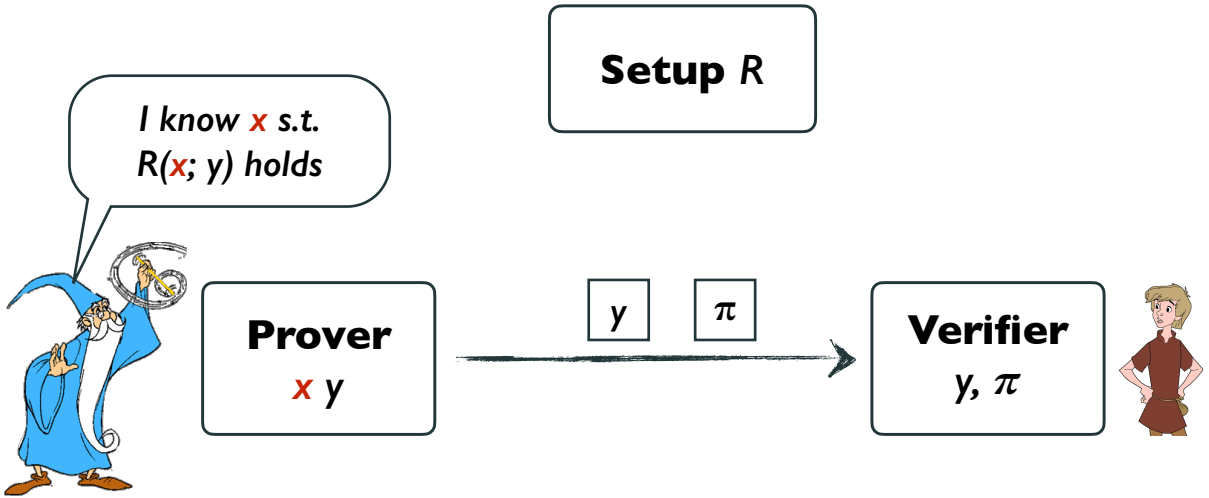
PLINK: Verified Constraints Generation for Zero Knowledge Proofs (PLONK)

•••

Pablo Castellanos, Ignacio Cascudo, Dario Fiore, Niki Vazou
IMDEA Software Institute

Zero Knowledge Proofs (ZKP)



from theoretical feasibility to real systems

- **1992** succinct arguments [Kilian92, Micali94]
- …
- **2013** first implemented systems
- **2018** app to cryptocurrencies
- **2023** app to ML
- privacy-preserving applications

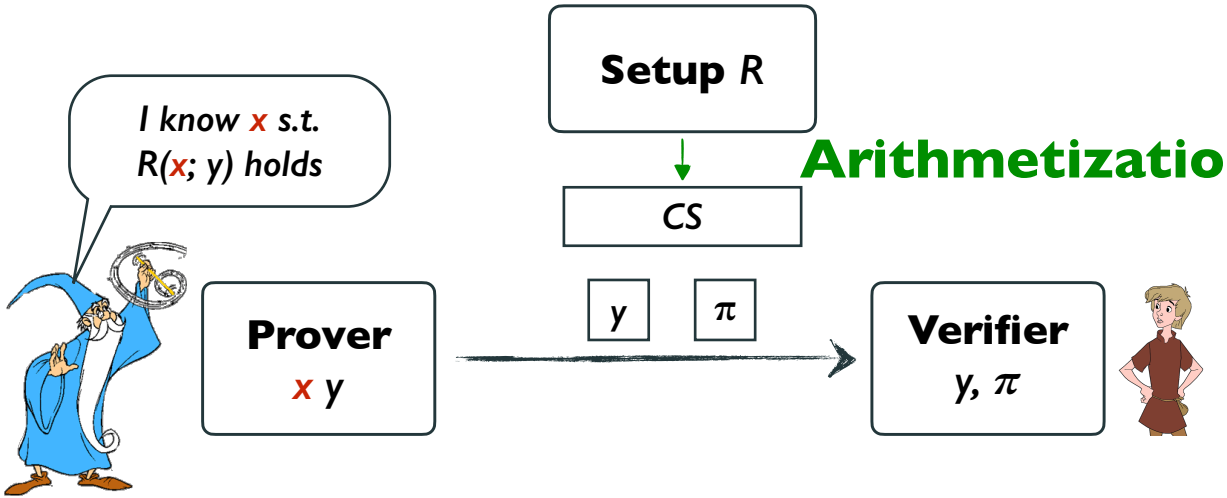
Knowledge Soundness: V accepts only when the P knows a valid *x*

Zero-knowledge: π hides *x*

Succinctness: proofs are short

Generality: can encode any NP relation *R*

Zero Knowledge Proofs (ZKP)



from theoretical feasibility to real systems

- 1992 — succinct arguments [Kilian92, Micali94]
- …
- 2013 — first implemented systems
- 2018 — app to cryptocurrencies
- 2023 — app to ML
- privacy-preserving applications

Knowledge Soundness: V accepts only when the P knows a valid x

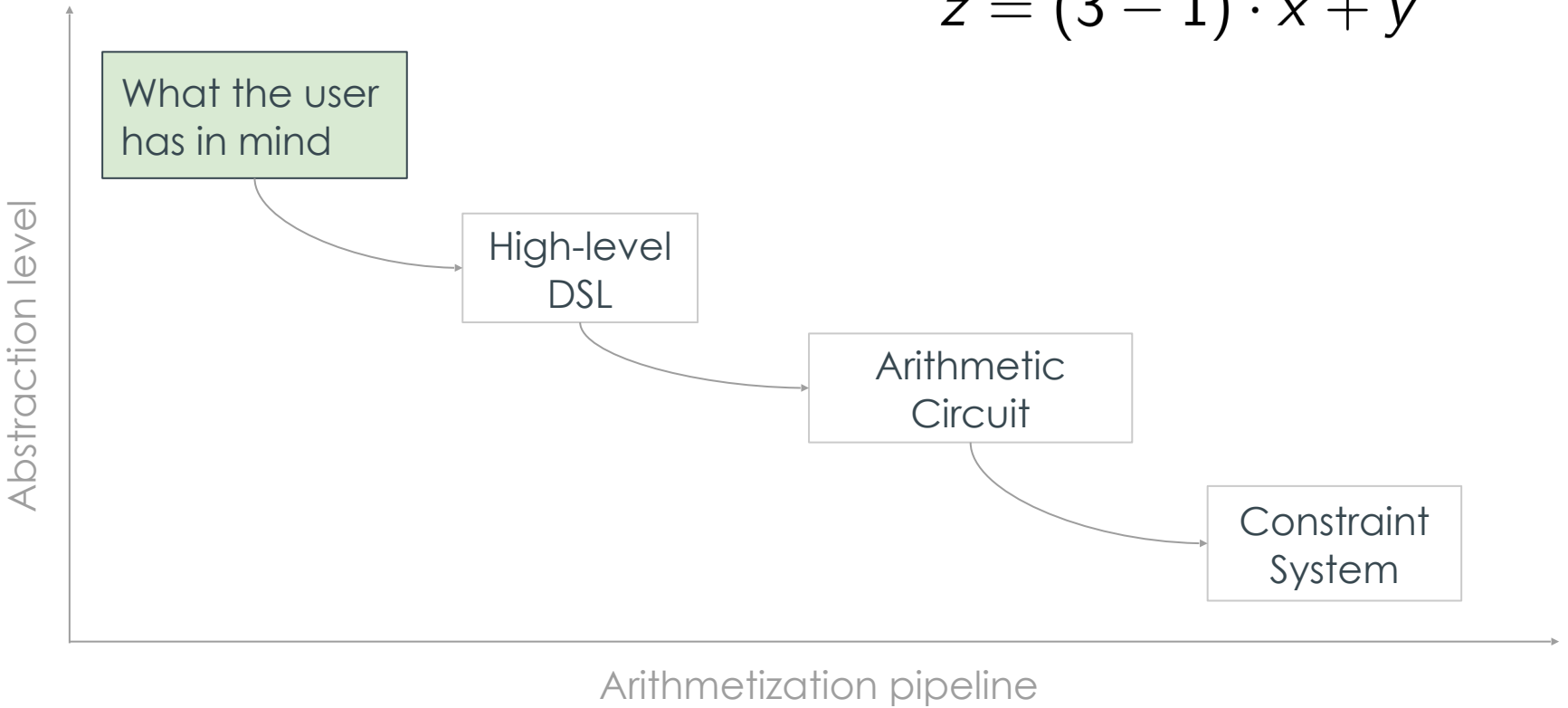
Zero-knowledge: π hides x

Succinctness: proofs are short

Generality: can encode any NP relation R

Arithmetization: Overview

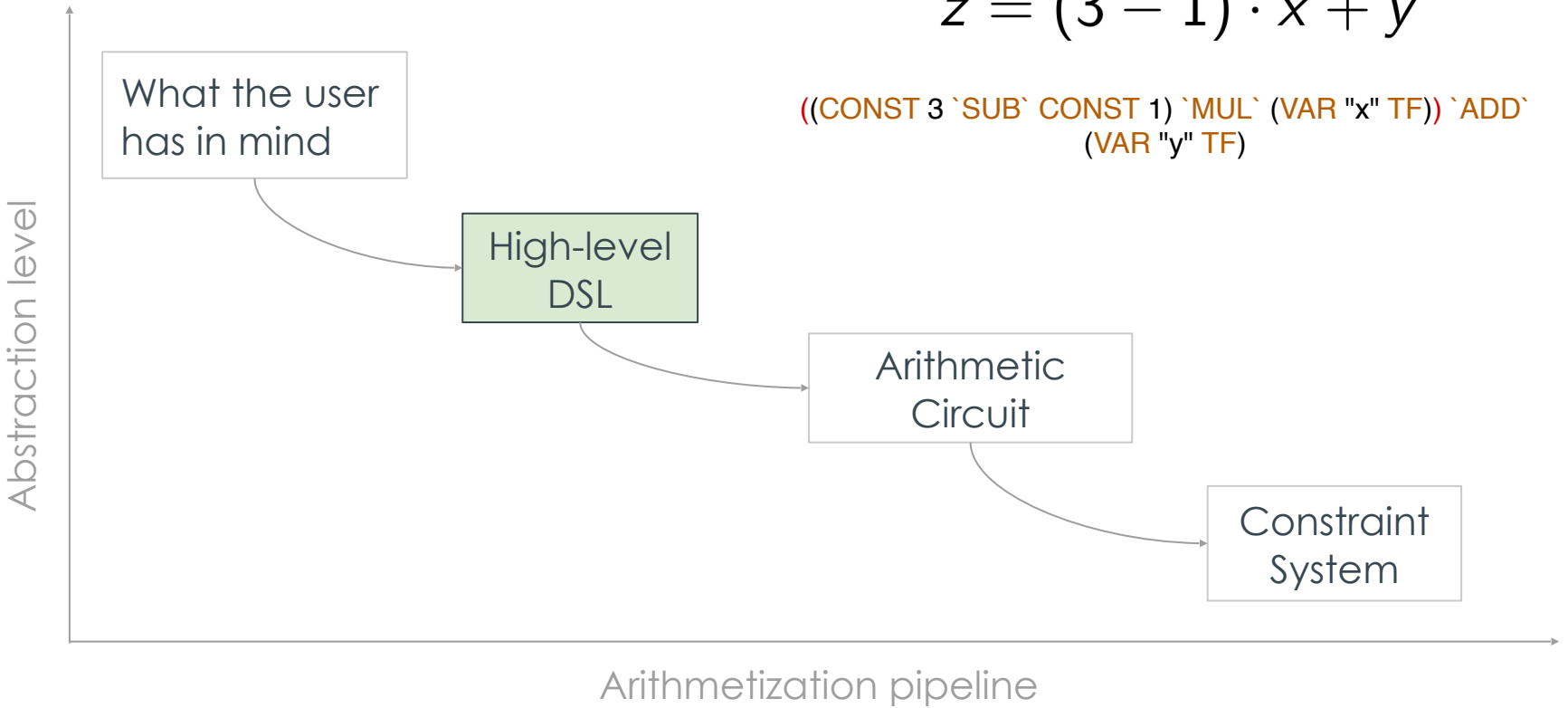
$$z = (3 - 1) \cdot x + y$$



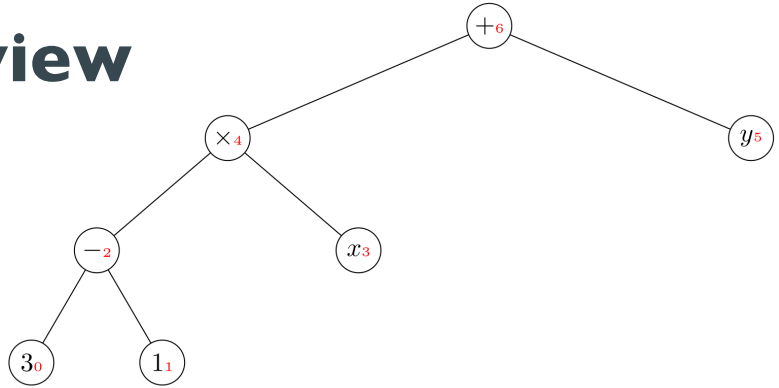
Arithmetization: Overview

$$z = (3 - 1) \cdot x + y$$

((CONST 3 `SUB` CONST 1) `MUL` (VAR "x" TF)) `ADD` (VAR "y" TF)



Arithmetization: Overview



What the user has in mind

High-level DSL

Arithmetic Circuit

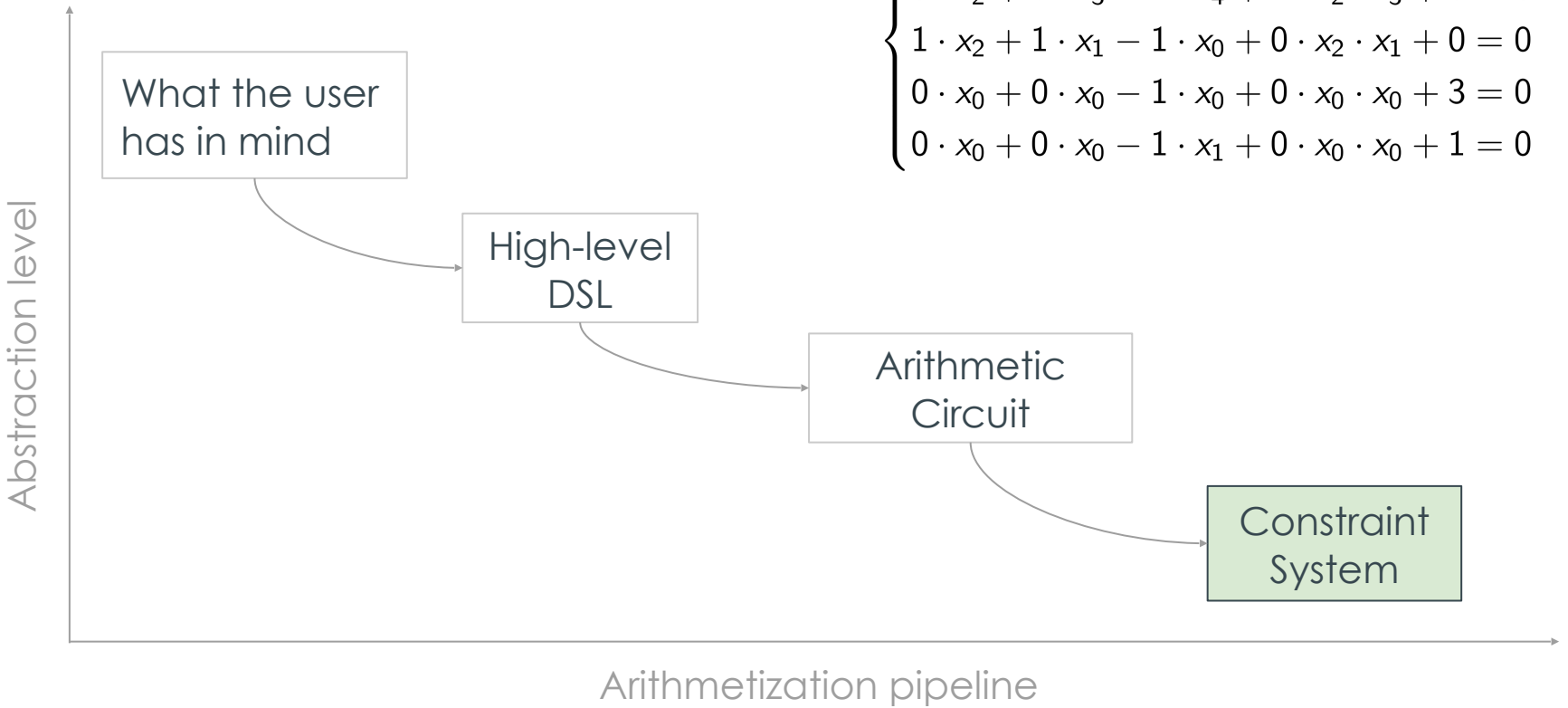
Constraint System

Abstraction level

Arithmetization pipeline

Arithmetization: Overview

$$\begin{cases} 1 \cdot x_4 + 1 \cdot x_5 - 1 \cdot x_6 + 0 \cdot x_4 \cdot x_5 + 0 = 0 \\ 0 \cdot x_2 + 0 \cdot x_3 - 1 \cdot x_4 + 1 \cdot x_2 \cdot x_3 + 0 = 0 \\ 1 \cdot x_2 + 1 \cdot x_1 - 1 \cdot x_0 + 0 \cdot x_2 \cdot x_1 + 0 = 0 \\ 0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_0 + 0 \cdot x_0 \cdot x_0 + 3 = 0 \\ 0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_1 + 0 \cdot x_0 \cdot x_0 + 1 = 0 \end{cases}$$



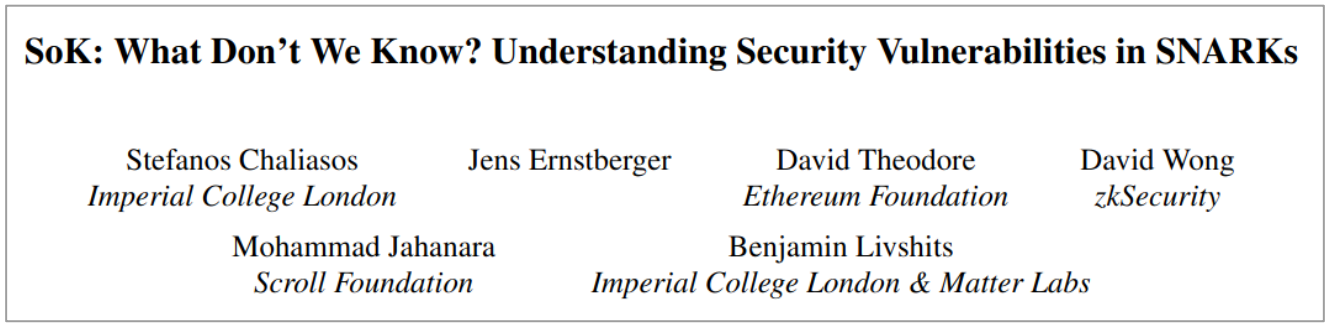
Arithmetization is Error Prone

Systematic Study of **141** known vulnerabilities of ZK proof systems

99/141: errors in the description (or coding) of the arithmetic relation

06/141: bugs in the compilation process (DSL → CS)

36/141: non arithmetization errors



SoK: What Don't We Know? Understanding Security Vulnerabilities in SNARKs

Stefanos Chaliasos
Imperial College London

Jens Ernstberger

David Theodore
Ethereum Foundation

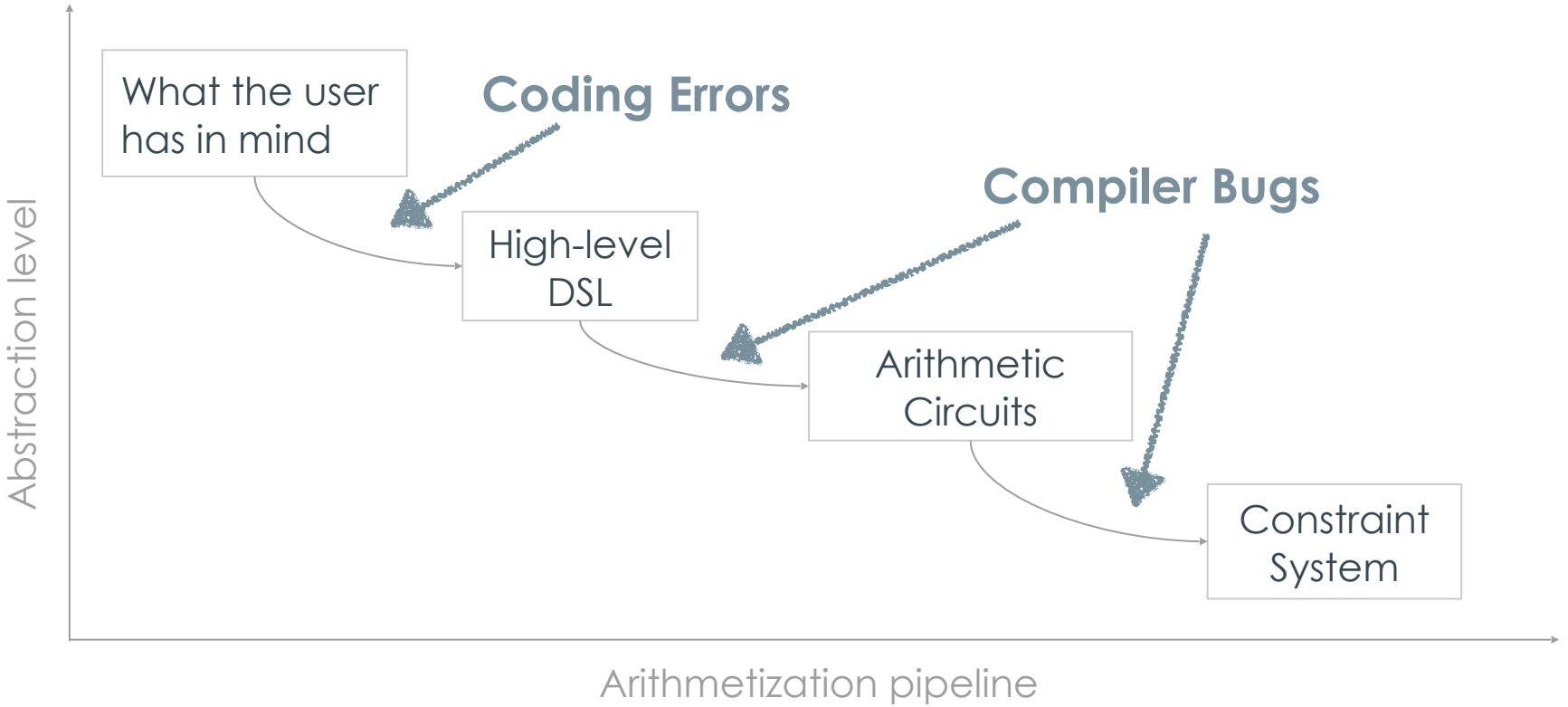
David Wong
zkSecurity

Mohammad Jahanara
Scroll Foundation

Benjamin Livshits
Imperial College London & Matter Labs

USENIX'24

Arithmetization is Error Prone



Formal Verification to the Rescue

Types of Verification:

Functional: the higher-level programs satisfy specifications

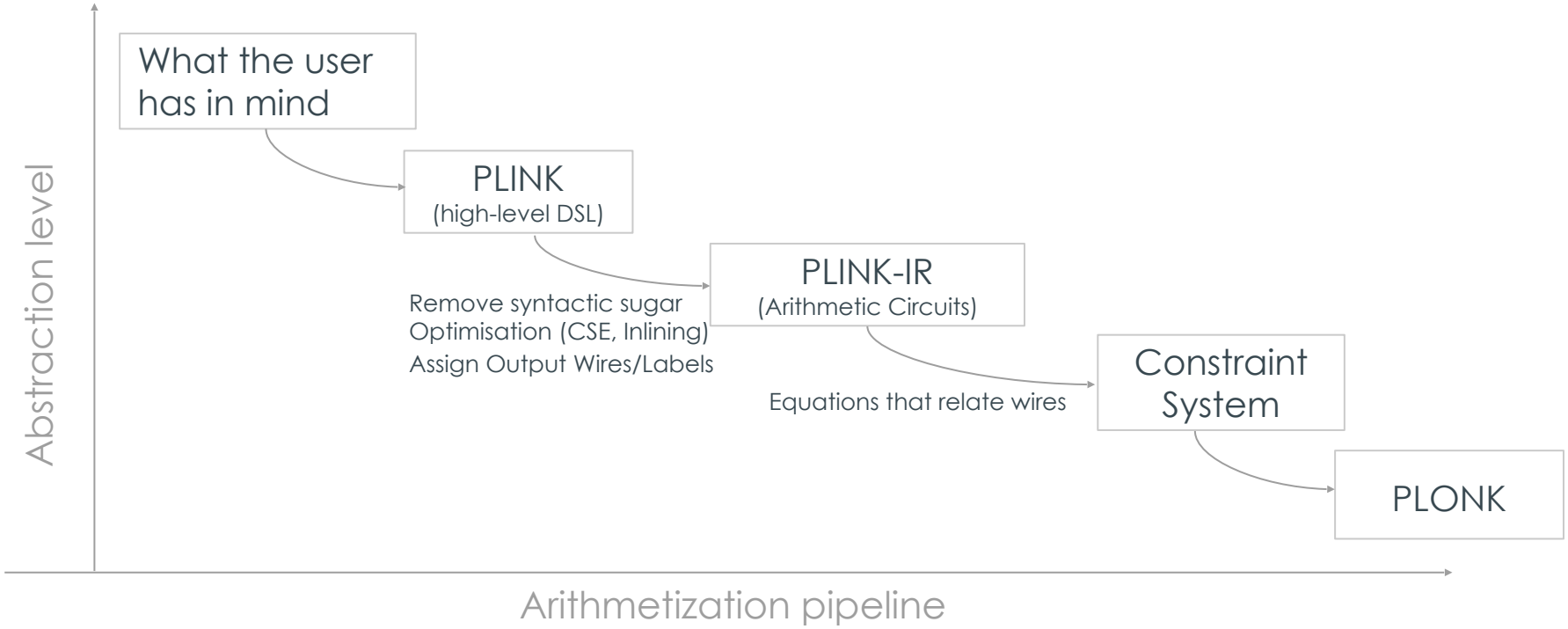
Compiler: generated CS preserve high level semantics

CS: the generated CS satisfy given specifications

PLINK

Programming Language for INtegrity and Knowledge

PLINK: DSL designed in Liquid Haskell, with types (bool, FT, Vector)



Example: Modular Addition

Given x and y, how to encode $z = x + y \pmod{32}$?

- New boolean variable b that represents the "overflow".
- Add equality constraint $x + y = z + b \cdot 32$.
- Add *in*equality constraint enforcing $0 \leq z < 32$.

Example: Modular Addition in PLINK

```
{-@ type FieldDSL p = {v:DSL p | typed v TF} @-}

{-@ addMod :: Field p => FieldDSL p → FieldDSL p → GlobalStore p (FieldDSL p) @-}
addMod :: Field p => DSL p → DSL p → GlobalStore p (DSL p)
addMod x y = do

  let b = VAR "overflow" TF
  let z = VAR "sum" TF

  witnessGenHint b (\x y -> if x + y < 32 then 0 else 1) x y
  witnessGenHint z (\x y -> (x + y) `mod` 32) x y

  assert $ BOOL b
  assert $ (x `ADD` y) `EQA` (z `ADD` (b `MUL` CONST 32))
  withNBits 5 z -- z can be encoded using 5 bits, so 0 <= z < 32

  return z
```

Add refinement types

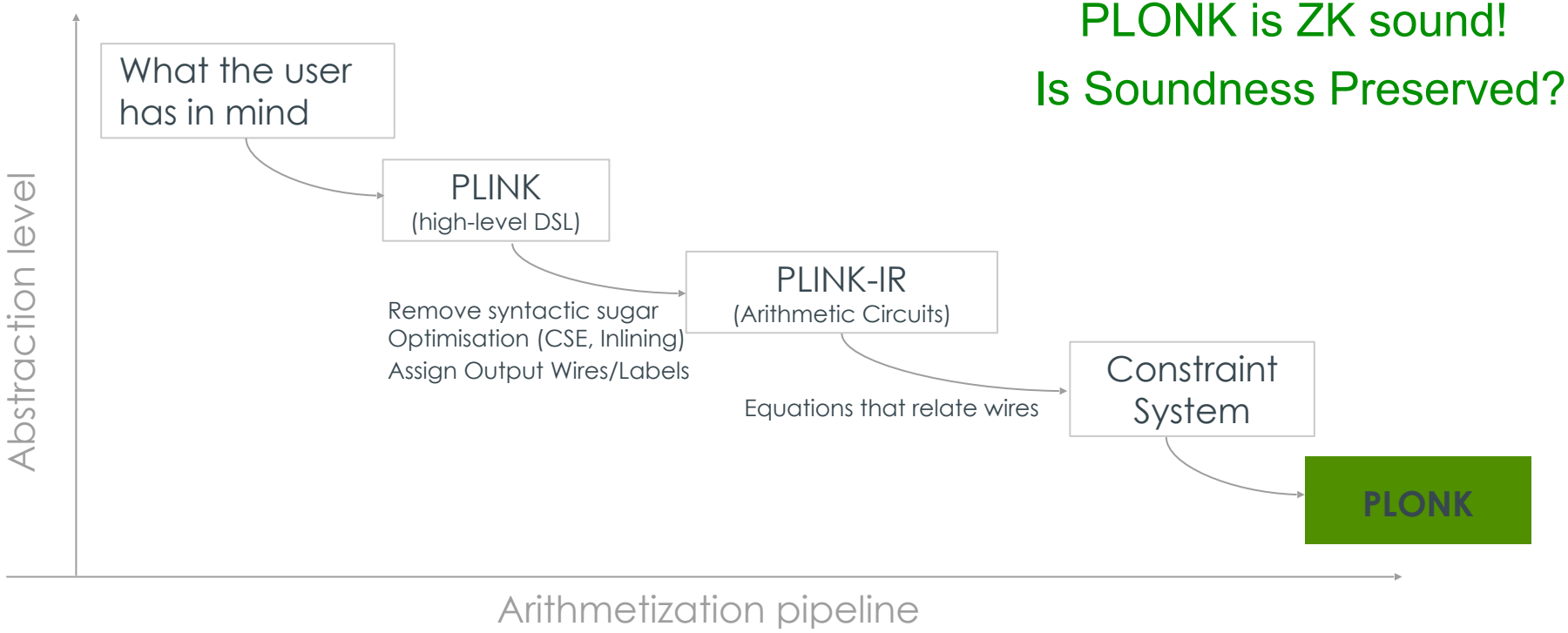
Declare variables

Give hints to the witness generator

Add constraints

Security Guarantees

Soundness of PLINK



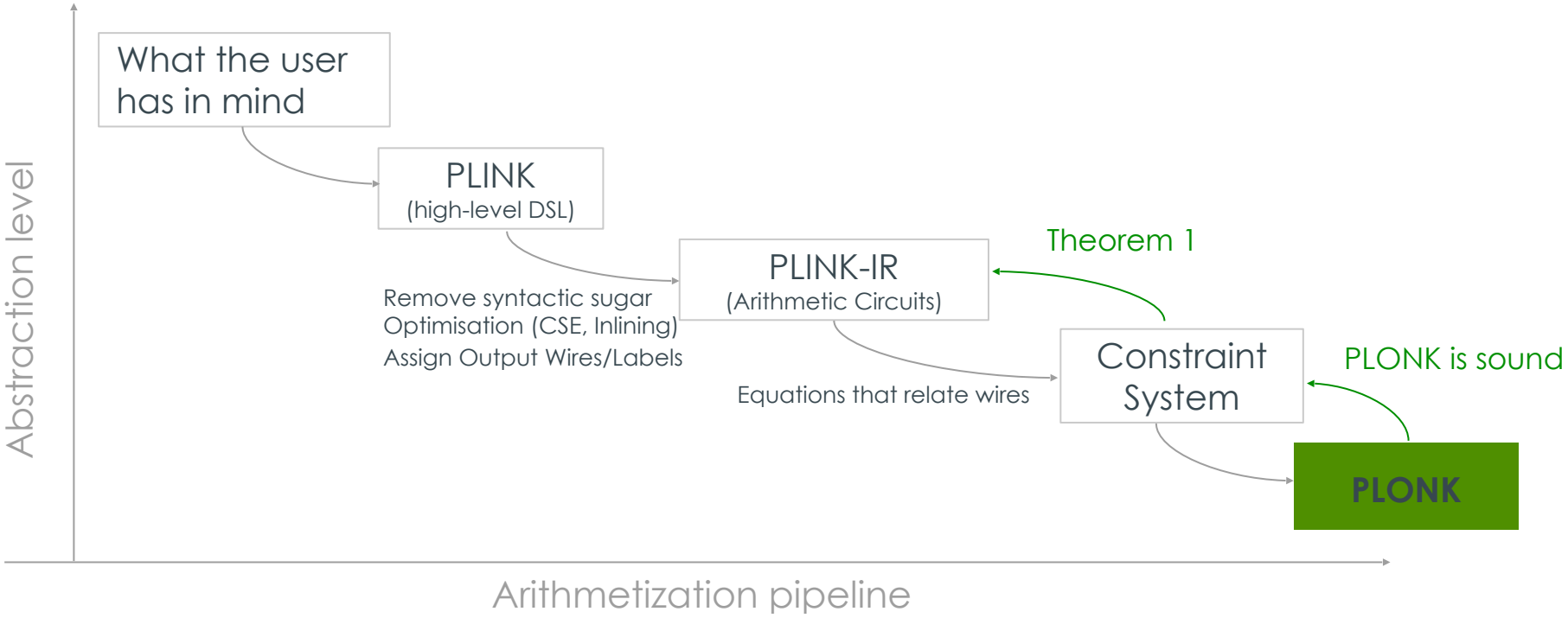
Semantics Preservation

Theorem 1: For each program $P_{IR} \in$ *PLINK-IR* and valuation σ of its variables, we have

$$\sigma \text{ satisfies } P_{IR} \iff \sigma \text{ satisfies } C(P_{IR}),$$

where C : *PLINK-IR* $\rightarrow$ *CS* is the compilation function and *CS* is the PLONK constraints.

Soundness Preservation



Semantics Preservation

Theorem 1: For each program $P_{IR} \in$ *PLINK-IR* and valuation σ of its variables, we have

$$\sigma \textit{ satisfies } P_{IR} \iff \sigma \textit{ satisfies } C(P_{IR}),$$

where C : *PLINK-IR* $\rightarrow$ *CS* is the compilation function and *CS* is the PLONK constraints.

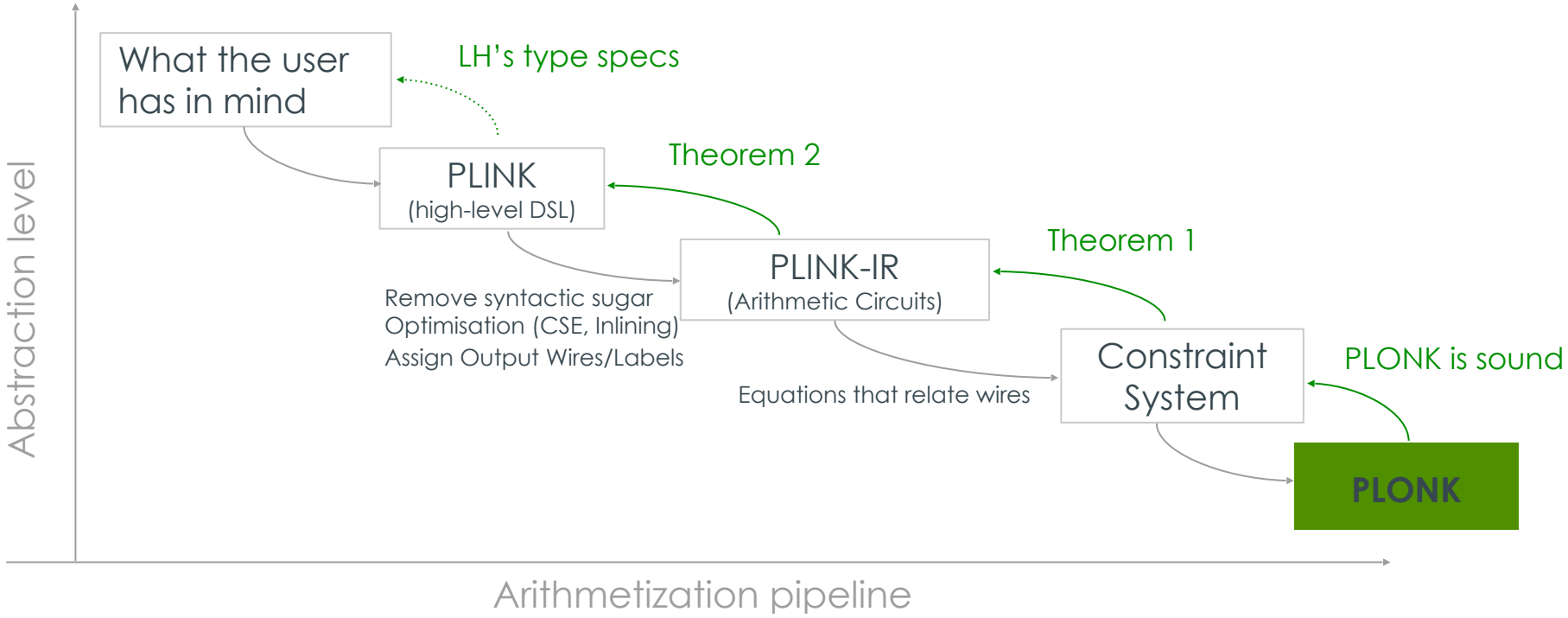
Theorem 2: For each program $P \in$ *PLINK* and valuation ρ of its variables, we have

$$\rho \textit{ satisfies } P \iff \sigma \textit{ satisfies } IR(P),$$

where IR : *PLINK* $\rightarrow$ *PLINK-IR* is the compilation function *and* σ = witnessGen(ρ,IR(P)).

Proven about the Haskell **implementation** *itself* using Liquid Haskell (in~500 LoC)

Soundness Preservation



Functional Guarantees

Since PLINK programs are Liquid Haskell programs,
refinement types specify and automatically check PLINK types.

Length preservation & exhaustive cases!

Benchmark: SHA-256

SHA-256 Implementation

SHA-256 is standard and widely used in cryptographic protocols hash function.

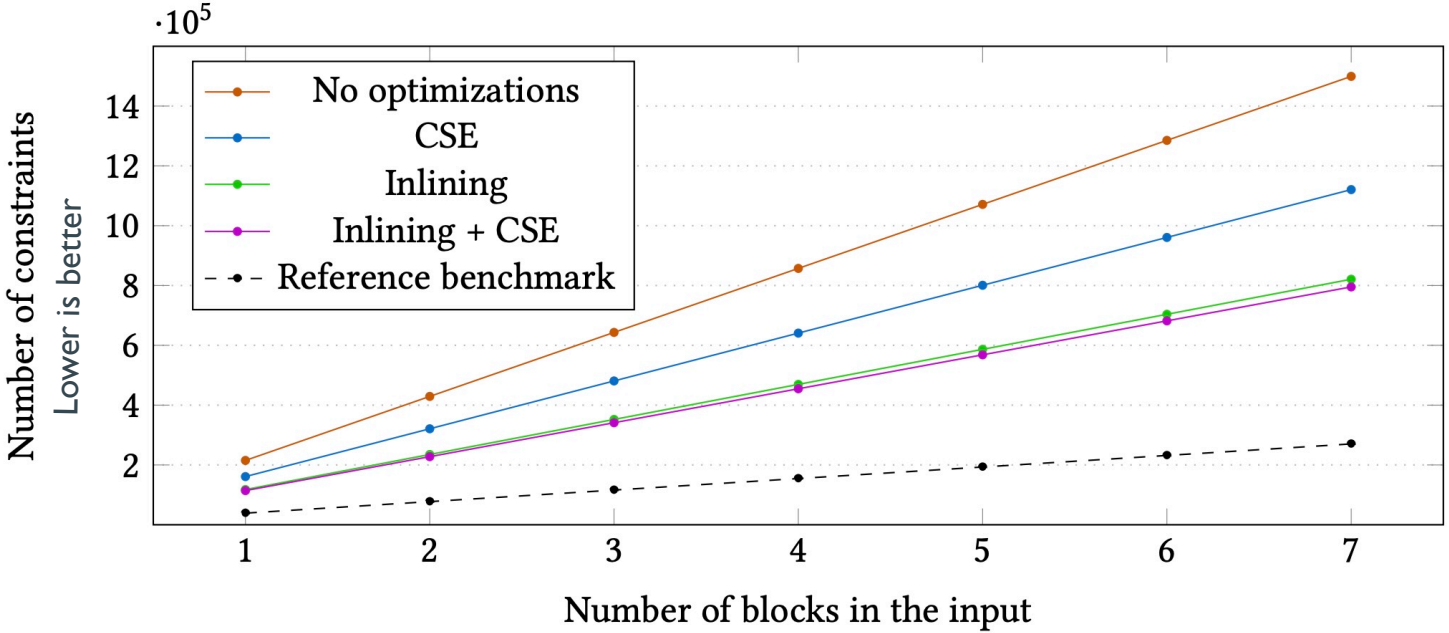
Built using library functions (e.g. modular addition, bitwise xor, vector rotate…).

- Some of these (e.g. bitwise xor, vector rotate…) can be implemented just as in Haskell.
- ~220 lines of Haskell + Liquid Haskell annotations

Standard functional implementation that combines these components.

- ~210 lines of Haskell + Liquid Haskell annotations

SHA-256 Implementation



block: 512-bit input

Reference benchmark is extrapolated from zk_bench

PLINK

Embedded **DSL** in Liquid Haskell for generation of PLONK CS.

Declarative, with support for **types** (bool, field, vectors).

Semantic preservation is mechanised using Liquid Haskell (~500 LoC).

Functional correctness imposes specifications on the DSL programs.

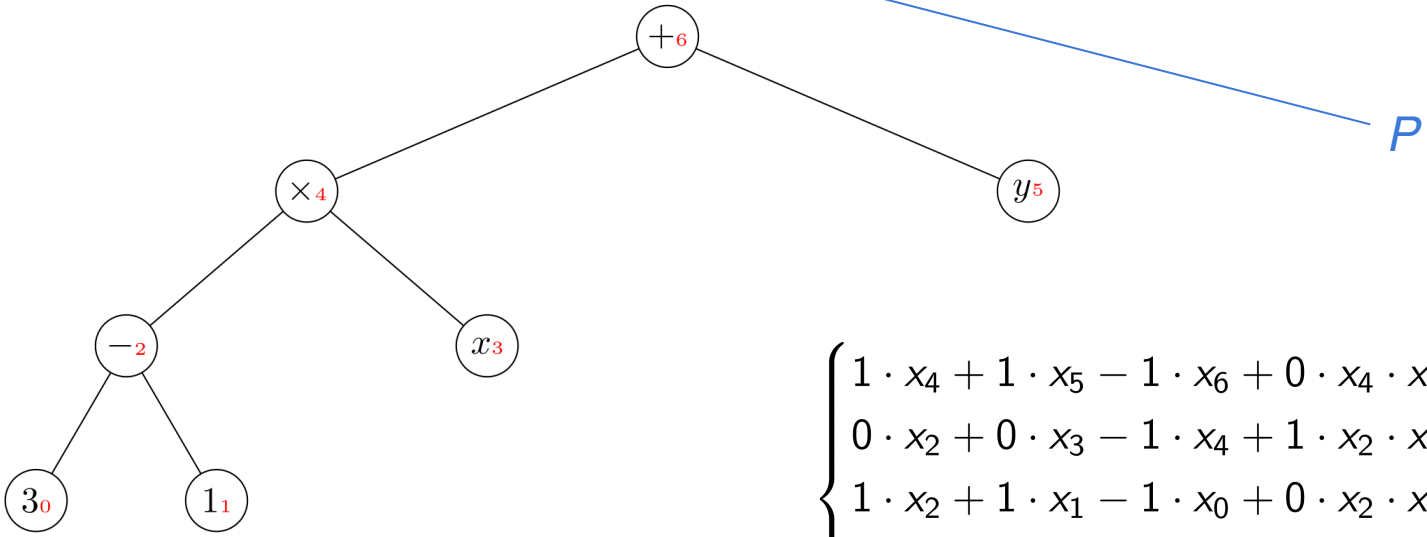
We tested PLINK by implementing SHA-256.

Thank you! Questions?

Optimizations

Optimizations

((CONST 3 `SUB` CONST 1) `MUL` (VAR "x" TF)) `ADD` (VAR "y" TF)

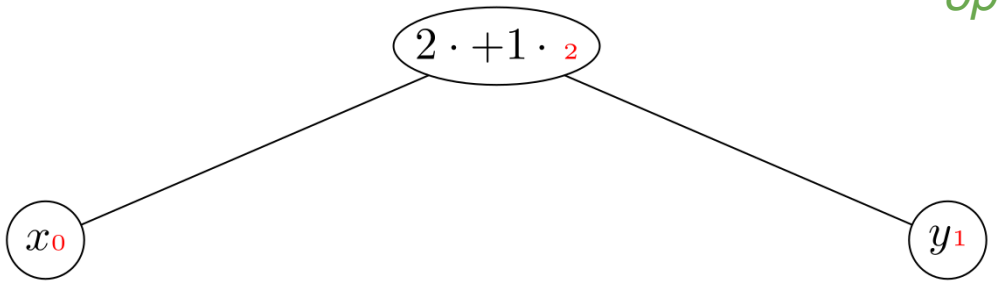


$$\begin{cases} 1 \cdot x_4 + 1 \cdot x_5 - 1 \cdot x_6 + 0 \cdot x_4 \cdot x_5 + 0 = 0 \\ 0 \cdot x_2 + 0 \cdot x_3 - 1 \cdot x_4 + 1 \cdot x_2 \cdot x_3 + 0 = 0 \\ 1 \cdot x_2 + 1 \cdot x_1 - 1 \cdot x_0 + 0 \cdot x_2 \cdot x_1 + 0 = 0 \\ 0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_0 + 0 \cdot x_0 \cdot x_0 + 3 = 0 \\ 0 \cdot x_0 + 0 \cdot x_0 - 1 \cdot x_1 + 0 \cdot x_0 \cdot x_0 + 1 = 0 \end{cases}$$

Optimizations

LINCOMB 2 (VAR “x” TF) 1 (VAR “y” TF)

opt(P)



$$\begin{cases} 2 \cdot x_0 + 1 \cdot x_1 - 1 \cdot x_2 + 0 \cdot x_0 \cdot x_1 + 0 = 0 \end{cases}$$

Optimizations are *proven* correct

- Optimizations happen at the DSL level.
  - They *change* what the user writes.
- **Theorem**: For each program $P \in PLINK$ and valuation σ of its variables

$$P \cong_\sigma opt(P)$$

where $opt : PLINK \to PLINK$ is the optimization function. Concretely, if P has value v under σ, then $opt(P)$ also has the same value.

  - Completely modular (e.g. in case we want to add new optimizations).