin{align*}
g & \leftarrow \text{PRG}(seed = x) \\
\text{while } g\text{.output()} \text{ is composite:} & \\
& \quad g\text{.advance()} \\
\text{Return } g\text{.output()}
\end{align*}
Pocklington Prime Generation

• Pocklington’s criterion:
  • If
    • $p$ is prime
    • $n < p$
    • $\exists a. \ a^{np} \equiv np + 1 \ \text{gcd}(a^n - 1, np + 1) = 1$
  • Then $np + 1$ is prime

• Basis for a recursive primality certificate
  • Idea: Rejection sampling of prime certificates

Base prime test
$p_0$
$P$’s Criterion with $n_1$
$p_1$
$P$’s Criterion with $n_2$
$p_2$
$P$’s Criterion with $n_3$
$p_3$

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

• Optimizations for 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: Constraints

- 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
Evaluation: Prover Time

- Includes RSA accumulator removal time (≈43s)
  - Computing $d'$ such that $d = d'\prod_i^{H_\Delta(y_i)}$
  - Independent of batch size, linear in storage size.

- Machine info:
  - 48 logical cores
  - 132GB memory
Future Directions

• Better investigation of concrete prover costs

• Integration with the proof system
  • Direct support for range-proofs ($z < 2^{32}$)
  • Arithmetic circuits over $\mathbb{Z}/pq\mathbb{Z}$ (crazy?)

• Managing non-proof prover costs
  • Multi-tiered accumulators?
  • Hybrid RSA-Merkle accumulators?
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

Paper: [ia.cr/2019/1494](https://ia.cr/2019/1494)
Implementation: [github.com/alex-ozdemir/bellman-bignat](https://github.com/alex-ozdemir/bellman-bignat)