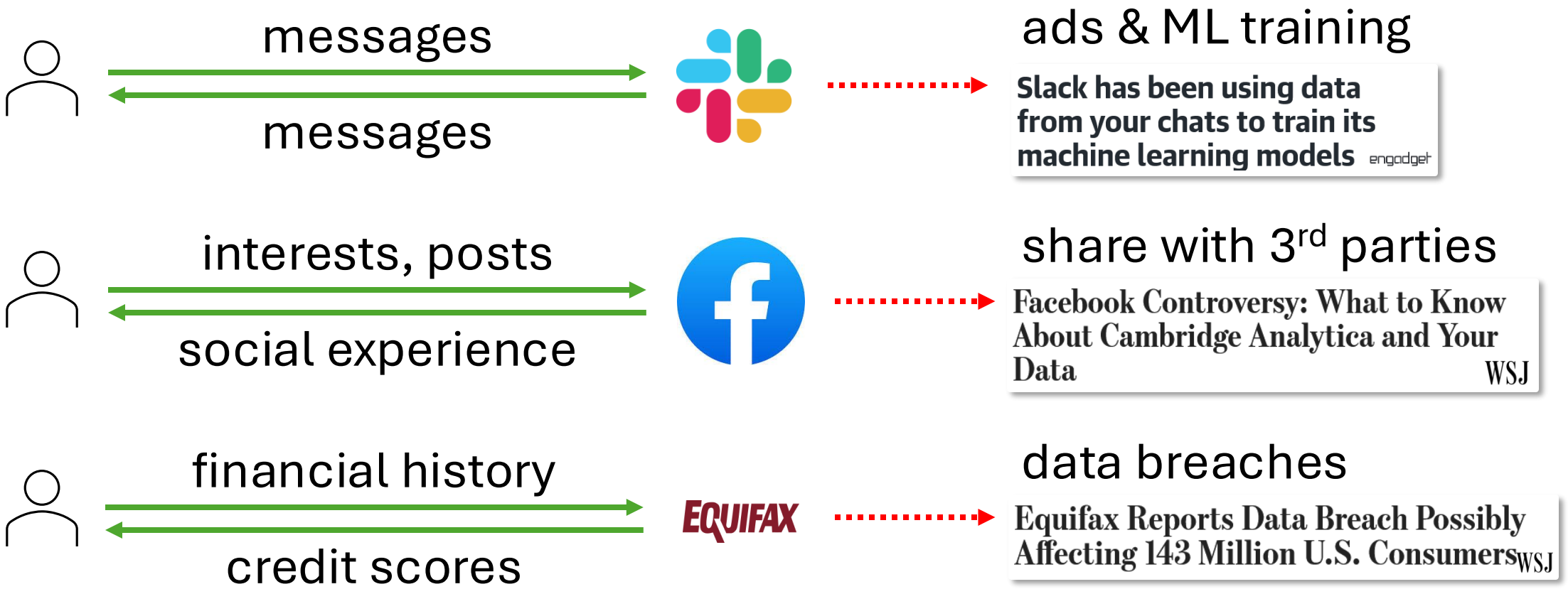
Security & **privacy** through **programmable cryptography**, **compilation**, and **verification**

Alex Ozdemir

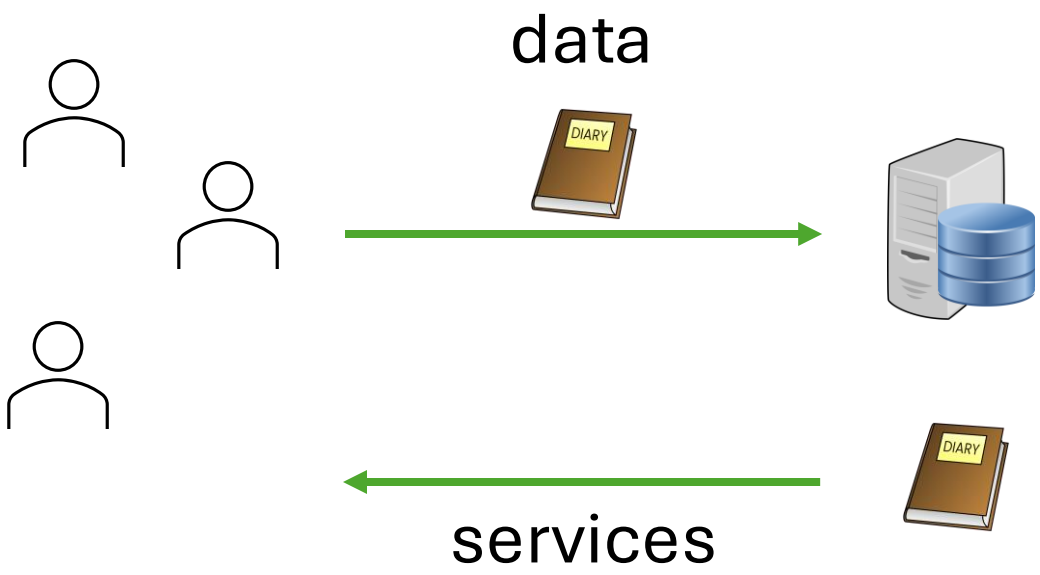
Stanford

Based on joint work with: Clark Barrett, Dan Boneh, Fraser Brown, Gereon Kremer, Cesare Tinelli, Riad S. Wahby

To use the internet is to be vulnerable



To use the internet is to be vulnerable

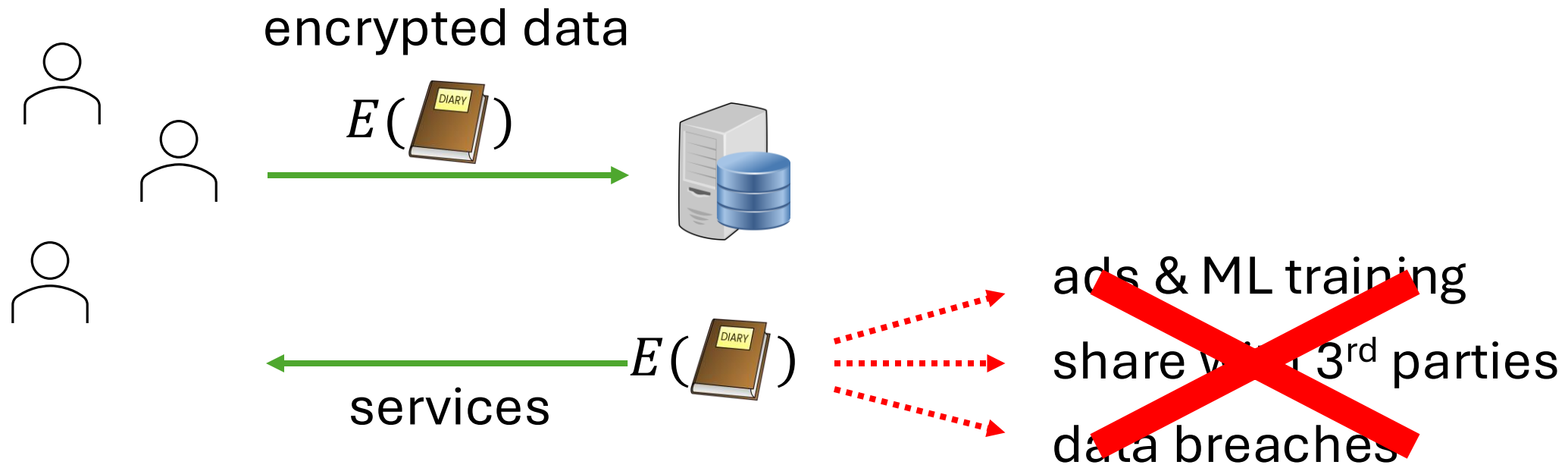


ads & ML training

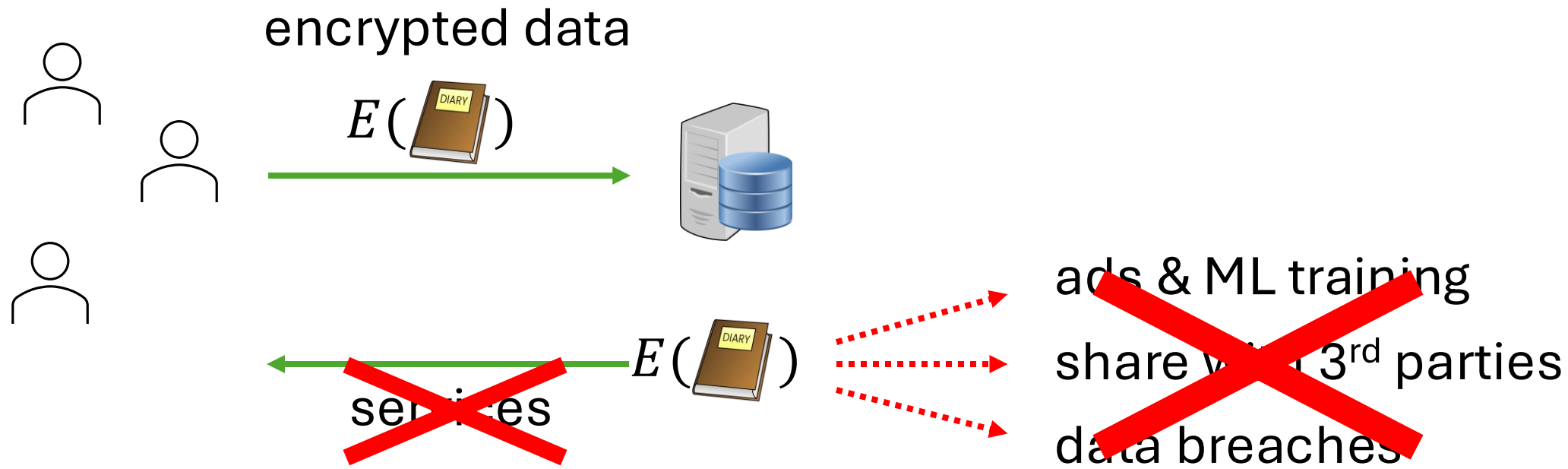
share with 3rd parties

data breaches

Encryption secures data



Encryption secures data, but prevents use



Programmable cryptography enables use

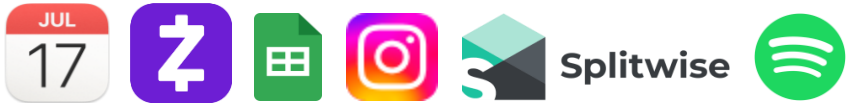
A **programmable cryptosystem** securely executes a developer-defined program on secret data.

Examples:

FHE	ZKP	MPC
Fully Homomorphic Encryption	Zero Knowledge Proof	Multi Party Computation
compute on encrypted data	**prove** properties of secret data	**compute** on distributed data
$E(x) \rightarrow E(f(x))$	$E(x) \rightarrow \phi(x)$	$x \parallel y \rightarrow f(x, y)$

Encryption secures ***data***.

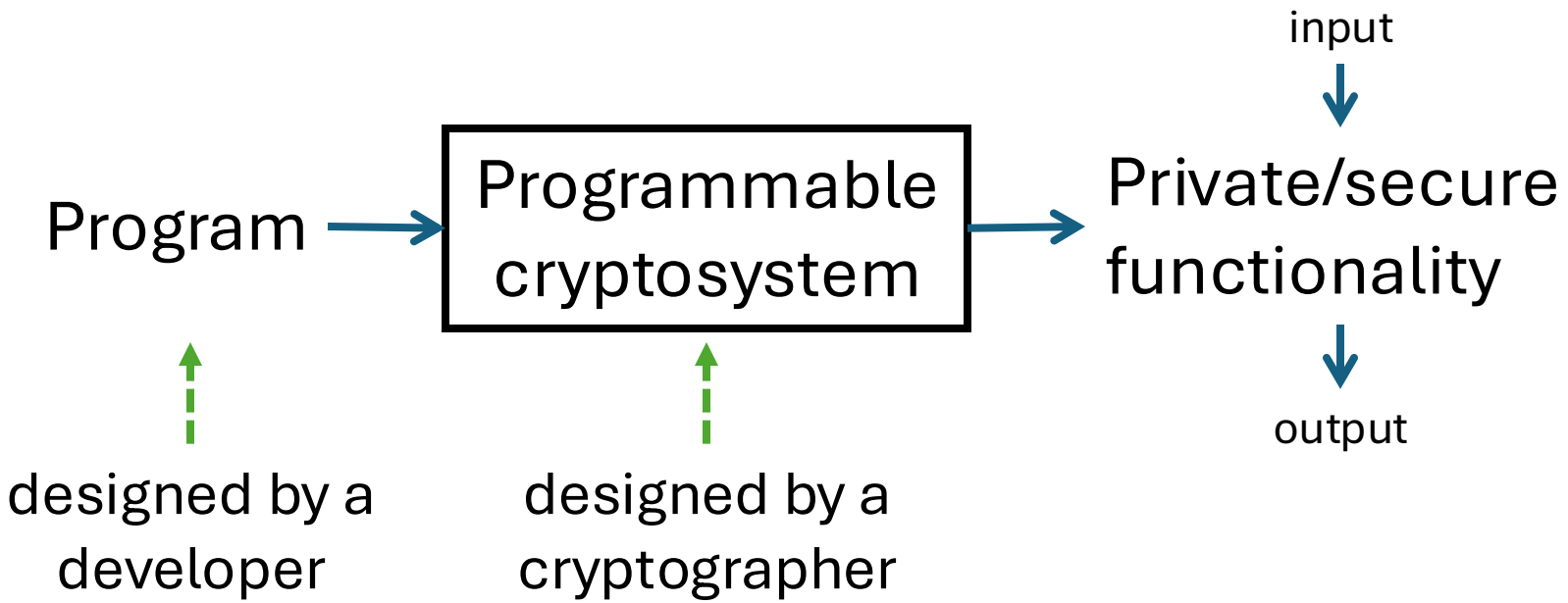
Programmable cryptography secures ***data in use***.



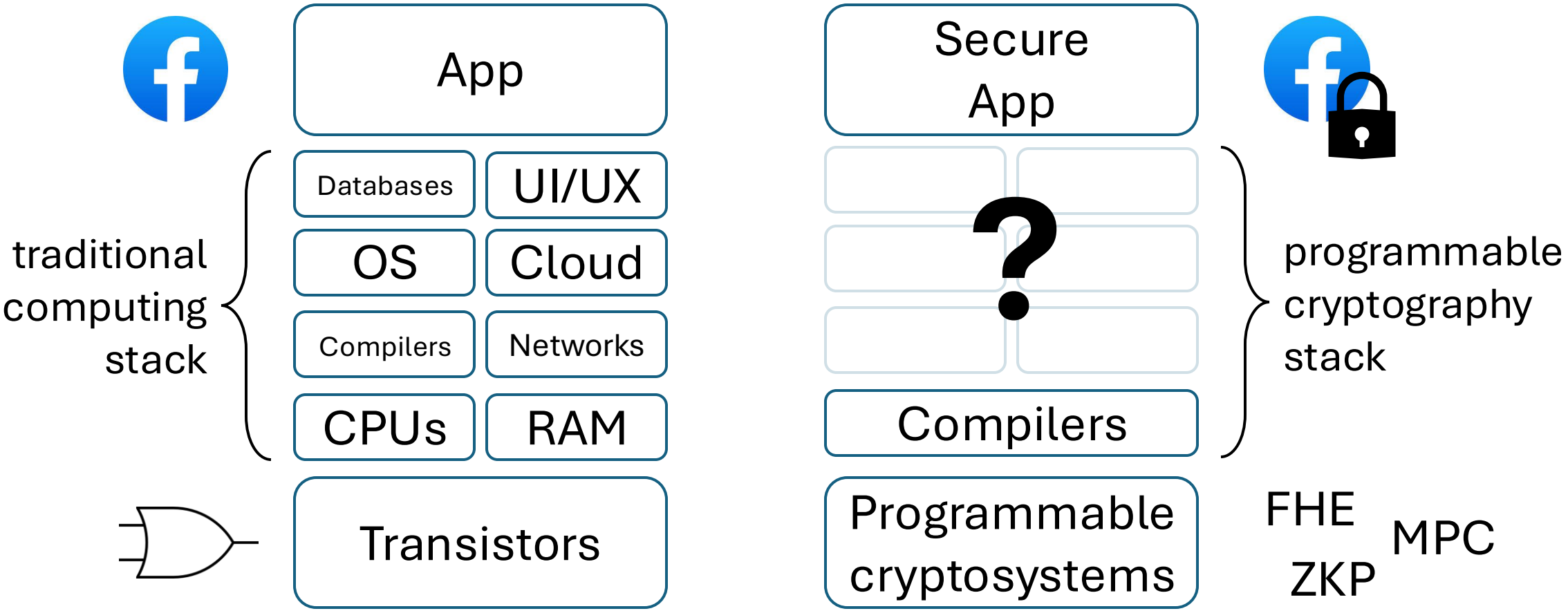
[Gentry'09] [RivAdlDer'78] [GolMicRac'85] [Yao'82]

Programmable Cryptography is general

A **programmable cryptosystem** securely executes a **developer-defined program** on secret data.



Programmable cryptography needs a **stack**

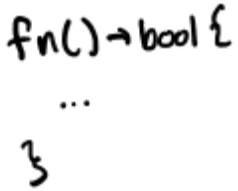


My focus, so far:

Idea

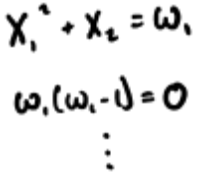
Programmer

High-level program



Compiler

Low-level Intermediate Representation (IR)



Programmable cryptosystem

Secure app

My approach:

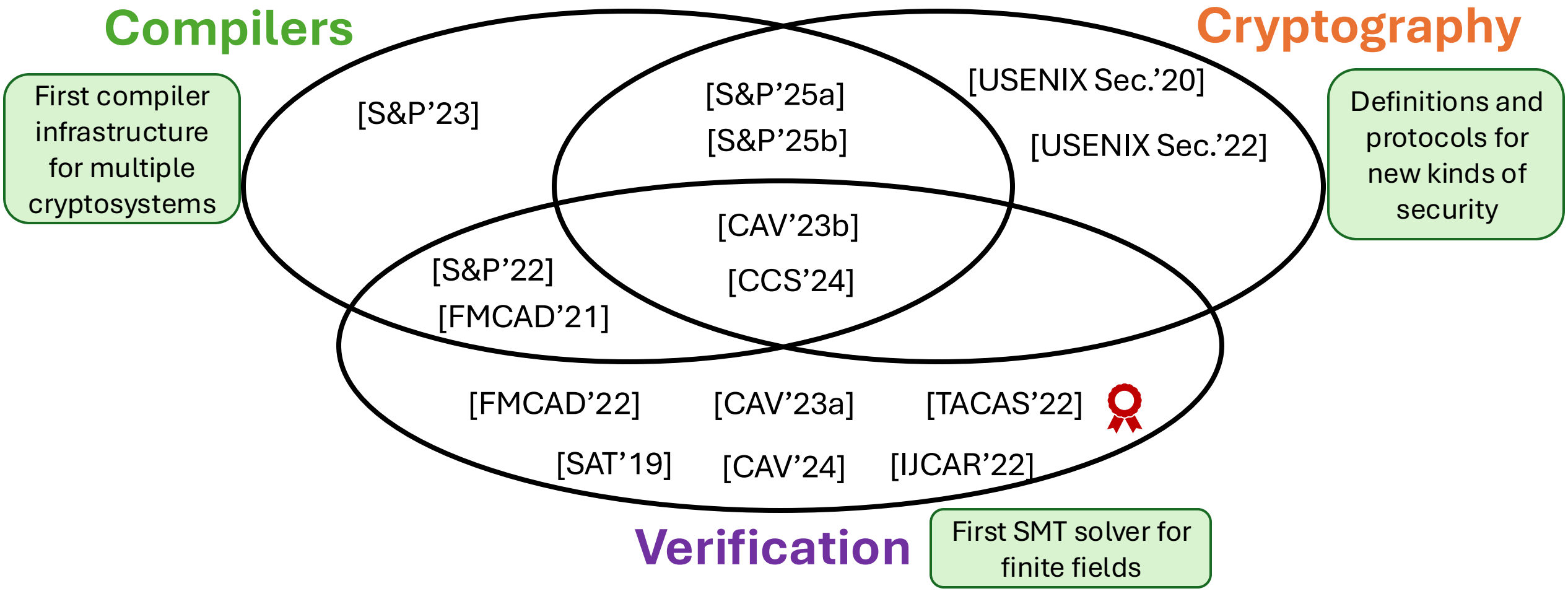
1. Adopt a full-stack view
2. Lay a foundation for others

academia

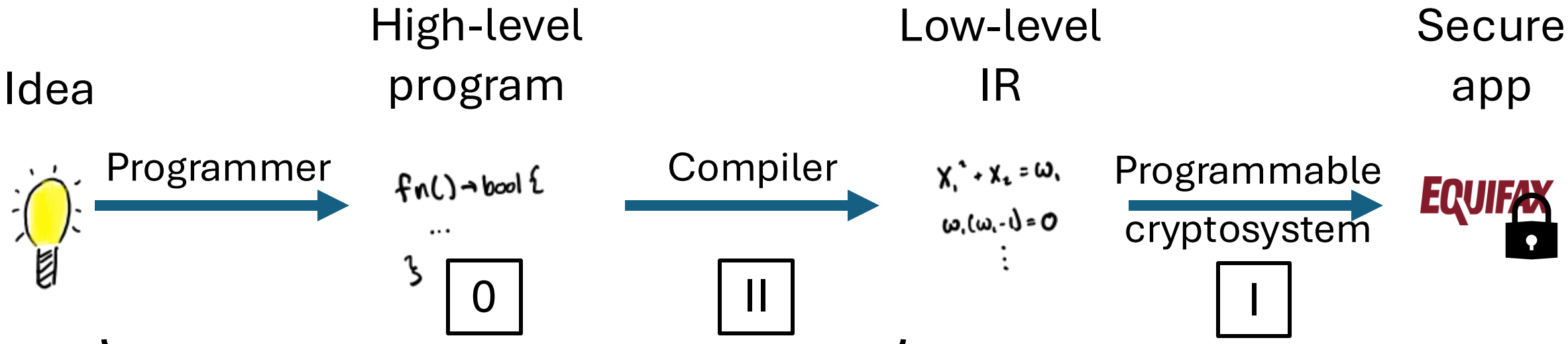
industry

3. Cross disciplinary boundaries
   - Cryptography
   - Verification
   - Compilers

My work:


Compilers
Cryptography
First compiler infrastructure for multiple cryptosystems
Definitions and protocols for new kinds of security
[S&P'23]
[S&P'25a]
[S&P'25b]
[USENIX Sec.'20]
[USENIX Sec.'22]
[CAV'23b]
[CCS'24]
[S&P'22]
[FMCAD'21]
[FMCAD'22]
[CAV'23a]
[TACAS'22]
[SAT'19]
[CAV'24]
[IJCAR'22]
Verification
First SMT solver for finite fields

This talk



Prevents undetectable money printing

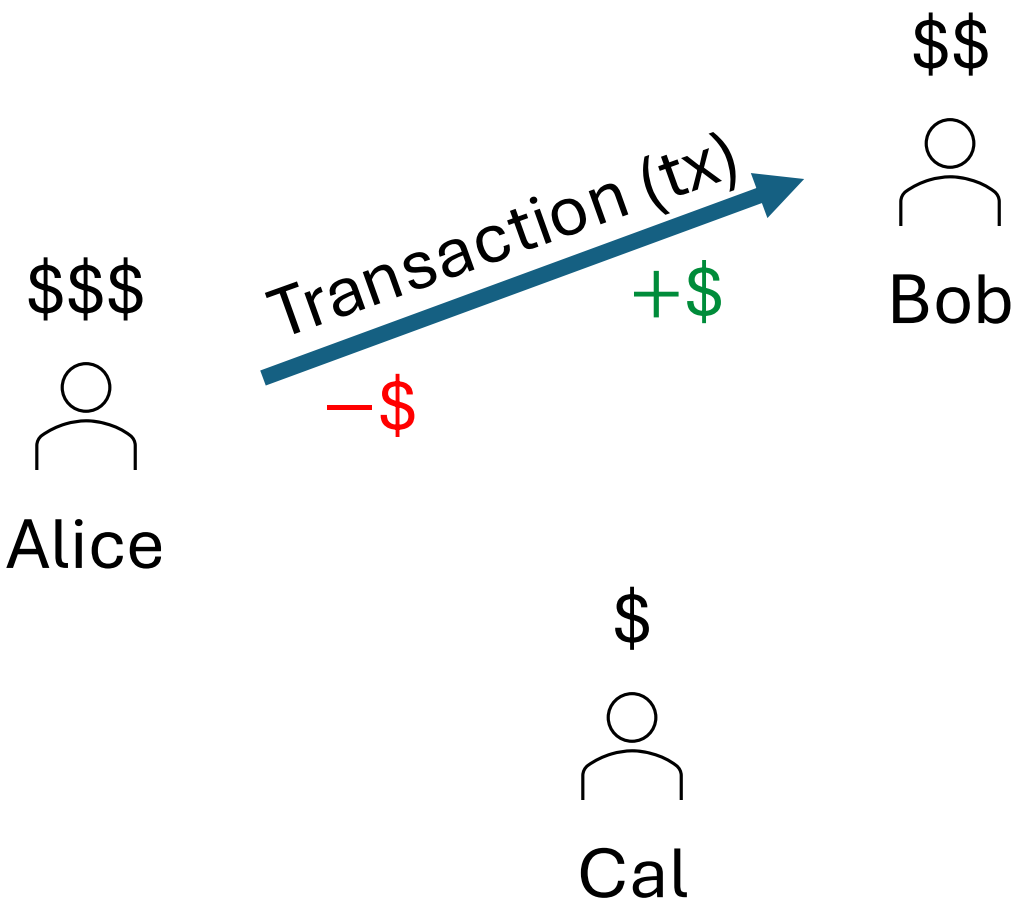
Part 0: An example: private payments

Part I: Collaborative zero knowledge (cryptography)

Part II: Common compiler infrastructure (compilers)

Part III: Verification via finite field solvers (verification)

Running Example: Private Payments



Security properties:

Privacy:
- only owners know balances
- only participants know tx amounts

Integrity:
- money is conserved
- all balances are positive
- participants authorize every tx

First attempt: payments from a public log

Alice has d_A \$.

Alice sends $\delta \leq [0, d_A]$ \$.

Public log

$A: d_A$

$B: d_B$

$A: d_A - \delta$

$B: d_B + \delta$

σ_A, σ_B

Bob has d_B \$.

all data is public
⇒ no ***privacy***

users check log
⇒ easy ***integrity***

Second attempt: just encrypt the log?

Alice has d_A \$.

$e_A = E(k_A, d_A)$

Alice sends $\delta \leq [0, d_A]$ \$.

$e'_A = E(k_A, d_A - \delta)$

Public log

e_A

e_B

e'_A

e'_B

σ_A, σ_B

Bob has d_B \$.

$e_B = E(k_B, d_B)$

$e'_B = E(k_B, d_B + \delta)$

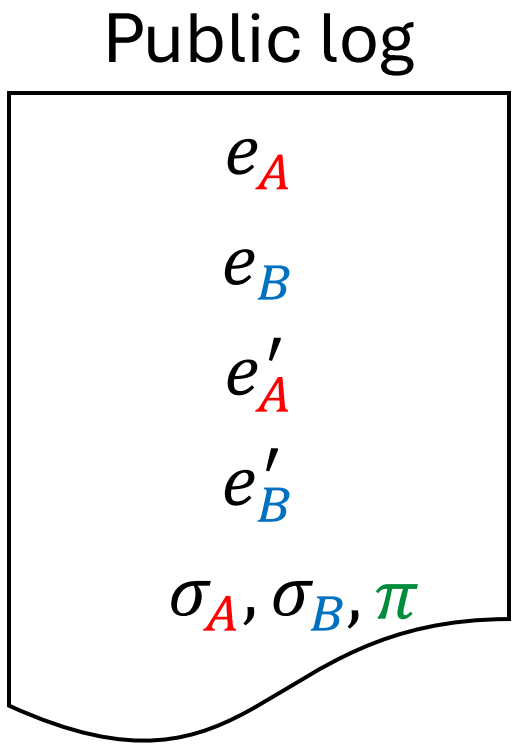
What if $e'_B = E(k_B, d_B + 2\delta)$?

Balances is encrypted
⇒ easy ***privacy***

Can't validate txs
⇒ no ***integrity***

*Let E be a suitable cipher (e.g., AEAD + CMT-4). "Confidential Transactions" [Maxwell '15]

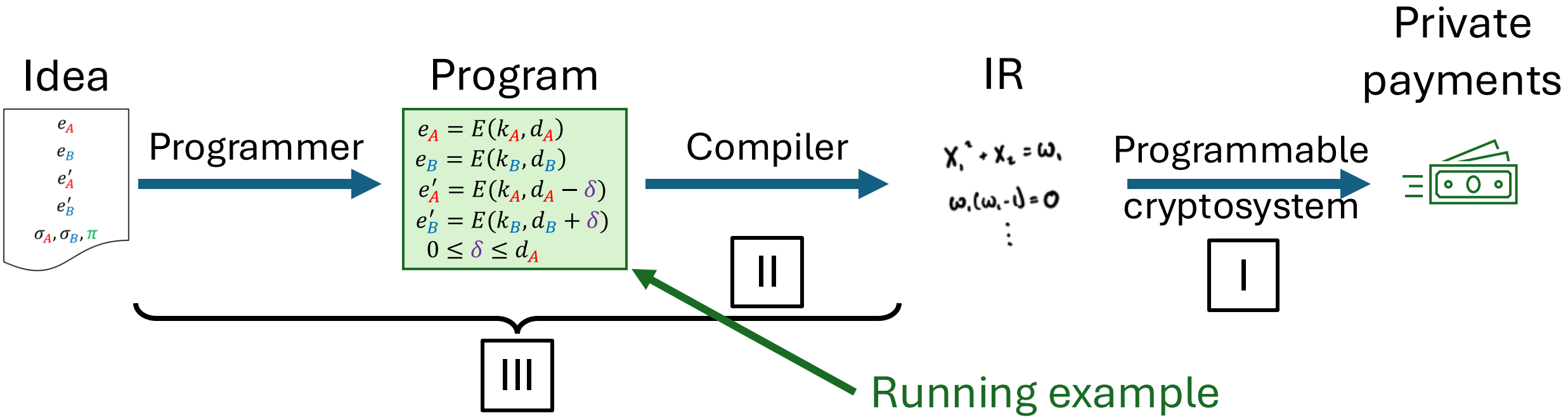
Fixing it, with programmable cryptography



Idea: add a zero knowledge proof (ZKP) π that the updates are correct.

$$e_A = E(k_A, d_A)$$
$$e_B = E(k_B, d_B)$$
$$e'_A = E(k_A, d_A - \delta)$$
$$e'_B = E(k_B, d_B + \delta)$$
$$0 \leq \delta \leq d_A$$

The full stack of programmable cryptography

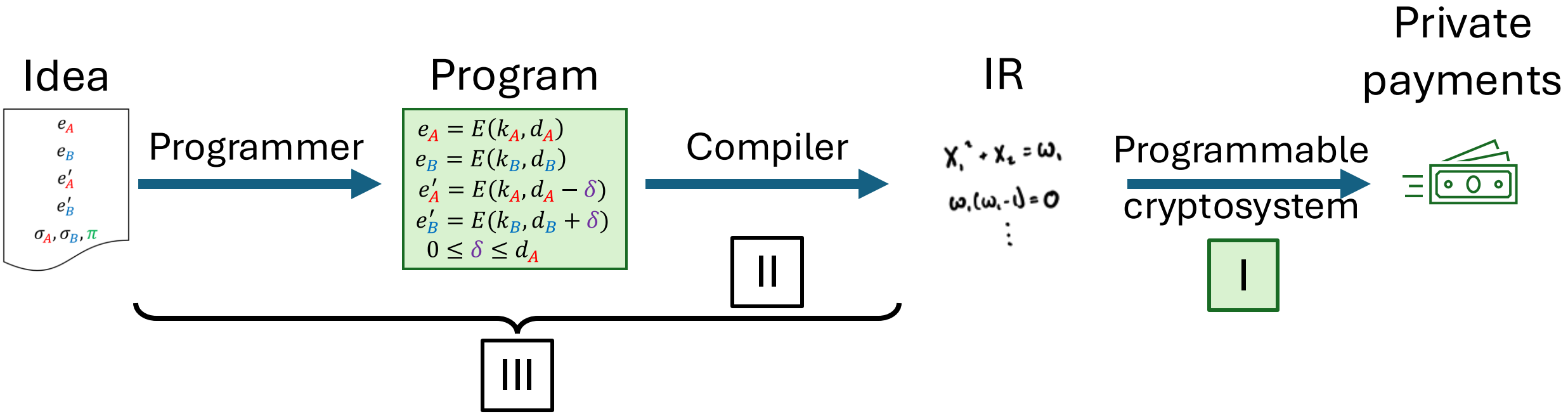


Part I: Collaborative zero knowledge

Part II: Common compiler infrastructure

Part III: Verification via finite field solving

The full stack of programmable cryptography



Part I: Collaborative zero knowledge [**Ozdemir**, Boneh; USENIX Security '22]

Part II: Common compiler infrastructure

Part III: Verification via finite field solving

What is a traditional ZKP?

Consider a property $P(x, w)$

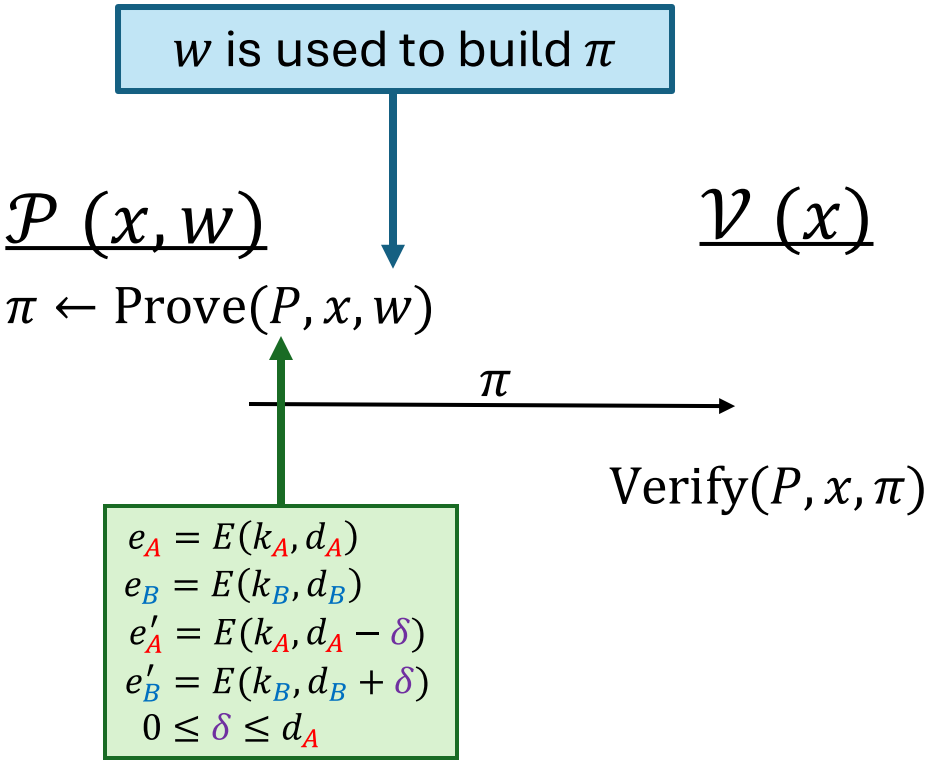
public data x (e_A, e_B, e'_A, e'_B)

secret data w $(d_A, d_B, \delta, k_A, k_B)$

A ZKP is two algorithms:

$\text{Prove}(P, x, w) \rightarrow \pi$

$\text{Verify}(P, x, \pi) \rightarrow \{\text{accept}, \text{reject}\}$



Security properties (informal):

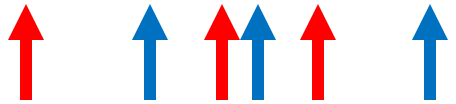
complete: a valid w yields a valid π

sound: constructing a valid π is infeasible without a valid w

zero-knowledge: π reveals no information about w

Problem: who creates the ZKP?

$w = (d_A, d_B, \delta, k_A, k_B)$ is used to build π



Alice knows some of w.

Bob knows some of w.

Problem: no **single party** can create a ZKP, because the secrets are **distributed**.

Sending w to one party is **not private**.

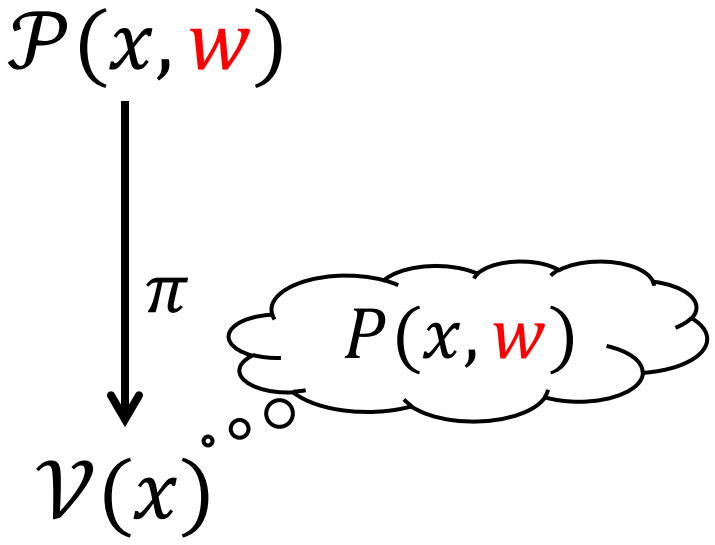
Contributions:

1. A new definition: the Collaborative ZKP
2. A very efficient construction

[Ozdemir & Boneh, USENIX Security '22]

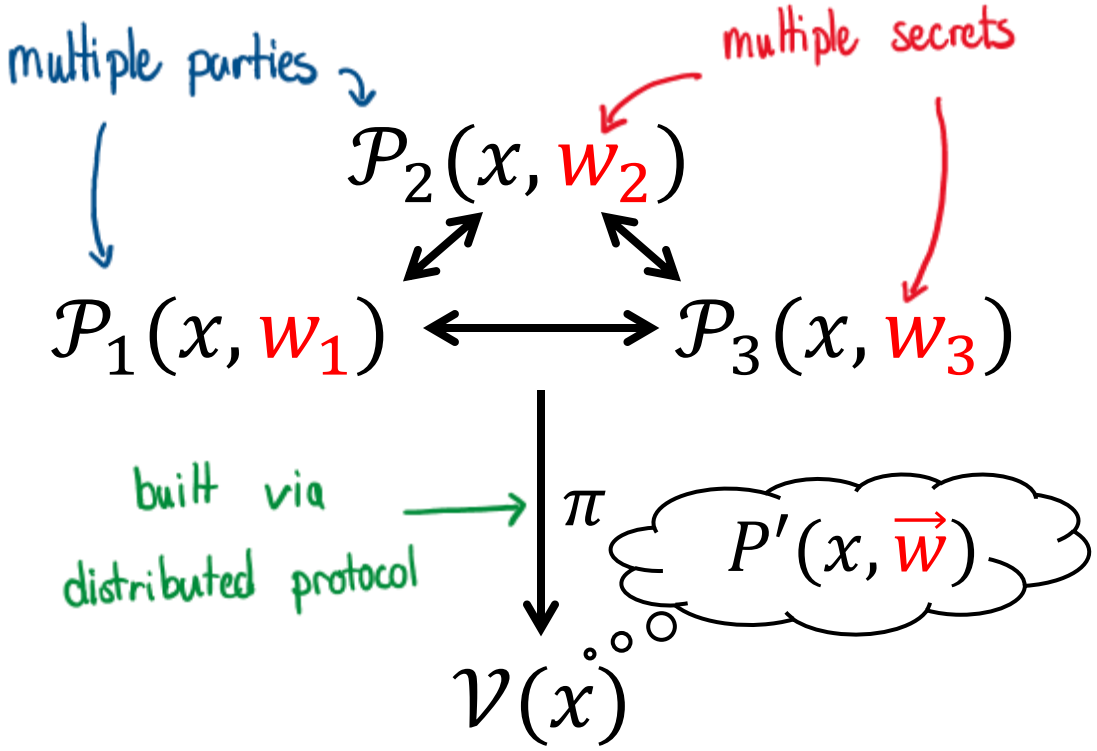
Definition: collaborative ZKPs

Prior: ZKP



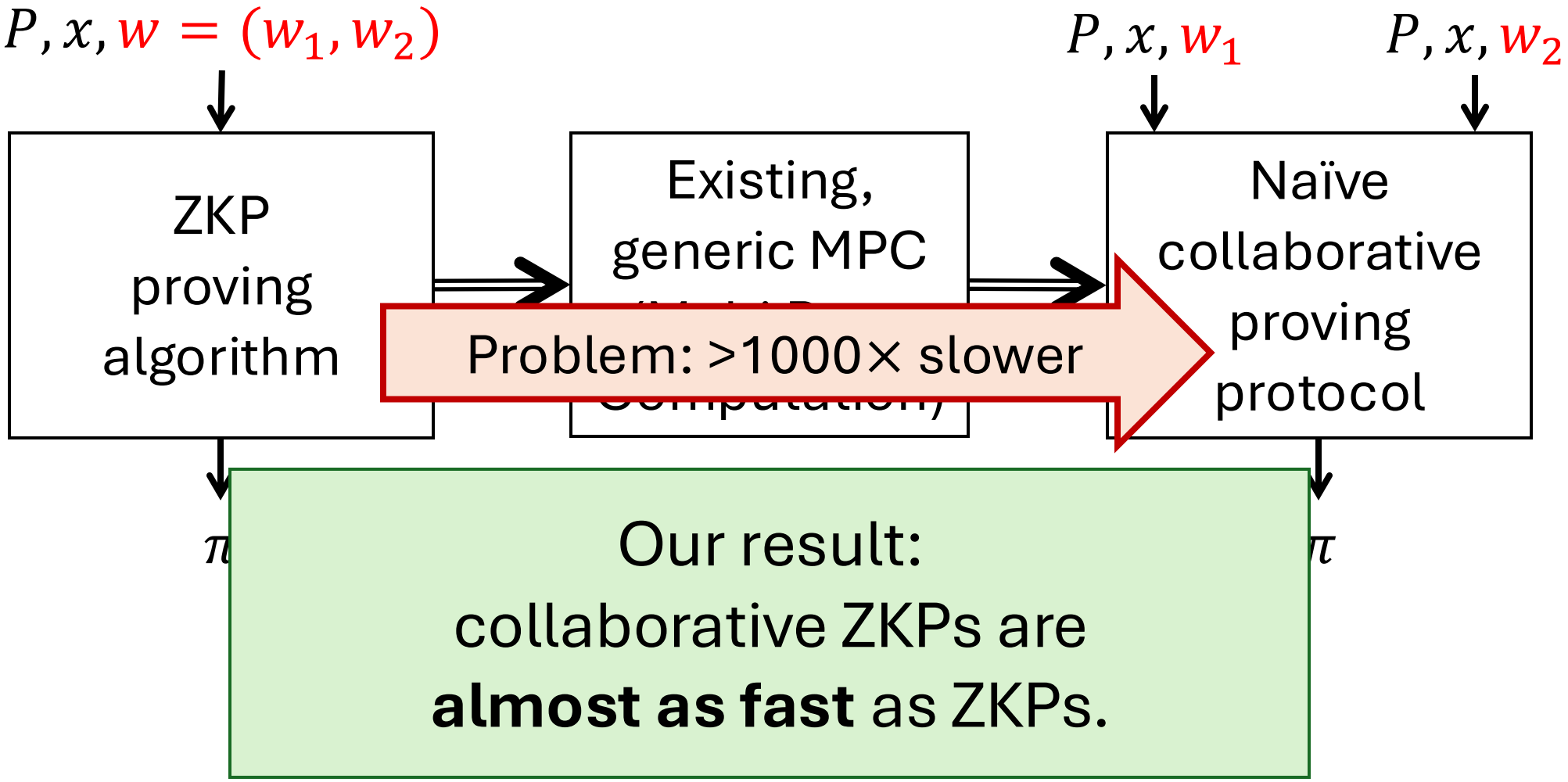
sound, complete, hides w from $\mathcal{V}$

My work: **collaborative ZKP**



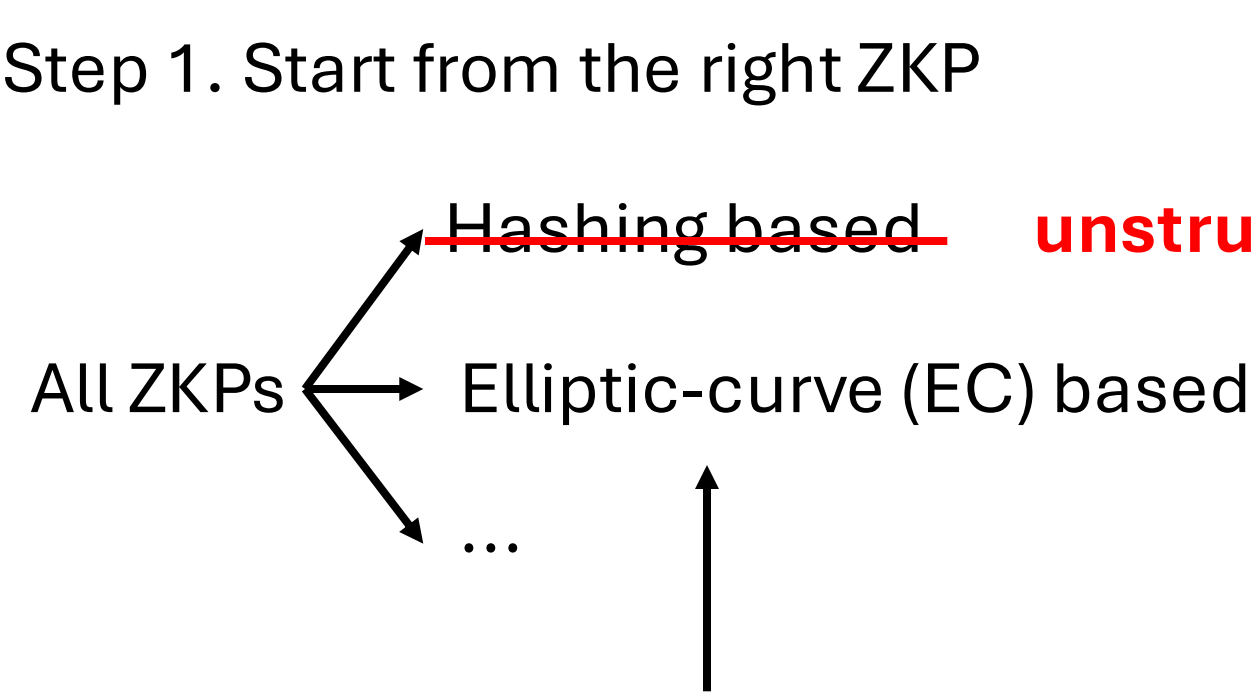
also hides w_i from $\mathcal{P}_j$

A naïve protocol



Choosing the right starting point

Step 1. Start from the right ZKP



the slowest part of proving

unstructured bottleneck

algebraic bottleneck

$$H \leftarrow \boxplus_{i=1}^{N} (z_i \boxdot G_i)$$

scalar
secret
(encodes $\vec{w}$)

EC point
public
(encodes P)

An EC: a set of p points $G, H, \ldots$ with

- addition ($G \boxplus H$)
- multiplication by scalars ($z \boxdot G$)
  - $z \in \mathbb{Z}_p, \ \mathbb{Z}_p = \{0, \ldots, p-1\}$

Intuition: how to handle the bottleneck

Step 2. Solve the bottleneck with $\mathbb{Z}_p$ **an**

$$H \leftarrow \boxplus_{i=1}^{N} (z_i \boxdot G_i)$$

Zooming out:
- This is one step in a big protocol
- Malicious security is necessary
- Sketch: lift SPDZ & GSZ'20 to ECs

		Local computation	Result:
Prover 1:	$\forall i$, gets random $z_{i,1}$	$H_1 \leftarrow \boxplus_{i=1}^{N} (z_{i,1} \boxdot G_i)$	
Prover 2:	$\forall i$, gets $z_{i,2} \leftarrow z_i - z_{i,1}$	$H_2 \leftarrow \boxplus_{i=1}^{N} (z_{i,2} \boxdot G_i)$	$H = H_1 \boxplus H_2$

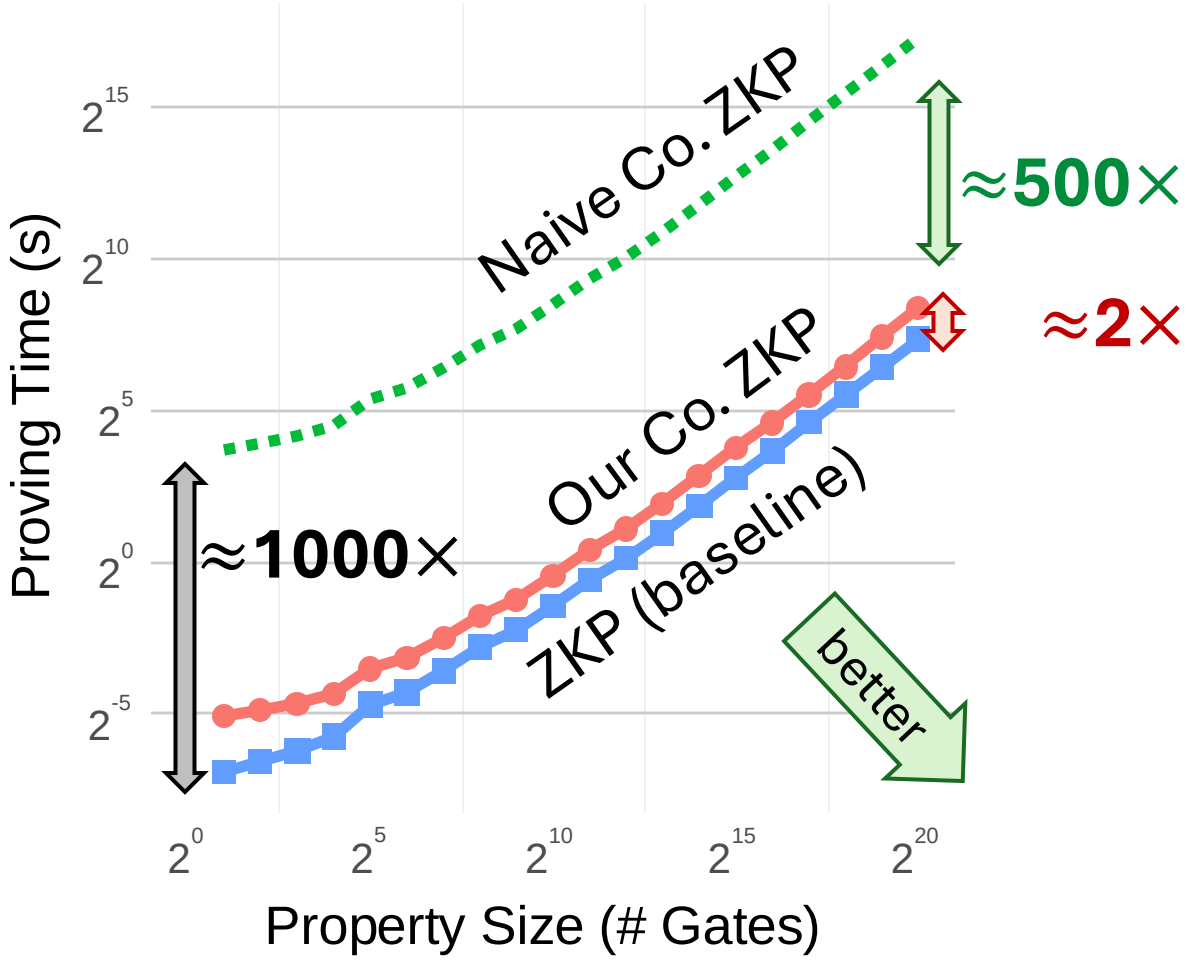
(note: $z_i = z_{i,1} + z_{i,2}$)

A $\mathbb{Z}_p$-linear sharing of z_i

Evaluated the bottleneck **without communication**

An EC-linear sharing of H

Efficient collaborative ZKPs & their impact



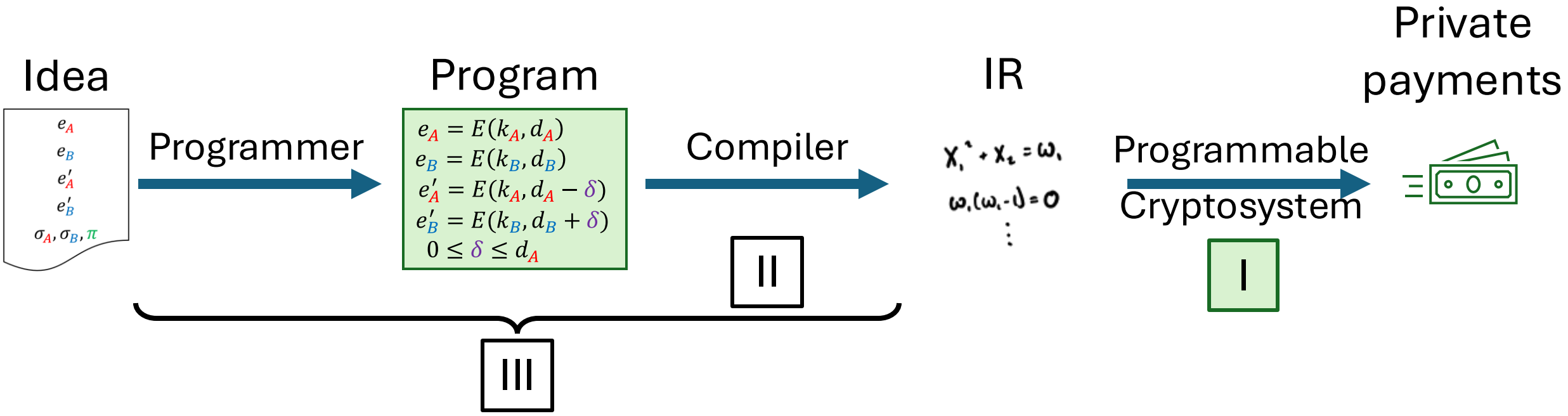
Launched an active research area

[HKKLMRRV'22], [CLM'23], [GGJPS'23], [GJMW'24], [DKR'23], [ZFS'24], [GGSSZ'25], [RGGJS'25],...

Multiple industrial implementations and compilers

Ongoing applications to private markets, ZKP outsourcing, ...

The full stack of programmable cryptography

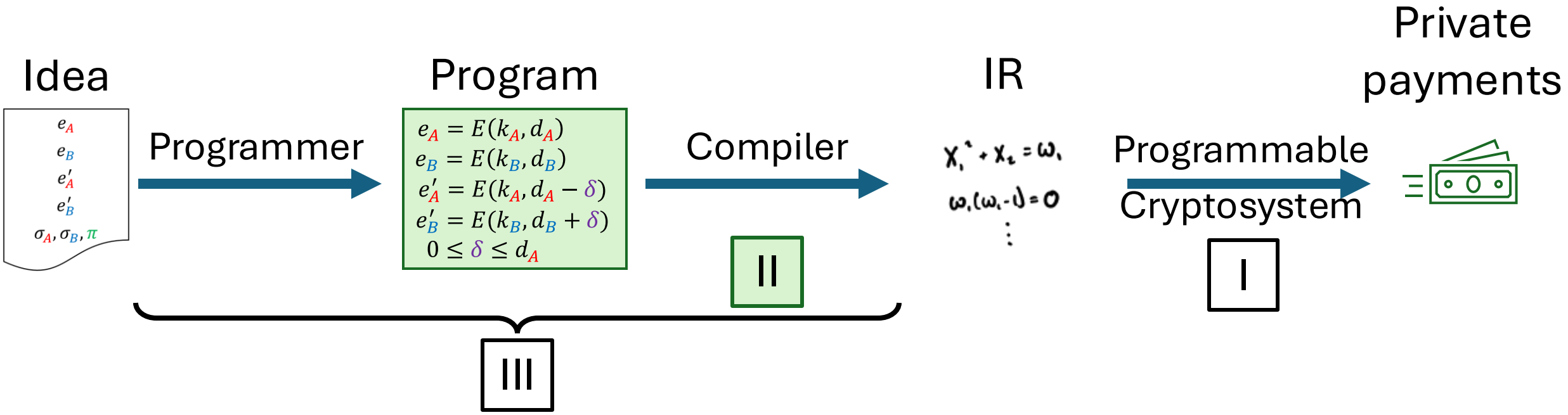


Part I: Collaborative zero knowledge

Part II: Common compiler infrastructure

Part III: Verification via finite field solving

The full stack of programmable cryptography



Part I: Collaborative zero knowledge

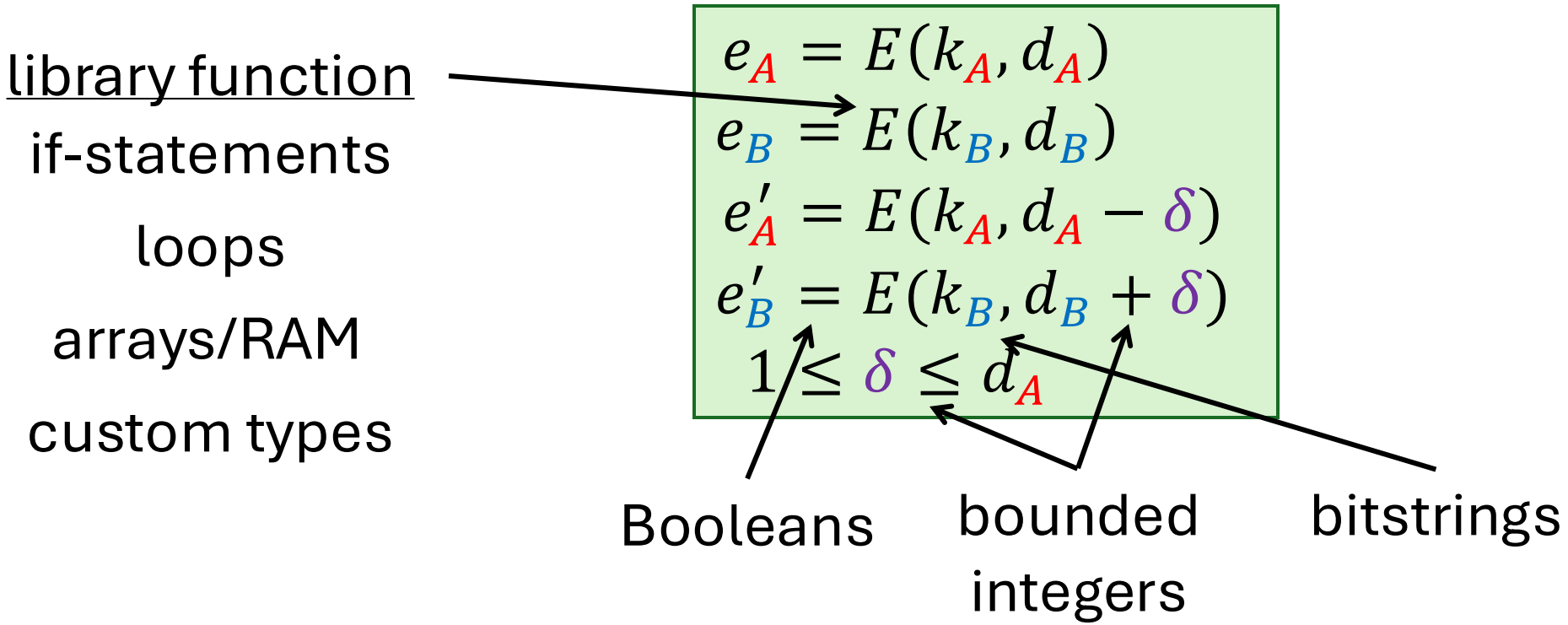
Part II: Common compiler infrastructure [**Ozdemir**, Brown, Wahby; IEEE S&P '22]

Part III: Verification via finite field solving

Problem: ZKPs for high-level programs

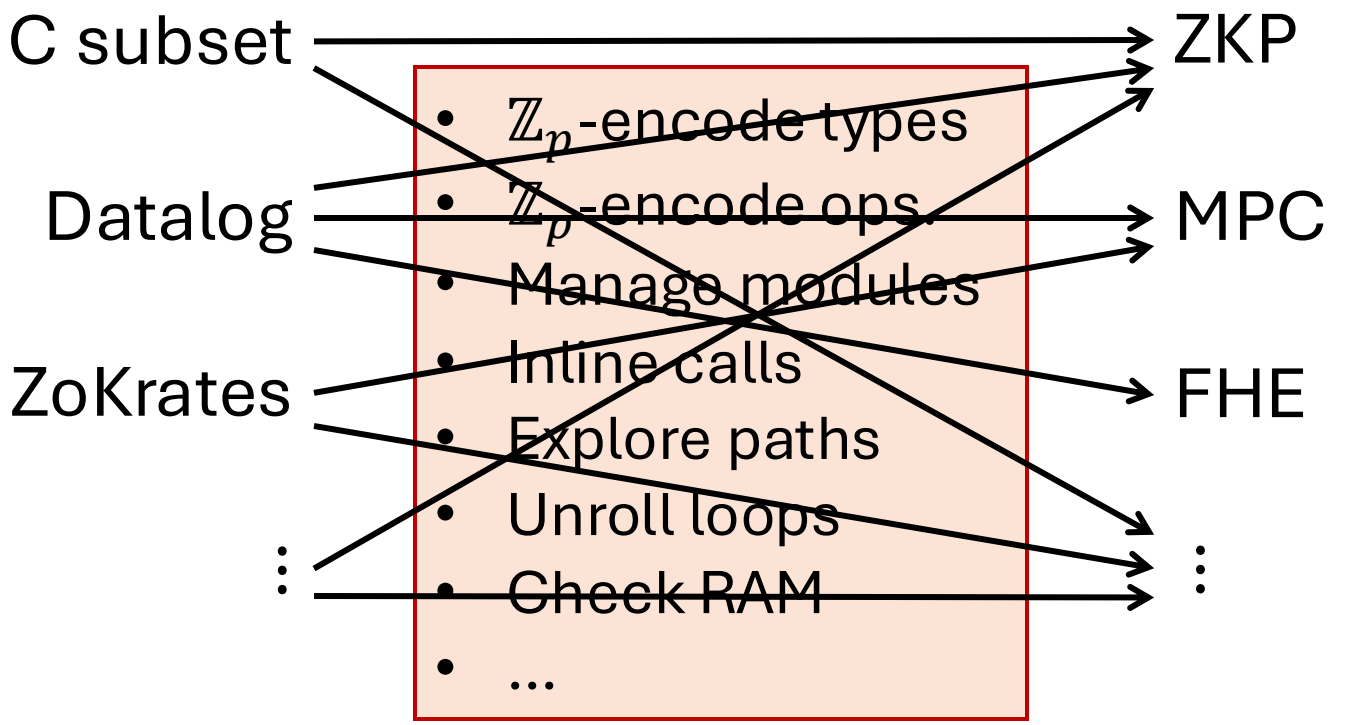
Complex applications require high-level languages

Even **simple** applications do too:



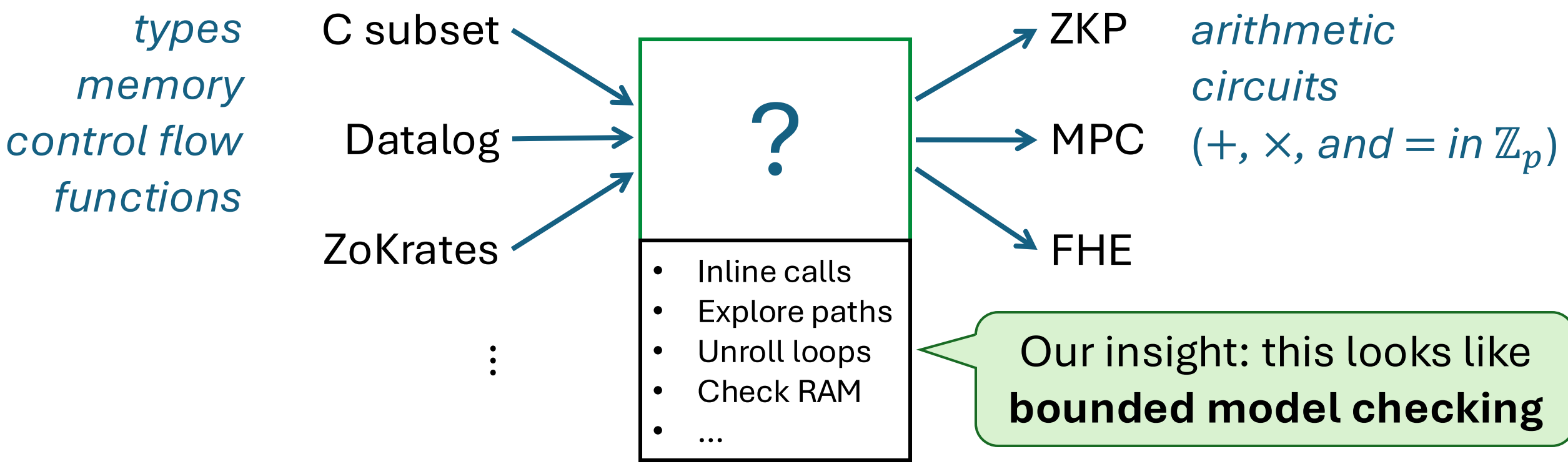
Compilation is hard

types
memory
control flow
functions



arithmetic circuits
(+, ×, and = in $\mathbb{Z}_p$*)*

Can we make common infrastructure?



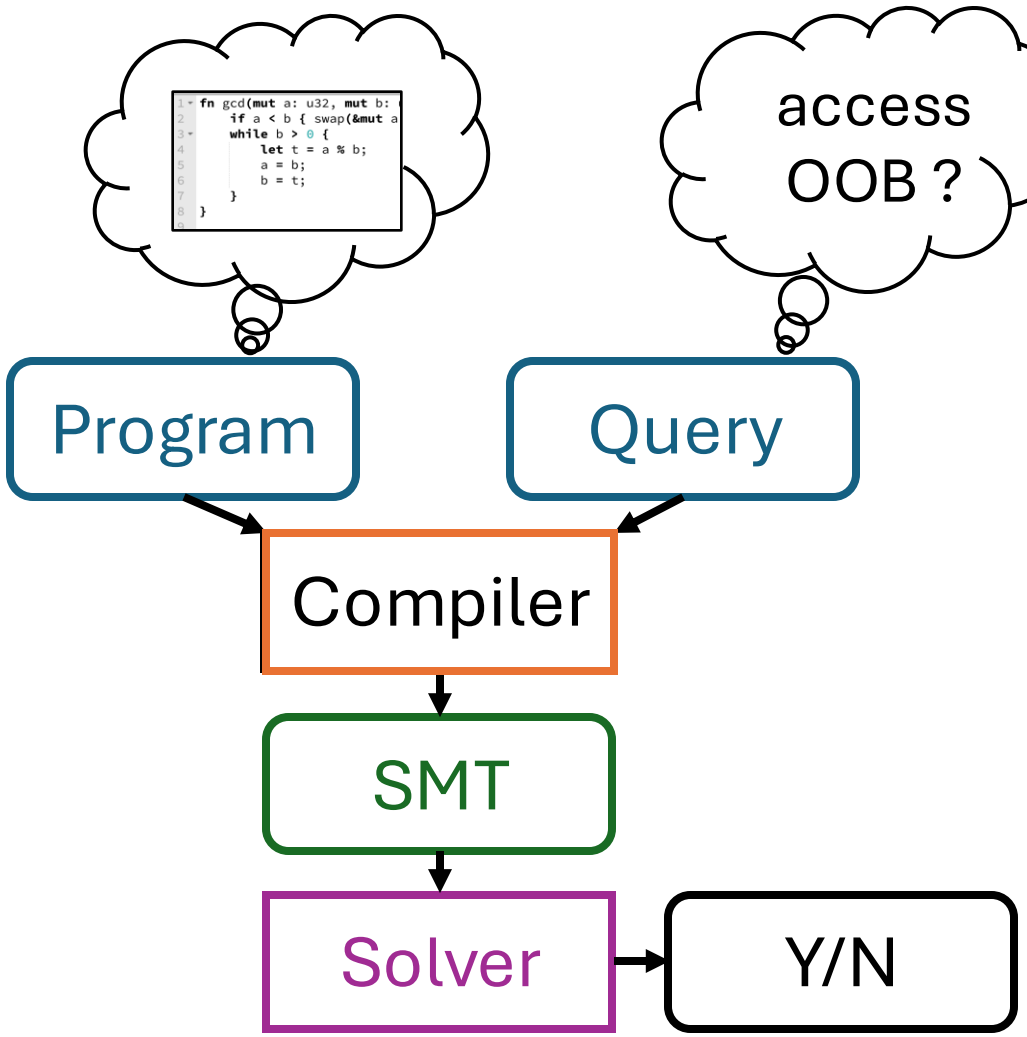
Key challenge: what is the common abstraction?

BMC: bounded model checking is compilation

1. Model the program and query as an SMT logical formula

   SMT: Satisfiability Modulo Theories
   (a many-typed formula)

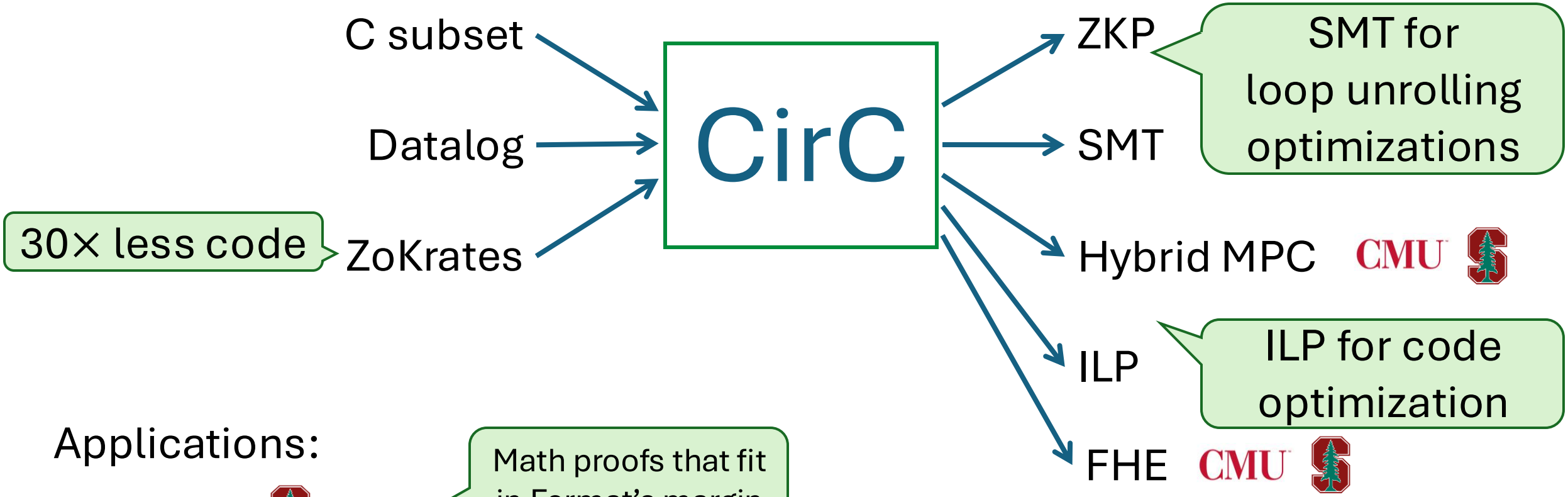
2. Check the formula with a solver



[Clarke, Emerson '81]

*the BMC connection is latent in projects like CBMC-GC

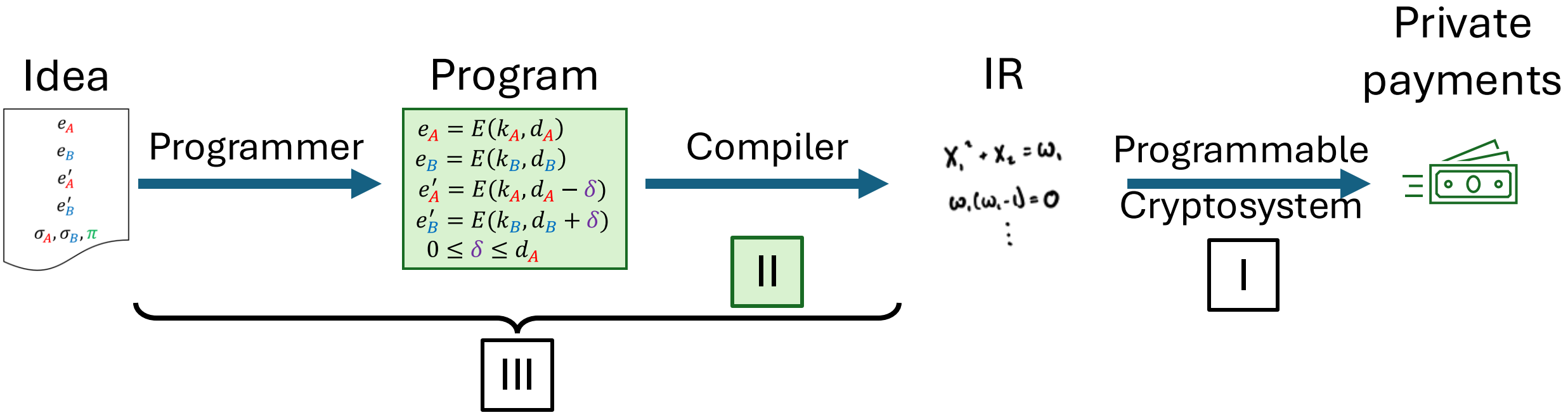
Building CirC (**Cir**cuit **C**ompiler infrastructure)



Applications:

- Lean — Math proofs that fit in Fermat's margin
- Penn — Numeric optimization
- Signature possession
- NYU — Private middleboxes
- Algorand — Cryptocurrency bridges

The full stack of programmable cryptography

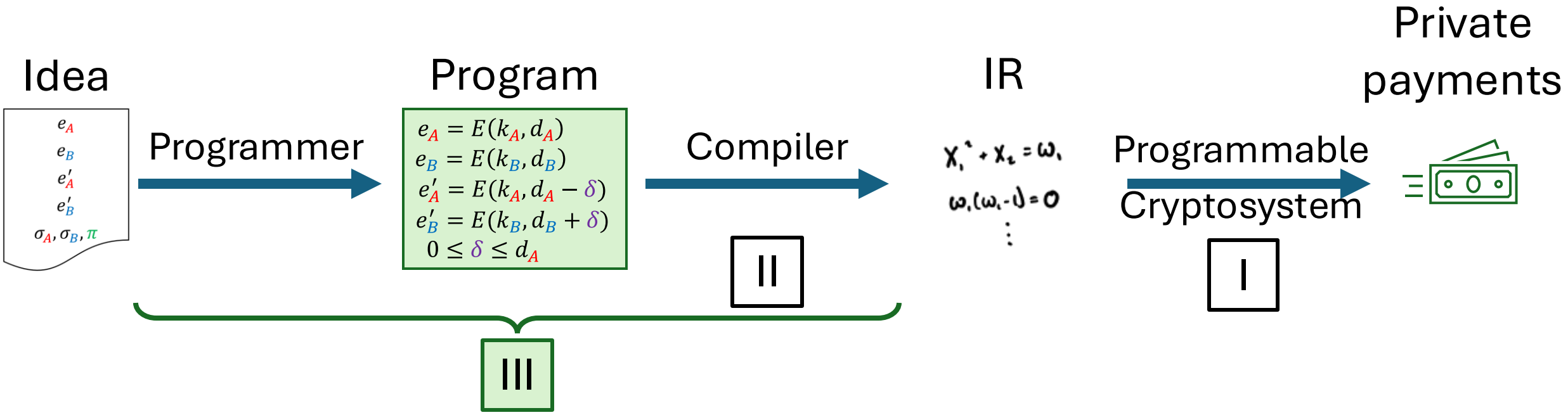


Part I: Collaborative zero knowledge

Part II: Common compiler infrastructure

Part III: SMT for finite fields

The full stack of programmable cryptography



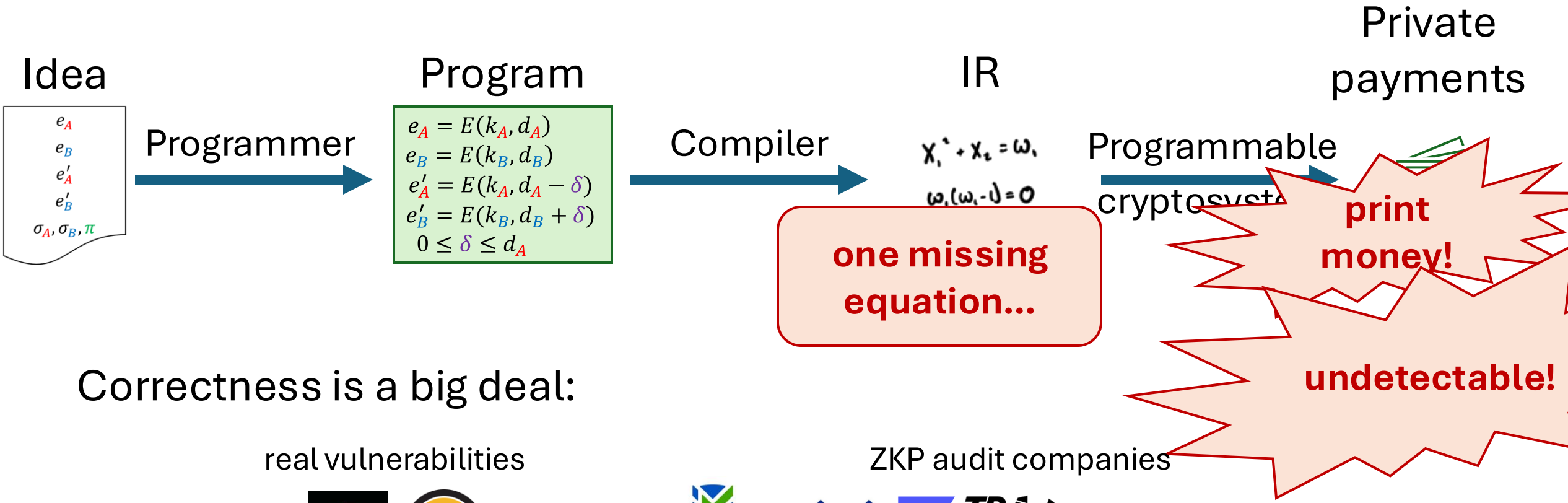
Part I: Collaborative zero knowledge

Part II: Common compiler infrastructure

Part III: SMT for finite fields

[**Ozdemir**, Kremer, Tinelli, Barrett; CAV '23]

Bugs have serious consequences



Correctness is a big deal:

real vulnerabilities

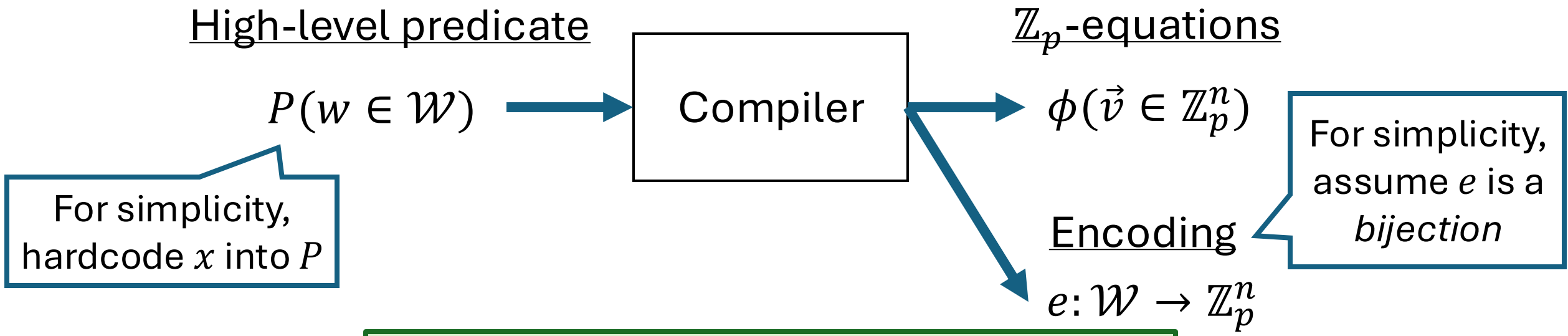
ZKP audit companies

Veridise

TRAIL OF BITS

ZKSECURITY

One kind of bug: ZKP compiler soundness



Def^n: **soundness**:
every ϕ-solution encodes a P-solution.

$$\forall \vec{v} \in \mathbb{Z}_p^n, \qquad \phi(\vec{v}) \Rightarrow P\left(e^{-1}(\vec{v})\right)$$

Idea: check soundness via satisfiability

ϕ is ***sound*** if:

$$\forall \vec{v} \in \mathbb{Z}_p^n, \qquad \phi(\vec{v}) \Rightarrow P\left(e^{-1}(\vec{v})\right)$$

ϕ is ***unsound*** if

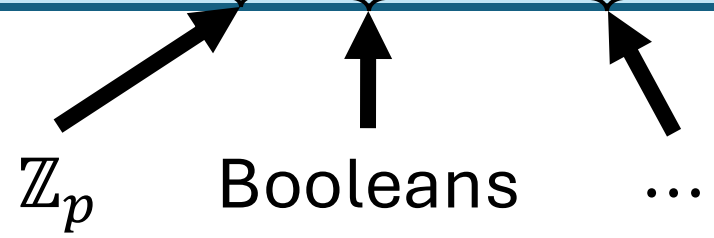
$$\neg\left(\phi(\vec{v}) \Rightarrow P\left(e^{-1}(\vec{v})\right)\right)$$

is **satisfiable**.

We need the right kind of solver

ϕ is ***unsound*** if U is satisfiable:

$$\neg\left(\underbrace{\phi(\vec{v})}\ \underbrace{\Rightarrow}\ \underbrace{P\left(e^{-1}(\vec{v})\right)}\right)$$



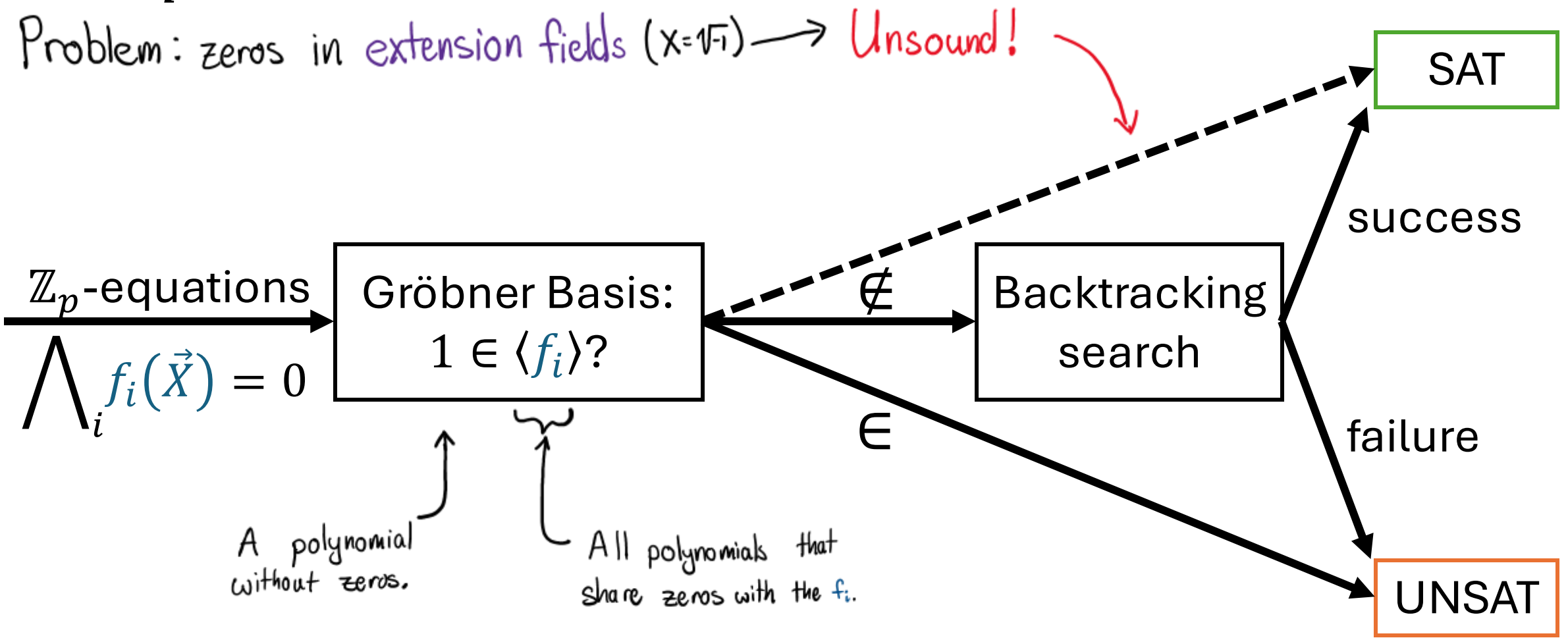
$\Rightarrow$ SMT solver

Problem:
Programmable cryptography involves $\mathbb{Z}_p$,
but no SMT solvers natively support $\mathbb{Z}_p$.

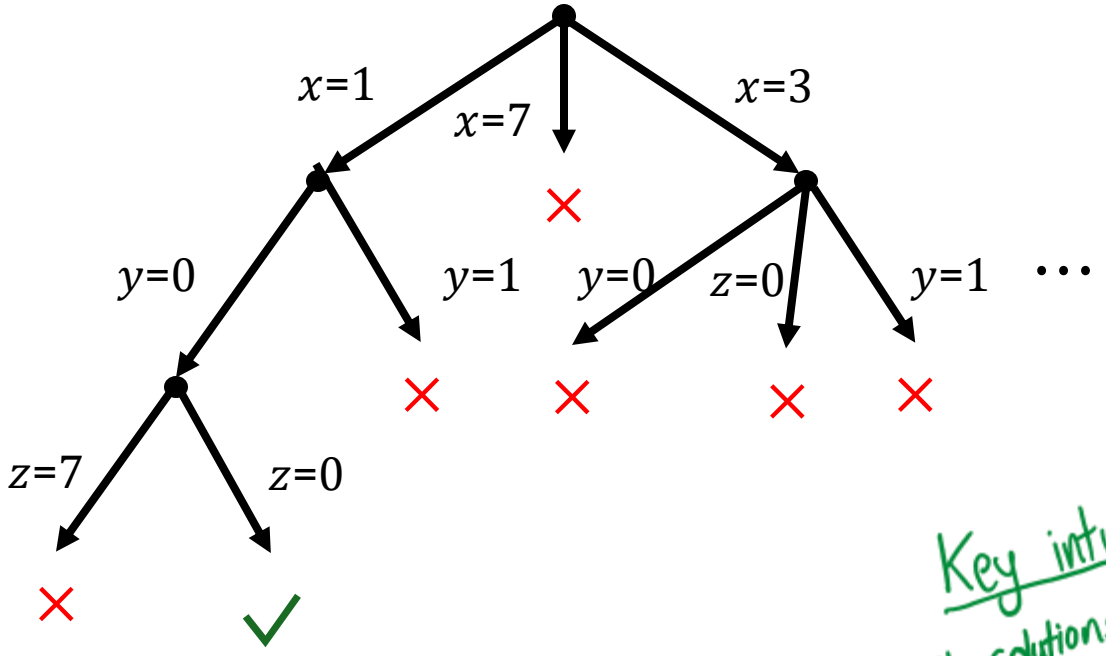
Our contribution:
The first SMT solver for $\mathbb{Z}_p$

[Ozdemir et al., CAV '23]

$\mathbb{Z}_p$-solving, Part I



$\mathbb{Z}_p$-solving, Part II: backtracking search



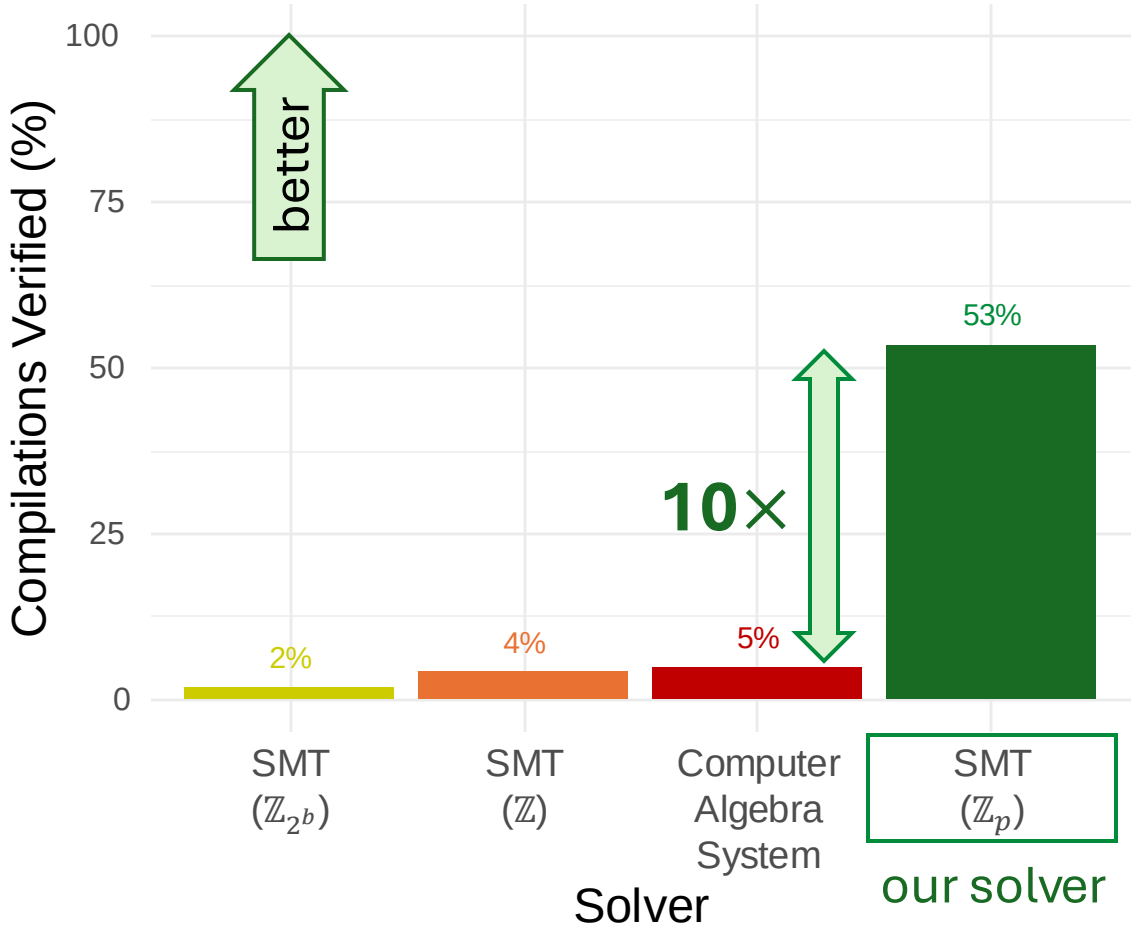
Key intuition:
(in $\overline{\mathbb{F}}_p$:) # solutions ~ $p^{\text{dim.}}$
⇒ exhaustion is reasonable

At each node:

- Compute a GB B
- Branch in one of three ways:
  - If ∃ univariate $p(X_i) \in B$:
    - factor p and branch on X_i values
  - If $\text{dimension} > 0$: random search:
    - $X_1 = 1$,
    - $X_2 = 1$,
    - $X_1 = 2$,
    - $X_2 = 2$, …
  - Else, if $\text{dimension} = 0$:
    - Use B to *compute* a univariate $p(X_i) \in \langle B \rangle$

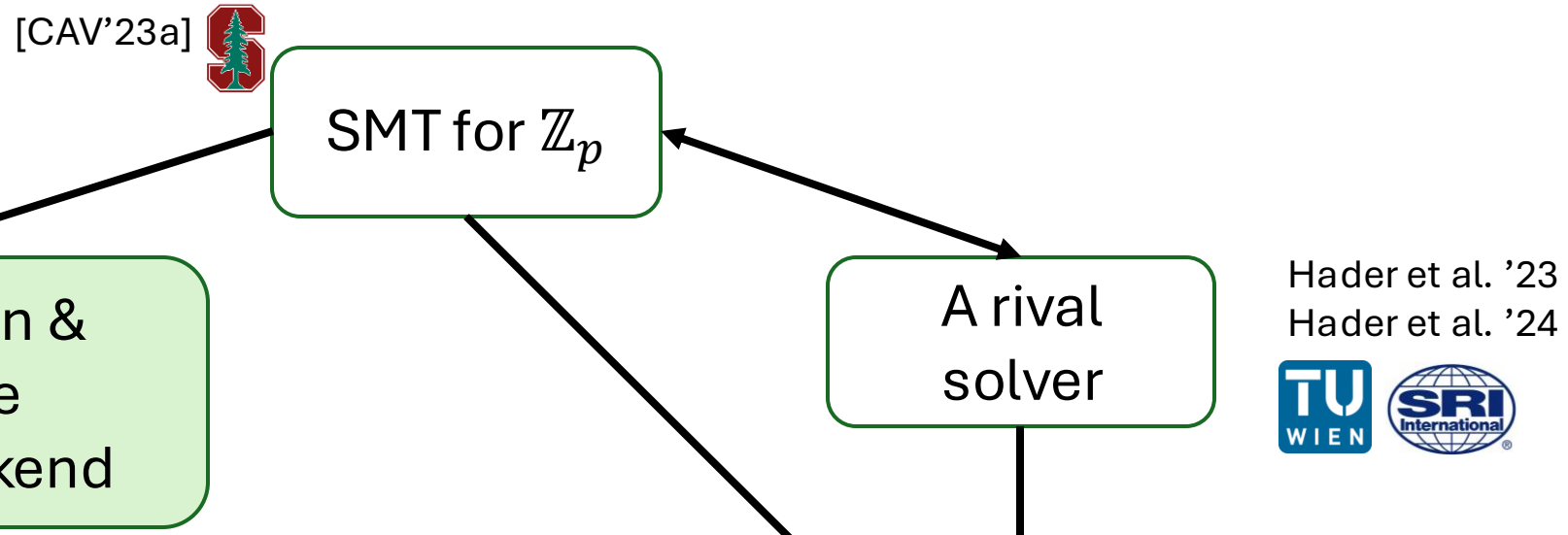
Experiment: translation validation

Verifying compilations of random* Boolean programs.



* With 2-12 input variables and 16-128 intermediate operators.

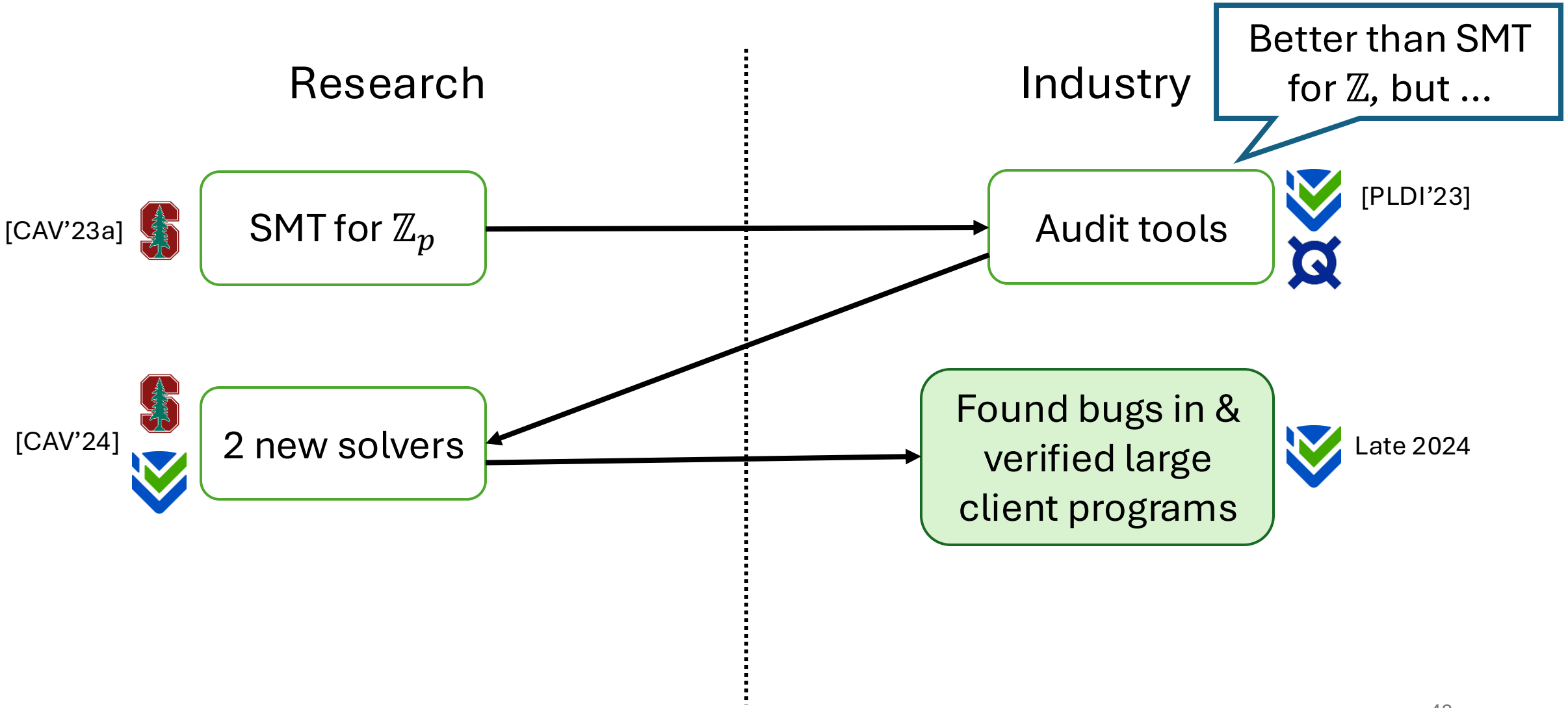
Academic follow-up & impact



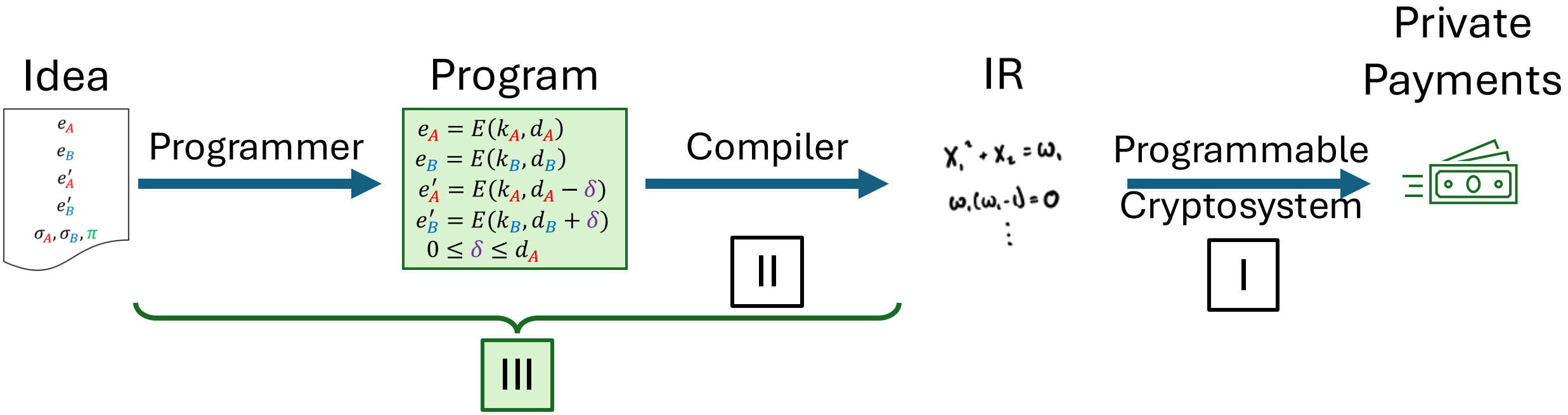
- 1st ZKP compiler correctness definition
  - **compilers** + **cryptography**
- 1st verified ZKP compiler pass
  - **compilers** + **verification**
- **Verification failed!?**
  - A miscompilation in our example
- After patches, **verification passed**

$$e_A = E(k_A, d_A)$$
$$e_B = E(k_B, d_B)$$
$$e'_A = E(k_A, d_A - \delta)$$
$$e'_B = E(k_B, d_B + \delta)$$
$$0 \le \delta \le d_A$$

Industrial use



The full stack of programmable cryptography



Part I: Collaborative ZKPs

Part II: Common Compiler Infrastructure

Part III: SMT for finite fields

Part IV:
What’s next?

Now is the right time

Encryption is ubiquitous

- **Good news:** people want privacy
- **Next frontier:** **data in use**

Programmable cryptography is becoming practical

- Deployments: payments (Monero/Zcash), statistics (iOS/Android), …
- **Good news:** efficient *enough*
- **Bad news:** so far, it requires **application & cryptographic** expertise

A stack that **separates concerns** will unlock more applications

The full stack of programmable cryptography

Idea

Programmer

Program

```
fn() → bool {
  ...
}
```

Compiler

IR

$$X_1^2 + X_2 = \omega_1$$

$$\omega_1(\omega_1 - 1) = 0$$

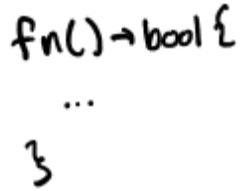
$$\vdots$$

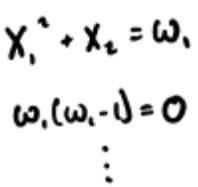
Programmable cryptosystem

System

Future work:

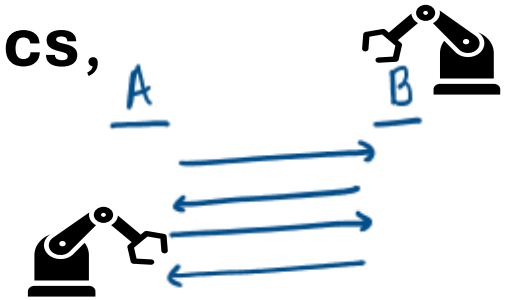
Idea → Programmer → Program → Compiler → IR → Programmable Cryptosystem → System





Formalizing compilation

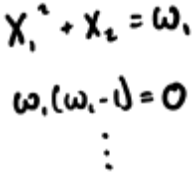
Can we **formalize compiler semantics**,
to discover **new optimizations**?
Perhaps for interactive proofs?



Starting from [“Algebraic Interactive Proofs” Ozdemir et al. ’25]

Future work:

Idea → Programmer → Program → Compiler → IR → Programmable Cryptosystem → System



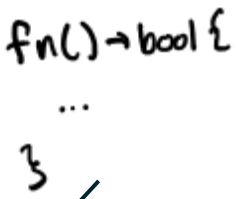
Cryptosystems for new models

Can we build cryptosystem for **new computational models**? Perhaps **sequential** digital logic?

Inspired by [“Tri-State Circuits” Heath et al. ’23]

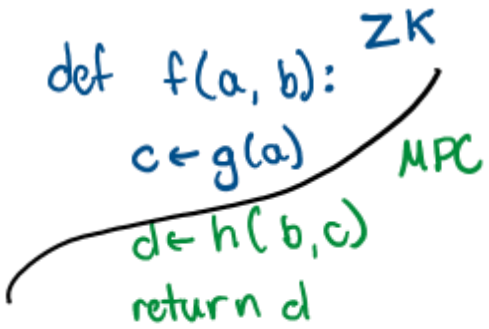
Future work:

Idea → Programmer → Program → Compiler → IR → Programmable Cryptosystem → System



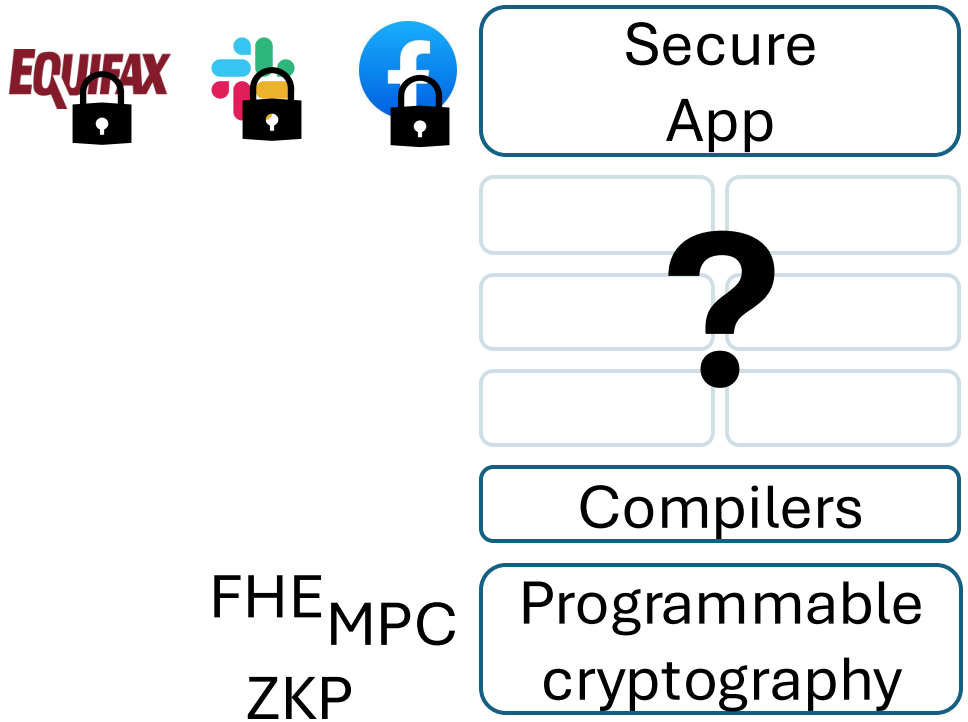
Combining cryptosystems

Can we compile **one program** to **multiple cryptosystems** to improve **security** or **performance**?

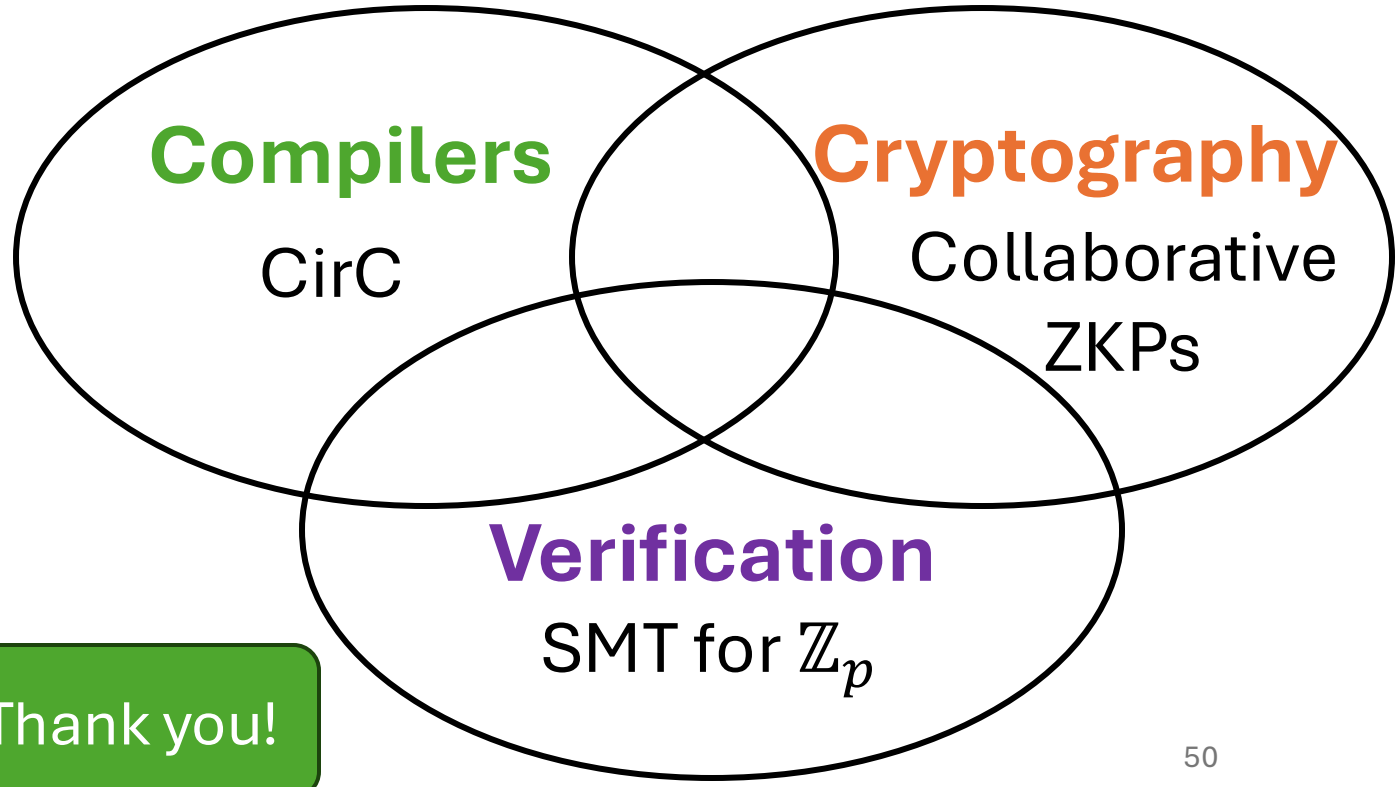


Building on [“Viaduct” Acay et al. ’21]

My vision:
The **full stack** of programmable cryptography



My work, so far:



Thank you!