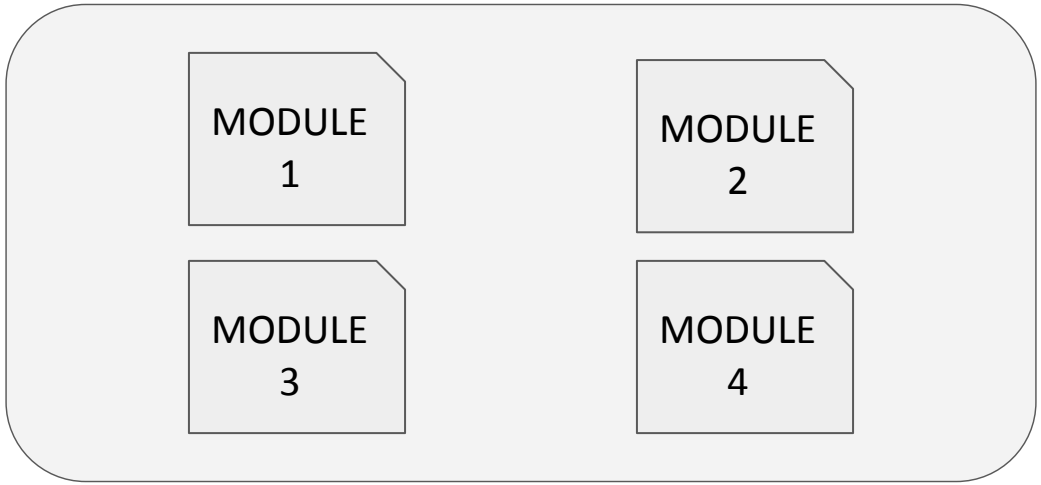
UC SNARKs with Transparent Setup without Programmable Random Oracle

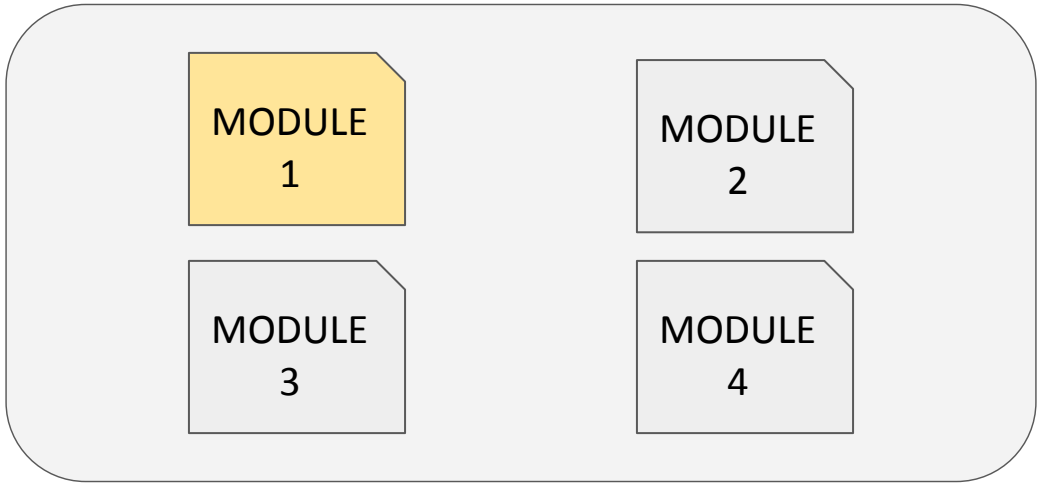
C. Badertscher, M. Campanelli, M. Ciampi, **L. Russo**, L. Siniscalchi

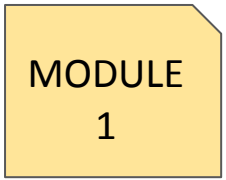
Universal Composability 101



Goal: Study the security of modular protocols
(with concurrent executions of multiple protocols)

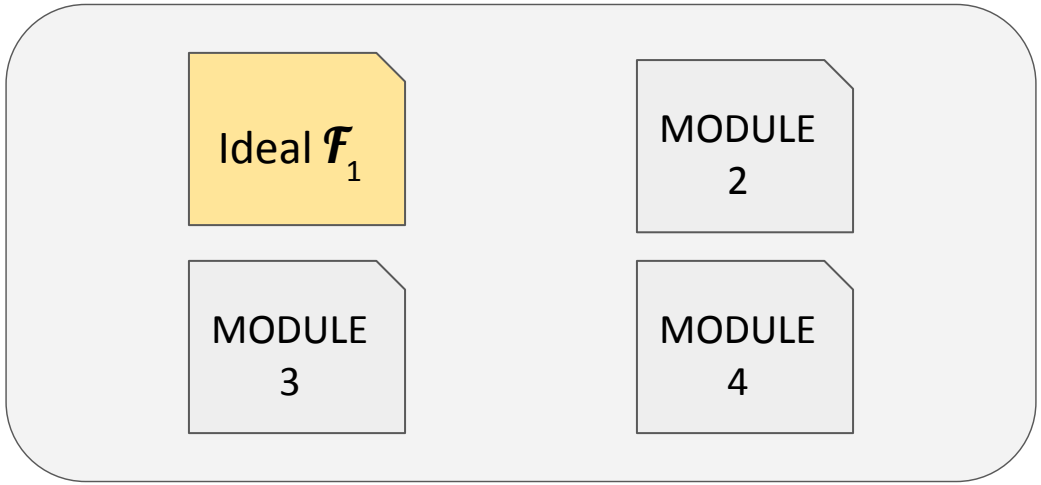
Universal Composability 101





- Define interface (~API)

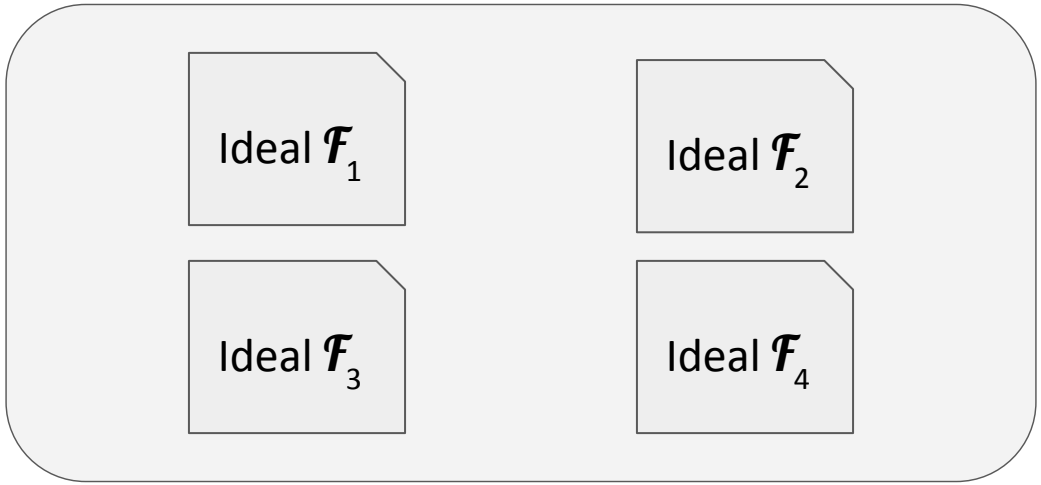
Universal Composability 101



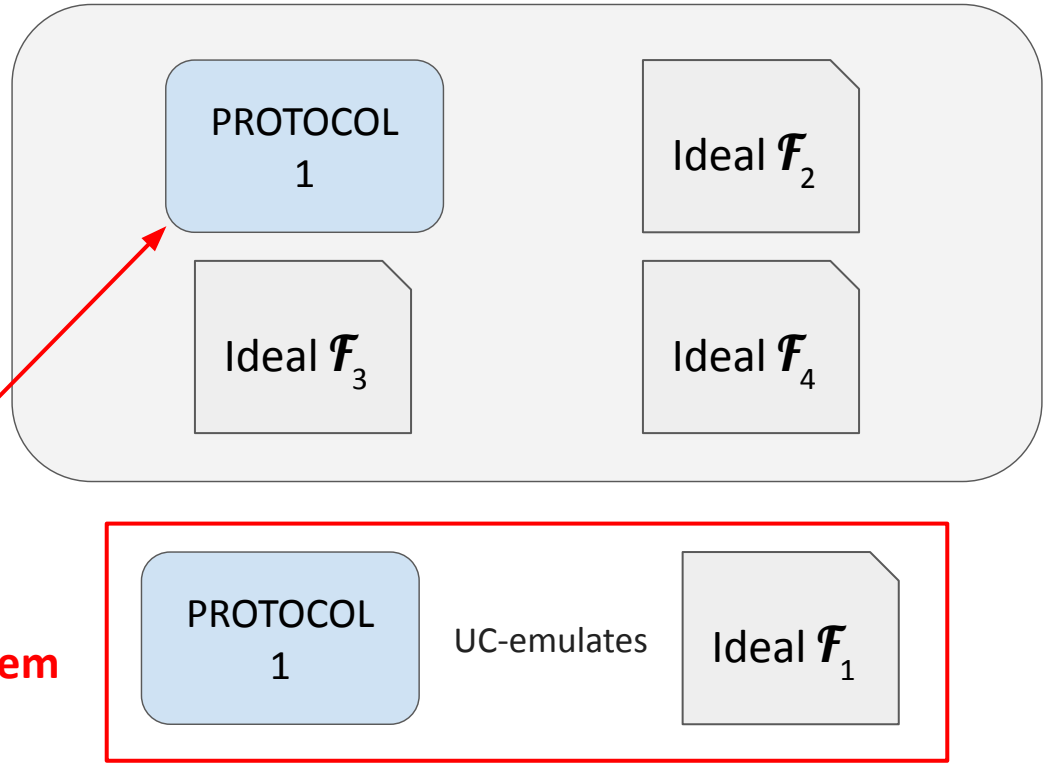
Ideal $\mathcal{F}_1$

- Define interface (~API)
- Define "**ideal**" requirements

Universal Composability 101

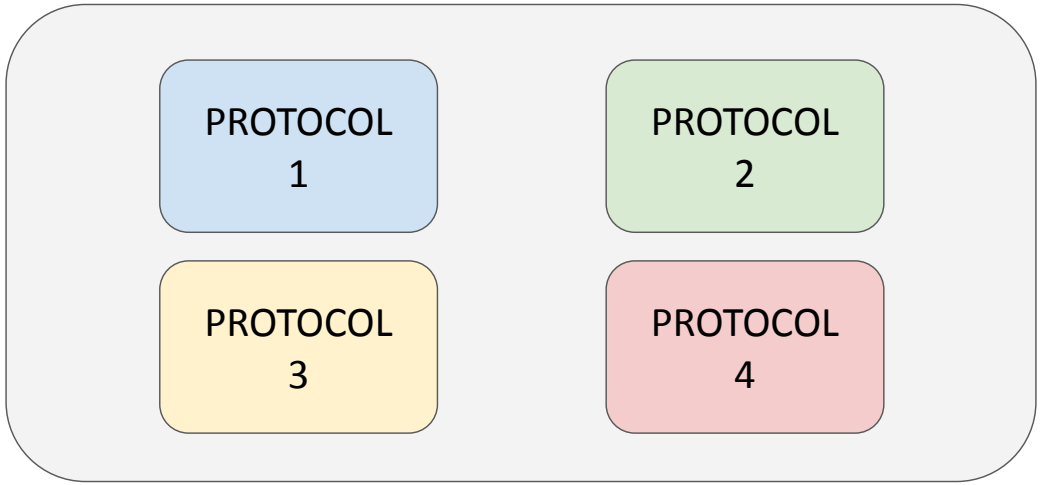


Universal Composability 101



UC Composition theorem

Universal Composability 101



UC Composition theorem

PROTOCOL 1 UC-emulates Ideal $\mathcal{F}_1$

PROTOCOL 2 UC-emulates Ideal $\mathcal{F}_2$

PROTOCOL 3 UC-emulates Ideal $\mathcal{F}_3$

PROTOCOL 4 UC-emulates Ideal $\mathcal{F}_4$

UC Functionalities

A variety of ideal functionalities, ranging from simple primitives to complex ones

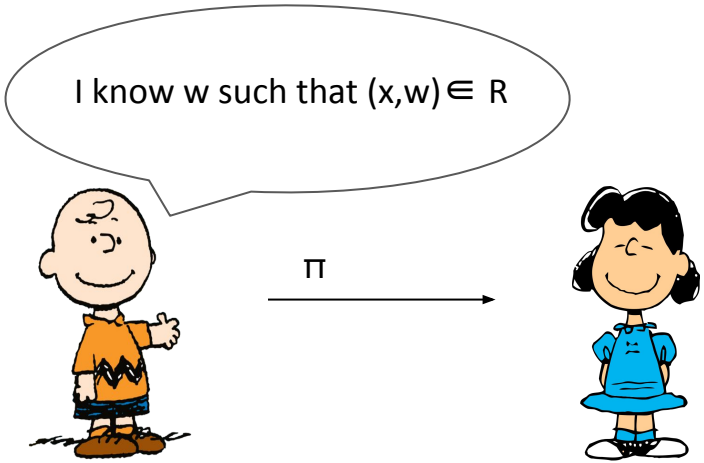
- Commitment schemes
- Oblivious transfer
- Secure multi-party computation
- Non-interactive ZK Proofs
- Ideal ledgers
- Voting
- …

UC Functionalities

A variety of ideal functionalities, ranging from simple primitives to complex ones

- Commitment schemes
- Oblivious transfer
- Secure multi-party computation
- zkSNARKs
- Ideal ledgers
- Voting
- …

zkSNARKs

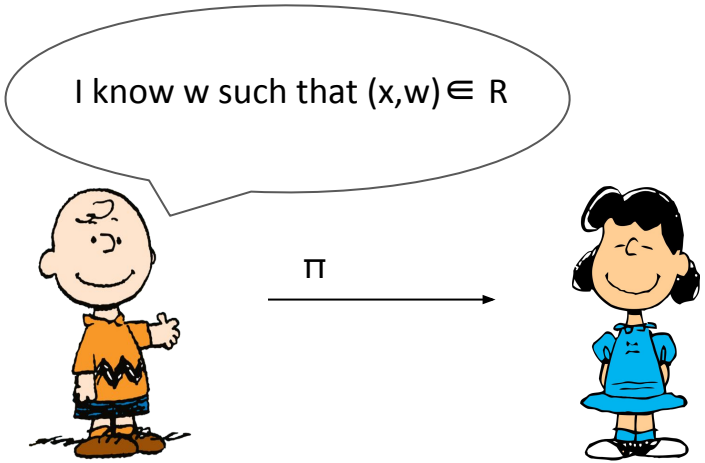


Input: R, x, w

Input: x, R

Output: accept/reject

zkSNARKs



Input: R, x, w

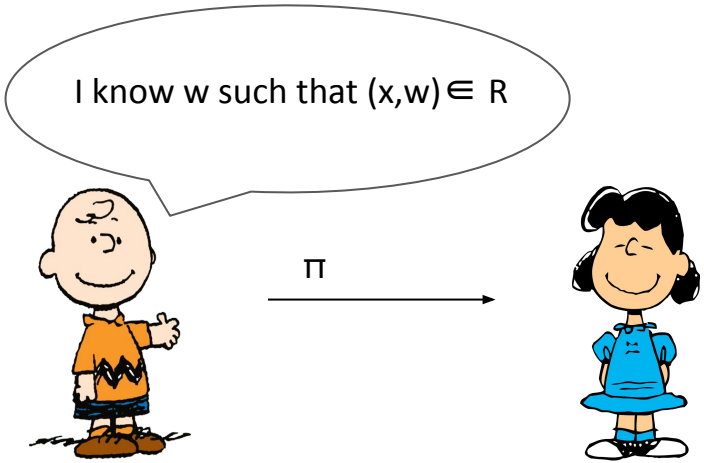
Input: x, R

Output: accept/reject

Zero-Knowledge

Verifier learns nothing besides that (x,w)∈ R

zkSNARKs



Input: R, x, w

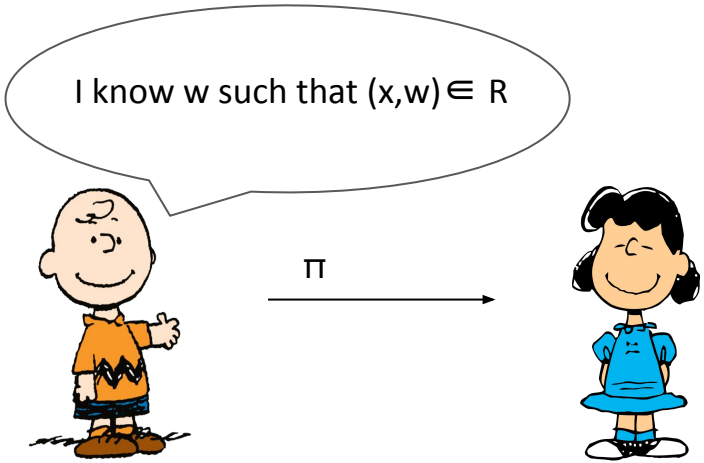
Input: x, R

Output: accept/reject

Zero-Knowledge

Succinct

zkSNARKs



Input: R, x, w

Input: x, R

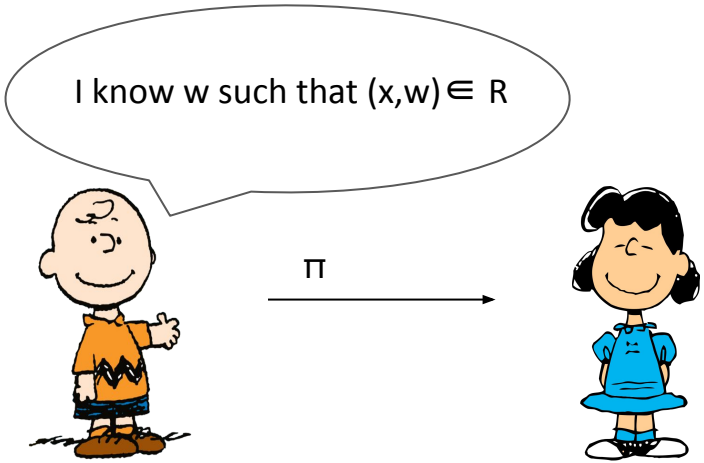
Output: accept/reject

Zero-Knowledge

Succinct

Non-interactive

zkSNARKs



Input: R, x, w

Input: x, R
Output: accept/reject

Zero-Knowledge

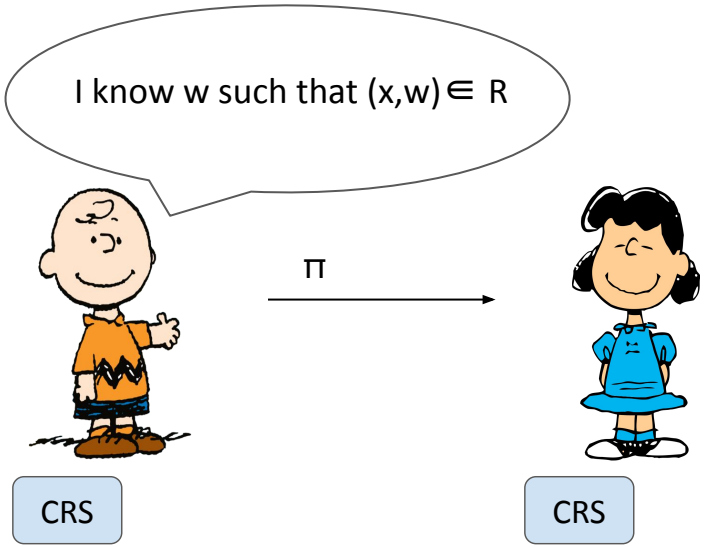
Succinct

Non-interactive

Arguments of Knowledge

If Verifier accepts, Prover “knows” w

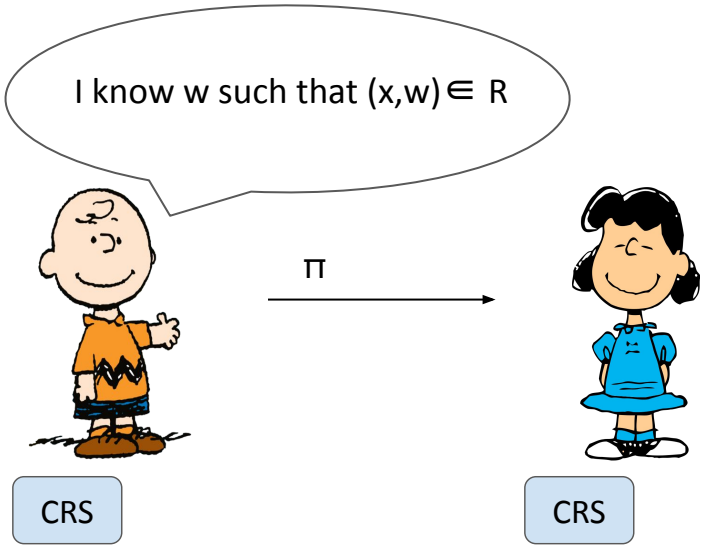
Setup of zkSNARKs



1. Trusted Setup

   CRS is generated by a trusted party

Setup of zkSNARKs



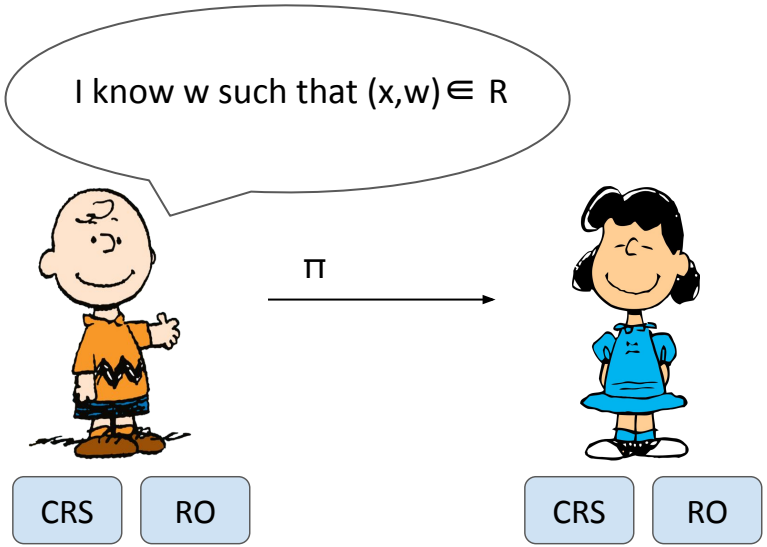
1. Trusted Setup

CRS is generated by a trusted party

“Zcash ‘19 Counterfeiting Vulnerability”

> The [BGG17] multi-party computation (MPC) protocol that produced Sprout parameters for the [BCTV14] construction follows the paper's setup procedure, including the computation of the extra elements. These are not included in the actual parameters distributed to Zcash nodes since they are omitted from the parameter file format used by the proving routine implementation of [BCTV14] in the libsnark library (used by Sprout). However, these elements do appear in the MPC ceremony transcript. Consequently, anyone with access to the ceremony transcript would have been able to create false proofs.

Setup of zkSNARKs

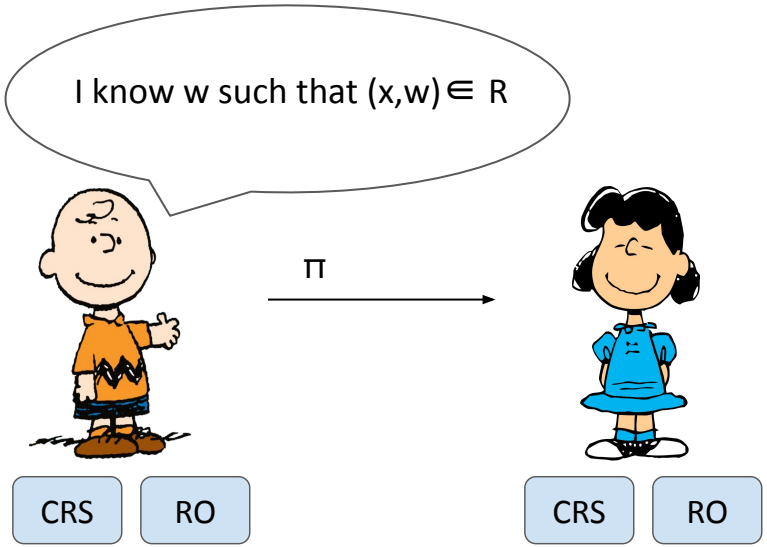


1. ~~Trusted Setup~~

2. Transparent Setup

   no trapdoor, toxic waste, etc.

Setup of zkSNARKs



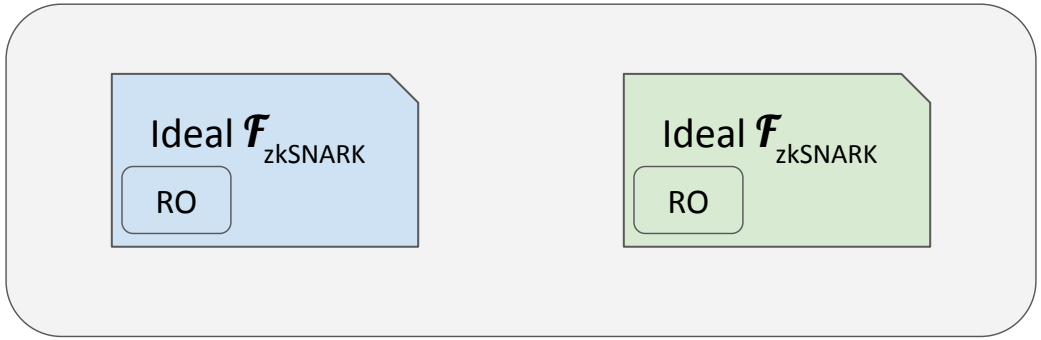
1. ~~Trusted Setup~~

2. Transparent Setup

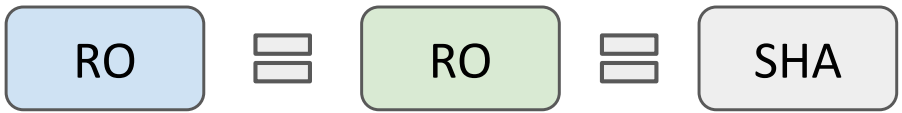
   no trapdoor, toxic waste, etc.

Security holds as long as the RO is used as a *local* resource

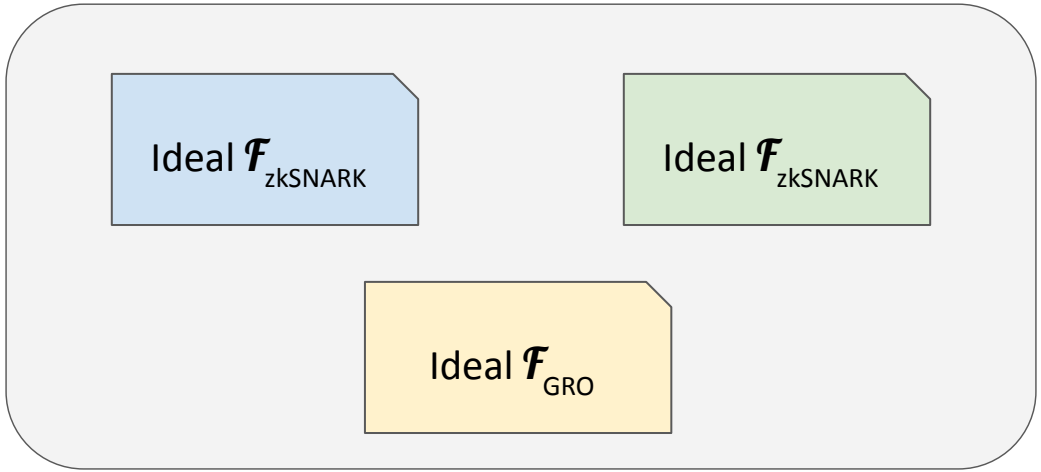
RO as a local resource



In practice, RO is is replaced by a single hash in many protocols



RO as a global resource



In practice, RO is is replaced by a single hash in many protocols

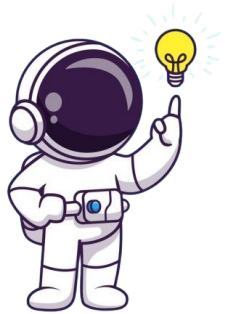
UC zkSNARKS

- Either we need to rely on a trusted setup [GKO+23, BFKT24]
- Or we need to be able to *program* the GRO [CF24]

UC zkSNARKS

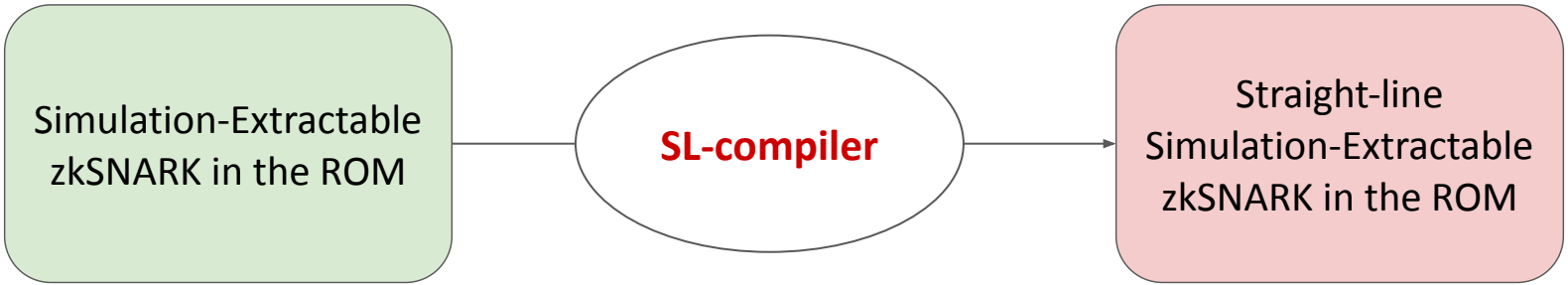
- Either we need to rely on a trusted setup [GKO+23, BFKT24]
- Or we need to be able to *program* the GRO [CF24]

Can we have transparent UC-SNARKs in the non-programmable GROM?



Our contributions

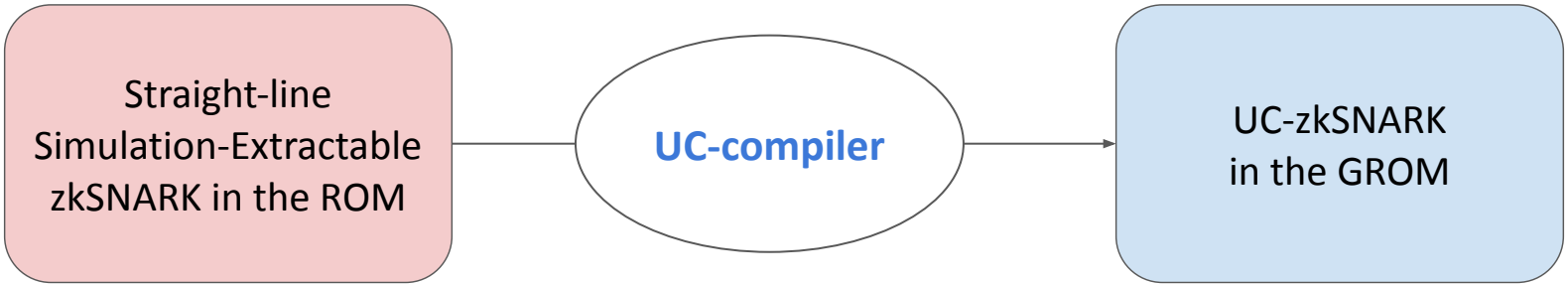
1. A compiler to straight-line simulation-extractable zkSNARKs



Preserves succinctness and setup transparency

Our contributions

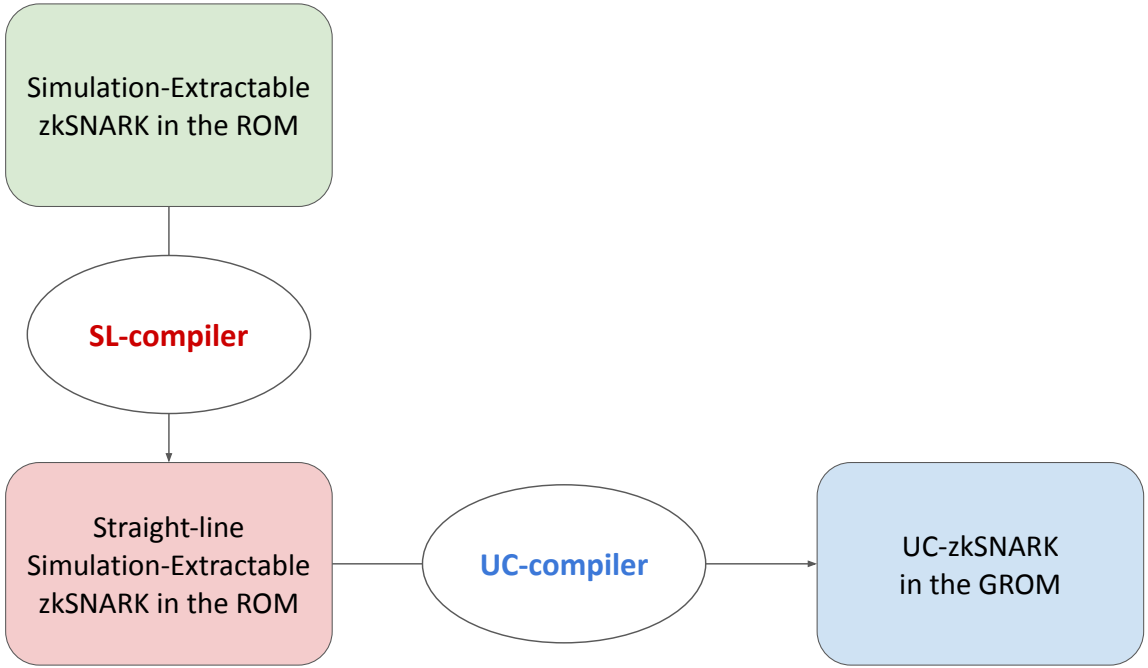
1. A compiler to straight-line simulation-extractable zkSNARKs
2. A compiler to UC-zkSNARKS in the Global ROM with Transparent setup



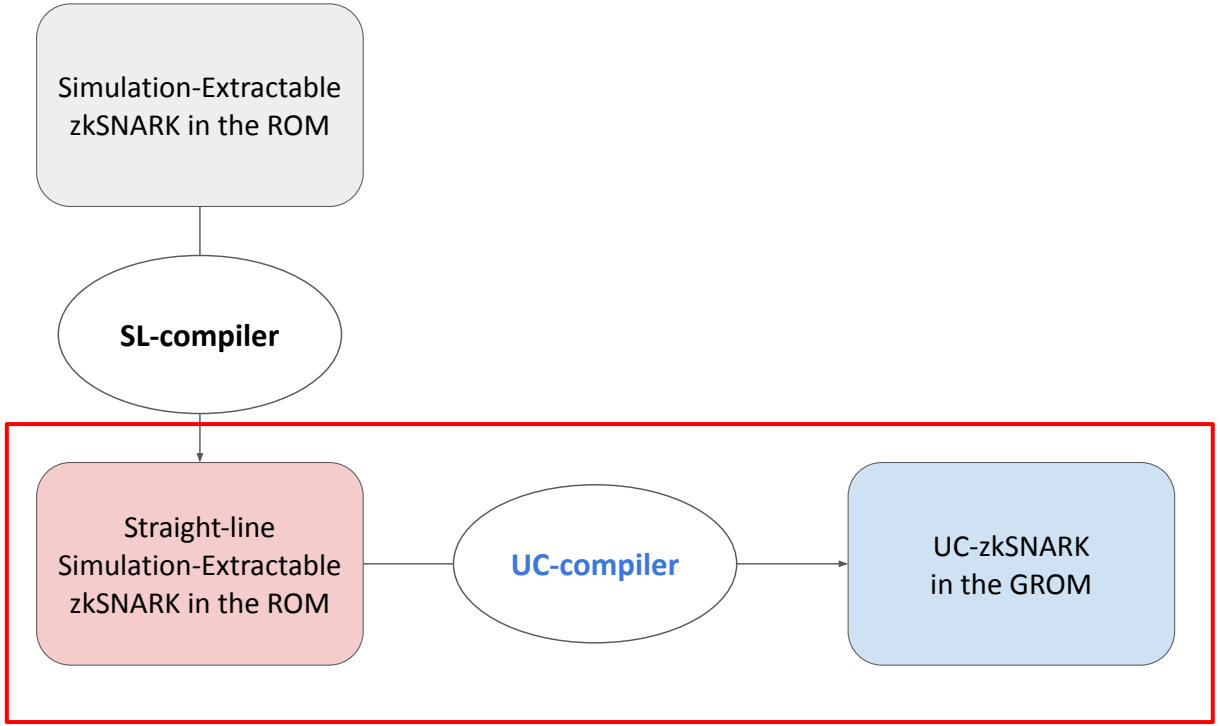
Preserves succinctness and setup transparency

Two lifting compilers

Micali/BCS
[CF24]



Two lifting compilers



UC-zkSNARKs in the GROM

Ideal $\mathcal{F}_{\text{GRO}}$

x, w

π

Prove(x,w)
If (x,w) ∈ R:
π = $\mathcal{S}$(x)

UC-zkSNARKs in the GROM

Ideal $\mathcal{F}_{\text{GRO}}$

x, w

π

Prove(x,w)
If (x,w) ∈ R:
π = $\mathcal{S}$(x)

$\mathcal{S}$ should simulate without trapdoor and without programming the RO

UC-zkSNARKs in the GROM

Ideal $\mathcal{F}_{\text{GRO}}$

x, w

π

Prove(x,w)
If (x,w) ∈ R:
π = $\mathcal{S}$(x)

$\mathcal{S}$ should simulate without trapdoor and without programming the RO
and this is impossible! [CV22]

UC-zkSNARKs in the GROM

Ideal $\mathcal{F}_{\text{GRO}}$

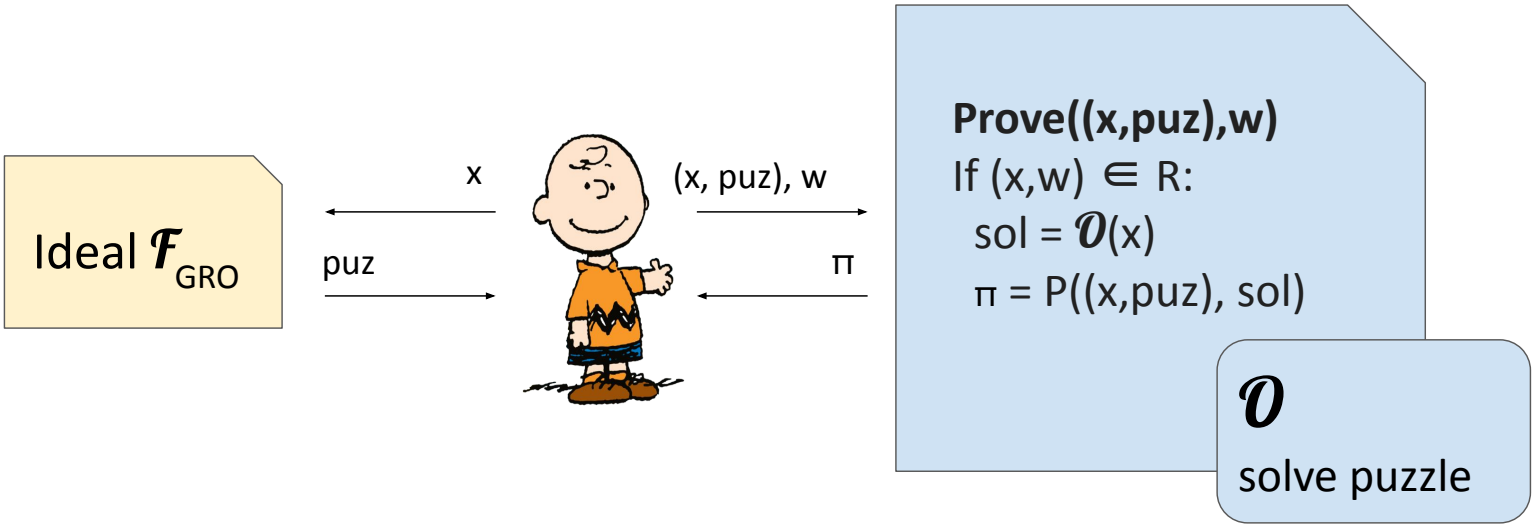
x, w

π

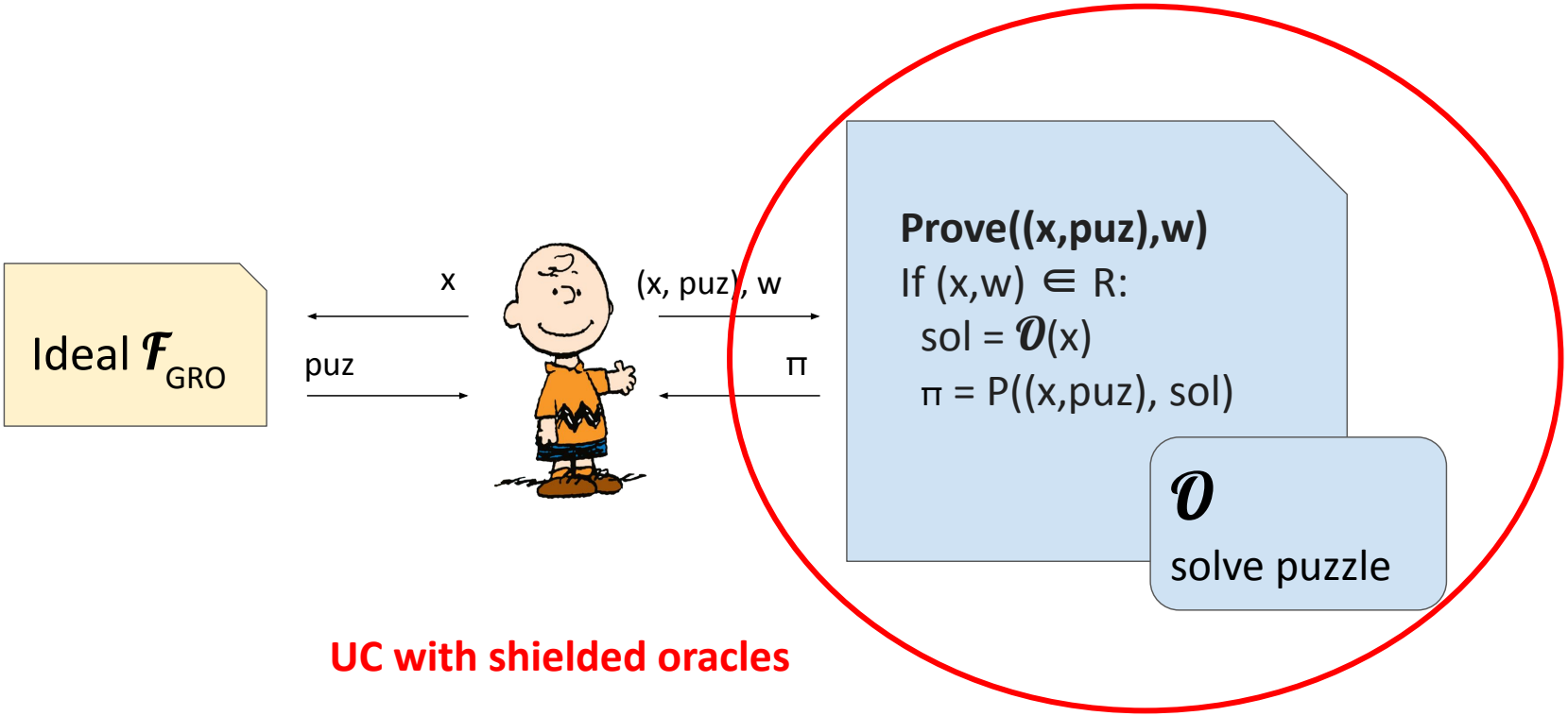
Prove(x,w)
If (x,w) ∈ R:
π = $\mathcal{S}$(x)

$\mathcal{S}$ should simulate without trapdoor and without programming the RO and this is impossible! [CV22]
unless we allow $\mathcal{S}$ to run in super-poly time (SPS) [Pas03]

Circumventing the impossibility

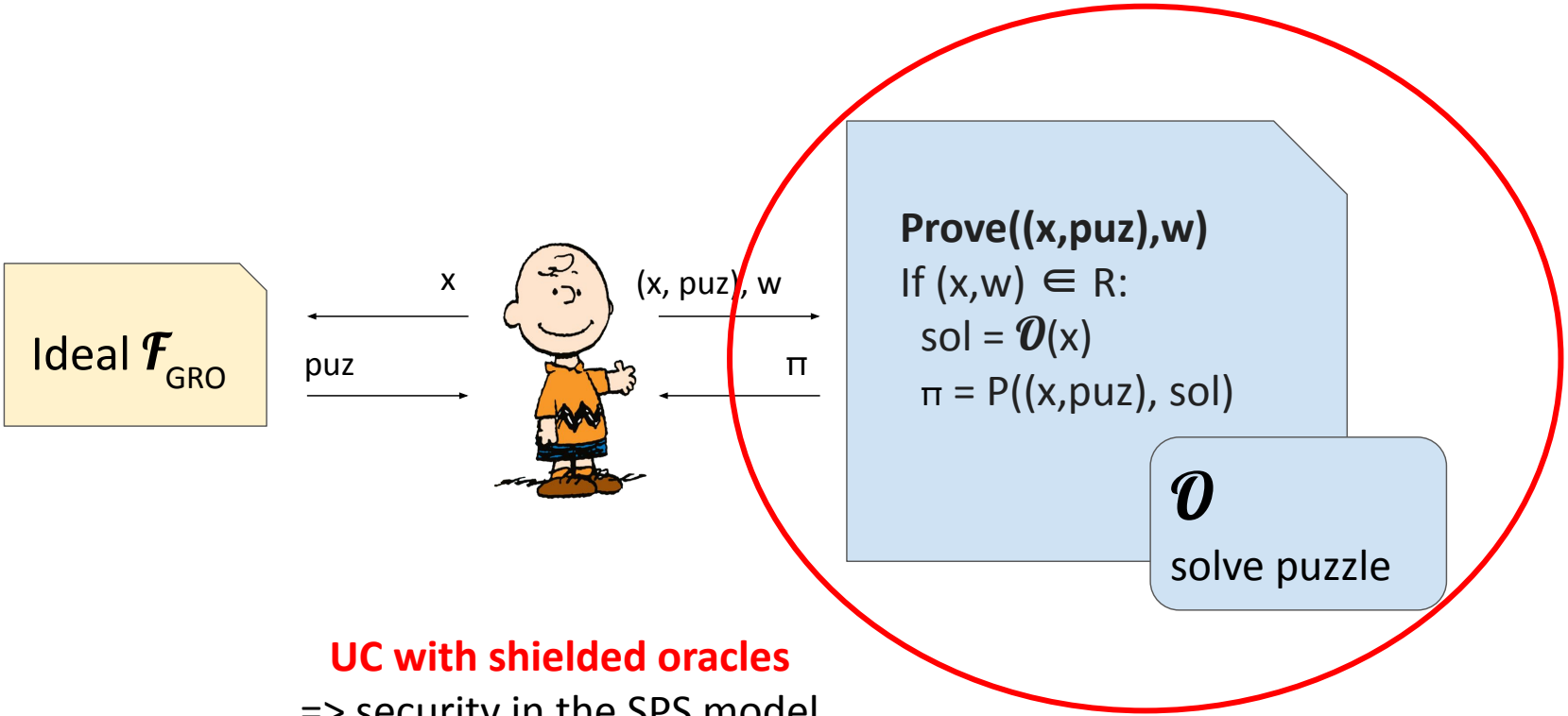


Circumventing the impossibility



UC with shielded oracles

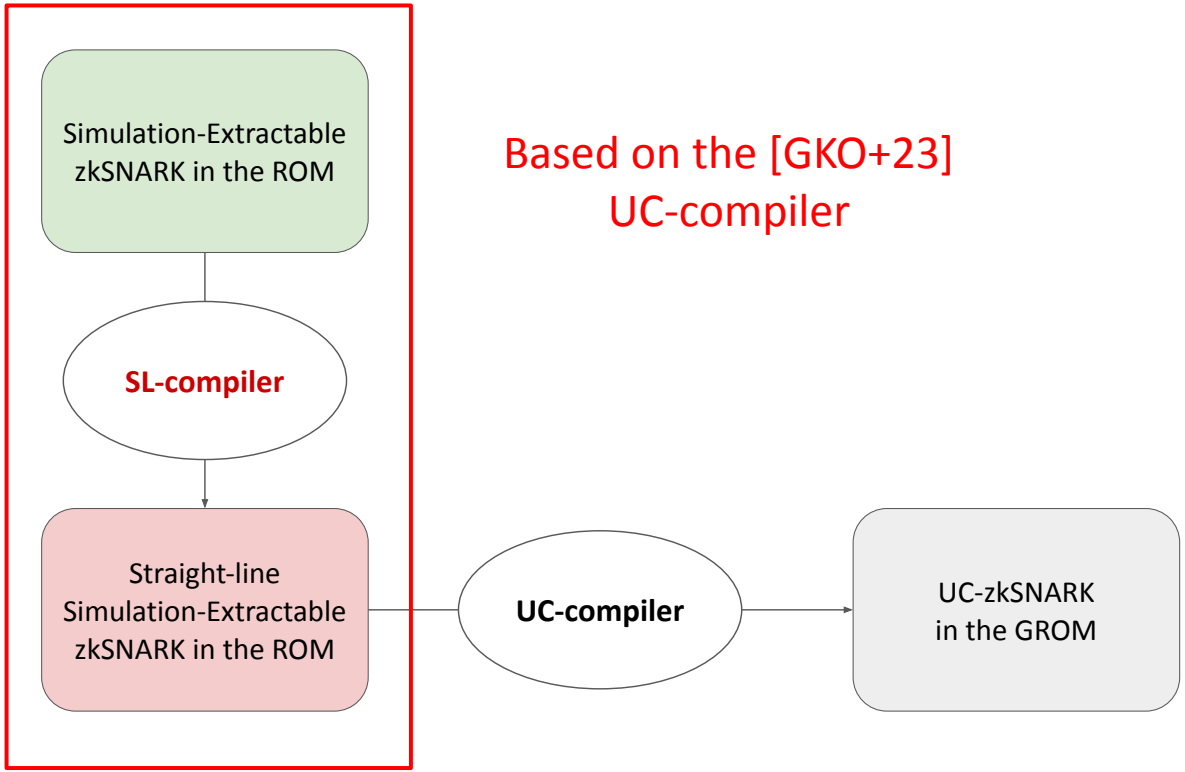
Circumventing the impossibility



UC with shielded oracles

=> security in the SPS model

Two lifting compilers



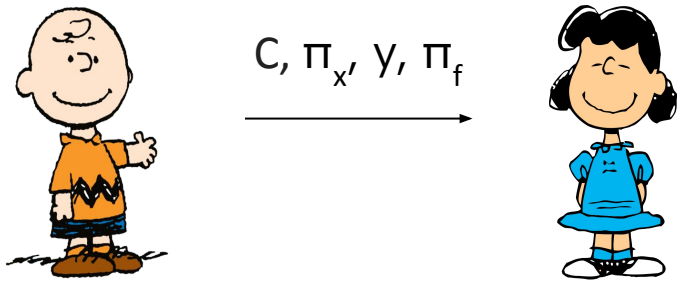
Based on the [GKO+23]
UC-compiler

The [GKO+23] compiler

TL;DR Fischlin + random point evaluation of a polynomial encoding the witness

The [GKO+23] compiler

TL;DR Fischlin + random point evaluation of a polynomial encoding the witness



1. f = Encode(w)
2. C = Com(f)
3. π_x = Prove “(x,w) $\in$ R and C = Com(f)”
4. π_f = Prove “f(z)=y”
5. RO(π_f) = 0000…

The [GKO+23] compiler

TL;DR Fischlin + random point evaluation of a polynomial encoding the witness

C, π_x, y, π_f

1. $f = \text{Encode}(w)$
2. $C = \text{Com}(f)$
3. $\pi_x = \text{Prove}$ “$(x,w) \in R$ and $C = \text{Com}(f)$”
4. $\pi_f = \text{Prove}$ “$f(z)=y$”
5. $\text{RO}(\pi_f) = 0000\ldots$

NIZK with:
1. Unique proofs
2. Evaluation hiding

Our SL-compiler

TL;DR Fischlin + random point evaluation of a polynomial encoding the witness,
based on secret sharing

C, π_x, y, π_f

1. f = Encode(w)
2. C = Com(f)
…

Randomized encoding using additive secret sharing and PKE

Our SL-compiler

TL;DR Fischlin + random point evaluation of a polynomial encoding the witness,
based on secret sharing

C, π_x, y, π_f

1. f = Encode(w)
2. C = Com(f)
…

Randomized encoding using additive secret sharing and PKE

Deterministic polynomial commitment based on Bulletproofs IPA

Thank you

ia.cr/2024/1549

References

[BFKT24] Jan Bobolz, Pooya Farshim, Markulf Kohlweiss, and Akira Takahashi. The brave new world of global generic groups and UC-secure zero-overhead SNARKs. TCC 2024

[CF24] Alessandro Chiesa and Giacomo Fenzi. zkSNARKs in the ROM with unconditional UC-security. TCC 202

[CV22] Michele Ciampi and Ivan Visconti. Efficient NIZK arguments with straight-line simulation and extraction. CANS 2022

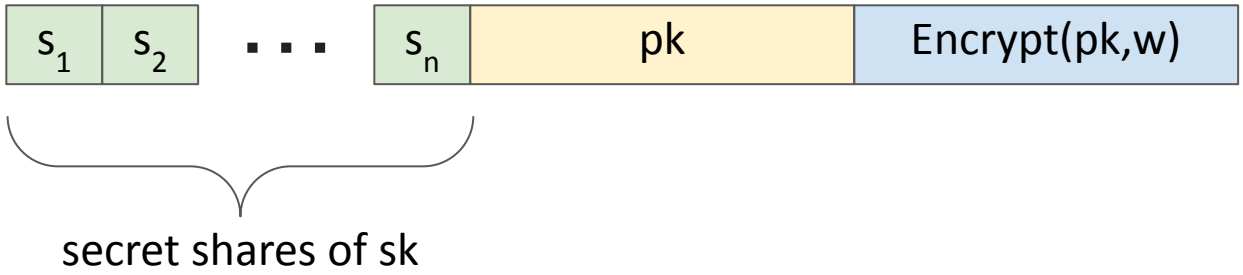
[GKO+23] Chaya Ganesh, Yashvanth Kondi, Claudio Orlandi, Mahak Pancholi, Akira Takahashi, and Daniel Tschudi. Witness-succinct universally-composable SNARKs. EUROCRYPT 2023

[Pas03] Rafael Pass. Simulation in quasi-polynomial time, and its application to protocol composition. EUROCRYPT 2003

Additional slides

Our polynomial encoding

1. (sk,pk) ← PKE.KeyGen()
2. Encode(w):



Credits

All images used in this presentation are being used for educational purposes in accordance with the principles of fair use. The copyright of these images remains with their respective owners. No infringement is intended.