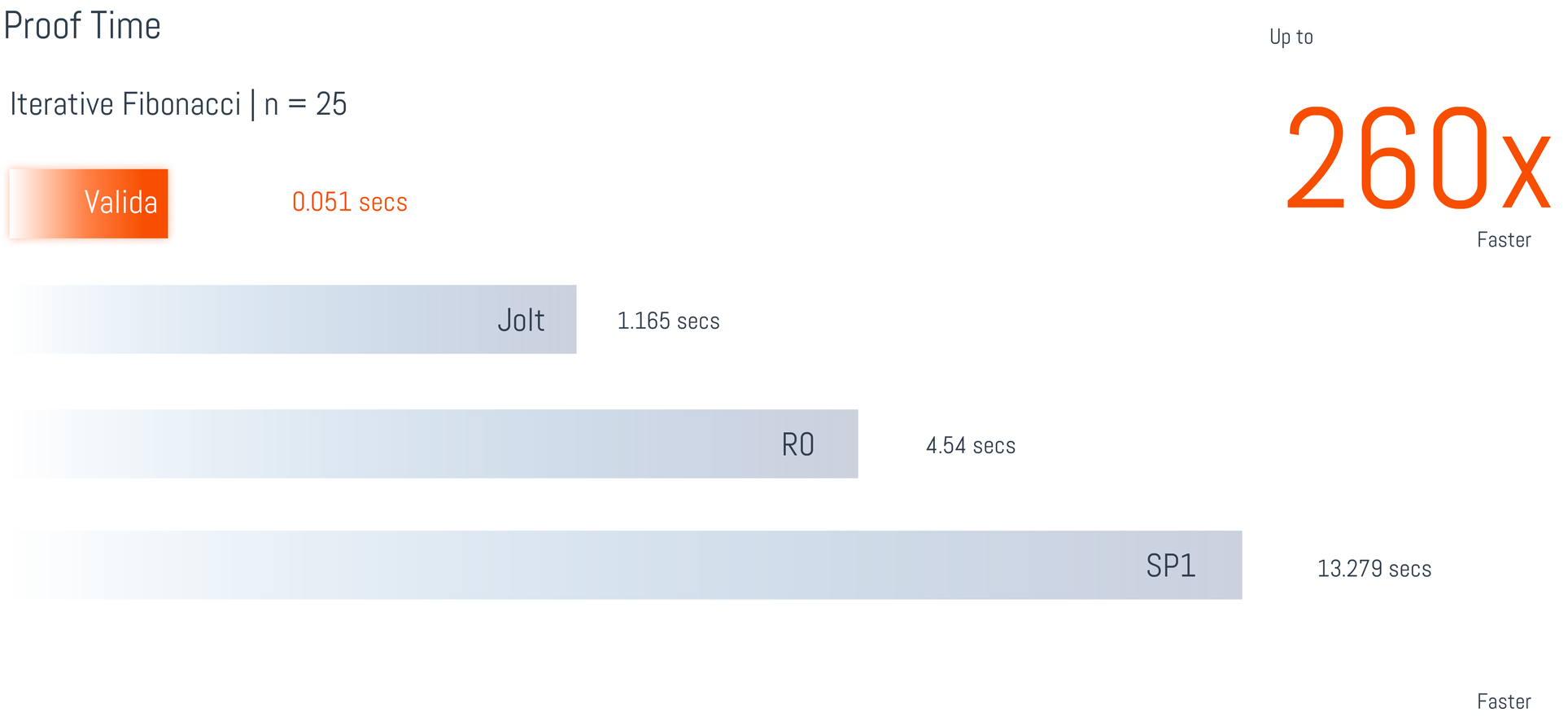
lita

Pushing the Performance and Usability of Zero Knowledge Proofs

Morgan Thomas, Daniel Lubarov, Marty Stumpf, Hadas Zeilberger,
Ventali Tan, Alan Li, Michael Liao, Dan Dore

Valida offers Industry Leading Speed



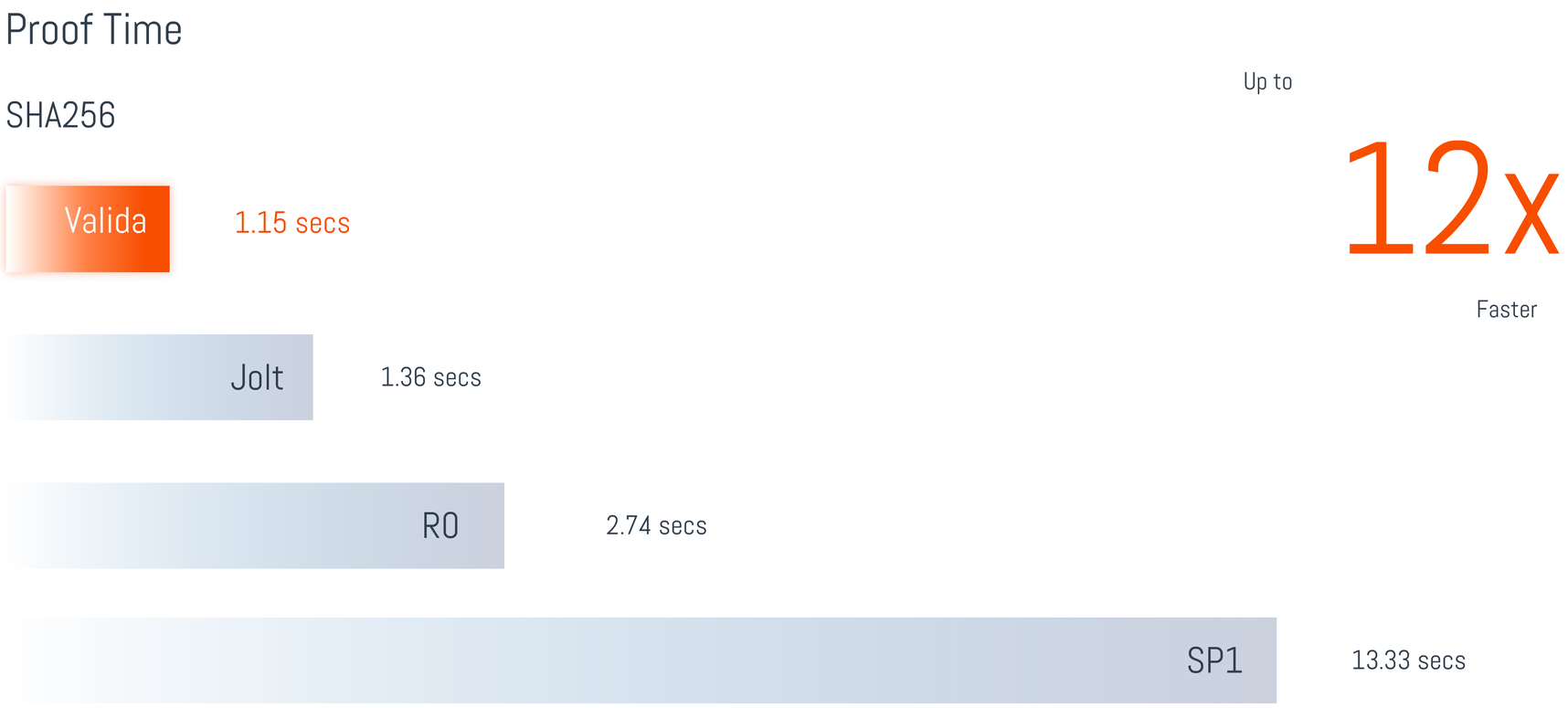
Up to

260x

Faster

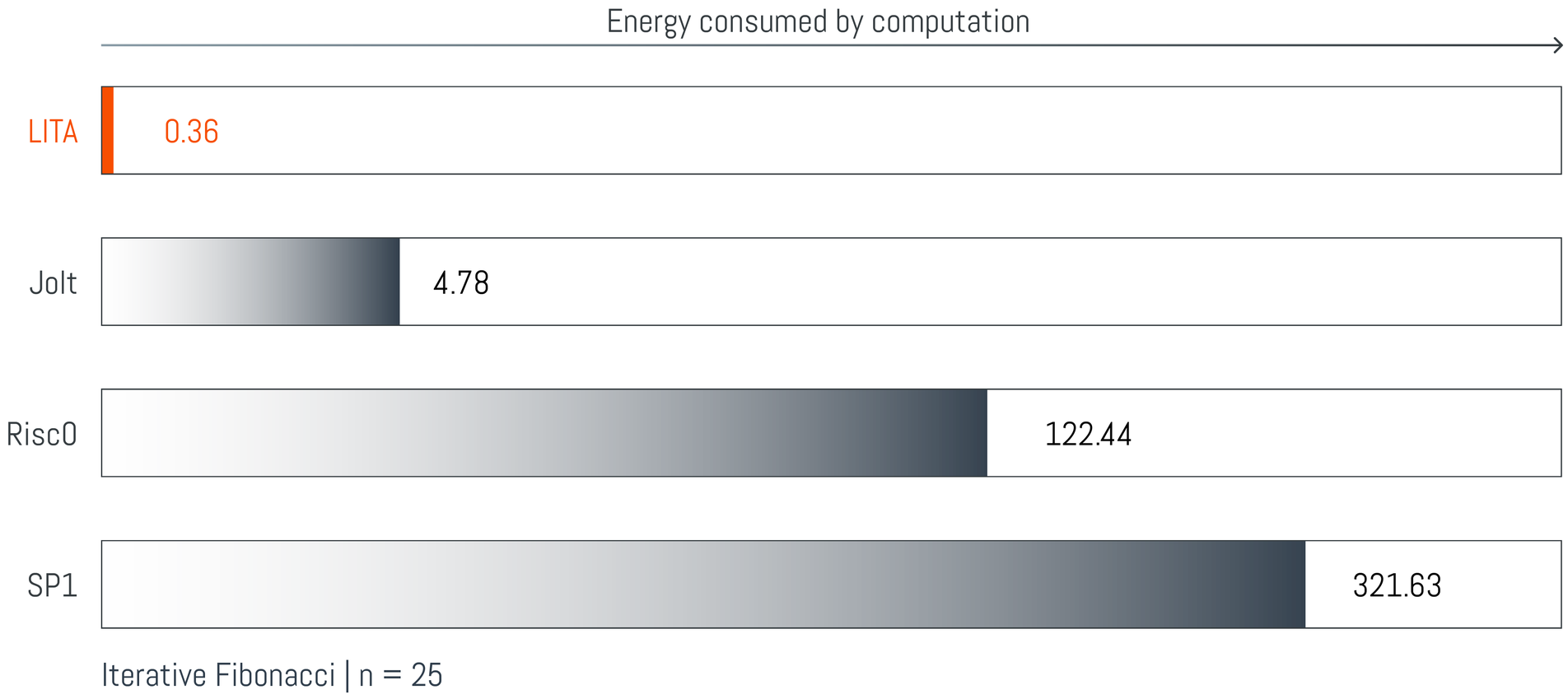
Speed is measured in Wall Clock Time: this measures the actual elapsed time from the start to the end of the computation. In the same example, the wall clock time would be 5 seconds, regardless of the number of cores used. Wall clock time reflects the waiting time for users.

Valida offers Industry Leading Speed



Speed is measured in Wall Clock Time: this measures the actual elapsed time from the start to the end of the computation. These benchmarks offer a limited view of the performance characteristics and the results presented do not fully capture the nuanced behavior and capabilities of the systems across diverse real-world use cases and deployment scenarios.

Valida Maximizes CPU Efficiency



User Time: The amount of computer time expended by the program code. For instance, if a program uses 5 cores for 5 seconds each, the user time is 25 seconds (5 cores x 5 seconds). User time is a crucial indicator of the energy consumed by the computation.

Table of Contents

1. Why zk-VM?: DSL, Direct Compilation, ISA / VM

2. Limitations with existing zk-VMs

3. Building the fastest VM: Optimizing prover, zkVM and compiler stack

4. How to provide an even better solution?

5. Benchmark & Demo

6. Future Directions

Motivations behind zk-VMs

How do you prove, succinctly and in zero knowledge, that a certain program was executed on a certain input and it had a certain result?

Motivations behind zk-VMs

Succinctness: proofs are not too long

Zero-Knowledge: the proofs contain no useful information beyond the facts that they are proving

Problems:

- the learning curve of the tooling, and
- the high time, space, and energy cost of creating proofs

DSL, Direct Compilation, ISA / VM

Approach	Description	Advantages	Disadvantages
DSL Approach	Use of domain-specific languages to build abstractions on top of low-level constraint systems like R1CS or PLONK, acting like constraint-writing macros.	Easy to iteratively build abstractions on constraints, allowing for hand-rolled circuit optimizations and a closer reasoning about performance.	Limited in high-level abstraction capabilities without complex compiler passes; different DSLs for different proof systems may introduce learning barriers.
Direct Approach	Directly compiling high-level programs to circuits without an intermediate instruction set, potentially using techniques like 'Compiling to Categories' for functional languages.	Potentially very high performance as it doesn't lose information to intermediate representations; clear mathematical model that is easy to verify.	Complexity in compiler pipeline and inability to reuse most existing tooling; may challenge developers in reasoning about program performance.
ISA/VM Approach	Defining an intermediate virtual machine and instruction set architecture to mediate between high-level programming languages and low-level ZKP systems.	Allows for the use of higher-level languages more familiar to developers, and, in some cases, reuses tons of existing tooling (e.g., LLVM for RISC-V).	Potential cost of abstraction in performance; more complex verification for correctness due to power and capability.

This table highlights the unique aspects of each approach, along with their potential benefits and challenges, providing a comprehensive overview for evaluating and choosing a suitable compilation strategy for zero-knowledge proof systems and privacy-preserving applications.

Development productivity, security

zk-VM will be the dominant way developers interact with zero-knowledge

Limitations with existing zk-VMs

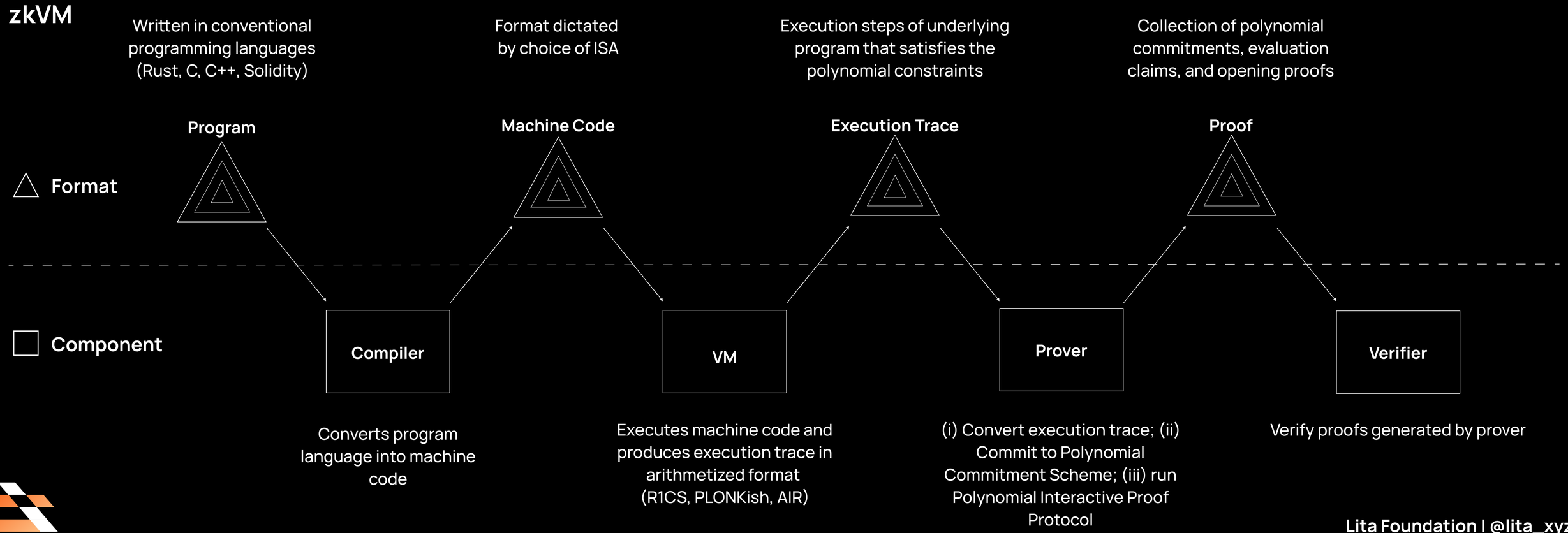
- DSL-based VMs (e.g. Cairo) require developers to work in an unfamiliar paradigm
- Lack of full-stack open-source support: prover, verifier, recursion, compilation/ constraint generation
- Inefficiencies in VM design: memory model, SNARK-unfriendliness
- Difficult to integrate cryptographic advances: e.g. support for small fields

We need a VM that is more extensible, performant and developer friendly

Building a VM from ground up

1. Select and build a base proving system from research literature that balances low time, space, and energy complexity with succinct proof sizes for optimal performance.
2. Develop zero-knowledge proofs of program execution using a zk-VM with a zk-friendly ISA, tailored for efficient zero-knowledge proving.
3. Create a compiler toolchain to convert popular programming languages to the zk-friendly ISA, enabling wider application development.
4. Implement hardware-accelerated zk-VM proving using advanced parallel computing technologies for enhanced efficiency.
5. Apply formal verification to ensure the integrity of critical system components like the verifier, STARK constraint system, and VM execution engine.

General Flow for a zkVM



Optimizing prover, zkVM and compiler design

Tricks to maximize prover performance

Prover

zk-VM

Compiler

differences between the zk-VM computing environment and the hardware computing environment can drive different ISA design decisions.

which instructions are more efficient to include in the ISA and check in the STARK constraint system

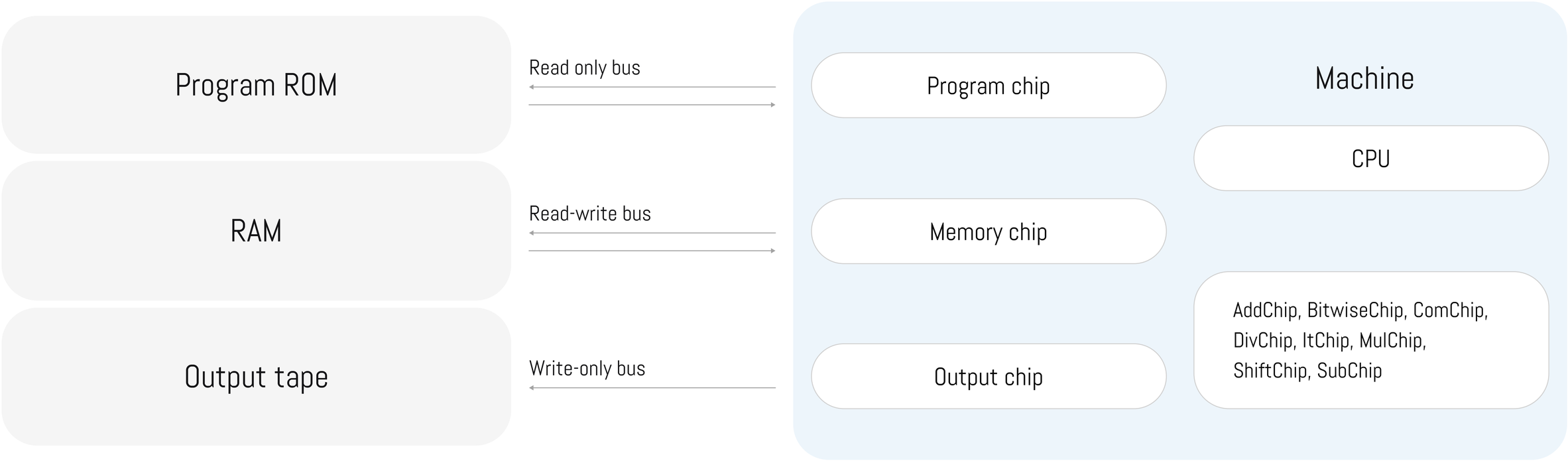
which instructions are more efficient to exclude from the ISA and eliminate from the compiler output by replacing them with sequences of instructions.

Valida Instruction Set Architecture (ISA)

- An ultra-RISC, modular, zk-friendly ISA
- Includes the following modules ("chips"):
  - 32-bit ALU chips
    - 32-bit unsigned integer arithmetic
    - Bitwise operations & bit shifts
  - CPU chip
  - Memory (RAM) chip
  - Program (ROM) chip
  - 8-bit range checker chip

Designed to be friendly for implementation in zero knowledge virtual machines (zk-VMs), specifically via STARKs over small fields

Diagram of a basic Valida VM



Valida ISA (continued)

- Two special-purpose registers
  - Program counter (PC)
  - Frame pointer (FP)
- No general-purpose registers
- Two RAM addressing modes
  - Relative to FP, with a constant offset
    - Supported by most opcodes
  - Absolute
    - Supported by dedicated load / store opcodes
- Program ROM can't be addressed

This simplifies code generation by eliminating the need for registers to be saved and restored in the calling convention, and eliminating the need for register spilling code to be emitted

In the zk-VM setting, accessing RAM has no more cost than accessing a register, whereas in hardware, accessing RAM is far more costly

Valida ISA (continued)

- There is no dedicated stack pointer (SP) register
- The calling convention uses constant sized stack frames
- Variable size locals and arguments would be held on the heap

Valida zk-VM overview

- Execution engine
  - Runs Valida programs and generates execution traces
- Prover
  - Proves the result of executing a program by proving the existence of a valid execution trace
- Verifier
  - Verifies the result of executing a program, given a proof

Valida zk-VM parameters & default values

- Base finite field
  - Baby bear field ($p = 2^{31} - 2^{27} + 1$)

- Extension field where challenges are drawn from
  - Degree 5 extension of the Baby bear field
    - Take the ring of polynomials of degree five over the Baby bear field
    - Quotient that ring by the ideal generated by ($x^5 - 2$)
    - Since ($x^5 - 2$) is irreducible, the resulting quotient ring is a field

- The polynomial commitment scheme
  - FRI PCS

Valida zk-VM prover

1. Run the program and generate the execution traces
    a. One main execution trace per chip
    b. Preprocessed traces for the program ROM chip and 8-bit range checker chip

2. Encode execution traces as univariate polynomials over a finite field, using Barycentric Lagrange interpolation

3. Commit to those polynomials

4. Generate permutation argument traces, encode them as univariate polynomials, and commit to those polynomials

Valida zk-VM prover (continued)

1. Plug the trace polynomials into the polynomial constraints,
   resulting in a collection of univariate polynomials: $P_0(x), \ldots, P_n(x)$

2. Take a linear combination of the resulting polynomials using a challenge a:
   $$P(x) = P_0(x) + a * P_1(x) + \ldots + a^n * P_n(x)$$

3. Let $Z(x)$ be the zerofier polynomial of the evaluation domain

4. Compute and commit to the quotient $Q(x) = P(x) / Z(x)$

5. Generate proofs of the evaluations of the committed trace polynomials and $Q(x)$ at a challenge point

Valida zk-VM prover (continued)

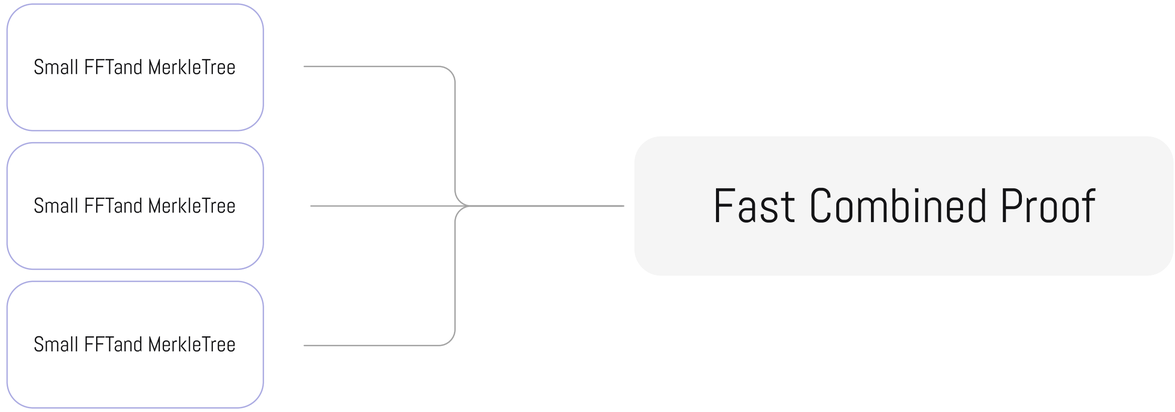
- The process just described is repeated for each sub-argument of the proof
- There is one sub-argument per chip, to validate its main execution trace
- There is a sub-argument for the permutation argument that checks all of the chip interactions

Techniques for Optimizing the Prover

- The proof system (STARK) that underlies the VM is very efficient and the only expensive cryptographic primitive it uses is a hash function
- Our VM uses gates with degree at most 3, keeping FFT and Hashing costs as low as possible

Execution Trace

ADDRESS	V1	V2
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...



Techniques For An Optimized Prover

- Valida uses a finite field that is extremely efficient. Most provers use finite fields with expensive arithmetic operations, where each operation is much slower than 32-bit or 64-bit arithmetic.
- Valida uses Baby Bear prime p = 2^31 - 2^27 + 1

Valida's Fast Finite Fields

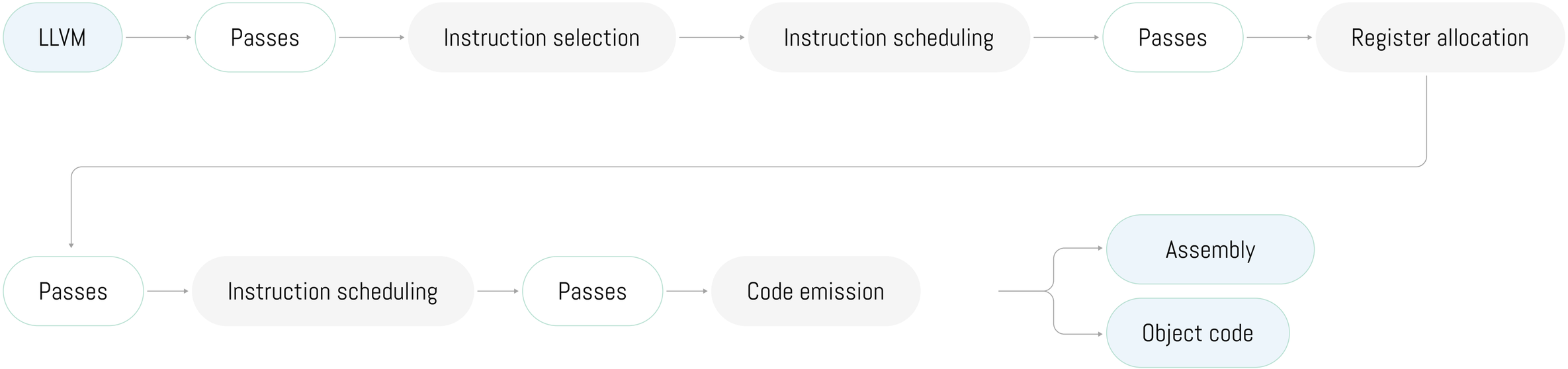
BabyBear
Mersenne31

≈

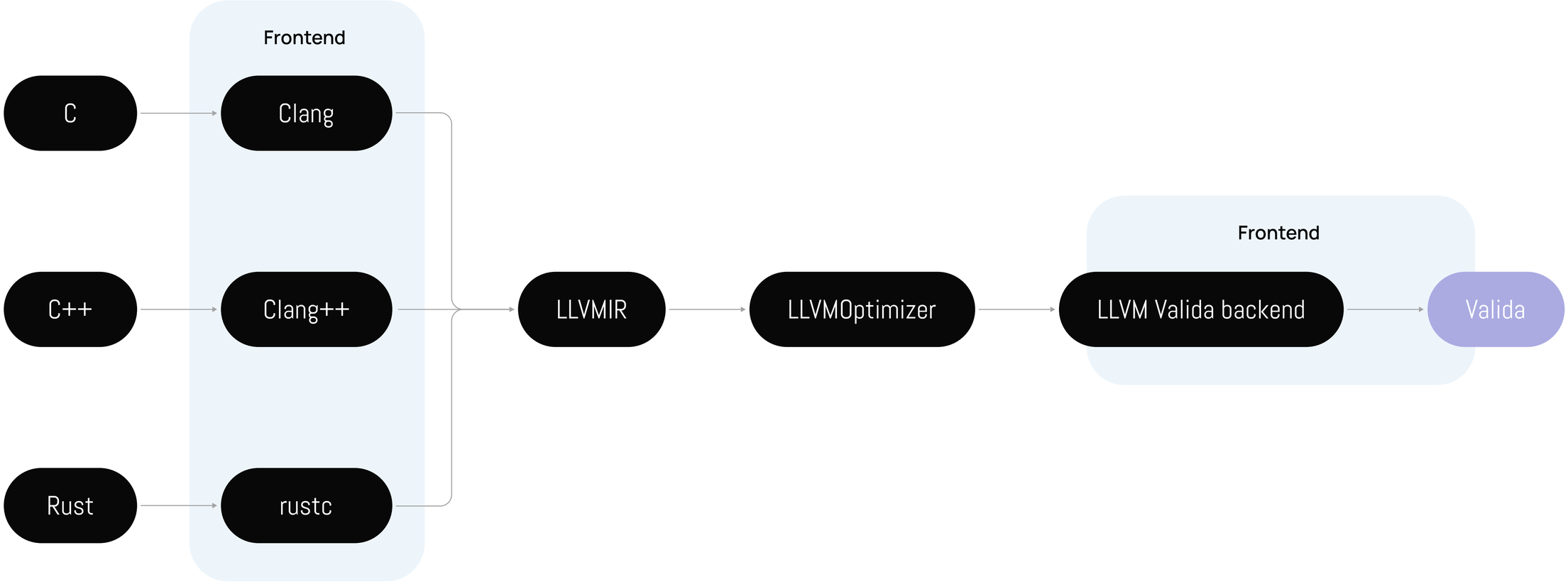
Native unsigned 32-bit integer arithmetic

The LLVM Valida compiler backend

- Utilizes the LLVM project to generate optimized IntermediateRepresentations (IR) from various frontends
- Current plans to support C, C++, Rust
- Compiles the LLVM IR to Valida assembly:



Compiler overview



Valida implementation: customizable

- Valida's implementation provides a framework for generating Valida VMs and provers using included and user-provided chips and compile-time parameters
- Application developers can use the included compile time macros to generate, with a little coding, a Valida VM and prover based on a chip set customized for their use case

https://github.com/valida-xyz/valida

Valida implementation: fully open-source

The Valida implementation is dual Apache / MIT licensed

https://github.com/valida-xyz/valida

Demo: Fibonacci

Valida / RISC Zero proving time benchmarks: Iterative Fibonacci

N	ASM VALIDA (SINGLE THREAD)		ASM VALIDA (32 THREADS)		RISC ZERO				
	USER TIME (S)	WALL CLOCK TIME (S)	USER TIME (S)	WALL CLOCK TIME (S)	USER TIME (S)	WALL CLOCK TIME (S)	BUILD TIME (S)	ADJUSTED USER TIME (S)	ADJUSTED WALL CLOCK TIME (S)
25	0.07	0.08	1.07	0.1	103.4	34.97	32.17	71.23	2.8
25	0.07	0.07	1.01	0.1	103.18	35.05	32.23	70.95	2.82
25	0.08	0.08	1.06	0.1	102.78	34.98	32.18	70.6	2.8
25	0.08	0.08	1.04	0.1	103.72	35.14	32.31	71.41	2.83
25	0.08	0.09	0.97	0.1	103	34.88	32.08	70.92	2.8
25	0.07	0.08	1.09	0.11	103.28	35.02	32.18	71.1	2.84
25	0.08	0.09	1	0.1	102.99	35.09	32.25	70.74	2.84
25	0.06	0.08	0.94	0.09	102.83	34.97	32.18	70.65	2.79
25	0.09	0.11	0.99	0.09	103.39	35.1	32.28	71.11	2.82
25	0.08	0.08	1.02	0.09	103.07	34.85	32.02	71.05	2.83
Mean	0.076	0.084	1.019	0.098				70.976	2.817
Median	0.08	0.08	1.015	0.1				71	2.82
Std Dev	0.008432740427	0.010749677	0.04677368681	000632455532				0.2579491767	0.0182878223
Valida Advantage	933.8947368	33.53571429	69.65260059	28.74489796					

Promising Intial Results: 33x Faster, 933x More Efficient

Valida / RISC Zero proving time benchmarks: Recursive Fibonacci

N	VALIDA (SINGLE THREAD)		VALIDA (32 THREADS)		RISC ZERO					
	USER TIME (S)	WALL CLOCK TIME (S)	USER TIME (S)	WALL CLOCK TIME (S)	USER TIME (S)	WALL CLOCK TIME (S)	BT, METHODS (S)	BT, HOST (S)	ADJUSTED USER TIME (S)	ADJUSTED WALL CLOCK TIME (S)
13	0.87	0.9	3.11	0.24	154.94	35.75	0.01	31.1	123.83	4.64
13	0.86	0.88	3.37	0.23	153.69	35.33	0.01	30.76	122.92	4.56
13	0.85	0.89	3.6	0.24	153.56	35.4	0.01	30.83	122.71	4.55
13	0.85	0.87	3.2	0.21	153.64	35.45	0.01	30.83	122.8	4.61
13	0.87	0.9	3.2	0.21	153.61	35.39	0.01	30.81	122.79	4.57
13	0.86	0.89	3	0.22	153.32	35.31	0.01	30.74	122.57	4.56
13	0.86	0.88	2.99	0.2	153.82	35.31	0.01	30.71	123.1	4.59
13	0.86	0.9	3.17	0.2	154.33	35.4	0.01	30.8	123.52	4.59
13	0.87	0.89	3.48	0.23	153.85	35.36	0.01	30.74	123.08	4.59
13	0.84	0.87	3.99	0.21	154.06	35.3	0.01	30.65	123.41	4.65
Mean	0.859	0.887	3.211	0.219					123.073	4.591
Median	0.86	0.89	3.185	0.215					123	4.59
Std Dev	0.00994428926	0.01159501809	0.2116837263	0.01523883927					0.40138787	0.03381321241
Valida Advantage	143.2747381	5.175873732	38.32855808	20.96347032						

Valida / RISC Zero proving time benchmarks: Recursive Fibonacci

N	VALIDA (SINGLE THREAD)		VALIDA (32 THREADS)		RISC ZERO					
	USER TIME (S)	WALL CLOCK TIME (S)	USER TIME (S)	WALL CLOCK TIME (S)	USER TIME (S)	WALL CLOCK TIME (S)	BT, METHODS (S)	BT, HOST (S)	ADJUSTED USER TIME (S)	ADJUSTED WALL CLOCK TIME (S)
16	3.73	3.83	9.68	0.65	273.63	39.67	0.08	30.8	242.75	8.79
16	3.61	3.73	9.52	0.67	277.27	39.57	0.01	30.69	246.57	8.87
16	3.72	3.81	9.26	0.65	277.66	39.68	0.01	30.81	246.84	8.86
16	3.73	3.81	9.46	0.68	277.92	39.7	0.01	30.77	247.14	8.92
16	3.68	3.82	8.84	0.68	277.21	39.55	0.01	30.69	246.51	8.85
16	3.7	3.8	9.29	0.66	273.11	39.63	0.01	30.77	242.33	8.85
16	3.67	3.79	8.96	0.67	276.87	39.59	0.01	30.69	246.17	8.89
16	3.68	3.77	10.13	0.68	275.56	39.51	0.01	30.69	244.86	8.81
16	3.69	3.78	9.13	0.66	278.22	39.92	0.01	31.05	247.16	8.86
16	3.64	3.74	9.5	0.66	277.45	39.66	0.01	30.81	246.63	8.84
Mean	3.685	3.788	9.377	0.666					245.696	8.854
Median	3.685	3.795	9.375	0.665					246.54	8.855
Std Dev	0.03865804502	0.03326659987	0.3720528636	0.01173787791					1.78787397	0.03687817783
Valida Advantage	66.67462687	2.337381204	26.20198358	13.29429429						

Valida / RISC Zero proving time benchmarks: Recursive Fibonacci

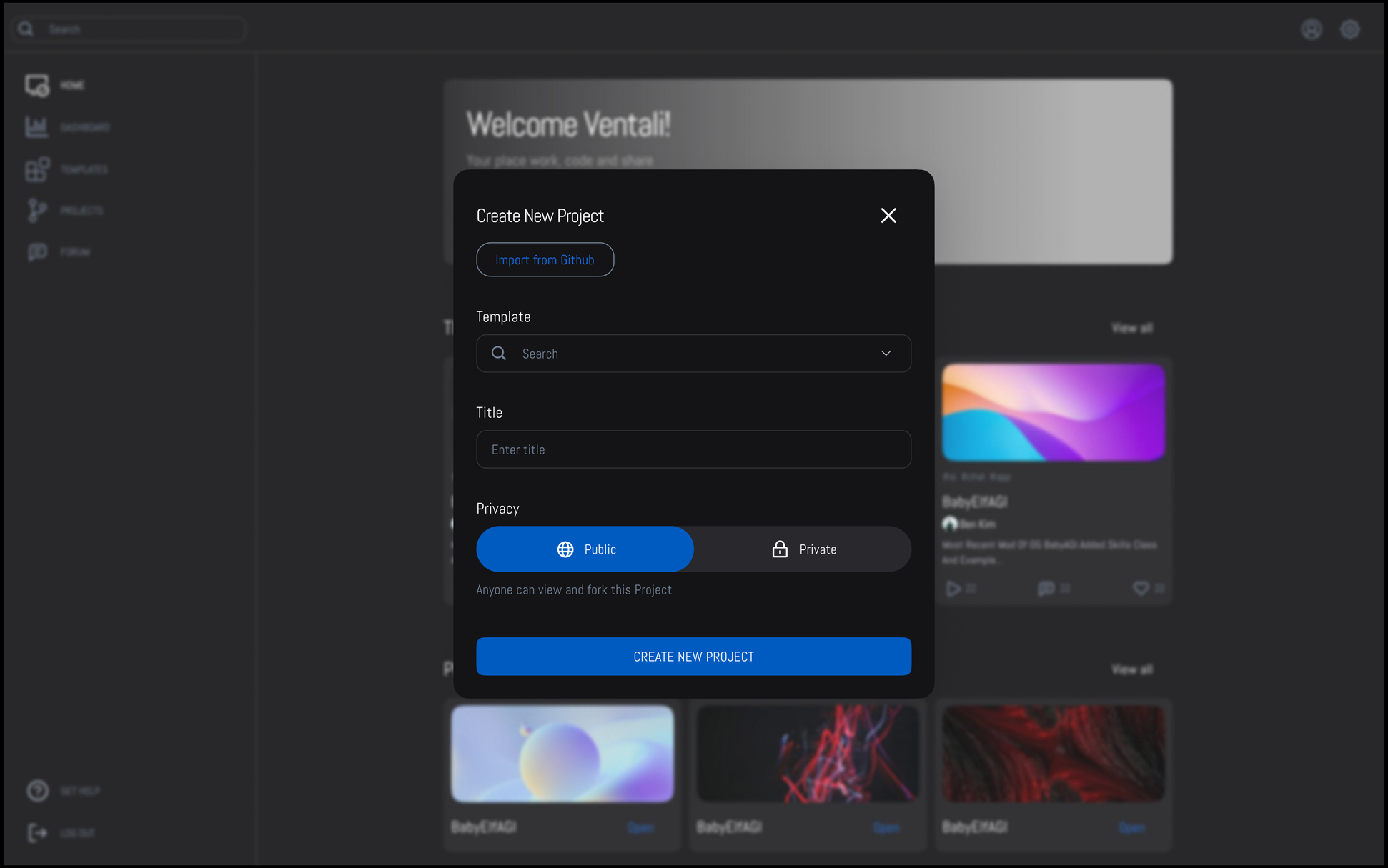
N	VALIDA (SINGLE THREAD)		VALIDA (32 THREADS)		RISC ZERO					
	USER TIME (S)	WALL CLOCK TIME (S)	USER TIME (S)	WALL CLOCK TIME (S)	USER TIME (S)	WALL CLOCK TIME (S)	BT, METHODS (S)	BT, HOST (S)	ADJUSTED USER TIME (S)	ADJUSTED WALL CLOCK TIME (S)
19	15.08	15.51	33.06	2.41	532.8	48.86	0.1	31.11	501.59	17.65
19	15.08	15.51	32.7	2.41	530.37	48.58	0.02	30.92	499.43	17.64
19	14.95	15.4	33.38	2.42	532.09	48.6	0.01	30.91	501.17	17.68
19	15.12	15.56	32.88	2.4	533.26	48.34	0.01	30.69	502.56	17.64
19	14.97	15.39	33.38	2.42	530.93	48.24	0.01	30.66	500.26	17.57
19	15.01	15.42	34.42	2.44	532.19	48.37	0.01	30.77	501.41	17.59
19	15.08	15.53	32.34	2.43	532.7	48.31	0.01	30.63	502.06	17.67
19	14.94	15.38	32.15	2.4	532.17	48.31	0.01	30.67	501.49	17.63
19	15.04	15.47	33.31	2.42	531.04	48.16	0.02	30.6	500.42	17.54
19	14.91	15.33	33.09	2.44	531.62	48.8	0.01	31.13	500.48	17.66
Mean	15.018	15.45	33.071	2.419					501.087	17.627
Median	15.025	15.445	33.075	2.42					501.29	17.64
Std Dev	0.07238784428	0.07630348762	0.6346556372	0.01449137675					0.9360680175	0.0457165178
Valida Advantage	33.36576109	1.140906149	15.1518551	7.286895411						

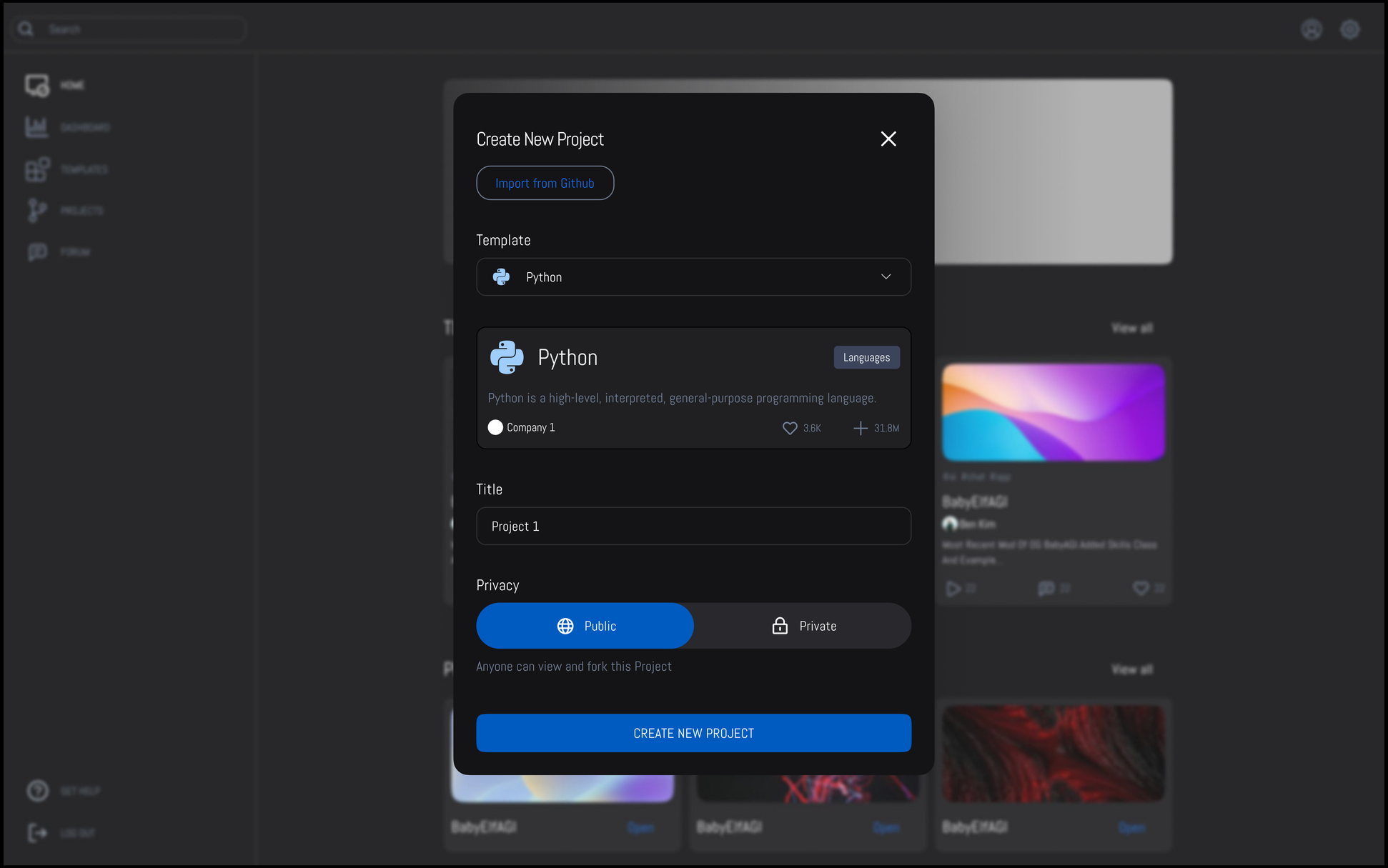
Ongoing Work

- Implement and benchmark multivariate SNARKs to accommodate more efficient chips (with fewer but more expressive gates)
- Include tailored, efficient cryptographic gadgets using new techniques such as Basefold
- Develop and integrate a proper vectorized/SIMD implementation of Keccak and/or blake3 to enhance the speed of hash computations, especially for handling many small inputs efficiently.
- Conduct optimizations on the FRI prover, focusing on vectorization and addressing non-vectorized segments to reduce the significant computational cost associated with it.
- Investigate alternative memory arguments to handle the long, narrow memory trace more efficiently in Plonky3, ensuring improved performance for such scenarios.jj
- Circle STARKs...
- EVM verifier...
- Hardware acceleration

Developer interface

Lita Pro: An all-in-one development, proving and deployment platform


Welcome Ventali!
Create New Project
Import from Github
Template
Search
Title
Enter title
Privacy
Public
Private
Anyone can view and fork this Project
CREATE NEW PROJECT


Create New Project
Import from Github
Template
Python
Python
Languages
Python is a high-level, interpreted, general-purpose programming language.
Company 1
3.6K
31.8M
Title
Project 1
Privacy
Public
Private
Anyone can view and fork this Project
CREATE NEW PROJECT

Search

HOME
DASHBOARD
TEMPLATES
PROJECTS
FORUM

GET HELP
LOG OUT

TEMPLATES

Search

FILTER

#gaming #app
Gaming - Chess
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class And Example...
22 22 22

#account #chat #app
Account System
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class And Example...
22 22 22

#app #rollup #sdk
App Rollup SDK
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class And Example...
22 22 22

#dex #orderbook #app
CLOB DEX
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class And Example...
22 22 22

#rpc #lightclients #app
Light Clients - Ethereum
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class And Example...
22 22 22

#move #app
Move
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class And Example...
22 22 22

#gaming #mmo #app
Gaming - MMO
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class

#gaming #rts #app
Gaming - RTS
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class

#finance #app
Proof Of Reserve
Ben Kim
Most Recent Mod Of OG BabyAGI.Added Skills Class

ProjectSlushy111
Ventali

EXPLORER

- NO FOLDER OPENED
- OUTLINE
  - version
  - [ ] configurations
    - { } 0
      - name.rs
      - type.rs
      - request.rs
      - runtimeExecutable.rs
      - [ ] args
        - 0.toml
      - [ ] out Files
        - 0.toml
      - preLaunchTask
    - { } 1
      - name
      - type
      - request
      - runtimeExecutable
      - [ ] args
        - 0.toml
      - [ ] out Files
        - 0.toml
      - preLaunchTask
- TIMELINE
- TOOLS

main.py | poetry.lock

```rust
#[derive(Machine, Default)]
pub struct BasicMachine {
    // Core instructions
    #[instruction]
    load32: Load32Instruction,
    #[instruction]
    store32: Store32Instruction,
    #[instruction]
    jal: JalInstruction,
    #[instruction]
    jalv: JalvInstruction,
    #[instruction]
    beq: BeqInstruction,
    #[instruction]
    bne: BneInstruction,
    #[instruction]
    imm32: Imm32Instruction,

    // ALU instructions
    #[instruction(add_u32)]
    add32: Add32Instruction,
    #[instruction(mul_u32)]
    mul32: Mul32Instruction,
```

Console | Proving | Deployments

Valida zkVM

40 per day

0.5 vCPU / 2 GiB RAM

- ✓ Deploy to any URL, cover page optional
- ✓ 20 GiB egress
- ✓ Configurable machine
- ✓ Always on

US$0.40/day (Approximately US$12.80/month).

Proof Aggregation

40 per day

0.5 vCPU / 2 GiB RAM

- universal aggregation layer
- custom aggregator
- Custom gas-optimized aggregator (with AI prediction tool)

US$0.40/day (Approximately US$12.80/month).

Network

40 per day

0.5 vCPU / 2 GiB RAM

- Polygon Mainnet
- Arbitrum
- Solana

US$0.40/day (Approximately US$12.80/month).

PURCHASE AND DEPLOY

Valida: extensible, performant and developer friendly zk-VM

We launched alpha release today!

- Free and early access
- Help design and validate product ideas

Join us!

- Innovation Team (product engineers)
- Cryptography Engineer
- LLVM Compiler Engineer
- Open Source Contributors (mentorship from the best people!)

Follow us!

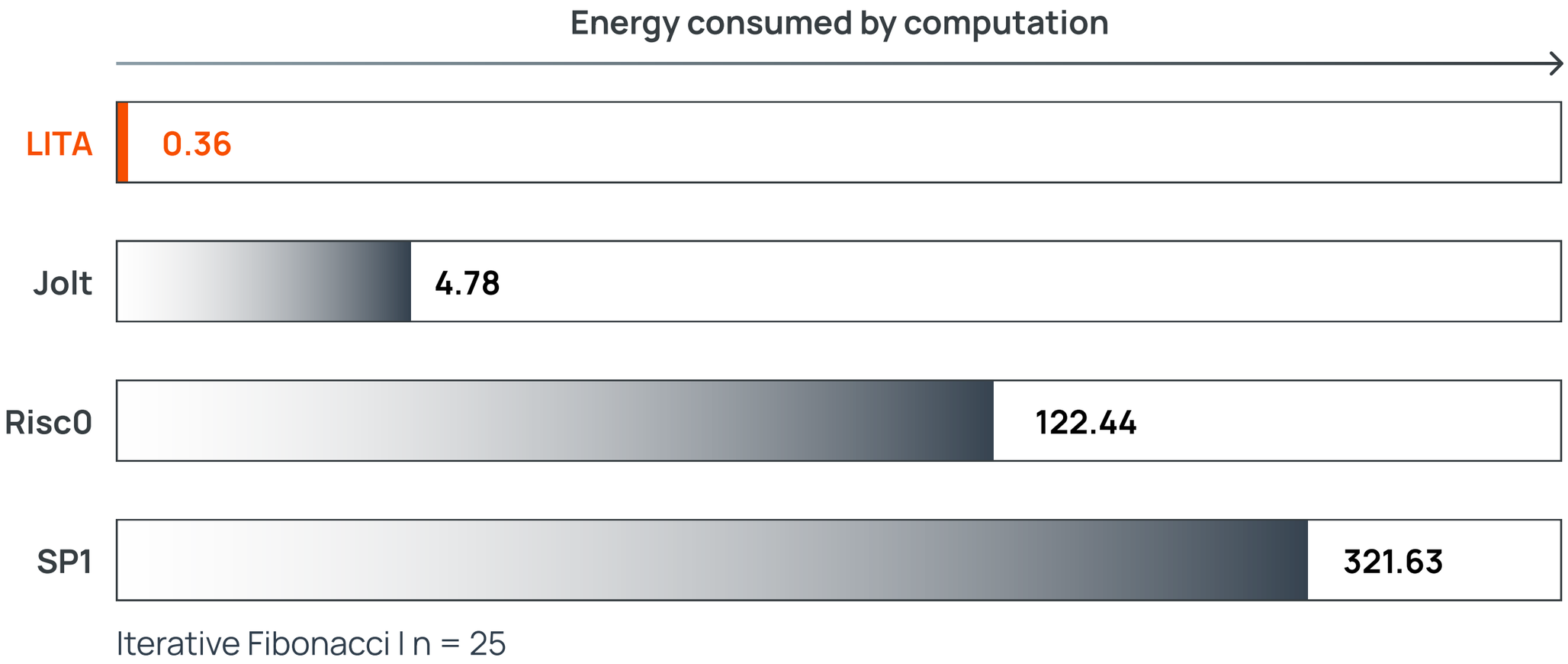
- Twitter: @lita_xyz
- Telegram: @ventali08, @tesseract_ion
- website: www.lita.foundation

Introducing Lita's

Valida zk-VM + C-Compiler

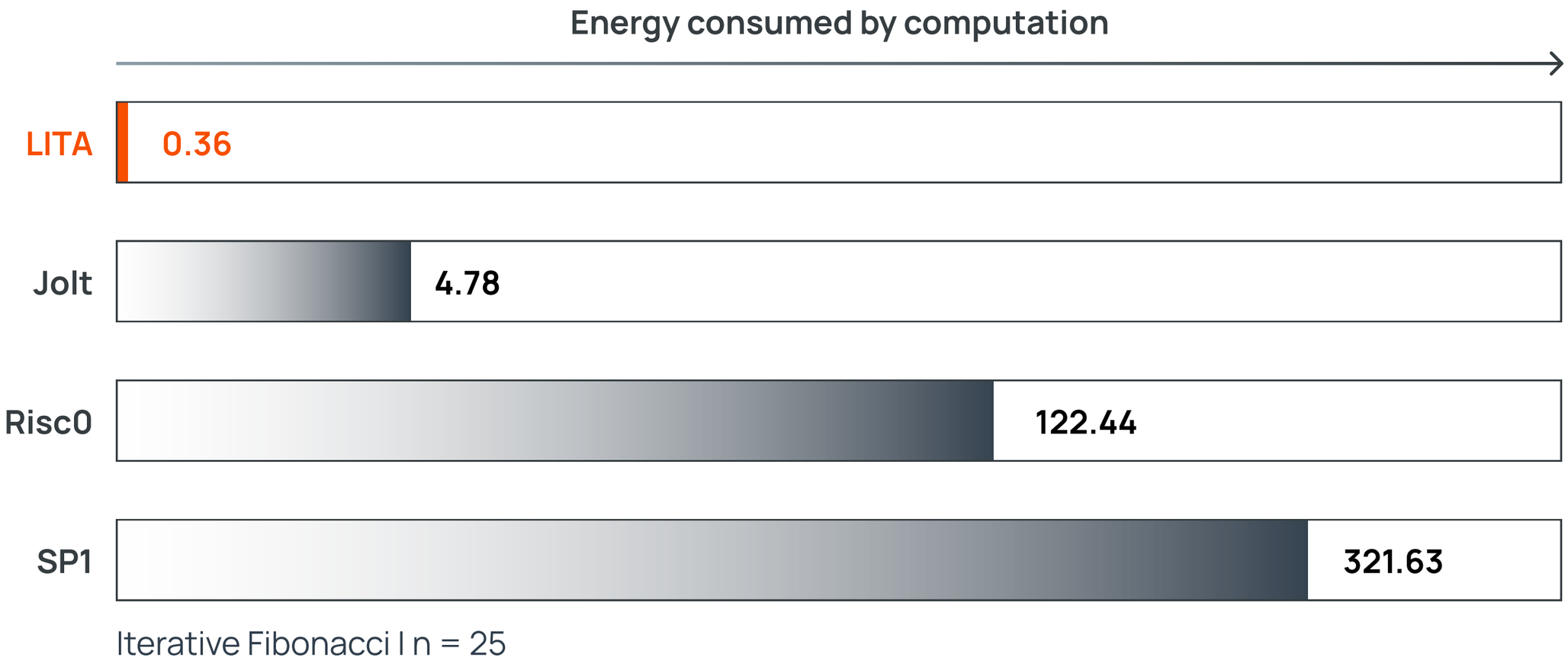
// First step towards **a Zero Knowledge Paradigm**

Valida Maximizes CPU Efficiency



User Time: The amount of computer time expended by the program code. For instance, if a program uses 5 cores for 5 seconds each, the user time is 25 seconds (5 cores x 5 seconds). User time is a crucial indicator of the energy consumed by the computation.

Valida Maximizes CPU Efficiency



User Time: The amount of computer time expended by the program code. For instance, if a program uses 5 cores for 5 seconds each, the user time is 25 seconds (5 cores x 5 seconds). User time is a crucial indicator of the energy consumed by the computation.

Valida offers Industry Leading Speed

Proof Time

Iterative Fibonacci | n = 25

10-20x

Iterative Fibonacci | n = 46

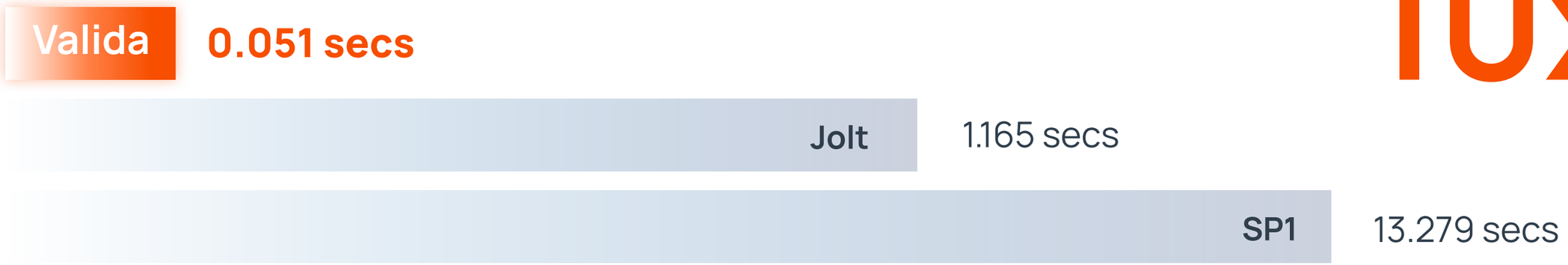
50x

Speed is measured in Wall Clock Time: this measures the actual elapsed time from the start to the end of the computation. In the same example, the wall clock time would be 5 seconds, regardless of the number of cores used. Wall clock time reflects the waiting time for users.

Valida offers Industry Leading Speed

Proof Time

SHA256



10x

SHA256-based Merkle Opening Proof

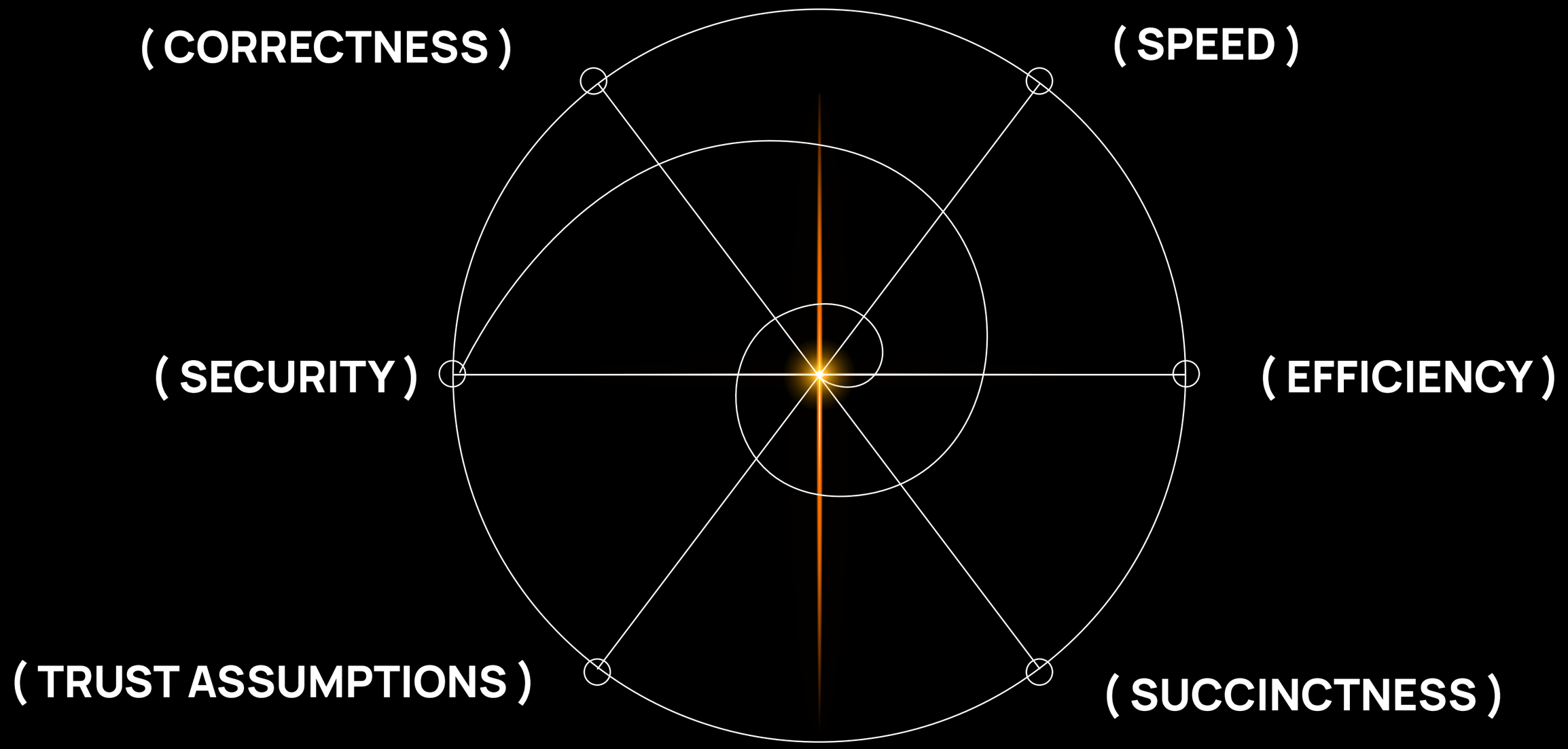
50x

Speed is measured in Wall Clock Time: this measures the actual elapsed time from the start to the end of the computation. In the same example, the wall clock time would be 5 seconds, regardless of the number of cores used. Wall clock time reflects the waiting time for users.

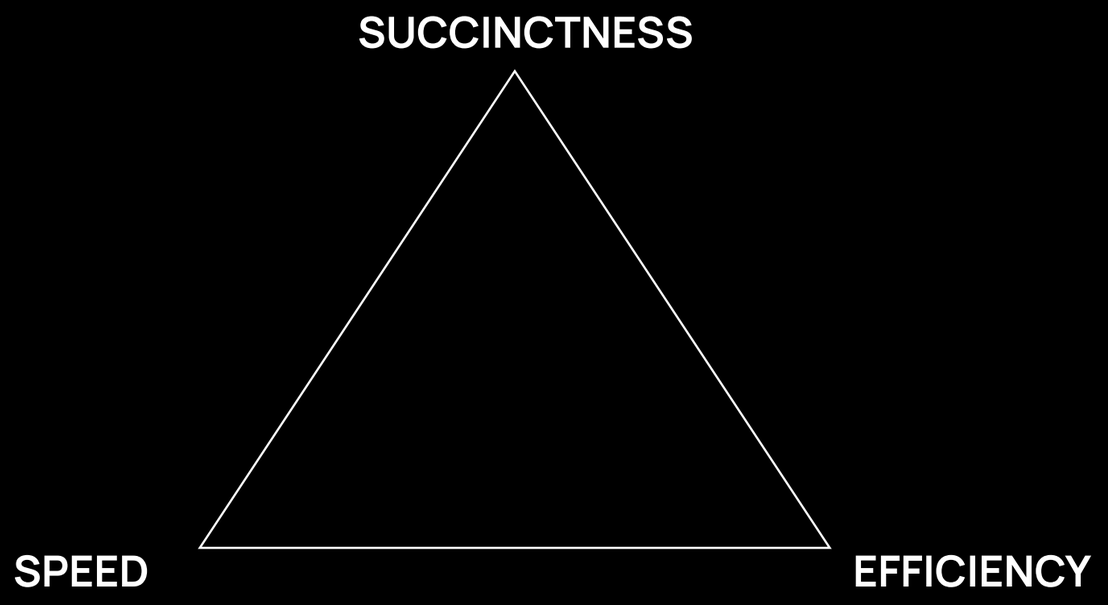
A Zero Knowledge Paradigm

Intro to zkVMs

Part 1


(CORRECTNESS)
(SPEED)
(SECURITY)
(EFFICIENCY)
(TRUST ASSUMPTIONS)
(SUCCINCTNESS)

zkVM Trilemma



speed, efficiency, and succinctness
are all sliding scale properties of a zkVM

System	Proof System	ISA (VM)	Compiler	Programming Language(s)
Starknet	STARKs	Cairo	Proprietary	Cairo
Lita	STARKs	Valida	LLVM	C, C++, Rust, Solidity
Risc Zero	STARKs	RISC-V	Rust Compiler	Rust
Succinct Labs	STARKs	RISC-V / Valida	Rust Pre-compiles	Rust
NEXUS	STARKs	NVM (Nexus Virtual Machine)	Rust	Rust
Polygon Miden	STARKs	Miden	Rust	Rust