Moz$\mathbb{Z}_{2^k}$arella: Efficient Vector-OLE and Zero-Knowledge Proofs Over $\mathbb{Z}_{2^k}$

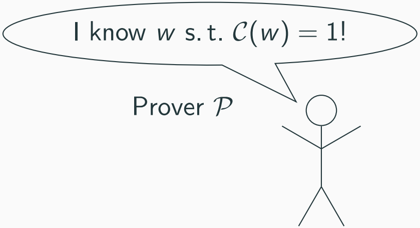
Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl

August 18, 2022 – Crypto'22

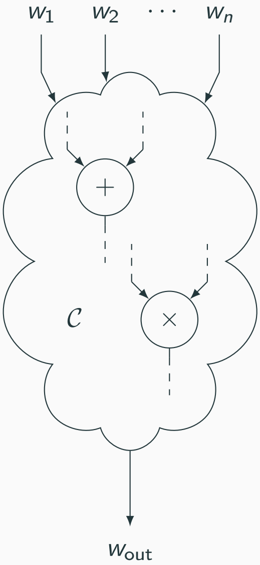
Aarhus University

AARHUS UNIVERSITY

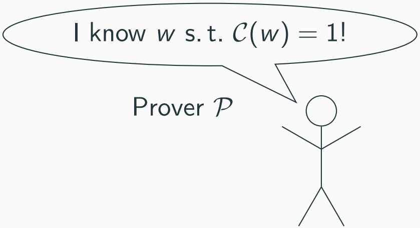
Zero-Knowledge Proofs for Arithmetic Circuits



$\mathcal{C}$ is an arithmetic circuit over the ring $\mathbb{Z}_{2^k}$



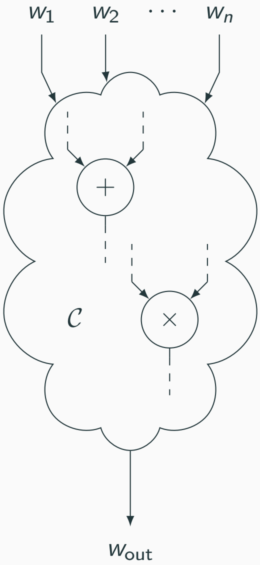
Zero-Knowledge Proofs for Arithmetic Circuits



$\mathcal{C}$ is an arithmetic circuit over the ring $\mathbb{Z}_{2^k}$

Security Properties:

- Soundness
- Zero-Knowledge
- Completeness



Computation over $\mathbb{Z}_{2^k}$ vs. Finite Fields $\mathbb{F}_{p^r}$

- $\mathbb{Z}_{2^k} = \mathbb{Z}/2^k\mathbb{Z} = (\{0, \ldots, 2^k - 1\}, +, \cdot)$ – the ring of integers modulo 2^k

Computation over $\mathbb{Z}_{2^k}$ vs. Finite Fields $\mathbb{F}_{p^r}$

- $\mathbb{Z}_{2^k} = \mathbb{Z}/2^k\mathbb{Z} = (\{0, \ldots, 2^k - 1\}, +, \cdot)$ – the ring of integers modulo 2^k

Advantages

- maps naturally to data types used by CPUs and programming languages
- e.g., `uint32_t`, `uint64_t` in C

$\rightsquigarrow$ easier to convert programs to corresponding circuits

$\rightsquigarrow$ more efficient protocol implementations

Computation over $\mathbb{Z}_{2^k}$ vs. Finite Fields $\mathbb{F}_{p^r}$

- $\mathbb{Z}_{2^k} = \mathbb{Z}/2^k\mathbb{Z} = (\{0, \ldots, 2^k - 1\}, +, \cdot)$ – the ring of integers modulo 2^k

Advantages

- maps naturally to data types used by CPUs and programming languages
- e.g., `uint32_t`, `uint64_t` in C

$\rightsquigarrow$ easier to convert programs to corresponding circuits

$\rightsquigarrow$ more efficient protocol implementations

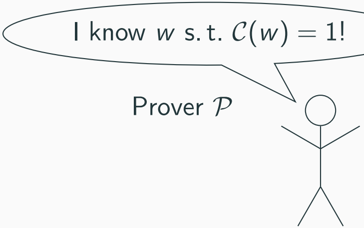
Disadvantages

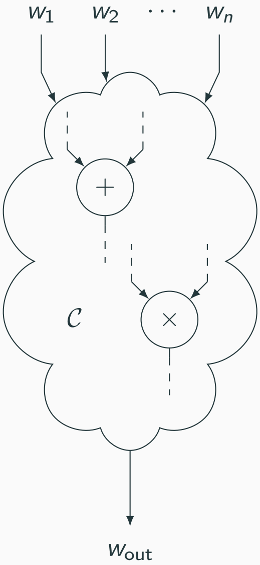
- $\mathbb{Z}_{2^k}$ is not a field
- zero-divisors
- no division by multiples of 2
- polynomials can have lots of roots

$\rightsquigarrow$ common tricks don't work and protocols get more complicated

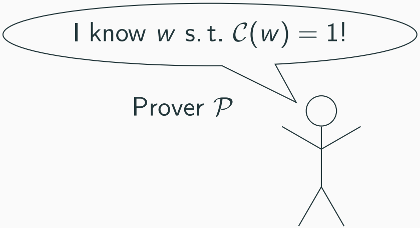
$\rightsquigarrow$ proofs of security are harder

Zero-Knowledge Proofs for Arithmetic Circuits



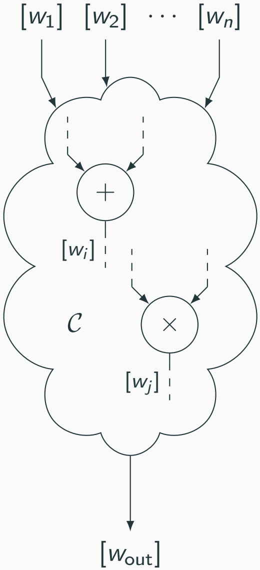


Zero-Knowledge Proofs for Arithmetic Circuits via Commit & Prove

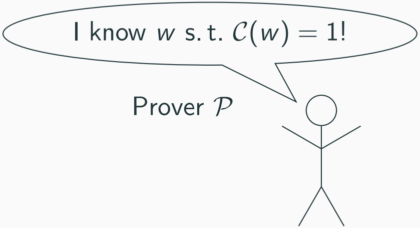


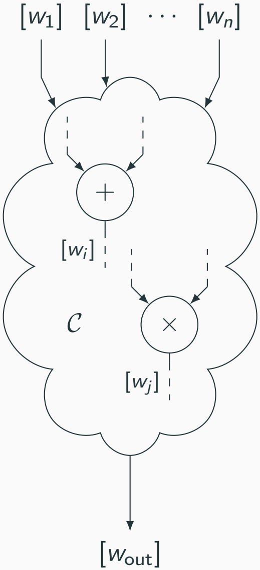
Ingredients:

1. linearly homomorphic commitments $[\,\cdot\,]$
   - can compute $[z] \leftarrow a \cdot [x] + [y] + b$



Zero-Knowledge Proofs for Arithmetic Circuits via Commit & Prove

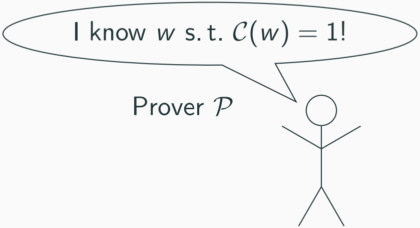




Ingredients:

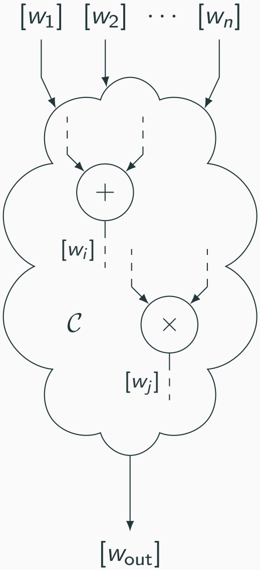
1. linearly homomorphic commitments $[\cdot]$
   - can compute $[z] \leftarrow a \cdot [x] + [y] + b$
2. multiplication check
   - given $([a], [b], [c])$, verify $a \cdot b \stackrel{?}{=} c$

Zero-Knowledge Proofs for Arithmetic Circuits via Commit & Prove



Setting:

- designated verifier
- linear communication
- linear time prover and verifier
- minimal overhead compared to circuit evaluation
  - computation and memory



Linearly Homomorphic Commitments from Vector OLE

For large fields: authenticate $x \in \mathbb{F}$ with information-theoretic MAC:

$$M[x] = \Delta \cdot x + K[x] \in \mathbb{F}$$

Prover holds value x and tag $M[x]$

Verifier holds global key $\Delta \in_R \mathbb{F}$ and key $K[x] \in_R \mathbb{F}$

(cf. Mac'n'Cheese [BMRS21], Wolverine [WYKW21])

Linearly Homomorphic Commitments from Vector OLE

For ring $\mathbb{Z}_{2^k}$: authenticate $x \in \mathbb{Z}_{2^k}$ over the larger ring $\mathbb{Z}_{2^{k+s}}$

$$M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^{k+s}} \qquad \text{with } x = \tilde{x} \pmod{2^k}$$

Prover holds value $\tilde{x}$ and tag $M[x]$ Verifier holds global key $\Delta \in_R \mathbb{Z}_{2^s}$ and key $K[x] \in_R \mathbb{Z}_{2^{k+s}}$

(cf. SPD$\mathbb{Z}_{2^k}$ [CDESX18], Appenzeller2Brie [BBMRS21])

Linearly Homomorphic Commitments from Vector OLE

For ring $\mathbb{Z}_{2^k}$: authenticate $x \in \mathbb{Z}_{2^k}$ over the larger ring $\mathbb{Z}_{2^{k+s}}$

$$M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^{k+s}} \qquad \text{with } x = \tilde{x} \pmod{2^k}$$

Prover holds value $\tilde{x}$ and tag $M[x]$ Verifier holds global key $\Delta \in_R \mathbb{Z}_{2^s}$ and key $K[x] \in_R \mathbb{Z}_{2^{k+s}}$

(cf. SPD$\mathbb{Z}_{2^k}$ [CDESX18], Appenzeller2Brie [BBMRS21])

Vector Oblivious Linear Evaluation:

such that $\vec{M} = \Delta \cdot \vec{x} + \vec{K} \pmod{2^\ell}$

VOLE for $\mathbb{Z}_{2^k}$

How to instantiate VOLE?

From Oblivious Transfer or Homomorphic Encryption

$\rightsquigarrow$ communication at least linear in output size

How to instantiate VOLE?

From Oblivious Transfer or Homomorphic Encryption

$\rightsquigarrow$ communication at least linear in output size

Via Pseudorandom Correlation Generators (PCGs)

- interactive generation of a short seed $\rightarrow$ non-interactive expansion to long correlated string
- communication <u>sublinear</u> in vector length n
- based on variants of Learning Parity with Noise (LPN)
- active security only for fields [WYKW21; Boy+19]

How to instantiate VOLE?

From Oblivious Transfer or Homomorphic Encryption

$\rightsquigarrow$ communication at least linear in output size

Via Pseudorandom Correlation Generators (PCGs)

- interactive generation of a short seed $\rightarrow$ non-interactive expansion to long correlated string
- communication sublinear in vector length n
- based on variants of Learning Parity with Noise (LPN)
- active security only for fields [WYKW21; Boy+19]

Here: actively secure VOLE for rings $\mathbb{Z}_{2^k}$ with sublinear communication

Learning Parity with Noise (LPN) Assumption

$$\begin{bmatrix} \\ \vec{s} \\ \\ \end{bmatrix}$$

$\vec{s}$: short, uniform seed

Learning Parity with Noise (LPN) Assumption

$$\begin{bmatrix} \\ \\ \quad A \quad \\ \\ \\ \end{bmatrix} \cdot \begin{bmatrix} \\ \vec{s} \\ \\ \end{bmatrix}$$

$\vec{s}$: short, uniform seed

A: generating matrix of a random linear code

Learning Parity with Noise (LPN) Assumption

$$\begin{bmatrix} \\ \\ & A & \\ \\ \\ \end{bmatrix} \cdot \begin{bmatrix} \\ \vec{s} \\ \\ \end{bmatrix} + \begin{bmatrix} \\ \\ \vec{e} \\ \\ \\ \end{bmatrix}$$

$\vec{s}$: short, uniform seed

A: generating matrix of a random linear code

$\vec{e}$: sparse error vector (regular error $\vec{e} = (\vec{e}_1 | \cdots | \vec{e}_t)^T$ with $\vec{e}_i$ one-hot)

Learning Parity with Noise (LPN) Assumption

$$\begin{bmatrix} \\ \\ A \\ \\ \\ \end{bmatrix} \cdot \begin{bmatrix} \\ \vec{s} \\ \\ \end{bmatrix} + \begin{bmatrix} \\ \\ \vec{e} \\ \\ \\ \end{bmatrix} \qquad\qquad \begin{bmatrix} \\ \\ \vec{u} \\ \\ \\ \end{bmatrix}$$

$\vec{s}$: short, uniform seed

$\vec{u}$: long, uniform vector

A: generating matrix of a random linear code

$\vec{e}$: sparse error vector (regular error $\vec{e} = (\vec{e}_1 | \cdots | \vec{e}_t)^T$ with $\vec{e}_i$ one-hot)

Learning Parity with Noise (LPN) Assumption

$$\begin{bmatrix} \\ \\ A \\ \\ \\ \end{bmatrix}, \begin{bmatrix} \\ \\ A \\ \\ \\ \end{bmatrix} \cdot \begin{bmatrix} \vec{s} \end{bmatrix} + \begin{bmatrix} \\ \\ \vec{e} \\ \\ \\ \end{bmatrix} \approx_C \begin{bmatrix} \\ \\ A \\ \\ \\ \end{bmatrix}, \begin{bmatrix} \\ \\ \vec{u} \\ \\ \\ \end{bmatrix}$$

$\vec{s}$: short, uniform seed

$\vec{u}$: long, uniform vector

A: generating matrix of a random linear code

$\vec{e}$: sparse error vector (regular error $\vec{e} = (\vec{e}_1 | \cdots | \vec{e}_t)^T$ with $\vec{e}_i$ one-hot)

VOLE Extension Protocol

Goal: expand m seed/base VOLEs into $n \gg m$ VOLEs: $\vec{z} = \Delta \cdot \vec{x} + \vec{y}$

VOLE Extension Protocol

Goal: expand m seed/base VOLEs into $n \gg m$ VOLEs: $\vec{z} = \Delta \cdot \vec{x} + \vec{y}$

1. start with $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ base VOLEs of length m

based on Wolverine [WYKW21]

VOLE Extension Protocol

Goal: expand m seed/base VOLEs into $n \gg m$ VOLEs: $\vec{z} = \Delta \cdot \vec{x} + \vec{y}$

1. start with $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ base VOLEs of length m
2. generate and concatenate t single-point VOLEs of length n/t

   $\rightsquigarrow$ obtain $\vec{c} = \Delta \cdot \vec{e} + \vec{b}$ where $\vec{e}$ has t non-zero entries

based on Wolverine [WYKW21]

VOLE Extension Protocol

Goal: expand m seed/base VOLEs into $n \gg m$ VOLEs: $\vec{z} = \Delta \cdot \vec{x} + \vec{y}$

1. start with $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ base VOLEs of length m
2. generate and concatenate t single-point VOLEs of length n/t

   $\rightsquigarrow$ obtain $\vec{c} = \Delta \cdot \vec{e} + \vec{b}$ where $\vec{e}$ has t non-zero entries
3. apply the generating matrix A to all seed VOLEs and add the error VOLEs

Sender computes $\begin{cases} \vec{x} := A \cdot \vec{u} + \vec{e} \\ \vec{z} := A \cdot \vec{w} + \vec{c} \end{cases} \implies \vec{z} := \Delta \cdot \vec{x} + \vec{y}$

Receiver computes $\vec{y} := A \cdot \vec{v} + \vec{b}$

based on Wolverine [WYKW21]

VOLE Extension Protocol

Goal: expand m seed/base VOLEs into $n \gg m$ VOLEs: $\vec{z} = \Delta \cdot \vec{x} + \vec{y}$

1. start with $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ base VOLEs of length m
2. generate and concatenate t single-point VOLEs of length n/t
   $\rightsquigarrow$ obtain $\vec{c} = \Delta \cdot \vec{e} + \vec{b}$ where $\vec{e}$ has t non-zero entries
3. apply the generating matrix A to all seed VOLEs and add the error VOLEs

$$\text{Sender computes} \quad \begin{cases} \vec{x} := A \cdot \vec{u} + \vec{e} \\ \vec{z} := A \cdot \vec{w} + \vec{c} \end{cases} \quad \Longrightarrow \quad \vec{z} := \Delta \cdot \vec{x} + \vec{y}$$

$$\text{Receiver computes} \quad \vec{y} := A \cdot \vec{v} + \vec{b}$$

4. LPN: $\vec{x}$ looks uniformly random

based on Wolverine [WYKW21]

VOLE Extension Protocol

Goal: expand m seed/base VOLEs into $n \gg m$ VOLEs: $\vec{z} = \Delta \cdot \vec{x} + \vec{y}$

1. start with $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ base VOLEs of length m
2. generate and concatenate t single-point VOLEs of length n/t

   $\rightsquigarrow$ obtain $\vec{c} = \Delta \cdot \vec{e} + \vec{b}$ where $\vec{e}$ has t non-zero entries
3. apply the generating matrix A to all seed VOLEs and add the error VOLEs

$$\text{Sender computes} \quad \begin{cases} \vec{x} := A \cdot \vec{u} + \vec{e} \\ \vec{z} := A \cdot \vec{w} + \vec{c} \end{cases} \qquad \Longrightarrow \qquad \vec{z} := \Delta \cdot \vec{x} + \vec{y}$$

$$\text{Receiver computes} \quad \vec{y} := A \cdot \vec{v} + \vec{b}$$

4. LPN: $\vec{x}$ looks uniformly random
5. (save m VOLEs and repeat)

based on Wolverine [WYKW21]

VOLE Extension Protocol

Goal: expand m seed/base VOLEs into $n \gg m$ VOLEs: $\vec{z} = \Delta \cdot \vec{x} + \vec{y}$

1. start with $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ base VOLEs of length m
2. generate and concatenate t single-point VOLEs of length n/t

   $\rightsquigarrow$ obtain $\vec{c} = \Delta \cdot \vec{e} + \vec{b}$ where $\vec{e}$ ha non-zero entries
3. apply the generating matrix A to all OLEs and add the error VOLEs

crucial step: need communication sublinear in n

Sender computes $\vec{z} := A \cdot \vec{w} + \vec{c}$ $\Longrightarrow$ $\vec{z} := \Delta \cdot \vec{x} + \vec{y}$

Receiver computes $\vec{y} := A \cdot \vec{v} + \vec{b}$

4. LPN: $\vec{x}$ looks uniformly random
5. (save m VOLEs and repeat)

based on Wolverine [WYKW21]

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

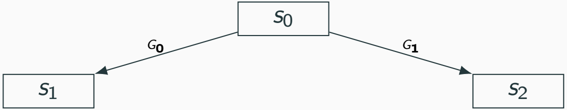
1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F

GGM Construction for a PRF

$\boxed{s_0}$

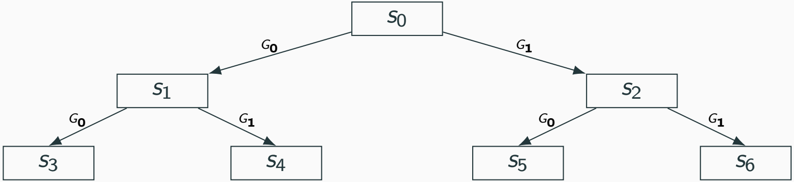
$G(x) = G(x)_0 \mid\mid G(x)_1$ length-doubling PRG

GGM Construction for a PRF



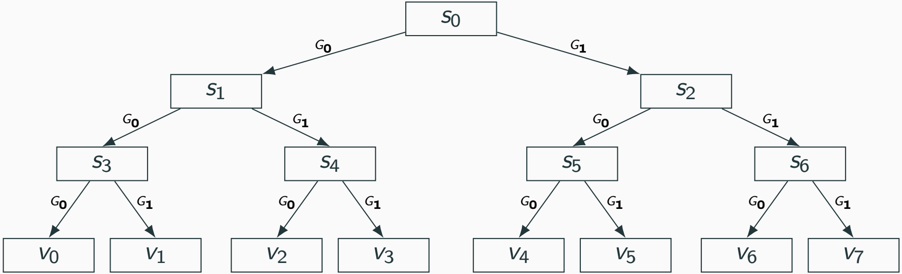
$G(x) = G(x)_0 \mid\mid G(x)_1$ length-doubling PRG

GGM Construction for a PRF



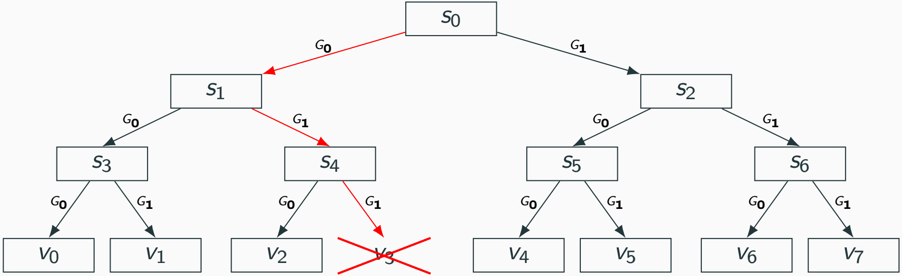
$G(x) = G(x)_0 \mid\mid G(x)_1$ length-doubling PRG

GGM Construction for a PRF



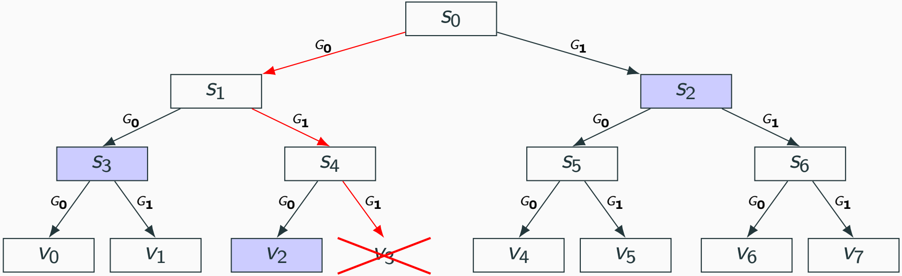
$G(x) = G(x)_0 \mid\mid G(x)_1$ length-doubling PRG

GGM Construction for a Punctured PRF



$G(x) = G(x)_0 \mid\mid G(x)_1$ length-doubling PRG

GGM Construction for a Punctured PRF



$G(x) = G(x)_0 \mid\mid G(x)_1$ length-doubling PRG

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F
   - $\mathcal{R}$ samples a PRF key k and
     use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $\mathcal{S}$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$

   $\rightsquigarrow$ $\mathcal{R}$ does not learn α and $\mathcal{S}$ does not learn v_α

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F
   - $\mathcal{R}$ samples a PRF key k and
     use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $\mathcal{S}$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$

   $\rightsquigarrow$ $\mathcal{R}$ does not learn α and $\mathcal{S}$ does not learn v_α
2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F
   - $\mathcal{R}$ samples a PRF key k and use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $\mathcal{S}$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$

   $\rightsquigarrow$ $\mathcal{R}$ does not learn α and $\mathcal{S}$ does not learn v_α
2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$

a malicious receiver $\mathcal{R}$ can
- use inconsistent values in the OTs
- send an incorrect d

$\rightsquigarrow$ leakage on the noise coordinate α

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F
   - $\mathcal{R}$ samples a PRF key k and use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $\mathcal{S}$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$

   $\rightsquigarrow$ $\mathcal{R}$ does not learn α and $\mathcal{S}$ does not learn v_α
2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$
3. ensure consistency

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F
   - $\mathcal{R}$ samples a PRF key k and use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $\mathcal{S}$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$

   $\rightsquigarrow$ $\mathcal{R}$ does not learn α and $\mathcal{S}$ does not learn v_α
2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$
3. ensure consistency
   - Wolverine's [WYKW21] approach: use a random linear combination over the field $\mathbb{F}_q$

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{v}$ or $\vec{w}$ with ... PRF F
   - $\mathcal{R}$ sa...
2. com...
   - use one bas...
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$
3. ensure consistency
   - Wolverine's [WYKW21] approach: use a random linear combination over the field $\mathbb{F}_q$

random linear combinations are tricky over $\mathbb{Z}_{2^k}$

– need to enlarge ring by s bit to ensure consistency of lower bits! cf. SPD$\mathbb{Z}_{2^k}$ [CDESX18]

$\rightsquigarrow$ cannot do multiple iterations

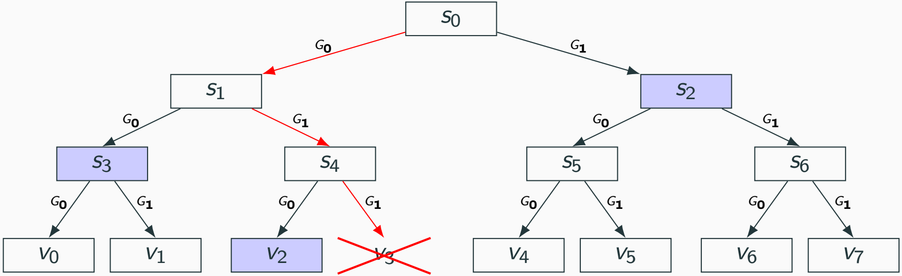
Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F
   - $\mathcal{R}$ samples a PRF key k and use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $\mathcal{S}$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$

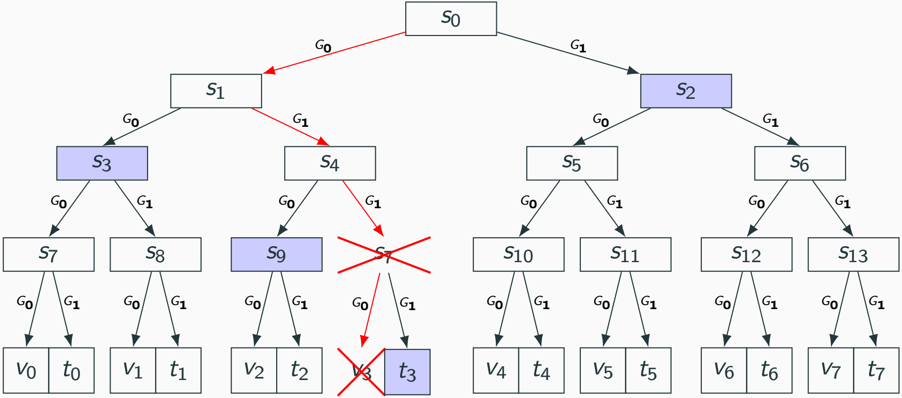
   $\rightsquigarrow$ $\mathcal{R}$ does not learn α and $\mathcal{S}$ does not learn v_α
2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$
3. ensure consistency
   - use universal hashing to verify consistency of the GGM tree (based on check in [Boy+19])

GGM Construction for a Punctured PRF



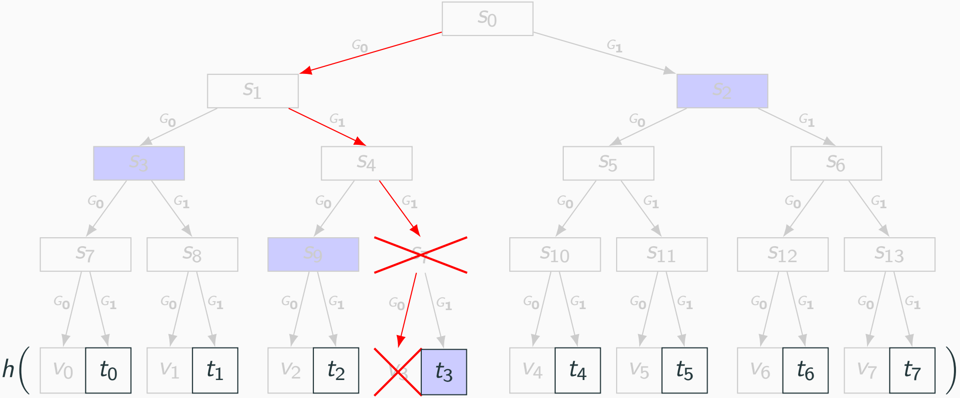
$G(x) = G(x)_0 \mid\mid G(x)_1$ length-doubling PRG

GGM Construction for a Punctured PRF with Consistency Check



$G(x) = G(x)_0 \parallel G(x)_1$ length-doubling PRG

GGM Construction for a Punctured PRF with Consistency Check



$G(x) = G(x)_0 \,||\, G(x)_1$ length-doubling PRG

$\rightsquigarrow$ compute and verify universal hash $h(t_0, \ldots, t_7)$

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F
   - $\mathcal{R}$ samples a PRF key k and use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $\mathcal{S}$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$

   $\rightsquigarrow$ $\mathcal{R}$ does not learn α and $\mathcal{S}$ does not learn v_α
2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$
3. ensure consistency
   - use universal hashing to verify consistency of the GGM tree (based on check in [Boy+19])

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in_R \mathbb{Z}_{2^\ell} \setminus \{0\}$ and $u_j = 0$ for $j \neq \alpha$

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF F
   - $\mathcal{R}$ samples a PRF key k and use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $\mathcal{S}$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$

   $\rightsquigarrow$ $\mathcal{R}$ does not learn α and $\mathcal{S}$ does not learn v_α
2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$
3. ensure consistency
   - use universal hashing to verify consistency of the GGM tree (based on check in [Boy+19])
   - use subset sum check for d with binary coefficients $\chi_1, \ldots, \chi_n \in_R \{0, 1\}$

Single-Point VOLE Protocol

Goal: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ wh

1. distribute l
   - $\mathcal{R}$
     us
   - set w_i

   $\rightsquigarrow$ $\mathcal{R}$ does not lear
2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $\mathcal{S}$ computes $w_\alpha := \delta - d - \sum_{j \neq \alpha} w_j$
3. ensure consistency
   - use universal hashing to verify consistency of the GGM tree (based on check in [Boy+19])
   - use subset sum check for d with binary coefficients $\chi_1, \ldots, \chi_n \in_R \{0, 1\}$

– malicious $\mathcal{R}$ can guess χ_α with probability $1/2$

$\rightsquigarrow$ adjust functionality to allow leakage of α with probability $1/2$

$\rightsquigarrow$ increase noise rate of the error to compensate

QuarkSilver – More Efficient Multiplication Check for $\mathbb{Z}_{2^k}$

Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

Goal: Given $([a], [b], [c])$, verify that $\tilde{a} \cdot \tilde{b} = \tilde{c} \pmod{2^k}$

Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

Goal: Given $([a], [b], [c])$, verify that $\tilde{a} \cdot \tilde{b} = \tilde{c} \pmod{2^k}$

Observation from QuickSilver [YSWW21]: Convert the three MAC equations $M[x] = K[x] + \tilde{x} \cdot \Delta$ for $x \in \{a, b, c\}$ into a polynomial in Δ:

$$\underbrace{\Delta \cdot K[c] - K[a] \cdot K[b]}_{\text{known by } \mathcal{V}} = \underbrace{(M[a] \cdot M[b])}_{\text{known by } \mathcal{P}} + \underbrace{(M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a])}_{\text{known by } \mathcal{P}} \cdot \Delta + \underbrace{(\tilde{c} - \tilde{a} \cdot \tilde{b})}_{= 0 \text{ if } \mathcal{P} \text{ honest}} \cdot \Delta^2$$

Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

Goal: Given $([a], [b], [c])$, verify that $\tilde{a} \cdot \tilde{b} = \tilde{c} \pmod{2^k}$

Observation from QuickSilver [YSWW21]: Convert the three MAC equations $M[x] = K[x] + \tilde{x} \cdot \Delta$ for $x \in \{a, b, c\}$ into a polynomial in Δ:

$$\underbrace{\Delta \cdot K[c] - K[a] \cdot K[b]}_{\text{known by } \mathcal{V}} = \underbrace{(M[a] \cdot M[b])}_{\text{known by } \mathcal{P}} + \underbrace{(M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a])}_{\text{known by } \mathcal{P}} \cdot \Delta + \underbrace{(\tilde{c} - \tilde{a} \cdot \tilde{b})}_{= 0 \text{ if } \mathcal{P} \text{ honest}} \cdot \Delta^2$$

use a random linear combination to verify many multiplications

Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

Goal: Given $([a], [b], [c])$, verify that $\tilde{a} \cdot \tilde{b} = \tilde{c} \pmod{2^k}$

Observation from QuickSilver [YSWW21]: Convert the three MAC equations $M[x] = K[x] + \tilde{x} \cdot \Delta$ for $x \in \{a, b, c\}$ into a polynomial in Δ:

$$\underbrace{\Delta \cdot K[c] - K[a] \cdot K[b]}_{\text{known by } \mathcal{V}} = \underbrace{(M[a] \cdot M[b])}_{\text{known by } \mathcal{P}} + \underbrace{(M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a])}_{\text{known by } \mathcal{P}} \cdot \Delta + \underbrace{(\tilde{c} - \tilde{a} \cdot \tilde{b})}_{= 0 \text{ if } \mathcal{P} \text{ honest}} \cdot \Delta^2$$

Soundness: cheating $\mathcal{P}$ needs to come up with $p(X) = e_0 + e_1 \cdot X + e \cdot X^2$ such that

Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

Goal: Given $([a], [b], [c])$, verify that $\tilde{a} \cdot \tilde{b} = \tilde{c} \pmod{2^k}$

Observation from QuickSilver [YSWW21]: Convert the three MAC equations $M[x] = K[x] + \tilde{x} \cdot \Delta$ for $x \in \{a, b, c\}$ into a polynomial in Δ:

$$\underbrace{\Delta \cdot K[c] - K[a] \cdot K[b]}_{\text{known by } \mathcal{V}} = \underbrace{(M[a] \cdot M[b])}_{\text{known by } \mathcal{P}} + \underbrace{(M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a])}_{\text{known by } \mathcal{P}} \cdot \Delta + \underbrace{(\tilde{c} - \tilde{a} \cdot \tilde{b})}_{= 0 \text{ if } \mathcal{P} \text{ honest}} \cdot \Delta^2$$

Soundness: cheating $\mathcal{P}$ needs to come up with $p(X) = e_0 + e_1 \cdot X + e \cdot X^2$ such that

- $p(\Delta) = 0$, and

Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

Goal: Given $([a], [b], [c])$, verify that $\tilde{a} \cdot \tilde{b} = \tilde{c} \pmod{2^k}$

Observation from QuickSilver [YSWW21]: Convert the three MAC equations $M[x] = K[x] + \tilde{x} \cdot \Delta$ for $x \in \{a, b, c\}$ into a polynomial in Δ:

$$\underbrace{\Delta \cdot K[c] - K[a] \cdot K[b]}_{\text{known by } \mathcal{V}} = \underbrace{(M[a] \cdot M[b])}_{\text{known by } \mathcal{P}} + \underbrace{(M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a])}_{\text{known by } \mathcal{P}} \cdot \Delta + \underbrace{(\tilde{c} - \tilde{a} \cdot \tilde{b})}_{= 0 \text{ if } \mathcal{P} \text{ honest}} \cdot \Delta^2$$

Soundness: cheating $\mathcal{P}$ needs to come up with $p(X) = e_0 + e_1 \cdot X + e \cdot X^2$ such that

- $p(\Delta) = 0$, and
- $e := \tilde{c} - \tilde{a} \cdot \tilde{b} \neq 0 \pmod{2^k} \implies p$ has degree 2

Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

Goal: Given $([a],[b],[c])$, verify that $\tilde{a} \cdot \tilde{b} = \tilde{c} \pmod{2^k}$

Observation from QuickSilver [YSWW21]: Convert the three MAC equations $M[x] = K[x] + \tilde{x} \cdot \Delta$ for $x \in \{a, b, c\}$ into a polynomial in Δ:

$$\underbrace{\Delta \cdot K[c] - K[a] \cdot K[b]}_{\text{known by } \mathcal{V}} = \underbrace{(M[a] \cdot M[b])}_{\text{known by } \mathcal{P}} + \underbrace{(M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a])}_{\text{known by } \mathcal{P}} \cdot \Delta + \underbrace{(\tilde{c} - \tilde{a} \cdot \tilde{b})}_{= 0 \text{ if } \mathcal{P} \text{ honest}} \cdot \Delta^2$$

Soundness: cheating $\mathcal{P}$ needs to come up with $p(X) = e_0 + e_1 \cdot X + e \cdot X^2$ such that

- $p(\Delta) = 0$, and
- $e := \tilde{c} - \tilde{a} \cdot \tilde{b} \neq 0 \pmod{2^k} \implies p$ has degree 2

- **Field** $\mathbb{F}_q$: polynomial p has at most two roots $\rightsquigarrow$ soundness error of $^2/_q$

Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

Goal: Given $([a],[b],[c])$, verify that $\tilde{a} \cdot \tilde{b} = \tilde{c} \pmod{2^k}$

Observation from QuickSilver [YSWW21]: Convert the three MAC equations $M[x] = K[x] + \tilde{x} \cdot \Delta$ for $x \in \{a, b, c\}$ into a polynomial in Δ:

$$\underbrace{\Delta \cdot K[c] - K[a] \cdot K[b]}_{\text{known by } \mathcal{V}} = \underbrace{(M[a] \cdot M[b])}_{\text{known by } \mathcal{P}} + \underbrace{(M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a])}_{\text{known by } \mathcal{P}} \cdot \Delta + \underbrace{(\tilde{c} - \tilde{a} \cdot \tilde{b})}_{= 0 \text{ if } \mathcal{P} \text{ honest}} \cdot \Delta^2$$

Soundness: cheating $\mathcal{P}$ needs to come up with $p(X) = e_0 + e_1 \cdot X + e \cdot X^2$ such that

- $p(\Delta) = 0$, and
- $e := \tilde{c} - \tilde{a} \cdot \tilde{b} \neq 0 \pmod{2^k} \implies p$ has degree 2

- **Field** $\mathbb{F}_q$: polynomial p has at most two roots $\rightsquigarrow$ soundness error of $2/q$
- **Ring** $\mathbb{Z}_{2^{k+s}}$: polynomial p has at most $2^{s/2}$ roots in the range $\{0, \ldots, 2^s - 1\}$

Performance & Summary

Performance

- Rust implementation of benchmarks: `https://github.com/AarhusCrypto/Mozzarella`
- For 64 bit arithmetic, use $\ell = 2^{162}$ (192 bit in implementation)

Performance

- Rust implementation of benchmarks: `https://github.com/AarhusCrypto/Mozzarella`
- For 64 bit arithmetic, use $\ell = 2^{162}$ (192 bit in implementation)

For $\sigma = 40$ statistical security large batches in LAN, we achieve:

VOLE

- $\approx$ 1 bit per VOLE
- $\approx$ 21 million VOLEs per second
- similar to Wolverine [WYKW21]

Performance

- Rust implementation of benchmarks: `https://github.com/AarhusCrypto/Mozzarella`
- For 64 bit arithmetic, use $\ell = 2^{162}$ (192 bit in implementation)

For $\sigma = 40$ statistical security large batches in LAN, we achieve:

VOLE

- $\approx$ 1 bit per VOLE
- $\approx$ 21 million VOLEs per second
- similar to Wolverine [WYKW21]

QuarkSilver Zero-Knowledge

- $\approx$ one ring element per multiplication
- $\approx$ 1.3 million multiplications per second
- QuickSilver [YSWW21]: $\approx 5\times$ more for 64 bit field
- QuarkSilver
  - larger rings
    $\rightsquigarrow$ more expensive arithmetic and communication
  - but provides native 64 bit arithmetic

Summary

Efficient VOLE for $\mathbb{Z}_{2^k}$

- practical actively secure protocol with sublinear communication
- 1 bit–1.3 bit communication per VOLE for large batches

by Popo le Chien (CC BY-SA 3.0), https://commons.wikimedia.org/wiki/File:Mozzarella_di_bufala2.jpg

Summary

Efficient VOLE for $\mathbb{Z}_{2^k}$

- practical actively secure protocol with sublinear communication
- 1 bit–1.3 bit communication per VOLE for large batches

QuarkSilver: More efficient VOLE-based zero-knowledge for $\mathbb{Z}_{2^k}$

- one ring element communication per multiplication

by Popo le Chien (CC BY-SA 3.0), https://commons.wikimedia.org/wiki/File:Mozzarella_di_bufala2.jpg

Summary

by Popo le Chien (CC BY-SA 3.0), https://commons.wikimedia.org/wiki/File:Mozzarella_di_bufala2.jpg

Efficient VOLE for $\mathbb{Z}_{2^k}$

- practical actively secure protocol with sublinear communication
- 1 bit–1.3 bit communication per VOLE for large batches

QuarkSilver: More efficient VOLE-based zero-knowledge for $\mathbb{Z}_{2^k}$

- one ring element communication per multiplication

Open problems

- The $\mathbb{Z}_{2^k}$ protocols need larger rings of size $2^\ell > 2^k$.
  $\Longrightarrow$ Can we reduce the communication overhead compared to k?
- Many recent protocols for fields use very efficient checks based on polynomials.
  $\Longrightarrow$ Can we get similar efficient alternatives over rings?

Summary

Efficient VOLE for $\mathbb{Z}_{2^k}$

- practical actively secure protocol with sublinear communication
- 1 bit–1.3 bit communication per VOLE for large batches

QuarkSilver: More efficient VOLE-based zero-knowledge for $\mathbb{Z}_{2^k}$

- one ring element communication per multiplication

Open problems

- The $\mathbb{Z}_{2^k}$ protocols need larger rings of size $2^\ell > 2^k$.
  $\implies$ Can we reduce the communication overhead compared to k?
- Many recent protocols for fields use very efficient checks based on polynomials.
  $\implies$ Can we get similar efficient alternatives over rings?

Full version of Mo$\mathbb{Z}_{2^k}$zarella on ePrint: `https://ia.cr/2022/819`

by Popo le Chien (CC BY-SA 3.0), https://commons.wikimedia.org/wiki/File:Mozzarella_di_bufala2.jpg

[BBMRS21] C. Baum et al. "Appenzeller to Brie: Efficient Zero-Knowledge Proofs for Mixed-Mode Arithmetic and Z2k". In: ACM CCS 2021. Nov. 2021.

[BMRS21] C. Baum et al. "Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions". In: CRYPTO 2021, Part IV. Aug. 2021.

[Boy+19] E. Boyle et al. "Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation". In: ACM CCS 2019. Nov. 2019.

[CDESX18] R. Cramer et al. "SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority". In: CRYPTO 2018, Part II. Aug. 2018.

[WYKW21] C. Weng et al. "Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits". In: 2021 IEEE Symposium on Security and Privacy. May 2021.

[YSWW21] K. Yang et al. "QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field". In: ACM CCS 2021. Nov. 2021.

Commitments

Over large fields: Authenticate $x \in \mathbb{F}$ with information-theoretic MAC:

$$M[x] = \Delta \cdot x + K[x] \in \mathbb{F}$$

Prover holds value x and tag $M[x]$

Verifier holds global key $\Delta \in_R \mathbb{F}$ and key $K[x] \in_R \mathbb{F}$

breaking binding $\implies$ guessing Δ

Commitments for $\mathbb{Z}_{2^k}$

Over large fields: Authenticate $x \in \mathbb{F}$ with information-theoretic MAC:

$$M[x] = \Delta \cdot x + K[x] \in \mathbb{F}$$

Prover holds value x and tag $M[x]$

Verifier holds global key $\Delta \in_R \mathbb{F}$ and key $K[x] \in_R \mathbb{F}$

$$\text{breaking binding} \implies \text{guessing } \Delta$$

Issue: $\mathbb{Z}_{2^k}$ is not a field and contains zero divisors

Commitments for $\mathbb{Z}_{2^k}$ – SPD$\mathbb{Z}_{2^k}$-style [BBMRS21]

Over large fields: Authenticate $x \in \mathbb{F}$ with information-theoretic MAC:

$$M[x] = \Delta \cdot x + K[x] \in \mathbb{F}$$

Prover holds value x and tag $M[x]$

Verifier holds global key $\Delta \in_R \mathbb{F}$ and key $K[x] \in_R \mathbb{F}$

breaking binding $\implies$ guessing Δ

Issue: $\mathbb{Z}_{2^k}$ is not a field and contains zero divisors

Idea: authenticate $x \in \mathbb{Z}_{2^k}$ over the larger ring $\mathbb{Z}_{2^\ell}$ with $\ell \geq k + s$ for security parameter s

Commitments for $\mathbb{Z}_{2^k}$ – SPD$\mathbb{Z}_{2^k}$-style [BBMRS21]

Over large fields: Authenticate $x \in \mathbb{F}$ with information-theoretic MAC:

$$M[x] = \Delta \cdot x + K[x] \in \mathbb{F}$$

Prover holds value x and tag $M[x]$

Verifier holds global key $\Delta \in_R \mathbb{F}$ and key $K[x] \in_R \mathbb{F}$

breaking binding $\implies$ guessing Δ

Issue: $\mathbb{Z}_{2^k}$ is not a field and contains zero divisors

Idea: authenticate $x \in \mathbb{Z}_{2^k}$ over the larger ring $\mathbb{Z}_{2^\ell}$ with $\ell \geq k + s$ for security parameter s

- represent x by $\tilde{x} \in \mathbb{Z}_{2^{k+s}}$ such that $x = \tilde{x} \pmod{2^k}$
- sample keys $\Delta \in_R \mathbb{Z}_{2^s}$, $K[x] \in_R \mathbb{Z}_{2^{k+s}}$
- compute tag $M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^{k+s}}$

Commitments for $\mathbb{Z}_{2^k}$ – SPD$\mathbb{Z}_{2^k}$-style [BBMRS21]

Over large fields: Authenticate $x \in \mathbb{F}$ with information-theoretic MAC:

$$M[x] = \Delta \cdot x + K[x] \in \mathbb{F}$$

Prover holds value x and tag $M[x]$

Verifier holds global key $\Delta \in_R \mathbb{F}$ and key $K[x] \in_R \mathbb{F}$

breaking binding $\implies$ guessing Δ

Issue: $\mathbb{Z}_{2^k}$ is not a field and contains zero divisors

Idea: authenticate $x \in \mathbb{Z}_{2^k}$ over the larger ring $\mathbb{Z}_{2^\ell}$ with $\ell \geq k + s$ for security parameter s

- represent x by $\tilde{x} \in \mathbb{Z}_{2^{k+s}}$ such that $x = \tilde{x} \pmod{2^k}$
- sample keys $\Delta \in_R \mathbb{Z}_{2^s}$, $K[x] \in_R \mathbb{Z}_{2^{k+s}}$
- compute tag $M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^k}$

- authenticates the lower k bits
- increases storage and communication costs

Linear Operations and Openings [BBMRS21]

Linear Operations: to compute $[z] \leftarrow \alpha \cdot [x] + [y] + c$, set

- $\tilde{z} \leftarrow \alpha \cdot \tilde{x} + \tilde{y} + c \pmod{2^\ell}$
- $M[z] \leftarrow \alpha \cdot M[x] + M[y] \pmod{2^\ell}$
- $K[z] \leftarrow \alpha \cdot K[x] + K[y] - \Delta \cdot c \pmod{2^\ell}$

Linear Operations and Openings

Linear Operations: to compute $[z] \leftarrow \alpha \cdot [x] + [y] + c$, set

- $\tilde{z} \leftarrow \alpha \cdot \tilde{x} + \tilde{y} + c \pmod{2^\ell}$
- $M[z] \leftarrow \alpha \cdot M[x] + M[y] \pmod{2^\ell}$
- $K[z] \leftarrow \alpha \cdot K[x] + K[y] - \Delta \cdot c \pmod{2^\ell}$

Open:

- $\mathcal{P}$ publishes $\tilde{x}, M[x]$
- $\mathcal{V}$ checks $M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^\ell}$

Linear Operations and Openings [BBMRS21]

Linear Operations: to compute $[z] \leftarrow \alpha \cdot [x] + [y] + c$, set

- $\tilde{z} \leftarrow \alpha \cdot \tilde{x} + \tilde{y} + c \pmod{2^\ell}$
- $M[z] \leftarrow \alpha \cdot M[x] + M[y] \pmod{2^\ell}$
- $K[z] \leftarrow \alpha \cdot K[x] + K[y] - \Delta \cdot c \pmod{2^\ell}$

Open:

- $\mathcal{P}$ publishes $\tilde{x}, M[x]$
- $\mathcal{V}$ checks $M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^\ell}$

upper bits of $\tilde{x}$ leak information about intermediate values

Linear Operations and Openings [BBMRS21]

Linear Operations: to compute $[z] \leftarrow \alpha \cdot [x] + [y] + c$, set

- $\tilde{z} \leftarrow \alpha \cdot \tilde{x} + \tilde{y} + c \pmod{2^\ell}$
- $M[z] \leftarrow \alpha \cdot M[x] + M[y] \pmod{2^\ell}$
- $K[z] \leftarrow \alpha \cdot K[x] + K[y] - \Delta \cdot c \pmod{2^\ell}$

Open:

- $\mathcal{P}$ publishes $\tilde{x}, M[x]$
- $\mathcal{V}$ checks $M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^\ell}$

upper bits of $\tilde{x}$ leak information about intermediate values

$\rightsquigarrow$ use random $[r]$ to mask upper bits before opening: $[z] \leftarrow [x] + 2^k \cdot [r]$

Linear Operations and Openings [BBMRS21]

Linear Operations: to compute $[z] \leftarrow \alpha \cdot [x] + [y] + c$, set

- $\tilde{z} \leftarrow \alpha \cdot \tilde{x} + \tilde{y} + c \pmod{2^\ell}$
- $M[z] \leftarrow \alpha \cdot M[x] + M[y] \pmod{2^\ell}$
- $K[z] \leftarrow \alpha \cdot K[x] + K[y] - \Delta \cdot c \pmod{2^\ell}$

Open:

- $\mathcal{P}$ publishes $\tilde{x}, M[x]$
- $\mathcal{V}$ checks $M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^\ell}$

upper bits of $\tilde{x}$ leak information about intermediate values

$\rightsquigarrow$ use random $[r]$ to mask upper bits before opening: $[z] \leftarrow [x] + 2^k \cdot [r]$

Batched Open: e.g., send $H(M[x_1], \ldots, M[x_n])$ instead of all $M[x_1], \ldots, M[x_n]$

Verifying Multiplications for $\mathbb{Z}_{2^k}$ [BBMRS21]

Task: Given $([a], [b], [c])$, verify that $a \cdot b = c \pmod{2^k}$.

Verifying Multiplications for $\mathbb{Z}_{2^k}$ [BBMRS21]

Task: Given $([a], [b], [c])$, verify that $a \cdot b = c \pmod{2^k}$.

Variant 1: Adaption of Wolverine's [WYKW21] check

- bucketing with untrusted multiplication triples $\rightsquigarrow$ Beaver multiplication

Verifying Multiplications for $\mathbb{Z}_{2^k}$ [BBMRS21]

Task: Given $([a], [b], [c])$, verify that $a \cdot b = c \pmod{2^k}$.

Variant 1: Adaption of Wolverine's [WYKW21] check

- bucketing with untrusted multiplication triples $\rightsquigarrow$ Beaver multiplication

Variant 2: Adaption of the Mac'n'Cheese [BMRS21] check

- needs large message space $\rightsquigarrow$ authenticate $\mathbb{Z}_{2^{k+s}}$ elements over $\mathbb{Z}_{2^{k+\mathbf{2}s}}$

Verifying Multiplications for $\mathbb{Z}_{2^k}$ [BBMRS21]

Task: Given $([a], [b], [c])$, verify that $a \cdot b = c \pmod{2^k}$.

Variant 1: Adaption of Wolverine's [WYKW21] check

- bucketing with untrusted multiplication triples $\rightsquigarrow$ Beaver multiplication

Variant 2: Adaption of the Mac'n'Cheese [BMRS21] check

- needs large message space $\rightsquigarrow$ authenticate $\mathbb{Z}_{2^{k+s}}$ elements over $\mathbb{Z}_{2^{k+2s}}$

Prover $\mathcal{P}$		Verifier $\mathcal{V}$
	$\xrightarrow{\text{Random}() \to [x]}$	
	$\xrightarrow{\text{Input}(x \cdot b) \to [z]}$	

Verifying Multiplications for $\mathbb{Z}_{2^k}$ [BBMRS21]

Task: Given $([a],[b],[c])$, verify that $a \cdot b = c \pmod{2^k}$.

Variant 1: Adaption of Wolverine's [WYKW21] check

- bucketing with untrusted multiplication triples $\rightsquigarrow$ Beaver multiplication

Variant 2: Adaption of the Mac'n'Cheese [BMRS21] check

- needs large message space $\rightsquigarrow$ authenticate $\mathbb{Z}_{2^{k+s}}$ elements over $\mathbb{Z}_{2^{k+2s}}$

Prover $\mathcal{P}$		Verifier $\mathcal{V}$
	$\mathrm{Random}() \to [x]$ $\longrightarrow$	
	$\mathrm{Input}(x \cdot b) \to [z]$ $\longrightarrow$	
	$\longleftarrow$ $\eta \in_R \mathbb{Z}_{2^s}$	

Verifying Multiplications for $\mathbb{Z}_{2^k}$ [BBMRS21]

Task: Given $([a],[b],[c])$, verify that $a \cdot b = c \pmod{2^k}$.

Variant 1: Adaption of Wolverine's [WYKW21] check

- bucketing with untrusted multiplication triples $\rightsquigarrow$ Beaver multiplication

Variant 2: Adaption of the Mac'n'Cheese [BMRS21] check

- needs large message space $\rightsquigarrow$ authenticate $\mathbb{Z}_{2^{k+s}}$ elements over $\mathbb{Z}_{2^{k+2s}}$

Prover $\mathcal{P}$ — Verifier $\mathcal{V}$

$\mathcal{P} \rightarrow \mathcal{V}$: $\mathrm{Random}() \to [x]$

$\mathcal{P} \rightarrow \mathcal{V}$: $\mathrm{Input}(x \cdot b) \to [z]$

$\mathcal{V} \rightarrow \mathcal{P}$: $\eta \in_R \mathbb{Z}_{2^s}$

$\mathcal{P} \rightarrow \mathcal{V}$: $\mathrm{Open}(\eta \cdot [a] - [x]) \to \epsilon$

$\mathcal{P} \rightarrow \mathcal{V}$: $\mathrm{CheckZero}(\eta \cdot [c] - [z] - \epsilon \cdot [b])$

VOLE Performace ($\sigma = 40$)

Table 1: Run-time of the Extend operation in ns per VOLE and the communication cost in bit per VOLE. The benchmarks are parametrized by the ring size ℓ (i.e., using $\mathbb{Z}_{2^\ell}$). The computational security parameter is set to $\kappa = 128$. For statistical security $\sigma \in \{40, 80\}$, we target batch sizes of $n_o = 10^7$ and $n_o = 10^8$, and use LPN parameters (m, t, n).

σ	ℓ	Run-time LAN	Run-time WAN	Communication $\mathcal{S} \to \mathcal{R}$	Communication $\mathcal{R} \to \mathcal{S}$	Communication total
	$m = 553\,600,\ t = 2\,186,\ n = 10\,558\,380$					
	64	27.3	190.8	0.467	0.927	1.394
	104	40.7	186.7	0.509	0.955	1.464
	144	55.2	212.6	0.551	0.983	1.534
40	244	80.7	255.0	0.593	1.011	1.604
	$m = 773\,200,\ t = 15\,045,\ n = 100\,816\,545$					
	64	20.1	46.0	0.318	0.636	0.954
	104	33.2	58.9	0.347	0.655	1.002
	144	46.7	75.1	0.376	0.674	1.050
	244	76.7	102.8	0.405	0.694	1.098

VOLE Performace ($\sigma = 80$)

Table 2: Run-time of the Extend operation in ns per VOLE and the communication cost in bit per VOLE. The benchmarks are parametrized by the ring size ℓ (i.e., using $\mathbb{Z}_{2^\ell}$). The computational security parameter is set to $\kappa = 128$. For statistical security $\sigma \in \{40, 80\}$, we target batch sizes of $n_o = 10^7$ and $n_o = 10^8$, and use LPN parameters (m, t, n).

σ	ℓ	Run-time		Communication		
		LAN	WAN	$\mathcal{S} \to \mathcal{R}$	$\mathcal{R} \to \mathcal{S}$	total
	$m = 830\,800,\ t = 2\,013,\ n = 10\,835\,979$					
	64	27.6	171.9	0.431	0.853	1.284
	104	42.6	194.1	0.469	0.879	1.349
	144	59.4	217.1	0.508	0.905	1.413
80	244	89.3	277.4	0.547	0.931	1.477
	$m = 866\,800,\ t = 18\,114,\ n = 100\,913\,094$					
	64	21.4	48.2	0.383	0.765	1.148
	104	34.3	61.0	0.418	0.789	1.206
	144	49.2	76.0	0.453	0.812	1.264
	244	79.8	106.8	0.487	0.835	1.322

VOLE Perfomance Comparison with Wolverine [WYKW21]

Table 3: Run-times in ns per VOLE in different bandwidth settings, when generating ca. 10^7 VOLEs with 5 threads and statistical security $\sigma \geq 40$. The parameter ℓ denotes the size of a ring or field element. The numbers for Wolverine are taken from [WYKW21].

	ℓ	20 Mbit/s	50 Mbit/s	100 Mbit/s	500 Mbit/s	1 Gbit/s	10 Gbit/s
this work	64	110.0	68.7	55.0	50.2	50.6	50.4
	104	142.0	95.2	80.1	73.2	71.5	73.6
	144	178.6	134.7	119.3	111.6	112.6	113.3
	244	266.3	219.1	201.7	194.5	193.7	196.5
Wolverine	61	101.0	87.0	85.0	85.0	85.0	—

QuarkSilver Performance

Table 4: We measure the run-time of a batch of $\approx 10^7$ multiplications and their verification in ns per multiplication and the communication cost in bit per multiplication. The benchmarks are parametrized by the statistical security parameter σ, and the computational security parameter is set to $\kappa = 128$. For $\sigma = 40$, we use the ring of size $\ell = 162$, for $\sigma = 80$, we use $\ell = 244$.

σ		Run-time		Communication		
		LAN	WAN	$\mathcal{S} \to \mathcal{R}$	$\mathcal{R} \to \mathcal{S}$	total
40	vole	78.5	265.5	0.5	1.0	1.5
	mult	663.2	2 101.5	192.0	0.0	192.0
	check	28.2	38.2	0.0	0.0	0.0
	total	769.9	2 405.2	192.5	1.0	193.5
80	vole	125.3	345.6	0.5	0.9	1.5
	mult	680.7	2 767.2	256.0	0.0	256.0
	check	42.3	52.4	0.0	0.0	0.0
	total	848.3	3 165.2	256.5	0.9	257.5