# Towards a verified Jolt zkVM

**James Parker**, Ben Hamlin, Benoit Razet, Ben Selfridge, Brett Decker

With

LindyLabs: Jakob von Raumer, Ryan Lahfa

Cambridge: Tobias Grosser, Leo Stefanesco, Leon Frenot, Arthur Adjedj, Alasdair Armstrong, Peter Sewell

Edinburgh: Brian Campbell

# Motivation

- Zero-knowledge proofs (ZKPs) allow smart contract updates and transactions to be validated quickly
    - Improves throughput and reduces gas fees
- Problem: ZK statements are complex and difficult to check
    - Bugs in frontend statements would result in security vulnerabilities, threatening assets entrusted to Ethereum
- Solution: Formally verify that ZK statements enforce RISC-V’s semantics

# Jolt

- Focusing on verifying Jolt's ZK statements[1]
    - R1CS, lookup tables, composition of lookup tables
- One of the most promising zkVMs
- RISC-V IM 32-bit machine
- Developed by Michael Zhu, Sam Ragsdale, Arasu Arun, Srinath Setty, and Justin Thaler

[1] Arun, Setty, Thaler. "Jolt: SNARKs for Virtual Machines via Lookups." EUROCRYPT 2024

# Long-term goal: Soundness

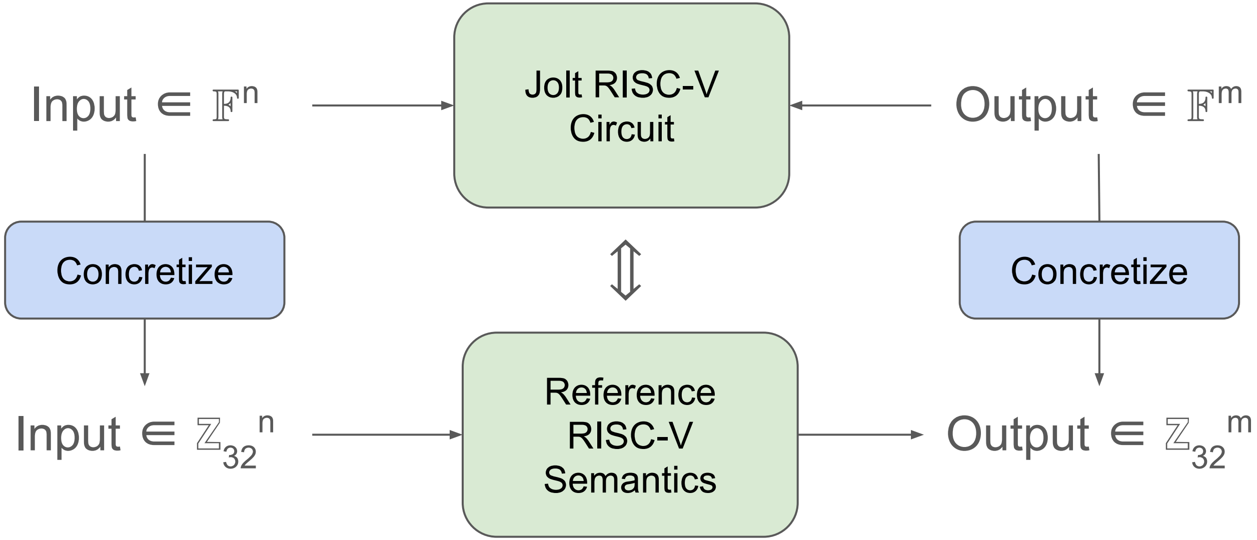
Theorem 1. If the witness is not well formed, the circuit should not be satisfied:

∀ w: Witness . (concretize(w) == None) => (jolt_circuit(w) == false)

Theorem 2. If the witness is well formed, the input and output match the RISC-V's semantics if and only if the circuit is satisfied:

∀ w: Witness . (concretize(w) == Some(input, output)) =>
(output == riscv(input)) <=> (jolt_circuit(w) == true))

# Long-term goal: Soundness


Input ∈ $\mathbb{F}^n$
Jolt RISC-V Circuit
Output ∈ $\mathbb{F}^m$
Concretize
⇕
Concretize
Input ∈ $\mathbb{Z}_{32}{}^n$
Reference RISC-V Semantics
Output ∈ $\mathbb{Z}_{32}{}^m$

# Approach

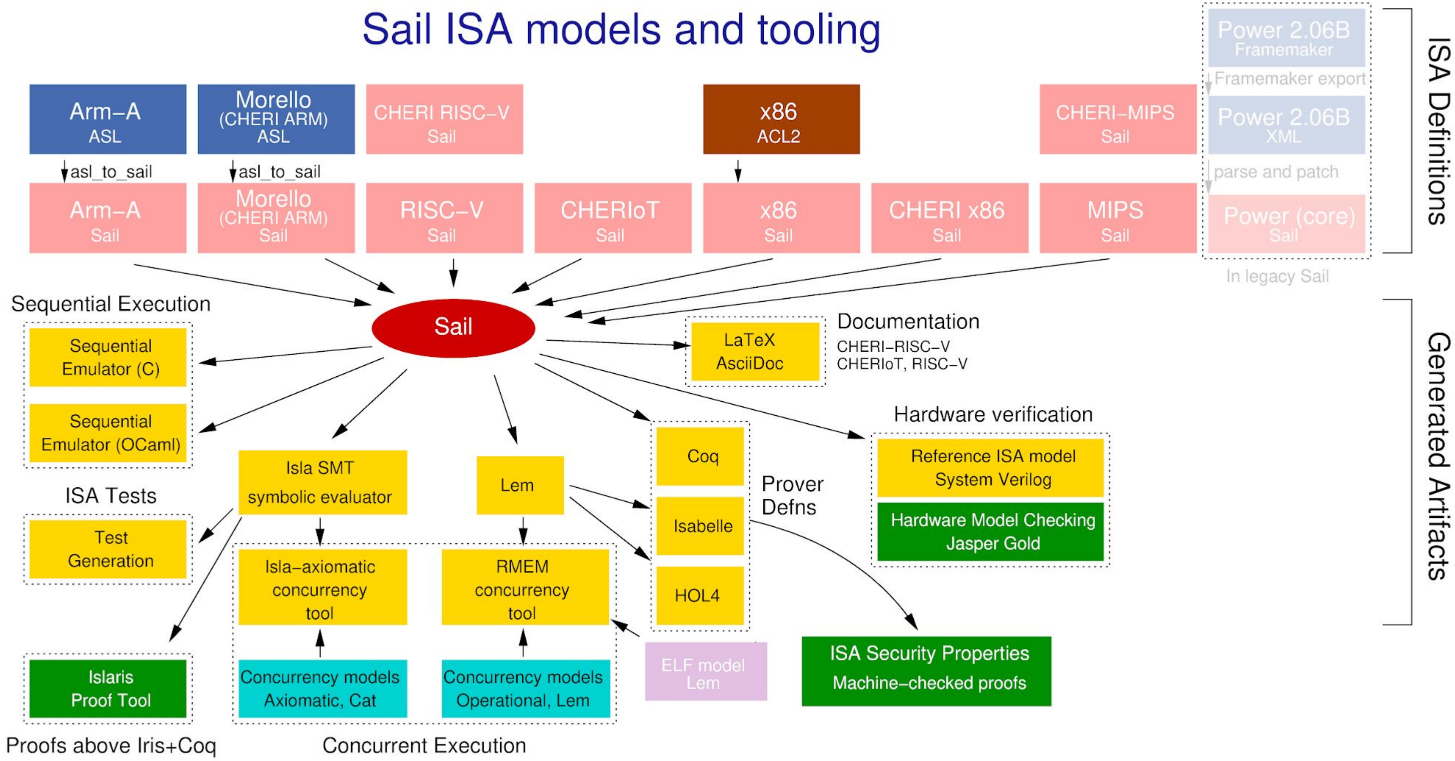
## Stage 1 (In progress)

1. Define reference RISC-V semantics with Sail
2. Implement DSL for specifying ZK statements in Lean (zkLean)
3. Tool that automatically extracts Jolt's arithmetization into zkLean

## Stage 2

1. Prove that Jolt's arithmetizations match the Sail RISC-V semantics
2. Integrate these proofs into Jolt's CI

# Background: Sail[1]



[1] Armstrong, et al. "ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS." POPL 2019

# Excerpt: Sail RISC-V model

```
mapping clause encdec = RTYPE (rs2, rs1, rd, RISCV_XOR)
 <-> 0b0000000 @ rs2 @ rs1 @ 0b100 @ rd @ 0b0110011
...
function clause execute (RTYPE (rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_XOR => rs1_val ^ rs2_val,
    ...
  };
  X(rd) = result;
}
```

# Sail to Lean

Building a Lean backend for Sail[1]

- Extracts Sail models to Lean
- For us: Reference RISC-V semantics in Lean
- For others: Reason about ISAs in Lean!

In collaboration with University of Cambridge and Lindy Labs

[1] https://github.com/rems-project/sail

# Sail to Lean Challenges

```
mapping clause encdec = LOAD(imm, rs1, rd, is_unsigned, ...)
 <-> imm @ encdec_reg(rs1) @ bool_bits(is_unsigned) @ ...
     if (size_bytes(size) < xlen_bytes)
```

No pattern guards in Lean!

→ Requires expanding out to nested pattern matches.

# Sail to Lean Challenges

```
def foo (data : BitVec (2 ^ 3 * 8)) : Int := ...
def bar (x : BitVec 32) : Int :=
  foo (@BitVec.append _ _ x x)
```

```
error: argument `x` has type `BitVec 32`but is expected to
have type `BitVec ((2 ^ 3).mul 7)`
```

Lean’s type system struggles to unify dependent integers

→ Requires adding manual coercions and wrapper functions around operations like `BitVec.append`

# Sail to Lean Status (RISC-V model)

**Lines of code:** 105,840

**Errors:** 2

- Due to mutating variables in while loops

# Sail to Lean Validation

- Run RISC-V tests on the extracted RISC-V semantics
  - Spike test suite[1]
- Differentially test the extracted RISC-V semantics
- Integrate testing into SAIL’s CI

[1] https://github.com/riscv-software-src/riscv-isa-sim

# zkLean

Lean4 library for encoding ZK statements

- Primary goal: DSL that is amenable to verification and automatic extraction from frontends
- Initially will support R1CS, lookup tables, and composition of lookup tables
- Extensible so that other features and constraint systems can be added in the future
- Agnostic to ZK backend

# zkLean example

```
def example1 [Field f] : ZKBuilder (ZKVar f) := do
  let x: ZKVar f <- witness
  constrain (x * (x - 1) === 0)
  return x
```

Monadic circuit builder: `ZKBuilder`

Polymorphic over field: `f`

# zkLean subtables

```
def EqSub [Field f] : Subtable f 16 := subtableFromMLE
  (fun (x : Vec 16 f) => (x[0]*x[8] + (1 - x[0])*(1 - x[8]))*...)
```

Jolt defines subtables of lookup arguments by defining their multilinear extensions (MLEs)

- Typically inputs of size $2^8$
- In zkLean, `subtableFromMLE` allows users to define a `Subtable` by providing the MLE as a lambda

# zkLean lookup tables

```
def Eq32 [Field f] : ComposedLookupTable f 16 4 :=
  composedLookupTable
    #[(EqSub, 0), (EqSub, 1), (EqSub, 2), (EqSub, 3)]
    (fun x => x[0]*x[1]*x[2]*x[3])
```

Jolt defines larger lookup tables by breaking up the input into chucks, applying the smaller subtables to each chunk, and composing the results

- In zkLean, `composedLookupTable` allows users to define a `ComposedLookupTable` by combining the subtables as a lambda

# Jolt extraction

- Jolt’s circuits are implemented in Rust
- Implementing a tool to automatically extract Jolt’s circuits into zkLean
- Rust binary that links to Jolt as a library, extracts AST representation of circuits, and outputs Lean files in zkLean

# Jolt extraction - Lookup

```
impl<F: JoltField> LassoSubtable<F> for EqSubtable<F> {
  fn evaluate_mle(&self, point: &[F]) -> F {
    let b = point.len() / 2;
    let (x, y) = point.split_at(b);
    let mut result = F::one();
    for i in 0..b {
      result *= x[i] * y[i] + (F::one() - x[i]) * (F::one() - y[i]);}
    result
}}
```

⇩

```
def EqSub [Field f] : Subtable f 16 := subtableFromMLE
  (fun (x : Vec 16 f) => (x[0]*x[8] + (1 - x[0])*(1 - x[8]))*...)
```

# Jolt extraction - R1CS

```
def uniform_jolt_constraints [Field f]
  (step_inputs : JoltCPUState f) : ZKBuilder f PUnit := do
    constrainR1CS
      step_inputs.ADDInstructionFlag
      (1 - step_inputs.ADDInstructionFlag)
      0
    constrainR1CS
     step_inputs.OpFlags_ConcatLookupQueryChunks
     (16777216*step_inputs.ChunkX_0 + 65536*step_inputs.ChunkX_1 +
256*step_inputs.ChunkX_2 + step_inputs.ChunkX_3 - step_inputs.LeftLookupArg)
     0
    ...
```

# Performance optimization

```
def TruncateOverflowSubtable [Field f] : Subtable f 16 :=
  subtableFromMLE (fun x => 0)
```

In the process of extracting Jolt’s subtables, we identified an unnecessary subtable!

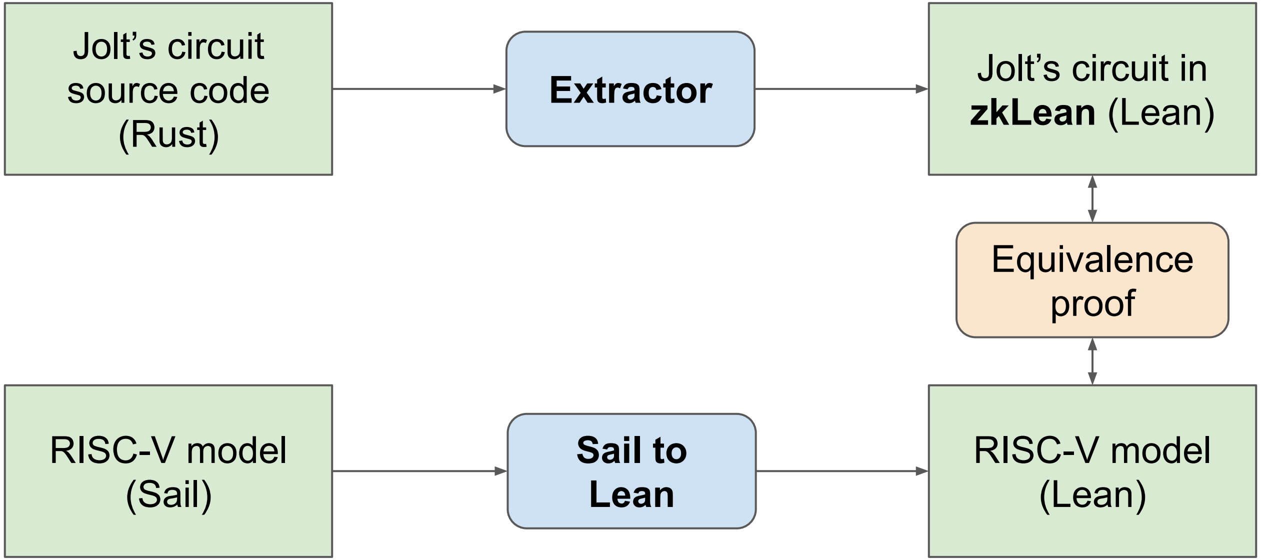
Removing the subtable improved prover runtime by ~2%

# Jolt extraction

Long-term goal: Integration with CI

- Automatic verification of compliance with Sail specification as the codebase evolves
- Automatically detect when theorems break due to updates to constraints, fields, etc
- Proofs will be automatically repaired?

# Summary



Jolt’s circuit source code (Rust)

**Extractor**

Jolt’s circuit in **zkLean** (Lean)

Equivalence proof

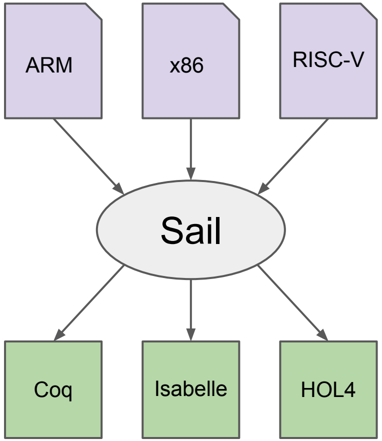
RISC-V model (Sail)

**Sail to Lean**

RISC-V model (Lean)

# Background: Sail

- Sail[1] is a dependently typed language for defining Instruction Set Architecture (ISA) semantics
- Provides models for ARM, x86, RISC-V
- Backends for Isabelle, HOL4, and Coq



[1] Armstrong, et al. "ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS." POPL 2019

# Automatic extraction from Jolt

- Jolt has abstract representation of R1CS for each RISC-V step
    - Straightforward for tool to extract R1CS constraints into zkLean
- Lookup tables currently need to be fully populated and manually compute the multi-linear extension
    - Switching to abstract representation of tables could improve extraction and verification
    - Alternatively, metaprogramming could be used

# zkLean example

```
riscv_step :: Field f => RISCVState f -> ZKBuilder (RISCVState f)
riscv_step st = do
    st’ <- witness

     instr <- Ram.read (pc st’)

     match instr {
          (Eq rd r1 r2) => {
               isEq <- lookup eq64 (r1 st) (r2 st)
               constrain (rd st’ == isEq)
          }
     }

    return st’
```

# zkLean example

```
eq64 :: Field f => LookupTable f
eq64 = compose_subtables $ \x y -> do
    e1 <- lookup eq8 (255 & x) (255 & y)
    ...
    return $ e1 * e2 * e3 * e4 * e5 * e6 * e7 * e8

eq8 :: LookupTable f
eq8 = subtable $ \x y -> do
    return $ if x == y then 1 else 0
```