Writing out the entire plan since the talk is a bit longer than usual.

1:15hrs

Outline:

- ZKVM
    - zkVM architecture
    - RISC Zero Terminology
    - zkVM features
    - Proof Stack - Full Perspective
        - Prove RISC-V execution with STARK -->
        - Prove recursion/aggregation with STARK -->
        - Run groth16 circuit to make it small
    - What’s possible with the zkVM - Demos
        - showcase different projects built on zkVM
            - Fibonacci walkthrough
            - Chess walkthrough
- Bonsai
    - Explain what it is
        - Where to use it
    - High level architecture
    - What's possible with Bonsai
        - showcase different projects built on Bonsai
        - Bonsai Pay
            - Architecture overview
            - code walkthrough
            - Demo

RISC
ZERO

RISC Zero
ZK Hack IV Online

Get Rusty with RISC Zero: Build your ZK Rust Application

Brian Retford, CEO
Dr Iryna Tsimashenka, DevRel
Hans Martin, Solutions Engineer

What will we cover?

Intro to RISC Zero

zkVM

- architecture
- terminology
- features
- proof system
- quick start

Break 5-10 min

Bonsai

- what is it? why?
- Bonsai Pay walk through

RISC ZERO

Introduction to R0

RISC Zero was started in 2021 and is focused on revolutionizing the internet by creating the infrastructure & tooling necessary for Web3 developers around the globe to build zero-knowledge software. We are bringing general-purpose computing to the zero-knowledge ecosystem – enabling users to trust programs run anywhere while allowing developers to use the tools they already know and love.

RISC Zero zkVM

is based on two main components:

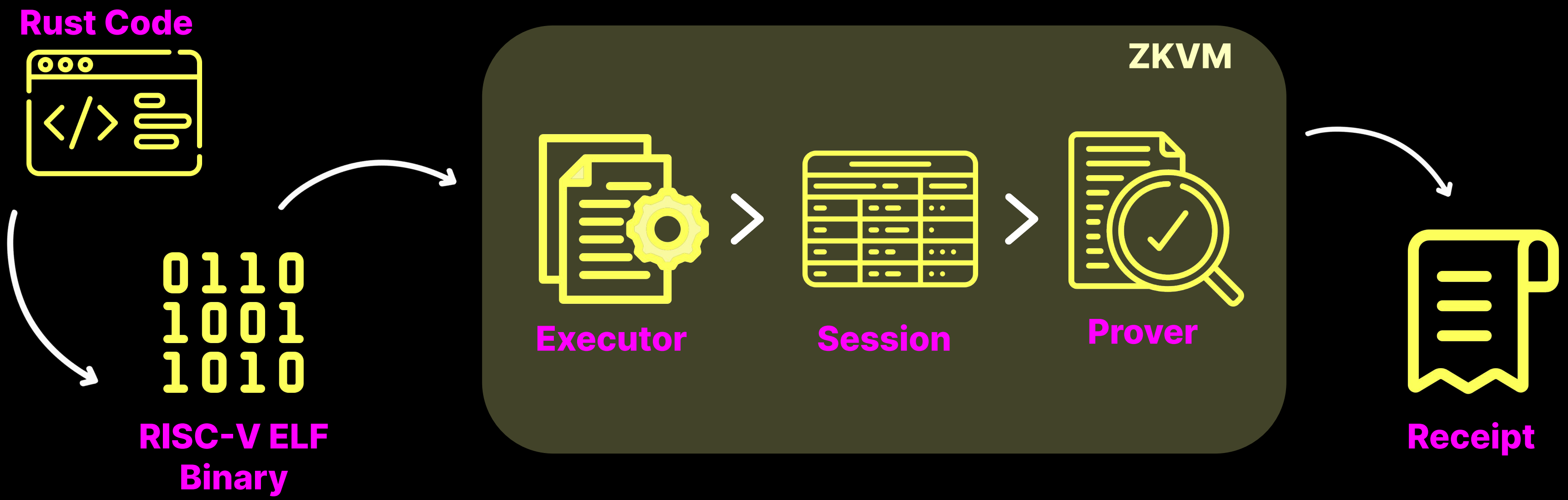
RISC-V ISA

open-source instruction set architecture based on the reduced instruction set computer that uses 32 int registers

zk-STARKs

zero knowledge, scalable, transparent argument of knowledge cryptography

zkVM Architecture



RISC ZERO

But, why is it important?
What can it do?

Developers can run Rust code through the zkVM and prove the execution was done correctly.

The zkVM makes verifiable computation easy to get started with

Q&A with Brian

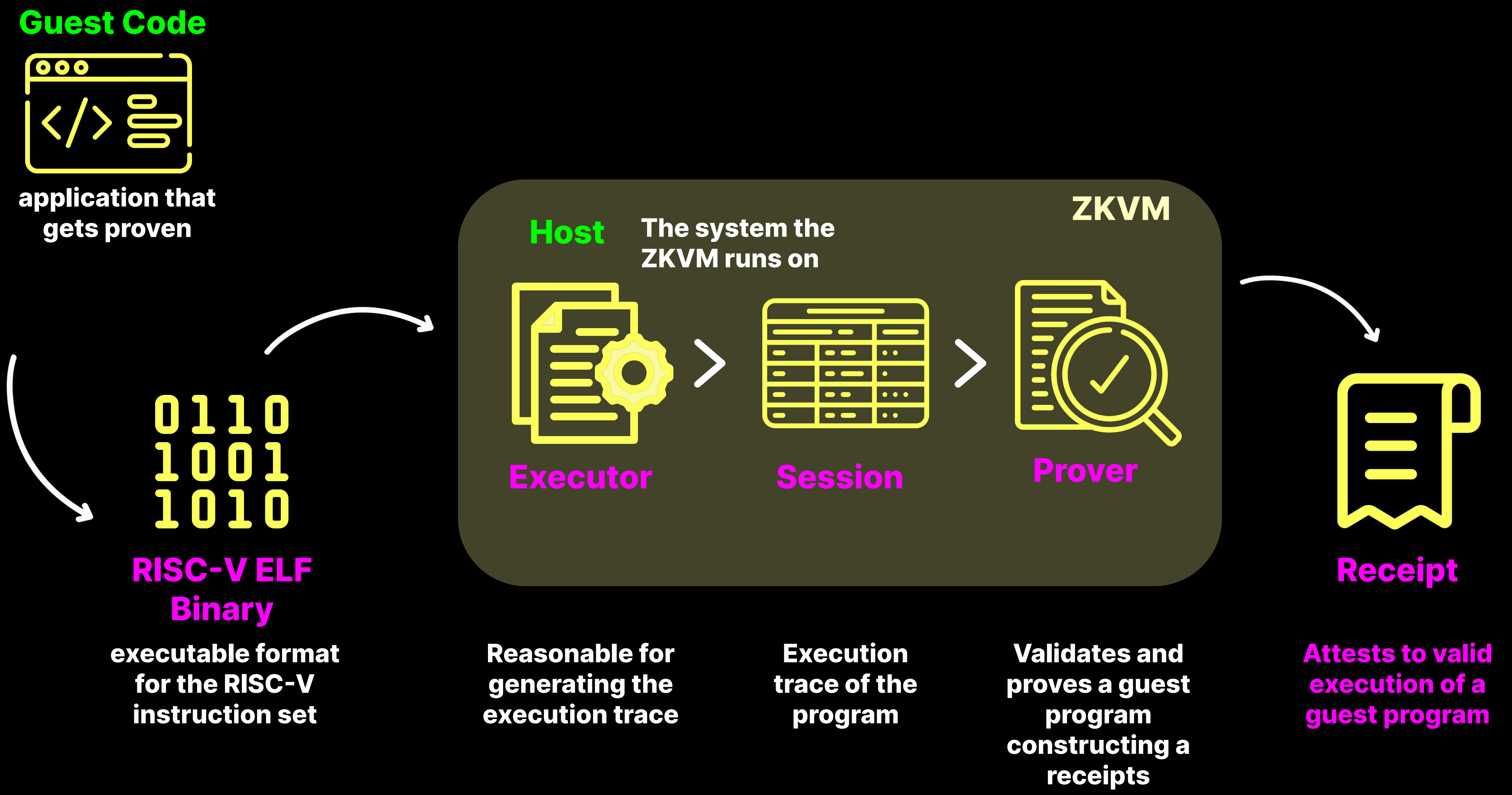
RISC ZERO

Terminology

Guest: The program running inside the zkVM.

Host: The system the zkVM runs on.

Prover: Part of zkVM that generates a proof.



Guest Code

application that gets proven

RISC-V ELF Binary

executable format for the RISC-V instruction set

ZKVM

Host The system the ZKVM runs on

Executor

Reasonable for generating the execution trace

Session

Execution trace of the program

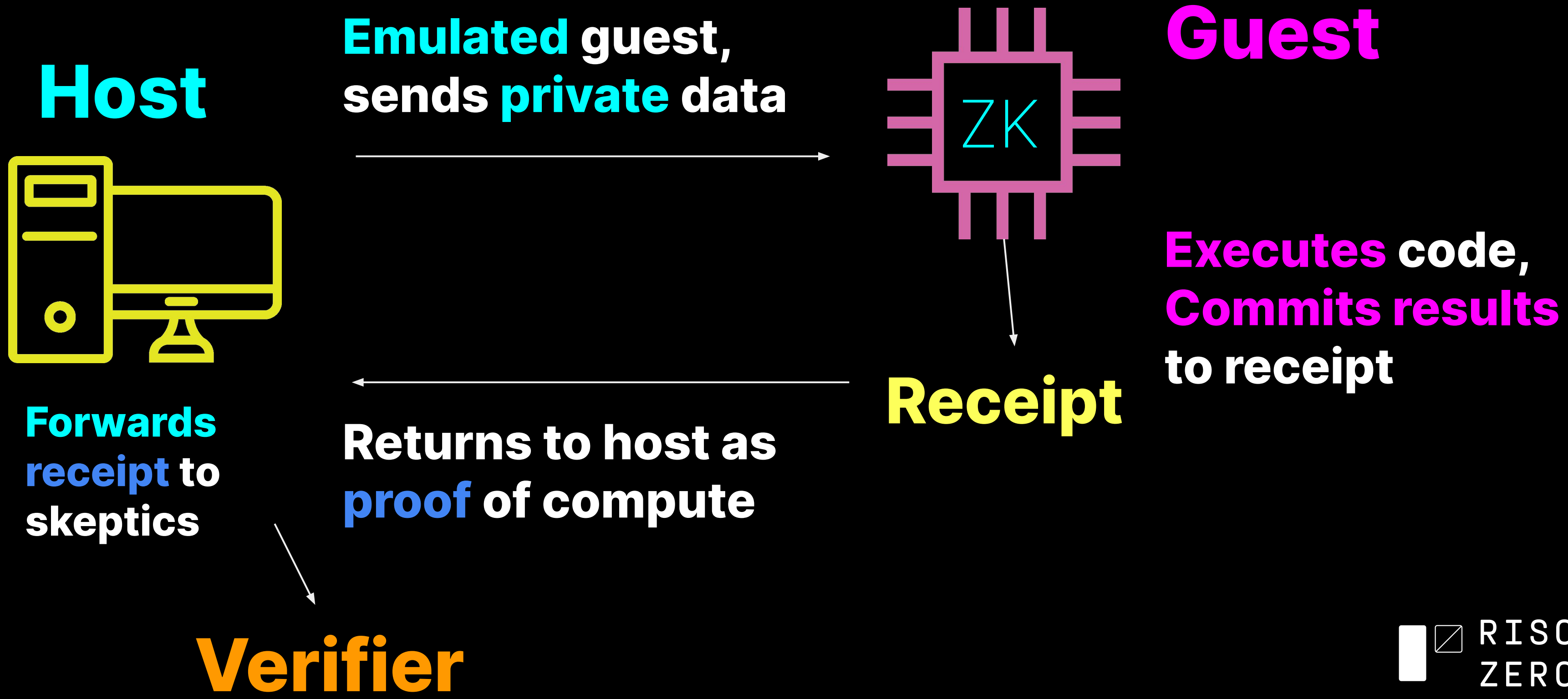
Prover

Validates and proves a guest program constructing a receipts

Receipt

Attests to valid execution of a guest program

Proving



Host

Emulated guest, sends private data

Guest

ZK

Executes code, Commits results to receipt

Receipt

Returns to host as proof of compute

Forwards receipt to skeptics

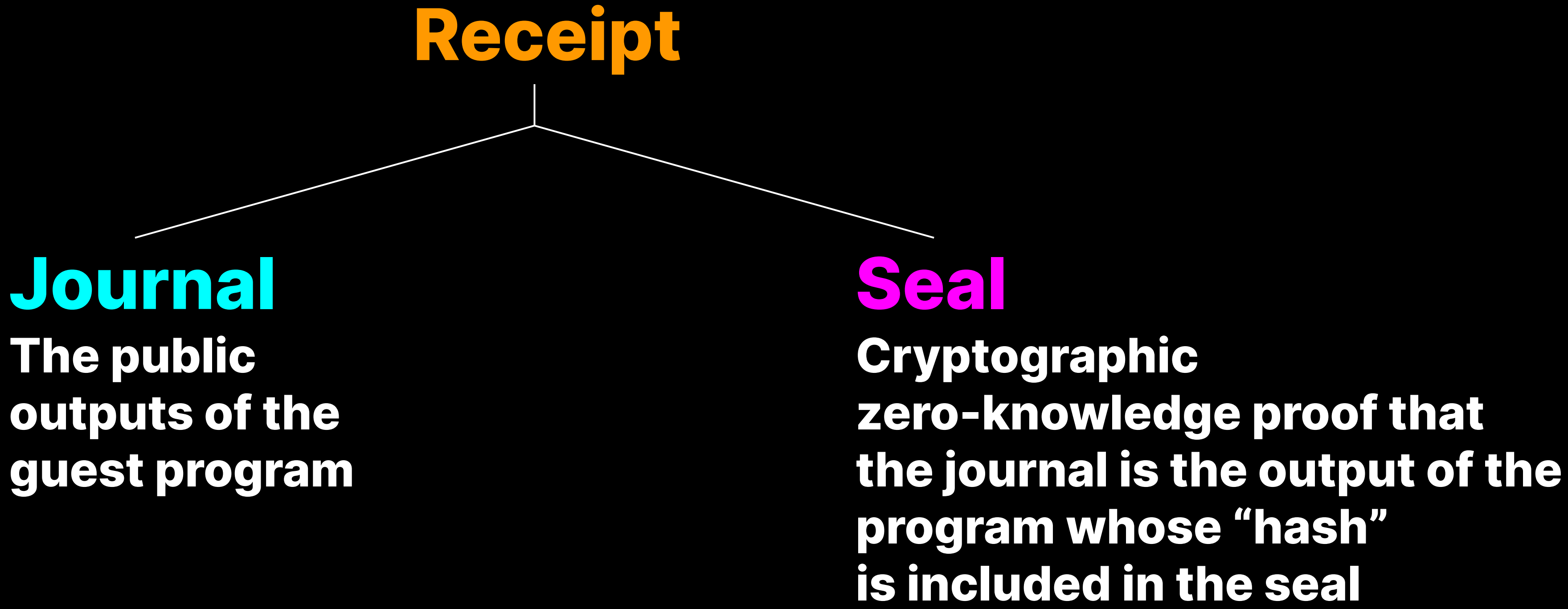
Verifier

Receipt: A receipt attests to valid execution of a guest program.

Journal: The portion of the receipt that contains the public outputs of a guest.

Seal: Hash of the proof that is passed for validation.

Receipt: Verification



Receipt

Journal

The public outputs of the guest program

Seal

Cryptographic zero-knowledge proof that the journal is the output of the program whose “hash” is included in the seal

Advanced:
Let's look under the hood

RISC ZERO

Advanced: What's in a Receipt?

The seal of a RISC Zero zkVM receipt is a zk-STARK

- Scalable Transparent ARgument of Knowledge

The prover & verifier

- Use FRI and DEEP-ALI
- With the Fiat-Shamir heuristic implemented using SHA-2
- To prove/verify that the execution trace
- Satisfies appropriate constraints

zkVM as a VM

The RISC Zero zkVM is a virtual machine with a RISC-V instruction set architecture (ISA)

- Open
- Lightweight
- Common compilation target

When you execute guest code, it executes instructions from this ISA in the same way any other implementation of this ISA in the same way any other implementation of this ISA would do

Extensions for SHA and finite fields

Advanced: Execution trace

- Not just a Virtual Machine
  - The prover records the state of the VM as an execution trace
    - Each row is a clock cycle
    - Each column is a register

If the only thing you care about is proving/checking correct execution, this is enough. But…

- zero-knowledge

Advanced: Trace as Witness

Why is verifying the trace?

- The initial state matches the claimed code
- The results are as claimed
- There is computational integrity: each step must be what a RISC-V processor would do

We encode the trace algebraically

- The above conditions become algebraic constraints
- The encoded trace is called witness

RISC ZERO

Continuations

Why get Excited???

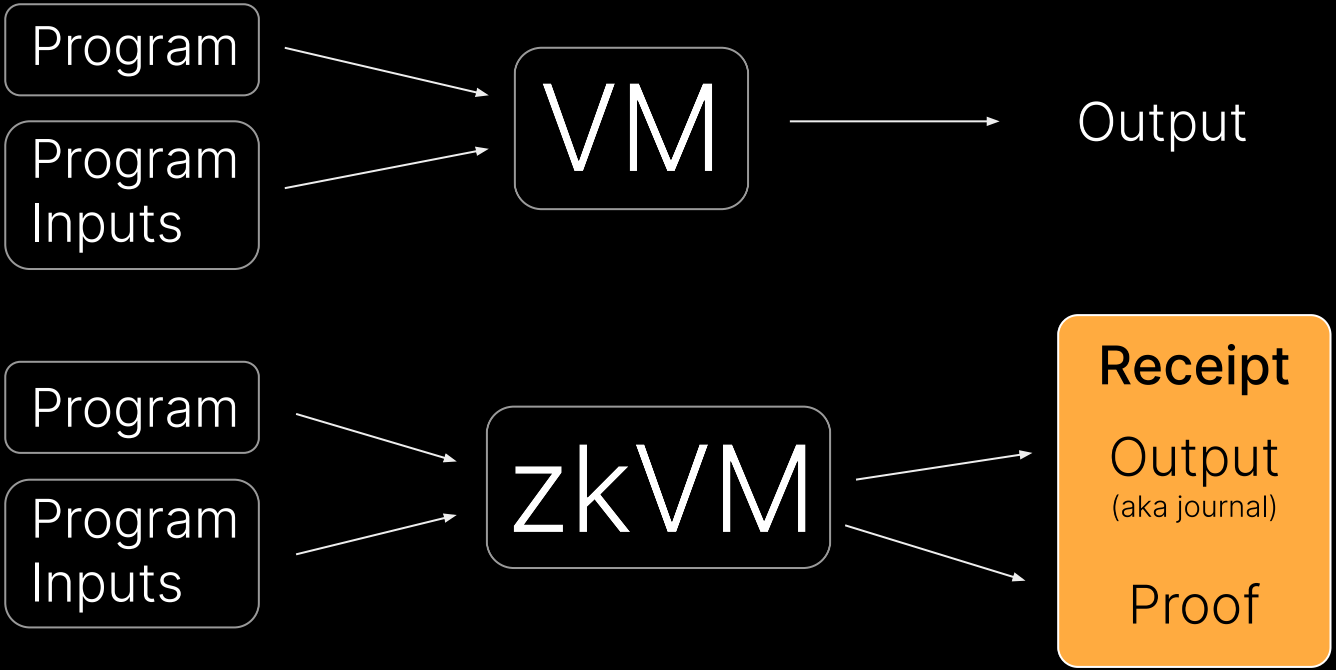
Max Computation Size

Before Continuations: 16 million cycles

After Continuations: **~ 10 billion cycles**

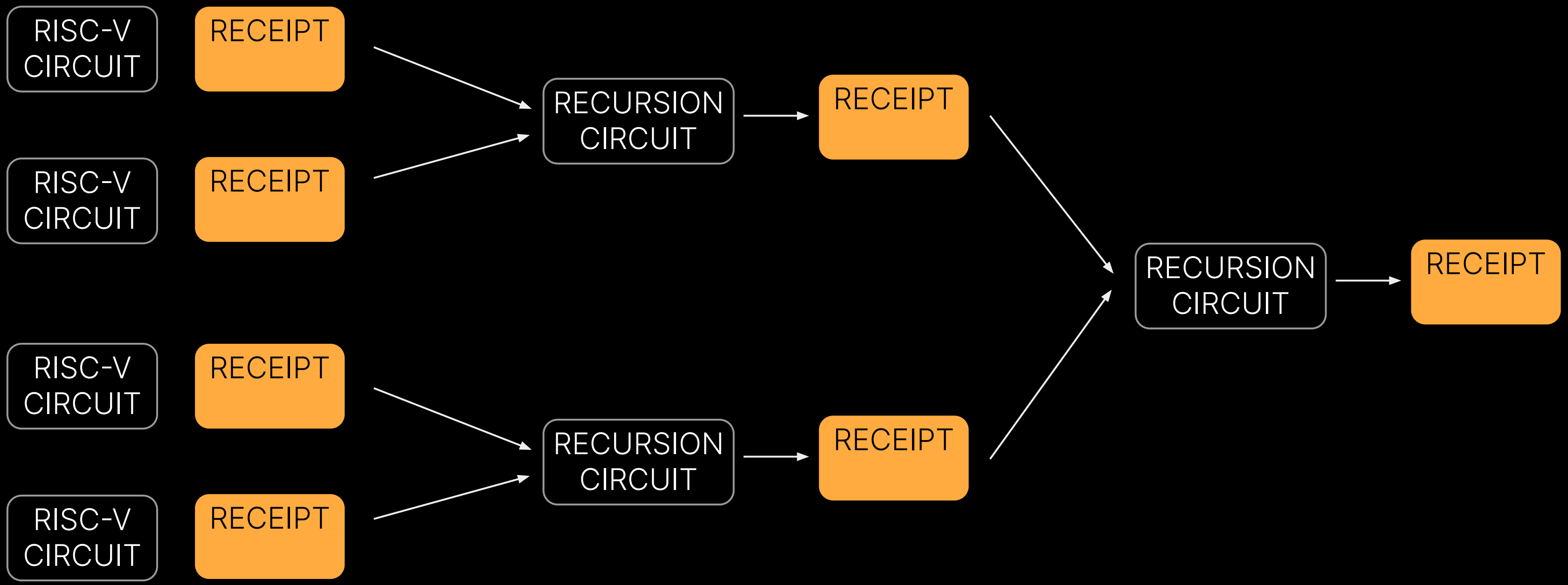
Proof System

General purpose zkVMs are here



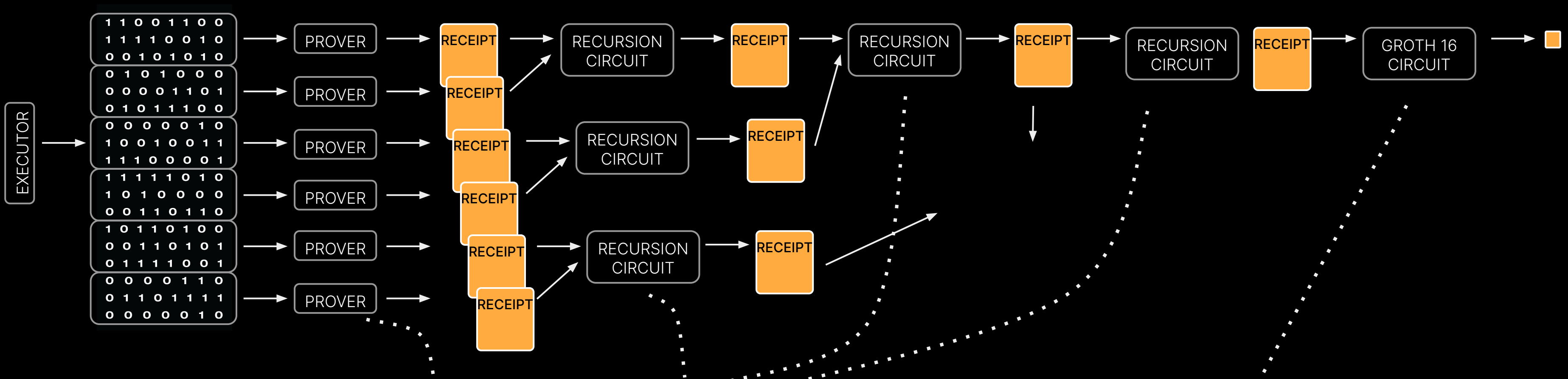
RISC ZERO

Two STARK Circuits



RISC ZERO

The Emergent Pattern



FRI

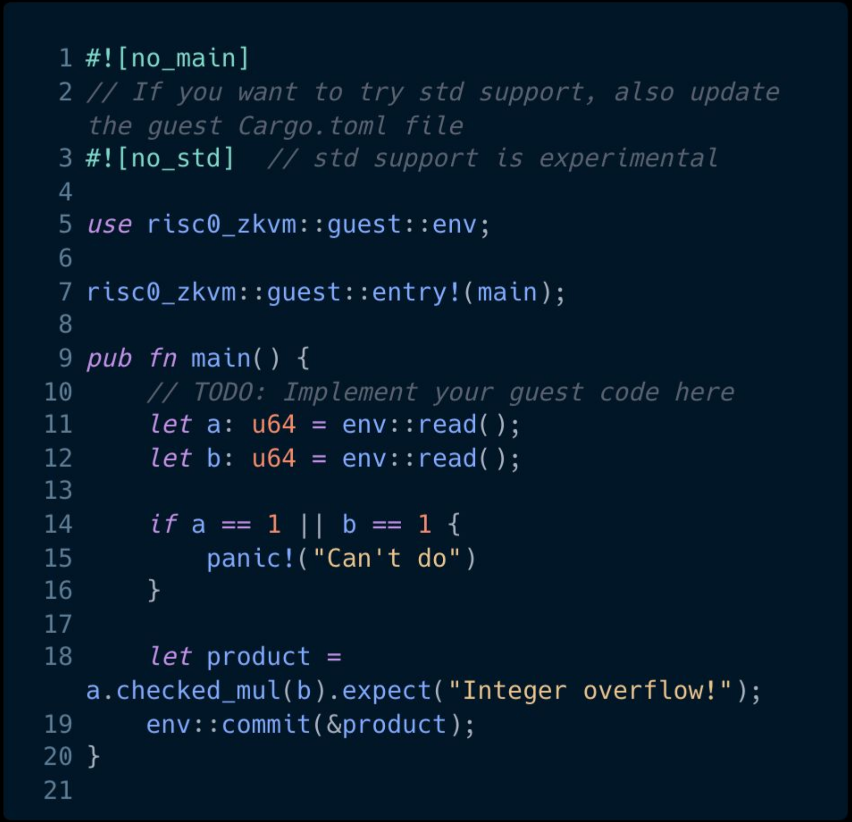
FRI

KZG/Groth16

Example Walkthrough of Factors

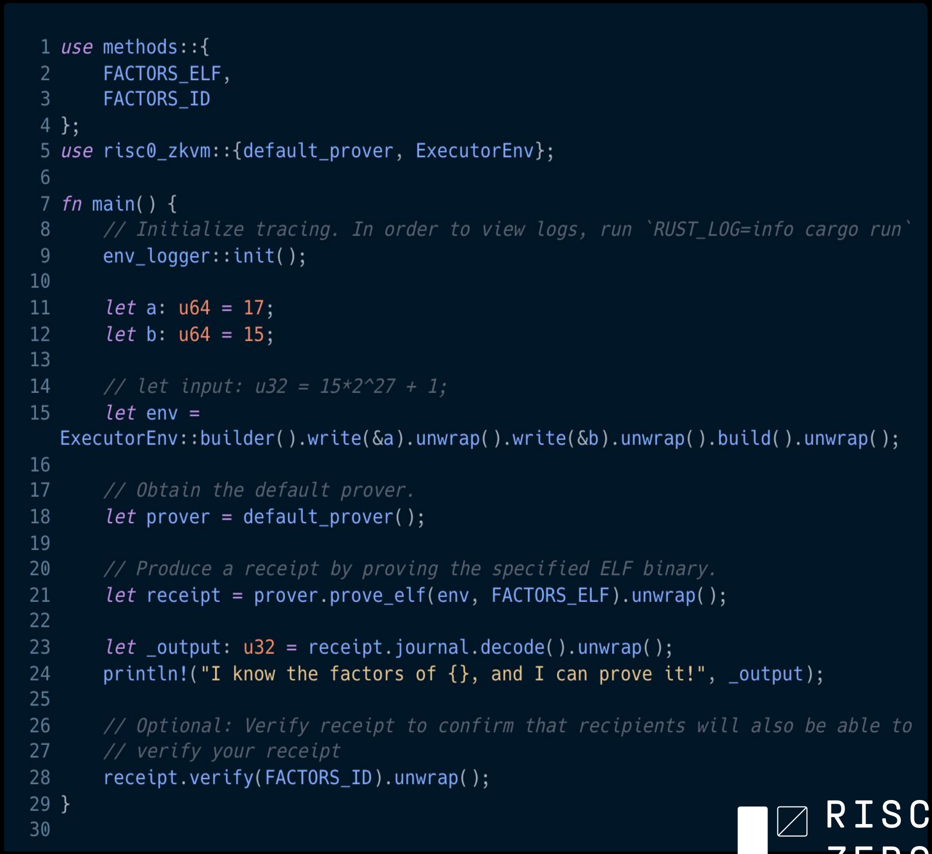
RISC ZERO

methods/guest/src/main.rs



```rust
#![no_main]
// If you want to try std support, also update the guest Cargo.toml file
#![no_std]  // std support is experimental

use risc0_zkvm::guest::env;

risc0_zkvm::guest::entry!(main);

pub fn main() {
    // TODO: Implement your guest code here
    let a: u64 = env::read();
    let b: u64 = env::read();

    if a == 1 || b == 1 {
        panic!("Can't do")
    }

    let product = a.checked_mul(b).expect("Integer overflow!");
    env::commit(&product);
}

```

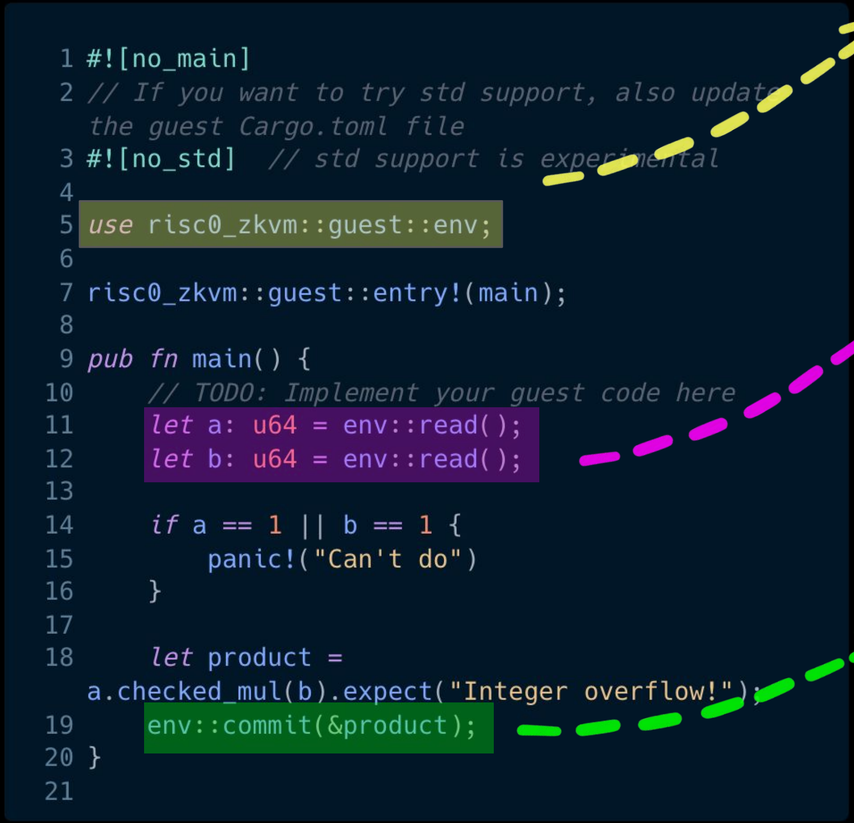
host/src/prover.rs



```rust
use methods::{
    FACTORS_ELF,
    FACTORS_ID
};
use risc0_zkvm::{default_prover, ExecutorEnv};

fn main() {
    // Initialize tracing. In order to view logs, run `RUST_LOG=info cargo run`
    env_logger::init();

    let a: u64 = 17;
    let b: u64 = 15;

    // let input: u32 = 15*2^27 + 1;
    let env = ExecutorEnv::builder().write(&a).unwrap().write(&b).unwrap().build().unwrap();

    // Obtain the default prover.
    let prover = default_prover();

    // Produce a receipt by proving the specified ELF binary.
    let receipt = prover.prove_elf(env, FACTORS_ELF).unwrap();

    let _output: u32 = receipt.journal.decode().unwrap();
    println!("I know the factors of {}, and I can prove it!", _output);

    // Optional: Verify receipt to confirm that recipients will also be able to
    // verify your receipt
    receipt.verify(FACTORS_ID).unwrap();
}

```

RISC ZERO

methods/guest/src/main.rs



```rust
#![no_main]
// If you want to try std support, also update the guest Cargo.toml file
#![no_std]  // std support is experimental

use risc0_zkvm::guest::env;

risc0_zkvm::guest::entry!(main);

pub fn main() {
    // TODO: Implement your guest code here
    let a: u64 = env::read();
    let b: u64 = env::read();

    if a == 1 || b == 1 {
        panic!("Can't do")
    }

    let product = a.checked_mul(b).expect("Integer overflow!");
    env::commit(&product);
}

```

Import functions for interacting with the host environment

Read the objects from the host

Commit the public output to the journal

Pick 2 numbers

Set up the Executor Environment

This holds configuration details that inform how the guest environment is set up prior to guest code execution

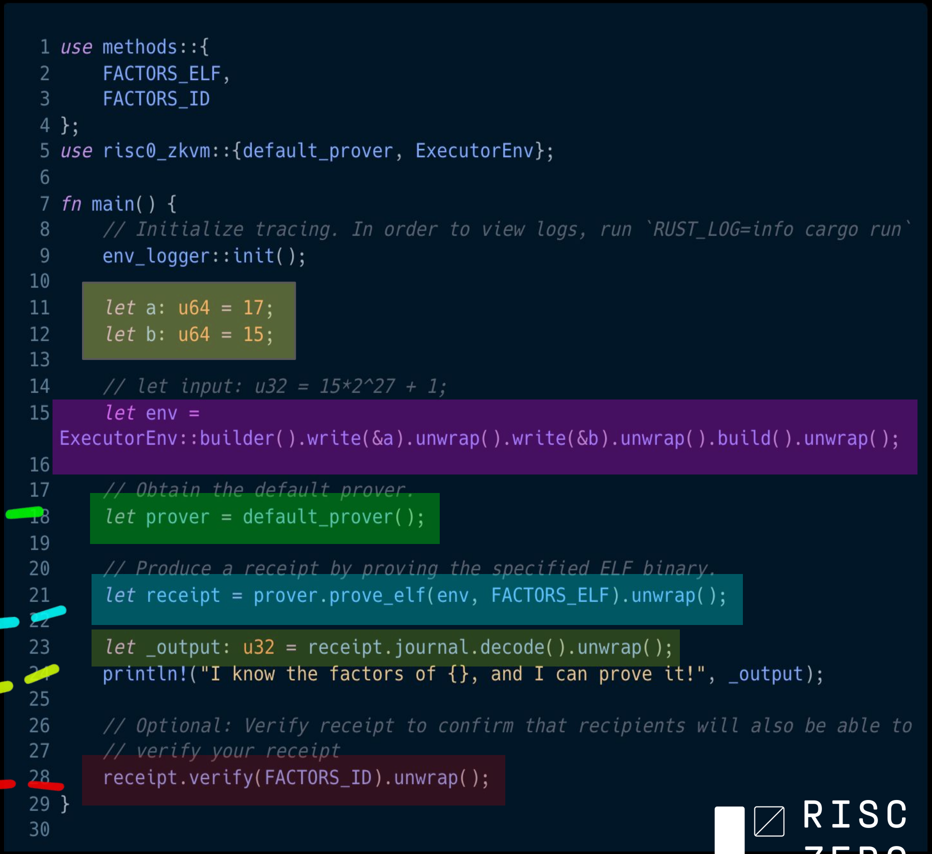
Obtain default prover

Prove the ELF Binary and return a receipt

Extract output from journal

Verify the integrity of this receipt w. image id

host/src/prover.rs



```rust
use methods::{
    FACTORS_ELF,
    FACTORS_ID
};
use risc0_zkvm::{default_prover, ExecutorEnv};

fn main() {
    // Initialize tracing. In order to view logs, run `RUST_LOG=info cargo run`
    env_logger::init();

    let a: u64 = 17;
    let b: u64 = 15;

    // let input: u32 = 15*2^27 + 1;
    let env = ExecutorEnv::builder().write(&a).unwrap().write(&b).unwrap().build().unwrap();

    // Obtain the default prover.
    let prover = default_prover();

    // Produce a receipt by proving the specified ELF binary.
    let receipt = prover.prove_elf(env, FACTORS_ELF).unwrap();

    let _output: u32 = receipt.journal.decode().unwrap();
    println!("I know the factors of {}, and I can prove it!", _output);

    // Optional: Verify receipt to confirm that recipients will also be able to
    // verify your receipt
    receipt.verify(FACTORS_ID).unwrap();
}

```

Quick start with zkVM

https://dev.risczero.com/api/zkvm/quickstart

What’s possible?

Fibonacci demo

RISC
ZERO

Chess demo

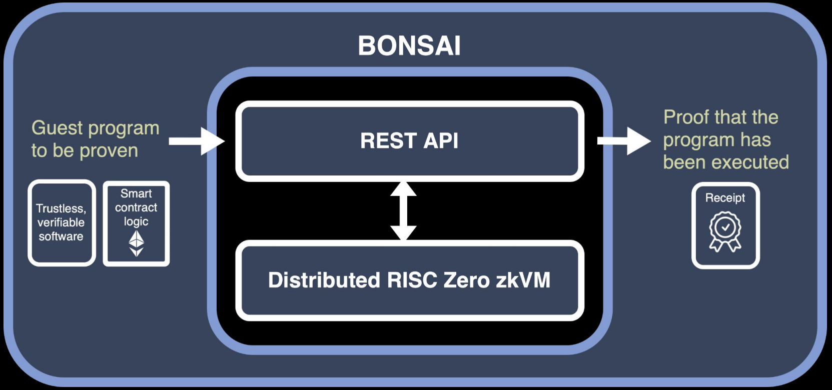
What if I don’t want to run the zkVM locally? Is there a remote option?

Bonsai

RISC ZERO

Bonsai

- Remote proving
- We provide the infrastructure
- Ideal for heavy computations & fast proving
- Offchain computation



BONSAI

Guest program to be proven

Trustless, verifiable software

Smart contract logic

REST API

Distributed RISC Zero zkVM

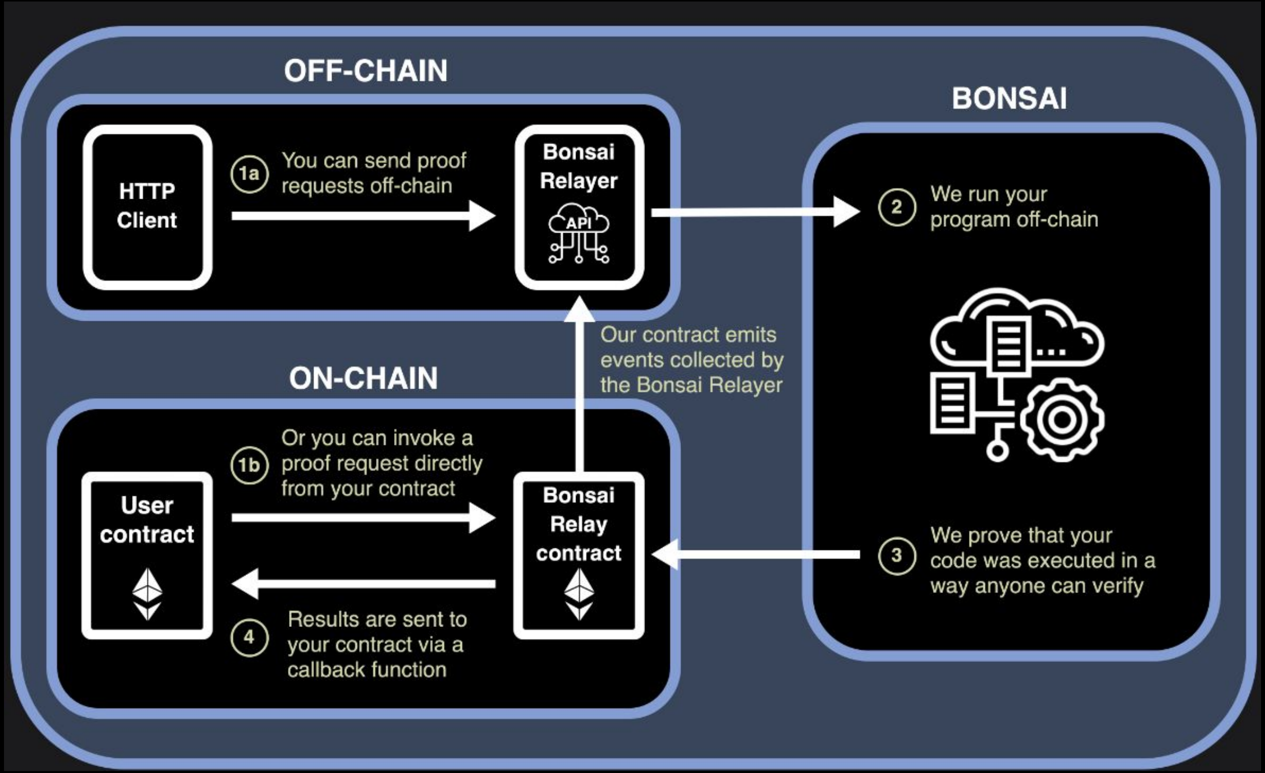
Proof that the program has been executed

Receipt

Bonsai API key

RISC ZERO

Bonsai Architecture



1a You can send proof requests off-chain

2 We run your program off-chain

Our contract emits events collected by the Bonsai Relayer

1b Or you can invoke a proof request directly from your contract

3 We prove that your code was executed in a way anyone can verify

4 Results are sent to your contract via a callback function

RISC ZERO

What’s possible?

Bonsai Pay

Questions?

SCAN ME!
(Bonsai Apply)
RISC ZERO

SCAN ME!
(Bonsai Docs)
RISC ZERO

SCAN ME!
(zkVM)
RISC ZERO