Zero Knowledge Proofs:
Challenges, Applications, and Real-world Deployment

NIST Workshop on Privacy Enhancing Cryptography

September 26th, 2024

Tjerand Silde & Akira Takahashi

NTNU

J.P.Morgan

AlgoCRYPT CoE

AI Research

This talk

1) Introduction to Zero Knowledge Proof (Akira)

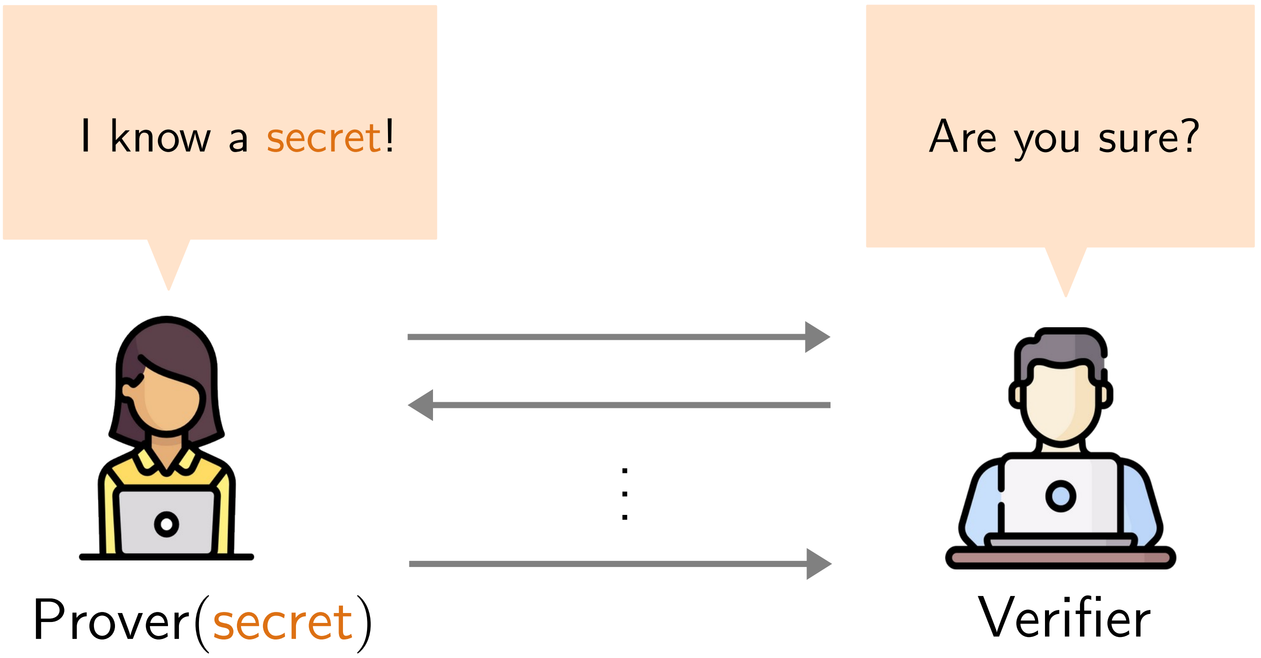
2) Technical Challenges (Akira)

3) Real-World Applications (Tjerand)

4) Insights from ZKP Workshop (Tjerand)

5) Resources and Standards (Tjerand)

What is Zero Knowledge Proof?



Basics

- ZKP is a two-party protocol, consisting of **Prover** and **Verifier**
- With ZKP, Prover can convince Verifier that she has some secret information without disclosing the secret
- Long history of research starting from the ‘80s [GMR85]. Lots of efficiency improvements during the last decade
  - cf. **ZK-SNARK** (Succinct Non-interactive Argument of Knowledge)

Syntax of ZKP

- x: statement (i.e. public input)
- w: witness (i.e. secret input)
- R: relation function, outputting 1 or 0
- "I know w s.t. $R(x, w) = 1$"

b: decision bit

Completeness

- If Prover and Verifier honestly follow the protocol, then Verifier halts by outputting $b = 1$

Prover(x, w)

Verifier(x)

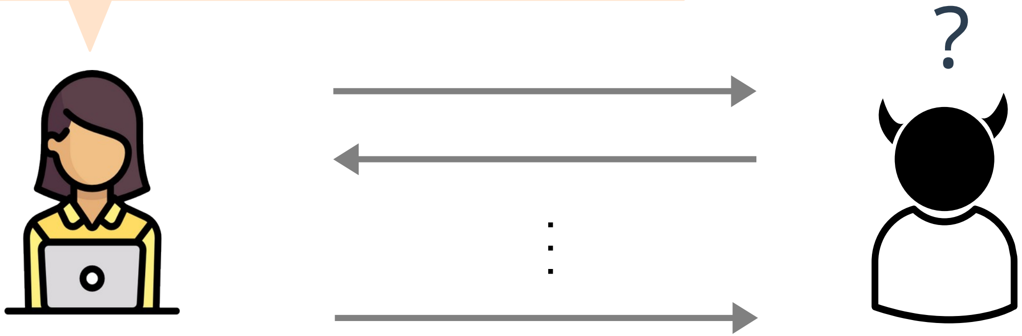
$b = 1$: "accept"

$b = 0$: "reject"

Security Goals of Zero Knowledge Proof

- x: statement (i.e. public input)
- w: witness (i.e. secret input)
- R: relation function, outputting 1 or 0
- "I know w s.t. $R(x, w) = 1$"

- Tries to steal w



Prover(x, w) Verifier(x)

Zero Knowledge (ZK)

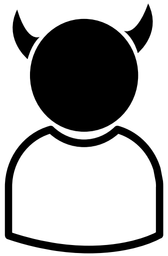
- Protecting against malicious verifier
- Verifier learns nothing about Prover's secret
- Formally, ZK is guaranteed by showing the existence of "Simulator"

Security Goals of Zero Knowledge Proof

- Maliciously interacts with Verifier
- Might use an "invalid" input

$$R(x^*, w^*) = 0$$

You lied!



Prover(x^*, w^*)

Verifier(x^*)

$b = 0$: "reject"

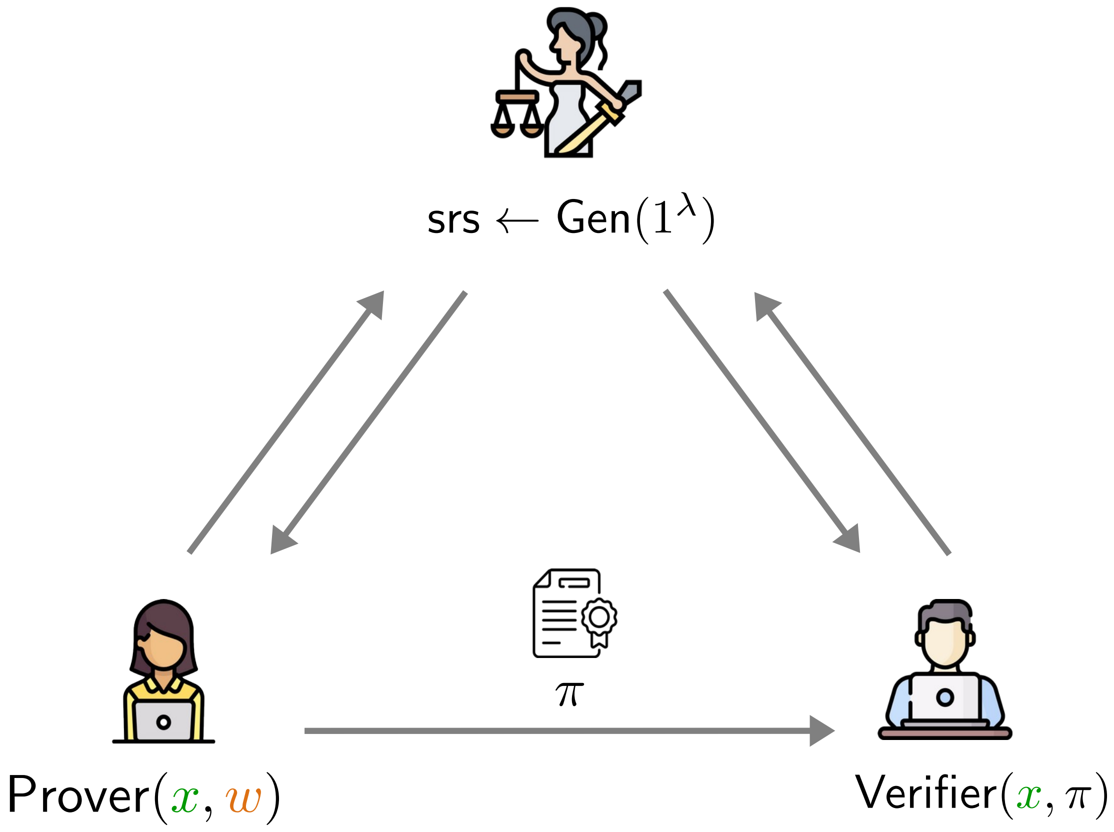
Zero Knowledge (ZK)

- Protecting against malicious verifier
- Verifier learns nothing about Prover's secret
- Formally, ZK is guaranteed by showing the existence of "Simulator"

Knowledge Soundness (KSND)

- Protecting against malicious prover
- If Prover uses an invalid secret, then Verifier catches it with high probability
- Formally, knowledge soundness is guaranteed by showing the existence of "Knowledge Extractor"

Non-interactive Zero Knowledge Proof (NIZK)



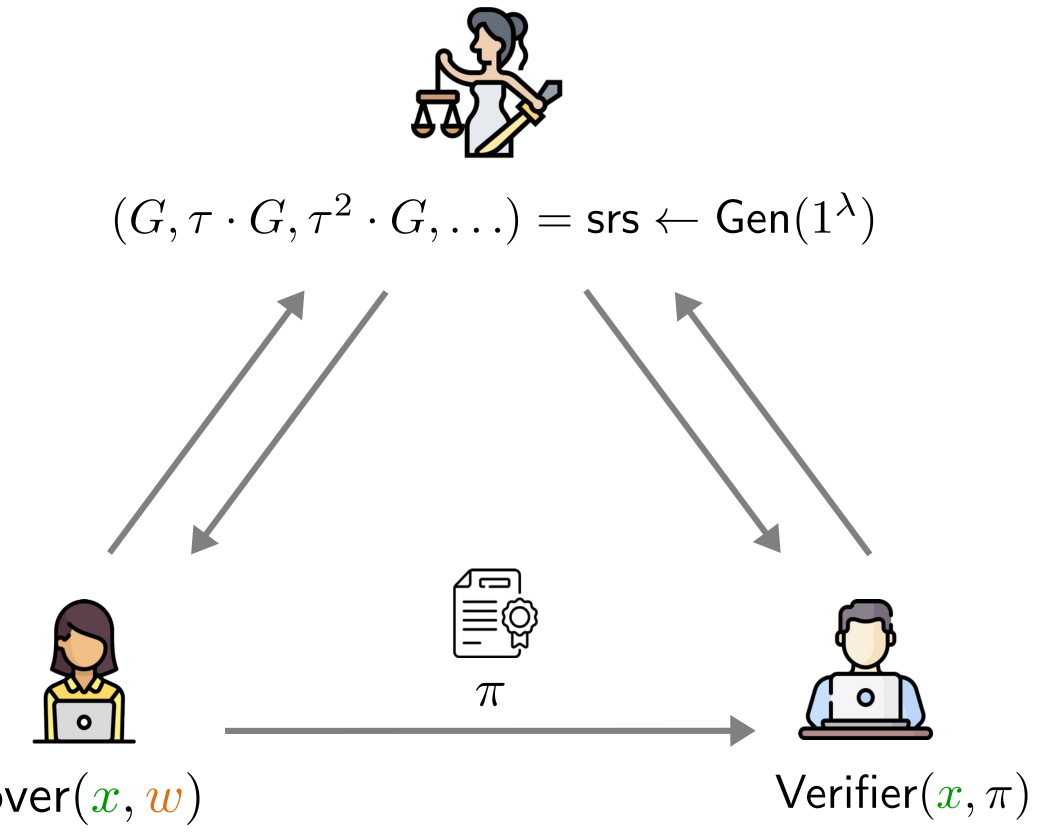
Removing Interactions

- Ideally, Prover should create a one-shot proof string π
- Verifier checks π asynchronously
- Such π is reusable and can be checked by potentially many verifiers

Types of Trusted Setup

- Structured Reference String (SRS)
- Hash function modeled as Random Oracle
- Or both!

Non-interactive Zero Knowledge Proof (NIZK)



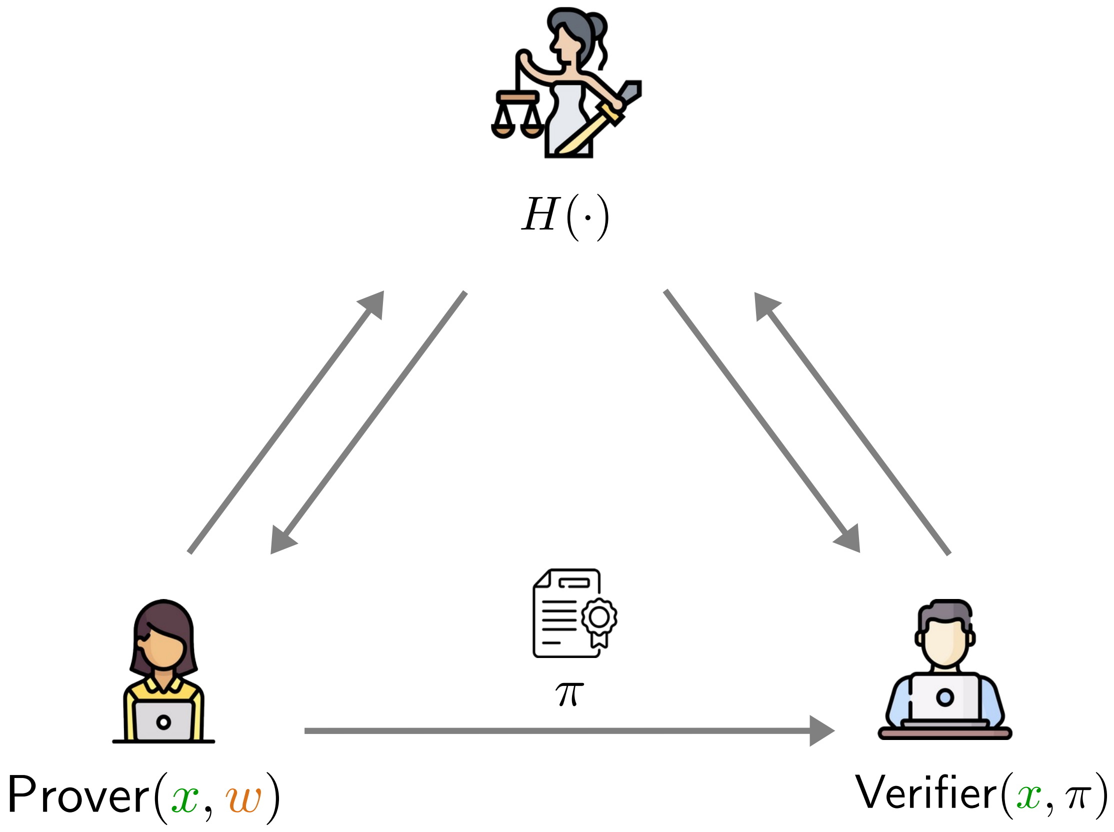
Removing Interactions

- Ideally, Prover should create a one-shot proof string π
- Verifier checks π asynchronously
- Such π is reusable and can be checked by potentially many verifiers

Types of Trusted Setup

- Structured Reference String (SRS)
- Hash function modeled as Random Oracle
- Or both!

Non-interactive Zero Knowledge Proof (NIZK)



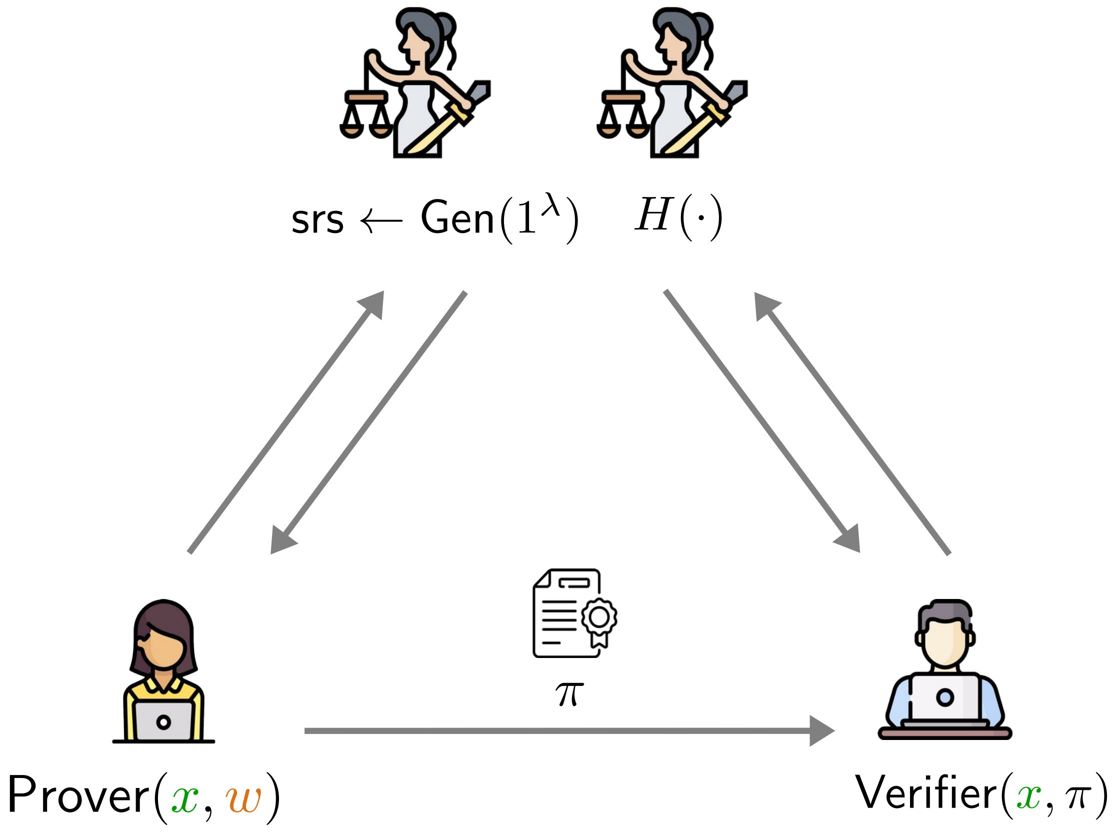
Removing Interactions

- Ideally, Prover should create a one-shot proof string π
- Verifier checks π asynchronously
- Such π is reusable and can be checked by potentially many verifiers

Types of Trusted Setup

- Structured Reference String (SRS)
- Hash function modeled as Random Oracle
- Or both!

Non-interactive Zero Knowledge Proof (NIZK)



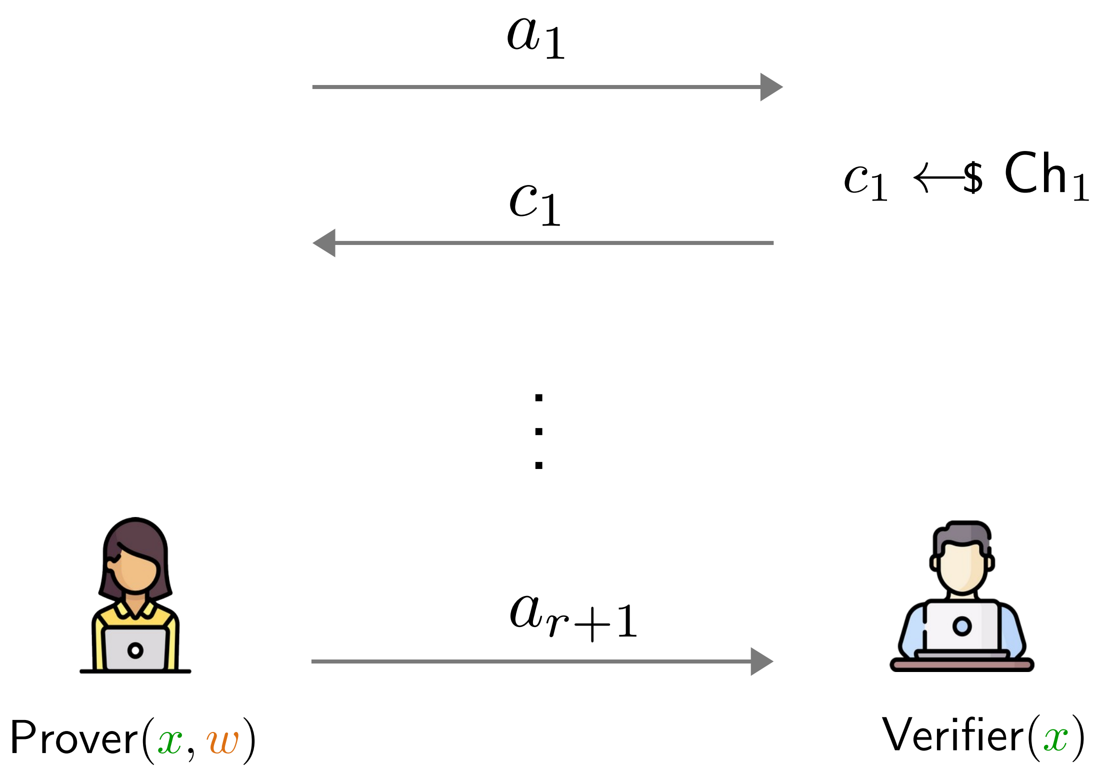
Removing Interactions

- Ideally, Prover should create a one-shot proof string π
- Verifier checks π asynchronously
- Such π is reusable and can be checked by potentially many verifiers

Types of Trusted Setup

- Structured Reference String (SRS)
- Hash function modeled as Random Oracle
- Or both!

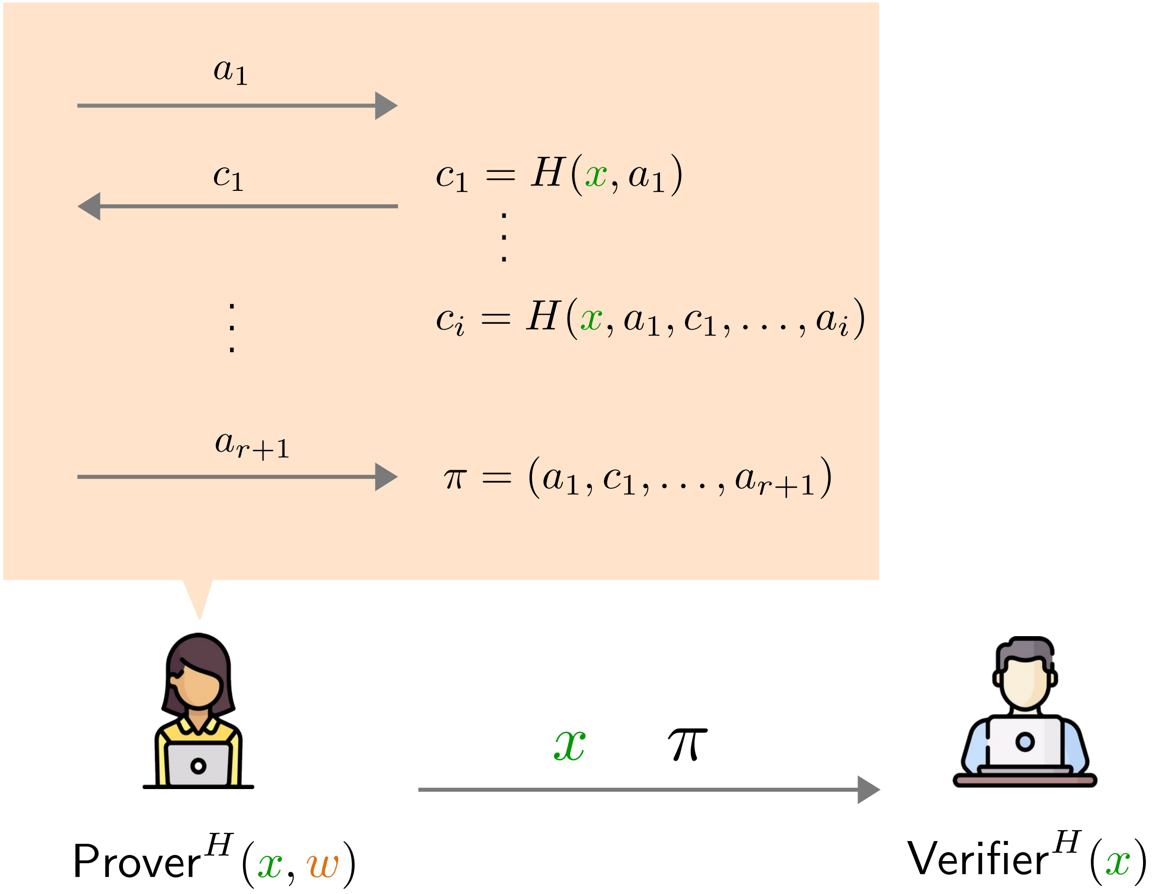
Paradigm of NIZK I: Fiat-Shamir [FS87]



Modular Design of NIZK

- **Step 1.** Construct a "public-coin" interactive protocol
  - Verifier does not require a secret state
  - ZK against semi-honest Verifier (Honest-Verifier ZK)
- **Step 2.** NI Prover and Verifier obtain challenge by locally hashing a partial transcript so far
- Bonus: By hashing the message, FS-NIZK gives rise to a digital signature
- Example: Schnorr/EdDSA, CRYSTALS-Dilithium, PLONK family, Bulletproofs, etc.
- Many modern SNARKs are constructed from (Polynomial) Interactive Oracle Proofs converted to NIZK via Fiat-Shamir [BCS16, CHMMVW19, BFS19, GWC19, CFFQR20,...]

Paradigm of NIZK I: Fiat-Shamir [FS87]



Modular Design of NIZK

- **Step 1.** Construct a "public-coin" interactive protocol
  - Verifier does not require a secret state
  - ZK against semi-honest Verifier (Honest-Verifier ZK)
- **Step 2.** NI Prover and Verifier obtain challenge by locally hashing a partial transcript so far
- Bonus: By hashing the message, FS-NIZK gives rise to a digital signature
- Example: Schnorr/EdDSA, CRYSTALS-Dilithium, PLONK family, Bulletproofs, etc.
- Many modern SNARKs are constructed from (Polynomial) Interactive Oracle Proofs converted to NIZK via Fiat-Shamir [BCS16, CHMMVW19, BFS19, GWC19, CFFQR20,...]

Paradigm of NIZK I: Fiat-Shamir [FS87]

a_1 →

c_1 ← $\quad c_1 = H(vk, a_1, \text{msg})$

$\vdots$

$c_i = H(vk, a_1, c_1, \ldots, a_i, \text{msg})$

a_{r+1} → $\quad \sigma = (a_1, c_1, \ldots, a_{r+1})$

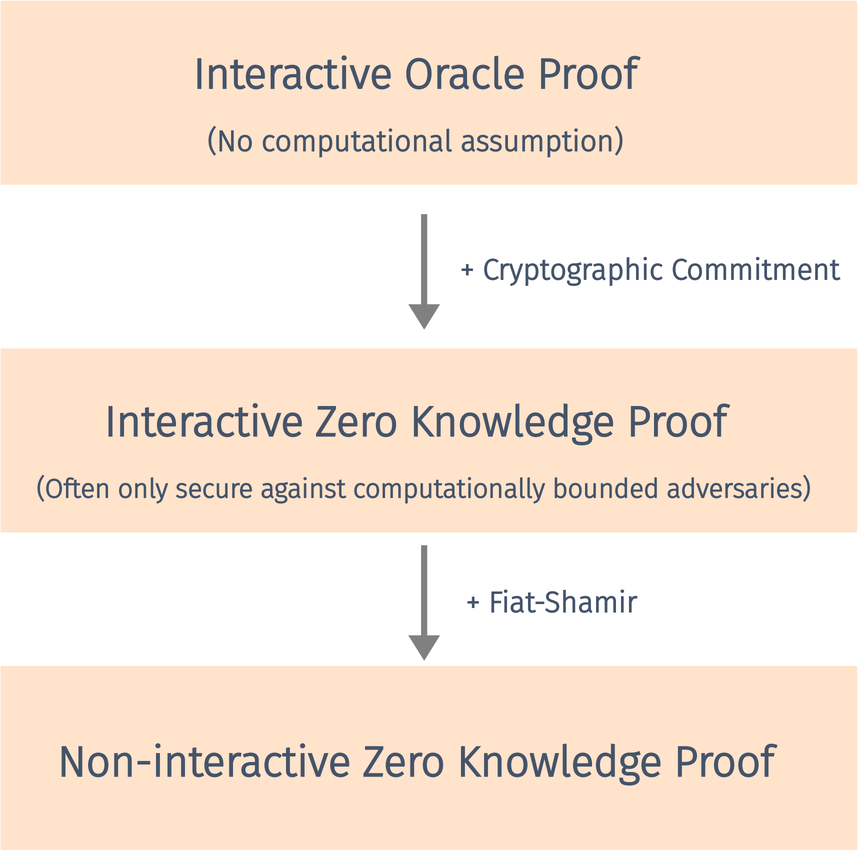
vk, msg, σ →

$\text{Signer}^H(vk, sk, \text{msg})$ $\qquad$ $\text{Verifier}^H(vk, \sigma, \text{msg})$

Modular Design of NIZK

- **Step 1.** Construct a "public-coin" interactive protocol
  - Verifier does not require a secret state
  - ZK against semi-honest Verifier (Honest-Verifier ZK)
- **Step 2.** NI Prover and Verifier obtain challenge by locally hashing a partial transcript so far
- Bonus: By hashing the message, FS-NIZK gives rise to a digital signature
- Example: Schnorr/EdDSA, CRYSTALS-Dilithium, PLONK family, Bulletproofs, etc.
- Many modern SNARKs are constructed from (Polynomial) Interactive Oracle Proofs converted to NIZK via Fiat-Shamir [BCS16, CHMMVW19, BFS19, GWC19, CFFQR20,...]

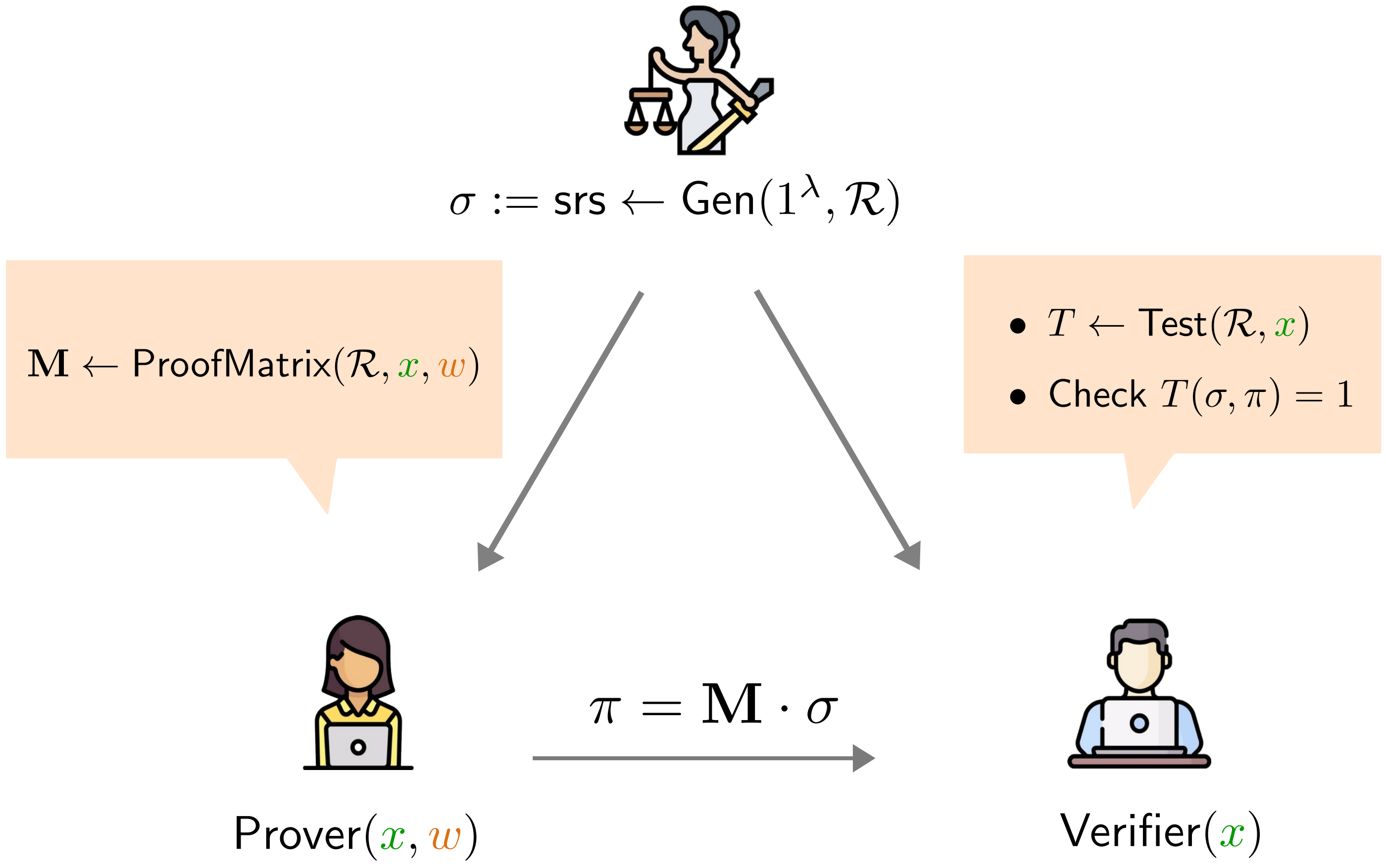
Paradigm of NIZK I: Fiat-Shamir [FS87]



Modular Design of NIZK

- **Step 1.** Construct a "public-coin" interactive protocol
  - Verifier does not require a secret state
  - ZK against semi-honest Verifier (Honest-Verifier ZK)
- **Step 2.** NI Prover and Verifier obtain challenge by locally hashing a partial transcript so far
- Bonus: By hashing the message, FS-NIZK gives rise to a digital signature
- Example: Schnorr/EdDSA, CRYSTALS-Dilithium, PLONK family, Bulletproofs, etc.
- Many modern SNARKs are constructed from (Polynomial) Interactive Oracle Proofs converted to NIZK via Fiat-Shamir [BCS16, CHMMVW19, BFS19, GWC19, CFFQR20,…]

Paradigm of NIZK II: Linear Interactive Proofs [GGPR13,BCI+13]



NIZK without Fiat-Shamir

- **Step 1.** srs generator outputs a relation-dependent vector
- **Step 2.** NI Prover applies linear transformation to srs
- **Step 3.** NI Verifier derives a testing function, allowing to check whether correct linear transformation has been applied
- Example: Groth16
- Important: Prover and Verifier should never learn internal randomness of Gen; otherwise, malicious prover can easily prove a false statement

Technical Challenges

1) Balancing Generality, Efficiency and Assumptions

2) Advanced Security

3) Interoperability

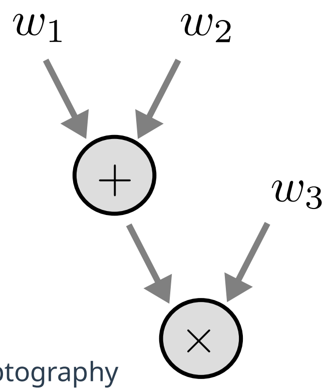
Types of ZKP

General-Purpose ZKP

- Supports arbitrary NP relation R
- Relation is often described using an arithmetic circuit

$$\mathcal{R}_C = \{(x, w) : C(x, w) = 1\}$$

- Pros:
  - Can prove correct execution of *any* program
  - Suitable for verifiable and outsourced computation
- Cons:
  - circuit gets complex for certain non-linear computations
  - E.g., elliptic curve arithmetic, comparison, table lookup, etc.



Specialized ZKP

- Designed for particular type of NP relation R

$$\mathcal{R}_{\mathrm{DL}} = \{(X, w) : X = w \cdot G\}$$

$$\mathcal{R}_{\mathrm{SIS}} = \{(\mathbf{x}, \mathbf{w}) : \mathbf{x} = \mathbf{A}\mathbf{w} \bmod q, \|\mathbf{w}\| \leq \beta\}$$

$$\mathcal{R}_{\mathrm{Lookup}} = \{(\mathbf{x}, \mathbf{w}) : \mathbf{w} \text{ is a subvector of } \mathbf{x}\}$$

- Pros:
  - Can prove and verify designated relations efficiently
  - Sufficient for some useful applications, e.g., proof of correct encryption, distributed key generation, signatures, etc.
- Cons:
  - Requires careful integration with general-purpose ZKP to support more complex statements

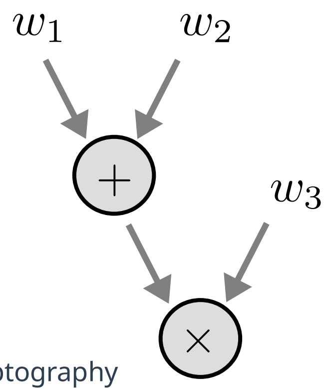
Types of ZKP

General-Purpose ZKP

- Supports arbitrary NP relation R
- Relation is often described using an arithmetic circuit

$$\mathcal{R}_C = \{(x, w) : C(x, w) = 1\}$$

- Pros:
  - Can prove correct execution of *any* program
  - Suitable for verifiable and outsourced computation
- Cons:
  - circuit gets complex for certain non-linear computations
  - E.g., elliptic curve arithmetic, comparison, table lookup, etc.



Specialized ZKP

- Designed for particular type of NP relation R

$$\mathcal{R}_{\mathrm{DL}} = \{(X, w) : X = w \cdot G\}$$

$$\mathcal{R}_{\mathrm{SIS}} = \{(\mathbf{x}, \mathbf{w}) : \mathbf{x} = \mathbf{A}\mathbf{w} \bmod q, \|\mathbf{w}\| \leq \beta\}$$

$$\mathcal{R}_{\mathrm{Lookup}} = \{(\mathbf{x}, \mathbf{w}) : \mathbf{w} \text{ is a subvector of } \mathbf{x}\}$$

- Pros:
  - Can prove and verify designated relations efficiently
  - Sufficient for some useful applications, e.g., proof of correct encryption, distributed key generation, signatures, etc.
- Cons:
  - Requires careful integration with general-purpose ZKP to support more complex statements

Desiderata

Proof Size

- Smaller proof saves storage and communication bandwidth
- Groth16 requires *only 3 group elements* from pairing-friendly curves
- State-of-the-art Polymath [Lip24] and PARI [DMS24] achieve even smaller proof sizes!

Assumptions

- To minimize a trust assumption, SRS should be avoided
- Better alternative: only trust the security of hash function modeled as RO (aka transparent setup), e.g., Bulletproofs, Brakedown, STARK, LaBRADOR, MPC/VOLE-in-the-Head, etc.
- Middle-ground solution: allows different parties to update SRS (aka updatable SRS) [GKMMM18]

Setup, Prover and Verifier Cost

- Universal Setup: Setup outputs SRS once and for all for arbitrary circuits

$$\mathsf{srs} \leftarrow \mathsf{Setup};\ \mathsf{srs}_C \leftarrow \mathsf{Derive}(\mathsf{srs}, C)$$

- Verifier sub-linear in $|C|$
- Prover time linear in #non-linear gates

Scalability

- How can we prove a large statement efficiently?
  - **Proof Aggregation**: aggregate many, asynchronously generated proofs, e.g., SnarkPack
  - **Incrementally Verifiable Computation [Valiant08]**: succinct proof of incremental computations via recursion or folding, e.g., Halo2, Nova, etc.

Desiderata

Proof Size

- Smaller proof saves storage and communication bandwidth
- Groth16 requires *only 3 group elements* from pairing-friendly curves
- State-of-the-art Polymath [Lip24] and PARI [DMS24] achieve even smaller proof sizes!

Assumptions

- To minimize a trust assumption, SRS should be avoided
- Better alternative: only trust the security of hash function modeled as RO (aka transparent setup), e.g., Bulletproofs, Brakedown, STARK, LaBRADOR, MPC/VOLE-in-the-Head, etc.
- Middle-ground solution: allows different parties to update SRS (aka updatable SRS) [GKMMM18]

Setup, Prover and Verifier Cost

- Universal Setup: Setup outputs SRS once and for all for arbitrary circuits

$$\mathsf{srs} \leftarrow \mathsf{Setup}; \mathsf{srs}_C \leftarrow \mathsf{Derive}(\mathsf{srs}, C)$$

- Verifier sub-linear in $|C|$
- Prover time linear in #non-linear gates

Scalability

- How can we prove a large statement efficiently?
  - **Proof Aggregation**: aggregate many, asynchronously generated proofs, e.g., SnarkPack
  - **Incrementally Verifiable Computation [Valiant08]**: succinct proof of incremental computations via recursion or folding, e.g., Halo2, Nova, etc.

Desiderata

Proof Size

- Smaller proof saves storage and communication bandwidth
- Groth16 requires *only 3 group elements* from pairing-friendly curves
- State-of-the-art Polymath [Lip24] and PARI [DMS24] achieve even smaller proof sizes!

Assumptions

- To minimize a trust assumption, SRS should be avoided
- Better alternative: only trust the security of hash function modeled as RO (aka transparent setup), e.g., Bulletproofs, Brakedown, STARK, LaBRADOR, MPC/VOLE-in-the-Head, etc.
- Middle-ground solution: allows different parties to update SRS (aka updatable SRS) [GKMMM18]

Setup, Prover and Verifier Cost

- Universal Setup: Setup outputs SRS once and for all for arbitrary circuits

$$\mathsf{srs} \leftarrow \mathsf{Setup}; \mathsf{srs}_C \leftarrow \mathsf{Derive}(\mathsf{srs}, C)$$

- Verifier sub-linear in $|C|$
- Prover time linear in #non-linear gates

Scalability

- How can we prove a large statement efficiently?
  - **Proof Aggregation**: aggregate many, asynchronously generated proofs, e.g., SnarkPack
  - **Incrementally Verifiable Computation [Valiant08]**: succinct proof of incremental computations via recursion or folding, e.g., Halo2, Nova, etc.

Desiderata

Proof Size

- Smaller proof saves storage and communication bandwidth
- Groth16 requires *only 3 group elements* from pairing-friendly curves
- State-of-the-art Polymath [Lip24] and PARI [DMS24] achieve even smaller proof sizes!

Assumptions

- To minimize a trust assumption, SRS should be avoided
- Better alternative: only trust the security of hash function modeled as RO (aka transparent setup), e.g., Bulletproofs, Brakedown, STARK, LaBRADOR, MPC/VOLE-in-the-Head, etc.
- Middle-ground solution: allows different parties to update SRS (aka updatable SRS) [GKMMM18]

Setup, Prover and Verifier Cost

- Universal Setup: Setup outputs SRS once and for all for arbitrary circuits

$$\mathsf{srs} \leftarrow \mathsf{Setup}; \mathsf{srs}_C \leftarrow \mathsf{Derive}(\mathsf{srs}, C)$$

- Verifier sub-linear in $|C|$
- Prover time linear in #non-linear gates

Scalability

- How can we prove a large statement efficiently?
  - **Proof Aggregation**: aggregate many, asynchronously generated proofs, e.g., SnarkPack
  - **Incrementally Verifiable Computation [Valiant08]**: succinct proof of incremental computations via recursion or folding, e.g., Halo2, Nova, etc.

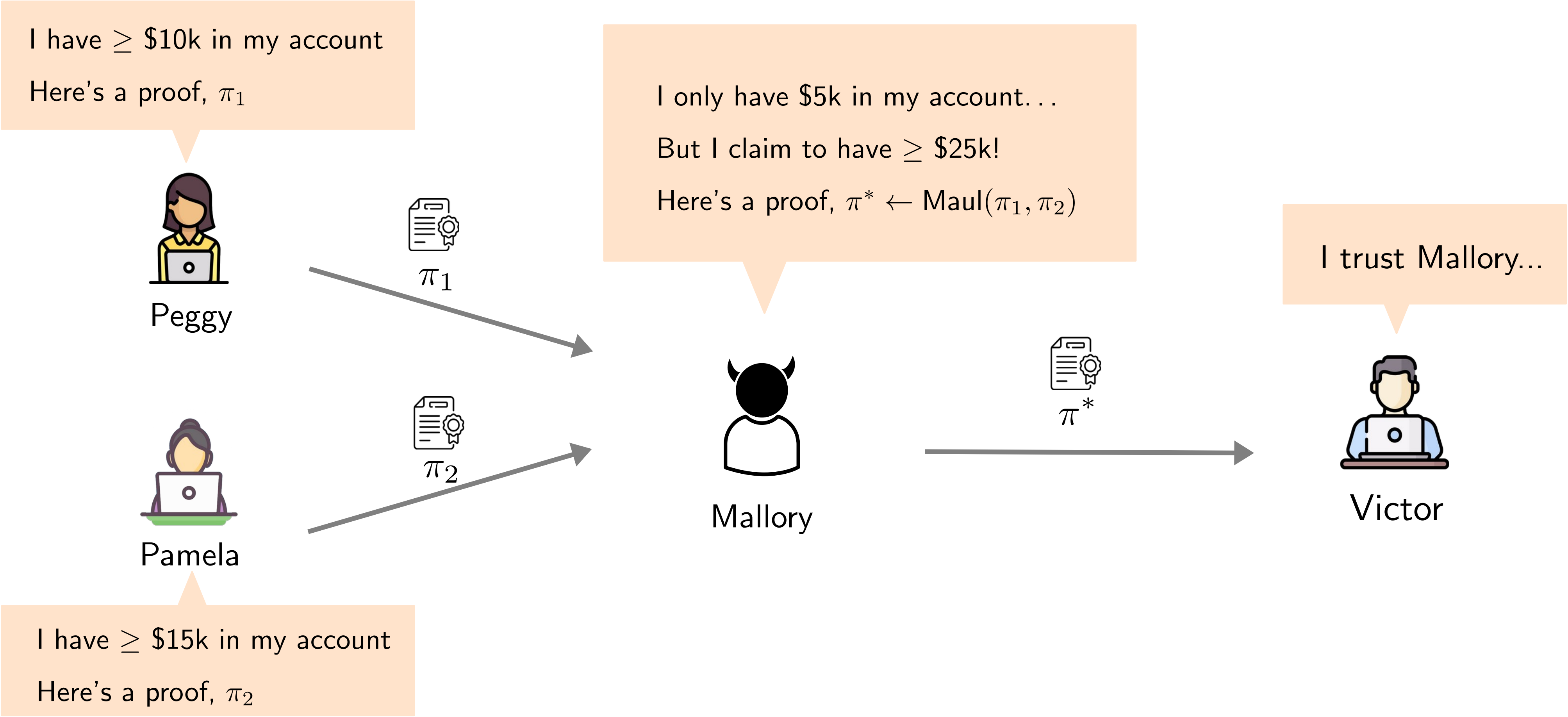
Technical Challenges

1) Balancing Generality, Efficiency and Assumptions

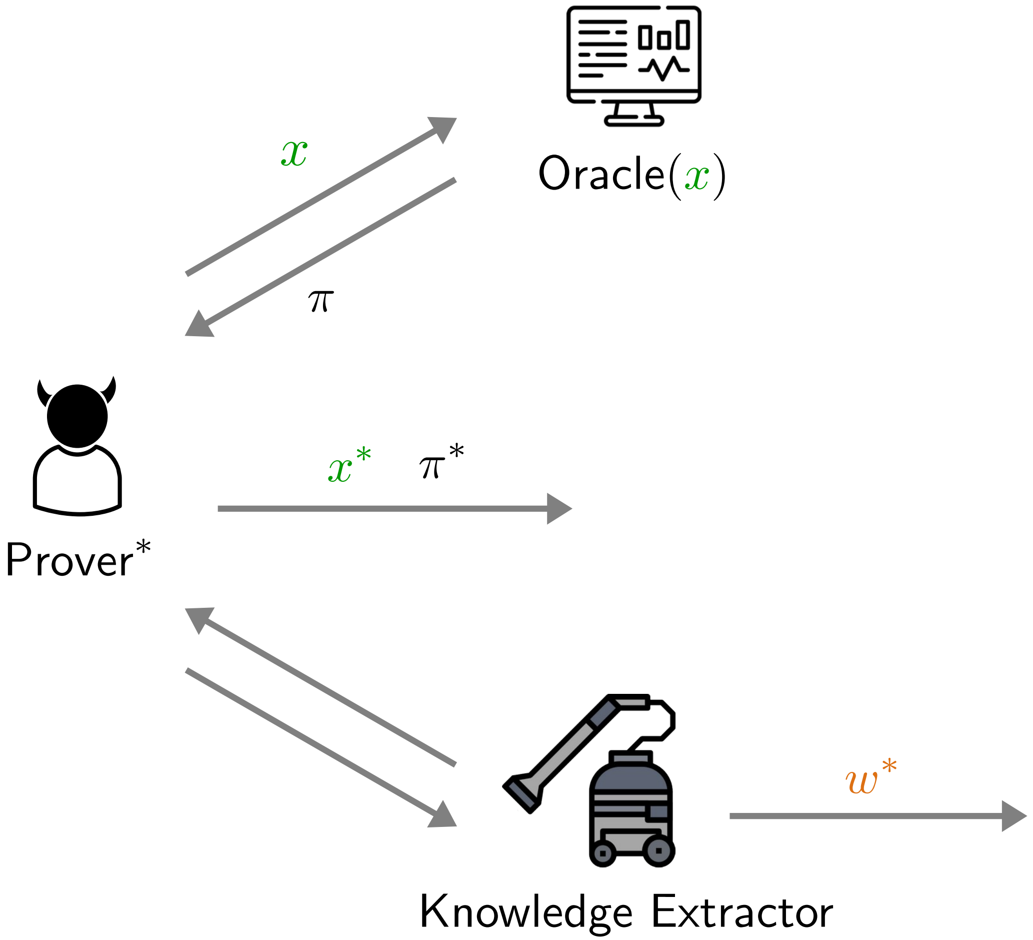
2) Advanced Security

3) Interoperability

ZK and Knowledge Soundness are not Enough: Malleability Attacks



Combined Notion: Simulation-Extractability



SIM-EXT Security

1. Prover* obtains fresh proof from Oracle
2. Prover* outputs "forgery" (x^*, π^*)
3. If (x^*, π^*) is accepting and not recorded by Oracle, then Prover* must know the corresponding witness w^*

- Intuitively, SIM-EXT guarantees **non-malleability:** a cheating prover cannot maul existing proofs to create a new one, without knowing a valid witness
- Cf. (S)EUF-CMA for signature and IND-CCA for PKE
- Crucial property NIZK should satisfy if used as a subroutine of another protocol
- Many practical NIZK schemes turn out to be SIM-EXT [GKKNZ22] [GOPTT22] [DG23] [FFKR23] [KPT23] [Lib24] [FFR24]
- Some schemes satisfy UC security [Canetti01] accepting some idealized setup [CF24] [BFKT24]

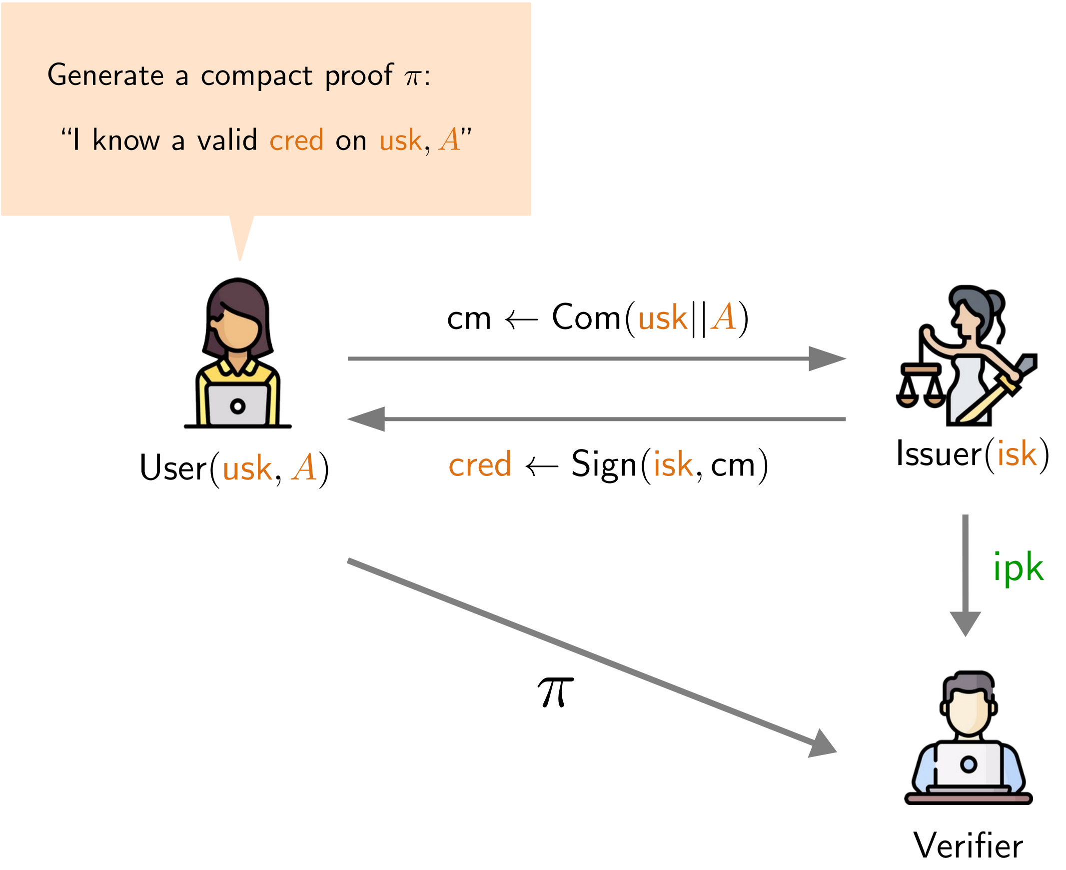
Technical Challenges

1) Balancing Generality, Efficiency and Assumptions

2) Advanced Security

3) Interoperability

Anonymous Credentials (High Level)

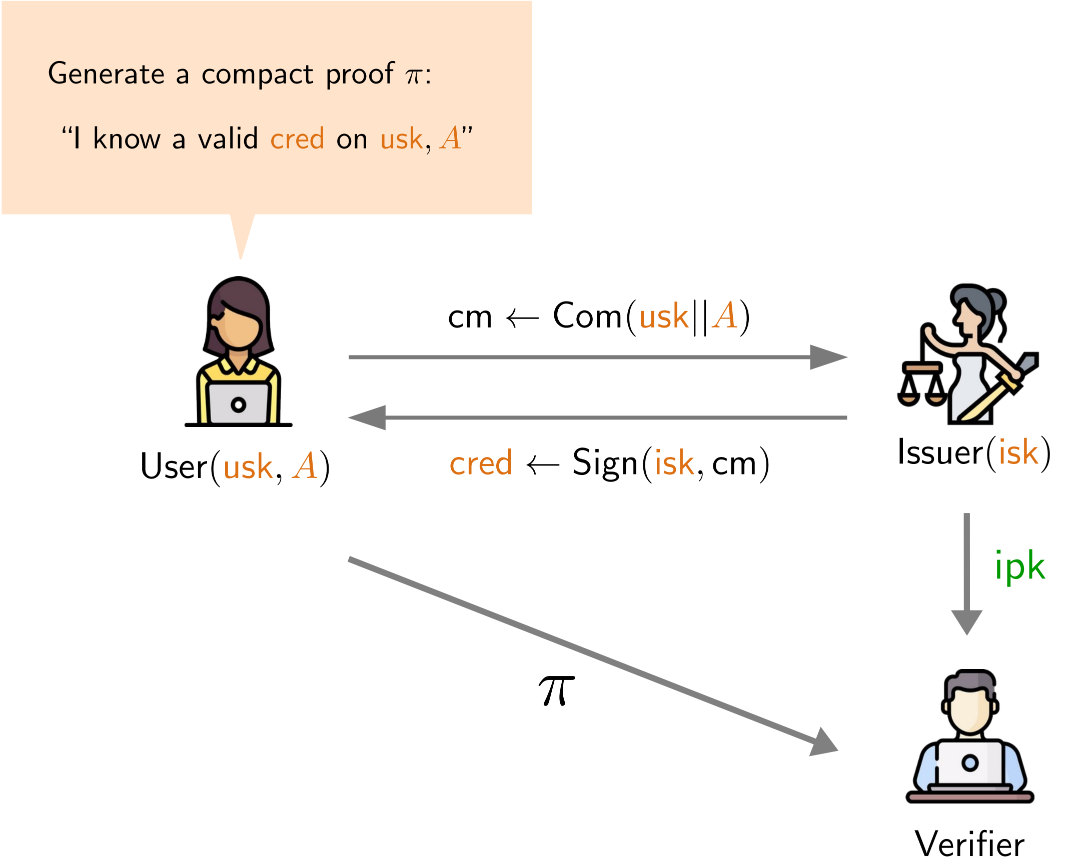


Protocol

- Issuer initially binds attributes and usk to secret credentials
- The owner of attributes produces a **proof string** in the form of ZKP
- By examining the proof string, Verifier gets convinced that User has valid attributes signed by Issuer
- Thanks to ZKP, the proof string only leaks minimum info about Prover's identity
- E.g., Verifier learns "User is => 21 years old" but nothing else

- isk: issuer secret key
- ipk: issuer public key
- usk: user secret key
- A: user attributes

Anonymous Credentials (High Level)



Interoperability

- Central ZKP for AC: Proof-of-Knowledge of valid signature
- If an arbitrary signature scheme is allowed, many efficient solutions exist: BBS+signature
- However, interoperability with standardized and widely deployed signature is often preferred in practice, e.g., RSA-PSS, ECDSA, EdDSA, etc.
- Verification condition of deployed schemes are not very ZK friendly. Can we make tailored ZKP more efficient?

- isk: issuer secret key
- ipk: issuer public key
- usk: user secret key
- A: user attributes

Takeaways

- ZKP allows Prover to prove the knowledge of a secret, while Verifier learns nothing about the secret
- Important Security Properties: **Knowledge Soundness** and **Zero Knowledge**
- Choose between general-purpose ZKP and specialized ZKP, or compose them carefully
- Which setup assumption is suitable for deployment?
  - Trusted, Transparent, Updatable, …
- What should you optimize?
  - Proof Size, Setup / Prover / Verifier Costs, Scalability, Assumptions, …
- Check whether ZKP satisfies advanced security such as SIM-EXT or UC if ZKP is used a building block of another protocol
- More research needed to optimize ZKP while retaining interoperability with standardized signatures or encryption schemes

References

- [GMR85] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In 17th ACM STOC, 1985.
- [FS87] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In CRYPTO'86, 1986.
- [BCI+13] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In TCC'13. 2013. https://ia.cr/2012/718
- [GGPR13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In EUROCRYP'13, 2013. https://ia.cr/2012/215
- [GKMMM18] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. Updatable and Universal Common Reference Strings with Applications to zk-SNARKs, CRYPTO'18, 2018. https://ia.cr/2018/280