Compute, but Verify: Efficient Multiparty Computation over Authenticated Inputs

Moumita Dutta
IISc Bangalore

Chaya Ganesh
IISc Bangalore

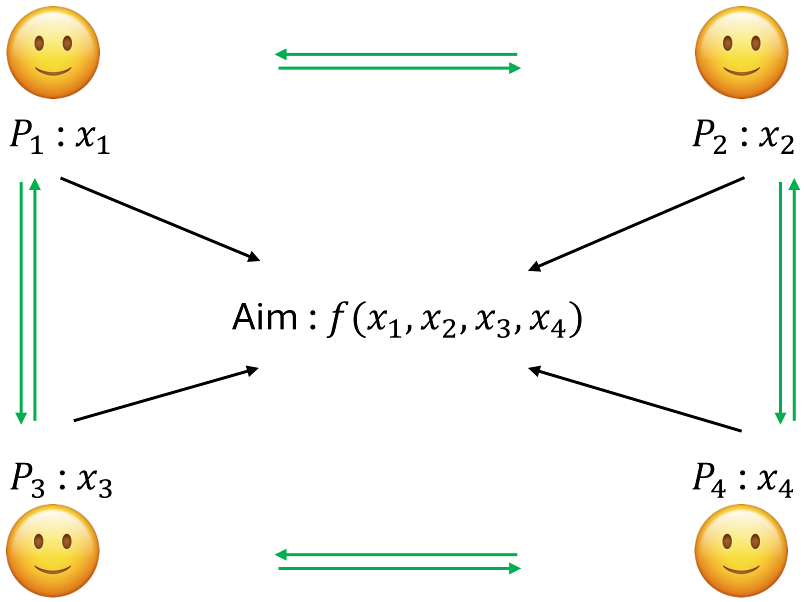
Sikhar Patranabis
IBM Research, India

Nitin Singh
IBM Research, India

Full version: https://ia.cr/2022/1648

Asiacrypt 2024 | December 11, 2024

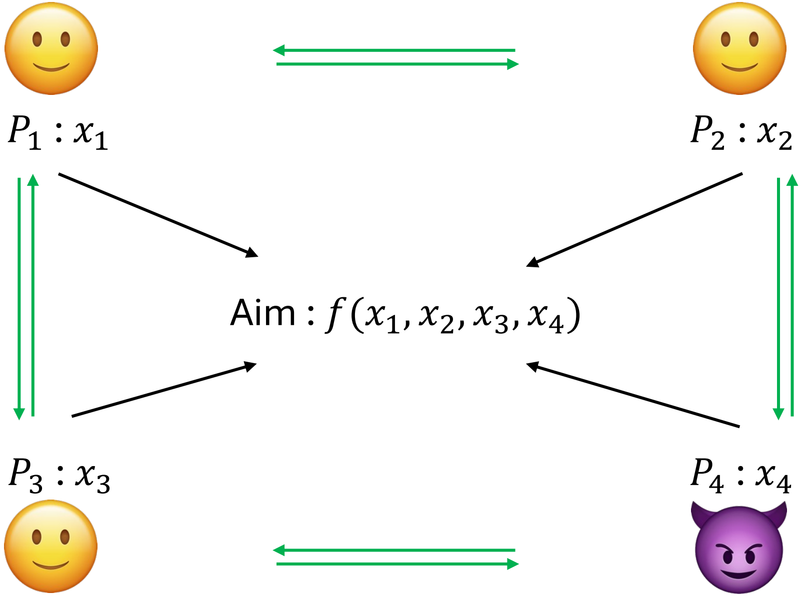
Multi-Party Computation



Goals:

- Privacy of Input
- Correctness of Output

Multi-Party Computation



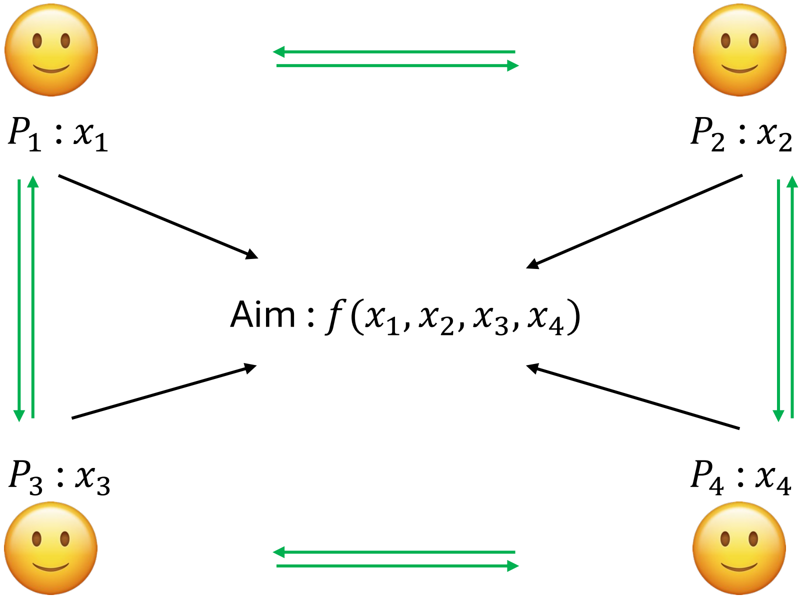
Goals:

- Privacy of Input
- Correctness of Output

Corruption:

- Semi-Honest
- Malicious

Multi-Party Computation Guarantees



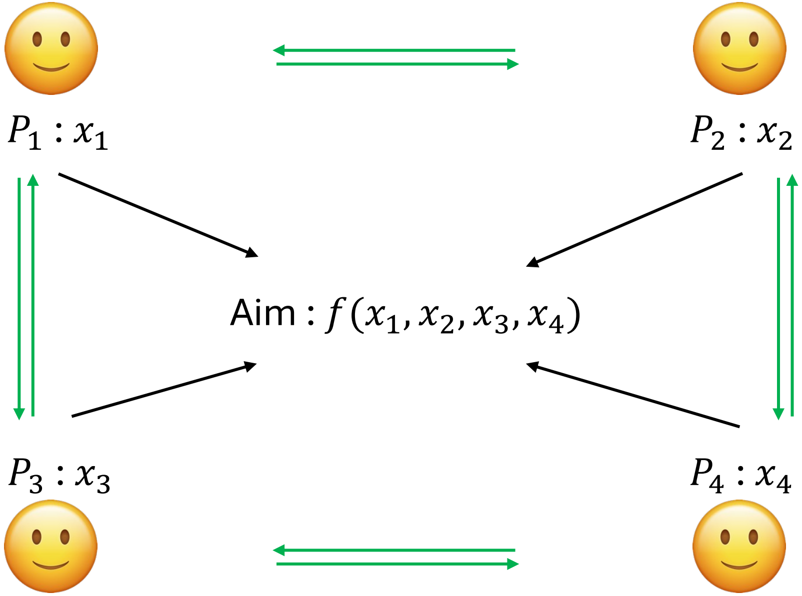
Not traditional MPC guarantee

What if inputs are corrupted?

Data-poisoning attack

- Eg. 2PC AND computation
- Application: Secure aggregation in ML

Multi-Party Computation Guarantees



Not traditional MPC guarantee
What if inputs are corrupted?

Solution
Inputs are authenticated

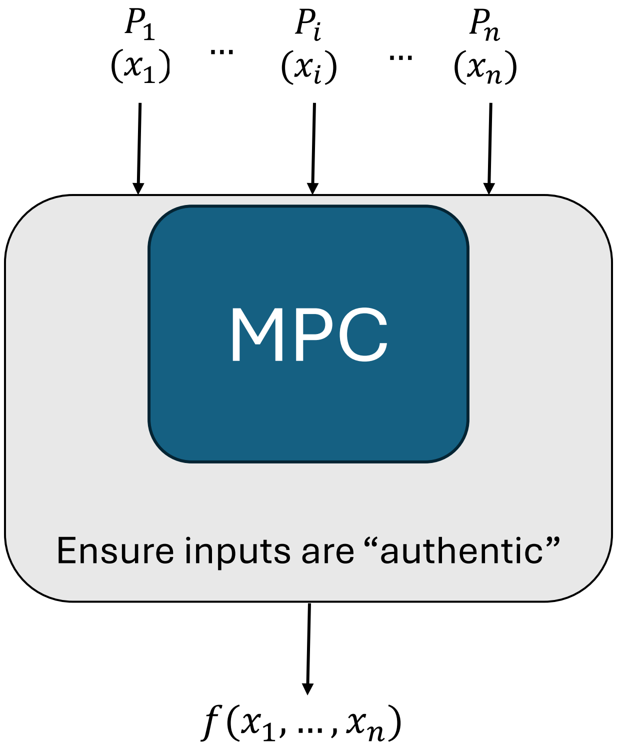
Problem Statement

Goal

Ensure "authenticated inputs" are used inside MPC

Performance Requirement

Negligible communication overhead on existing MPC
(supports existing infrastructure)



How to achieve authentication?

What are authentic inputs?	How to ensure the input provided is authentic?
Inputs signed by certifying authority	Verify: signature corresponds to the private input

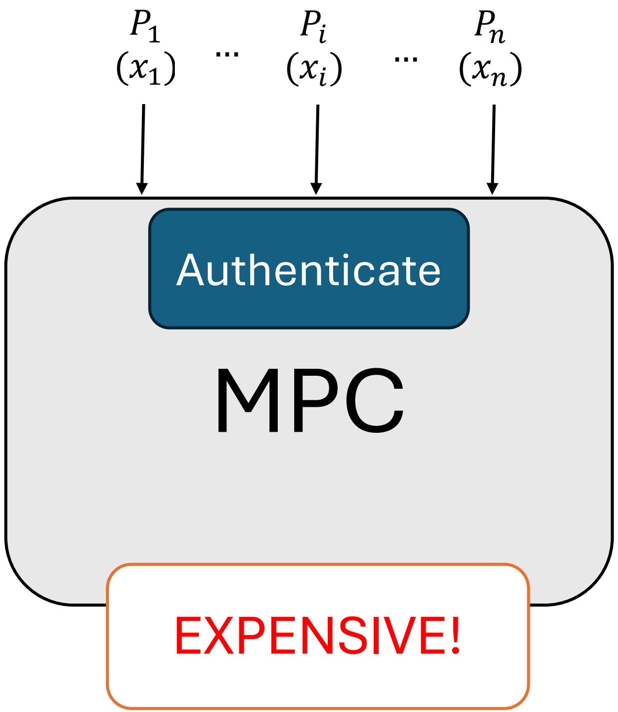
Step1

Authentic inputs signed by certifying authority

Step2

Determine if the input used inside MPC is authentic (consistent with signature)

Attempt: Authenticate inside MPC



Signature verification done inside MPC

Expensive

Requires circuit representation of algebraic signature verification

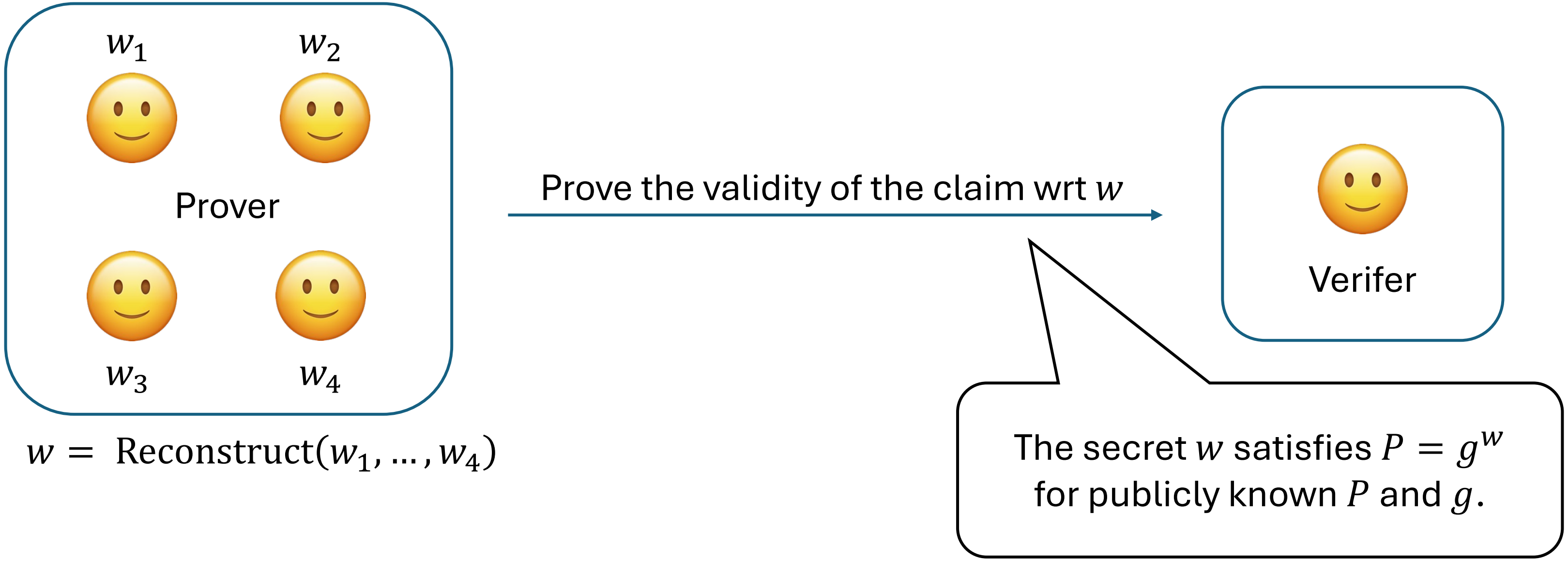
- Authenticate outside MPC
- Challenge: linking of signed message and MPC input

Our Contributions

- Efficient compiler that transforms secret-sharing based honest majority MPC into one with authenticated inputs.
- Building block: Robust Distributed Proof of Knowledge (DPoK) for algebraically structured signature schemes.
  DPoKs are of independent interest as a primitive.
- Communication overhead of $O(n^2 \log \ell)$ to authenticate ℓ-sized input of n parties in secret-sharing based honest majority MPC.

Our Contributions: DPoK

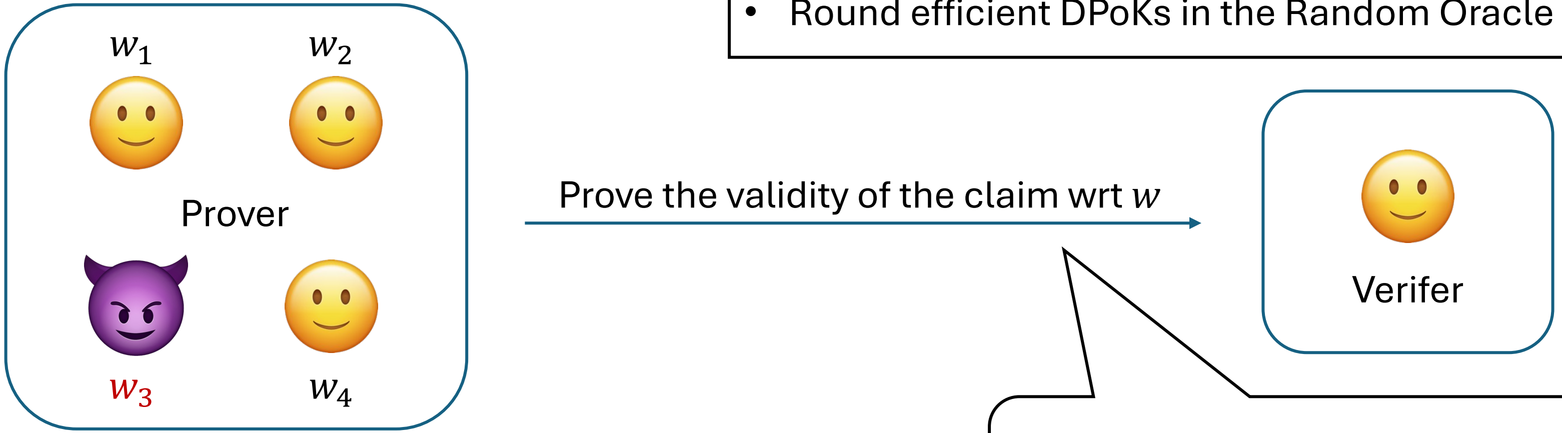
Distributed Proof of Knowledge



Our Contributions: DPoK

Distributed Proof of Knowledge

- Robust DPoK
  (security in presence of dishonest usage of shares)
- Construction of DPoK for Discrete Logarithm.
- DPoKs for algebraic signatures – BBS+ & PS.
- Round efficient DPoKs in the Random Oracle Model.

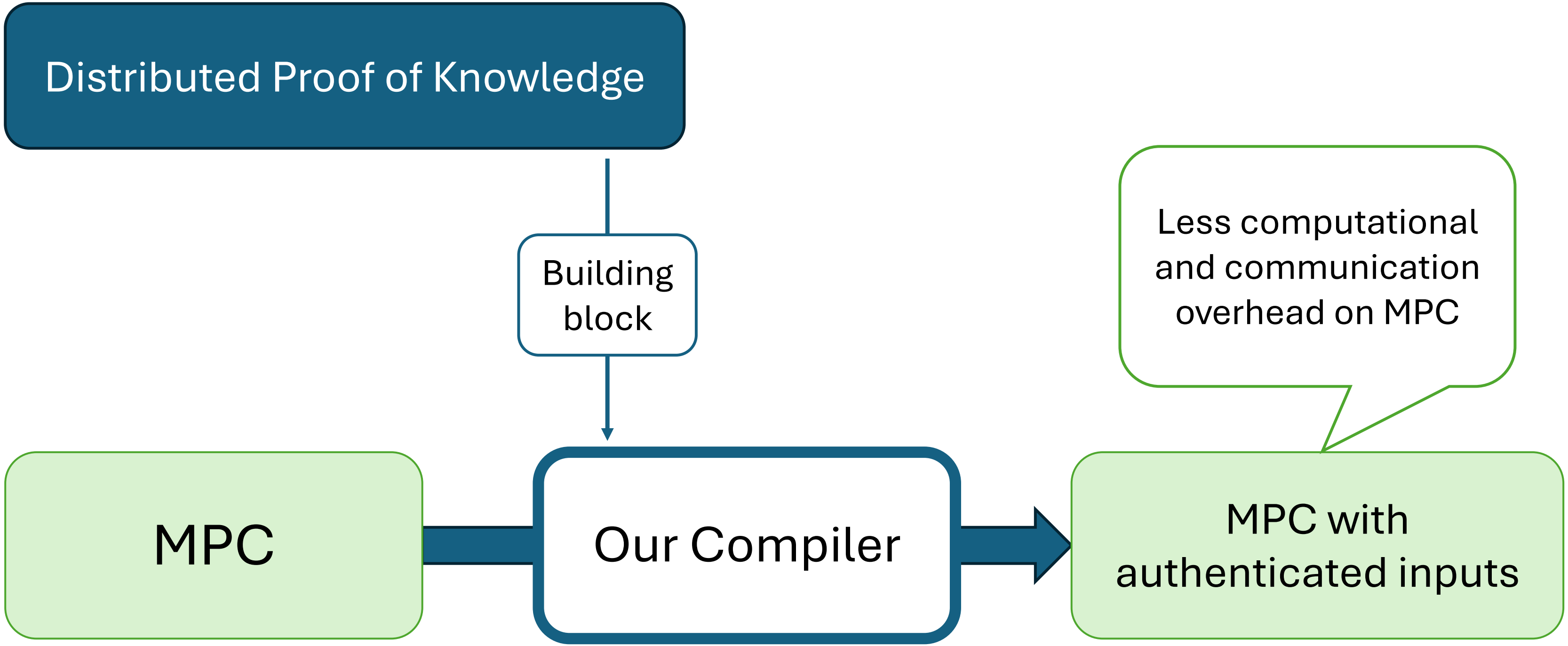


$w = \text{Reconstruct}(w_1, \dots, w_4)$

The secret w satisfies $P = g^w$ for publicly known P and g.

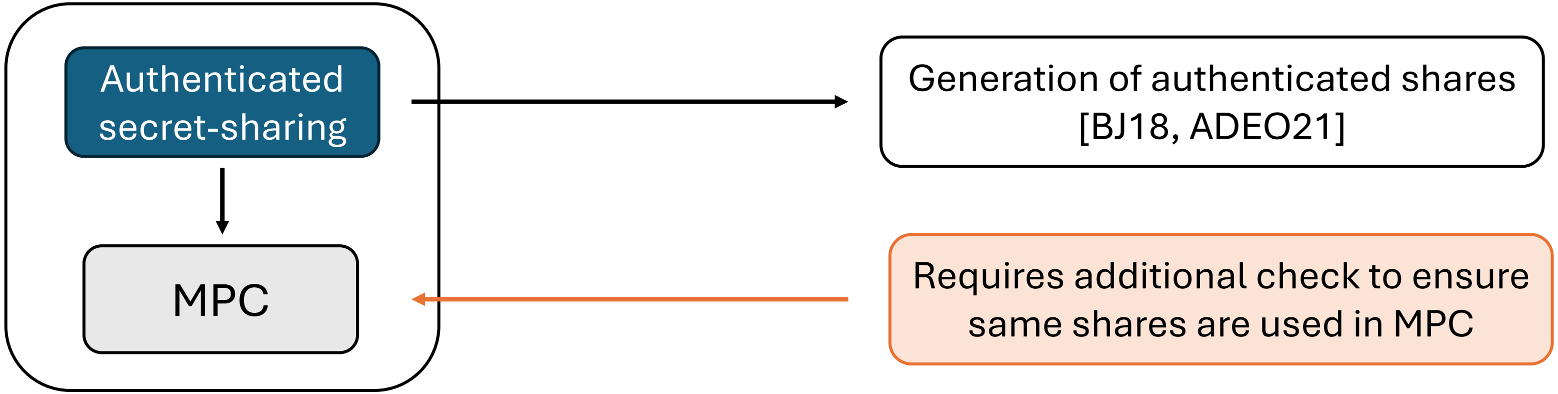
[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. CRYPTO 2004.
[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA.
[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. CT-RSA 2016.

Our Contributions: Our Compiler



Related Work: Input authentication

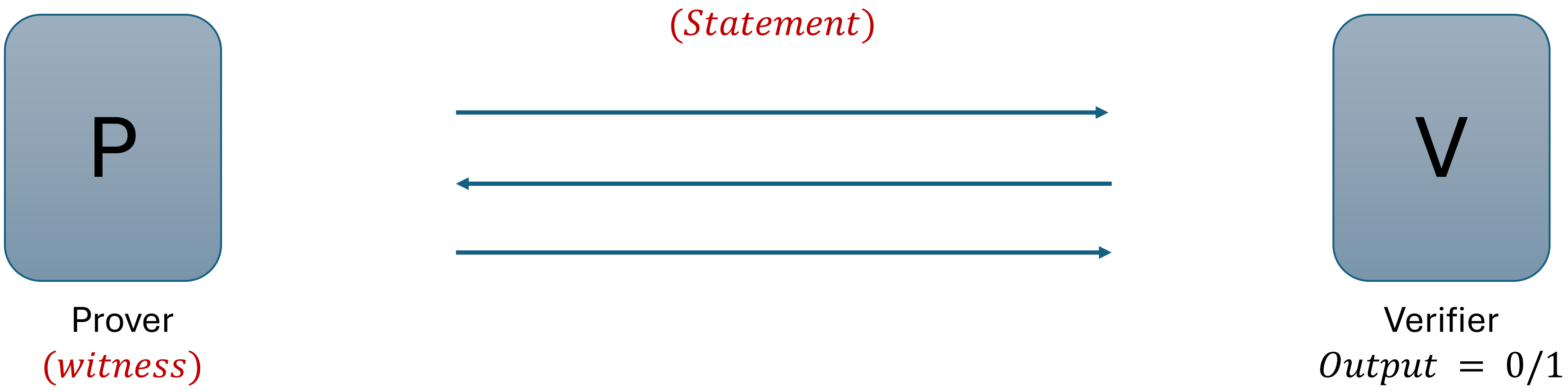
- [Bau16,KMW16,ZBB17]: Input validation for 2PC using Garbled Circuits.
- [BJ18]: Constructs MPC with certified input for MPC [DKL+13,DN07].
- [ADEO21]: Signature verification inside MPC for PS signature scheme, using bilinear pairing over secret-shared data.



Distributed Proofs of Knowledge: Motivation

Classical Proofs

- Completeness [Honest Prover should succeed]
- Soundness [Malicious Prover should fail]
- Zero-Knowledge [Malicious Verifier learns nothing extra]

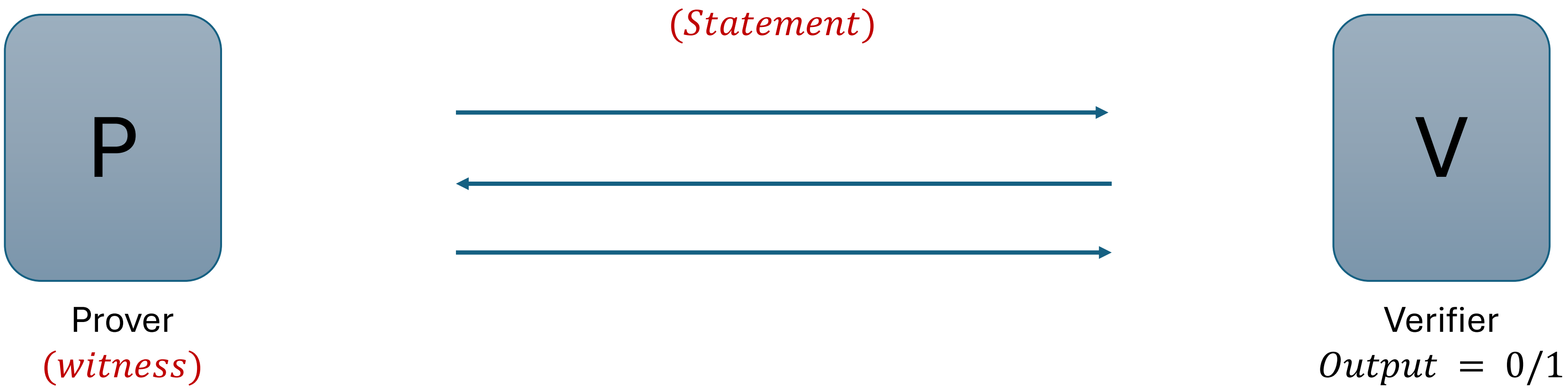


Distributed Proofs of Knowledge: Motivation

Classical Proofs

How to deploy in MPC?

Each party in MPC has to act as prover **to prove its input's authenticity** to every other party.

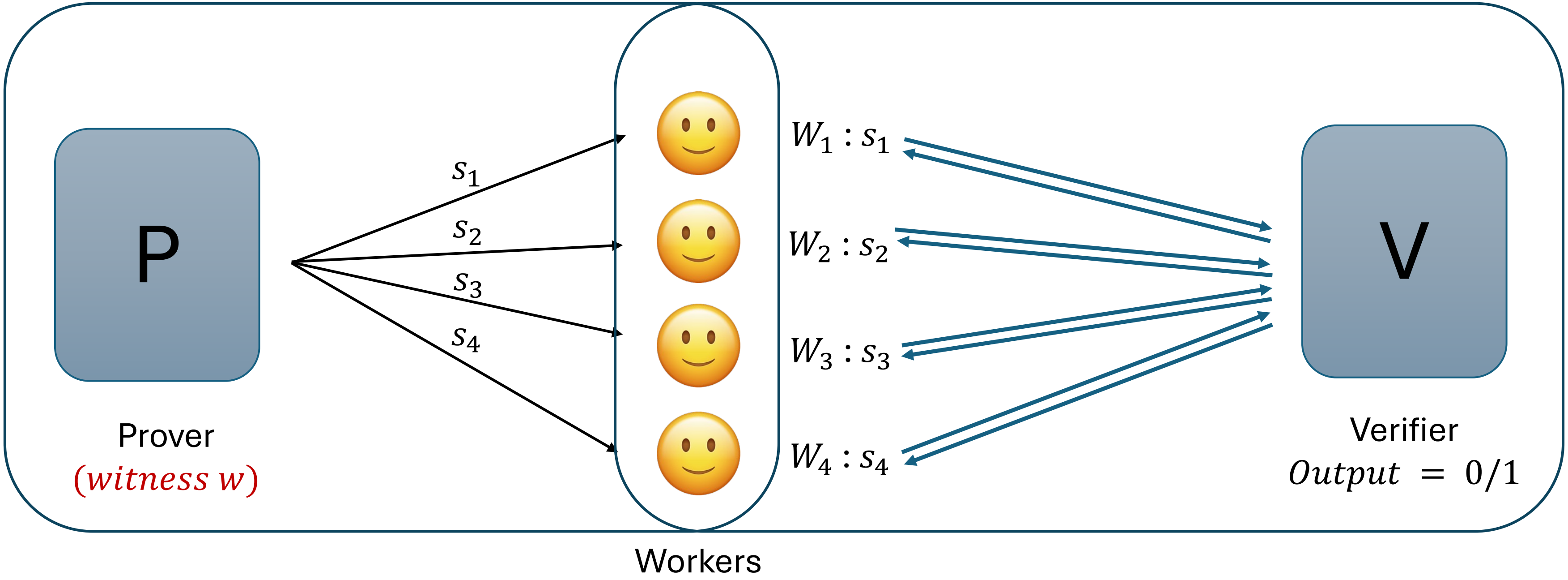


Distributed Proofs of Knowledge

- Proof generated by workers
- Distribute witness amongst workers

Share $(w) \rightarrow (s_1, s_2, s_3, s_4)$

Reconstruct $(s_1, s_2, s_3, s_4) = w$

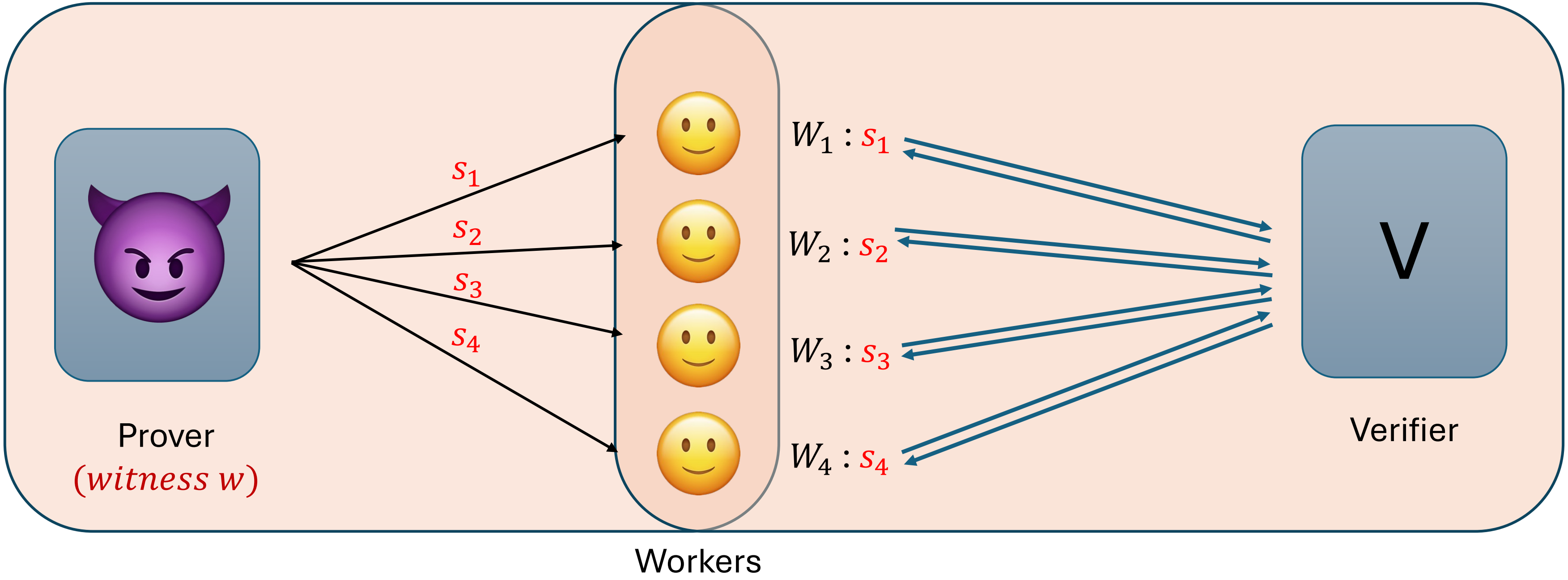


Distributed Proofs of Knowledge: Guarantees

Soundness
[Malicious Prover should fail]

Share $(w) \rightarrow (s_1, s_2, s_3, s_4)$

Reconstruct $(s_1, s_2, s_3, s_4) = w$

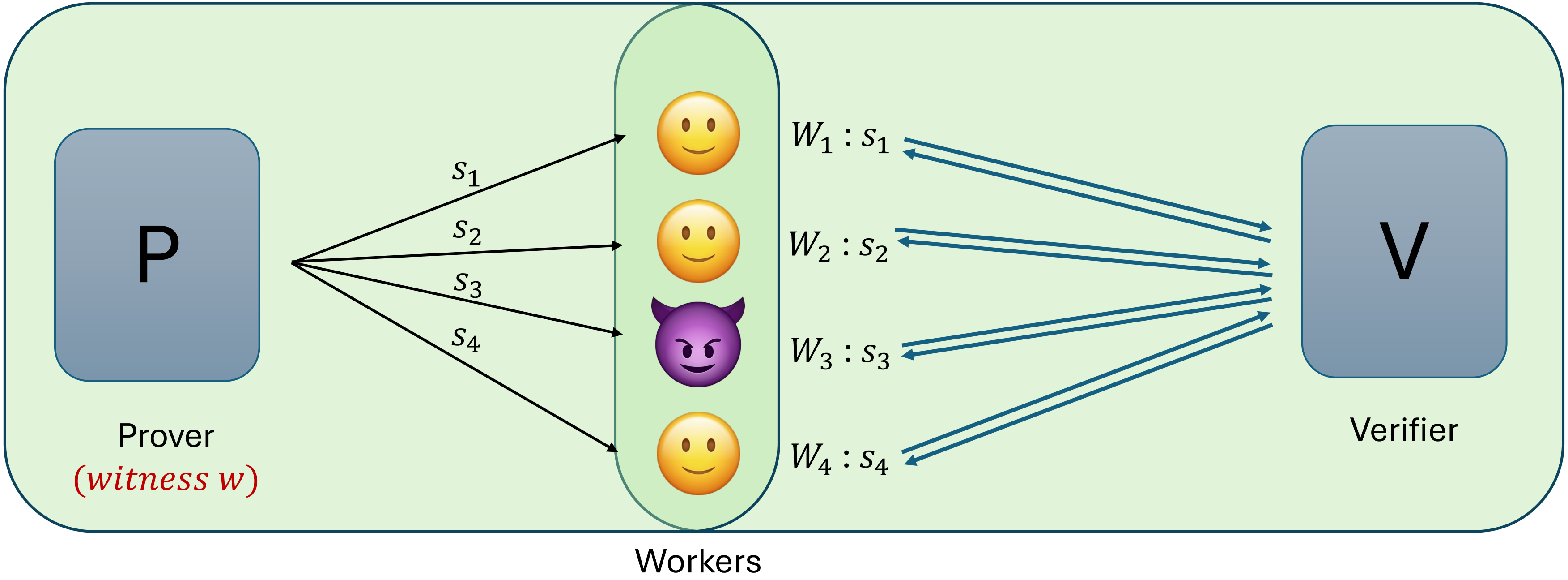


Distributed Proofs of Knowledge: Guarantees

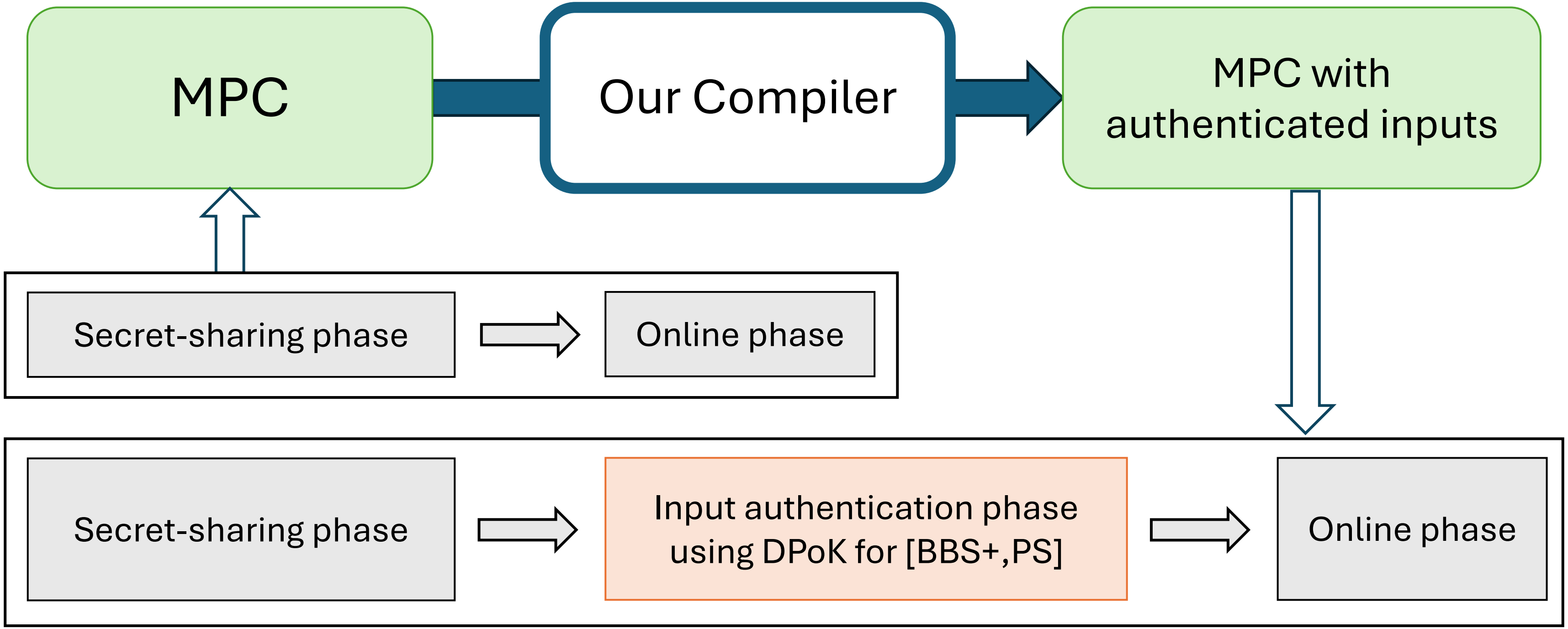
Robust Completeness
[Honest Prover always succeeds]

Share $(w) \rightarrow (s_1, s_2, s_3, s_4)$

Reconstruct $(s_1, s_2, s_3, s_4) = w$



Our Compiler using DPoK



[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. CRYPTO 2004.
[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. CT-RSA 2016.

Input authentication via signatures

Proof of Knowledge of PS Signature

Proof of Knowledge of BBS+ Signature

Proof of Knowledge for Discrete Logarithm Relation

For publicly known P and g, prove knowledge of x such that $P = g^x$

Outline

- Robust DPoK for discrete log relation
- Robust DPoK for PS Signatures

  (Robust DPoK for BBS+ Signatures)
- Overview of our compiler

Outline

- Robust DPoK for discrete log relation

  PoK for discrete log relation

  DPoK for discrete log relation

  Robust DPoK for discrete log relation

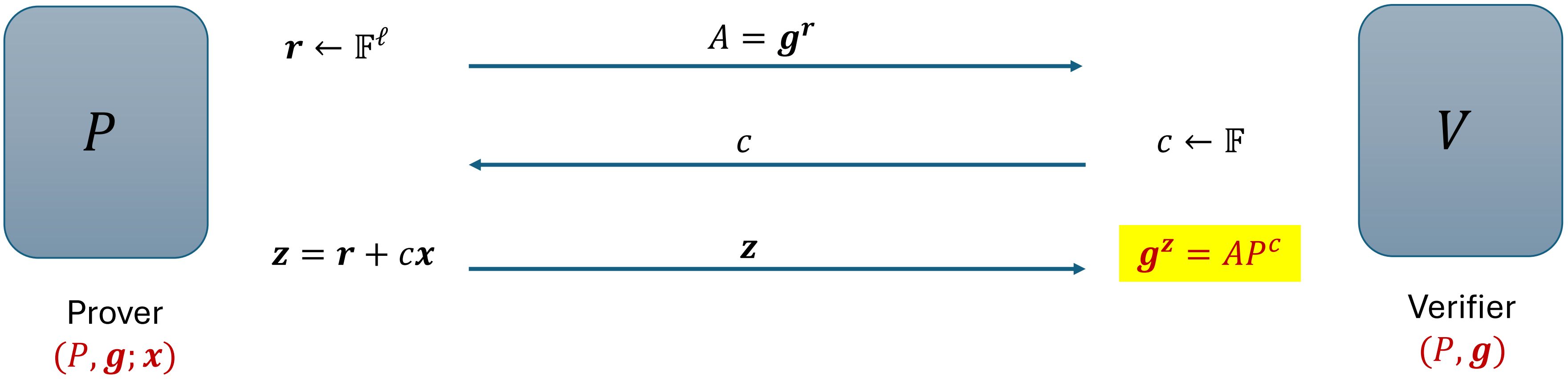
- Robust DPoK for PS Signatures

  (Robust DPoK for BBS+ Signatures)

- Overview of our compiler

Σ-Protocol (PoK) for discrete log relation

- Proof of Knowledge (PoK) of $\boldsymbol{x} \in \mathbb{F}^{\ell}$ such that $P = \boldsymbol{g}^{\boldsymbol{x}} = g_1^{x_1} \cdots g_{\ell}^{x_{\ell}}$
- Sigma Protocol (3 move protocol): PoK for discrete logarithm relation



Check : $\boldsymbol{g}^{\boldsymbol{z}} = \boldsymbol{g}^{\boldsymbol{r}+c\boldsymbol{x}} = \boldsymbol{g}^{\boldsymbol{r}}(\boldsymbol{g}^{\boldsymbol{x}})^{c} = AP^{c}$

Σ-Protocol (PoK) for discrete log relation

- Proof (
- Sigma ... n

Next:
Distributed Σ-Protocol for discrete log relation

- Prover secret-shares its witness amongst workers.
- For simplicity of presentation, we consider additive-secret sharing.

P

Prover
$(P, \boldsymbol{g}; \boldsymbol{x})$

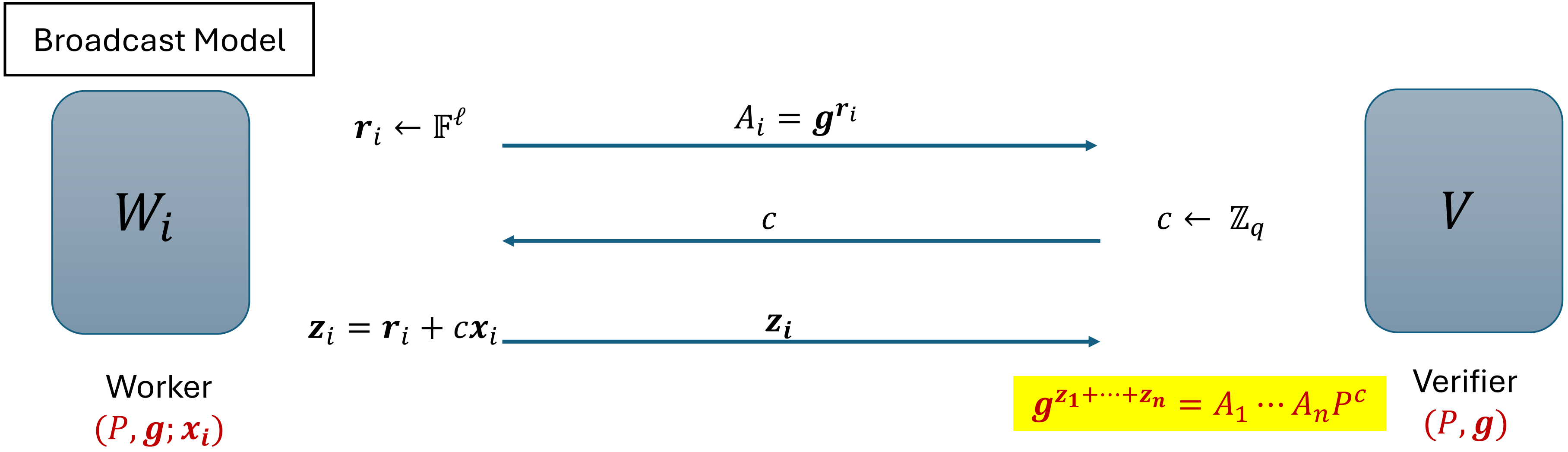
V

Verifier
$(P, \boldsymbol{g})$

Check : $\boldsymbol{g}^{z} = \boldsymbol{g}^{r+cx} = \boldsymbol{g}^{r}(\boldsymbol{g}^{x})^{c} = AP^{c}$

DPoK for discrete log relation

- Proof of Knowledge (PoK) of $\boldsymbol{x}$ such that $P = \boldsymbol{g}^{\boldsymbol{x}}$
- P : computes $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ such that $\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_n = \boldsymbol{x}$
- P : sends $\boldsymbol{x}_i \rightarrow W_i$ (over private channel) such that $P = \boldsymbol{g}^{\boldsymbol{x}} = \boldsymbol{g}^{\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_n}$



Check : $\boldsymbol{g}^{\boldsymbol{z}_1 + \cdots + \boldsymbol{z}_n} = \boldsymbol{g}^{\boldsymbol{r}_1 + c\boldsymbol{x}_1 + \cdots} = \boldsymbol{g}^{\boldsymbol{r}_1} \cdots \boldsymbol{g}^{\boldsymbol{r}_n} (\boldsymbol{g}^{\boldsymbol{x}_1 + \cdots + \boldsymbol{x}_n})^c = A_1 \cdots A_n P^c$

DPoK for discrete log relation

- Proof of Kn
- P : comput
- P : sends x

$^{x_1+\cdots+x_n}$

Check fails even if one worker is corrupt!

Broadcast Model

W_i

$\boldsymbol{r}_i \leftarrow \mathbb{F}^\ell$

$A_i = \boldsymbol{g}^{\boldsymbol{r}_i}$

c

$c \leftarrow \mathbb{Z}_q$

$\boldsymbol{z}_i = \boldsymbol{r}_i + c\boldsymbol{x}_i$

$\boldsymbol{z_i}$

V

Worker
$(P, \boldsymbol{g}; \boldsymbol{x}_i)$

$\boldsymbol{g}^{\boldsymbol{z}_1+\cdots+\boldsymbol{z}_n} = A_1 \cdots A_n P^c$

Verifier
$(P, \boldsymbol{g})$

Check : $\boldsymbol{g}^{z_1+\cdots+z_n} = \boldsymbol{g}^{r_1+cx_1+\cdots} = \boldsymbol{g}^{r_1} \cdots \boldsymbol{g}^{r_n}(\boldsymbol{g}^{x_1+\cdots+x_n})^c = A_1 \cdots A_n P^c$

Robust DPoK

Honest input is authenticated
even if some workers are corrupt

- (t, n) - linear secret sharing to enable error-correction
- Properties of linear codes :
  1. Linear combination of codewords is also a codeword
  2. “Error-preserving” :
     linear combination retains the position of error in a codeword
  3. “Error-preserving” property is provable for corruption $\leq distance/3$

Error-correcting Linear Codes

Codeword	P_1	P_2	...	P_{n-1}	P_n
a	a_1	a_2	...	a_{n-1}	a_n
b	b_1	b_2	...	b_{n-1}	b_n
c	c_1	c_2	...	c_{n-1}	c_n

error is preserved

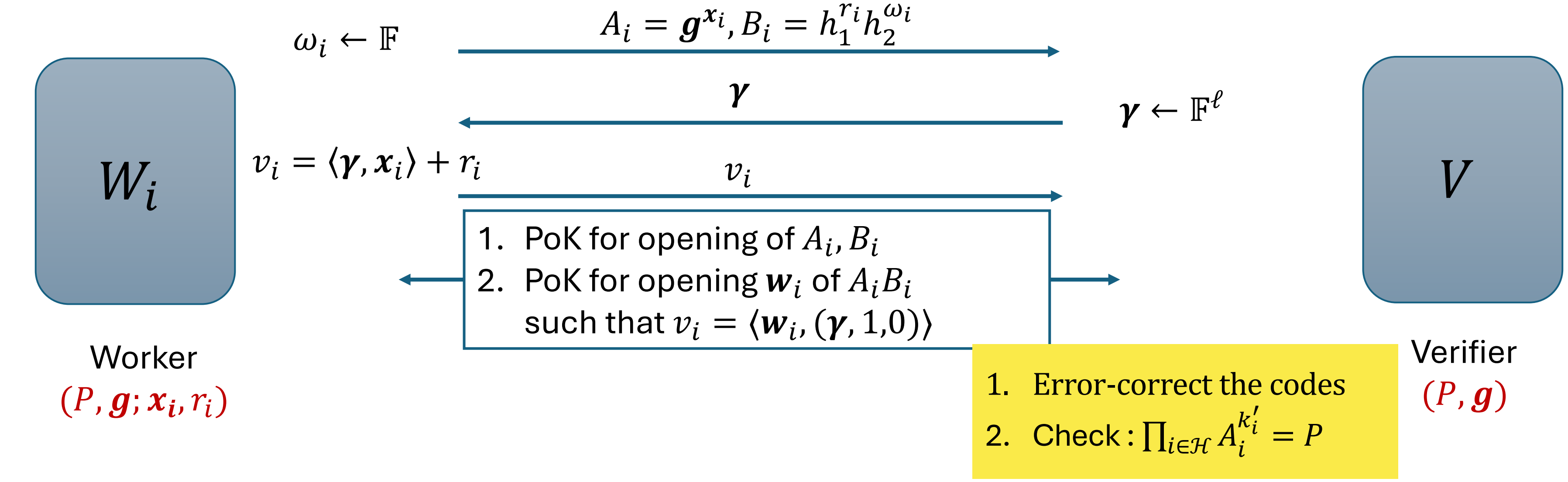
$L(a, b, c)$	$L(a_1, b_1, c_1)$	$L(a_2, b_2, c_2)$	...	$L(a_{n-1}, b_{n-1}, c_{n-1})$	$L(a_n, b_n, c_n)$

Property: Error preserved while taking linear combination

Robust DPoK for discrete log relation

- P : computes $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \leftarrow \mathrm{Share}(\boldsymbol{x})$ and then shares $\boldsymbol{x}_i \rightarrow W_i$
- P : samples $r \leftarrow \mathbb{F}$, computes $(r_1, \ldots, r_n) \leftarrow \mathrm{Share}(r)$, and then shares $r_i \rightarrow W_i$

Broadcast Model



Outline

- Robust DPoK for discrete log relation
- Robust DPoK for PS Signatures
  - PS Signatures
  - PoK for PS Signatures
  - Robust DPoK for PS Signatures
  - (Similarly: Robust DPoK for BBS+ Signatures)
- Overview of our compiler

Bilinear Group

$$(q, G_1, G_2, G_T, e, g, h)$$

- $G_1, G_2,$ and G_T are groups of order q (prime).
- g, h are generators of G_1, G_2 respectively.
- $e : G_1 \times G_2 \rightarrow G_T$ is a bilinear map.
- For all $g \in G_1, h \in G_2\ ; x, y \in \mathbb{F}\ (\mathbb{F}_q)$

$$e(g^x, h^y) = e(g, h)^{xy} = e(g^y, h^x)$$

PS Signatures

Keygen

- Setup: $(q, G_1, G_2, G_T, e, g, h)$
- Sample $(x, y_1, \dots, y_\ell) \leftarrow \mathbb{F}^{n+1}$
- Set $(X, Y_1, \dots, Y_\ell) = (h^x, h^{y_1}, \dots, h^{y_\ell})$
- Output (sk, pk), where
  - $sk = (x, y_1, \dots, y_\ell)$
  - $pk = (h, X, Y_1, \dots, Y_\ell)$

Sign

- Input: $sk, \boldsymbol{m} = (m_1, \dots, m_\ell)$
- Output: $\sigma = (\sigma_1, \sigma_2)$, where
  - $\sigma_1 \leftarrow G_1^*$
  - $\sigma_2 = \sigma_1^{x + m_1 y_1 + \cdots + m_\ell y_\ell}$

Verify

- Input: $pk, \boldsymbol{m} = (m_1, \dots, m_\ell), \sigma = (\sigma_1, \sigma_2)$
- Output: 1 iff

$$\sigma_1 \neq e_1 \ \wedge \ e\left(\sigma_1, X Y_1^{m_1} \cdots Y_\ell^{m_\ell}\right) = e(\sigma_2, h)$$

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. CT-RSA 2016.

PS Signatures

Keygen

- Setup: $(q, G_1, G_2, G_T, e, g, h)$
- Sample $(x, y_1, \ldots, y_\ell) \leftarrow \mathbb{F}^{n+1}$
- Set $(X, Y_1, \ldots, Y_\ell) = (h^x, h^{y_1}, \ldots, h^{y_\ell})$
- Output (sk, pk), where
  - $sk = (x, y_1, \ldots, y_\ell)$
  - $pk = (h, X, Y_1, \ldots, Y_\ell)$

Sign

- Input: $sk, \boldsymbol{m} = (m_1, \ldots, m_\ell)$
- Output: $\sigma = (\sigma_1, \sigma_2)$, where
  - $\sigma_1 \leftarrow G_1^*$
  - $\sigma_2 = \sigma_1^{x + m_1 y_1 + \cdots + m_\ell y_\ell}$

Verify

- Input: $pk, \boldsymbol{m} = (m_1, \ldots, m_\ell), \sigma = (\sigma_1, \sigma_2)$
- Output: 1 iff

$$\sigma_1 \neq e_1 \wedge e\left(\sigma_1, X Y_1^{m_1} \cdots Y_\ell^{m_\ell}\right) = e(\sigma_2, h)$$

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. CT-RSA 2016.

PoK for PS Signatures

Rerandomization of signature

- $r, t \leftarrow \mathbb{Z}_q$
- $\sigma' = (\sigma_1^r, (\sigma_2 \cdot \sigma_1^t)^r)$

P — Prover $(pk, \boldsymbol{m}, \sigma = (\sigma_1, \sigma_2))$

V — Verifier $(pk = h, X, Y_1, \dots, Y_\ell)$

$P \rightarrow V$: $\sigma' = (\sigma'_1, \sigma'_2)$

PoK for opening of:

$$e(\sigma'_1, X) \cdot \prod_{j=1}^{\ell} \left(e(\sigma'_1, Y_j)\right)^{m_j} \cdot e(\sigma'_1, h)^t = e(\sigma'_2, h)$$

Reduces to PoK for discrete log in G_T

V accepts if PoK is valid

Robust DPoK for PS Signatures

Rerandomization of signature

- $r, t \leftarrow \mathbb{Z}_q$
- $\sigma' = (\sigma_1^r, (\sigma_2 \cdot \sigma_1^t)^r)$

- $(\boldsymbol{m_1}, \dots, \boldsymbol{m_n}) \leftarrow \text{Share}(\boldsymbol{m})$ for $\boldsymbol{m} \in \mathbb{F}^\ell$
- Shares $\boldsymbol{m_i} \rightarrow W_i$

Prover

W_i

Worker

$(pk, \boldsymbol{m_i})$

Prover broadcasts: $(\sigma' = (\sigma_1', \sigma_2'))$

Robust DPoK for opening of:

$$e(\sigma_1', X) \cdot \prod_{j=1}^{\ell} \left(e(\sigma_1', Y_j)\right)^{m_j} \cdot e(\sigma_1', h)^t = e(\sigma_2', h)$$

Similar to PS, PoK for BBS+ also reduces to PoK for discrete log

V accepts if PoK is valid

V

Verifier

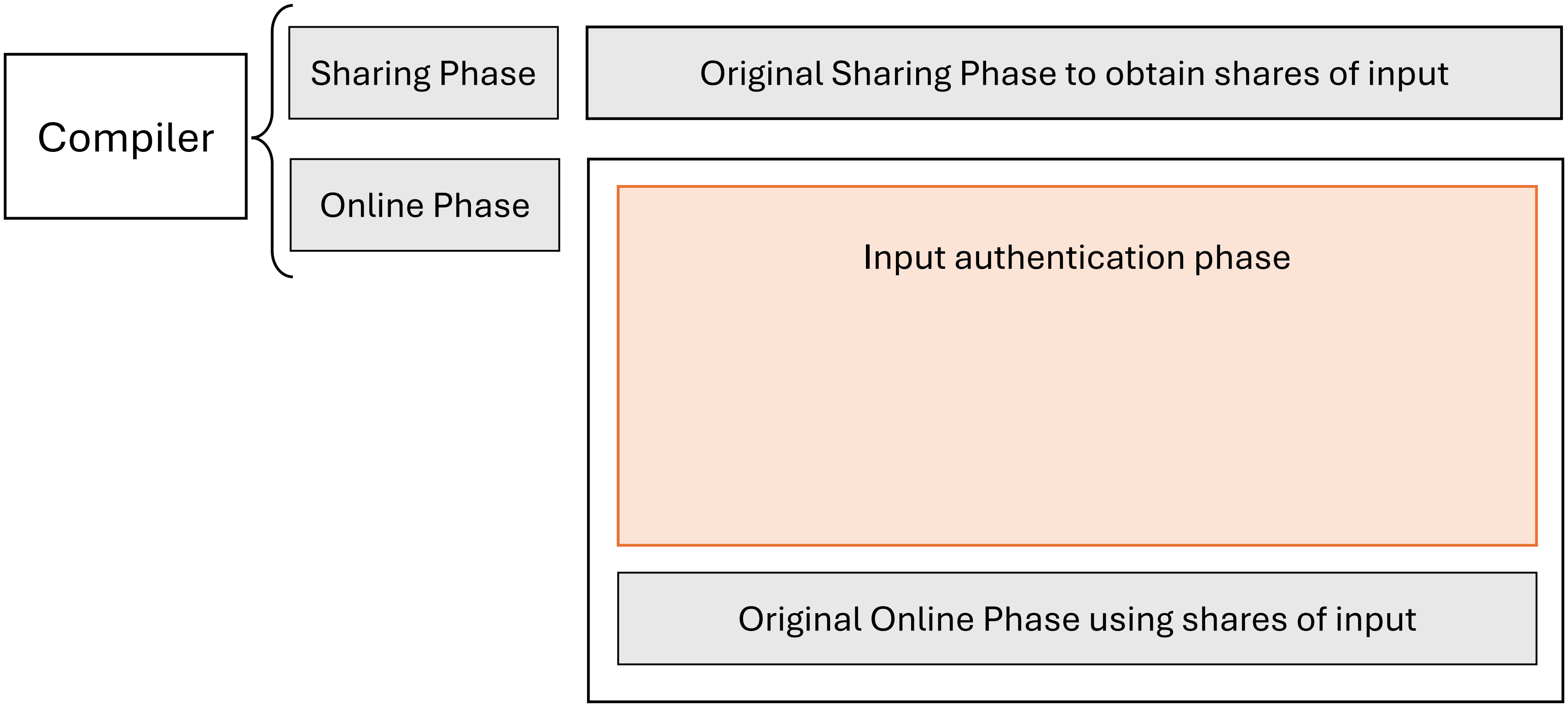
$(pk = h, X, Y_1, \dots, Y_\ell)$

Outline

- Robust DPoK for discrete log relation
- Robust DPoK for PS Signatures

  (Robust DPoK for BBS+ Signatures)
- Overview of our compiler

Input Authentication using our DPoK

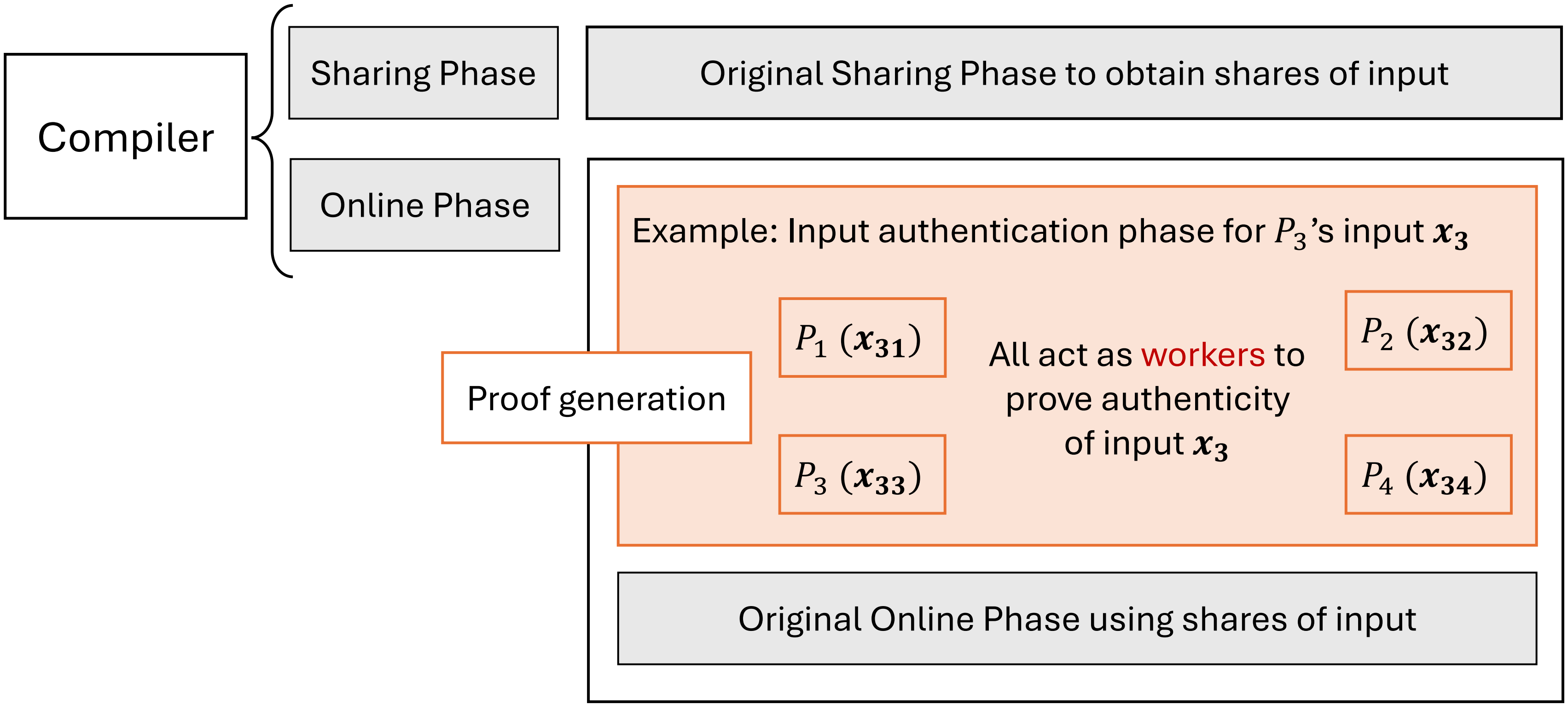


Input Authentication using our DPoK

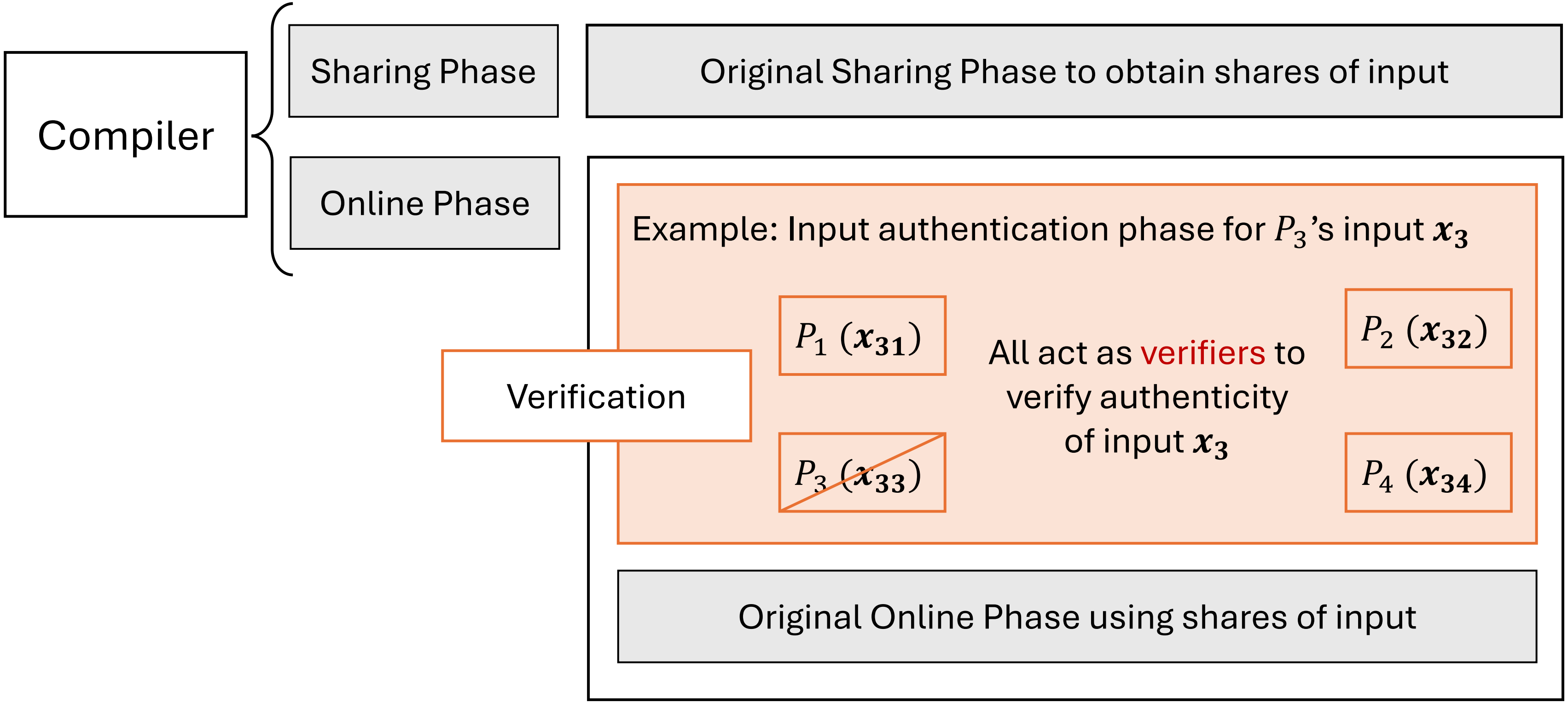
Compiler

- Sharing Phase
  - Original Sharing Phase to obtain shares of input
- Online Phase
  - Input authentication phase
    - Goal: Prove $\boldsymbol{x_i}$ ($P_i's$ input) is valid wrt signature σ_i corresponding to public key pk.
    - Each P_k acts as a worker to generate the proof.
    - Each P_k $(k \neq i)$ acts as a verifier.
  - Original Online Phase using shares of input

Input Authentication using our DPoK



Input Authentication using our DPoK



Input Authentication using our DPoK

Compiler

- Sharing Phase
- Online Phase

Original Sharing Phase to obtain shares of input

Input authentication phase

- Goal: Prove $\boldsymbol{x_i}$ ($P_i's$ input) is valid wrt signature σ_i corresponding to public key pk.
- Each P_k acts as a worker to generate the proof.
- Each P_k $(k \neq i)$ acts as a verifier.

Original Online Phase using shares of input

Less computational and communication overhead on underlying unauthenticated MPC execution

Our Performance

Number of Parties	Vanilla MPC		Auth MPC with MiMC Hash		DPoK Overhead	
	Time (sec)	Communication (MB)	Time (sec)	Communication (MB)	Time (sec)	Communication (kB)
3	33s	8437 MB	273s	13979 MB	5.7s	14.4 kB
5	125s	43823 MB	1369s	14498 MB	6.2s	30 kB
7	386.2s	127057 MB	3645.33s	207427 MB	8.2s	52 kB

Figure. Comparison of our DPoK-based approach for MPC input authentication with the naïve approach of validating BBS+ signatures inside MPC (which involves computing MiMC hashes inside MPC) for datasets of size 500×10.

Our Performance

Number of Parties	Vanilla MPC		Auth MPC with MiMC Hash		DPoK Overhead	
	Time (sec)	Communication (MB)	Time (sec)	Communication (MB)	Time (sec)	Communication (kB)
3	33s	8437 MB	273s	13979 MB	5.7s	14.4 kB
5	125s	43823 MB	1369s	14498 MB	6.2s	30 kB
7	386.2s	127057 MB	3645.33s	207427 MB	8.2s	52 kB

Figure. Comparison of our DPoK-based approach for MPC input authentication with the naïve approach of validating BBS+ signatures inside MPC (which involves computing MiMC hashes inside MPC) for datasets of size 500×10.

Summary

- Robust Distributed Proof of Knowledge (multi-prover)
- Construction of Robust DPoK for BBS+ and PS.
- Our compiler: preserves the security guarantees of underlying MPC (eg. Id-abort, GOD)

Thank you!

https://ia.cr/2022/1648

References

[ADEO21] Diego F. Aranha, Anders P. K. Dalskov, Daniel Escudero, and Claudio Orlandi. Improved threshold signatures, proactive secret sharing, and input certification from LSS isomorphisms. LATINCRYPT 2021.

[Bau16] Carsten Baum. On garbling schemes with and without privacy.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. CRYPTO 2004.

[ASM06] (BBS+) Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA.

[BJ18] Marina Blanton and Myoungin Jeong. Improved signature schemes for secure multi-party computation with certified inputs. ESORICS 2018.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. ESORICS 2013.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. CRYPTO 2007.

[KMW16] Jonathan Katz, Alex J. Malozemoff, and Xiao Wang. Efficiently enforcing input validity in secure two-party computation.

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. CT-RSA 2016.

[ZBB17] Yihua Zhang, Marina Blanton, and Fattaneh Bayatbabolghani. Enforcing input correctness via certification in garbled circuit evaluation. ESORICS 2017.

Our Performance

Number of Parties	Number of Rows	Vanilla MPC		DPoK Overhead	
		Time (sec)	Communication (MB)	Time (sec)	Communication (kB)
3	100	6.67	1733	0.519	13
	1000	64	16754	18	15
	2000	129	33398	65	15.3
	4000	260	66502	246	15.8
5	100	26	8838	0.643	28
	1000	265	87747	20	31
	2000	521	175671	76	32
	4000	958	350658	312	33

Figure. Comparison of our DPoK-based approach for MPC input authentication (using BBS+ signatures) with 3 and 5 parties on datasets with 10 columns. For example, datasets containing statistics of shipments are used as inputs to compute the industry-wide average of those statistics.