LegoSNARK: Compose ZKPs Simply and Efficiently

Matteo Campanelli
Dario Fiore
Anaïs Querol

Instituto IMDEA Software

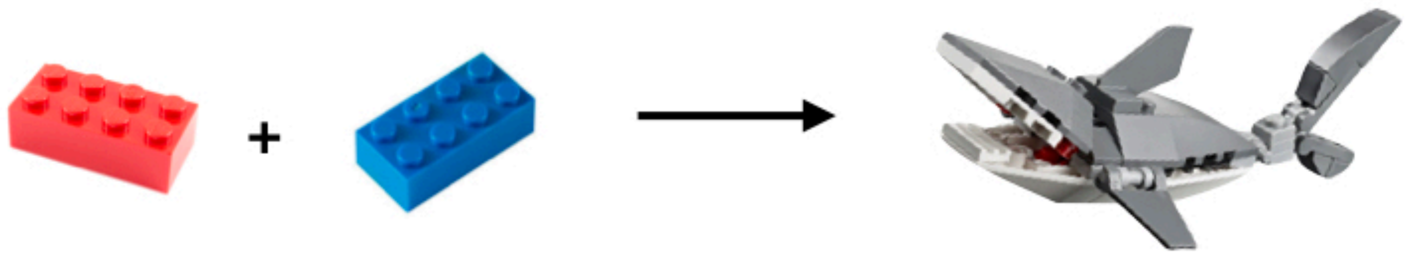
eprint: ia.cr/2019/142

API (soon): github.com/imdea-software/legosnark

Modular SNARKs

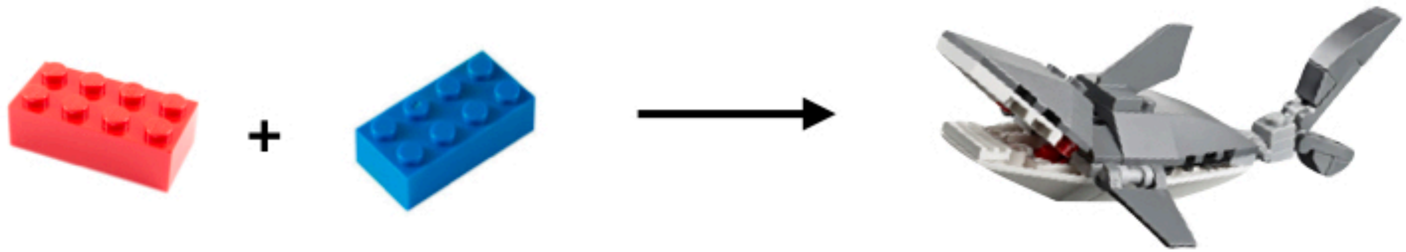
Modular SNARKs

Modularity (roughly)**:**



Modular SNARKs

Modularity (roughly)**:**



Our focus: Non-Interactive and Succinct arguments

Modularity:
“*Why?*” and "*What Exactly?*”

Through two Tales…

A Tale of Efficiency

A Tale of Efficiency

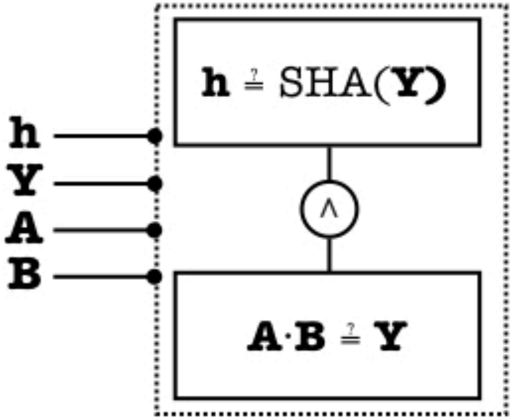
$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$

"$\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A} \cdot \mathbf{B})$"

A Tale of Efficiency

$$R(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$

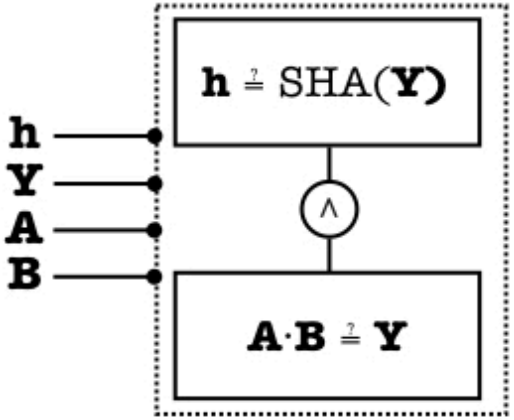
$$\text{“}\mathbf{h} \stackrel{?}{=} \text{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$



A Tale of Efficiency

$$R(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$

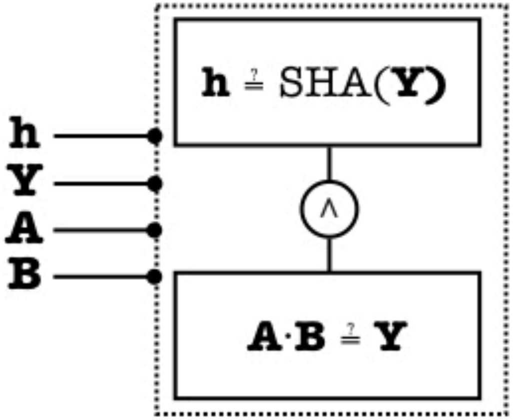
$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$



A Tale of Efficiency

$$R(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$

$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A} \cdot \mathbf{B})\text{”}$$



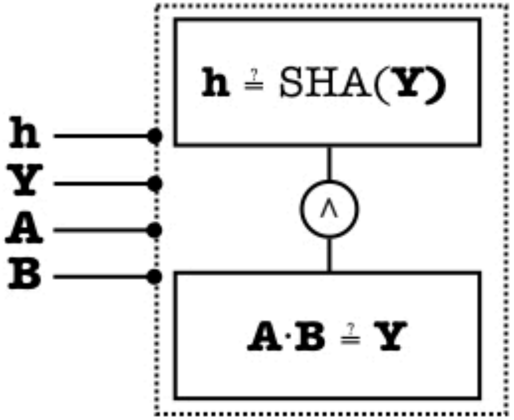
ZKP1

ZKP2

A Tale of Efficiency

$$R(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$

$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$



Bool

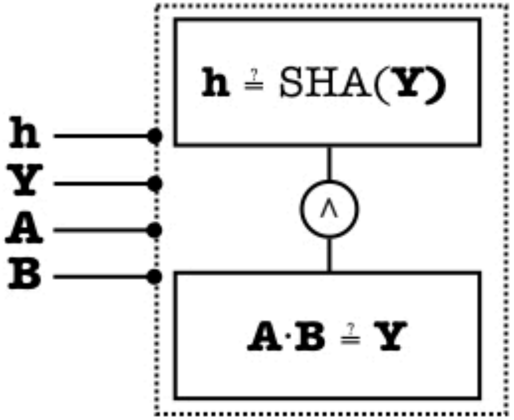
Algebra

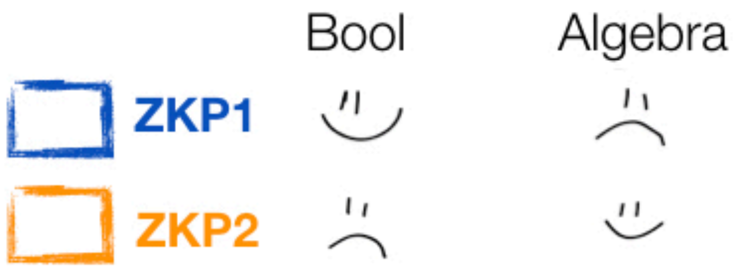
ZKP1

ZKP2

A Tale of Efficiency

$$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$
$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$

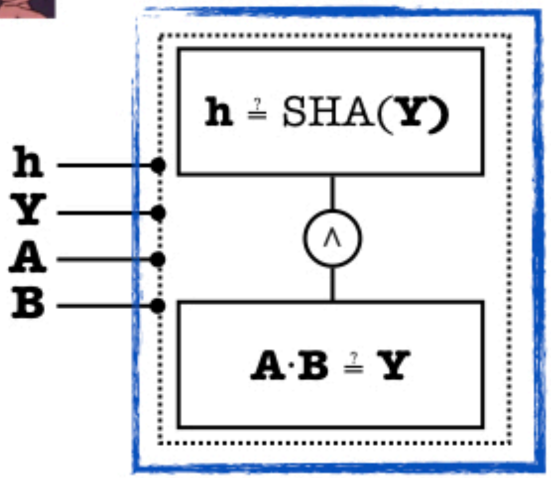




A Tale of Efficiency

$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$

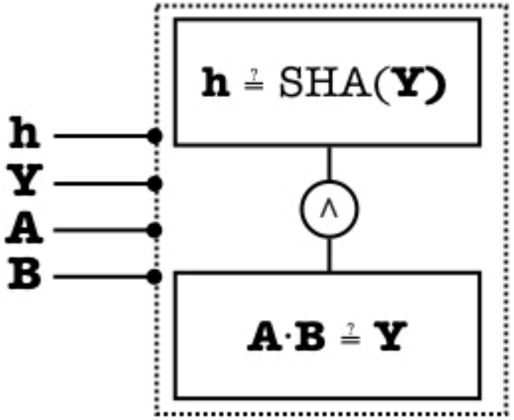
$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$

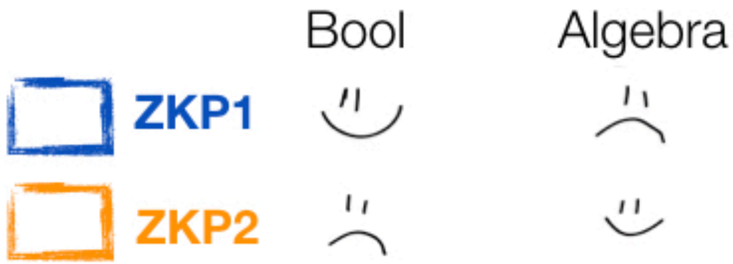


A Tale of Efficiency

$$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$

$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$

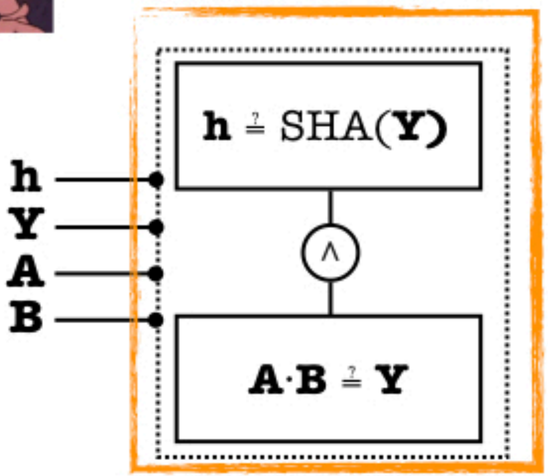




A Tale of Efficiency

$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$

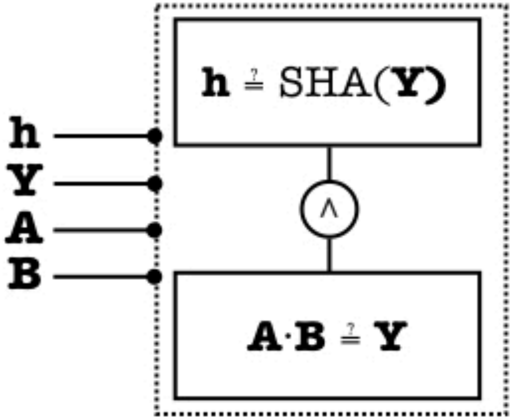
"$\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})$"



A Tale of Efficiency

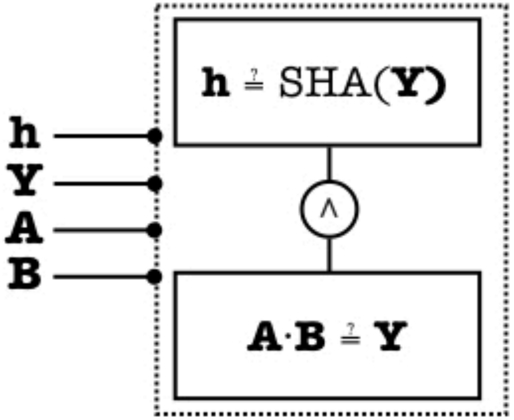
$R(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$

$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$



A Tale of Efficiency

$$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) := $$
$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$

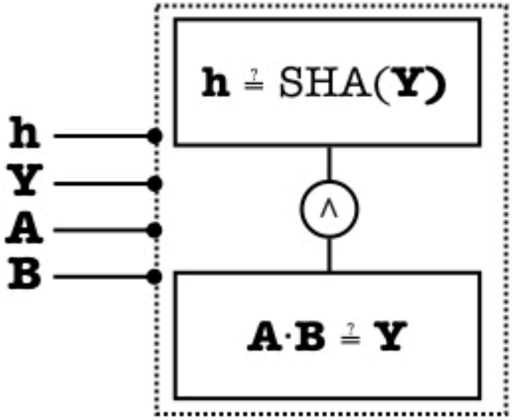


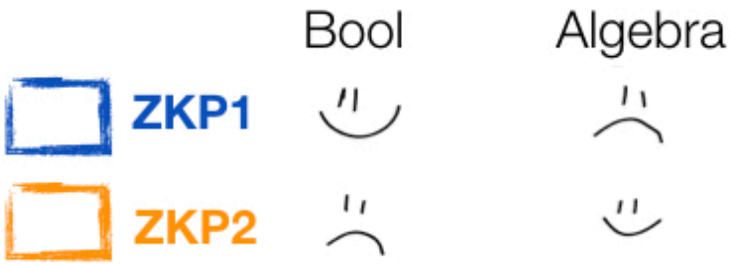
This is suboptimal!

A Tale of Efficiency

$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$

“$\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})$”





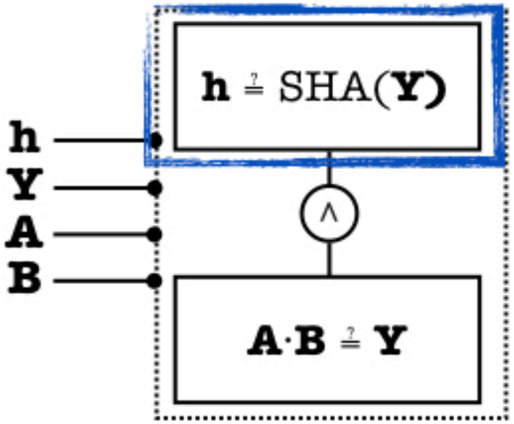
This is suboptimal!

Q: Can’t we get the best of both worlds?

A Tale of Efficiency

$$R(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$

$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$



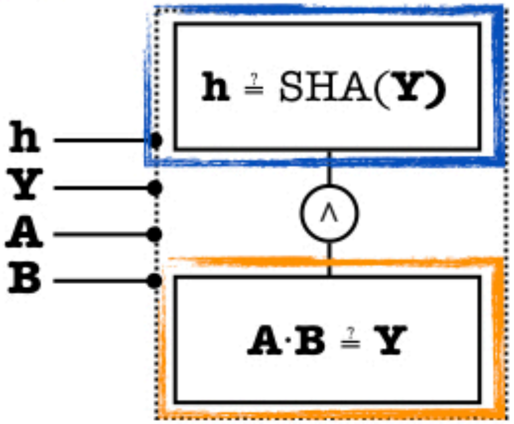
	Bool	Algebra
ZKP1	:)	:(
ZKP2	:(	:)

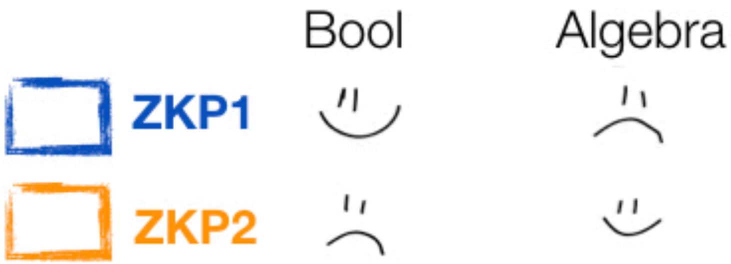
This is suboptimal!

Q: Can’t we get the best of both worlds?

A Tale of Efficiency

$$R(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$
$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$



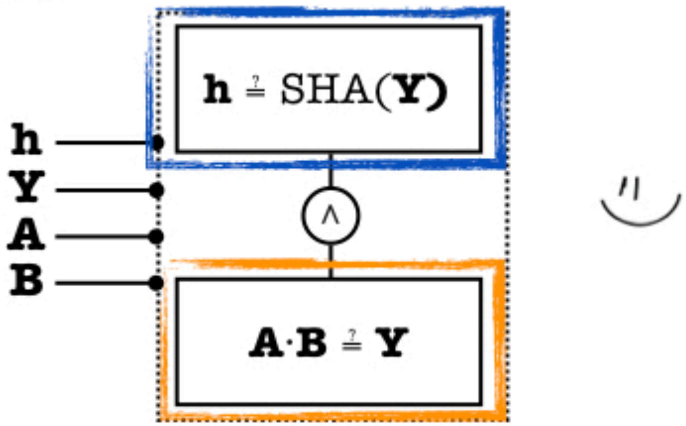


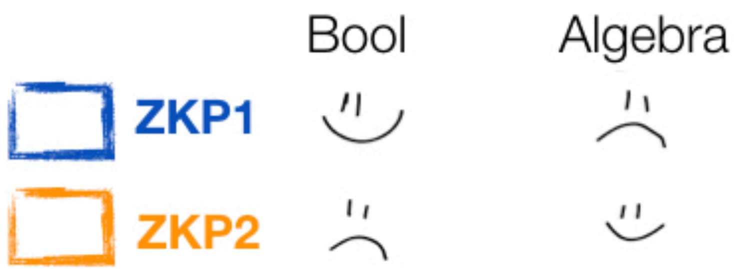
This is suboptimal!

Q: Can’t we get the best of both worlds?

A Tale of Efficiency

$$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$
$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$



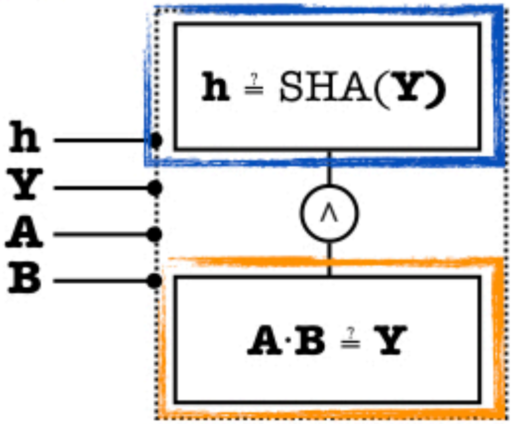


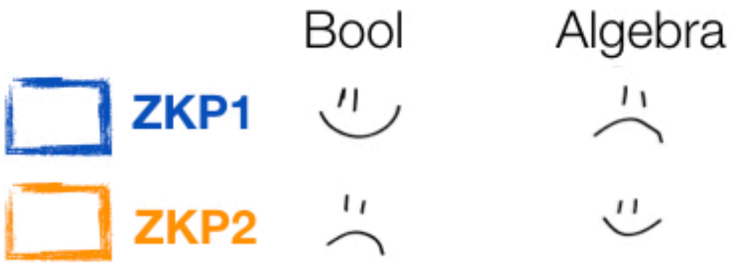
This is suboptimal!

Q: Can’t we get the best of both worlds?

A Tale of Efficiency

$$\mathrm{R}(\mathbf{h}, \mathbf{A}, \mathbf{B}) :=$$
$$\text{“}\mathbf{h} \stackrel{?}{=} \mathrm{SHA}(\mathbf{A}\cdot\mathbf{B})\text{”}$$





This is suboptimal!

Q: Can’t we get the best of both worlds?

“Splittable" relations are common
(e.g. select+aggregate in a DB, polynomial evaluation)

A Tale of Simplicity

There once were two bearded wizards…

A Tale of Simplicity

There once were two bearded wizards…

Digression: UNIX's simplicity

Do Not *Write* Programs; *Glue* Them Together.

UNIX shell commands are simple.

grep '*pattern*' *file*	Prints lines matching a pattern

head -n 5 *file*	prints first five lines from a file
tail *file*	prints last lines from a file

sort -u *file*	sorts and return unique lines
uniq -c *file*	filters adjacent repeated lines

cut -f 1,3 *file*	retrieves data from selected columns in a tab-delimited file

Do Not *Write* Programs; *Glue* Them Together.

UNIX shell commands are simple.

grep '*pattern*' *file*	Prints lines matching a pattern

head -n 5 *file*	prints first five lines from a file
tail *file*	prints last lines from a file

sort -u *file*	sorts and return unique lines
uniq -c *file*	filters adjacent repeated lines

cut -f 1,3 *file*	retrieves data from selected columns in a tab-delimited file

Problem: find the top 5 most frequent first names in a digital phone book.

```
[…]
Mr. Groucho Marx 612345783
Ms. Emmy Noether 612567105
[…]
```

Do Not *Write* Programs; *Glue* Them Together.

UNIX shell commands are simple.

grep '*pattern*' *file*	Prints lines matching a pattern

head -n 5 *file*	prints first five lines from a file
tail *file*	prints last lines from a file

sort -u *file*	sorts and return unique lines
uniq -c *file*	filters adjacent repeated lines

cut -f 1,3 *file*	retrieves data from selected columns in a tab-delimited file

Problem: find the top 5 most frequent first names in a digital phone book.

```
[…]
Mr. Groucho Marx 612345783
Ms. Emmy Noether 612567105
[…]
```

Solution:

```
cut -d ' ' -f 2 book.txt | sort | uniq -c | sort -rn | head -5
```

Do Not *Write* SNARKs;
Glue Them Together.

UNIX Philosophy

Do Not *Write* SNARKs; *Glue* Them Together.

UNIX Philosophy

- Write programs that do one thing and do it well.

Do Not *Write* SNARKs; *Glue* Them Together.

UNIX Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.

Do Not *Write* SNARKs;
Glue Them Together.

UNIX Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

Do Not *Write* SNARKs; *Glue* Them Together.

UNIX Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

LegoSNARK Philosophy

Do Not *Write* SNARKs; *Glue* Them Together.

UNIX Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

LegoSNARK Philosophy

- Write SNARKs that do one thing and do it well and fast.

Do Not *Write* SNARKs; *Glue* Them Together.

UNIX Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

LegoSNARK Philosophy

- Write SNARKs that do one thing and do it well and fast.
- Write SNARKs to work together.

Do Not *Write* SNARKs; *Glue* Them Together.

UNIX Philosophy	LegoSNARK Philosophy
• Write programs that do one thing and do it well.	• Write SNARKs that do one thing and do it well and fast.
• Write programs to work together.	• Write SNARKs to work together.
• Write programs to handle text streams, because that is a universal interface.	• Write SNARKs to handle commitments, because that is a universal ZK interface.

Do Not *Write* <u>SNARKs</u>; *Glue* Them Together.

UNIX Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

The pipeline |

```
cut -d ' ' -f 2 book.txt | sort | uniq -c |
sort -rn | head -5
```

LegoSNARK Philosophy

- Write SNARKs that do one thing and do it well and fast.
- Write SNARKs to work together.
- Write SNARKs to handle commitments, because that is a universal ZK interface.

Do Not *Write* SNARKs; *Glue* Them Together.

UNIX Philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

The pipeline |

```
cut -d ‘ ’ -f 2 book.txt | sort | uniq -c |
sort -rn | head -5
```

LegoSNARK Philosophy

- Write SNARKs that do one thing and do it well and fast.
- Write SNARKs to work together.
- Write SNARKs to handle commitments, because that is a universal ZK interface.

We need a “cryptographic pipeline”

(should preserve soundness, ZK, succinctness, etc.)

Our Pipeline:
Commit-and-Prove (CP)

Our Pipeline: Commit-and-Prove (CP)

Commitments.

$$\mathbf{u} \longrightarrow \mathbf{u}^{✉}$$

Our Pipeline: Commit-and-Prove (CP)

Commitments.

Important:
can be opened in only one way.

Our Pipeline:
Commit-and-Prove (CP)

Commitments.

Important:
can be opened in only one way.

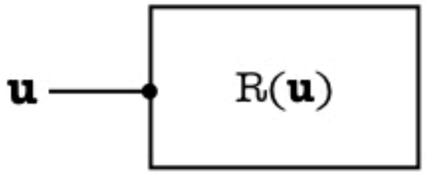
CP-SNARKs.

Our Pipeline:
Commit-and-Prove (CP)

Commitments.

Important:
can be opened in only one way.

CP-SNARKs.

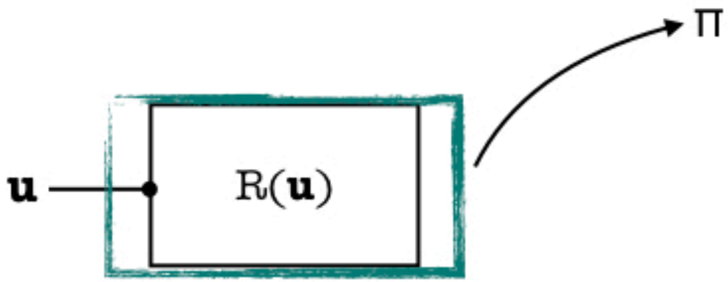


Our Pipeline: Commit-and-Prove (CP)

Commitments.

Important:
can be opened in only one way.

CP-SNARKs.



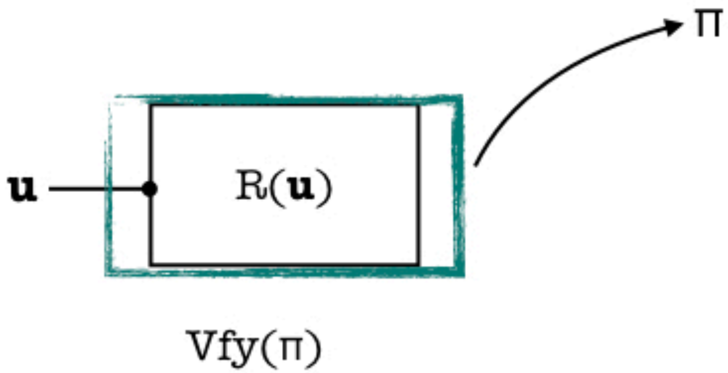
"Relation **R** holds for some input **u**"

Our Pipeline: Commit-and-Prove (CP)

Commitments.

Important:
can be opened in only one way.

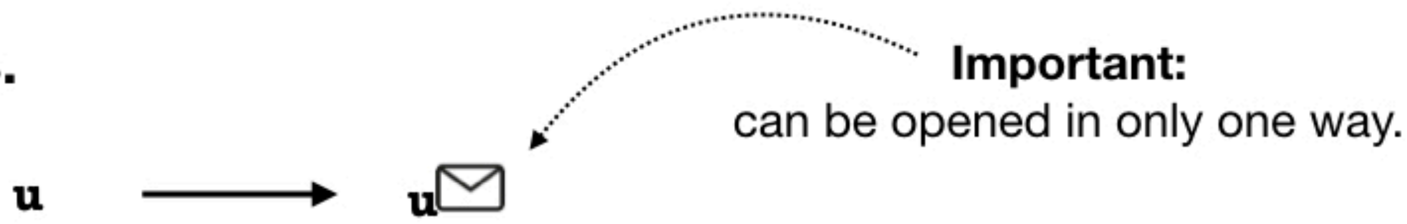
CP-SNARKs.



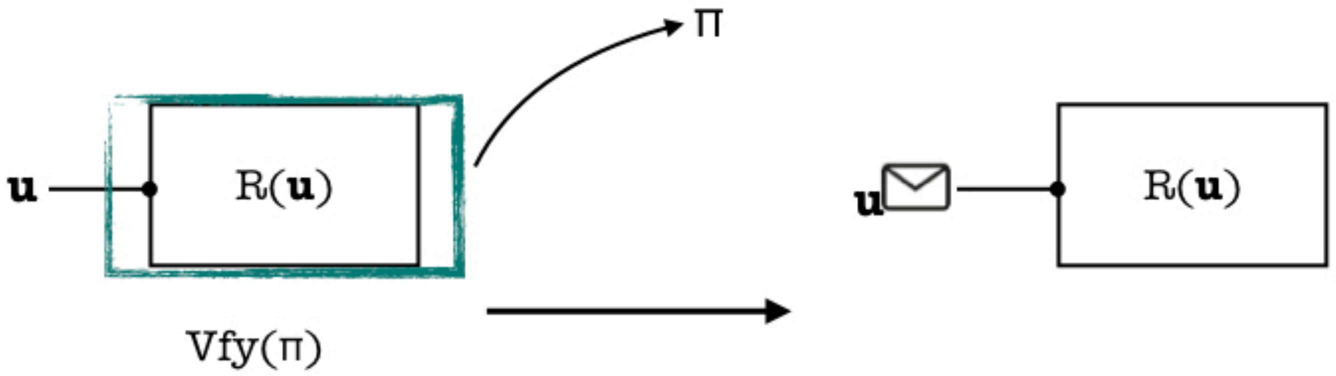
"Relation **R** holds for some input **u**"

Our Pipeline: Commit-and-Prove (CP)

Commitments.



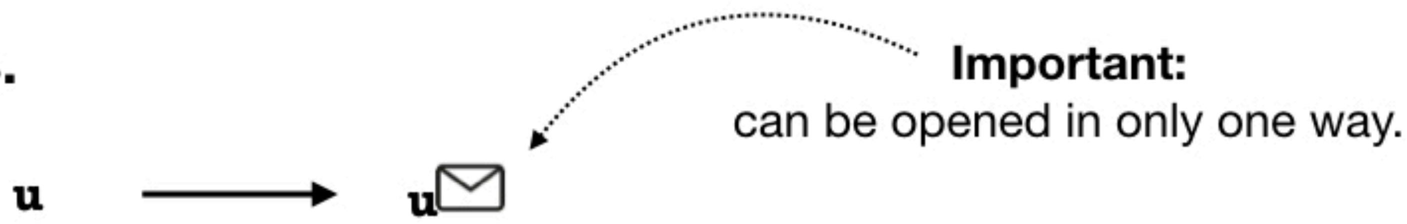
CP-SNARKs.



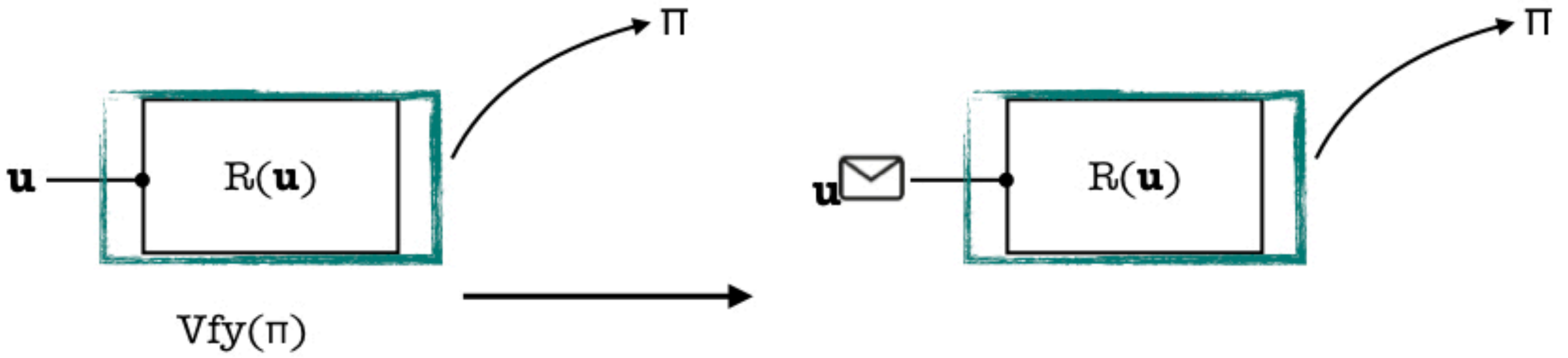
"Relation **R** holds for some input **u**"

Our Pipeline: Commit-and-Prove (CP)

Commitments.



CP-SNARKs.



"Relation **R** holds for some input **u**"

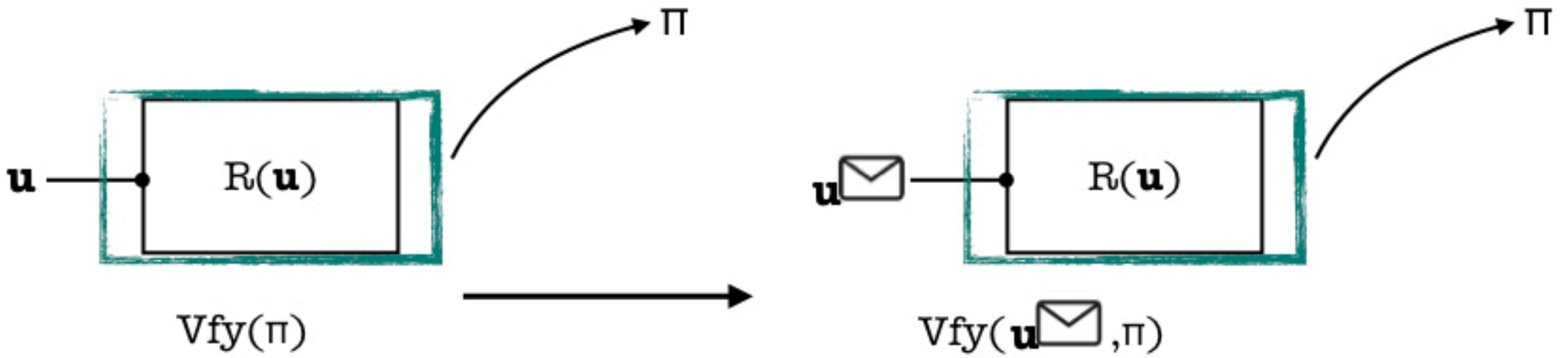
"Relation **R** holds for what is inside **u**"

Our Pipeline: Commit-and-Prove (CP)

Commitments.

Important:
can be opened in only one way.

CP-SNARKs.



"Relation **R** holds for some input **u**"

"Relation **R** holds for what is inside **u**"

Our Pipeline: Commit-and-Prove (CP) (cont.)

Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

$\mathrm{Vfy}(\mathbf{u}$✉$, \pi 1)$ $\mathrm{Vfy}(\mathbf{u}$✉$, \pi 2)$

Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

Vfy($\mathbf{u}$,π1)   Vfy($\mathbf{u}$,π2)

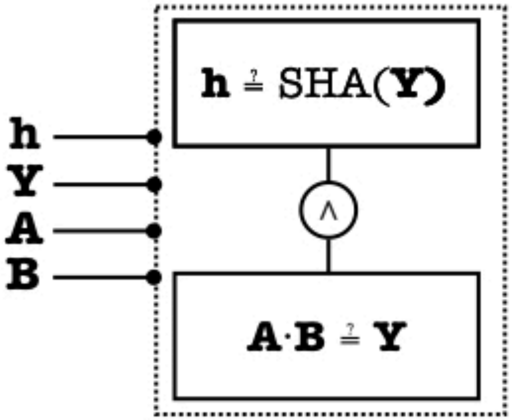
That the two properties hold for the same $\mathbf{u}$.

Our Pipeline:
Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

$\mathrm{Vfy}(\mathbf{u}, \pi 1)$ $\quad$ $\mathrm{Vfy}(\mathbf{u}, \pi 2)$

That the two properties hold for the same **u.**

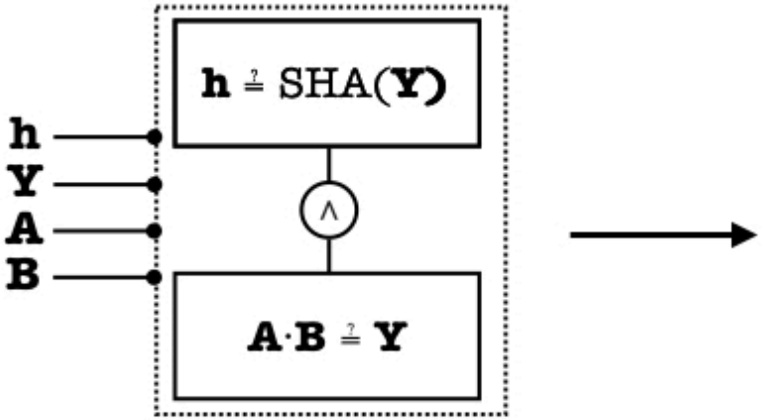


Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

Vfy(**u**✉,π1)   Vfy(**u**✉,π2)

That the two properties hold for the same **u.**

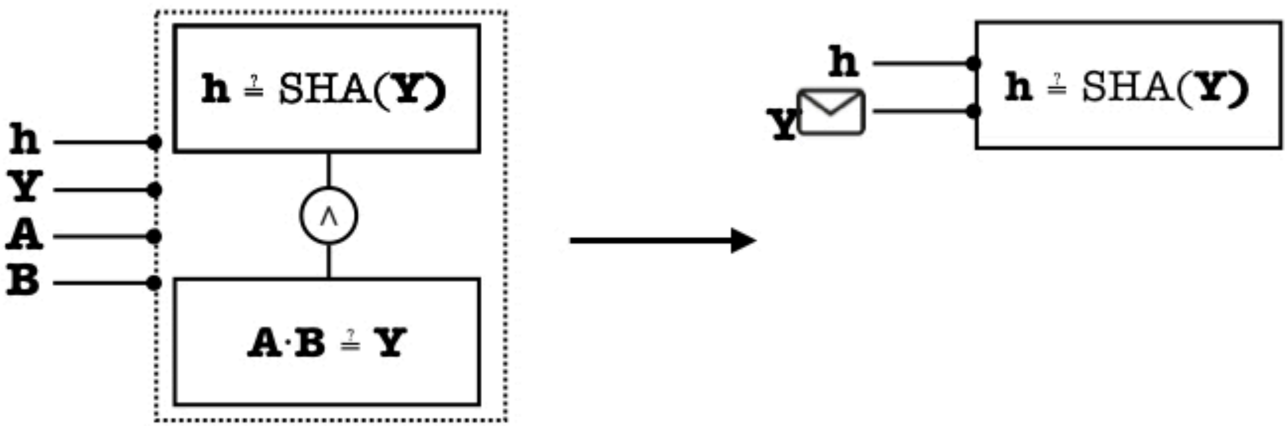


Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

$\mathrm{Vfy}(\mathbf{u}, \pi 1) \qquad \mathrm{Vfy}(\mathbf{u}, \pi 2)$

That the two properties hold for the same **u.**

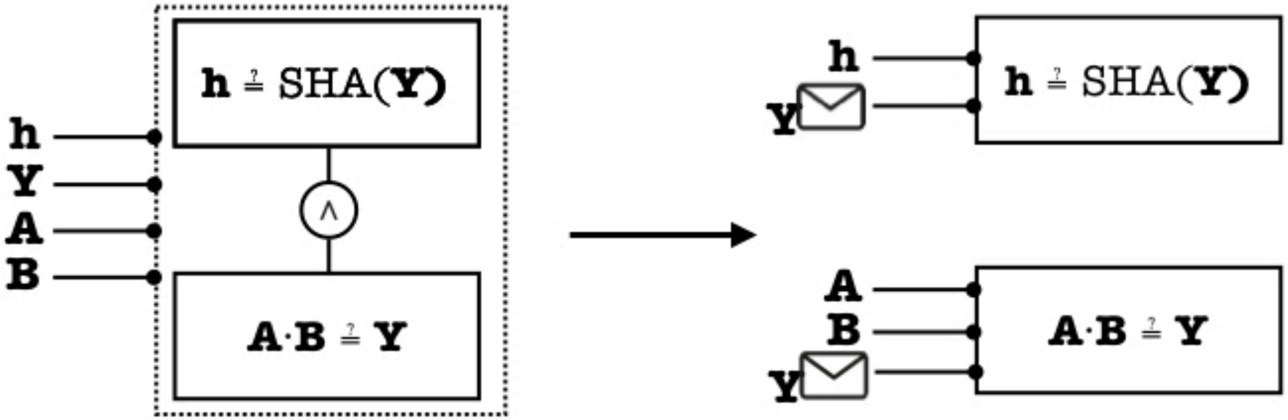


Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

Vfy(**u**, π1)   Vfy(**u**, π2)

That the two properties hold for the same **u.**

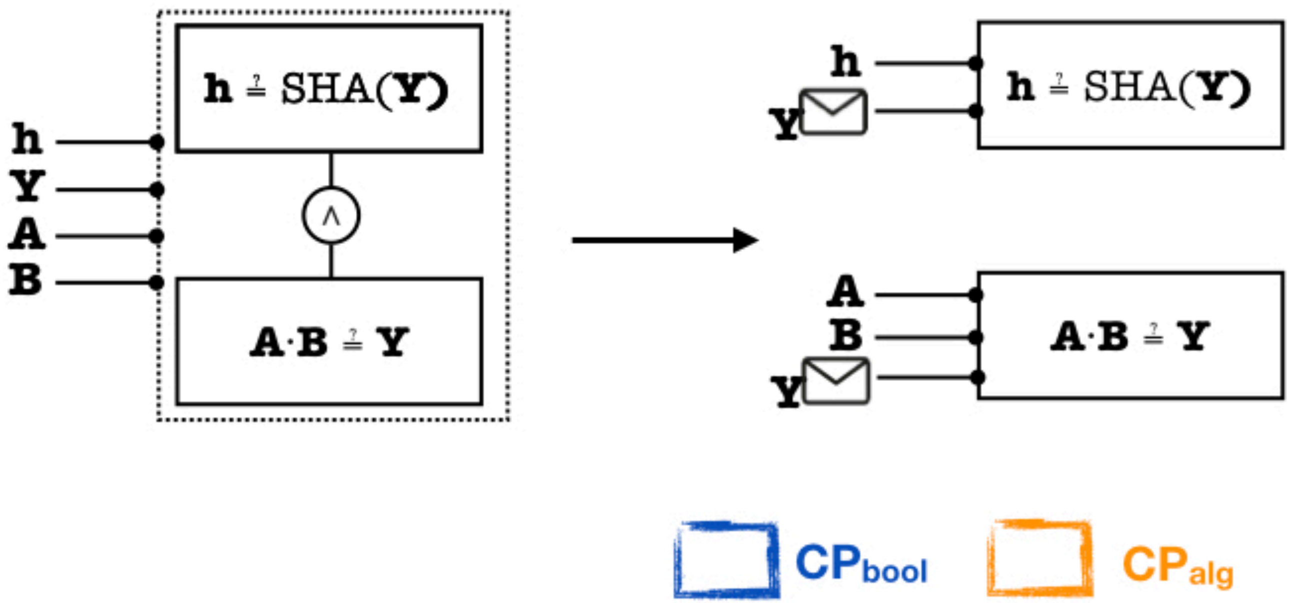


Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

Vfy($\mathbf{u}$✉,π1) Vfy($\mathbf{u}$✉,π2)

That the two properties hold for the same **u.**

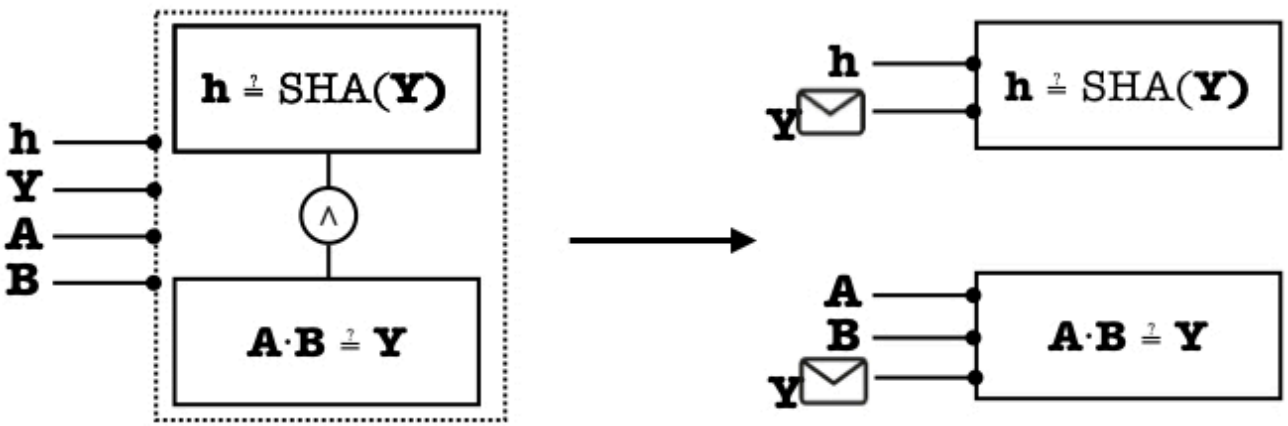


Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

Vfy(**u**✉,π1)   Vfy(**u**✉,π2)

That the two properties hold for the same **u.**



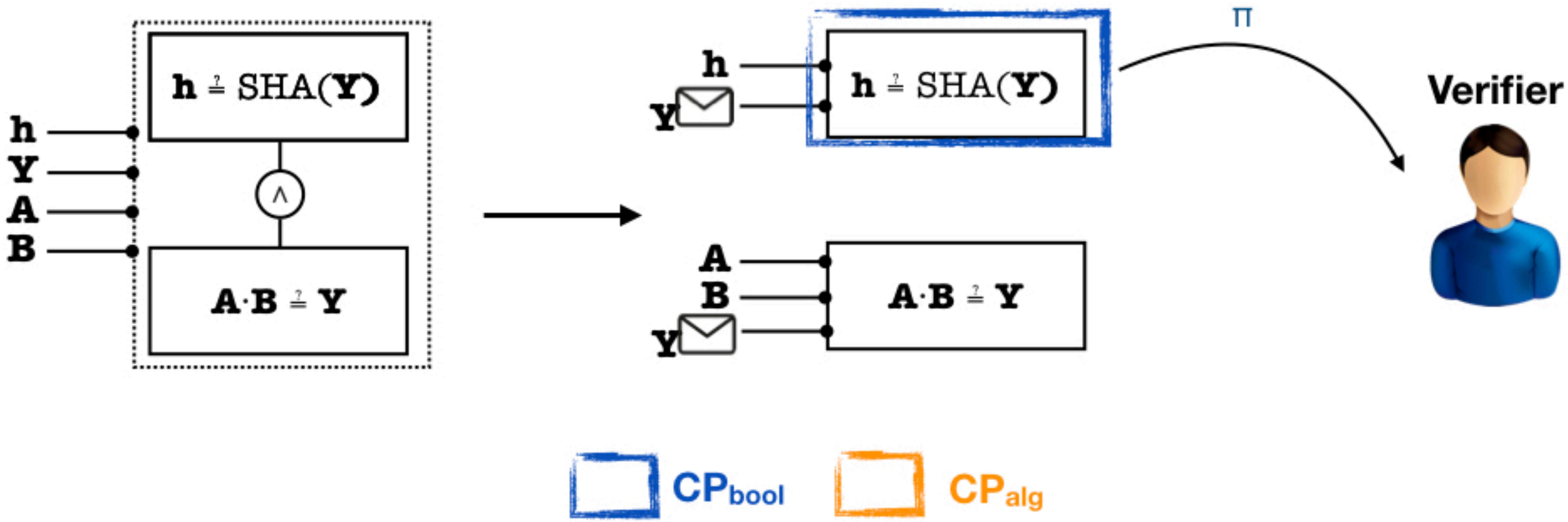
Verifier

CP_{bool} CP_{alg}

Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

That the two properties hold for the same **u.**

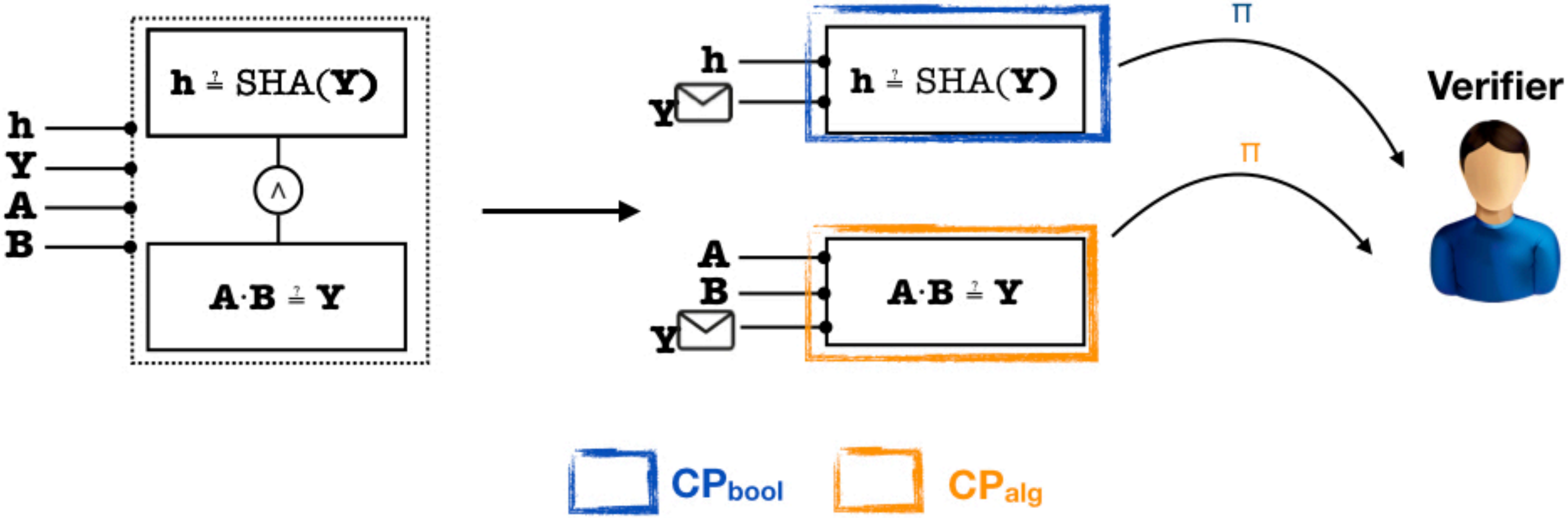


Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

Vfy(**u**,π1)   Vfy(**u**,π2)

That the two properties hold for the same **u.**

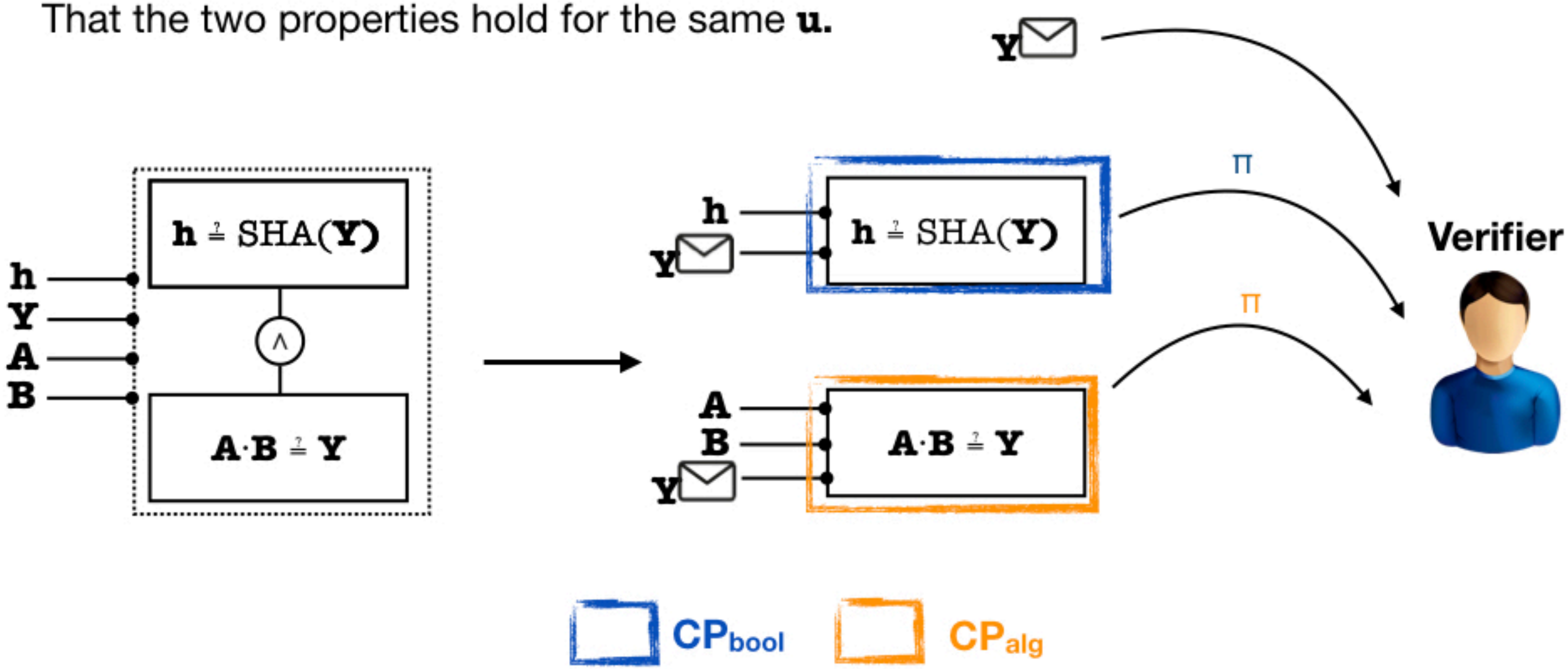


Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

Vfy(**u**✉,π1) Vfy(**u**✉,π2)

That the two properties hold for the same **u.**

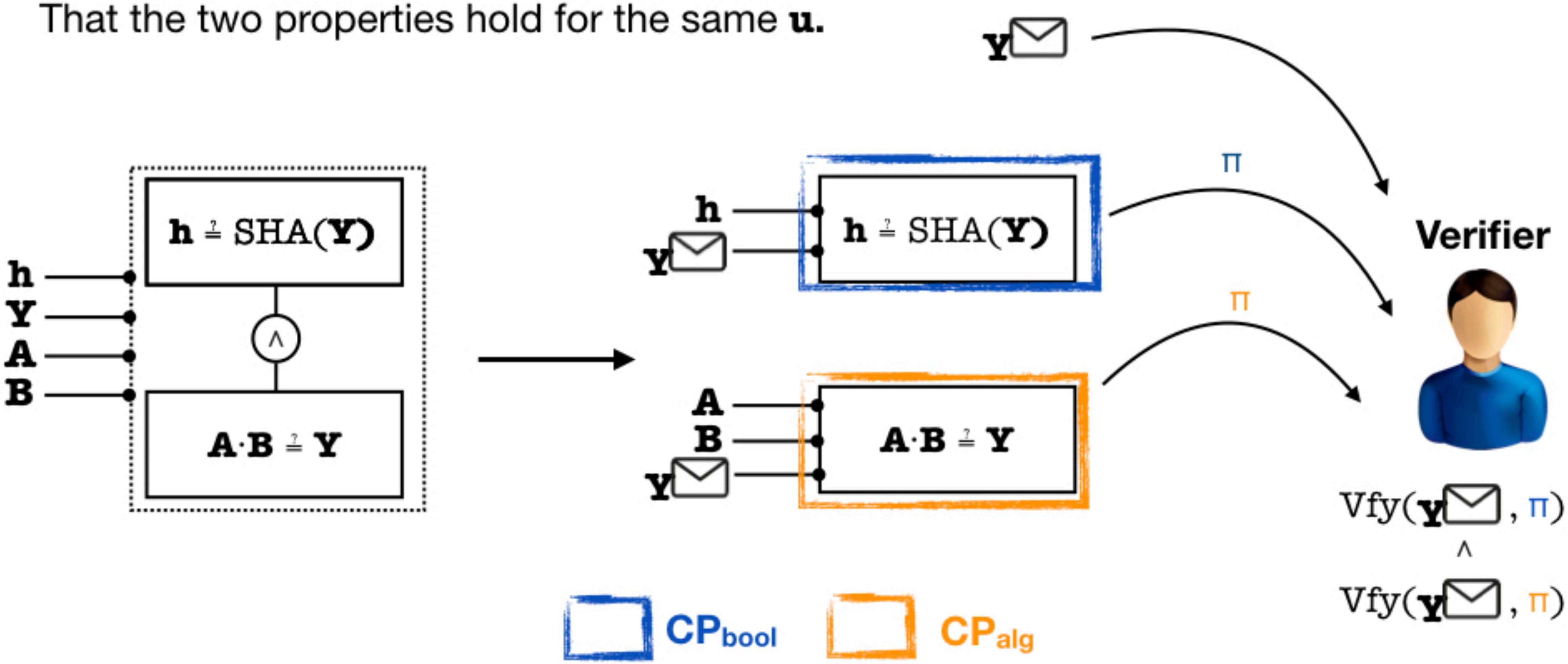


Our Pipeline: Commit-and-Prove (CP) (cont.)

Warm-up: What do we learn from verifying two CP-proofs on the same commitment?

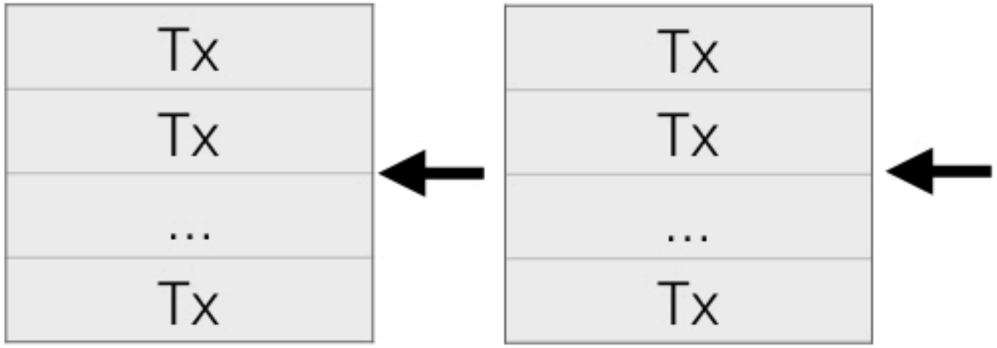
Vfy(**u**✉,π1)   Vfy(**u**✉,π2)

That the two properties hold for the same **u**.

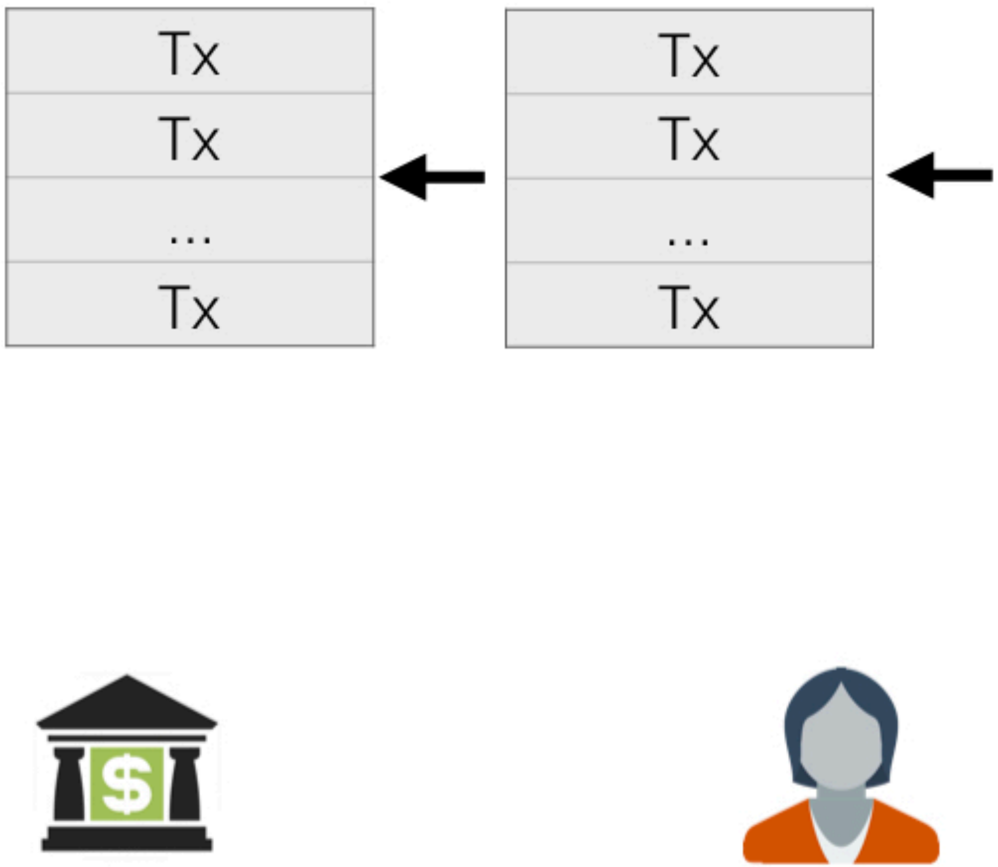


Application:
Commit-(ahead-of-time)-and-Prove

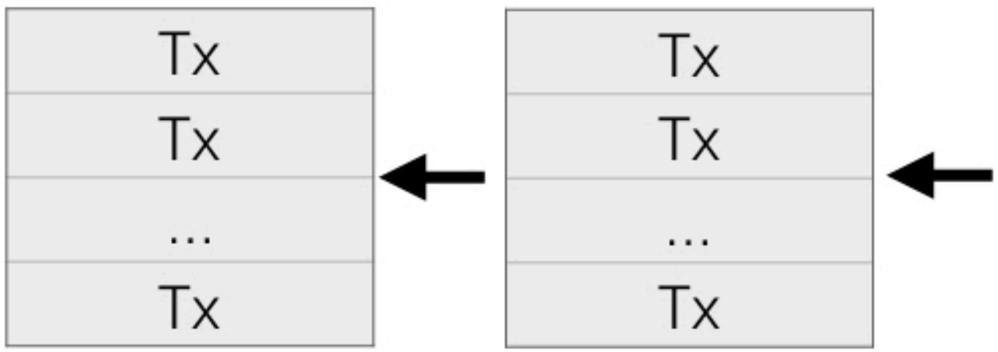
Application:
Commit-(ahead-of-time)-and-Prove

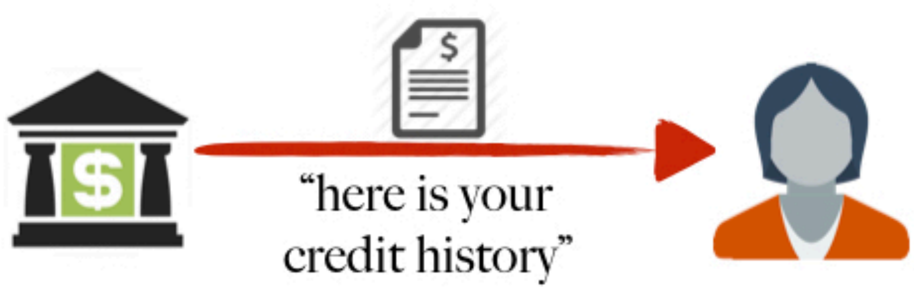


Application:
Commit-(ahead-of-time)-and-Prove

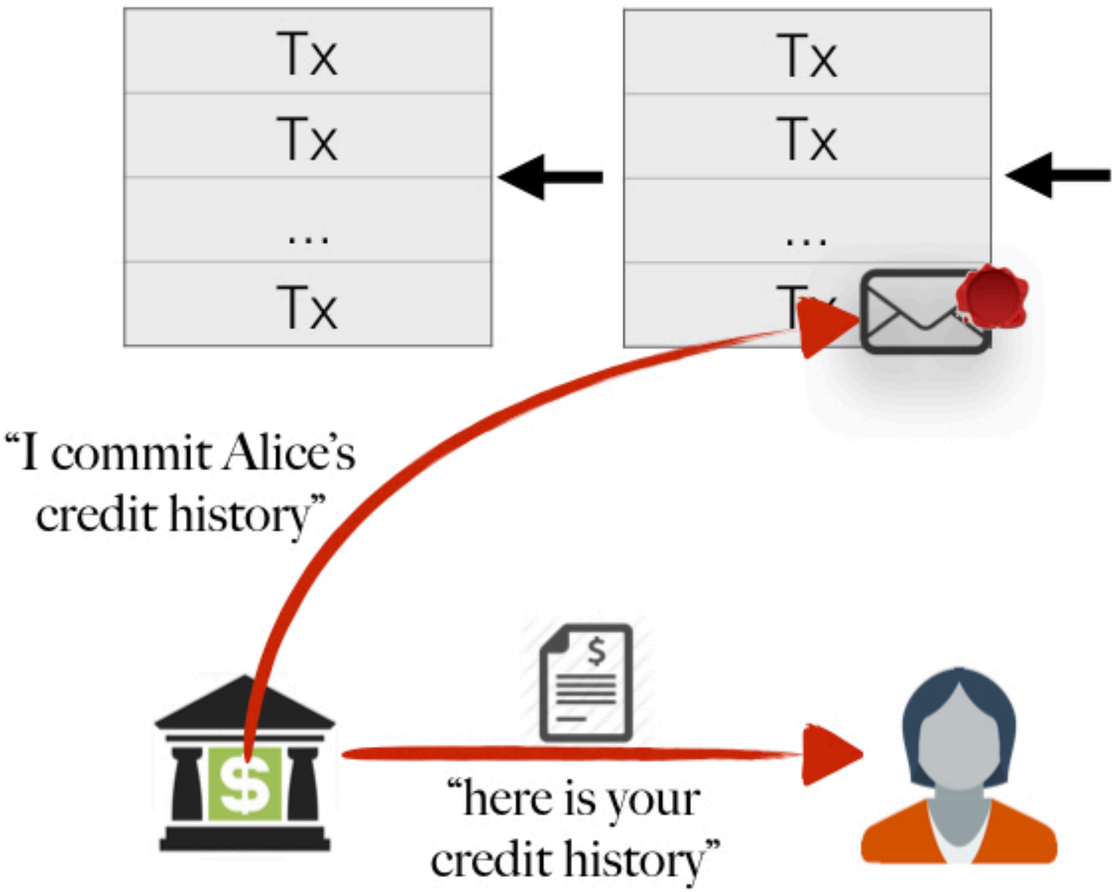


Application:
Commit-(ahead-of-time)-and-Prove

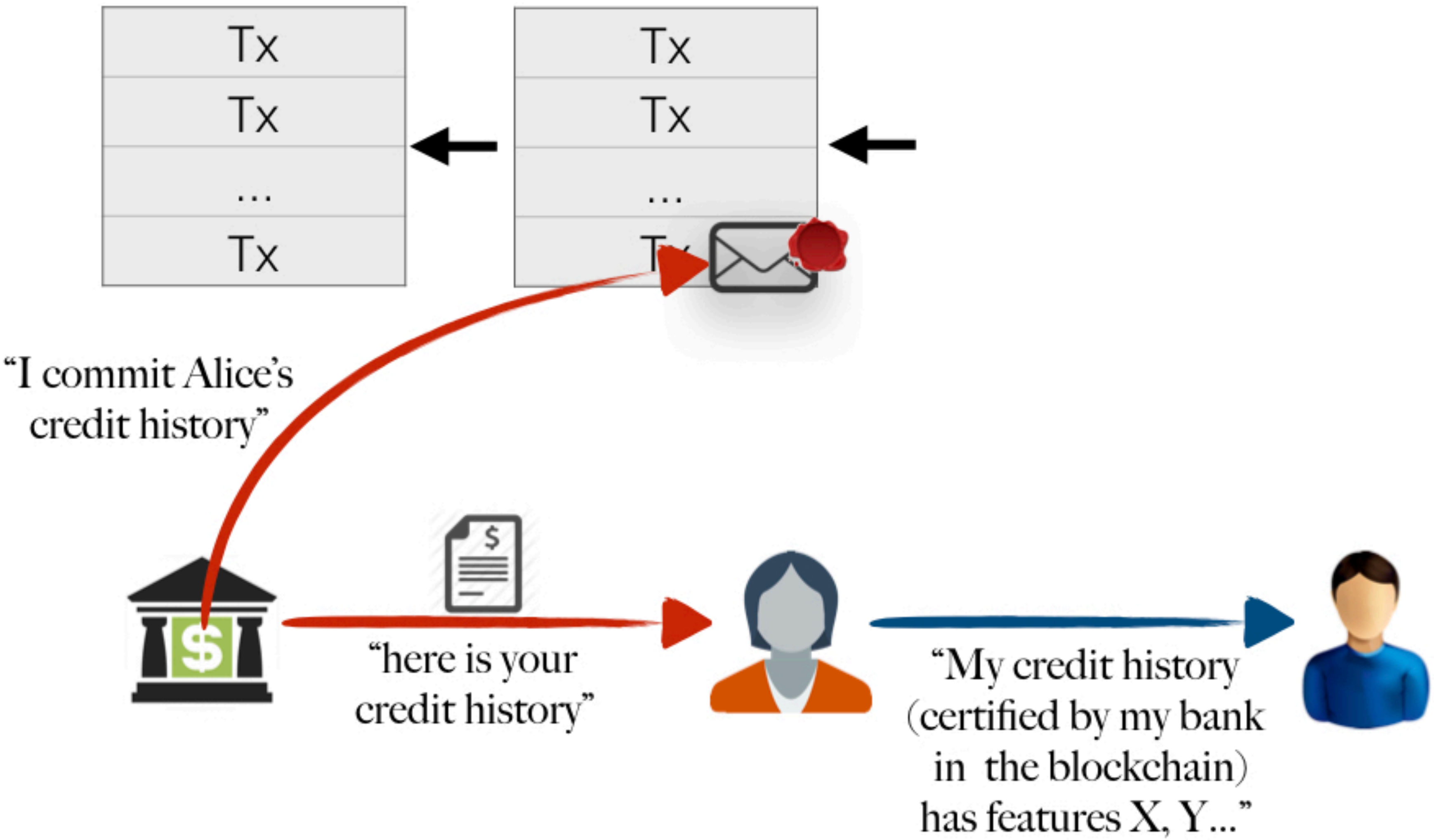




Application:
Commit-(ahead-of-time)-and-Prove



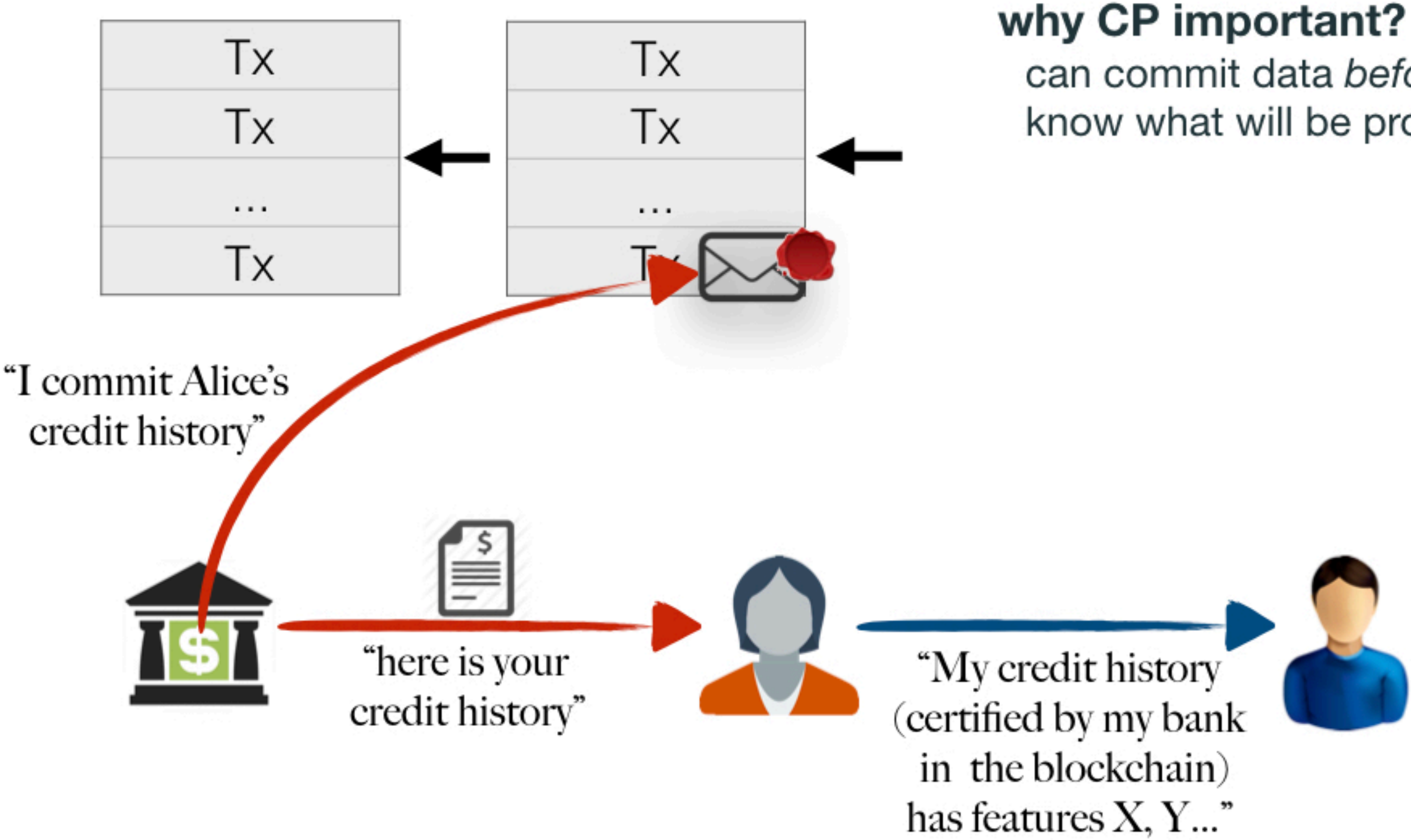
Application: Commit-(ahead-of-time)-and-Prove



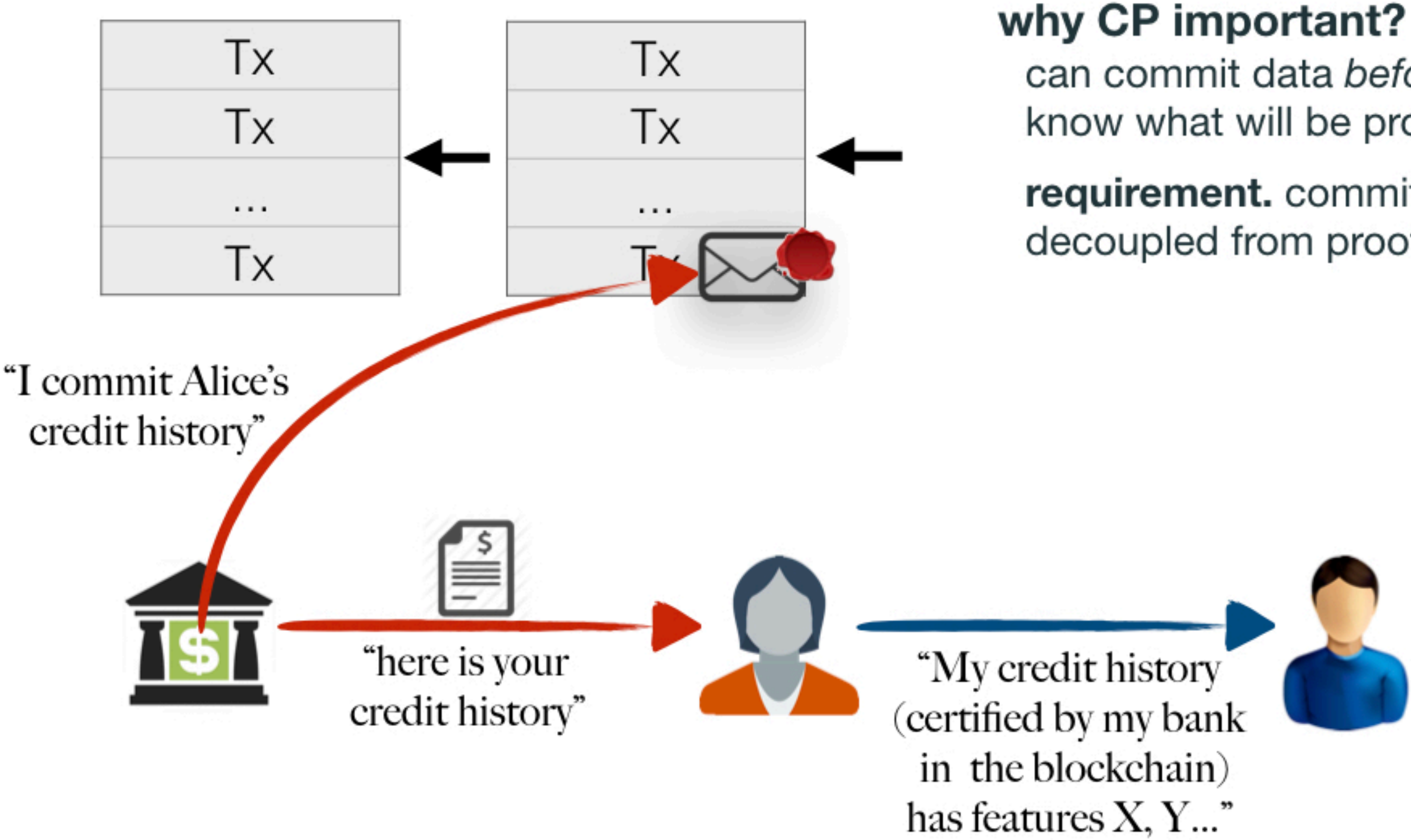
Application:
Commit-(ahead-of-time)-and-Prove

why CP important?

can commit data *before* we know what will be proved



Application: Commit-(ahead-of-time)-and-Prove

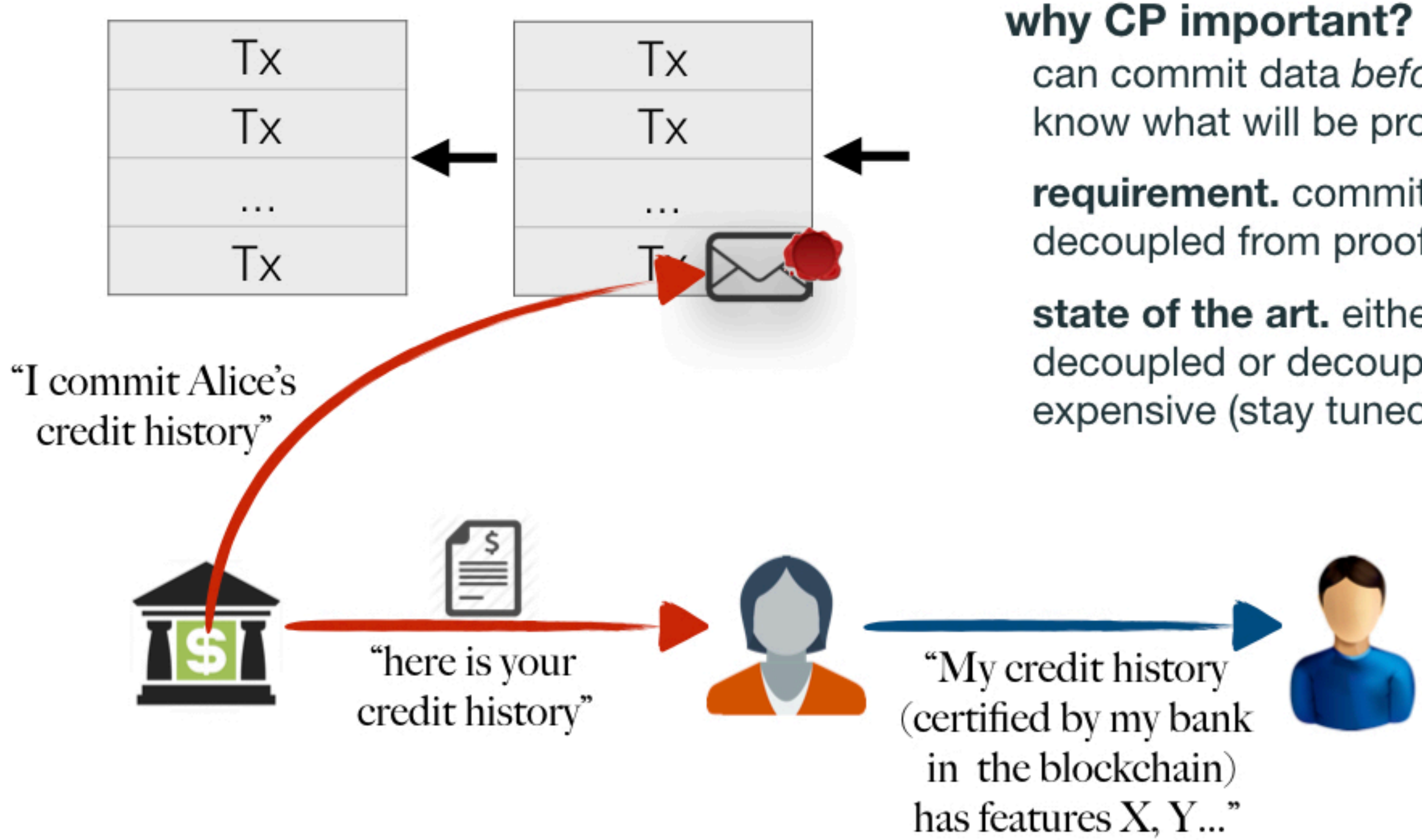


why CP important?

can commit data *before* we know what will be proved

requirement. commitments decoupled from proof system

Application: Commit-(ahead-of-time)-and-Prove



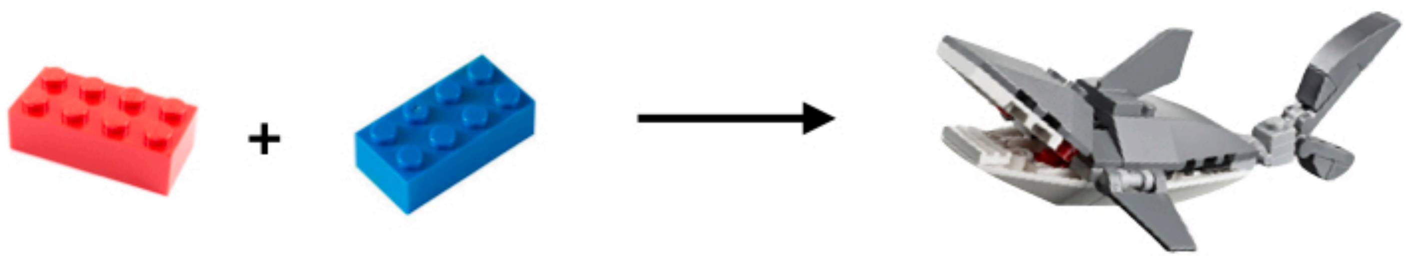
why CP important?

can commit data *before* we know what will be proved

requirement. commitments decoupled from proof system

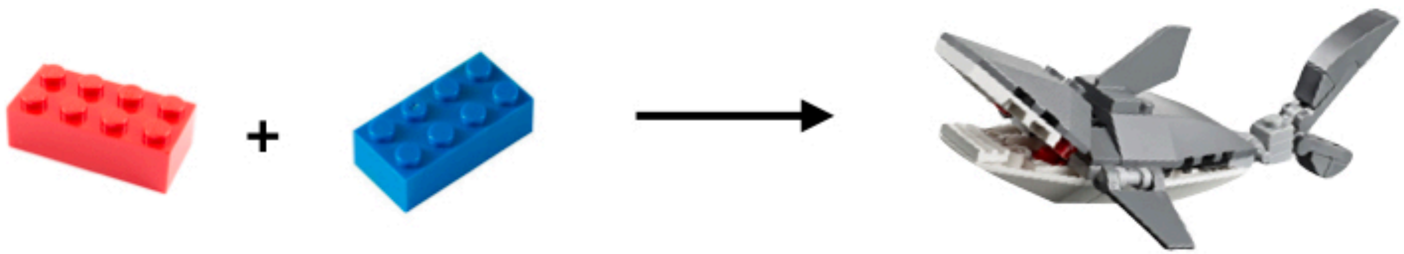
state of the art. either not decoupled or decoupling is expensive (stay tuned)

So Far



So Far

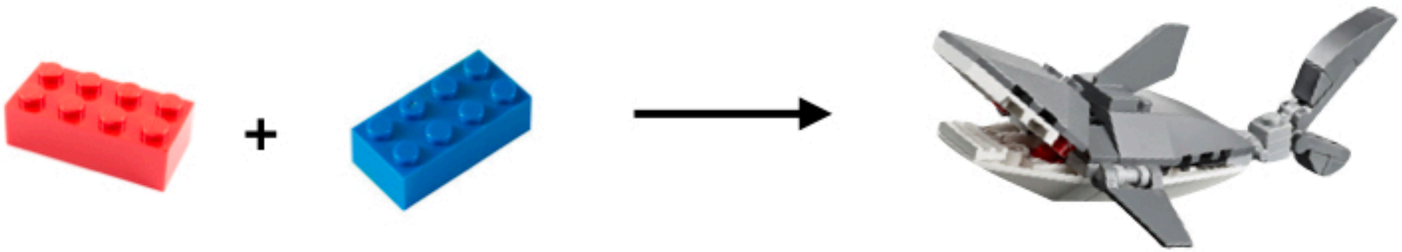
We want to make SNARKs **modular** for better **efficiency** and better **design**



So Far

We want to make SNARKs **modular** for better **efficiency** and better **design**

To compose SNARKs: make them use **commitments as a "glue"** (commit-and-prove, or CP, SNARKs)



This Talk

This Talk

Building CP gadgets and its challenges

This Talk

Building CP gadgets and its challenges

Applications and Efficiency of LegoSNARK

This Talk

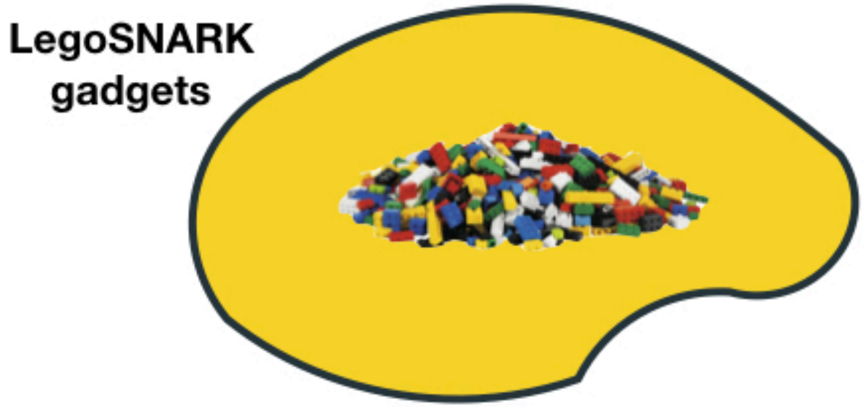
Building CP gadgets
and its challenges

Applications and Efficiency
of LegoSNARK

LegoSNARK in practice:
a C++ API

Building a Pool of Gadgets

Building Gadgets

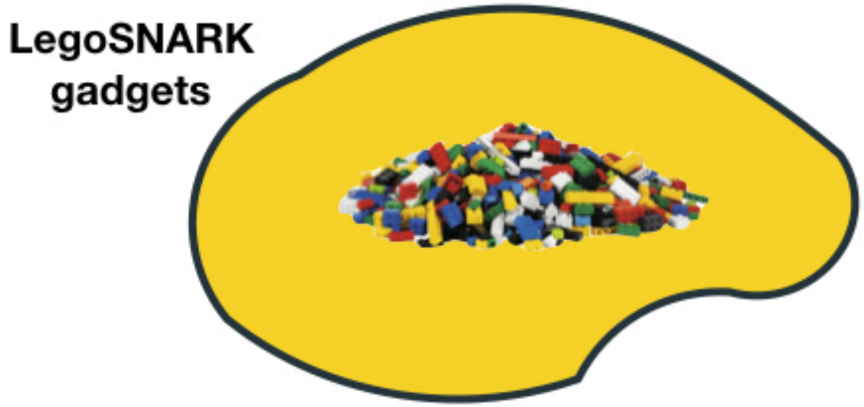


Building Gadgets

LegoSNARK
gadgets

1. **Import** existing zkSNARKs in the framework

Building Gadgets

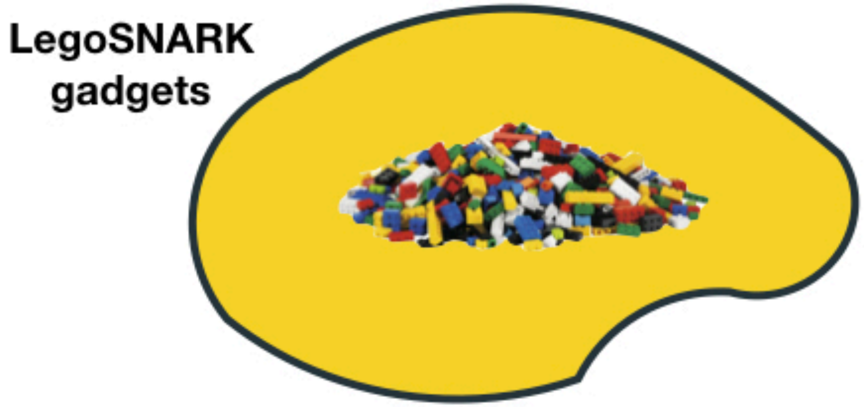


1. **Import** existing zkSNARKs in the framework

   don't want to throw away years of research…

   + may want general-purpose systems as fallback option

Building Gadgets



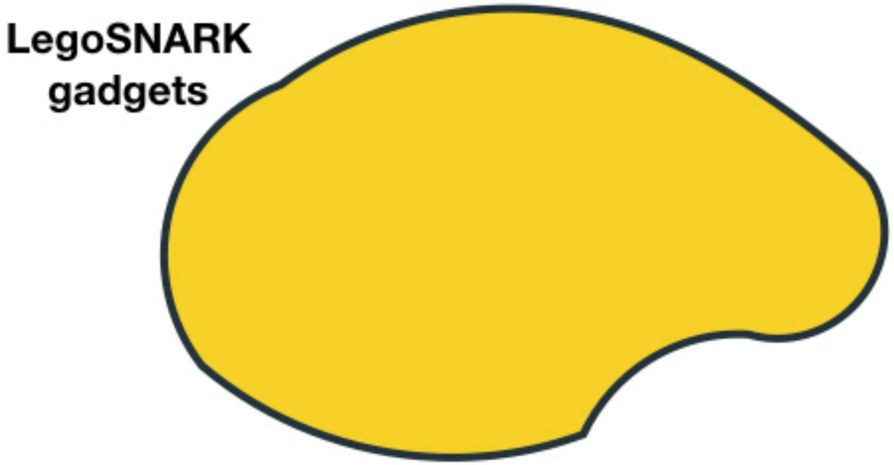
1. **Import** existing zkSNARKs in the framework

   don't want to throw away years of research…
   + may want general-purpose systems as fallback option

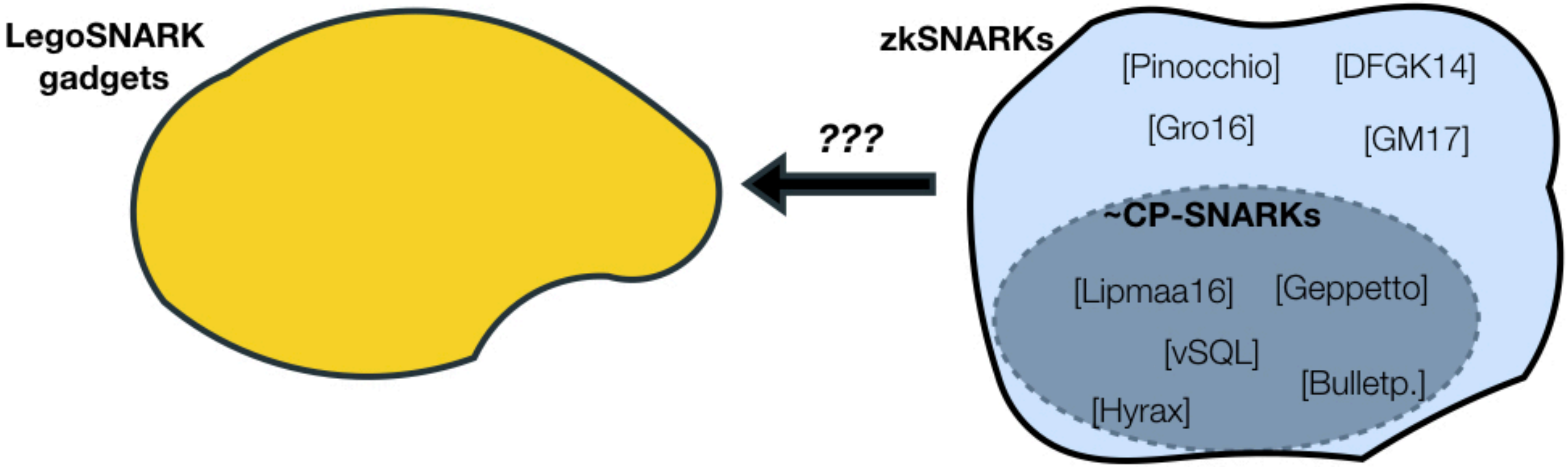
2. **Construct** new CP-SNARKs

   exploit the power of specialization

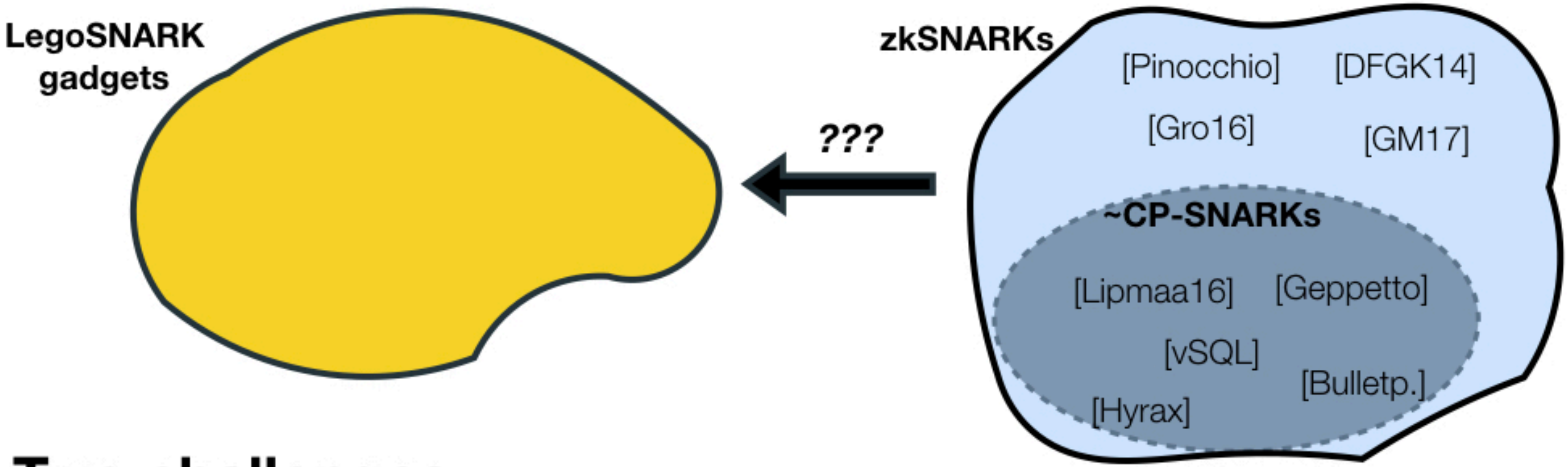
1. Import Existing zkSNARKs



1. Import Existing zkSNARKs

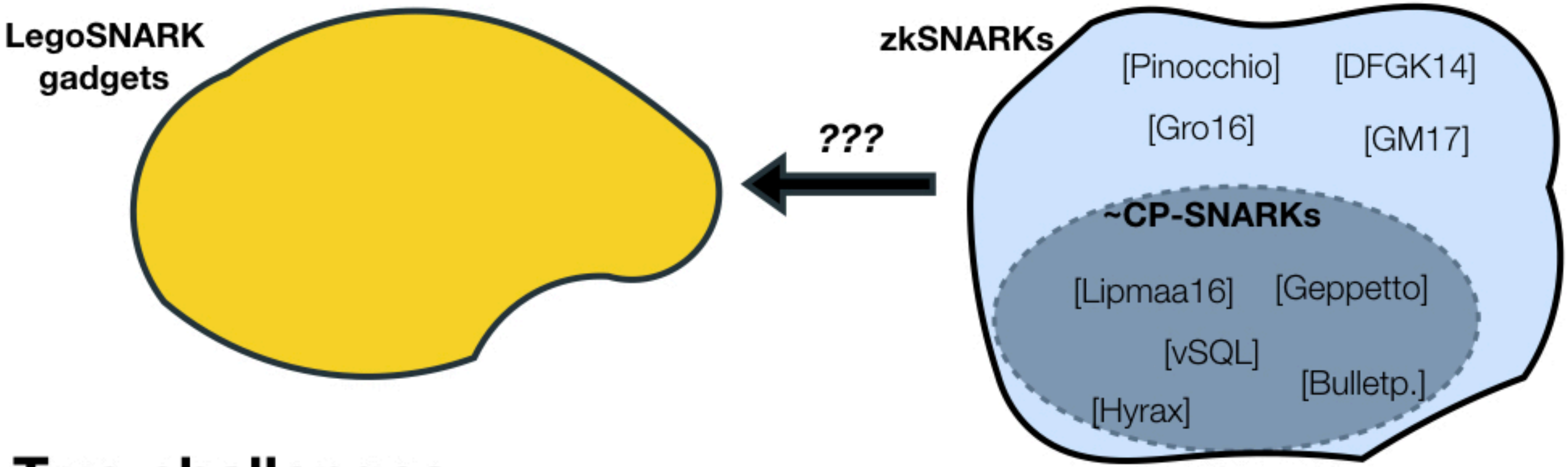


1. Import Existing zkSNARKs



Two challenges:

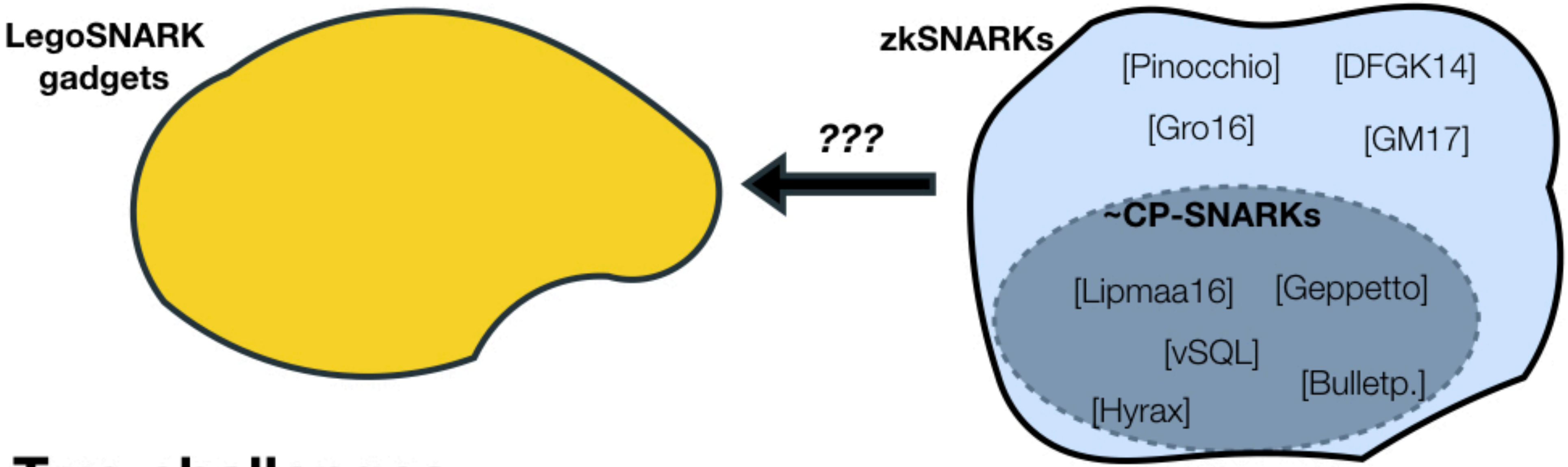
1. Import Existing zkSNARKs



Two challenges:

A. **Many Popular zkSNARKs are not CP**

1. Import Existing zkSNARKs

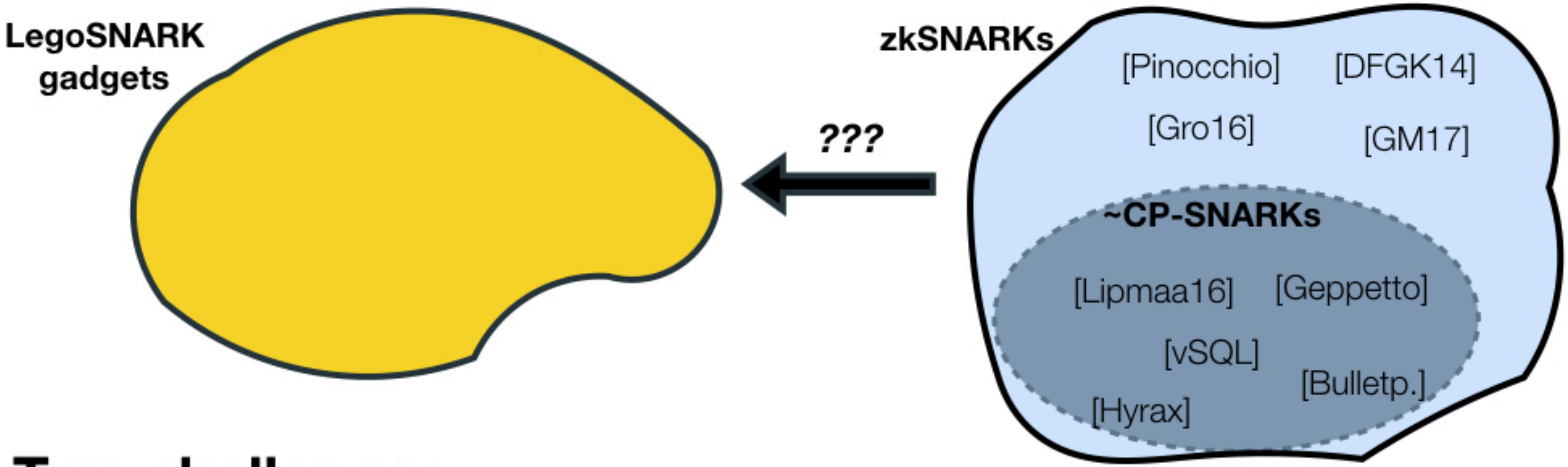


Two challenges:

A. Many Popular zkSNARKs are not CP

- **A real limitation?** If Π general-purpose, it can also prove "$c_{ck}(x)$ opens to x"

1. Import Existing zkSNARKs

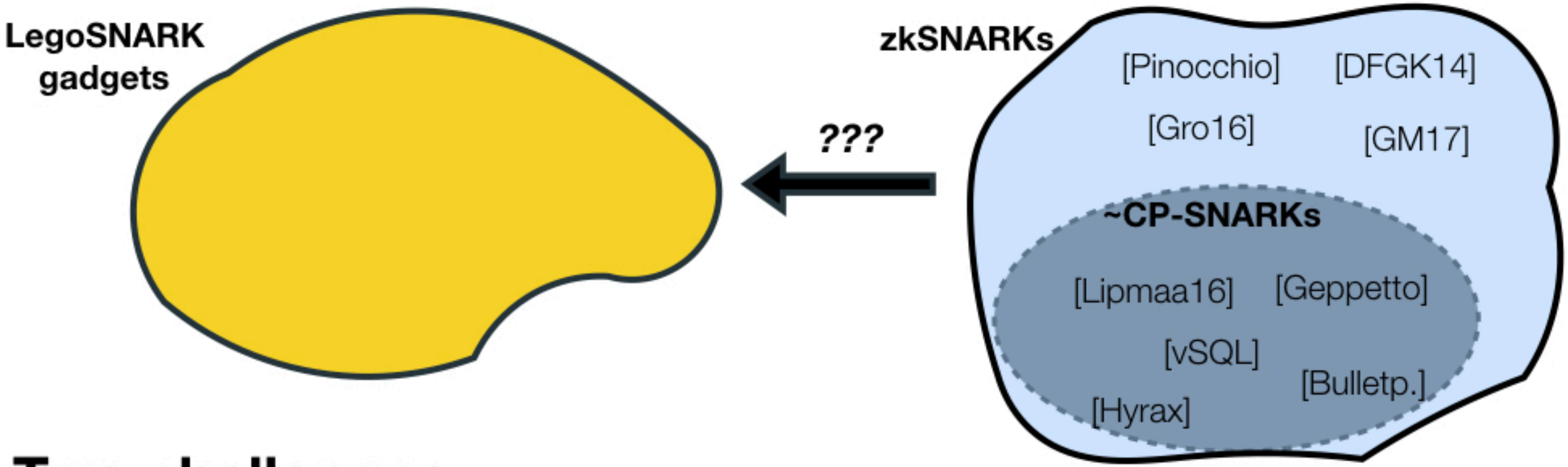


Two challenges:

A. Many Popular zkSNARKs are not CP

- **A real limitation?** If Π general-purpose, it can also prove "$c_{ck}(x)$ opens to x"
  - **Yes, in practice.** Encoding the circuit for opening can be costly
    (e.g. Pedersen commitment of 2048 bits: ~ 7minutes

1. Import Existing zkSNARKs



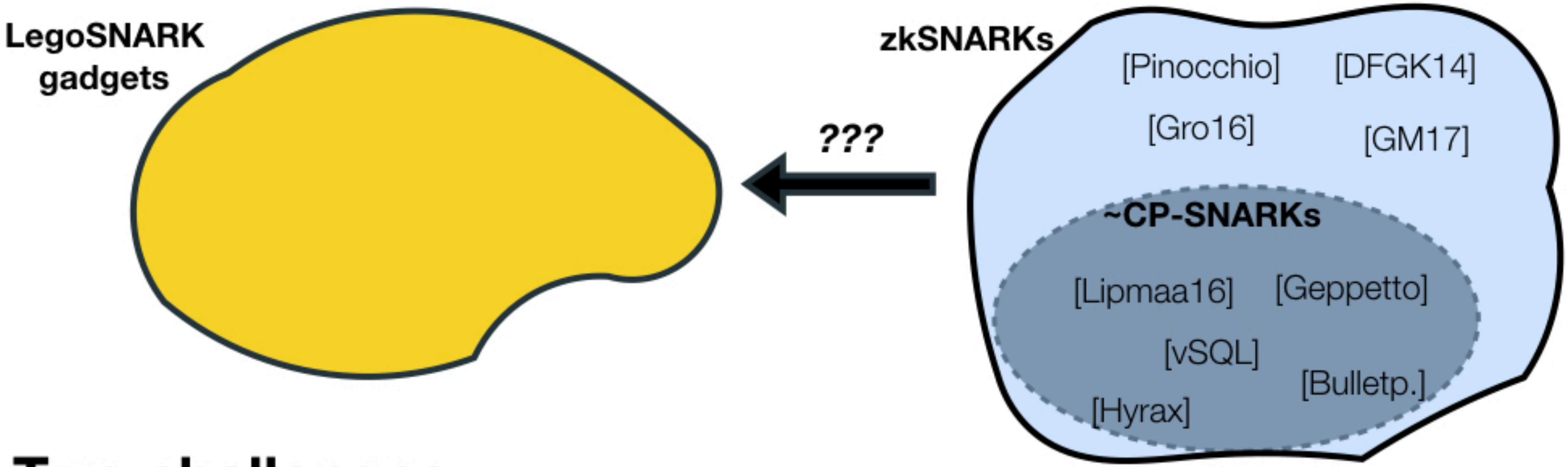
Two challenges:

A. Many Popular zkSNARKs are not CP

- **A real limitation?** If Π general-purpose, it can also prove "$c_{ck}(x)$ opens to x"
  - **Yes, in practice.** Encoding the circuit for opening can be costly
    (e.g. Pedersen commitment of 2048 bits: ~ 7minutes

B. Others are CP but in a weaker sense / have different comm. schemes or keys

1. Import Existing zkSNARKs



Two challenges:

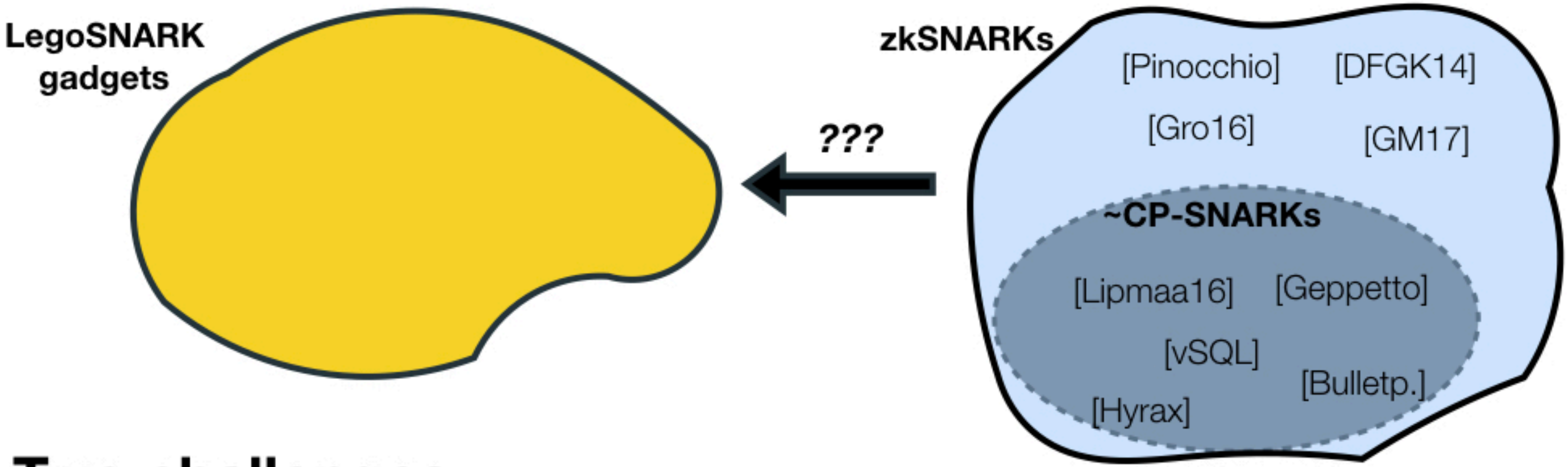
A. **Many Popular zkSNARKs are not CP**

- **A real limitation?** If Π general-purpose, it can also prove "$c_{ck}(x)$ opens to x"
  - **Yes, in practice.** Encoding the circuit for opening can be costly (e.g. Pedersen commitment of 2048 bits: ~ 7minutes

B. **Others are CP but in a weaker sense / have different comm. schemes or keys**

- How can they talk to each other?

1. Import Existing zkSNARKs



Two challenges:

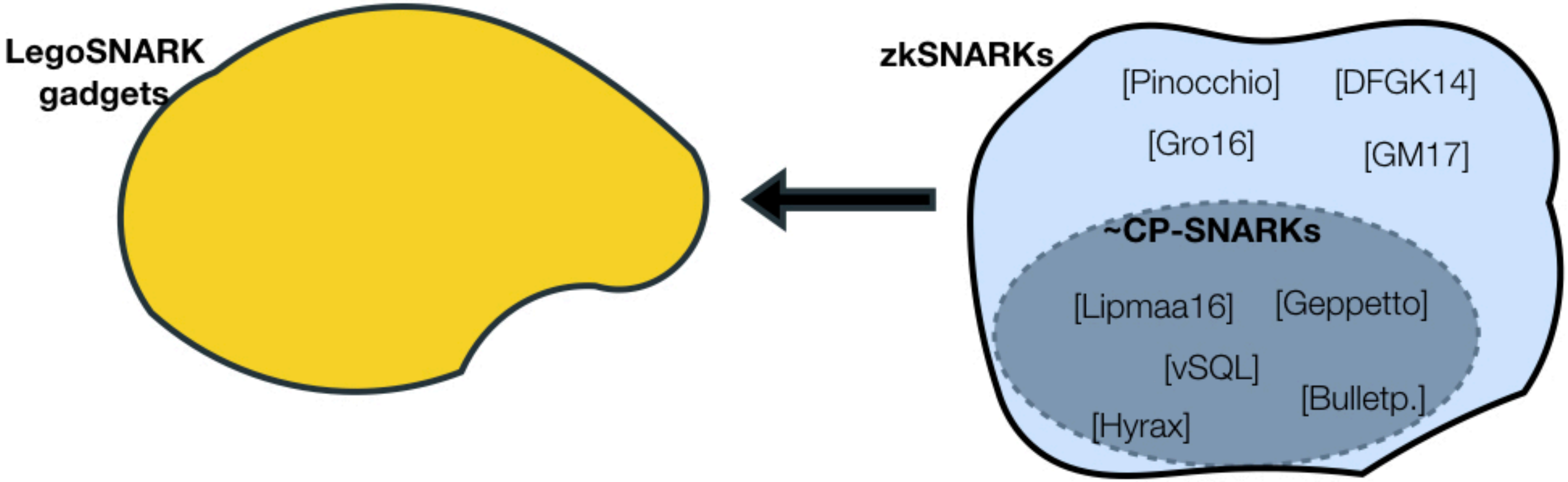
A. Many Popular zkSNARKs are not CP

- **A real limitation?** If Π general-purpose, it can also prove "$c_{ck}(x)$ opens to x"
  - **Yes, in practice.** Encoding the circuit for opening can be costly (e.g. Pedersen commitment of 2048 bits: ~ 7minutes

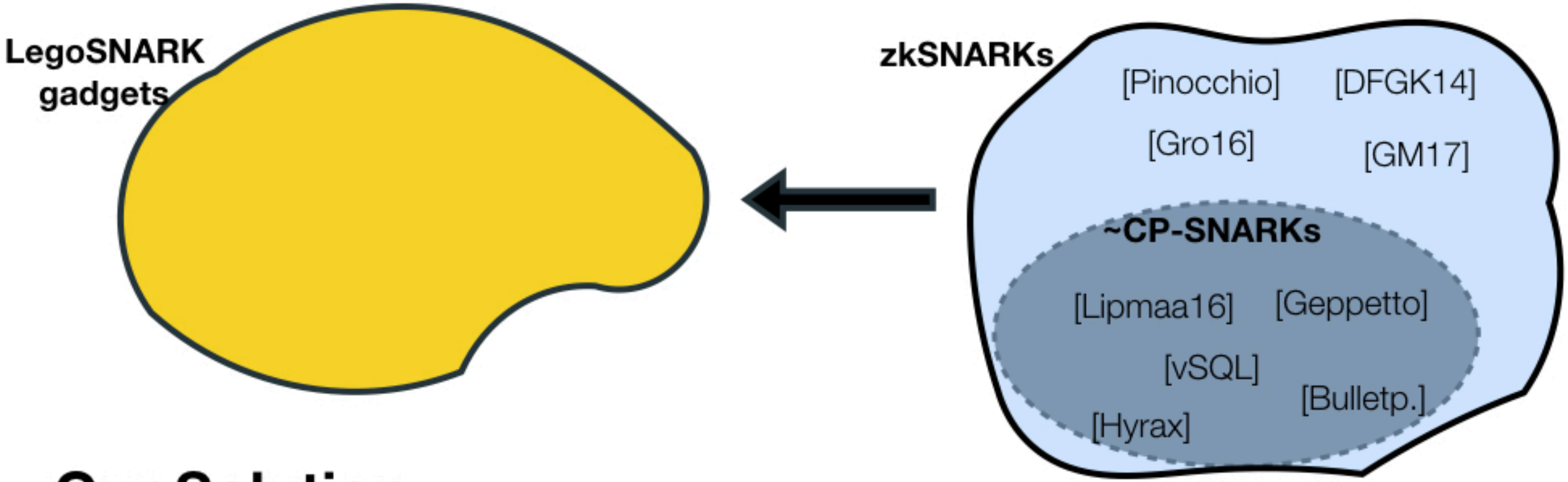
B. Others are CP but in a weaker sense / have different comm. schemes or keys

- How can they talk to each other?

Compiling zkSNARKs into CP-SNARKs

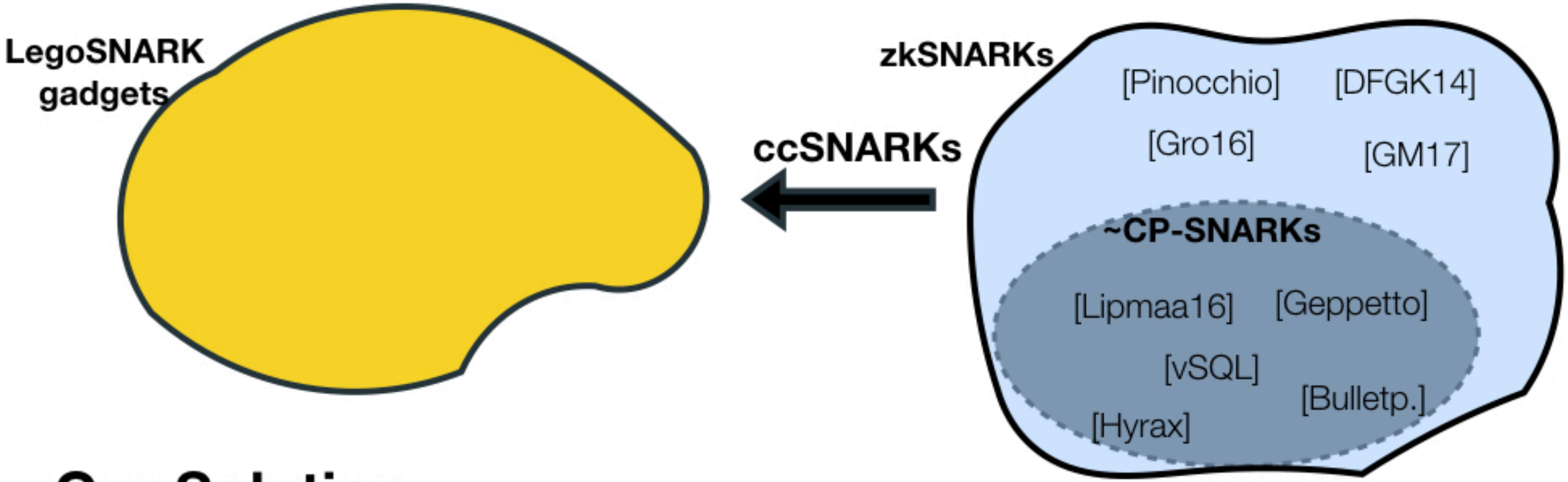


Compiling zkSNARKs into CP-SNARKs



Our Solution:

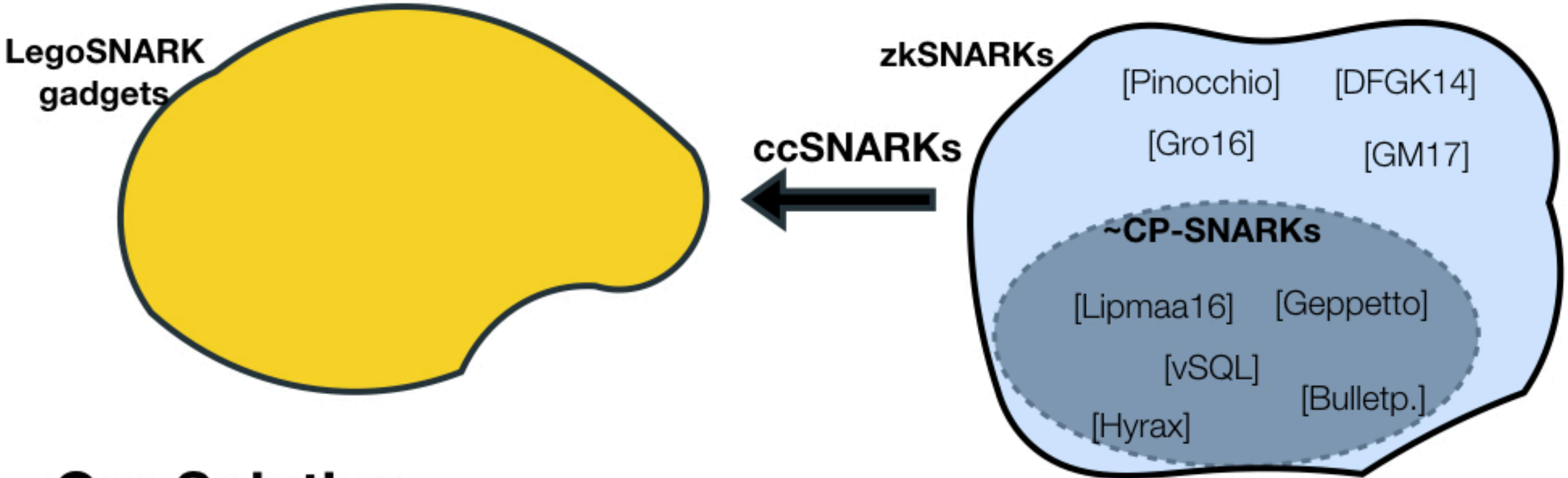
Compiling zkSNARKs into CP-SNARKs



Our Solution:

- Observe many zkSNARKs satisfy a new intermediate notion that we call **ccSNARK** (*commit-carrying SNARK*)

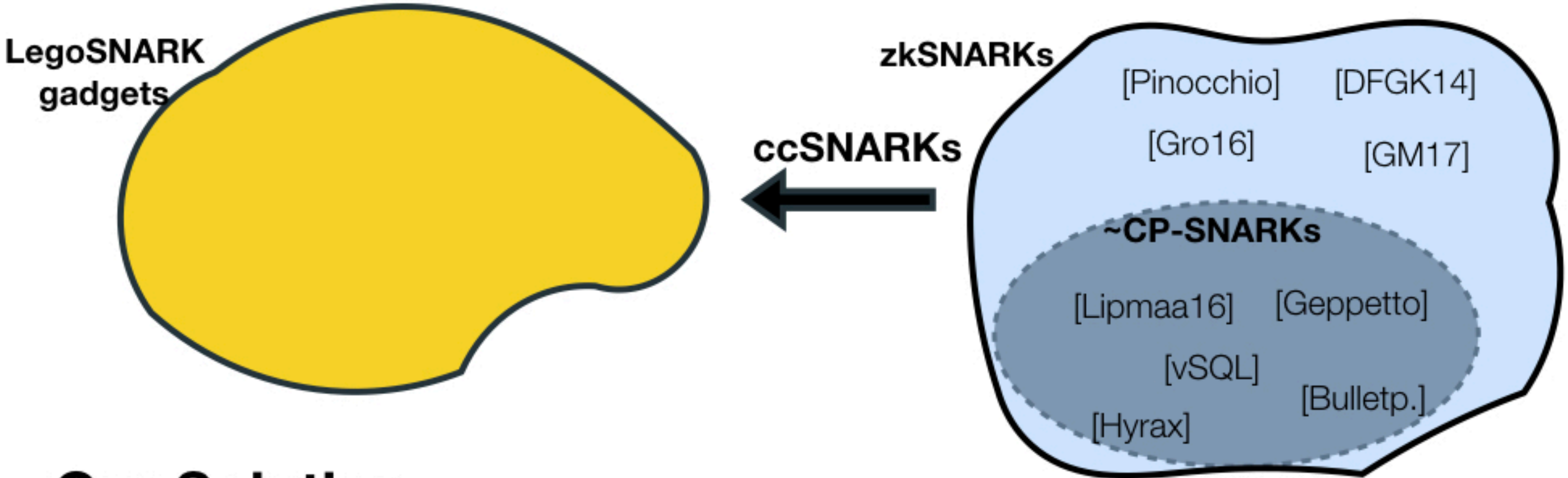
Compiling zkSNARKs into CP-SNARKs



Our Solution:

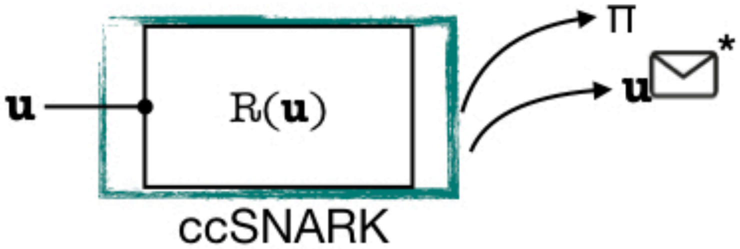
- Observe many zkSNARKs satisfy a new intermediate notion that we call **ccSNARK** (*commit-carrying SNARK*)
- Build a compiler: ccSNARK → efficient CP-SNARK

Compiling zkSNARKs into CP-SNARKs

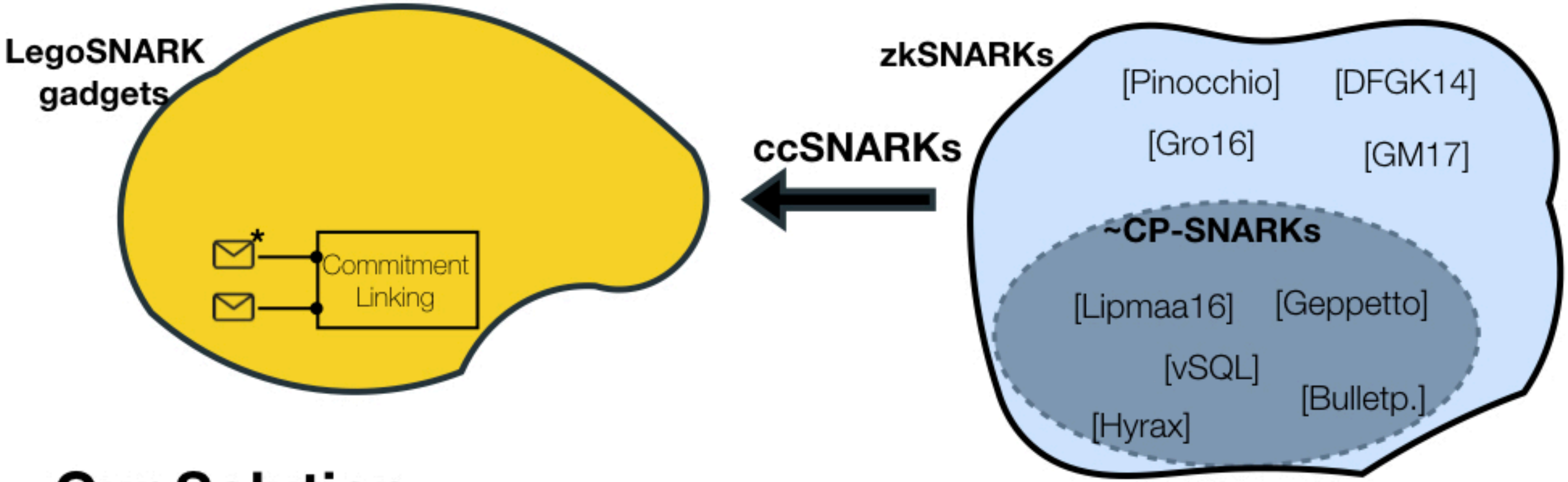


Our Solution:

- Observe many zkSNARKs satisfy a new intermediate notion that we call **ccSNARK** (*commit-carrying SNARK*)
- Build a compiler: ccSNARK → <u>efficient</u> CP-SNARK

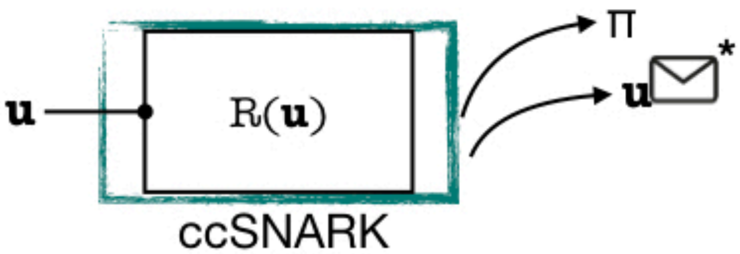


Compiling zkSNARKs into CP-SNARKs

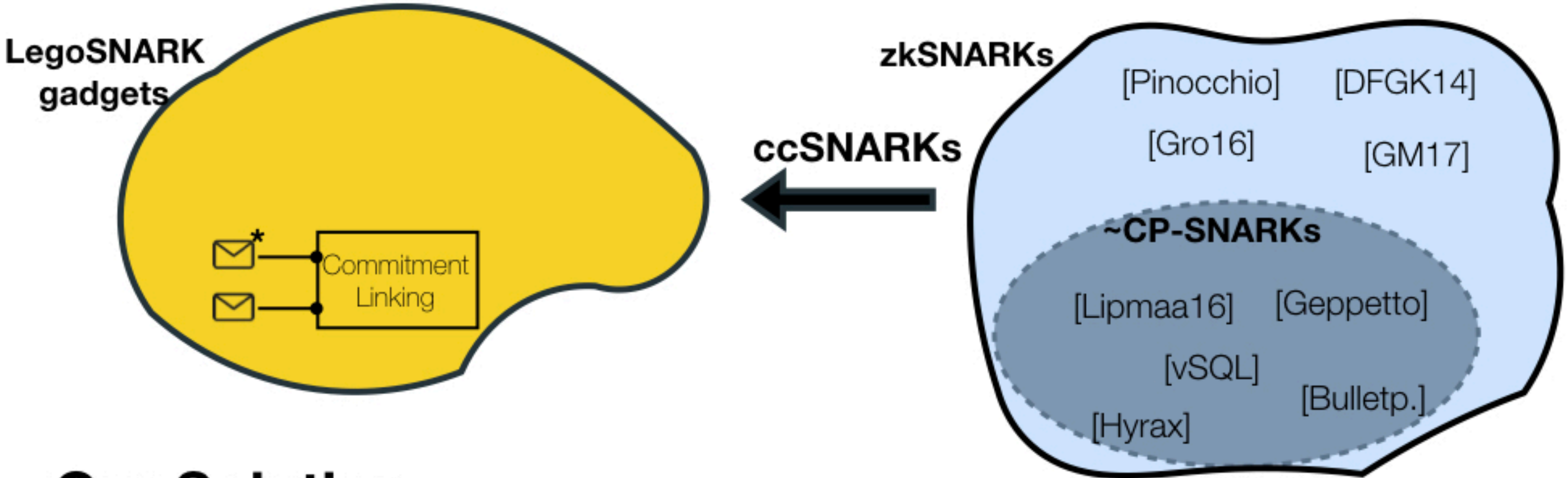


Our Solution:

- Observe many zkSNARKs satisfy a new intermediate notion that we call **ccSNARK** (*commit-carrying SNARK*)
- Build a compiler: ccSNARK → <u>efficient</u> CP-SNARK

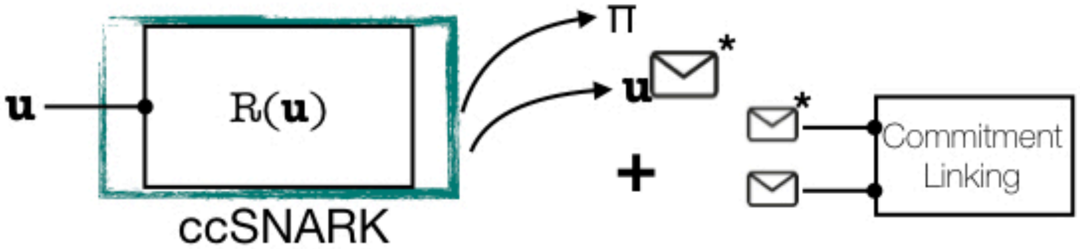


Compiling zkSNARKs into CP-SNARKs

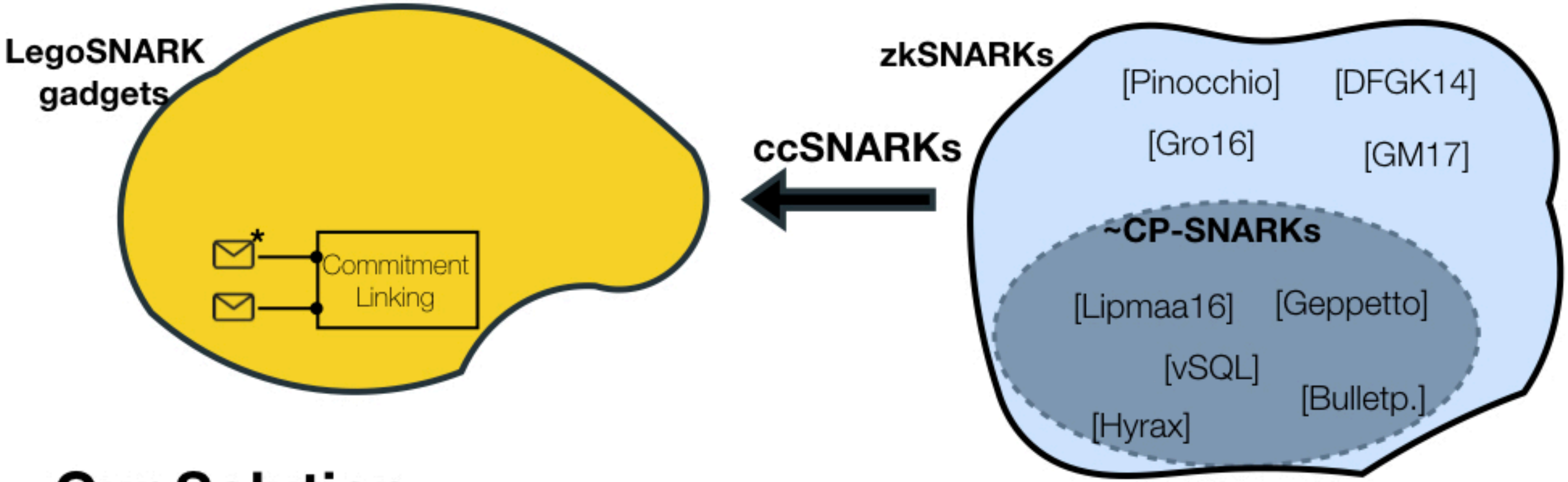


Our Solution:

- Observe many zkSNARKs satisfy a new intermediate notion that we call **ccSNARK** (*commit-carrying SNARK*)
- Build a compiler: ccSNARK → <u>efficient</u> CP-SNARK

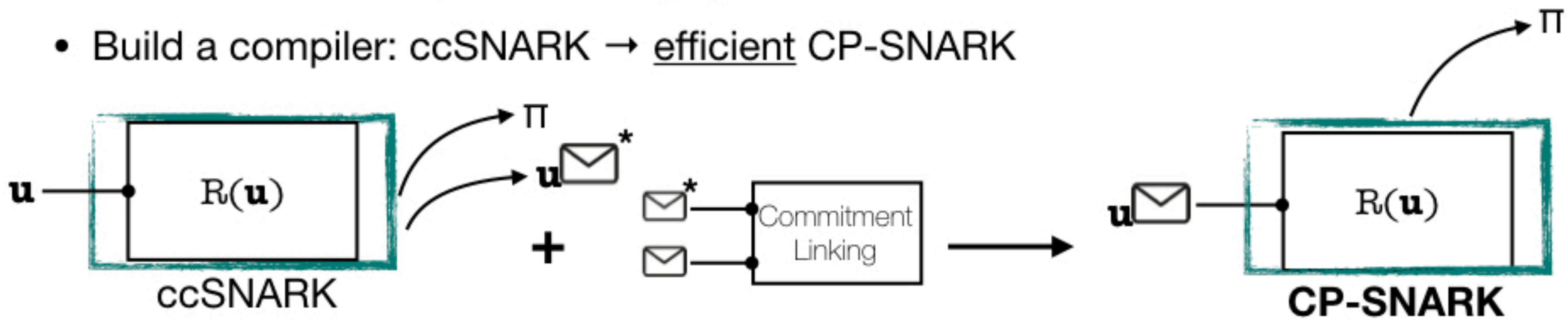


Compiling zkSNARKs into CP-SNARKs

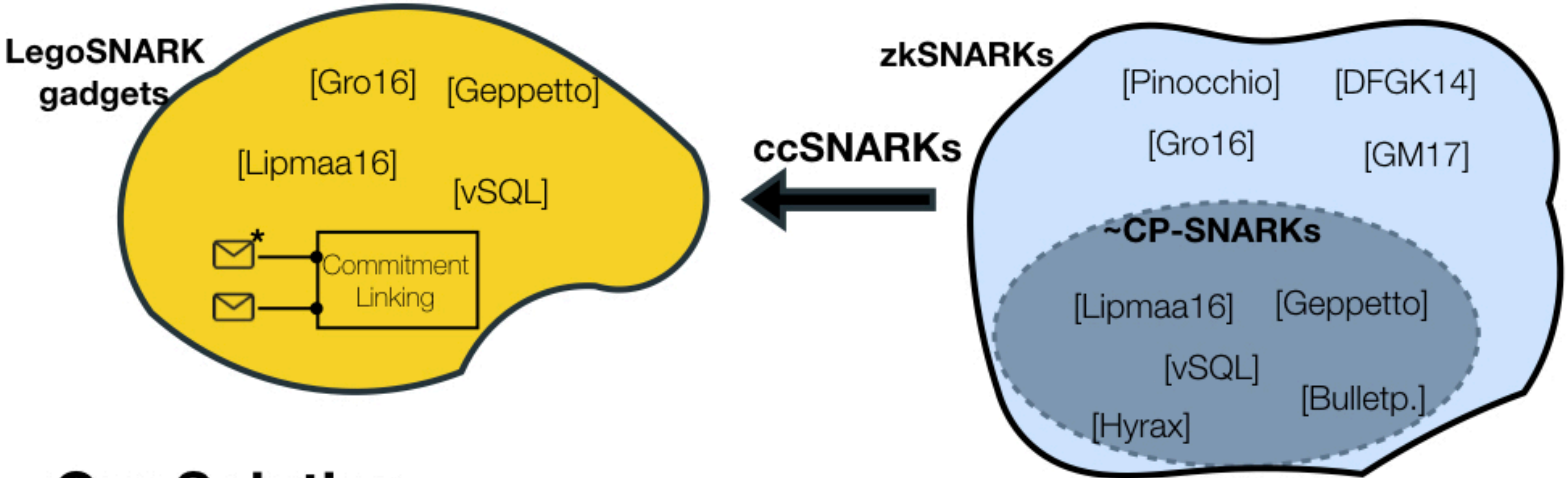


Our Solution:

- Observe many zkSNARKs satisfy a new intermediate notion that we call **ccSNARK** (*commit-carrying SNARK*)
- Build a compiler: ccSNARK → <u>efficient</u> CP-SNARK

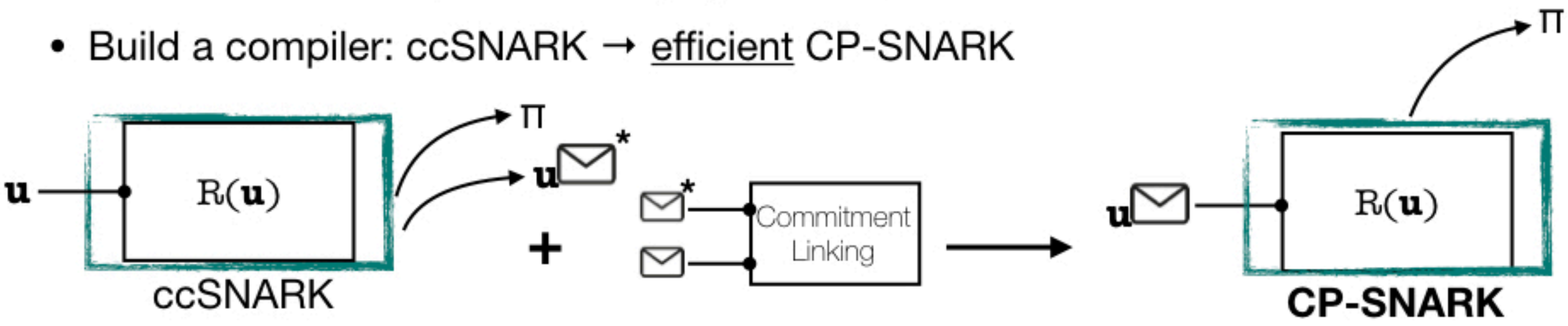


Compiling zkSNARKs into CP-SNARKs



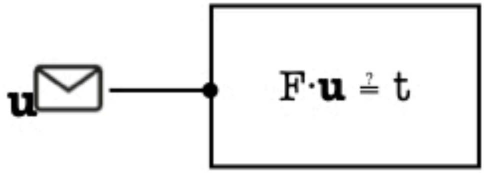
Our Solution:

- Observe many zkSNARKs satisfy a new intermediate notion that we call **ccSNARK** (*commit-carrying SNARK*)
- Build a compiler: ccSNARK → <u>efficient</u> CP-SNARK



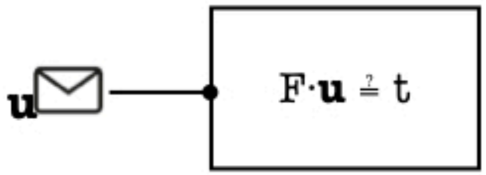
2. Specialized Proof Gadgets in LegoSNARK

2. Specialized Proof Gadgets in LegoSNARK

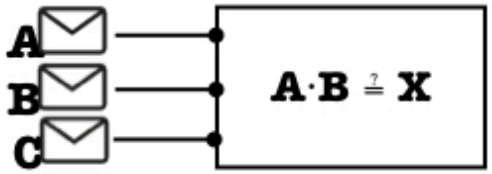


Linear Properties

2. Specialized Proof Gadgets in LegoSNARK

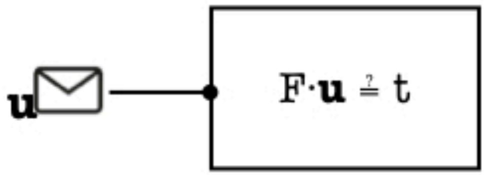


Linear Properties

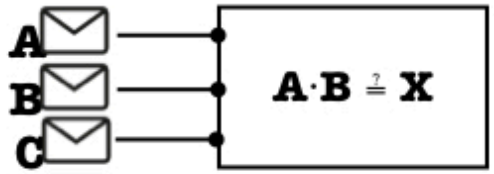


Matrix Multiplication

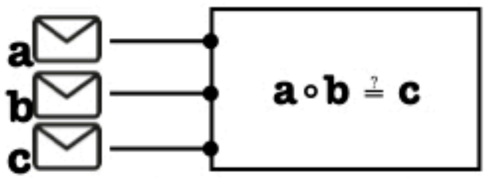
2. Specialized Proof Gadgets in LegoSNARK



Linear Properties

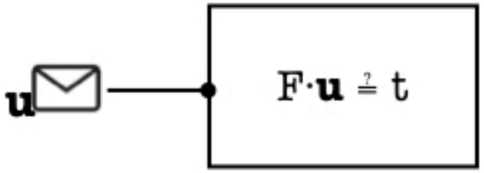


Matrix Multiplication

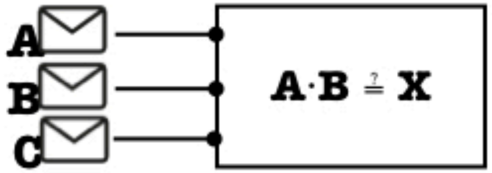


Hadamard Product

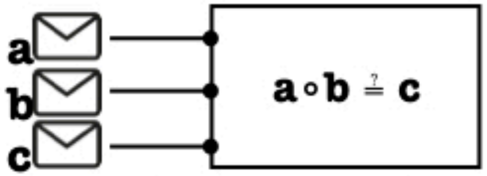
2. Specialized Proof Gadgets in LegoSNARK



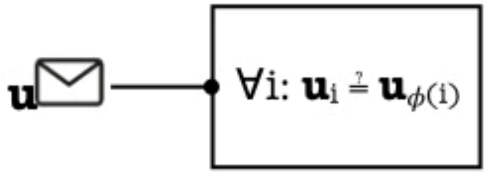
Linear Properties



Matrix Multiplication

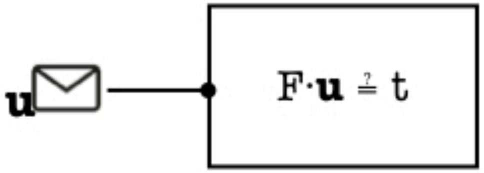


Hadamard Product

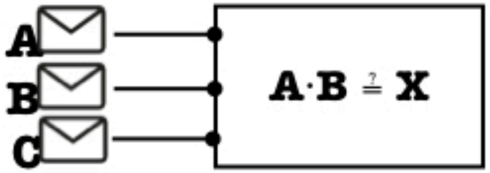


Self-Permutation

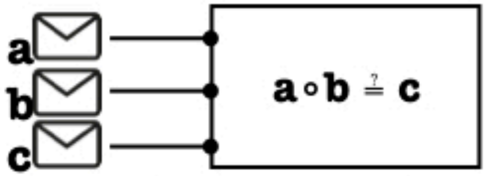
2. Specialized Proof Gadgets in LegoSNARK



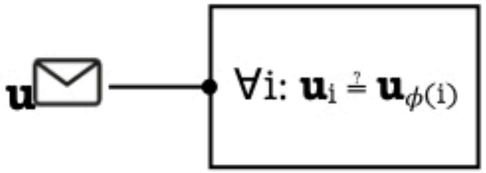
Linear Properties



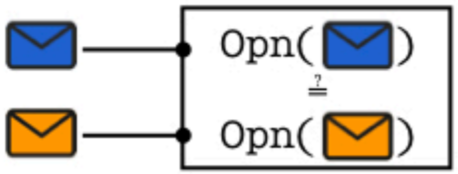
Matrix Multiplication



Hadamard Product



Self-Permutation



Commitment Linking

Is This Really Any Good?

Checking Modularity on its Promises

Checking Modularity on its Promises

- Can we build/glue new SNARKs for complex relations?

Checking Modularity on its Promises

- Can we build/glue new SNARKs for complex relations?
- Is any of this really efficient?

New General-purpose SNARKs

New General-purpose SNARKs

General-Purpose Efficient CPSnark:

LegoGroth16: efficient CP version of Groth16
(5000x faster than trivially opening a commitment in Groth16)

New General-purpose SNARKs

<u>**General-Purpose Efficient CPSnark**</u>**:**

LegoGroth16: efficient CP version of Groth16

(5000x faster than trivially opening a commitment in Groth16)

<u>**One of the *first* (CP)Snark with universal SRS**</u>**:** [concurrent to [Sonic]]

LegoUAC

(O(N) SRS; O(N) proving; $O(\log^2(N))$ proof)

New General-purpose SNARKs

General-Purpose Efficient CPSnark:

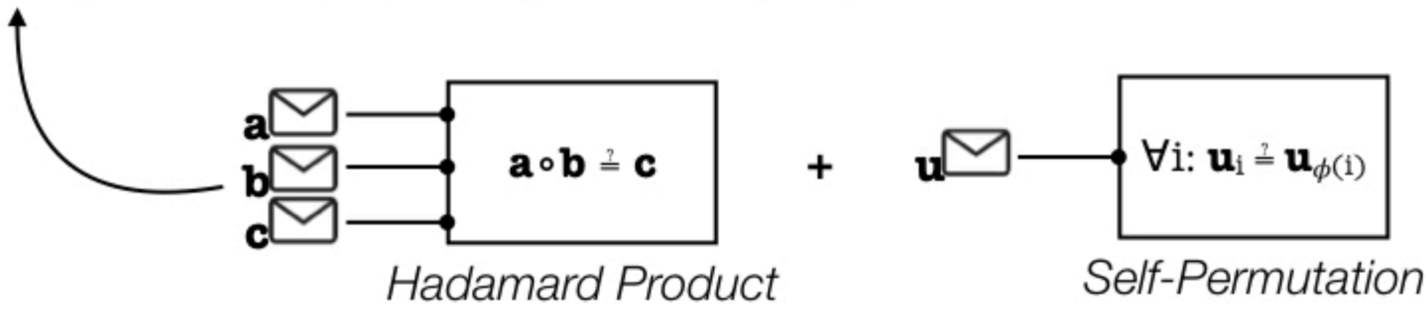
LegoGroth16: efficient CP version of Groth16

(5000x faster than trivially opening a commitment in Groth16)

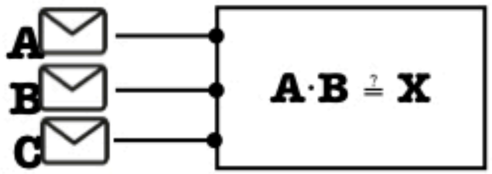
One of the *first* (CP)Snark with universal SRS: [concurrent to [Sonic]]

LegoUAC

(O(N) SRS; O(N) proving; O($\log^2$(N)) proof)

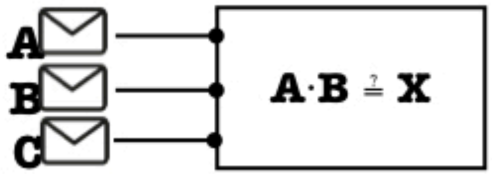


Gadgets with Optimal Proving Time: Matrix Multiplication

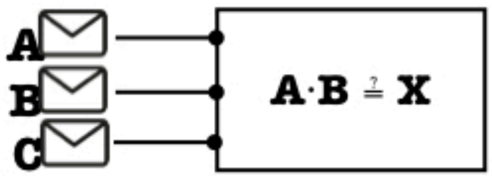


Gadgets with Optimal Proving Time: Matrix Multiplication

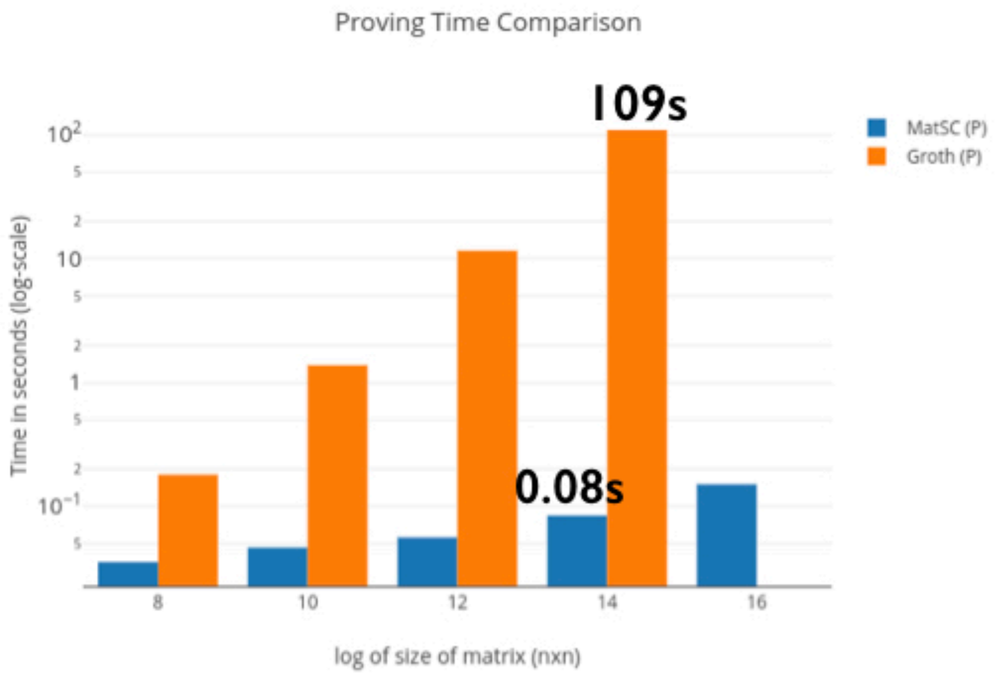
Our gadget vs Groth16:



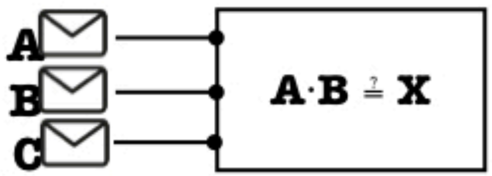
Gadgets with Optimal Proving Time: Matrix Multiplication



Our gadget vs Groth16:



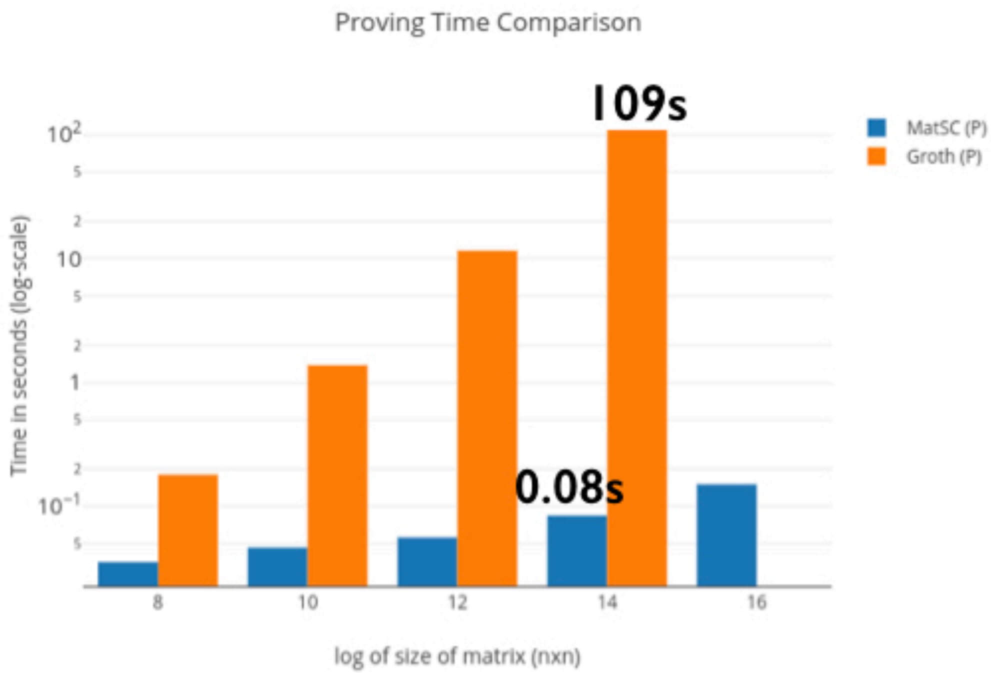
Gadgets with Optimal Proving Time: Matrix Multiplication



Our gadget vs Groth16:

n^2 **(optimal proving time)**

$n^3 \log(n)$

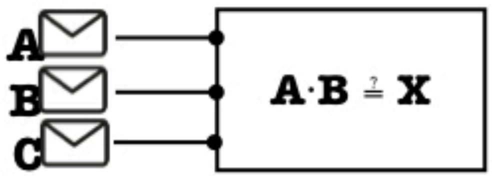


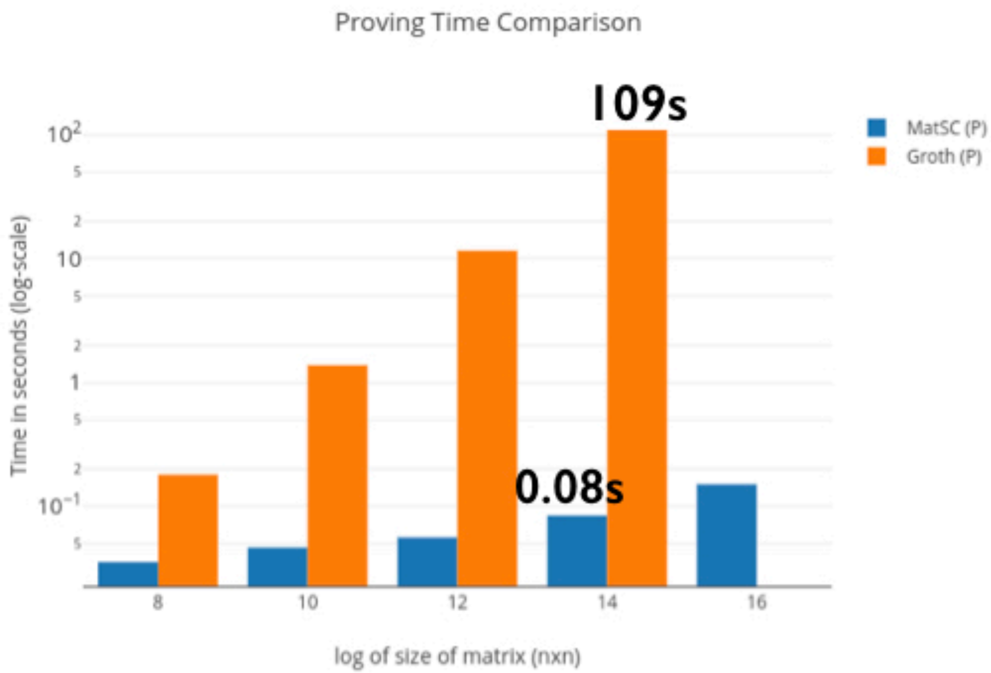
Gadgets with Optimal Proving Time: Matrix Multiplication

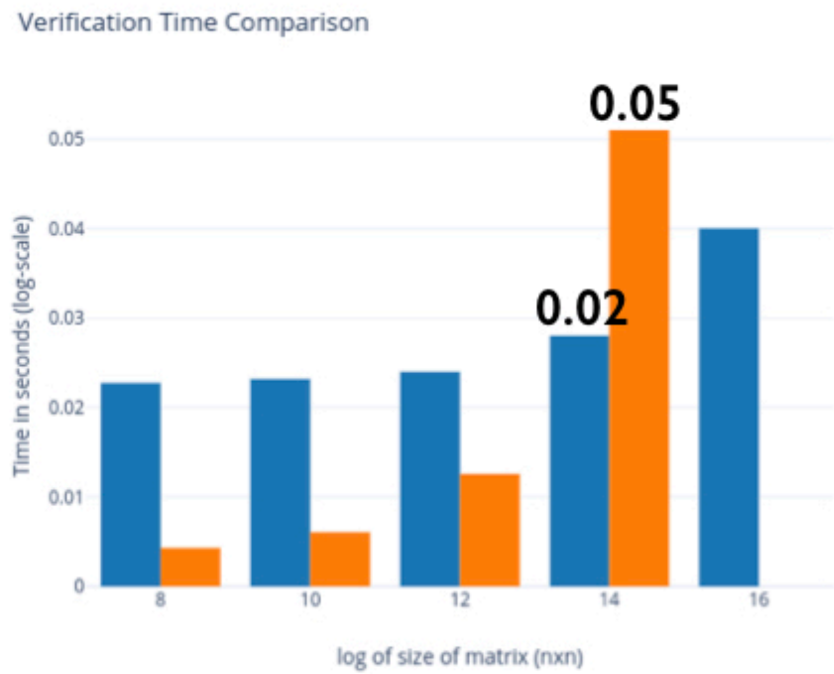
Our gadget vs Groth16:

n^2 **(optimal proving time)**

$n^3 \log(n)$





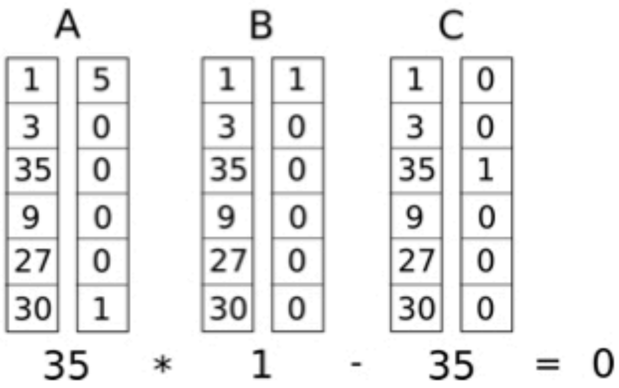


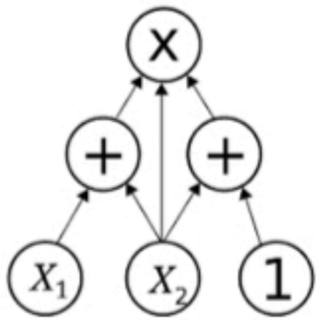
LegoSNARK in Practice

github.com/imdea-software/legosnark

Abstraction in SNARK APIs

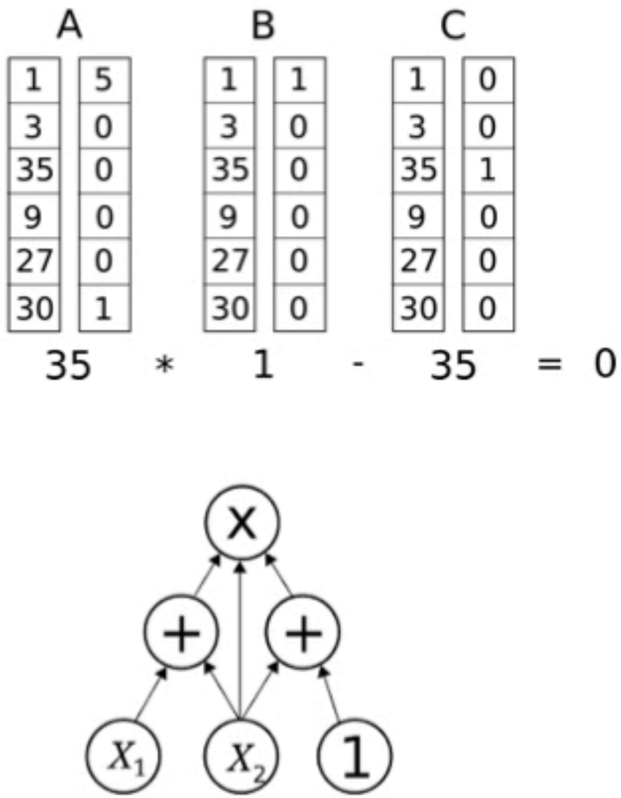
Abstraction in SNARK APIs



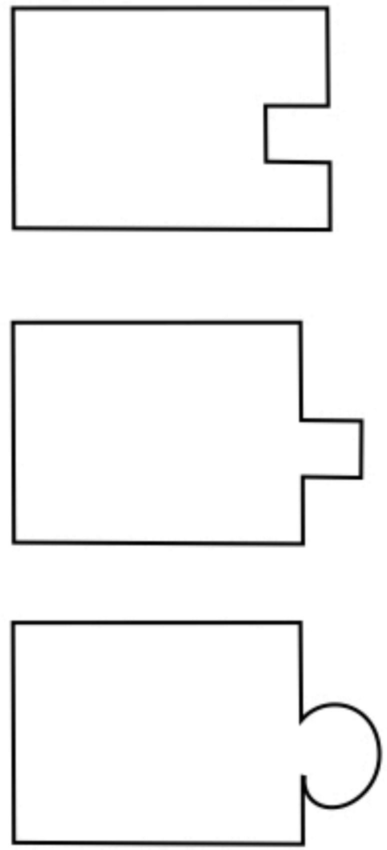


Standard Abstractions in SNARK Libraries

Abstraction in SNARK APIs



Standard Abstractions in SNARK Libraries



Abstractions in LegoSNARK

Goals of Our API

Goals of Our API

Being an EDSL.

Goals of Our API

Being an EDSL.

- Abstractions for gadgets and relations

Goals of Our API

Being an EDSL.

- Abstractions for gadgets and relations
- Strong Typing! (non-trivial in C++)

Goals of Our API

Being an EDSL.

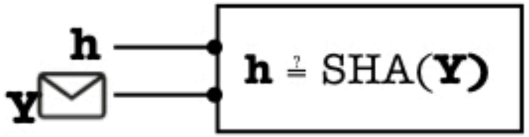
- Abstractions for gadgets and relations
- Strong Typing! (non-trivial in C++)
- Super easy to compose gadgets and relations (exploiting automatic type deduction)

Goals of Our API

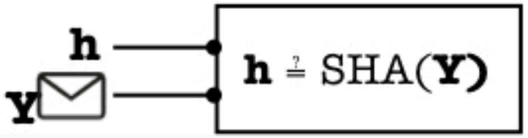
Being an EDSL.

- Abstractions for gadgets and relations
- Strong Typing! (non-trivial in C++)
- Super easy to compose gadgets and relations (exploiting automatic type deduction)
- Easy to define your own gadgets/relations

Defining Relations

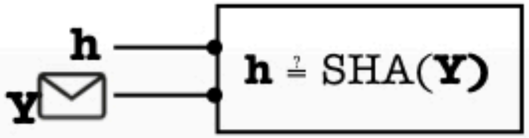


Defining Relations



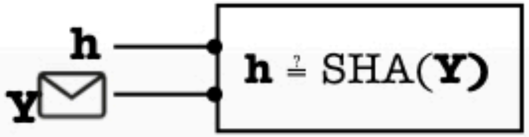

```
struct ShaSyntax
{
// [ ... ]
        decl("h"_s) .as<FldVec>() .public(), // h is a public input
        decl("Y"_s) .as<FldMatrix>() .committed() // Y is a committed input
// [ ... ]
};

// automatically defines ShaR::instance (next slide)
using ShaR = Rel<ShaSyntax>;
```

Compile-Time Checks in LegoSNARK



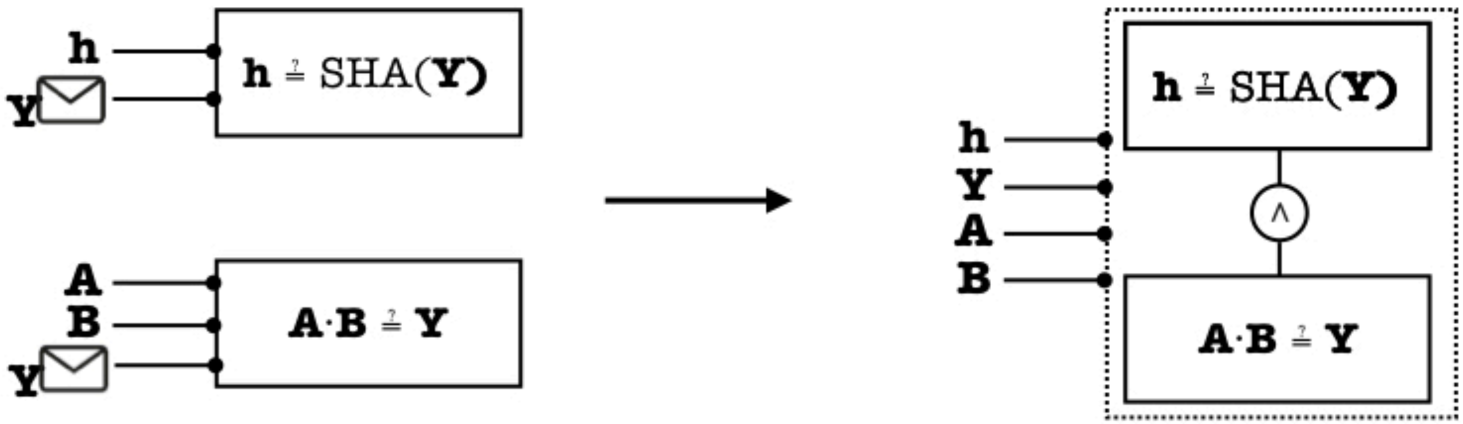

```
// Defining Instance of SHA relation
ShaR::instance sha_in;
sha_in["Y"_s] = someMatrix; // GOOD: right type
/*
sha_in["Y"_s] = someFieldElement; // WON'T COMPILE: wrong type
*/
```

Compile-Time Checks in LegoSNARK

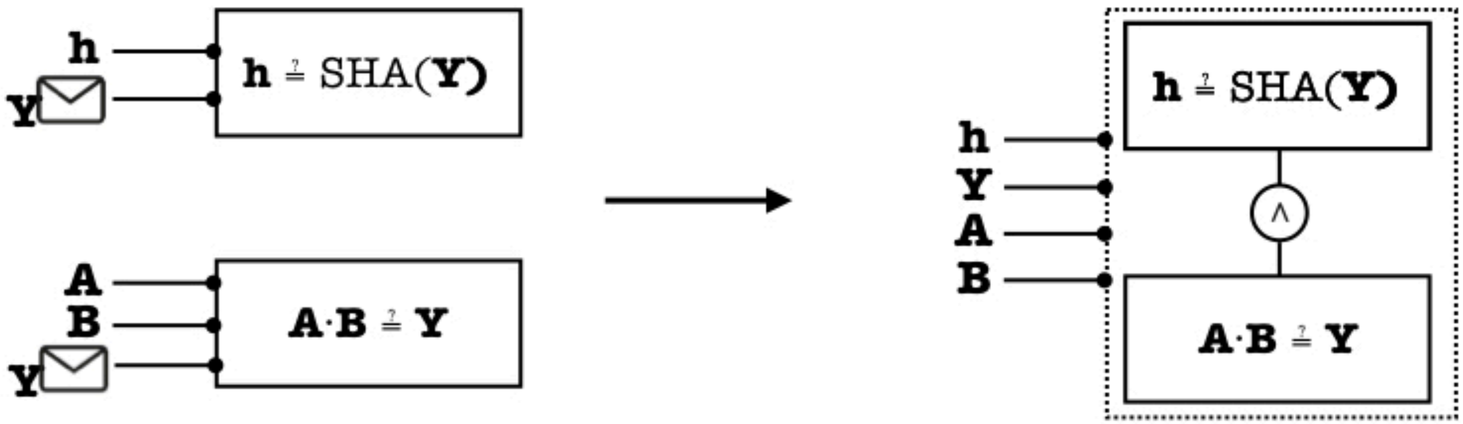



```
// Defining Instance of SHA relation
ShaR::instance sha_in;
sha_in["Y"_s] = someMatrix; // GOOD: right type
/*
sha_in["Y"_s] = someFieldElement; // WON'T COMPILE: wrong type
*/

SHAGadget sha_zkp;
// [ ... ]
sha_zkp.prv(sha_in); // WON'T COMPILE: variable "h" is not initialized

sha_in["h"_s] = someFieldVec;
sha_zkp.prv(sha_in); // GOOD: now it's initialized
```

Composition



Composition




```
using ComposedGadget = Compose<MatMulGadget, SHAGadget>;
ComposedGadget cmp_gadg;

// ...
auto matrix_sha_in = sha_in || mat_mult_in; // automatically checks compatibility
cmp_gadg.prv(matrix_sha_in);
```

Wrapping up

Wrapping up

Modular Approach to SNARK Design

Wrapping up

Modular Approach to SNARK Design

Efficiency + Ease of Design

Wrapping up

Modular Approach to SNARK Design

Efficiency + Ease of Design

Programming SNARKs differently

github.com/imdea-software/legosnark

Wrapping up

Modular Approach to SNARK Design

Efficiency + Ease of Design

Programming SNARKs differently

github.com/imdea-software/legosnark

Recent Extension:

Accumulate-and-Prove ia.cr/2019/1255

Wrapping up

Modular Approach to SNARK Design

Efficiency + Ease of Design

Programming SNARKs differently

github.com/imdea-software/legosnark

Recent Extension:
Accumulate-and-Prove ia.cr/2019/1255

Thanks!

specialized LegoSNARK gadgets

Relation	commit. scheme	CP scheme	time Prove	time Ver	space crs	space π	assumpt.	uni	upd
Pedersen commitments open to the same vector $R_{link}(\mathbf{c'}, \mathbf{u}, \mathbf{o'}) := \mathbf{c'} \stackrel{?}{=} \mathrm{Ped}(\mathbf{u}, \mathbf{o'})$ $n=\lvert\mathbf{u}\rvert$	Pedersen*	CP_{link}	*n*	*1*	*n*	*1*	AGM		
Linear properties $R_{F,c}(\mathbf{u}) := \mathbf{F} \cdot \mathbf{u} \stackrel{?}{=} \mathbf{c}$ $\mathbf{F}$ $m \times n$	Pedersen*	CP_{lin}	*n*	*1*	*n*	*1*	AGM		
	PolyCom	CP'_{lin}	$\lvert F\rvert+m+n$	$\log m \cdot n$	$m \cdot n$	$\log m \cdot n$	q-SDH, KoE, ROM		
Matrix multiplication $R_{mm}(\mathbf{X}, \mathbf{A}, \mathbf{B}) := \mathbf{X} \stackrel{?}{=} \mathbf{A} \cdot \mathbf{B}$ $n \times n$	PolyCom	CP_{mmul}	n^2	$n^2+\log n$	n^2	$\log n$	q-SDH, KoE, ROM		
Hadamard product $R_{had}(\mathbf{a}, \mathbf{b}, \mathbf{c}) := \mathbf{c} \stackrel{?}{=} \mathbf{a} \circ \mathbf{b}$ $n=\lvert\mathbf{u}\rvert$	PolyCom	CP_{had}	*n*	$\log n$	*n*	$\log n$	q-SDH, KoE, ROM		
Self permutation $R_{\phi}(\mathbf{u}) := \forall i: u_i \stackrel{?}{=} u_{\phi(i)}$ $n=\lvert\mathbf{u}\rvert$	PolyCom	CP_{sfprm}	*n*	$\log n$	*n*	$\log n$	q-SDH, KoE, ROM		

Pedersen* = any Pedersen-like commitment. **PolyCom** from [zk-vSQL]

AGM='Algebraic Group Model'. **uni**versal crs (yes, no). **upd**atable crs (yes, to be proven)

Thanks!