Constructions for digital signature Part I: Introduction to MPC-in-the-Head

Thibauld Feneuil

NIST PQC Seminar

May 21, 2024, online
Table of Contents

- Introduction
- MPC-in-the-Head: general principle
- From MPC-in-the-Head to signatures
- Optimisations and variants
- Conclusion
Introduction
How to build signature schemes?

**Hash & Sign**

\[ H(m) \xrightarrow{F_{pk}} \sigma \]

- Short signatures
- “Trapdoor” in the public key

Very hard to compute
How to build signature schemes?

**Hash & Sign**

- $m \rightarrow H(m) \rightarrow \sigma$

**From an identification scheme**

- I know the private key.
- I am convinced.

**Short signatures**
- "Trapdoor" in the public key

**Large(r) signatures**
- Short public key
How to build signature schemes?

**Hash & Sign**

\[ H(m) \]

\[ F_{pk} \]

\[ F^{-1}_{pk} \]

**Short signatures**

- "Trapdoor" in the public key

**Large(r) signatures**

- Very hard to compute

From an identification scheme

I know the private key.

I am convinced.

- Short public key
**Identification Scheme**

- **Completeness:** $\text{Pr}[\text{verif } \checkmark \mid \text{honest prover}] = 1$
- **Soundness:** $\text{Pr}[\text{verif } \checkmark \mid \text{malicious prover}] \leq \varepsilon$ (e.g. $2^{-128}$)
- **Zero-knowledge:** verifier learns nothing on $\cdot$.
Identification Scheme

I know \(\mathbf{0}\).  

\[
\text{Challenge } 1 = \text{Hash}(m, \text{Commitment}) \\
\vdots \\
\text{Challenge } n = \text{Hash}(m, \text{Response } n - 1)
\]

Prover

Verifier

Fiat-Shamir Transformation

\(m\): message to sign
MPC in the Head

- [IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Amit Sahai: “Zero-knowledge from secure multiparty computation” (STOC 2007)

- Turn a multiparty computation (MPC) into an identification scheme / zero-knowledge proof of knowledge

- **Generic**: can be applied to any cryptographic problem
MPC in the Head

- [IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Amit Sahai: “Zero-knowledge from secure multiparty computation” (STOC 2007)

- Convenient to build (candidate) post-quantum signature schemes

- Picnic: submission to NIST (2017)

- First round of recent NIST call: 7~9 MPCitH schemes / 40 submissions
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \( [x] \)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

\( x \quad \text{Hash function} \quad \text{signature} \quad \text{msg} \)

Zero-knowledge proof

Prover \( x \) \quad \text{Verifier} \( y \)

OK you know \( x \)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

\[ x \]

signature

Hash function

Zero-knowledge proof

Prover

\[ x \]

Verifier

\[ y \]

OK you know \(x\)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]_i\)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

Hash function

Zero-knowledge proof

Prover

OK you know \(x\)

Verifier

\(x\)

\(y\)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([\lfloor x \rfloor]\)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

\[ x \]

Hash function

signature

Zero-knowledge proof

Prover

Verifier

OK you know \(x\)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \( \|x\| \)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

Hash function

Prover

Verifier

OK you know \( x \)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\)

Joint evaluation of:

\[ g(x) = \begin{cases} 
  \text{Accept} & \text{if } F(x) = y \\
  \text{Reject} & \text{if } F(x) \neq y
\end{cases} \]

**MPC-in-the-Head transform**

Signature scheme

\[ x \]

\[ \text{Hash function} \]

\[ \text{msg} \]

\[ \text{signature} \]

Zero-knowledge proof

Prover

Verifier

OK you know \(x\)
MPCitH: general principle
MPC-in-the-Head Framework

Secret \( x \) which satisfies some public relation \( y = F(x) \)

How to build a zero-knowledge proof of knowledge for \( x \)?
MPC-in-the-Head Framework

Secret $x$ which satisfies some public relation $y = F(x)$

Sharing $[[x]]$ of the secret $x$

Additive secret sharing:

$$x = [[x]]_1 + [[x]]_2 + \ldots + [[x]]_N$$

Shamir’s secret sharing:

$$\forall i, [[x]]_i = P(e_i),$$

where $P$ is a random degree-$\ell$ polynomial such that $P(0) = x$. 
MPC-in-the-Head Framework

Secret $x$ which satisfies some public relation $y = F(x)$

Sharing $[x]$ of the secret $x$

Additive secret sharing:
$x = [x]_1 + [x]_2 + \ldots + [x]_N$

Shamir’s secret sharing:
$\forall i, [x]_i = P(e_i)$,
where $P$ is a random degree-$\ell$ polynomial such that $P(0) = x$. 
MPC-in-the-Head Framework

Secret $x$ which satisfies
some public relation $y = F(x)$

Sharing $[[x]]$ of the secret $x$

Additive secret sharing:
$x = [[x]]_1 + [[x]]_2 + \ldots + [[x]]_N$

Shamir’s secret sharing:
$\forall i, [[x]]_i = P(e_i)$,
where $P$ is a random degree-$\ell$
 polynomial such that $P(0) = x$.

If $x := 42$ lives in $\mathbb{F}_{1021}$, a possible sharing of $x$ is
\[
x = 429 + 19 + 583 + 231 + 822 \quad \text{over} \quad \mathbb{F}_{1021}
\]
MPC-in-the-Head Framework

Secret $x$ which satisfies some public relation $y = F(x)$

Sharing $[[x]]$ of the secret $x$

Input sharing $[[x]]$

Joint evaluation of:

$$g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases}$$
MPC model: discrete logarithm

- Secret $x$ satisfies $y = z^x$, with $z$ public.
- We want a multiparty computation that computes

$$g(x) = \begin{cases} 
\text{Accept} & \text{if } z^x = y \\
\text{Reject} & \text{if } z^x \neq y
\end{cases}$$
MPC model: discrete logarithm

- Secret \( x \) satisfies \( y = z^x \), with \( z \) public.

- We want a multiparty computation that computes

\[
g(x) = \begin{cases} 
  \text{Accept} & \text{if } z^x = y \\
  \text{Reject} & \text{if } z^x \neq y
\end{cases}
\]

- Party \( i \):
  
  - Receive the \( i^{\text{th}} \) share \( [x]_i \)
  
  - Compute \( [z^x]_i \leftarrow z^{[x]_i} \).
  
  - Broadcast \( [z^x]_i \).
  
  - Receive all the broadcasted values \( [z^x]_1, \ldots, [z^x]_N \)
  
  - Recover \( z^x \) and check that \( y \).
MPC model: discrete logarithm

- Secret $x$ satisfies $y = z^x$, with $z$ public.

- We want a multiparty computation that computes

$$g(x) = \begin{cases} 
\text{Accept} & \text{if } z^x = y \\
\text{Reject} & \text{if } z^x \neq y 
\end{cases}$$

- Party $i$:
  - Receive the $i^{th}$ share $\llbracket x \rrbracket_i$
  - Compute $\llbracket z^x \rrbracket_i \leftarrow z^{\llbracket x \rrbracket_i}$.
  - Broadcast $\llbracket z^x \rrbracket_i$.
  - Receive all the broadcasted values $\llbracket z^x \rrbracket_1, \ldots, \llbracket z^x \rrbracket_N$
  - Recover $z^x$ and check that $y$.

\[
\begin{align*}
z &= 3 \pmod{1907} \\
x &= 575 \\
y &= 1467 = z^x \pmod{1907}
\end{align*}
\]
MPC model: discrete logarithm

- Secret $x$ satisfies $y = z^x$, with $z$ public.
- We want a multiparty computation that computes

$$g(x) = \begin{cases} 
\text{Accept} & \text{if } z^x = y \\
\text{Reject} & \text{if } z^x \neq y 
\end{cases}$$

- **Party $i$:**
  - Receive the $i^{th}$ share $\llbracket x \rrbracket_i$
  - Compute $\llbracket z^x \rrbracket_i \leftarrow z^{\llbracket x \rrbracket_i}$.
  - Broadcast $\llbracket z^x \rrbracket_i$.
  - Receive all the broadcasted values $\llbracket z^x \rrbracket_1, \ldots, \llbracket z^x \rrbracket_N$
  - Recover $z^x$ and check that $y$.

\[ z = 3 \pmod{1907} \quad x = 575 \quad y = 1467 = z^x \pmod{1907} \]
\[ \llbracket x \rrbracket_1 = 180, \quad \llbracket x \rrbracket_2 = 397, \quad \llbracket x \rrbracket_3 = 649, \quad \llbracket x \rrbracket_4 = 713, \quad \llbracket x \rrbracket_5 = 542 \]
\[ x = \llbracket x \rrbracket_1 + \llbracket x \rrbracket_2 + \llbracket x \rrbracket_3 + \llbracket x \rrbracket_4 + \llbracket x \rrbracket_5 \pmod{953} \]
MPC model: discrete logarithm

- Secret $x$ satisfies $y = z^x$, with $z$ public.
- We want a multiparty computation that computes
  
  \[ g(x) = \begin{cases} 
  \text{Accept} & \text{if } z^x = y \\
  \text{Reject} & \text{if } z^x \neq y 
  \end{cases} \]

- **Party $i$:**
  - Receive the $i^{th}$ share $\llbracket x \rrbracket_i$
  - Compute $\llbracket z^x \rrbracket_i \leftarrow z^{\llbracket x \rrbracket_i}$.
  - Broadcast $\llbracket z^x \rrbracket_i$.
  - Receive all the broadcasted values $\llbracket z^x \rrbracket_1, \ldots, \llbracket z^x \rrbracket_N$
  - Recover $z^x$ and check that $y$.

\[
\begin{align*}
  z &= 3 \pmod{1907} & x &= 575 & y &= 1467 = z^x \pmod{1907} \\
  \llbracket x \rrbracket_1 &= 180, \quad \llbracket x \rrbracket_2 = 397, \quad \llbracket x \rrbracket_3 = 649, \quad \llbracket x \rrbracket_4 = 713, \quad \llbracket x \rrbracket_5 = 542 \\
  x &= \llbracket x \rrbracket_1 + \llbracket x \rrbracket_2 + \llbracket x \rrbracket_3 + \llbracket x \rrbracket_4 + \llbracket x \rrbracket_5 \pmod{953}
\end{align*}
\]
MPC model: discrete logarithm

- Secret $x$ satisfies $y = z^x$, with $z$ public.
- We want a multiparty computation that computes

$$g(x) = \begin{cases} 
\text{Accept} & \text{if } z^x = y \\
\text{Reject} & \text{if } z^x \neq y 
\end{cases}$$

- **Party $i$:**
  - Receive the $i^{th}$ share $[[x]]_i$
  - Compute $[[z^x]]_i \leftarrow z^{|x|}_i$.
  - Broadcast $[[z^x]]_i$.
  - Receive all the broadcasted values $[[z^x]]_1, \ldots, [[z^x]]_N$
  - Recover $z^x$ and check that $y$.

\[
\begin{align*}
  z &= 3 \pmod{1907} & x &= 575 & y &= 1467 = z^x \pmod{1907} \\
  [[x]]_1 &= 180, & [[x]]_2 &= 397, & [[x]]_3 &= 649, & [[x]]_4 &= 713, & [[x]]_5 &= 542 \\
  x &= [[x]]_1 + [[x]]_2 + [[x]]_3 + [[x]]_4 + [[x]]_5 \pmod{953}
\end{align*}
\]
MPC model: discrete logarithm

- Secret $x$ satisfies $y = z^x$, with $z$ public.
- We want a multiparty computation that computes
  \[ g(x) = \begin{cases} 
  \text{Accept} & \text{if } z^x = y \\
  \text{Reject} & \text{if } z^x \neq y 
  \end{cases} \]
- **Party $i$:**
  - Receive the $i^{th}$ share $[x]_i$
  - Compute $[z^x]_i \leftarrow z^{[x]_i}$.
  - Broadcast $[z^x]_i$.
  - Receive all the broadcasted values $[z^x]_1, \ldots, [z^x]_N$
  - Recover $z^x$ and check that $y$. 

\[ z = 3 \pmod{1907} \quad x = 575 \quad y = 1467 = z^x \pmod{1907} \]

\[ [x]_1 = 180, \quad [x]_2 = 397, \quad [x]_3 = 649, \quad [x]_4 = 713, \quad [x]_5 = 542 \]

\[ x = [x]_1 + [x]_2 + [x]_3 + [x]_4 + [x]_5 \pmod{953} \]
MPC model: discrete logarithm

- Secret $x$ satisfies $y = z^x$, with $z$ public.
- We want a multiparty computation that computes
  \[g(x) = \begin{cases} 
  \text{Accept} & \text{if } z^x = y \\
  \text{Reject} & \text{if } z^x \neq y
  \end{cases}\]
- Party $i$:
  - Receive the $i^{th}$ share $\|x\|_i$
  - Compute $\|z^x\|_i \leftarrow z^{\|x\|_i}$.

If someone sees the computation of all the parties except one, it leaks no information on $x$. 😃

\[
\begin{align*}
  z &= 3 \pmod{1907} \\
  x &= \mathbf{180}, \quad y = 1467 \\
  \|x\|_1 &= 180, \quad \|x\|_2 = 397, \quad \|x\|_3 = 649, \quad \|x\|_4 = 713, \quad \|x\|_5 = 542 \\
  x &= \|x\|_1 + \|x\|_2 + \|x\|_3 + \|x\|_4 + \|x\|_5 \pmod{953}
\end{align*}
\]
MPC model

- Jointly compute
  
  \[ g(x) = \begin{cases} 
  \text{Accept} & \text{if } F(x) = y \\
  \text{Reject} & \text{if } F(x) \neq y 
  \end{cases} \]

- \((N - 1)\) private: the views of any \(N - 1\) parties provide no information on \(x\)

- Semi-honest model: assuming that the parties follow the steps of the protocol

\[ x = [x]_1 + [x]_2 + \ldots + [x]_N \]
MPC model

- Jointly compute
  \[ g(x) = \begin{cases} 
  \text{Accept} & \text{if } F(x) = y \\
  \text{Reject} & \text{if } F(x) \neq y 
  \end{cases} \]

- \((N - 1)\) private: the views of any \(N - 1\) parties provide no information on \(x\)

- Semi-honest model: assuming that the parties follow the steps of the protocol

- Broadcast model
  - Parties locally compute on their shares \([[x]] \mapsto [[\alpha]]\)
  - Parties broadcast \([[\alpha]]\) and recompute \(\alpha\)
  - Parties start again (now knowing \(\alpha\))

\[ x = [[x]]_1 + [[x]]_2 + \ldots + [[x]]_N \]
MPCitH transform

Prover

Verifier
① Generate and commit shares
\([x] = ([x]_1, \ldots, [x]_N)\)

\[
\begin{align*}
\text{Com}^{\rho_1}([x]_1) \\
\cdots \\
\text{Com}^{\rho_N}([x]_N)
\end{align*}
\]
MPCitH transform

1. Generate and commit shares
\[ [x] = ([x]_1, \ldots, [x]_N) \]

2. Run MPC in their head

\[
\begin{align*}
\text{Com}^\rho([x]_1) \\
\vdots \\
\text{Com}^\rho([x]_N)
\end{align*}
\]

send broadcast
\[ [[\alpha]]_1, \ldots, [[\alpha]]_N \]
MPCitH transform

1. Generate and commit shares
   \[ [[x]] = ([[x]]_1, ..., [[x]]_N) \]

2. Run MPC in their head

   Prover

   \[ \text{Com}^{\rho_i}([[x]]_1) \]
   \[ \text{...} \]
   \[ \text{Com}^{\rho_N}([[x]]_N) \]

   \[ \text{send broadcast} \]
   \[ [[\alpha]]_1, ..., [[\alpha]]_N \]

3. Choose a random party
   \[ i^* \leftarrow \$ \{1, ..., N\} \]

Verifier
MPCitH transform

① Generate and commit shares
\[[x] = ([x]_1, \ldots, [x]_N)\]

② Run MPC in their head

Com^\rho_1([x]_1)
\ldots
Com^\rho_N([x]_N)

send broadcast
\[\|\alpha\|_1, \ldots, \|\alpha\|_N\]

③ Choose a random party
\[i^* \leftarrow \{1, \ldots, N\}\]

④ Open parties \{1, \ldots, N\}/\{i^*\}

Prover

Verifier
MPCitH transform

① Generate and commit shares
\[ [x] = ([x]_1, \ldots, [x]_N) \]

② Run MPC in their head

③ Choose a random party
\[ i^* \leftarrow \{1, \ldots, N\} \]

④ Open parties \(\{1, \ldots, N\}\)\(\backslash\{i^*\}\)

⑤ Check \(\forall i \neq i^*\)
- Commitments \(\text{Com}^\rho([x]_i)\)
- MPC computation \([\alpha]_i = \varphi([x]_i)\)
Check \(\tilde{g}(y, \alpha) = \text{Accept}\)

Prover

Verifier
MPCitH transform

① Generate and commit shares
\[ [x] = ([x]_1, \ldots, [x]_N) \]

② Run MPC in their head

Com^{\rho_1}([x]_1) \\
\ldots \\
Com^{\rho_N}([x]_N)

send broadcast
\[ \|\alpha\|_1, \ldots, \|\alpha\|_N \]

③ Choose a random party
\[ i^* \leftarrow \$ \{1, \ldots, N\} \]

⑤ Check \( \forall i \neq i^* \)
- Commitments \( \text{Com}^{\rho_i}([x]_i) \)
- MPC computation \( \|\alpha\|_i = \varphi([x]_i) \)
Check \( \tilde{g}(y, \alpha) = \text{Accept} \)

④ Open parties \( \{1, \ldots, N\}\setminus\{i^*\} \)

Prover

Verifier

✅ Completeness

✅ Zero-Knowledge
MPCitH transform

1. Generate and commit shares
\[ [x] = ([x]_1, \ldots, [x]_N) \]

We have \( F(x) \neq y \) where
\[ x := [x]_1 + \ldots + [x]_N \]
MPCitH transform

① Generate and commit shares

\[
\llbracket x \rrbracket = (\llbracket x \rrbracket_1, \ldots, \llbracket x \rrbracket_N)
\]

We have \( F(x) \neq y \) where

\[
x := \llbracket x \rrbracket_1 + \ldots + \llbracket x \rrbracket_N
\]

② Run MPC in their head

Com\(^{\rho_1}(\llbracket x \rrbracket_1) \]

\[
\ldots
\]

Com\(^{\rho_N}(\llbracket x \rrbracket_N)\]

send broadcast

\[
\llbracket \alpha \rrbracket_1, \ldots, \llbracket \alpha \rrbracket_N
\]

Malicious Prover

Verifier
**MPCitH transform**

1. Generate and commit shares
   \[ \llbracket x \rrbracket = (\llbracket x \rrbracket_1, \ldots, \llbracket x \rrbracket_N) \]
   
   We have \( F(x) \neq y \) where
   \[ x := \llbracket x \rrbracket_1 + \ldots + \llbracket x \rrbracket_N \]

2. Run MPC in their head
   \[ \llbracket x \rrbracket_1 \quad \llbracket x \rrbracket_2 \quad \llbracket x \rrbracket_3 \quad \llbracket x \rrbracket_4 \quad \llbracket x \rrbracket_N \]

3. Choose a random party
   \[ i^* \leftarrow \$ \{1, \ldots, N\} \]

---

**Malicious Prover**
MPCitH transform

1. Generate and commit shares
   \[ [[x]] = ([[x]]_1, \ldots, [[x]]_N) \]
   We have \( F(x) \neq y \) where
   \( x := [[x]]_1 + \cdots + [[x]]_N \)

2. Run MPC in their head
   \[ \text{Com}^{\rho_1}([[x]]_1) \]
   \[ \cdots \]
   \[ \text{Com}^{\rho_N}([[x]]_N) \]
   send broadcast
   \[ [[\alpha]]_1, \ldots, [[\alpha]]_N \]

3. Choose a random party
   \( i^* \leftarrow \$ \{1, \ldots, N\} \)
   \( ([[[x]]_i, \rho_i])_{i \neq i^*} \)

4. Open parties \( \{1, \ldots, N\} \setminus \{i^*\} \)

Malicious Prover

Verifier
MPCitH transform

1. Generate and commit shares
   \[ [[x]] = ([[x]]_1, \ldots, [[x]]_N) \]
   We have \( F(x) \neq y \) where
   \( x := [[x]]_1 + \ldots + [[x]]_N \)

2. Run MPC in their head

3. Choose a random party
   \( i^* \leftarrow \{1, \ldots, N\} \)

4. Open parties \( \{1, \ldots, N\} \setminus \{i^*\} \)

5. Check \( \forall i \neq i^* \)
   - Commitments \( \text{Com}^{\rho_i}([[x]]_i) \)
   - MPC computation \( [[\alpha]]_i = \varphi([[x]]_i) \)
   Check \( \tilde{g}(y, \alpha) = \text{Accept} \)

Malicious Prover

Verifier

Cheating detected!
MPCitH transform

1. Generate and commit shares
   \[ [x] = ([x]_1, \ldots, [x]_N) \]
   We have \( F(x) \neq y \) where
   \( x := [x]_1 + \cdots + [x]_N \)

2. Run MPC in their head

3. Choose a random party
   \( i^* \leftarrow \{1, \ldots, N\} \)

4. Open parties \( \{1, \ldots, N\} \setminus \{i^*\} \)

5. Check \( \forall i \neq i^* \)
   - Commitments \( \text{Com}^\rho([x]_i) \)
   - MPC computation \( [\alpha]_i = \varphi([x]_i) \)
   Check \( \tilde{g}(y, \alpha) = \text{Accept} \)

**Malicious Prover**

**Verifier**

Seems OK.
MPCitH transform

- Zero-knowledge $\iff$ MPC protocol is $(N - 1)$-private
MPCitH transform

- Zero-knowledge $\iff$ MPC protocol is $(N - 1)$-private

- Soundness:

  $\Pr(\text{malicious prover convinces the verifier}) = \Pr(\text{corrupted party remains hidden}) = \frac{1}{N}$
MPCitH transform

- **Zero-knowledge** \iff MPC protocol is \((N - 1)\)-private

- **Soundness:**

  \[
P(\text{malicious prover convinces the verifier}) = P(\text{corrupted party remains hidden}) = \frac{1}{N}
  \]

- **Parallel repetition**

  Protocol repeated \(\tau\) times in parallel \(\rightarrow\) soundness error \(\left(\frac{1}{N}\right)^\tau\)
From MPC-in-the-Head to signatures
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \( \llbracket x \rrbracket \)
Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

\( x \)

Hash function

signature

Zero-knowledge proof

Prover

\( x \)

OK you know \( x \)

Verifier

\( y \)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\]

Joint evaluation of:

\[ g(x) = \begin{cases} 
  \text{Accept} & \text{if } F(x) = y \\
  \text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Zero-knowledge proof

MPC-in-the Head transform

Signature scheme

\[ x \xrightarrow{\text{Hash function}} \text{signature} \xrightarrow{\text{msg}} y \]
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \( [[x]] \)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

\[ x \]

Hash function

\[ \text{msg} \]

signature

Zero-knowledge proof

\[ x \]

Prover

\[ y \]

Verifier

\[ \text{OK you know } x \]
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding
Invention of the MPC-in-the-Head framework

Signature size (in kilobytes)

SPHINCS+

LowMC

AES

Rain

Signature size (in kilobytes)

Logarithmic scale

**Syndrome Decoding Problem:**
From a matrix $H$ and a vector $y$, find $x$ such that
- $y = Hx$,
- $x$ has at most $w$ non-zero coordinates.
Syndrome Decoding Problem:
From a matrix $H$ and a vector $y$, find $x$ such that
- $y = Hx$,
- $x$ has at most $w$ non-zero coordinates.
**Syndrome Decoding Problem:**
From a matrix \( H \) and a vector \( y \), find \( x \) such that

- \( y = Hx \),
- \( x \) has at most \( w \) non-zero coordinates.
**Syndrome Decoding Problem:**
From a matrix $H$ and a vector $y$, find $x$ such that
- $y = Hx$,
- $x$ has at most $w$ non-zero coordinates.
Exploring other assumptions

- **Subset Sum Problem**: $\geq 100$ KB $\Rightarrow 19.1$ KB [FMRV22,Fen23]

- **Multivariate Quadratic Problem**: $6.3 - 7.3$ KB [Fen22,BFR23]

- **MinRank Problem**: $\approx 5 - 6$ KB [ARV22,Fen22,ABB+23]

- **Rank Syndrome Decoding Problem**: $\approx 5 - 6$ KB [Fen22]

- **Permutated Kernel Problem (or variant)**: $\approx 6$ KB [BG22,BBD+24]

- ...

*Remark*: the displayed signature sizes correspond to the state-of-the-art for the NIST submission deadline of the call for additional post-quantum signatures, better sizes can be achieved using newer results.
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Three approaches:

- **Rely on standard symmetric primitives**
  - AES: BBQ (2019), Banquet (2021), Limbo-Sign (2021), Helium+AES (2022), FAEST (2023)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Three approaches:

- Rely on **standard symmetric primitives**
- Rely on **MPC-friendly symmetric primitives**
  - Rain: Rainier (2021), BN++Rain (2022)
  - AIM: AIMer (2022)
Three approaches:

- Rely on **standard symmetric primitives**
- Rely on **MPC-friendly symmetric primitives**
- Rely on **well-known hard problems** (*non-exhaustive list*)
  - Syndrome Decoding: *SDitH* (2022), *RYDE* (2023)
  - MinRank: *MiRitH* (2022), *MIRA* (2023)
  - Multivariate Quadratic: *MQOM* (2023), *Biscuit* (2023)
  - Permutted Kernel: *PERK* (2023)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Three approaches:

- Rely on standard symmetric primitives
- Rely on MPC-friendly symmetric primitives
- Rely on well-known hard problems
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Expressed as an arithmetic circuit, enabling us to use existing MPCitH-based proof systems (as BN++)

Three approaches:

- Rely on **standard symmetric primitives**
- Rely on **MPC-friendly symmetric primitives**
- Rely on **well-known hard problems**
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system,
Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\]
Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Three approaches:

- Rely on standard symmetric primitives
- Rely on MPC-friendly symmetric primitives
- Rely on well-known hard problems

Expressed as an arithmetic circuit, enabling us to use existing MPCitH-based proof systems (as BN++)

Should be rephrased to achieve interesting performances
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\]
Joint evaluation of:

\[ g(x) = \begin{cases} 
    \text{Accept} & \text{if } F(x) = y \\
    \text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Expressed as an arithmetic circuit, enabling us to use existing MPCitH-based proof systems (as BN++)

Three approaches:
- Rely on standard symmetric primitives
- Rely on MPC-friendly symmetric primitives
- Rely on well-known hard problems

Should be rephrased to achieve interesting performances

Example (RYDE): how to check that a vector \( x \in \mathbb{F}_{q^m}^n \) has a rank weight smaller than some public bound \( r \)?

By checking that \( x_1, \ldots, x_n \) are roots of a degree-\( q^r \) \( q \)-polynomial \( \sum_{i=0}^{r} a_i X^{q^i} \).
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\)

Joint evaluation of:

\[ g(x) = \begin{cases} 
  \text{Accept} & \text{if } F(x) = y \\
  \text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

Hash function

Prover

Verifier

OK you know \(x\)
One-way function

\[ F : x \mapsto y \]

E.g. AES, MQ system, Syndrome decoding

Multiparty computation (MPC)

Input sharing \([x]\)

Joint evaluation of:

\[ g(x) = \begin{cases} 
\text{Accept} & \text{if } F(x) = y \\
\text{Reject} & \text{if } F(x) \neq y 
\end{cases} \]

Signature scheme

Prover

Verifier

Zero-knowledge proof

Fiat-Shamir transform

Should take [KZ20] attack into account (when there are more than 3 rounds)!

[KZ20] Kales, Zaverucha. “An attack on some signature schemes constructed from five-pass identification schemes” (CANS20)
### MPCitH-based NIST Candidates

<table>
<thead>
<tr>
<th>Assumption</th>
<th>Size (in KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMer</td>
<td>3.8-5.9</td>
</tr>
<tr>
<td>Biscuit</td>
<td>4.8-6.7</td>
</tr>
<tr>
<td>FAEST*</td>
<td>4.6-6.3</td>
</tr>
<tr>
<td>MIRA</td>
<td>5.6-7.4</td>
</tr>
<tr>
<td>MiRitH</td>
<td>5.7-9.1</td>
</tr>
<tr>
<td>PERK</td>
<td>6.8-8.4</td>
</tr>
<tr>
<td>MQOM</td>
<td>6.3-7.8</td>
</tr>
<tr>
<td>RYDE</td>
<td>6.0-7.4</td>
</tr>
<tr>
<td>SDitH</td>
<td>8.3-10.4</td>
</tr>
</tbody>
</table>

* FAEST has not been formally introduced as an MPCitH-based scheme.
Optimisations and variants
Optimisations and variants

With SDitH-L1-gf251 as example.

Field $GF(251)$

NIST Category I
① Generate and commit shares
\[ [[x]] = ([[x]_1, ..., [x]]_N) \]

② Run MPC in their head

Com^\rho_1([[x]]_1)
\vdots
Com^\rho_N([[x]]_N)

\text{send broadcast}
\[[\alpha]_1, ..., [\alpha]_N\]

③ Choose a random party
\[ i^* \leftarrow \{1, ..., N\} \]

⑤ Check \( \forall i \neq i^* \)
- Commitments \( \text{Com}^\rho([[x]]_i) \)
- MPC computation \( [[\alpha]]_i = \varphi([[x]]_i) \)
Check \( \tilde{g}(y, \alpha) = \text{Accept} \)
Naive MPCitH transformation

Size \approx \tau \cdot \left(N \cdot 2\lambda + N \cdot |\alpha| + (N - 1) \cdot |x|\right)

\tau \approx \frac{\lambda}{\log_2 N}

- Size of the broadcast (per party)
- Size of the MPC input (per party)
- Number of repetitions to achieve the desired security level
- Size of a commitment digest
Naive MPCitH transformation

$\text{Size} \approx \tau \cdot \left( N \cdot 2\lambda + N \cdot |\alpha| + (N - 1) \cdot |x| \right)$

$\tau \approx \frac{\lambda}{\log_2 N}$

SDitH-L1-gf251:  
the input $x$ of the MPC protocol is around 323 bytes,  
The broadcast value $\alpha$ of the MPC protocol is around 36 bytes.
Naive MPCitH transformation

SDitH-L1-gf251:
- the input $x$ of the MPC protocol is around 323 bytes,
- The broadcast value $\alpha$ of the MPC protocol is around 36 bytes
MPCitH transform

① Generate and commit shares
\[ [x] = ([x]_1, \ldots, [x]_N) \]

② Run MPC in their head

③ Choose a random party
\[ i^* \leftarrow \$ \{1, \ldots, N\} \]

④ Open parties \{1, \ldots, N\} \{i^*\}

4 \begin{array}{c}
\text{Prover} \\
\begin{array}{c}
\text{Com}^{\rho_1}(\lbrack x \rbrack_1) \\
\vdots \\
\text{Com}^{\rho_N}(\lbrack x \rbrack_N) \\
\end{array} \\
\text{send broadcast} \\
\begin{array}{c}
\lbrack \alpha \rbrack_1, \ldots, \lbrack \alpha \rbrack_N \\
i^* \\
\end{array} \\
\end{array}

⑤ Check \forall i \neq i^*
\begin{array}{c}
- \text{Commitments } \text{Com}^{\rho_i}(\lbrack x \rbrack_i) \\
- \text{MPC computation } \lbrack \alpha \rbrack_i = \varphi(\lbrack x \rbrack_i) \\
\end{array}

Check \tilde{g}(y, \alpha) = \text{Accept}

\text{Verifier}
MPCitH transform

1. Generate and commit shares
   \([x] = ([x]_1, \ldots, [x]_N)\)
   Compute
   \(\forall i, \text{com}_i = \text{Com}^{\rho_i}([x]_i)\)

2. Run MPC in their head

3. Choose a random party
   \(i^* \leftarrow \{1, \ldots, N\}\)

4. Open parties \(\{1, \ldots, N\} \setminus \{i^*\}\)

5. Compute \(\forall i \neq i^*\)
   - Commitments \(\text{Com}^{\rho_i}([x]_i)\)
   - MPC computation \([\alpha]_i = \varphi([x]_i)\)
   Check \(\tilde{g}(y, \alpha) = \text{Accept}\)
   Check \(h_1 = \text{Hash}(\text{com}_1, \ldots, \text{com}_N)\)
   Check \(h_2 = \text{Hash}([\alpha]_1, \ldots, [\alpha]_N)\)

Prover

Verifier
MPCitH transform

① Generate and commit shares
\[
\llbracket x \rrbracket = (\llbracket x \rrbracket_1, \ldots, \llbracket x \rrbracket_N)
\]
Compute
\[
\forall i, com_i = \text{Com}^{\rho_i}(\llbracket x \rrbracket_i)
\]

② Run MPC in their head

③ Choose a random party
\[
i^* \leftarrow \{1, \ldots, N\}
\]

④ Open parties \{1, \ldots, N\}\{i^*\}

⑤ Compute \forall i \neq i^*
- Commitments \text{Com}^{\rho_i}(\llbracket x \rrbracket_i)
- MPC computation \llbracket x \rrbracket_i = \varphi(\llbracket x \rrbracket_i)
Check \tilde{g}(y, \alpha) = \text{Accept}
Check \ h_1 = \text{Hash}(com_1, \ldots, com_N)
Check \ h_2 = \text{Hash}(\llbracket x \rrbracket_1, \ldots, \llbracket x \rrbracket_N)

Prover

Verifier
Using a Seed Tree


\[ x = \llbracket x \rrbracket_1 + \llbracket x \rrbracket_2 + \llbracket x \rrbracket_3 + \ldots + \llbracket x \rrbracket_{N-1} + \llbracket x \rrbracket_N \]
Using a Seed Tree


\[ x = \llbracket x \rrbracket_1 + \llbracket x \rrbracket_2 + \llbracket x \rrbracket_3 + \ldots + \llbracket x \rrbracket_{N-1} + \llbracket x \rrbracket_N \]
Using a Seed Tree


\[ x = \frac{[x]}{1} + \frac{[x]}{2} + \frac{[x]}{3} + \ldots + \frac{[x]}{N-1} + \frac{[x]}{N} \]
Using a Seed Tree


\[
x = [x]_1 + [x]_2 + [x]_3 + \ldots + [x]_{N-1} + [x]_N
\]
Using a Seed Tree


\[ x = \=[[x]_1 + \quiring{x}_2 + \quiring{x}_3 + \ldots + \quiring{x}_{N-1} + \quiring{x}_N \]
Using a Seed Tree

Sibling path

→ log(N) seeds

\[ x = \lfloor x \rfloor_1 + \lfloor x \rfloor_2 + \lfloor x \rfloor_3 + \cdots + \lfloor x \rfloor_{N-1} + \lfloor x \rfloor_N \]

Traditional MPCitH transformation

\[ \text{Size} \approx \tau \cdot (|\Delta x| + |\alpha| + \lambda \cdot \log_2 N + 2\lambda) \]

- Size of the broadcast (of the hidden party)
- Size of the auxiliary value
- Path in the seed (GGM) tree
- Number of repetitions to achieve the desired security level
- Commitment of the hidden party

\[ \tau \approx \frac{\lambda}{\log_2 N} \]
Traditional MPCitH transformation

SDitH-L1-gf251:
the input $x$ of the MPC protocol is around 323 bytes,
The broadcast value $\alpha$ of the MPC protocol is around 36 bytes.
Traditional MPCitH transformation

**Signing algorithm**

- Proof Size (in kB)
- Number $N$ of parties
- Running times @3.80Ghz

**Verification algorithm**

- Proof Size (in kB)
- Verification time (in ms)
- Number $N$ of parties
Traditional MPCitH transformation

**Signing algorithm**

**Verification algorithm**

Running times @3.80Ghz
Traditional MPCitH transformation

**Signing algorithm**

Running times @3.80Ghz

- **Symmetric**
- **MPC Emulation**
- **Misc**

Signing time for $N := 256$ parties (19 ms)
The Hypercube Technique

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

*Traditional*: one sharing of $x$

$$x = r_1 + r_2 + \ldots + r_N + \Delta x$$
The Hypercube Technique

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

**Traditional:** one sharing of $x$

$$x = r_1 + r_2 + \ldots + r_N + \Delta x$$

**Hypercube:** $D$ sharings of $x$, with the same auxiliary value $\Delta x$

$$x = \begin{cases} 
  r_{1,1} + r_{1,2} + \ldots + r_{1,N_1} \\
  r_{2,1} + r_{2,2} + \ldots + r_{2,N_2} \\
  \ldots \\
  r_{D,1} + r_{D,2} + \ldots + r_{D,N_D} 
\end{cases} + \Delta x$$

such that $N = N_1 \cdot N_2 \cdot \ldots \cdot N_D$
The **Hypercube Technique**

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

\[
x = \left\{ r_{1,1} + r_{1,2} + \ldots + r_{1,N_1} \right\} + \Delta x
\[
\left\{ r_{2,1} + r_{2,2} + \ldots + r_{2,N_2} \right\}
\[
\left. \ldots \right\}
\[
\left\{ r_{D,1} + r_{D,2} + \ldots + r_{D,N_D} \right\}
\]

\[
N = N_1 \cdot N_2 \cdot \ldots \cdot N_D
\]

**How to build these** \(D\) **sharings?**
The Hypercube Technique

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

\[
x = \left\{ r_{1,1} + r_{1,2} + \ldots + r_{1,N_1} \\
r_{2,1} + r_{2,2} + \ldots + r_{2,N_2} \\
\ldots \\
r_{D,1} + r_{D,2} + \ldots + r_{D,N_D} \right\} + \Delta x
\]

\[ N = N_1 \cdot N_2 \cdot \ldots \cdot N_D \]

How to build these \( D \) sharings?
The Hypercube Technique

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: "The Return of the SDitH" (Eurocrypt 2023)

\[ x = \left\{ \begin{array}{l}
r_{1,1} + r_{1,2} + \ldots + r_{1,N_1} \\
r_{2,1} + r_{2,2} + \ldots + r_{2,N_2} \\
\vdots \\
r_{D,1} + r_{D,2} + \ldots + r_{D,N_D} \\
\end{array} \right\} + \Delta x \\
N = N_1 \cdot N_2 \cdot \ldots \cdot N_D \\
\]

How to build these $D$ sharings?

For $D = 2$
The Hypercube Technique

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

\[
x = \left\{ r_{1,1} + r_{1,2} + \ldots + r_{1,N_1} \right. \\
\left. r_{2,1} + r_{2,2} + \ldots + r_{2,N_2} \right. \\
\ldots \\
\left. r_{D,1} + r_{D,2} + \ldots + r_{D,N_D} \right\} + \Delta x
\]

\[N = N_1 \cdot N_2 \cdot \ldots \cdot N_D\]

How to build these \( D \) sharings?

For \( D = 2 \)
The **Hypercube Technique**

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

\[
N = N_1 \cdot N_2 \cdot \ldots \cdot N_D
\]

\[
x = \left\{ r_{1,1} + r_{1,2} + \ldots + r_{1,N_1} \\
r_{2,1} + r_{2,2} + \ldots + r_{2,N_2} \\
\vdots \\
r_{D,1} + r_{D,2} + \ldots + r_{D,N_D} \right\} + \Delta x
\]

**How to build these \( D \) sharings?**

*For \( D \geq 2 \)*

Source: Figure from [AGHHJY23]
The Hypercube Technique

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

\[ x = \left\{ \begin{array}{c} r_{1,1} + r_{1,2} + \ldots + r_{1,N_1} \\ r_{2,1} + r_{2,2} + \ldots + r_{2,N_2} \\ \vdots \\ r_{D,1} + r_{D,2} + \ldots + r_{D,N_D} \end{array} \right\} + \Delta x \]

\[ N = N_1 \cdot N_2 \cdot \ldots \cdot N_D \]

**Performance**

- Same soundness error as before: \(1/N\)
- Same signature size as before: 1 auxiliary value + 1 seed tree of \(N\) leaves
The Hypercube Technique

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

Performance

- Same soundness error as before: $1/N$
- Same signature size as before: 1 auxiliary value + 1 seed tree of $N$ leaves
- Emulation cost: one needs to emulate

$$D = \log_2 N$$
$$N_1 = \ldots = N_D = 2$$

$$1 + \log_2 N$$

instead of $N = N_1 \cdot N_2 \cdot \ldots \cdot N_D$
The Hypercube Technique

[AGHHJY23] Aguilar-Melchor, Gama, Howe, Hülsing, Joseph, Yue: “The Return of the SDitH” (Eurocrypt 2023)

Traditional: \(N\) party emulations per repetition

\[
D = \log_2 N \\
N_1 = \ldots = N_D = 2
\]

Hypercube: \(1 + \log_2 N\) party emulations per repetition

\[
1 + \log_2 N = 9
\]
The Hypercube Technique

Signing algorithm

Verification algorithm

Running times @3.80Ghz
The Hypercube Technique

**Signing algorithm**

**Verification algorithm**

- **Symmetric**
- **Packing**
- **MPC Emulation**
- **Misc**

Running times @3.80Ghz
The Hypercube Technique

**Signing algorithm**

- **Proof Size (in kB)**
  - 256 parties

- **Signing time (in ms)**
  - Running times @3.80Ghz

**Graph:**
- Number $N$ of parties
- Proof size and signing time for $N = 256$ parties

**Pie Chart:**
- Symmetric: 69%
- Packing: 13%
- MPC Emulation: 12%
- Misc: 6%

**Summary:**
- Signing time for $N := 256$ parties (7 ms)

Running times @3.80Ghz
The (original) Threshold Approach


In the *threshold* approach, we used a **low-threshold** sharing scheme. For example, Shamir’s \((\ell + 1,N)\)-secret sharing scheme.

To share a value \(x\),

- sample \(r_1, r_2, \ldots, r_\ell\) uniformly at random,
- build the polynomial \(P(X) = x + \sum_{k=0}^{\ell} r_k \cdot X^k\),
- Set the share \([[x]]_i \leftarrow P(e_i)\), where \(e_i\) is publicly known.
In the threshold approach, we used a low-threshold sharing scheme. For example, Shamir’s \((\ell + 1, N)\)-secret sharing scheme.

The prover reveals only \(\ell\) shares to the verifier (instead of \(N - 1\)).

\textit{In practice}, \(\ell \in \{1, 2, 3\}\).
In the threshold approach, we used a low-threshold sharing scheme. For example, Shamir’s \((\ell + 1, N)\)-secret sharing scheme.

The prover reveals only \(\ell\) shares to the verifier (instead of \(N - 1\)).

In practice, \(\ell \in \{1, 2, 3\}\).

Construction:

- The verifier just needs to re-emulate \(\ell\) parties (per repetition);
The (original) **Threshold Approach**


In the *threshold* approach, we used a low-threshold sharing scheme. For example, Shamir’s \((\ell + 1, N)\)-secret sharing scheme.

The prover reveals only \(\ell\) shares to the verifier (instead of \(N - 1\)).

In practice, \(\ell \in \{1, 2, 3\}\).

**Construction:**

- The verifier just needs to re-emulate \(\ell\) parties (per repetition);
- The prover just needs to emulate \(1 + \ell\) parties (per repetition);
The (original) **Threshold Approach**


In the *threshold* approach, we used a **low-threshold** sharing scheme. For example, Shamir’s \((\ell + 1, N)\)-secret sharing scheme.

The prover reveals only \(\ell\) shares to the verifier (instead of \(N - 1\)).

*In practice, \(\ell \in \{1, 2, 3\}\).*

**Construction:**

- The verifier just needs to re-emulate \(\ell\) *parties* (per repetition);
- The prover just needs to emulate \(1 + \ell\) *parties* (per repetition);
- The prover uses a Merkle tree to commit the shares;
In the *threshold* approach, we used a **low-threshold** sharing scheme. For example, Shamir’s $(\ell + 1,N)$-secret sharing scheme.

The prover reveals only $\ell$ shares to the verifier (instead of $N - 1$). In practice, $\ell \in \{1,2,3\}$.

**Construction:**
- The verifier just needs to re-emulate $\ell$ parties (per repetition);
- The prover just needs to emulate $1 + \ell$ parties (per repetition);
- The prover uses a Merkle tree to commit the shares;
- The obtained signature size is larger;
In the \textit{threshold} approach, we used a \textbf{low-threshold} sharing scheme. For example, Shamir’s $(\ell + 1, N)$-secret sharing scheme.

The prover reveals only $\ell$ shares to the verifier (instead of $N - 1$).

\textit{In practice}, $\ell \in \{1, 2, 3\}$.

\textbf{Construction:}

- The verifier just needs to re-emulate $\ell$ \textbf{parties} (per repetition);
- The prover just needs to emulate $1 + \ell$ \textbf{parties} (per repetition);
- The prover uses a Merkle tree to commit the shares;
- The obtained signature size is \textbf{larger};
- We have the constraint: $N \leq |\mathbb{F}|$. 

\textbf{The (original) Threshold Approach}

\textbf{[FR22]} Feneuil, Rivain: “Threshold Linear Secret Sharing to the Rescue of MPC-in-the-Head” (Asiacrypt 2023)
The (original) **Threshold** Approach

**Signing algorithm**

**Verification algorithm**

Running times @3.80Ghz
The (original) **Threshold Approach**

**Signing algorithm**

**Verification algorithm**

Running times @3.80Ghz
The (original) **Threshold Approach**

**Signing algorithm**

Diagram showing the proof size and signing time as a function of the number of parties. For 251 parties:
- **Proof size:** 1.6 ms
- **Signing time:** 69%
- **Share Computing:** 20%
- **MPC Emulation:** 5%
- **Miscellaneous:** 6%

Running times @3.80Ghz

For $N := 251$ parties

(1.6 ms)
The (original) **Threshold Approach**

- Symmetric
- MPC Emulation
- Misc

**Verification time**

for $N := 251$ parties

(0.2 ms)

**Running times @3.80Ghz**

![Pie chart showing verification time for 251 parties](image)

**Verification algorithm**

- Proof Size (in kB)
- Verification time (in ms)

*Graph showing proof size and verification time for different numbers of parties.*

*251 parties*
The existing MPCitH transforms

Traditional

- Shorter signature sizes
- Highly parallelizable
- Slower signing time
- Signing time $\approx$ Verification time
- Computational cost is mainly due to symmetric primitives

Hypercube

Threshold

- Faster signing time
- Highly parallelizable
- Very fast verification
- Larger signature size
- Restriction # of parties
- Computational cost is mainly due to arithmetics
### MPCitH-based NIST candidates

<table>
<thead>
<tr>
<th></th>
<th>Short Instance</th>
<th>Fast Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlMer</td>
<td>Traditional (256-1615)</td>
<td>Traditional (16-57)</td>
</tr>
<tr>
<td>Biscuit</td>
<td>Traditional (256)</td>
<td>Traditional (16)</td>
</tr>
<tr>
<td>MIRA</td>
<td>Hypercube (256)</td>
<td>Hypercube (32)</td>
</tr>
<tr>
<td>MiRitH</td>
<td>Traditional (256)</td>
<td>Traditional (16)</td>
</tr>
<tr>
<td></td>
<td>Hypercube (256)</td>
<td>Hypercube (16)</td>
</tr>
<tr>
<td>MQOM</td>
<td>Hypercube (256)</td>
<td>Hypercube (32)</td>
</tr>
<tr>
<td>RYDE</td>
<td>Hypercube (256)</td>
<td>Hypercube (32)</td>
</tr>
<tr>
<td>SDitH</td>
<td>Hypercube (256)</td>
<td>Threshold (251-256)</td>
</tr>
</tbody>
</table>

FAEST and PERK rely on other MPCitH techniques.
Conclusion
Advantages and limitations

- **Limitations**
  - Relatively *slow* (*few milliseconds*)
    - Greedy use of symmetric cryptography
  - Relatively *large* signatures (*3-10 KB for L1*)
  - Signature size: *quadratic* growth in the security level
Advantages and limitations

- **Limitations**
  - Relatively *slow* *(few milliseconds)*
    - Greedy use of symmetric cryptography
  - Relatively *large* signatures *(3-10 KB for L1)*
  - Signature size: *quadratic* growth in the security level

- **Advantages**
  - *Conservative* hardness assumption:
    - No structure (often), no trapdoor
  - *Small* (public) keys
  - *Good* public key + signature size
  - Adaptive and *tunable* parameters
Conclusion

- **MPC-in-the-Head**
  - Very versatile and tunable
  - Can be applied on any one-way function
  - A practical tool to build *conservative* signature schemes
Conclusion

- **MPC-in-the-Head**
  - Very versatile and tunable
  - Can be applied on any one-way function
  - A practical tool to build *conservative* signature schemes

- **Recent MPCitH techniques**

  - **VOLE-in-the-Head**
    - Vector-Oblivious-Linear-Evaluation-in-the-Head
    - presented by *Carsten Baum*
    - June 18, 2024

  - **TC-in-the-Head**
    - Threshold-Computation-in-the-Head
    - presented by *Matthieu Rivain*
    - July 2, 2024
Conclusion

- **MPC-in-the-Head**
  - Very versatile and tunable
  - Can be applied on any one-way function
  - A practical tool to build conservative signature schemes

- **Recent MPCitH techniques**
  - **VOLE-in-the-Head**
    - Vector-Oblivious-Linear-Evaluation-in-the-Head
    - Presented by *Carsten Baum*
    - June 18, 2024
  - **TC-in-the-Head**
    - Threshold-Computation-in-the-Head
    - Presented by *Matthieu Rivain*
    - July 2, 2024

Thank you for your attention.
References


[Fen22] Feneuil. Building MPCitH-based Signatures from MQ, MinRank, and Rank SD. ACNS 2024.


