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
Zero-Knowledge Proofs for Arithmetic Circuits

I know $w$ s.t. $C(w) = 1$!

Prover $\mathcal{P}$

$C$ is an arithmetic circuit over the ring $\mathbb{Z}_{2^k}$
I know $w$ s.t. $C(w) = 1!$

$C$ 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 $\mathbb{F}_p$

- $\mathbb{Z}_{2^k} = \mathbb{Z}/2^k\mathbb{Z} = \{0, \ldots, 2^k - 1\}, +, \cdot$ – 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

  - Easier to convert programs to corresponding circuits
  - 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
  - Common tricks don't work and protocols get more complicated
  - Proofs of security are harder
Computation over $\mathbb{Z}_{2^k}$ vs. Finite Fields $\mathbb{F}_p$

- $\mathbb{Z}_{2^k} = \mathbb{Z}/2^k\mathbb{Z} = \{0, \ldots, 2^k - 1\}, +, \cdot$ – 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
- easier to convert programs to corresponding circuits 😊
- 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
  - easier to convert programs to corresponding circuits 😊
  - 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
  - common tricks don’t work and protocols get more complicated 😞
  - proofs of security are harder 😞
I know w s.t. $C(w) = 1$!
I know $w$ s.t. $C(w) = 1!$

Prover $\mathcal{P}$

Ingredients:

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

Diagram:

- Input nodes: $[w_1], [w_2], \ldots, [w_n]$
- Intermediate node: $C$
- Output node: $[w_{\text{out}}]$
I know \( w \) s.t. \( C(w) = 1! \)

Ingredients:
1. linearly homomorphic commitments \([\cdot]\)
   - can compute \([z] \leftarrow a \cdot [x] + [y] + b\)
2. multiplication check
   - given \(([a], [b], [c])\), verify \( a \cdot b = c\)
I know $w$ s.t. $C(w) = 1!$

Prover $\mathcal{P}$

Setting:
- designated verifier
- linear communication
- linear time prover and verifier
- minimal overhead compared to circuit evaluation
  - computation and memory
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])
For ring $\mathbb{Z}_{2^k}$: authenticate $x \in \mathbb{Z}_{2^k}$ over the larger ring $\mathbb{Z}_{2^k+s}$

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

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

(cf. SPD$\mathbb{Z}_{2^k}$ [CDESX18], Appenzeller2Brie [BBMRS21])
For ring $\mathbb{Z}_{2^k}$: authenticate $x \in \mathbb{Z}_{2^k}$ over the larger ring $\mathbb{Z}_{2^{k+s}}$

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

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

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

Vector Oblivious Linear Evaluation:

\[\tilde{x} \in \mathbb{Z}_{2^\ell} \quad \Delta \in \mathbb{Z}_{2^s} \quad \tilde{K} \in \mathbb{Z}_{2^\ell} \quad \tilde{M} \in \mathbb{Z}_{2^\ell}\]

such that $\tilde{M} = \Delta \cdot \tilde{x} + \tilde{K} \pmod{2^\ell}$
VOLE for $\mathbb{Z}_{2^k}$
How to instantiate VOLE?

From Oblivious Transfer or Homomorphic Encryption

~⇒ communication at least linear in output size
How to instantiate VOLE?

From Oblivious Transfer or Homomorphic Encryption

\[ \rightarrow \text{ communication at least linear in output size} \]

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]
How to instantiate VOLE?

From Oblivious Transfer or Homomorphic Encryption

\[ \Rightarrow \text{communication at least linear in output size} \]

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

\[ \vec{s} \text{: short, uniform seed} \]
Learning Parity with Noise (LPN) Assumption

\[ \vec{s} \]: 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

\[ \vec{s} \cdot A + \vec{\epsilon} \]

- \( \vec{s} \): short, uniform seed
- \( A \): generating matrix of a random linear code
- \( \vec{\epsilon} \): sparse error vector (regular error \( \vec{\epsilon} = (\vec{\epsilon}_1 \ldots |\vec{\epsilon}_t)^T \) with \( \vec{\epsilon}_i \) one-hot)
Learning Parity with Noise (LPN) Assumption

\[ \vec{s} \text{: short, uniform seed} \]

\[ \vec{u} \text{: long, uniform vector} \]

\[ A \text{: generating matrix of a random linear code} \]

\[ \vec{e} \text{: sparse error vector (regular error } \vec{e} = (\vec{e}_1 | \cdots | \vec{e}_t)^T \text{ with } \vec{e}_i \text{ one-hot}) \]
Learning Parity with Noise (LPN) Assumption

\[ A \cdot \vec{s} + \vec{e} \approx C \]

- \( \vec{s} \): short, uniform seed
- \( \vec{e} \): sparse error vector (regular error \( \vec{e} = (\vec{e}_1 | \cdots | \vec{e}_t)^T \) with \( \vec{e}_i \) one-hot)
- \( \vec{u} \): long, uniform vector
- \( A \): generating matrix of a random linear code
Goal: expand \( m \) seed/base VOLEs into \( n \gg m \) VOLEs: \( \vec{z} = \Delta \cdot \vec{x} + \vec{y} \)
**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]
**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 $n/t$
   $\leadsto$ obtain $\vec{c} = \Delta \cdot \vec{e} + \vec{b}$ where $\vec{e}$ has $t$ non-zero entries

based on Wolverine [WYKW21]
**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 $n/t$ to 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

\[
\begin{align*}
\vec{x} &:= A \cdot \vec{u} + \vec{e} \\
\vec{z} &:= A \cdot \vec{w} + \vec{c} \\
\end{align*}
\Rightarrow \quad \vec{z} := \Delta \cdot \vec{x} + \vec{y}
\]

Receiver computes

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

based on Wolverine [WYKW21]
**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 $n/t$
   ~⇒ 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
\[
\begin{align*}
\vec{x} & := A \cdot \vec{u} + \vec{e} \\
\vec{z} & := A \cdot \vec{w} + \vec{c}
\end{align*}
\]  \[\Rightarrow \vec{z} := \Delta \cdot \vec{x} + \vec{y}\]

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

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

based on Wolverine [WYKW21]
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
\[
\begin{align*}
\vec{x} & := A \cdot \vec{u} + \vec{e} \\
\vec{z} & := A \cdot \vec{w} + \vec{c} \quad \Rightarrow \quad \vec{z} := \Delta \cdot \vec{x} + \vec{y}
\end{align*}
\]

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]
**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 $n/t$
   $\leadsto$ 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
\[
\begin{align*}
\vec{x} & := A \cdot \vec{u} + \vec{e} \\
\vec{z} & := A \cdot \vec{w} + \vec{c}
\end{align*}
\Rightarrow \vec{z} := \Delta \cdot \vec{x} + \vec{y}
\]

Receiver computes
\[
\begin{align*}
\vec{y} & := A \cdot \vec{v} + \vec{b}
\end{align*}
\]

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

\[ G(x) = G(x)_0 \parallel G(x)_1 \text{ length-doubling PRG} \]
GGM Construction for a PRF

\[ G(x) = G(x)_0 \parallel G(x)_1 \] length-doubling PRG
GGM Construction for a PRF

\[ G(x) = G(x)_0 \parallel G(x)_1 \text{ length-doubling PRG} \]
GGM Construction for a PRF

\[ G(x) = G(x)_0 \| G(x)_1 \] length-doubling PRG
GGM Construction for a **Punctured** PRF

\[ G(x) = G(x)_0 \| G(x)_1 \] length-doubling PRG
GGM Construction for a Punctured PRF

\[ G(x) = G(x)_0 \| G(x)_1 \] 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 $\mathbf{w} = \Delta \cdot \mathbf{u} + \mathbf{v}$ 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 $\mathbf{w} \approx \mathbf{v}$ 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 $S$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$
   - $\leadsto \mathcal{R}$ does not learn $\alpha$ and $S$ does not learn $v_\alpha$
**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 $F$
   - $\mathcal{R}$ samples a PRF key $k$ and
     - use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $S$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$
   - $\mathcal{R}$ does not learn $\alpha$ and $S$ does not learn $v_\alpha$

2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $S$ 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 \( S \)
   - 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 \( \alpha \) and \( S \) does not learn \( v_\alpha \)

2. compute \( w_\alpha = \Delta \cdot \beta + v_\alpha \)
   - use one base VOLE \( \delta = \Delta \cdot \beta + \gamma \)
   - \( \mathcal{R} \) sends \( d := \gamma - \sum_j v_j \)
   - \( S \) 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 \( \alpha \)
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 $F$
   - $\mathcal{R}$ samples a PRF key $k$ and
     - use $\log(n/t)$ OTs to obliviously transfer a punctured key $k\{\alpha\}$ to $S$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$
     - $\Rightarrow$ $\mathcal{R}$ does not learn $\alpha$ and $S$ does not learn $v_\alpha$

2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
     - $S$ 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 $u_\alpha = \beta \in \mathbb{R} \mathbb{Z}_{2^\ell} \setminus \{0\}$ 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 $S$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$
   - $\Rightarrow \mathcal{R}$ does not learn $\alpha$ and $S$ does not learn $v_\alpha$

2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $S$ 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 $\vec{w}$ with 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 $S$
   - $\mathcal{R}$ does not learn $\alpha$ and $S$ does not learn $v_\alpha$

2. compute
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $S$ 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 $S$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$
   - $\mathcal{R}$ does not learn $\alpha$ and $S$ does not learn $v_\alpha$

2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $S$ 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

\[ G(x) = G(x)_0 \parallel G(x)_1 \] 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

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

\[ \rightsquigarrow \text{compute and verify universal hash } h(t_0, \ldots, t_7) \]
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 $S$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$
   - $\mathcal{R}$ does not learn $\alpha$ and $S$ does not learn $v_\alpha$

2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $S$ 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])
**Goal**: generate $\vec{w} = \Delta \cdot \vec{u} + \vec{v}$ where $u_\alpha = \beta \in \mathbb{Z}_{2^\ell} \setminus \{0\}$ 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 $S$
   - set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$
   - $\mathcal{R}$ does not learn $\alpha$ and $S$ does not learn $v_\alpha$

2. compute $w_\alpha = \Delta \cdot \beta + v_\alpha$
   - use one base VOLE $\delta = \Delta \cdot \beta + \gamma$
   - $\mathcal{R}$ sends $d := \gamma - \sum_j v_j$
   - $S$ 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 $\chi_1, \ldots, \chi_n \in \{0, 1\}$
Single-Point VOLE Protocol

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

1. distribute long random vector $\vec{w} \approx \vec{v}$ with a punctured PRF
   - $\mathcal{R}$ samples a PRF key $k$ and
   - $\mathcal{S}$ set $w_i := v_i := F(k, i)$ for all $i \neq \alpha$ and $v_\alpha := F(k, \alpha)$
   - $\mathcal{R}$ does not learn $\alpha$ and $\mathcal{S}$ does not learn $v_\alpha$

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 $\chi_1, \ldots, \chi_n \in \{0, 1\}$

- malicious $\mathcal{R}$ can guess $\chi_\alpha$ with probability 1/2
  - adjust functionality to allow leakage of $\alpha$ with probability 1/2
  - increase noise rate of the error to compensate
QuarkSilver – More Efficient Multiplication Check for $\mathbb{Z}_{2^k}$
Goal: Given ([a], [b], [c]), verify that $\tilde{a} \cdot \tilde{b} = \tilde{c}$ (mod $2^k$)
Verifying Multiplications for $\mathbb{Z}_{2^k}$ — QuarkSilver

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

**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 $\Delta$:

\[
\Delta \cdot K[c] - K[a] \cdot K[b] = (M[a] \cdot M[b]) + (M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a]) \cdot \Delta + (\tilde{c} - \tilde{a} \cdot \tilde{b}) \cdot \Delta^2
\]

- Known by $\mathcal{V}$
- Known by $\mathcal{P}$
- Known by $\mathcal{P}$
- $= 0$ if $\mathcal{P}$ honest
Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

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

**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 $\Delta$:

\[
\Delta \cdot K[c] - K[a] \cdot K[b] = (M[a] \cdot M[b]) + (M[c] \cdot \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a]) \cdot \Delta + (\tilde{c} - \tilde{a} \cdot \tilde{b}) \cdot \Delta^2
\]

known by $\mathcal{V}$
known by $\mathcal{P}$
known by $\mathcal{P}$

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 $\Delta$:

\[
\begin{align*}
\Delta \cdot K[c] - K[a] \cdot K[b] &= (M[a] \cdot M[b]) + (M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a]) \cdot \Delta + (\tilde{c} - \tilde{a} \cdot \tilde{b}) \cdot \Delta^2 \\
&= 0 \text{ if } \mathcal{P} \text{ honest}
\end{align*}
\]

**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 \( \tilde{a} \cdot \tilde{b} = \tilde{c} \) (mod \( 2^k \))

**Observation from QuickSilver [YSWW21]:** Convert the three MAC equations

\[
M[x] = K[x] + \tilde{x} \cdot \Delta \quad \text{for } x \in \{a, b, c\}
\]

into a polynomial in \( \Delta \):

\[
\Delta \cdot K[c] - K[a] \cdot K[b] = (M[a] \cdot M[b]) + (M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a]) \cdot \Delta + (\tilde{c} - \tilde{a} \cdot \tilde{b}) \cdot \Delta^2
\]

- known by \( \mathcal{V} \)
- known by \( \mathcal{P} \)
- known by \( \mathcal{P} \)

\( = 0 \) if \( \mathcal{P} \) honest

**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 $\Delta$:

$$\Delta \cdot K[c] - K[a] \cdot K[b]$$

known by $\mathcal{V}$

$$= (M[a] \cdot M[b]) + (M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a]) \cdot \Delta + (\tilde{c} - \tilde{a} \cdot \tilde{b}) \cdot \Delta^2$$

known by $\mathcal{P}$

known by $\mathcal{P}$

$= 0$ if $\mathcal{P}$ honest

**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
- $e := \tilde{c} - \tilde{a} \cdot \tilde{b} \not\equiv 0 \pmod{2^k} \implies p$ has degree 2
Verifying Multiplications for $\mathbb{Z}_{2^k}$ – QuarkSilver

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

**Observation from QuickSilver [YSWW21]:** Convert the three MAC equations

\[
M[x] = K[x] + \tilde{x} \cdot \Delta \quad \text{for} \quad x \in \{a, b, c\}
\]

into a polynomial in $\Delta$:

\[
\Delta \cdot K[c] - K[a] \cdot K[b] = (M[a] \cdot M[b]) + (M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a]) \cdot \Delta + (\tilde{c} - \tilde{a} \cdot \tilde{b}) \cdot \Delta^2
\]

known by $V$

known by $P$

known by $P$

= 0 if $P$ honest

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

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

- **Field $\mathbb{F}_q$:** polynomial $p$ has at most two roots $\Leftrightarrow$ 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 $\Delta$:

$$\Delta \cdot K[c] - K[a] \cdot K[b] = (M[a] \cdot M[b]) + (M[c] - \tilde{a} \cdot M[b] - \tilde{b} \cdot M[a]) \cdot \Delta + (\tilde{c} - \tilde{a} \cdot \tilde{b}) \cdot \Delta^2$$

known by $\mathcal{V}$

known by $\mathcal{P}$

known by $\mathcal{P}$

$= 0$ if $\mathcal{P}$ honest

**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 $\Rightarrow$ 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
  - $\Rightarrow$ more expensive arithmetic and communication
  - but provides native 64 bit arithmetic
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
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
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 $2^\ell > 2^k$.
  \[ \Rightarrow \quad \text{Can we reduce the communication overhead compared to } k? \]
- Many recent protocols for fields use very efficient checks based on polynomials.
  \[ \Rightarrow \quad \text{Can we get similar efficient alternatives over rings?} \]
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 Mozzarella on ePrint: https://ia.cr/2022/819


Emoji graphics licensed under CC-BY 4.0: https://creativecommons.org/licenses/by/4.0/ Copyright 2020 Twitter, Inc and other contributors
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 $\Delta$
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 $K[x] \in R \mathbb{F}$

breaking binding $\implies$ 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 $\Delta$

**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 $\Delta$

**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 \[\text{[BBMRS21]}\]

**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 $\Delta$

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

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

$P$ publishes $\tilde{x}$, $M[x]$
$V$ checks $M[x] = \Delta \cdot \tilde{x} + K[x] \pmod{2^\ell}$
Upper bits of $\tilde{x}$ leak information about intermediate values

$\Rightarrow$ use random $[r]$ to mask upper bits before opening:
- $[z] \leftarrow [x] + 2^k \cdot [r]$
Linear Operations: to compute $[z] \leftarrow \alpha \cdot [x] + [y] + c$, set

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

Open:

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

- $K[z] \leftarrow \alpha \cdot K[x] + K[y] - \Delta \cdot c \pmod{2^\ell}$
Linear Operations and Openings

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

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

**Open**:

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

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

*upper bits of \( \tilde{x} \) leak information about intermediate values*
**Linear Operations**:

To compute \( [z] \leftarrow \alpha \cdot [x] + [y] + c \), set

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

**Open**:

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

⚠️ upper bits of \( \tilde{x} \) leak information about intermediate values

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

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

\(\Rightarrow\) 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 \(M[x_1], \ldots, M[x_n]\)
Task: Given ([a], [b], [c]), verify that \( a \cdot b = c \) (mod \( 2^k \)).
Verifying Multiplications for $\mathbb{Z}_{2^k}$

**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 $\leadsto$ Beaver multiplication
Verifying Multiplications for $\mathbb{Z}_{2^k}$

**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 $\mathbb{Z}_{2^{k+2s}}$
Verifying Multiplications for $\mathbb{Z}_{2^k}$

**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 $\leadsto$ Beaver multiplication

**Variant 2**: Adaption of the Mac’n’Cheese [BMRS21] check
- needs large message space $\leadsto$ authenticate $\mathbb{Z}_{2^{k+s}}$ elements over $\mathbb{Z}_{2^{k+2s}}$

<table>
<thead>
<tr>
<th>Prover $\mathcal{P}$</th>
<th>Random() $\rightarrow [x]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input($x \cdot b$)</td>
<td>$\rightarrow [z]$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verifier $\mathcal{V}$</th>
<th></th>
</tr>
</thead>
</table>
**Task**: Given ([a], [b], [c]), verify that \( a \cdot b = c \) (mod \( 2^k \)).

**Variant 1**: Adaption of Wolverine's [WYKW21] check
- bucketing with untrusted multiplication triples \( \leadsto \) Beaver multiplication

**Variant 2**: Adaption of the Mac’n’Cheese [BMRS21] check
- needs large message space \( \leadsto \) authenticate \( \mathbb{Z}_{2^k+s} \) elements over \( \mathbb{Z}_{2^{k+2s}} \)

**Diagram**:

\[
\begin{align*}
\text{Prover } \mathcal{P} & \quad \text{Random()} \rightarrow [x] \quad \text{Verifier } \mathcal{V} \\
& \quad \text{Input}(x \cdot b) \rightarrow [z] \\
& \quad \eta \in_R \mathbb{Z}_{2^s}
\end{align*}
\]
Verifying Multiplications for $\mathbb{Z}_{2^k}$

**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 $\leadsto$ Beaver multiplication

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

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

\[
\begin{align*}
\text{Prover } & \mathcal{P} \\
\text{Verifier } & \mathcal{V} \\
\text{Random()} & \rightarrow [x] \\
\text{Input}(x \cdot b) & \rightarrow [z] \\
\eta & \in_R \mathbb{Z}_{2^s} \\
\text{Open}(\eta \cdot [a] - [x]) & \rightarrow \epsilon \\
\text{CheckZero}(\eta \cdot [c] - [z] - \epsilon \cdot [b]) & 
\end{align*}
\]
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 $\ell$ (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 $(m, t, n)$.

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$\ell$</th>
<th>Run-time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LAN</td>
<td>WAN</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>244</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>104</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>144</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>244</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$m = 553 600, \ t = 2 186, \ n = 10 558 380$

$m = 773 200, \ t = 15 045, \ n = 100 816 545$
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 $\ell$ (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 $(m, t, n)$.

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$\ell$</th>
<th>Run-time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>LAN</td>
<td>WAN</td>
</tr>
<tr>
<td>$m = 830800$, $t = 2013$, $n = 10,835,979$</td>
<td>64</td>
<td>27.6</td>
<td>171.9</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>42.6</td>
<td>194.1</td>
</tr>
<tr>
<td></td>
<td>144</td>
<td>59.4</td>
<td>217.1</td>
</tr>
<tr>
<td></td>
<td>244</td>
<td>89.3</td>
<td>277.4</td>
</tr>
<tr>
<td>$m = 866800$, $t = 18,114$, $n = 100,913,094$</td>
<td>64</td>
<td>21.4</td>
<td>48.2</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>34.3</td>
<td>61.0</td>
</tr>
<tr>
<td></td>
<td>144</td>
<td>49.2</td>
<td>76.0</td>
</tr>
<tr>
<td></td>
<td>244</td>
<td>79.8</td>
<td>106.8</td>
</tr>
</tbody>
</table>
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 $\ell$ denotes the size of a ring or field element. The numbers for Wolverine are taken from [WYKW21].

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>20 Mbit/s</th>
<th>50 Mbit/s</th>
<th>100 Mbit/s</th>
<th>500 Mbit/s</th>
<th>1 Gbit/s</th>
<th>10 Gbit/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>this work</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>110.0</td>
<td>68.7</td>
<td>55.0</td>
<td>50.2</td>
<td>50.6</td>
<td>50.4</td>
</tr>
<tr>
<td>104</td>
<td>142.0</td>
<td>95.2</td>
<td>80.1</td>
<td>73.2</td>
<td>71.5</td>
<td>73.6</td>
</tr>
<tr>
<td>144</td>
<td>178.6</td>
<td>134.7</td>
<td>119.3</td>
<td>111.6</td>
<td>112.6</td>
<td>113.3</td>
</tr>
<tr>
<td>244</td>
<td>266.3</td>
<td>219.1</td>
<td>201.7</td>
<td>194.5</td>
<td>193.7</td>
<td>196.5</td>
</tr>
<tr>
<td>Wolverine</td>
<td>61</td>
<td>101.0</td>
<td>87.0</td>
<td>85.0</td>
<td>85.0</td>
<td>85.0</td>
</tr>
</tbody>
</table>
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 $\sigma$, 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 $\ell = 244$.

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>Run-time</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAN</td>
<td>WAN</td>
</tr>
<tr>
<td>vole</td>
<td>78.5</td>
<td>265.5</td>
</tr>
<tr>
<td>mult</td>
<td>663.2</td>
<td>2101.5</td>
</tr>
<tr>
<td>check</td>
<td>28.2</td>
<td>38.2</td>
</tr>
<tr>
<td>total</td>
<td>769.9</td>
<td>2405.2</td>
</tr>
<tr>
<td>vole</td>
<td>125.3</td>
<td>345.6</td>
</tr>
<tr>
<td>mult</td>
<td>680.7</td>
<td>2767.2</td>
</tr>
<tr>
<td>check</td>
<td>42.3</td>
<td>52.4</td>
</tr>
<tr>
<td>total</td>
<td>848.3</td>
<td>3165.2</td>
</tr>
</tbody>
</table>