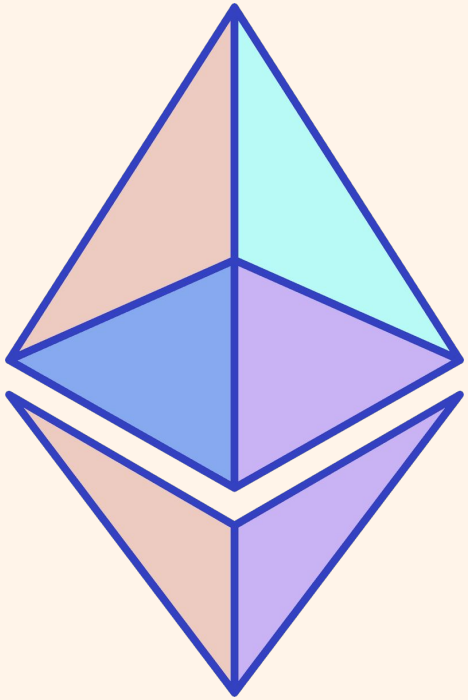


Formally verifying zk(E)VMs with the Ethereum Foundation

Alexander Hicks
Ethereum Foundation Research

ZKProofs 7

The promise of zkVMs

General Purpose zkVMs

Execute any program* and generate a cryptographic proof of its execution

➔ No need to write a circuit for each program

RISC-V

Open source modular ISA with stable compilers

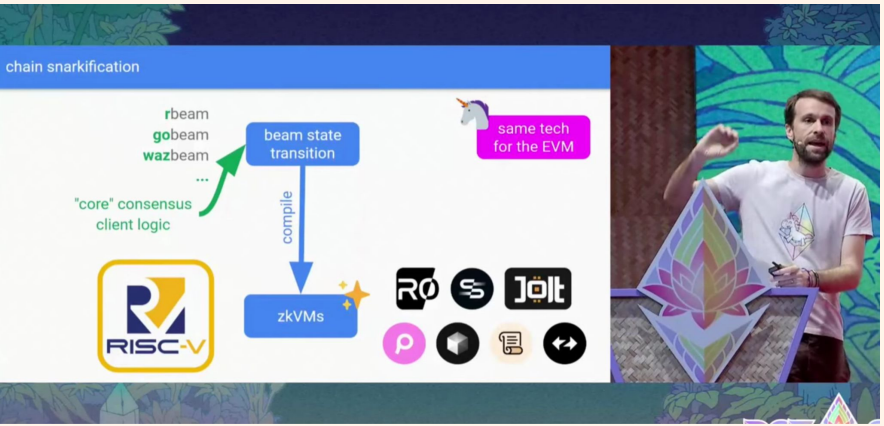
* Some caveats

zkVM	ISA	Continuations	Parallelizable Proving	Precompiles	GPU	Frontend
cairo	Cairo	❌	❌	✅		Cairo
ceno	RISC-V	❌	❌	✅		Rust
eigen zkvm	RISC-V	✅	✅	✅	✅	Circom, PIL
jolt	RISC-V	❌	❌	❌		Rust
miden	MASM(Miden Assembly)	❌	❌	✅	✅	Rust, Wasm
mozak vm	RISC-V	❌	❌	❌		Rust
nexus	RISC-V	✅	✅	✅		Rust
o1vm	MIPS	❌	❌	❌		Go
olavm	Ola Assembly	❌	❌	✅		Ola Assembly
openvm	RISC-V	✅	✅	✅		Rust
pico	RISC-V	✅	✅	✅		Rust
powdrVM	RISC-V	✅	✅	✅		ASM assembly
risc0	RISC-V	✅	✅	✅	✅	Rust
sp1	RISC-V	✅	✅	✅	✅	Rust
sphinx	RISC-V	✅	✅	✅		Rust, Lurk
triton vm	Triton Assembly	❌	❌	❌		Triton Assembly
valida	Valida	❌	❌	❌		Rust, C
zisk	RISC-V	✅	✅	✅		PIL
zkm	MIPS	✅	✅	✅		Rust, Go
zkWasm	Wasm	✅	✅	✅		C, C++, rust, etc (wasm compilable)

github.com/rkdud007/awesome-zkvm

zkVMs in Ethereum

Beam Chain at Devcon



Native rollups

Native rollups—superpowers from L1 execution

Layer 2 stateless

JustinDrake 1 Jan 20

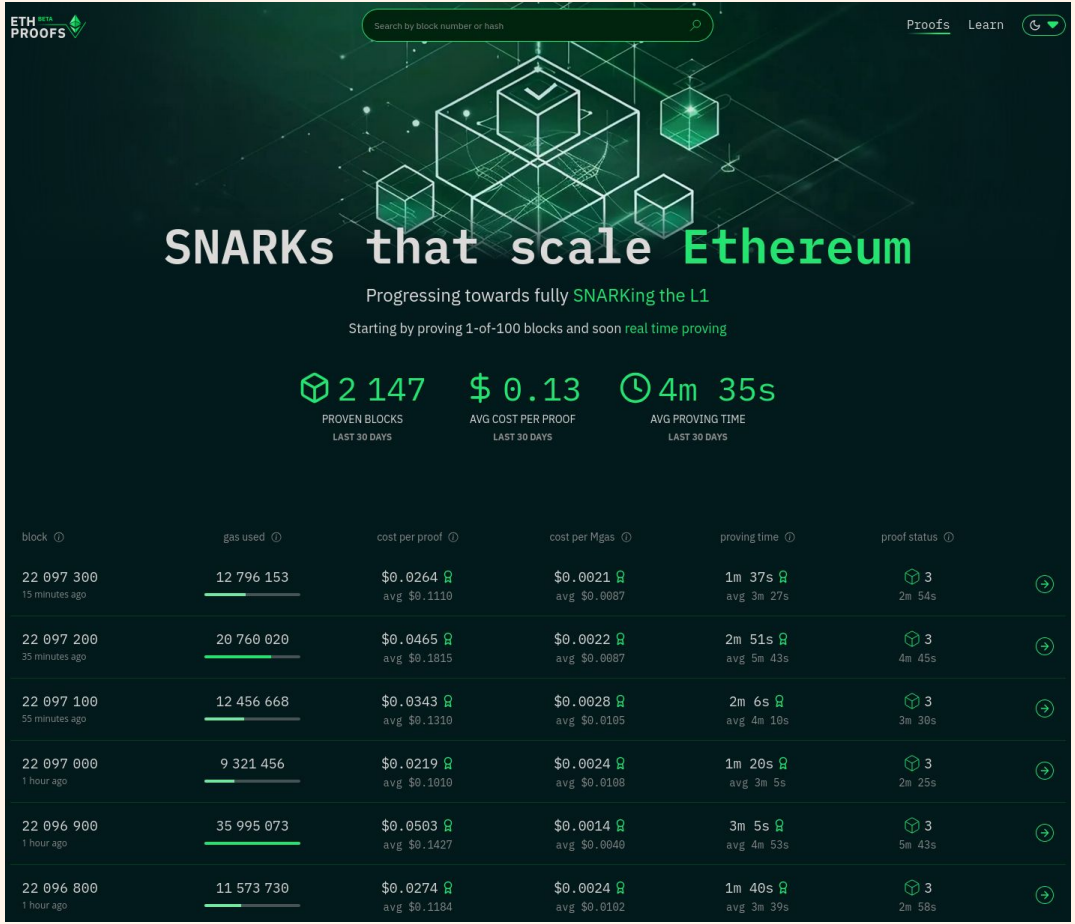
TLDR: This post suggests a pathway for EVM-equivalent rollups to rid themselves of security councils and other attack vectors, unlocking full Ethereum L1 security.

RISC-V native execution

Given today's de facto convergence towards RISC-V zkVMs 54 there may be an opportunity to expose RISC-V state transitions natively to the EVM (similarly to WASM in the context of Arbitrum Stylus 11) and remain SNARK-friendly.

Performance is only getting better

ethproofs.org



Formally verified RISC-V zk(E)VMs?

zkVMs will be security-critical

The EF is investing millions into making sure we can formally verify them over the next two years

➔ **Grants & bounties!**

verified-zkevm.org

ELI5 Formal verification of programs

A proof
which can be verified by your computer
that your code does exactly what its specification says it does

No more, no less

What are we verifying?

zkVM track
RISC-V circuits & arithmetization are such that you can only generate a proof if and only if the computation is valid

EVM track
An EVM implementation compiled to RISC-V is equivalent to a formal EVM specification

Cryptography track
Implementations of proof systems are equivalent to executable specifications tied to formalized security proofs

zkVM track

(1) Reference Sail RISC-V specification

(2) Circuit Verification

Extract key circuits (plonky3, Jolt) to proof assistants
Write circuits directly in proof assistants and proving them correct
Develop an MLIR dialect for circuits that will provide a unified layer to work from

(3) Figure out common abstractions

Can we develop general purpose proof infrastructure (representations, tactics, …)?

EVM track

Many existing formal specifications of the EVM
In K, Lean, ACL2, HOL4

Ongoing

Verify revm compiled to RISC-V against KEVM, equivalence of KEVM with Lean specification
Rocq implementation, aiming to be standalone & performant, with a certified compilation pipeline

More to come!
Several other formal specifications and teams that can verify EVM implementations

Cryptography track

Now: zkLib
Lean Library of executable implementations of proof systems directly tied to security proofs

Next: Verifying implementations of proof systems

Challenges

Formal Verification is hard
And zkVMs themselves are complex

No single zkVM
Software diversity means more work

Maintenance
zkVMs and the EVM are always changing

Lack of tooling & libraries
We need to make formal verification more practical and accessible

Measuring progress

Security stage 1: Correct protocols

Stage 1a

A formally verified proof of soundness for the PIOP.

Stage 1b

A formally verified proof that the PCS is binding under some cryptographic assumption or an idealized model.

Stage 1c

If using Fiat-Shamir, a formally verified proof that the succinct argument obtained by combining the PIOP and the PCS is secure in the random oracle model (augmented with other cryptographic assumptions as appropriate).

Stage 1d

A formally verified proof that the constraint system to which the PIOP is applied is equivalent to the VM's semantics.

Stage 1e

A comprehensive "gluing together" of all these pieces into a single, formally verified proof of a secure SNARK for running any program specified by the VM's bytecode. If the protocol intends to achieve zero knowledge, this property must also be formally verified, ensuring that no sensitive information about the witness is revealed.

Recursion Caveat: If the zkVM uses recursion, *every* PIOP, commitment scheme, and constraint system involved anywhere in that recursion must be verified to consider a sub-stage complete.

Security stage 2: Correct verifier implementation

A formally verified proof that an actual implementation of the zkVM verifier (in Rust, Solidity, etc.) matches the protocol verified in Stage 1. Achieving this ensures the *implemented* protocol is sound (rather than merely the design on paper, or an inefficient specification written in, say, Lean).

The reason Stage 2 focuses on the verifier implementation alone (and not the prover) is two fold. First, getting the verifier right is already sufficient for *soundness* (i.e., ensuring that the verifier cannot be convinced that any false statement is in fact true). Second, zkVM verifier implementations are over an order of magnitude simpler than prover implementations.

Security stage 3: Correct prover implementation

A formally verified proof that an actual implementation of the zkVM prover correctly generates proofs for the proof system verified in Stages 1 and 2. This ensures completeness — that is, any system using the zkVM will not get "stuck" with a statement that cannot be proven. If the prover intends to achieve zero knowledge, this property must be formally verified.

Key caveats: Fiat-Shamir security and verified bytecode

https://a16zcrypto.com/posts/article/secure-efficient-zkvms-progress/

How do we define success?

Success is zkVMs that are better: more secure, more performant, more useable

Formal verification shouldn't slow down development
It should allow us to push further instead

An unexpected discovery: Automated reasoning often makes systems more efficient and easier to maintain

by Byron Cook | on 17 OCT 2024 | in Foundational (100), Security, Identity, & Compliance, Thought Leadership | Permalink | Comments | Share

The reason is that the bug fixes we make during the process of formal verification often positively impact the code's runtime. Automated reasoning also gives our builders confidence to explore additional optimizations that improve system performance even further. We've found that formally verified code is easier to update, modify, and operate, leading to fewer late-night log analysis and debugging sessions. In this post, I'll share three examples that came up

If not from me, take it from AWS

zkEVM Formal Verification Project

A project by the Ethereum Foundation to accelerate the application of formal verification methods to zkEVMs

Interested? Get in touch! verified-zkevm@ethereum.org

Project website - verified-zkevm.org

Github - github.com/Verified-zkEVM

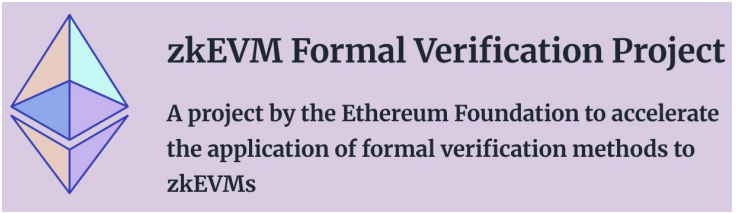
zkLib - Verified Proof Systems in Lean

Quang Dao (CMU)

Joint work with Devon Tuma (UMinnesota) & Gregor Mitscha-Baude (zkSecurity)

ZKProof 7, Sofia

Formal Verification for SNARKs



This work!

- Interactive (Oracle) Proofs-based SNARKs
- Linear-PCP-based SNARKs (e.g. Groth16)
- Verified verifier's implementation for Groth16 / Plonk
- DSLs for formally verified circuits
- (Semi-)automated solvers for under constrained bugs

Backend — Implementation — Frontend

zkLib: Formally Verifying SNARK Backends in Lean

Goals:

- Mechanize the security proofs of current SNARKs (IOP-based)
- Extracting verified implementations from verified SNARK specifications

Desiderata:

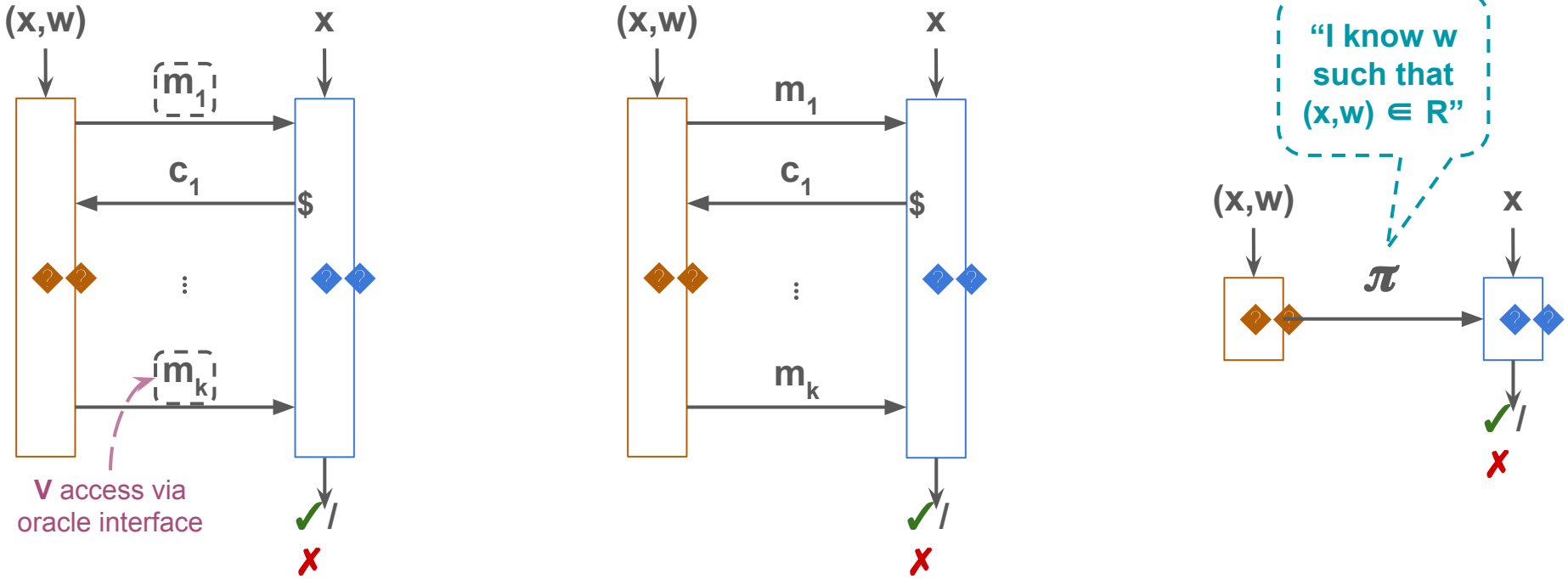
- SNARKs should be specified & proven secure in a ***modular*** and ***compositional*** manner
- Develop a ***core language*** & ***program logic*** for Interactive Oracle Reductions

  $\Rightarrow$ aimed to clarify existing constructions & streamline security proofs

Talk Outline

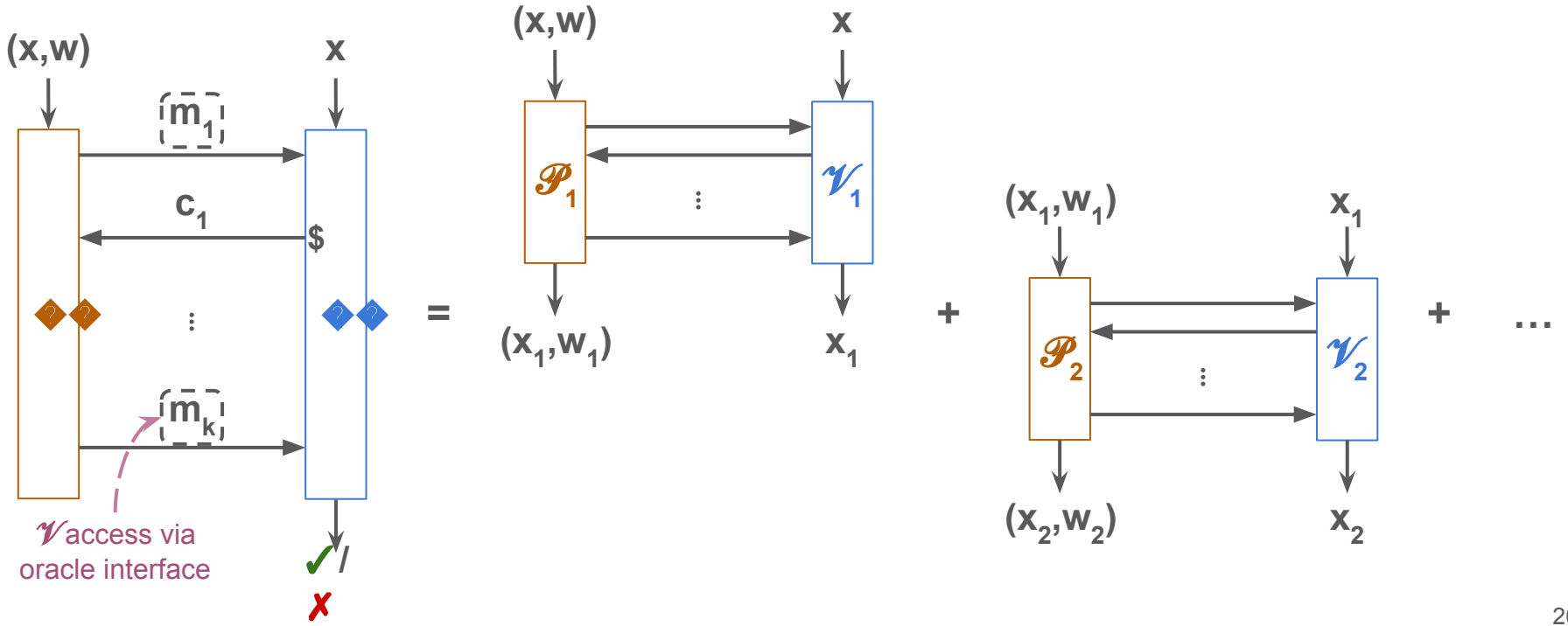
1. Anatomy of IOP-based SNARKs
2. (WIP) Program logic for IORs
3. zkLib’s current development, and next steps

What are IOP-based SNARKs?

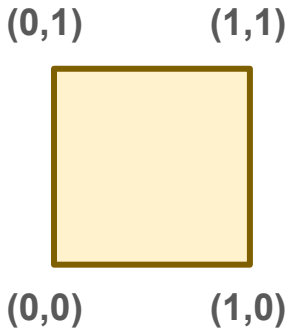


How are IOPs constructed?

Via composing a series of **interactive oracle reductions (IORs)**



IOR Example: The Sum-Check Protocol



Relation R_{in} :

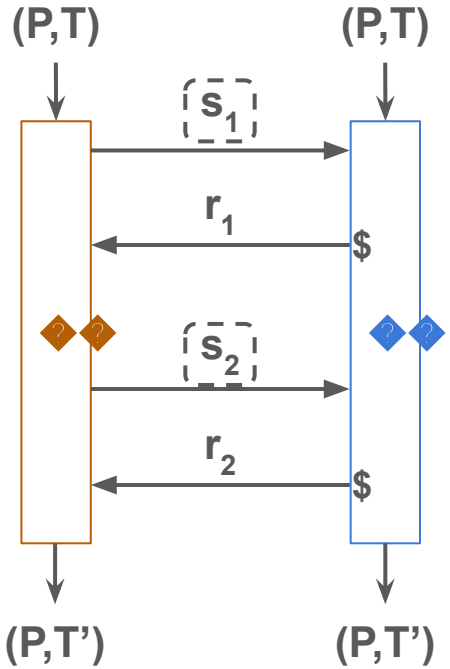
$P(0,0) + P(1,0) + P(0,1) + P(1,1) = T$

Relation R_{out} :

$P(r_1,r_2) = s_2(r_2)$

$s_1(X) = P(X,0) + P(X,1)$

$s_2(X) = P(r_1,X)$



$s_1(0) + s_1(1) \stackrel{?}{=} T$

$s_2(0) + s_2(1) \stackrel{?}{=} s_1(r_1)$

$T' := s_2(r_2)$

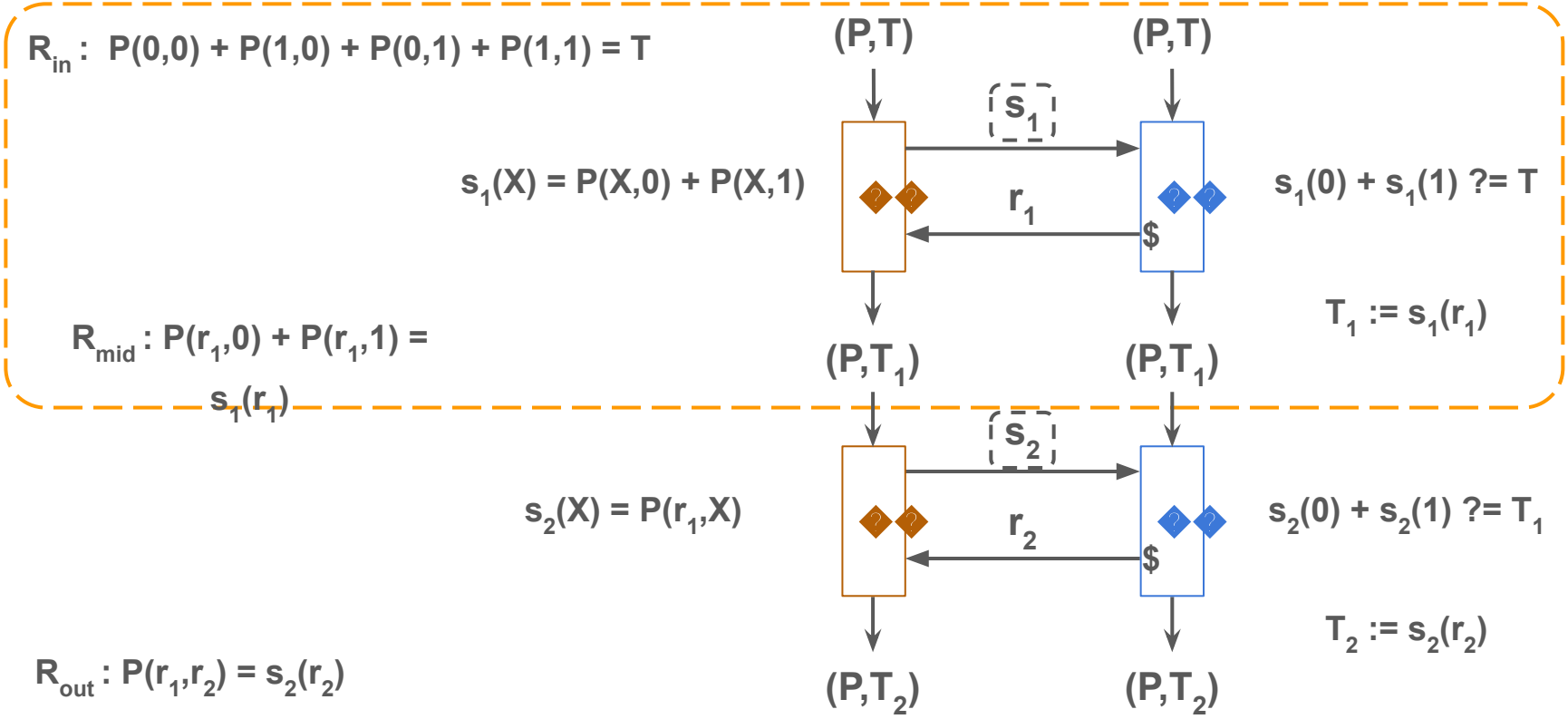
Talk Outline

1. Anatomy of IOP-based SNARKs

2. (WIP) Program logic for IORs

3. zkLib’s current development, and next steps

The Sum-Check Protocol, Revisited



The Sum-Check Protocol, Revisited

R_{in} : $P(0,0) + P(1,0) + P(0,1) + P(1,1) = T$

Pr[fail] = 0

$s_1(X) = P(X,0) + P(X,1)$

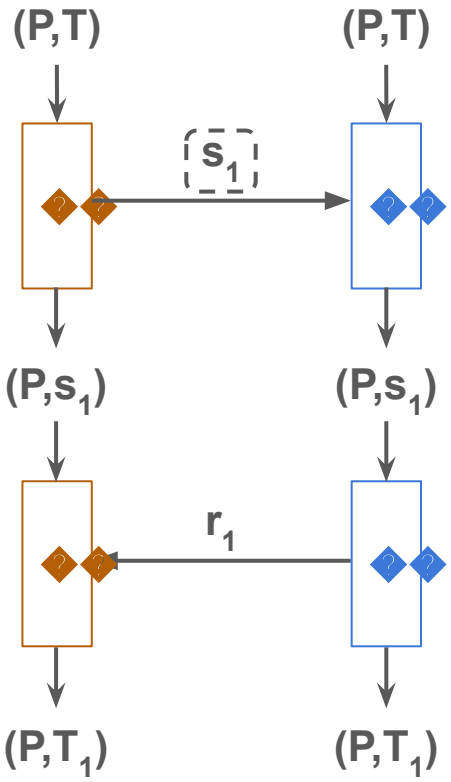
R_{mid}' : $P(X,0) + P(X,1) = s_1(X)$

Pr[fail] ≤ d / |F|

Round-by-round guarantees come for free!

R_{mid} : $P(r_1,0) + P(r_1,1) = s_1(r_1)$

Pr[fail] := $V(\cdot) = \checkmark \wedge R_{out}(\cdot,\cdot) = \times$



$s_1(0) + s_1(1) \stackrel{?}{=} T$

$T_1 := s_1(r_1)$

The Sum-Check Protocol, Revisited

R_{in} : $P_1(0) + P_1(1) = T$

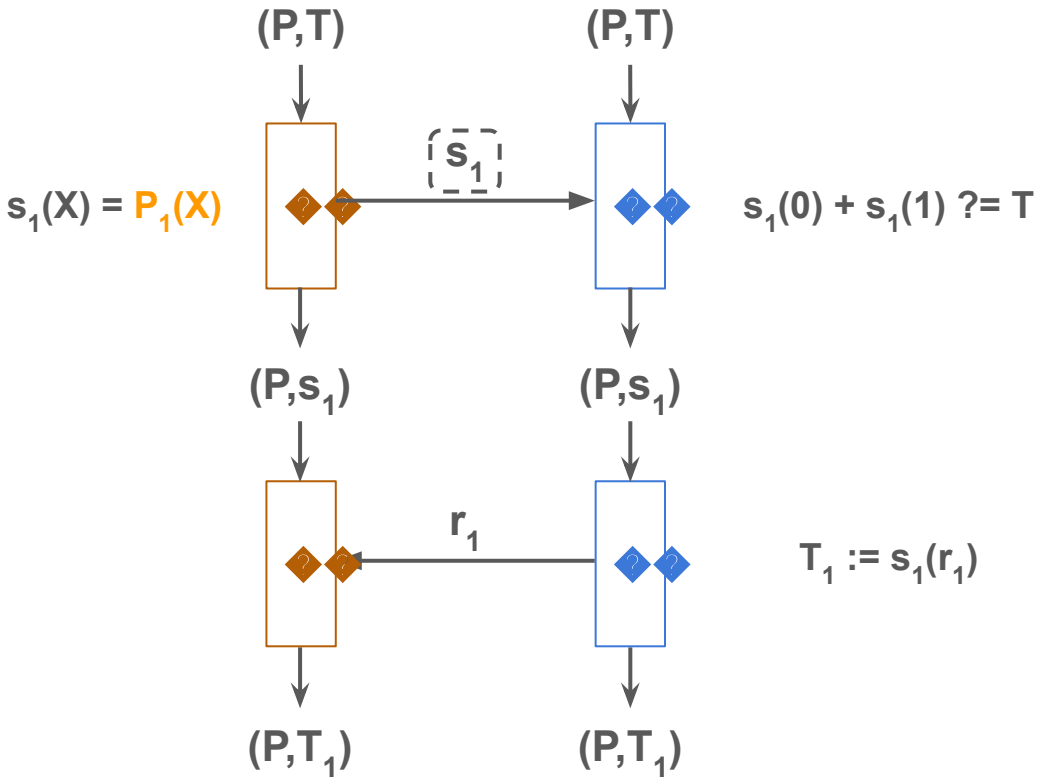
$P_1(X) := P(X,0) + P(X,1)$

is a **virtual** polynomial

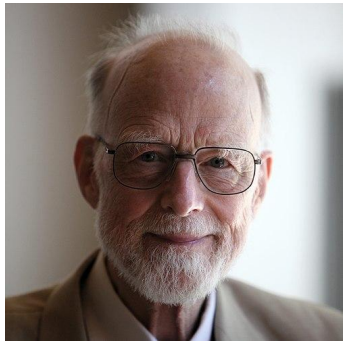
R_{mid}' : $P_1(X) = s_1(X)$

How do we formalize virtual protocols?

R_{mid} : $P_1(r_1) = s_1(r_1)$



Connection to Hoare-style Program Verification



Tony Hoare

Robert Floyd

Hoare Triples: {P} C {Q}

"Program C, when executed with precondition P, attains postcondition Q"

Hoare Rules:

$$\textsc{Consequence}\quad \frac{F \vdash F' \qquad \vdash \{F'\}\ S\ \{G'\} \qquad G' \vdash G}{\vdash \{F\}\ S\ \{G\}}$$

$$\textsc{Loop}\quad \frac{\vdash \{F \wedge c\}\ S\ \{F\}}{\vdash \{F\}\ \textbf{while}\ c\ \textbf{do}\ S\ \{F \wedge \neg c\}}$$

$$\textsc{Sequencing}\quad \frac{\vdash \{F\}\ S_1\ \{F'\} \qquad \vdash \{F'\}\ S_2\ \{F''\}}{\vdash \{F\}\ S_1; S_2\ \{F''\}}$$

$$\textsc{Conditional}\quad \frac{\vdash \{F \wedge c\}\ S\ \{F'\} \qquad \vdash \{F \wedge \neg c\}\ S'\ \{F'\}}{\vdash \{F\}\ \textbf{if}\ c\ \textbf{then}\ S\ \textbf{else}\ S'\ \{F'\}}$$

$$\textsc{Assignment}\quad \frac{}{\vdash \{F[v \mapsto t]\}\ v := t\ \{F\}}$$

IOR Triples: {R_1} <P,V> {R_2}

"Reduction <P,V>, when executed on inputs satisfying R_1, results in outputs satisfying R_2"

Can we build a program logic for IORs?

Talk Outline

1. Anatomy of IOP-based SNARKs

2. (WIP) Program logic for IORs

3. Current state of zkLib and next steps

Immediate goals for zkLib

Currently, we have:

1. Definitions of Interactive Oracle Reductions & computable polynomial data types
2. Specification & proof of security for a single round of sum-check

In a few months, we plan to formalize:

1. Composition of IORs, and proofs that they preserve security
2. Executable spec & security proof for the Spartan Polynomial IOP
3. Blueprints for FRI, STIR, WHIR

Longer-term goals for zkLib

1. Compilation steps:

   IOPs =[Commitments]=> IPs =[Fiat-Shamir]=> SNARKs

2. Mechanize rewinding knowledge soundness & zero-knowledge

3. Improve verifier's performance (compiled from Lean), *and/or*

   Establish functional equivalence with extracted impl' from Rust

4. Tutorials & onboarding documentation

Summary

- Formally verifying SNARKs is important - and should be done right now!
- We are building **zkLib**, a Lean framework to mechanize your favorite (IOP-based) SNARKs
- **Key ideas:**
  - Reductions as the main building block
  - Leverage existing program verification ideas

Verified-zkEVM / **ZKLib**

Thank you!