Ligetron and the Llama Inference

Muthu Venkitasubramaniam
CEO, Ligero Inc.
Georgetown University

Joint work with Ruihan Wang (Ligero Inc.) and Carmit Hazay (Ligero Inc., Bar-Ilan University)

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution Statement A. Approved for public release. Distribution Unlimited.

Thanks
Stealth Software Tech.

ZK evolution

- **At the Beginning:** Reduce to an NP-complete problem (eg, Graph Hamiltonicity, Graph 3-Coloring)
- **Then:** Boolean and Arithmetic circuits
- **Since Blockchains:** R1CS, Circom, Cairo, Gnark, etc...
- **Today:** zkVMs

Instrumenting ZK today - zkVMs

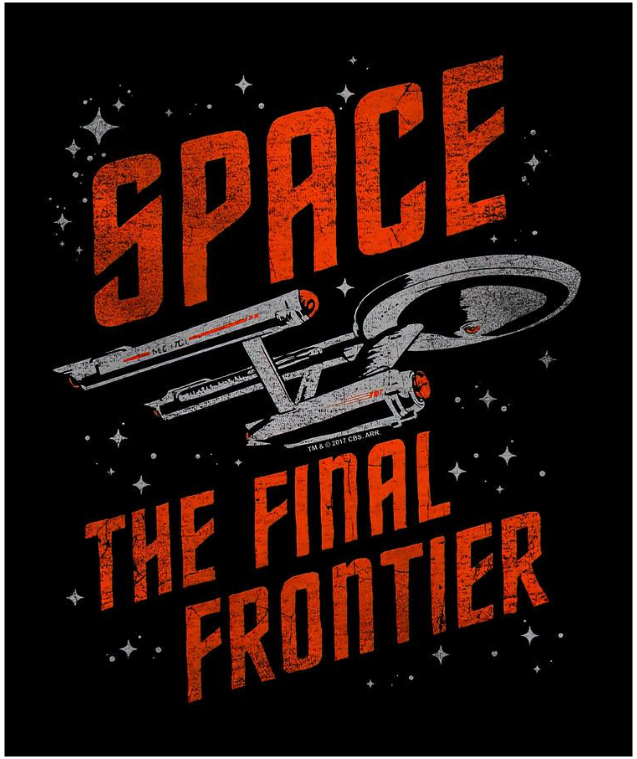
1. Code application in a high-level language (C,C++,Rust)
2. Compile down to popular VMs. Eg, RISC-V, WASM
3. Prove correct execution of VM

Challenges in Scaling ZK

First, it was (circuit) representation

Then, optimizing prover running time

Today, optimize prover memory



Main Question

Do there exist time and space preserving ZK-SNARKs from minimal assumptions?

YES* - Based on hash functions [BBHV 2022]

Prover time $\tilde{O}(T(n))$ and space $\tilde{O}(S(n))$. Proof Length $\tilde{O}(T(n)/S(n))$

Today: Concretely Efficient!

Introducing Ligetron

A Time and Space Efficient ZK-SNARK

Key Ingredients

1. WASM as an intermediate representation – **Ligetron is a zkWASM**
2. A **space-efficient** variant of the Ligero ZKP [Ames et al, 2017]

Ligetron

Post quantum

Black-box

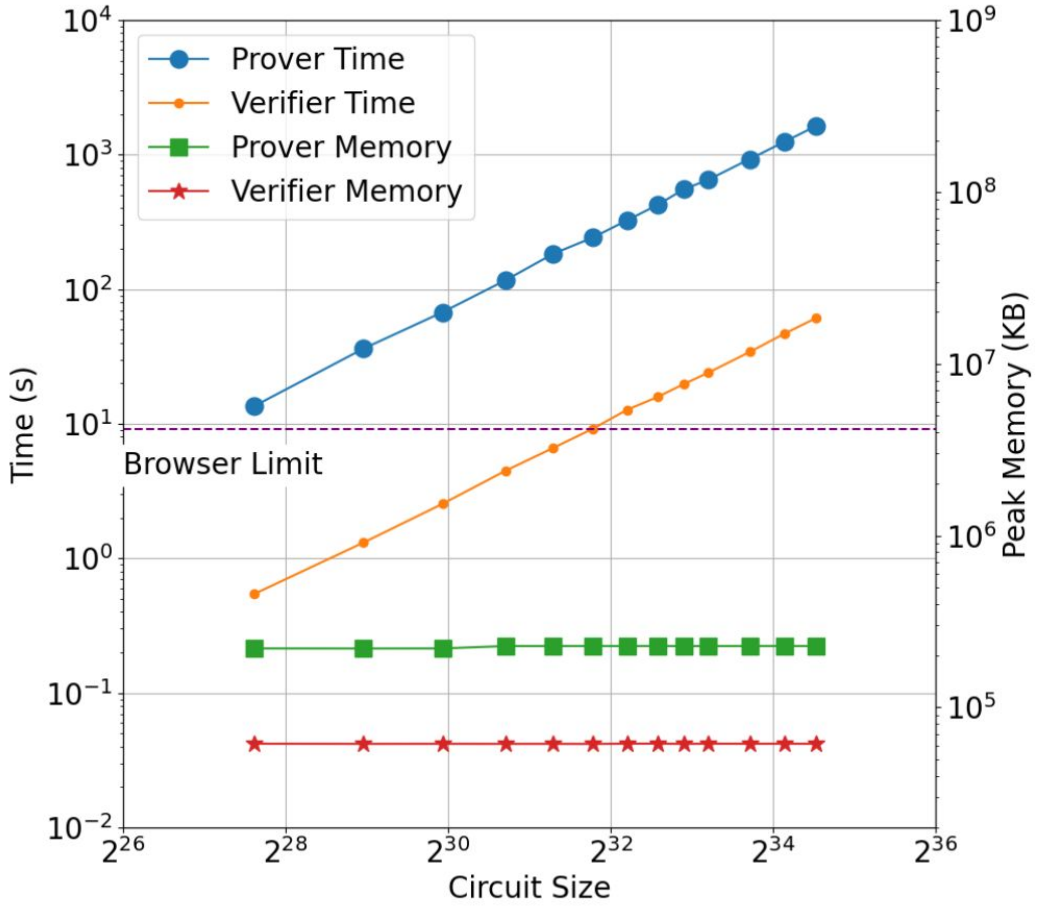
Non-interactive	Yes
Succinct proof	Sublinear
Succinct verification	No*
Time efficient	Yes (quasilinear)
Space efficient	Yes (linear)
Implementable	Yes (runs from a browser)

Why WASM?

- Space/time efficiency from operational semantics of WASM (stacks, blocks)
- Few set of instructions to compile
- Numerous compilers from high-level languages to WASM (C,C++,Rust,etc)
- Clean sandbox (no I/Os, no system calls, no malloc*)
- Extern function calls for gadgets / pre-compiles

Performance

- Prover time: 500 ns/g
- Verifier time: 250 ns/g
- Prover memory: < 100 MB
- Verifier memory: <10 MB



Performance

	Prover Time Batched	Verifier Time Batched	Prover Time Random	Verifier Time Random
Browser	1.2 us/g	20 ns/g	5 us/g	2.5 us/g
Macbook	65 ns/g	3 ns/g	500 ns/g	250 ns/g

Where we are?

Proof Length

- **Unstructured circuit:** Square root circuit size. But, parameterizable
- **Structured circuit (M copies of size T circuit):** O(M+k.T)

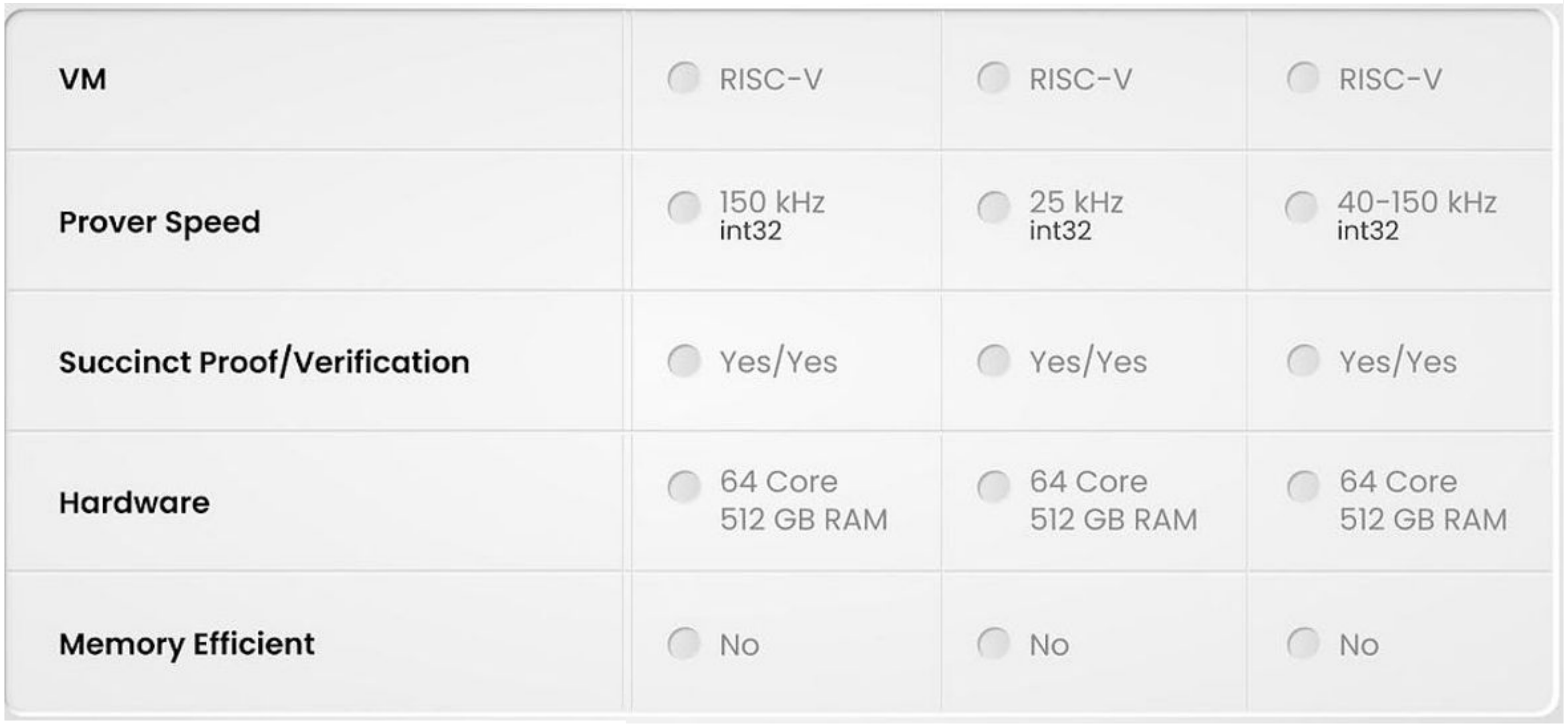
Verification

- **Unstructured circuit:** Quasilinear in circuit size
- **Structured circuit:** O(M+k.T)

Extent of WASM integration

- All 32-bit and 64-bit integer operations
- Oblivious control flow
- More recently, RAM, 32-bit floating operations

Are ZKVMs Scalable?



	Jolt	Risc0	SP1
VM	RISC-V	RISC-V	RISC-V
Prover Speed	150 kHz int32	25 kHz int32	40-150 kHz int32
Succinct Proof/Verification	Yes/Yes	Yes/Yes	Yes/Yes
Hardware	64 Core 512 GB RAM	64 Core 512 GB RAM	64 Core 512 GB RAM
Memory Efficient	No	No	No

Are ZKVMs Scalable?

Current Techniques for Memory Efficiency

- Recursive Composition
- Incrementally Verifiable Computation (IVC)

1. Break computation into small units
2. Prove units in distributed manner
3. Compose

Bottlenecks

- Need a lot of hardware (each unit needs to be run in parallel)
- Gadgets/pre-compiles are harder in RISC-V
- zkVMs today are really succinctVMs

Benchmark 1 (Structured Circuits): Rollups

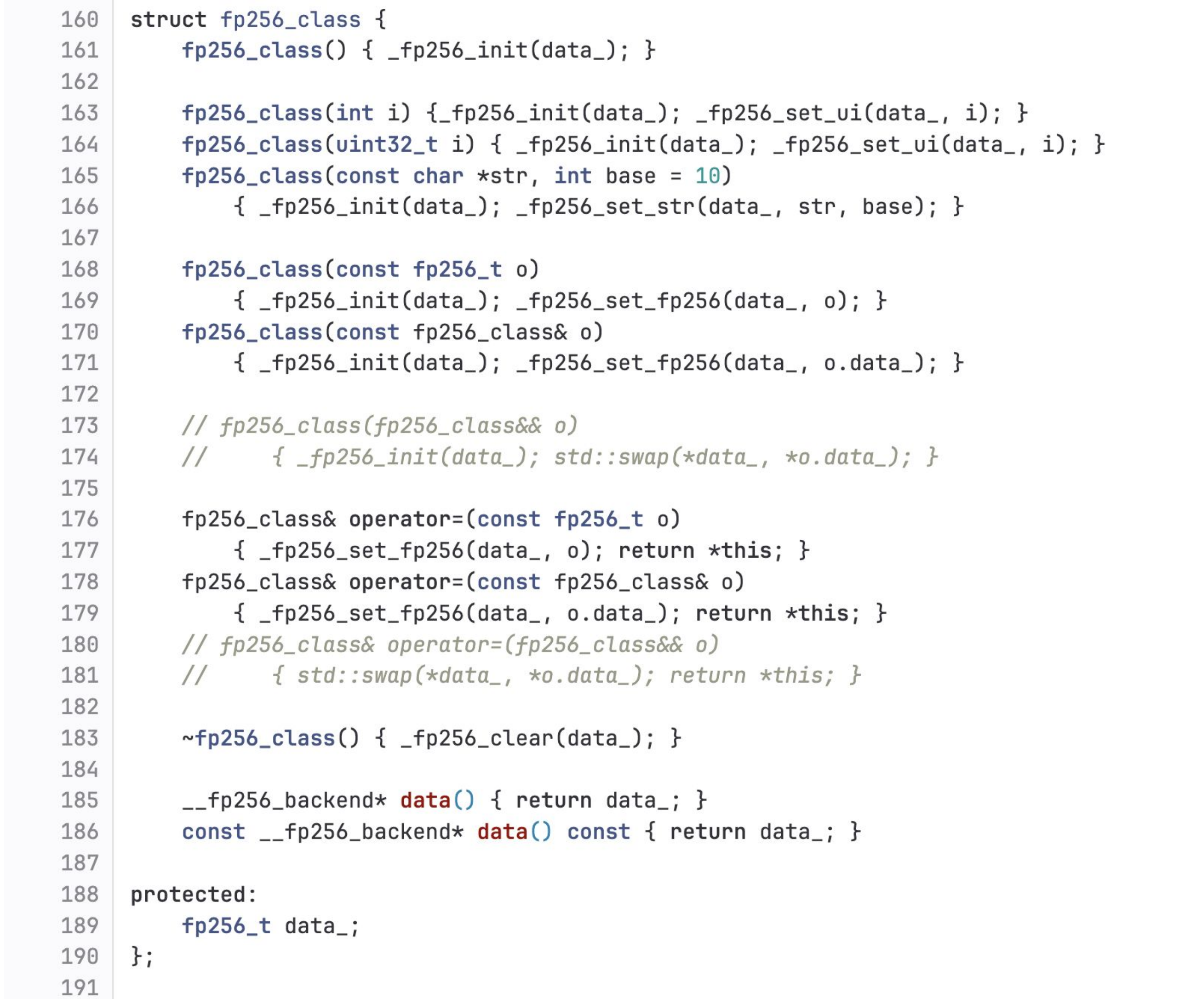
Simple payment circuit

- EdDSA signatures
- Poseidon hash function for Merkle Trees

Starknet: 200 nodes with 400MB RAM
Benchmark: 2000 tps*

Ligetron: g5.xlarge (single A10 GPU)
Benchmark: 500tps

WASM extern for 256-bit

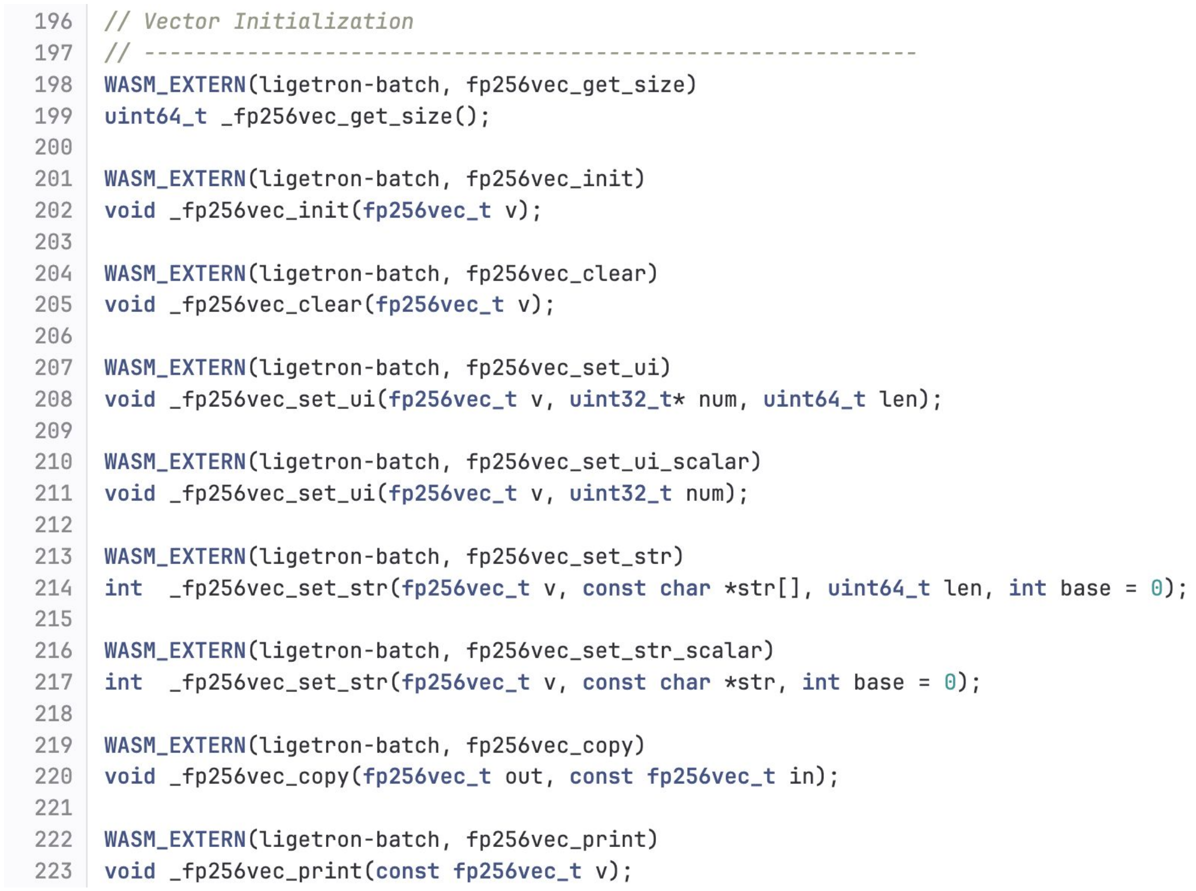


```cpp
struct fp256_class {
    fp256_class() { _fp256_init(data_); }

    fp256_class(int i) {_fp256_init(data_); _fp256_set_ui(data_, i); }
    fp256_class(uint32_t i) { _fp256_init(data_); _fp256_set_ui(data_, i); }
    fp256_class(const char *str, int base = 10)
        { _fp256_init(data_); _fp256_set_str(data_, str, base); }

    fp256_class(const fp256_t o)
        { _fp256_init(data_); _fp256_set_fp256(data_, o); }
    fp256_class(const fp256_class& o)
        { _fp256_init(data_); _fp256_set_fp256(data_, o.data_); }

    // fp256_class(fp256_class&& o)
    //     { _fp256_init(data_); std::swap(*data_, *o.data_); }

    fp256_class& operator=(const fp256_t o)
        { _fp256_set_fp256(data_, o); return *this; }
    fp256_class& operator=(const fp256_class& o)
        { _fp256_set_fp256(data_, o.data_); return *this; }
    // fp256_class& operator=(fp256_class&& o)
    //     { std::swap(*data_, *o.data_); return *this; }

    ~fp256_class() { _fp256_clear(data_); }

    __fp256_backend* data() { return data_; }
    const __fp256_backend* data() const { return data_; }

protected:
    fp256_t data_;
};
```

WASM extern for 256-bit

Implementations:

- EdDSA signatures
- Poseidon hash function



```
// Vector Initialization
// -----------------------------------------------------------
WASM_EXTERN(ligetron-batch, fp256vec_get_size)
uint64_t _fp256vec_get_size();

WASM_EXTERN(ligetron-batch, fp256vec_init)
void _fp256vec_init(fp256vec_t v);

WASM_EXTERN(ligetron-batch, fp256vec_clear)
void _fp256vec_clear(fp256vec_t v);

WASM_EXTERN(ligetron-batch, fp256vec_set_ui)
void _fp256vec_set_ui(fp256vec_t v, uint32_t* num, uint64_t len);

WASM_EXTERN(ligetron-batch, fp256vec_set_ui_scalar)
void _fp256vec_set_ui(fp256vec_t v, uint32_t num);

WASM_EXTERN(ligetron-batch, fp256vec_set_str)
int  _fp256vec_set_str(fp256vec_t v, const char *str[], uint64_t len, int base = 0);

WASM_EXTERN(ligetron-batch, fp256vec_set_str_scalar)
int  _fp256vec_set_str(fp256vec_t v, const char *str, int base = 0);

WASM_EXTERN(ligetron-batch, fp256vec_copy)
void _fp256vec_copy(fp256vec_t out, const fp256vec_t in);

WASM_EXTERN(ligetron-batch, fp256vec_print)
void _fp256vec_print(const fp256vec_t v);
```

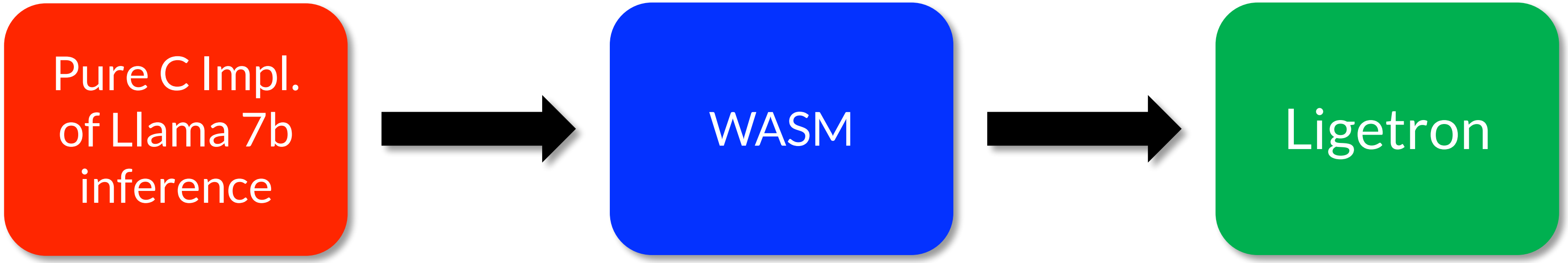
Benchmark 2 (“Unstructured”): LLM Inference

Modulus Labs: 200+ hours for verifying GPT2-XL 1.5B using 128-core 1TB CPU with 10TB disk space

Ligetron: 14 hours for verifying the Llama 7B using 8-core CPU with peak memory 10GB*

*In collaboration with nim

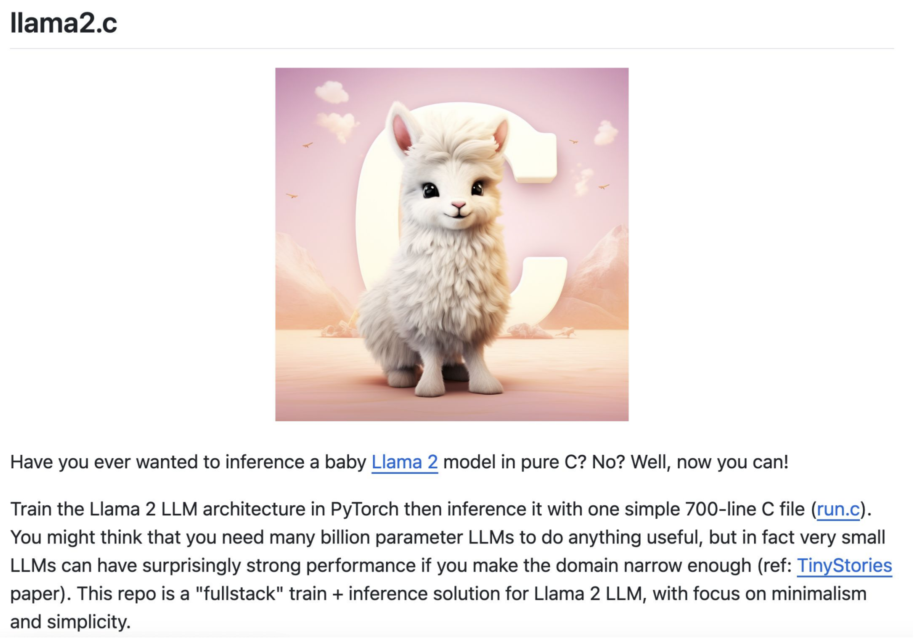
Our Approach for Llama inference



Pure C implementation of Llama 7B

https://github.com/karpathy/llama2.c

Main Challenge
floating point ops

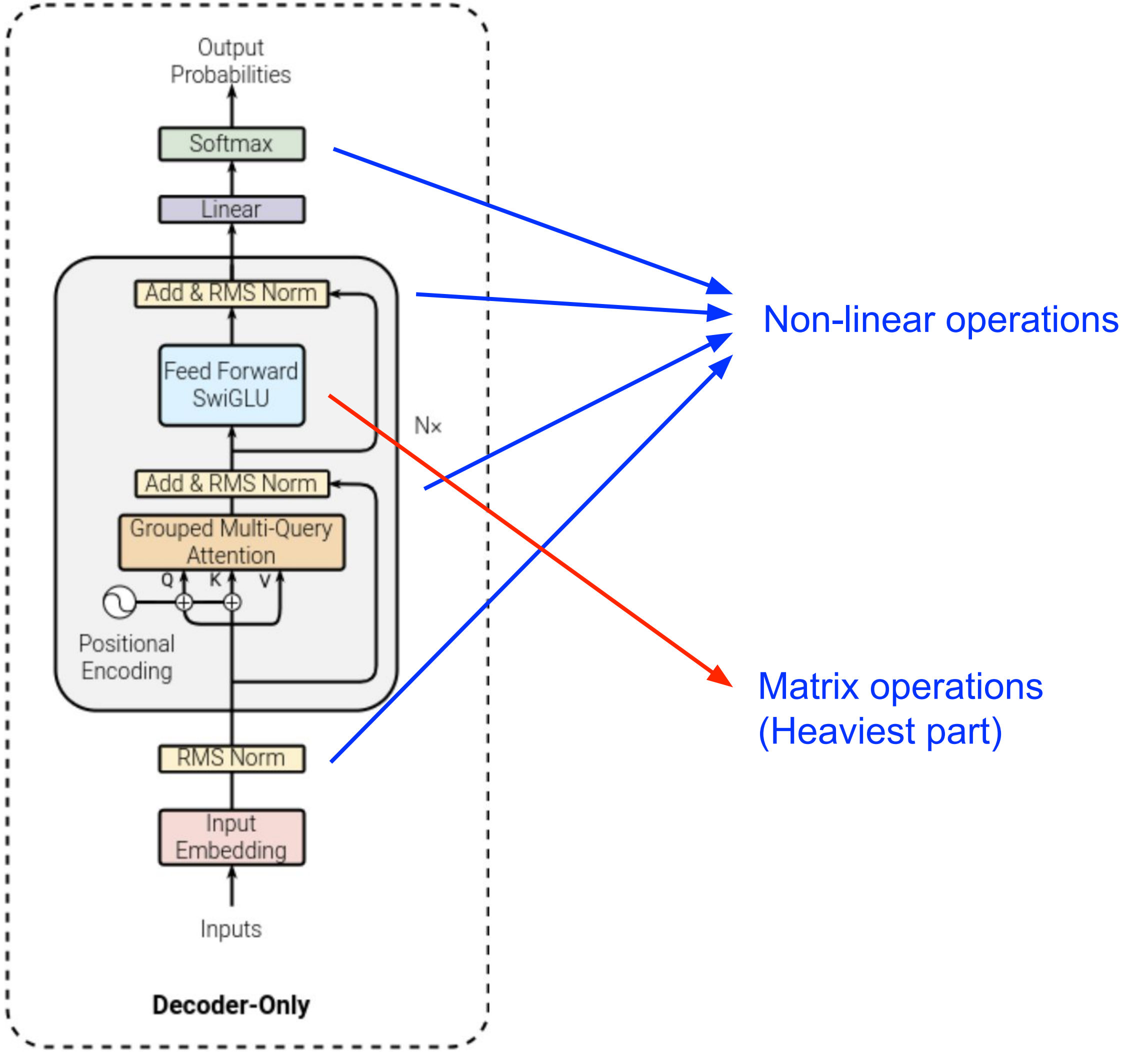


llama2.c

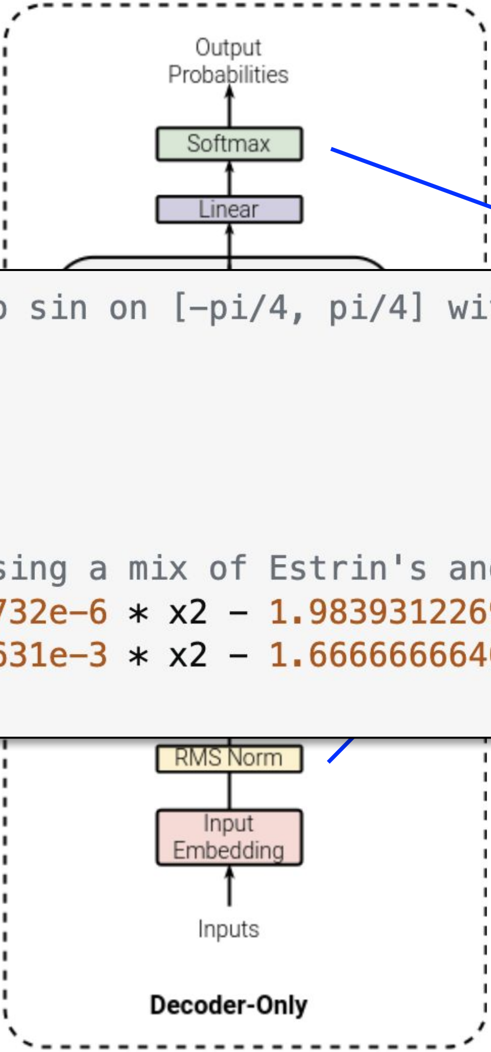
Have you ever wanted to inference a baby Llama 2 model in pure C? No? Well, now you can!

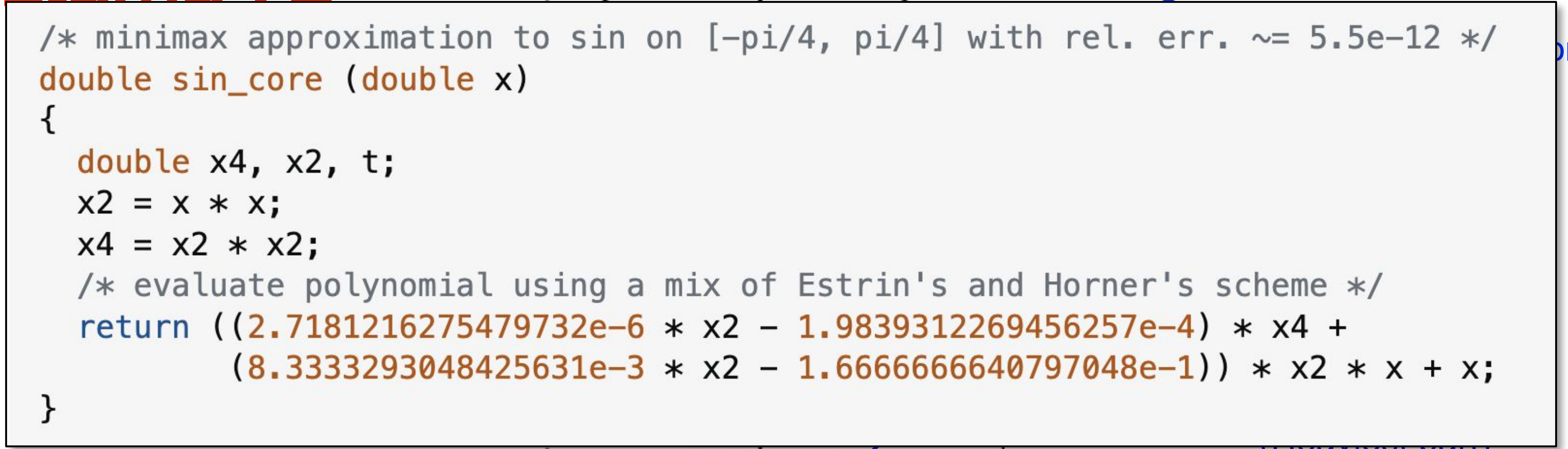
Train the Llama 2 LLM architecture in PyTorch then inference it with one simple 700-line C file (run.c). You might think that you need many billion parameter LLMs to do anything useful, but in fact very small LLMs can have surprisingly strong performance if you make the domain narrow enough (ref: TinyStories paper). This repo is a "fullstack" train + inference solution for Llama 2 LLM, with focus on minimalism and simplicity.

Llama 7B


Output Probabilities
Softmax
Linear
Add & RMS Norm
Feed Forward SwiGLU
Nx
Add & RMS Norm
Grouped Multi-Query Attention
Q
K
V
Positional Encoding
RMS Norm
Input Embedding
Inputs
Decoder-Only
Non-linear operations
Matrix operations (Heaviest part)

Llama 7B





```
/* minimax approximation to sin on [-pi/4, pi/4] with rel. err. ~= 5.5e-12 */
double sin_core (double x)
{
  double x4, x2, t;
  x2 = x * x;
  x4 = x2 * x2;
  /* evaluate polynomial using a mix of Estrin's and Horner's scheme */
  return ((2.7181216275479732e-6 * x2 - 1.9839312269456257e-4) * x4 +
          (8.3333293048425631e-3 * x2 - 1.6666666640797048e-1)) * x2 * x + x;
}
```

(eg, sin, exp, etc)

Llama 7B

Operation distribution

fp32 add (100 million)

~1000 constraints

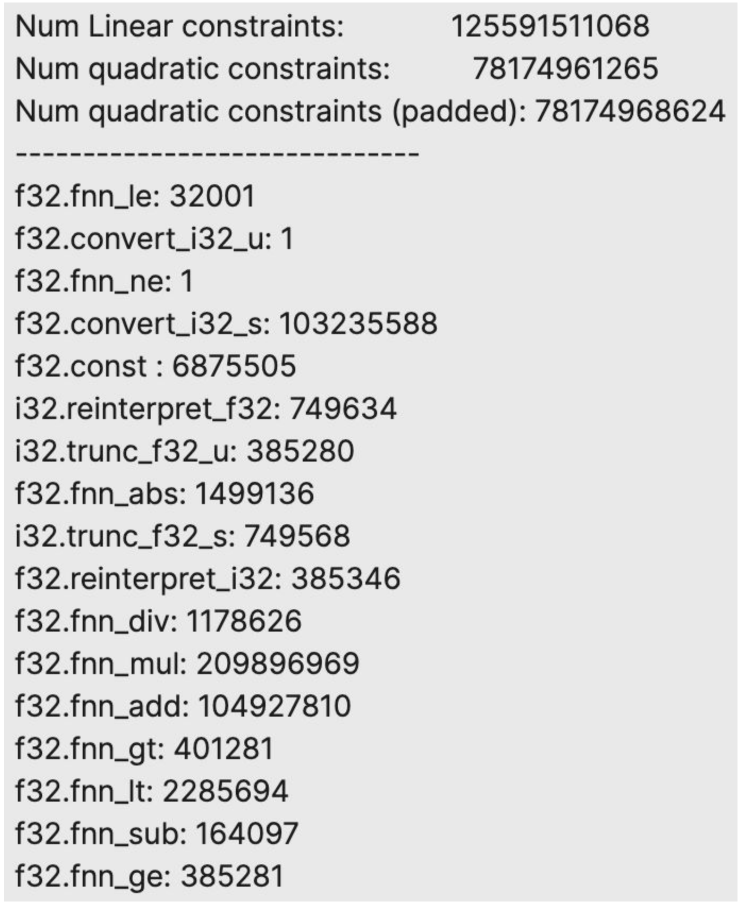
fp32 mul (200 million)

~200 constraints

Total constraints

~130 billion add

~78 billion mul



```
Num Linear constraints:          125591511068
Num quadratic constraints:        78174961265
Num quadratic constraints (padded): 78174968624
-----------------------------
f32.fnn_le: 32001
f32.convert_i32_u: 1
f32.fnn_ne: 1
f32.convert_i32_s: 103235588
f32.const : 6875505
i32.reinterpret_f32: 749634
i32.trunc_f32_u: 385280
f32.fnn_abs: 1499136
i32.trunc_f32_s: 749568
f32.reinterpret_i32: 385346
f32.fnn_div: 1178626
f32.fnn_mul: 209896969
f32.fnn_add: 104927810
f32.fnn_gt: 401281
f32.fnn_lt: 2285694
f32.fnn_sub: 164097
f32.fnn_ge: 385281
```

Benchmark 2 (“Unstructured”): LLM Inf.

Modulus Labs: 200+ hours for verifying GPT2-XL 1.5B using 128-core 1TB CPU with 10TB disk space

Ligetron: 14 hours for verifying the Llama 7B using 8-core CPU with peak mem. 10GB

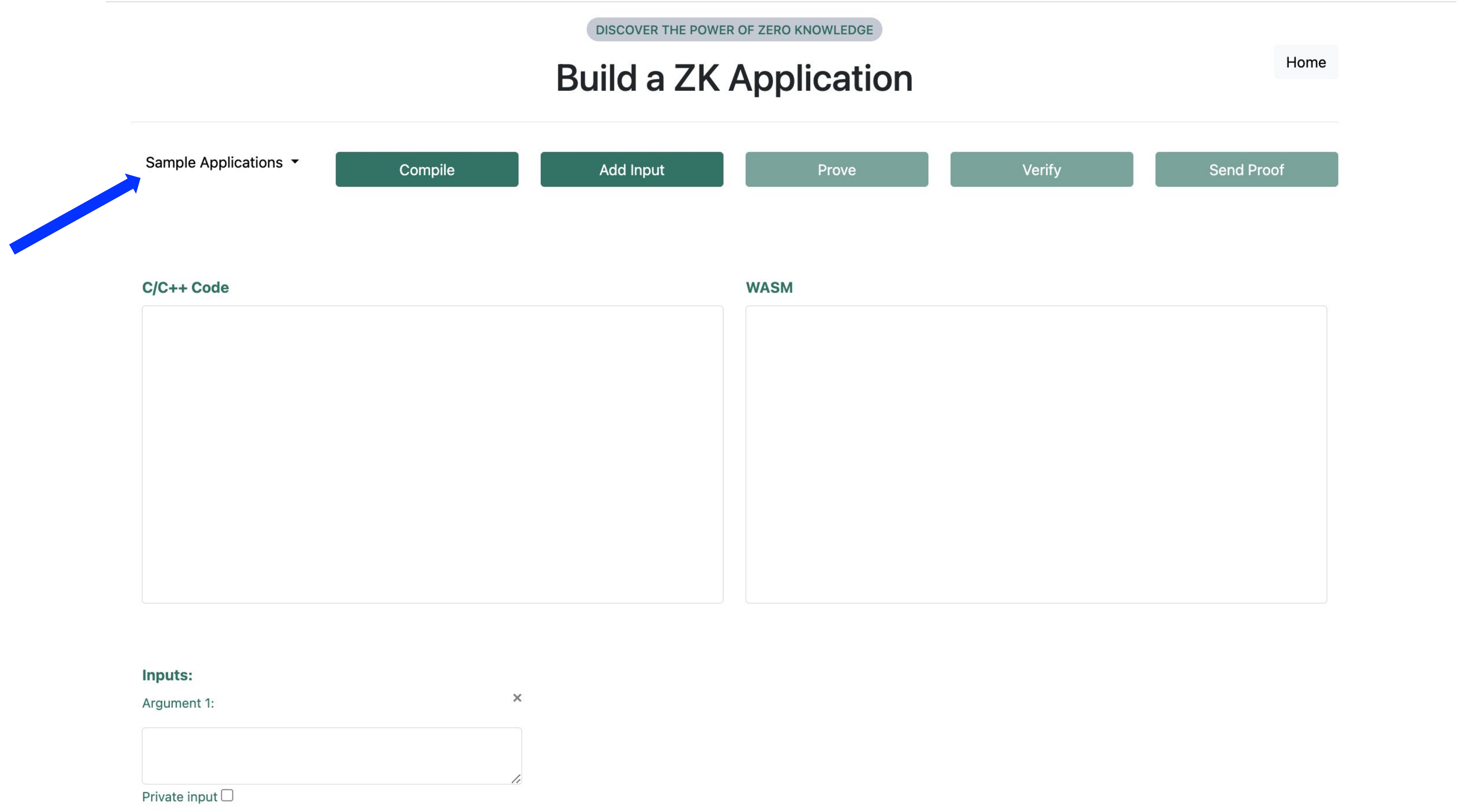
Time to benchmark: 15 days

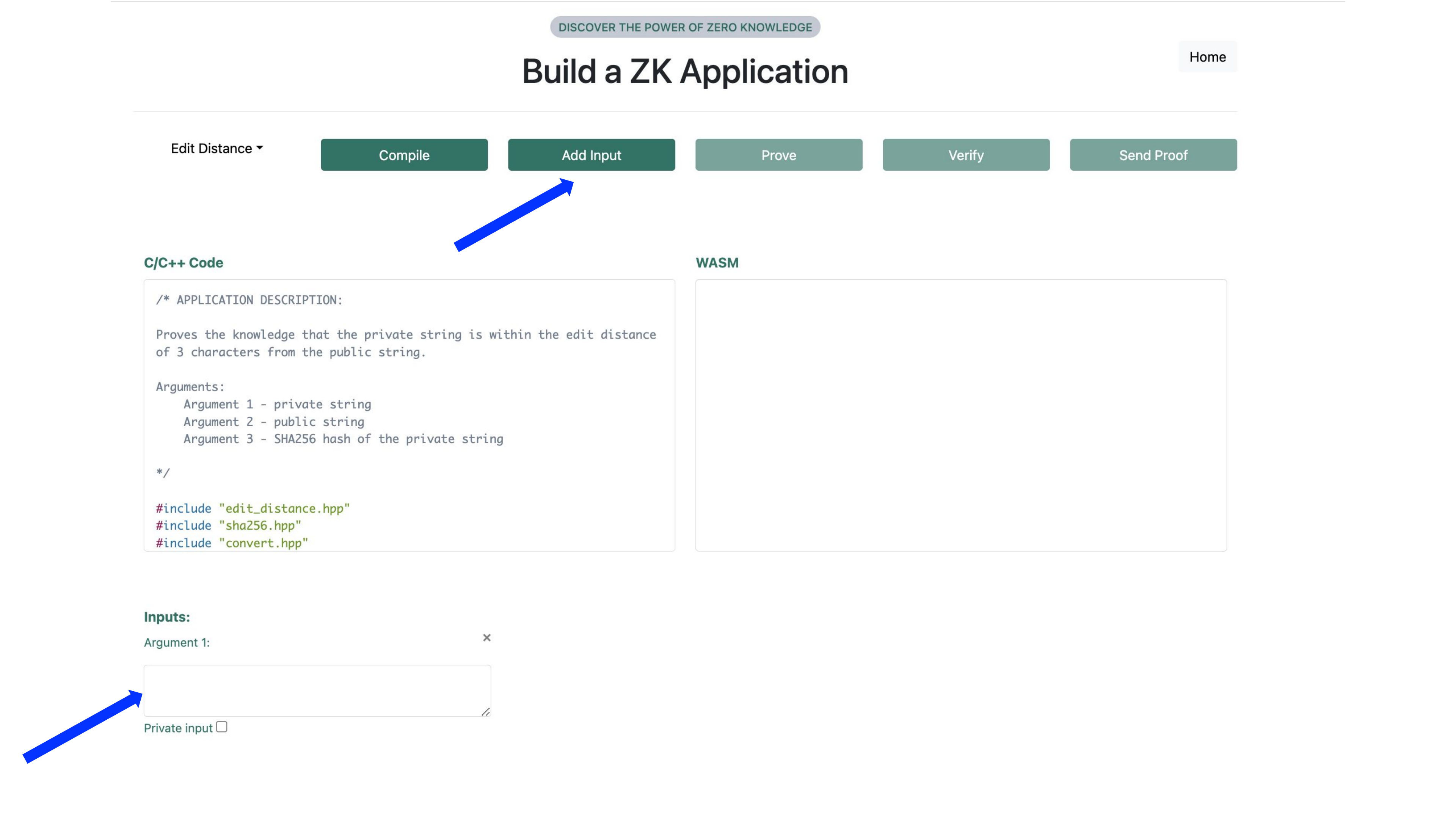
*In collaboration with nim

Ligetron 1.0 Compared to Other zkVMs

	● Ligertron 1.0	○ Jolt	○ Risc0	○ SP1
VM	● WASM	○ RISC-V	○ RISC-V	○ RISC-V
Prover Speed	● 50 kHz int32, int64, fp32	○ 150 kHz int32	○ 25 kHz int32	○ 40-150 kHz int32
Succinct Proof/Verification	● Yes/No	○ Yes/Yes	○ Yes/Yes	○ Yes/Yes
Hardware	● 8 Core 16GM RAM	○ 64 Core 512 GB RAM	○ 64 Core 512 GB RAM	○ 64 Core 512 GB RAM
Memory Efficient	● Yes	○ No	○ No	○ No

Demo


DISCOVER THE POWER OF ZERO KNOWLEDGE
Build a ZK Application
Home
Sample Applications
Compile
Add Input
Prove
Verify
Send Proof
C/C++ Code
WASM
Inputs:
Argument 1:
Private input



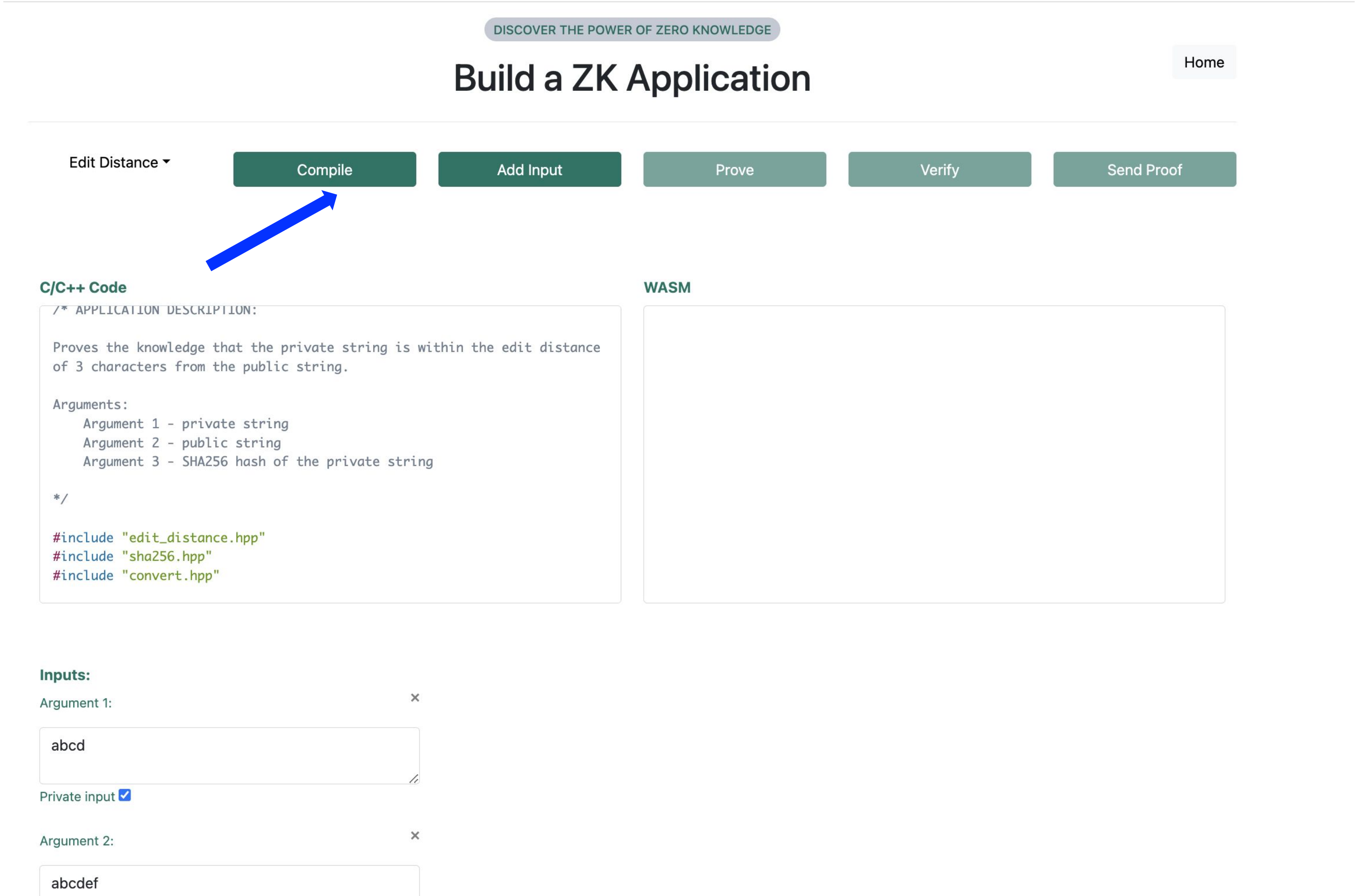
Build a ZK Application

C/C++ Code

```
/* APPLICATION DESCRIPTION:

Proves the knowledge that the private string is within the edit distance
of 3 characters from the public string.

Arguments:
    Argument 1 - private string
    Argument 2 - public string
    Argument 3 - SHA256 hash of the private string

*/

#include "edit_distance.hpp"
#include "sha256.hpp"
#include "convert.hpp"
```

Inputs:

Argument 1:



Build a ZK Application

C/C++ Code

```
/* APPLICATION DESCRIPTION:

Proves the knowledge that the private string is within the edit distance
of 3 characters from the public string.

Arguments:
    Argument 1 - private string
    Argument 2 - public string
    Argument 3 - SHA256 hash of the private string

*/

#include "edit_distance.hpp"
#include "sha256.hpp"
#include "convert.hpp"
```

WASM

Inputs:

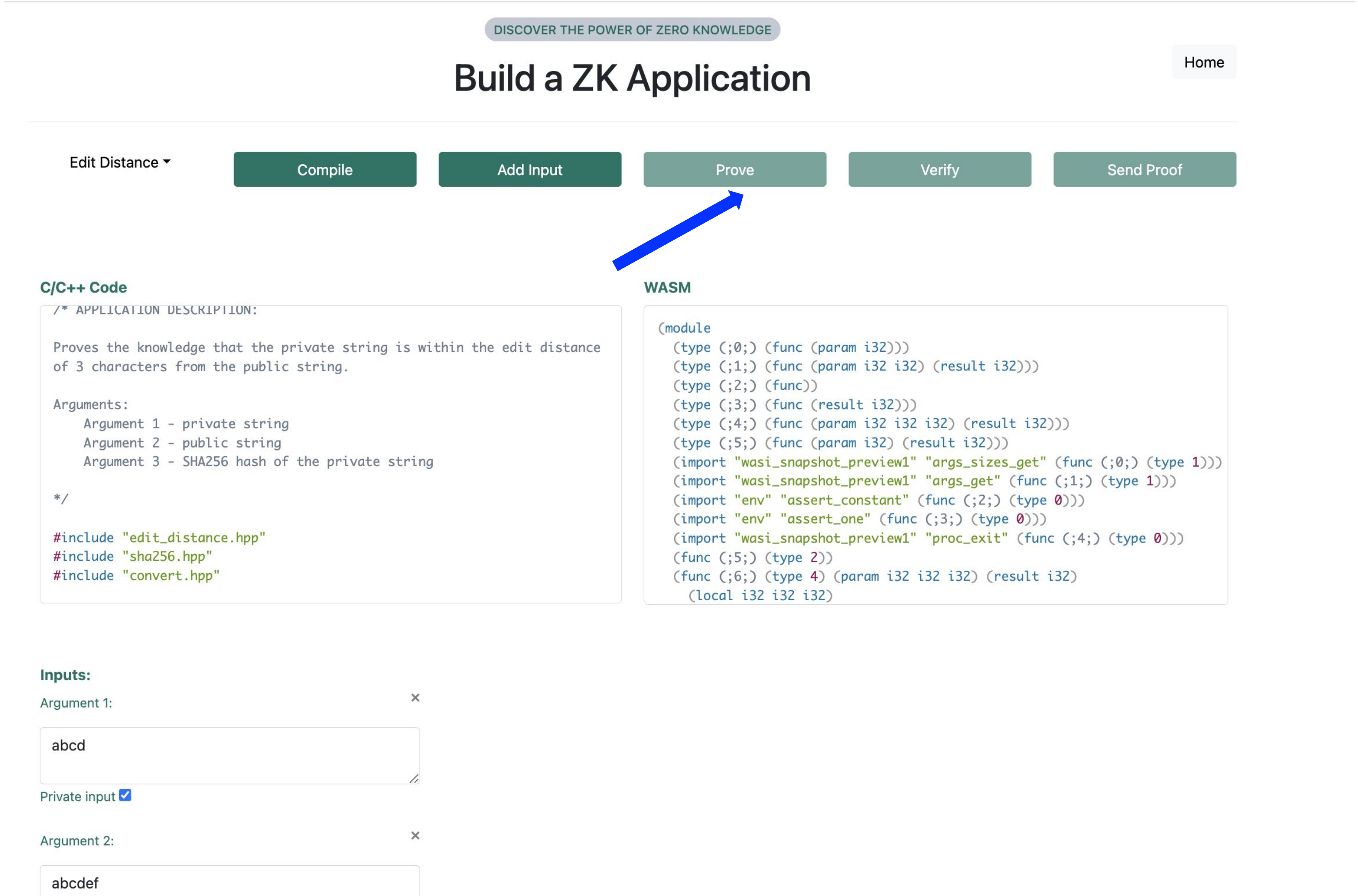
Argument 1:

abcd

Private input ✓

Argument 2:

abcdef



Build a ZK Application

C/C++ Code

```
/* APPLICATION DESCRIPTION:

Proves the knowledge that the private string is within the edit distance
of 3 characters from the public string.

Arguments:
    Argument 1 - private string
    Argument 2 - public string
    Argument 3 - SHA256 hash of the private string

*/

#include "edit_distance.hpp"
#include "sha256.hpp"
#include "convert.hpp"
```

WASM

```
(module
  (type (;0;) (func (param i32)))
  (type (;1;) (func (param i32 i32) (result i32)))
  (type (;2;) (func))
  (type (;3;) (func (result i32)))
  (type (;4;) (func (param i32 i32 i32) (result i32)))
  (type (;5;) (func (param i32) (result i32)))
  (import "wasi_snapshot_preview1" "args_sizes_get" (func (;0;) (type 1)))
  (import "wasi_snapshot_preview1" "args_get" (func (;1;) (type 1)))
  (import "env" "assert_constant" (func (;2;) (type 0)))
  (import "env" "assert_one" (func (;3;) (type 0)))
  (import "wasi_snapshot_preview1" "proc_exit" (func (;4;) (type 0)))
  (func (;5;) (type 2))
  (func (;6;) (type 4) (param i32 i32 i32) (result i32)
    (local i32 i32 i32)
```

Inputs:

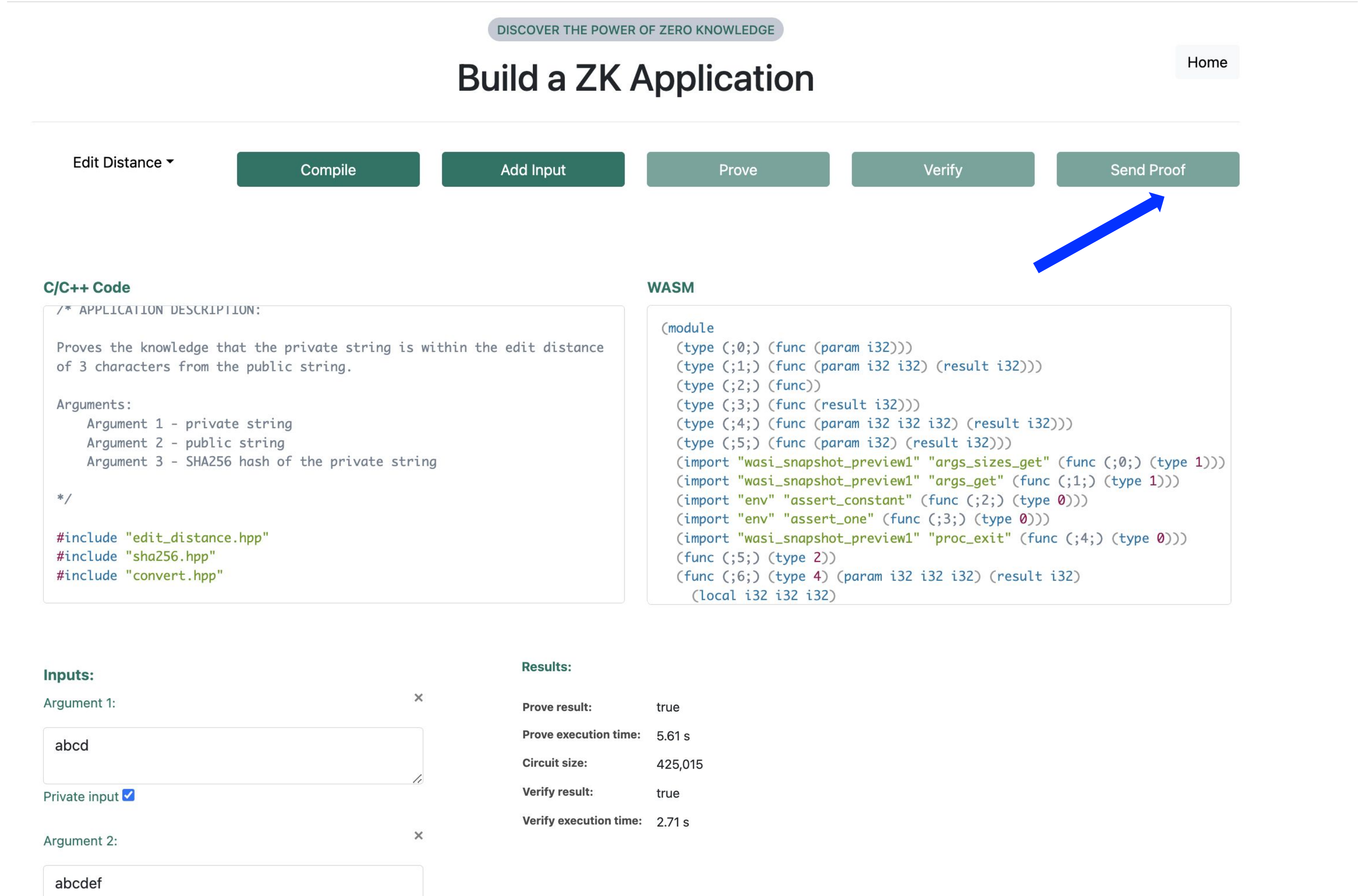
Argument 1:

abcd

Private input ☑

Argument 2:

abcdef



Build a ZK Application

C/C++ Code

```
/* APPLICATION DESCRIPTION:

Proves the knowledge that the private string is within the edit distance
of 3 characters from the public string.

Arguments:
    Argument 1 - private string
    Argument 2 - public string
    Argument 3 - SHA256 hash of the private string

*/

#include "edit_distance.hpp"
#include "sha256.hpp"
#include "convert.hpp"
```

WASM

```
(module
  (type (;0;) (func (param i32)))
  (type (;1;) (func (param i32 i32) (result i32)))
  (type (;2;) (func))
  (type (;3;) (func (result i32)))
  (type (;4;) (func (param i32 i32 i32) (result i32)))
  (type (;5;) (func (param i32) (result i32)))
  (import "wasi_snapshot_preview1" "args_sizes_get" (func (;0;) (type 1)))
  (import "wasi_snapshot_preview1" "args_get" (func (;1;) (type 1)))
  (import "env" "assert_constant" (func (;2;) (type 0)))
  (import "env" "assert_one" (func (;3;) (type 0)))
  (import "wasi_snapshot_preview1" "proc_exit" (func (;4;) (type 0)))
  (func (;5;) (type 2))
  (func (;6;) (type 4) (param i32 i32 i32) (result i32)
    (local i32 i32 i32)
```

Inputs:

Argument 1:

abcd

Private input ✓

Argument 2:

abcdef

Results:

Prove result:	true
Prove execution time:	5.61 s
Circuit size:	425,015
Verify result:	true
Verify execution time:	2.71 s

Our Roadmap

1. You don't have succinct verification, duh!

   Pre-processing Ligetron for succinct verification

2. But your proofs are not short, duh!

   Compose with SNARKs to verify on-chain (Groth16/Halo2)

3. But you can't handle non-oblivious code, duh!

- https://eprint.iacr.org/2023/1257 (CCS 2023 – Distinguished Paper Award)
- https://eprint.iacr.org/2024/456 (Precompiles)

4. What about lookup arguments?

   We can do that as well

Develop on Ligetron:
Build your ZK app at ligetron.com

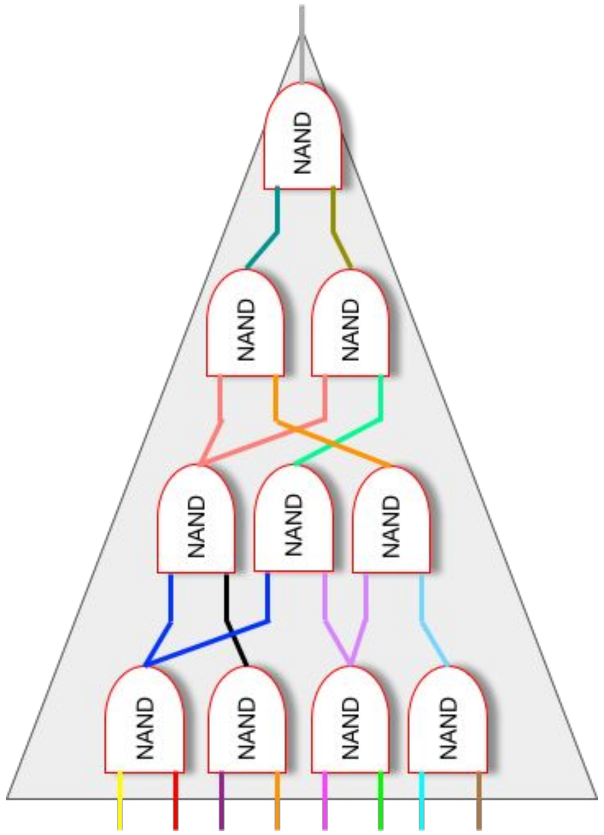
Develop with Ligetron:
We are looking for developers
Please email **muthu@ligero-inc.com**

Ligero

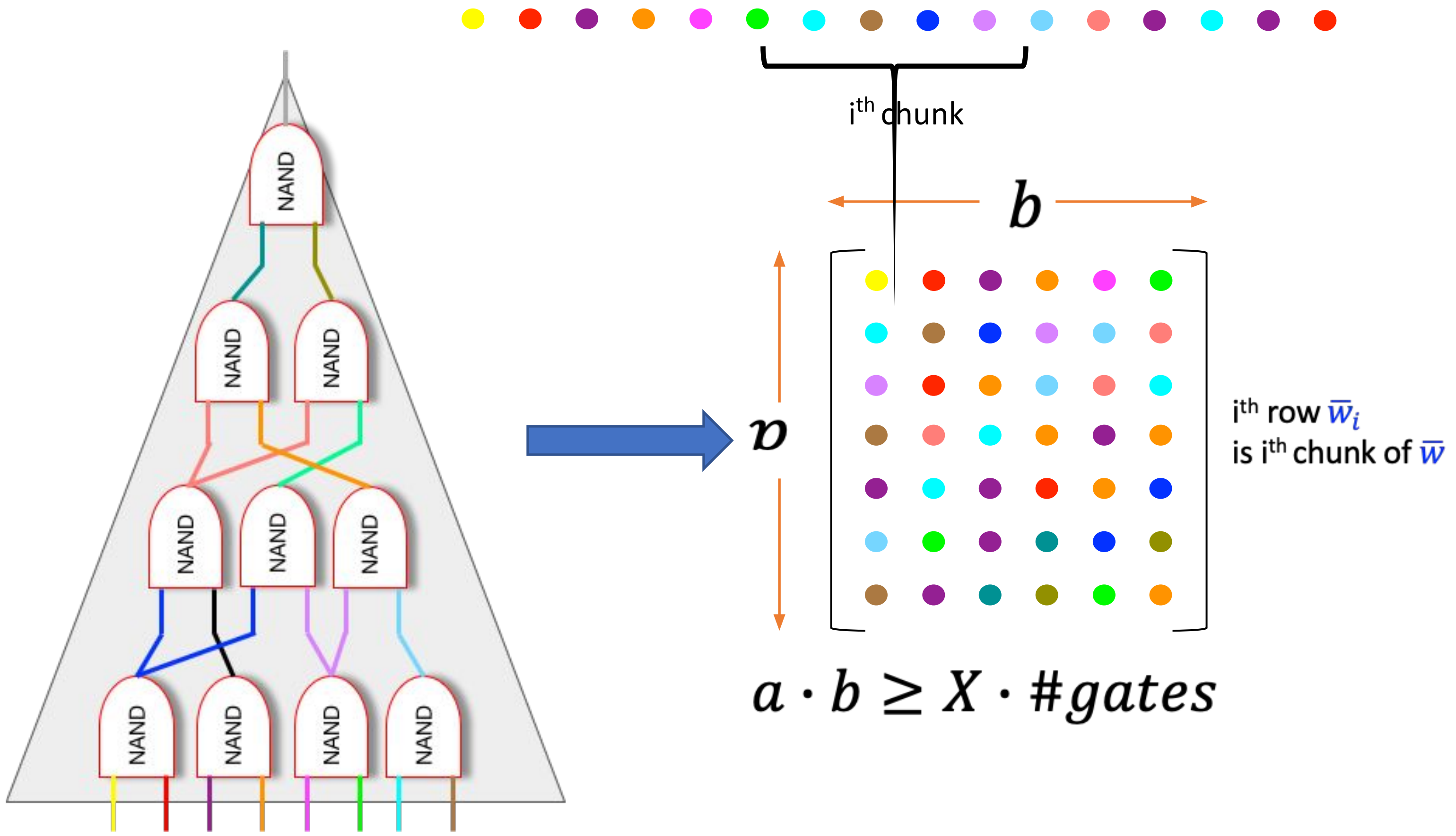
Our Techniques

Extended witness $\overline{w}$

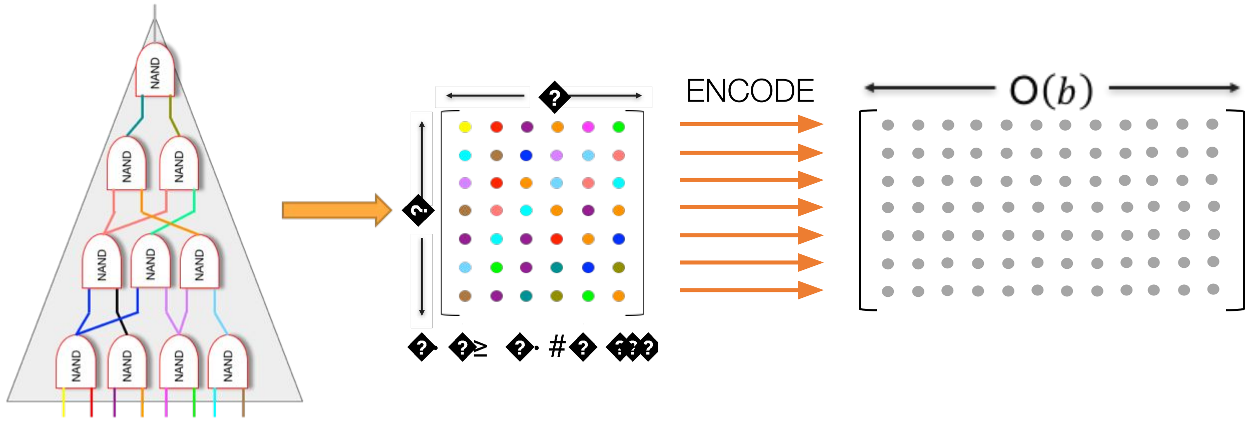
● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ●



Extended witness $\overline{w}$

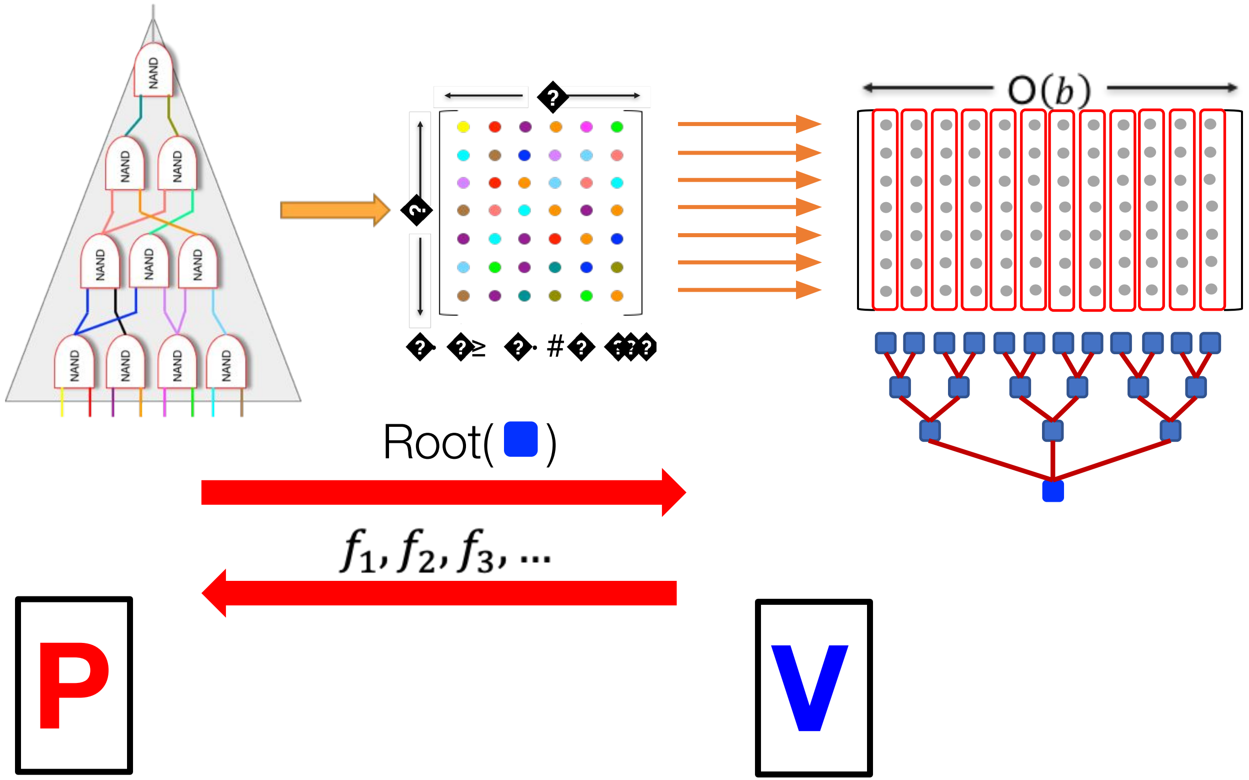


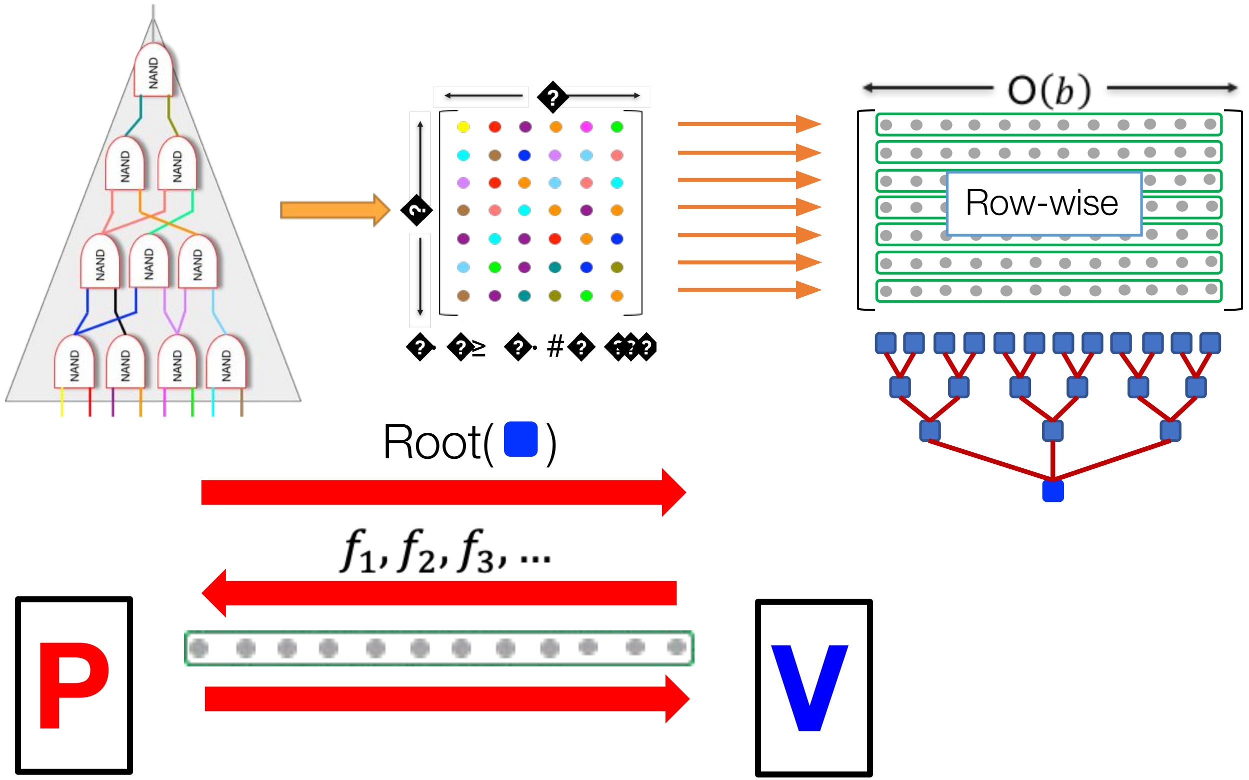
$$a \cdot b \geq X \cdot \#gates$$

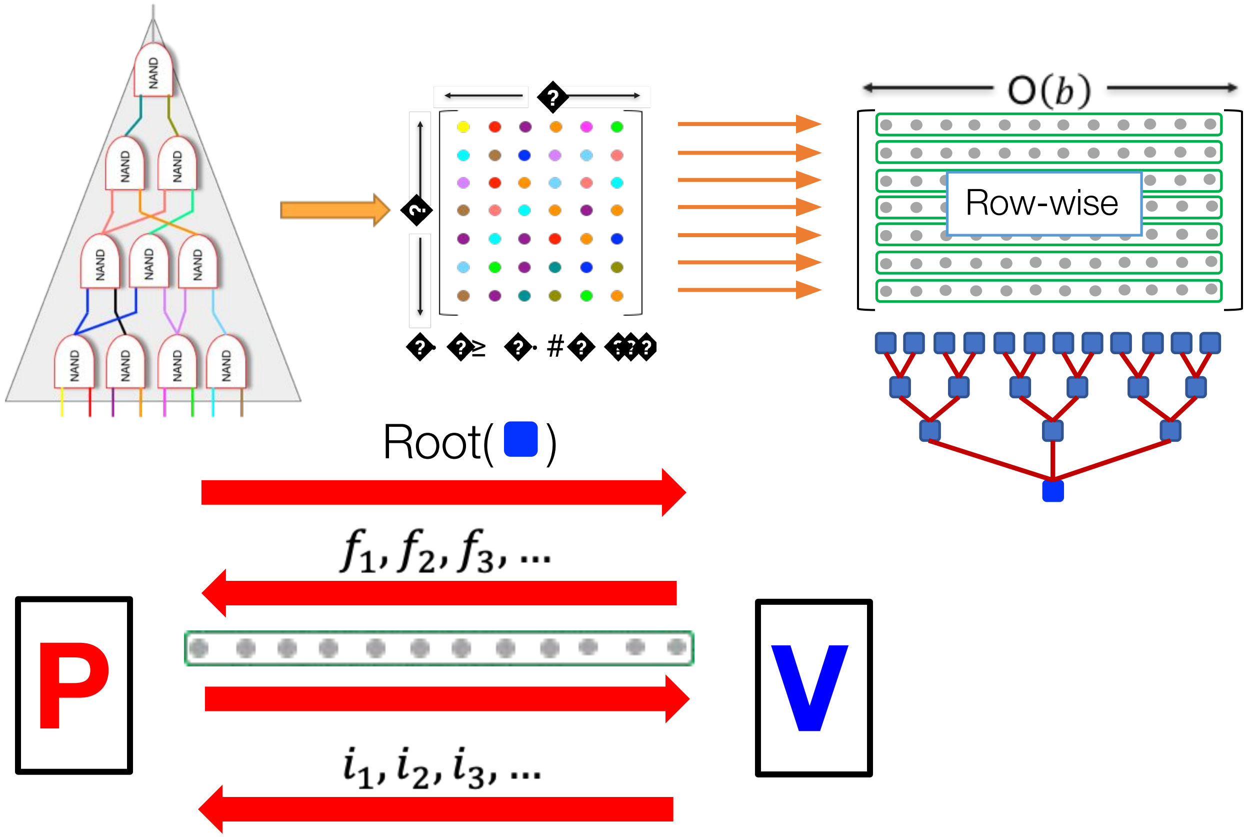


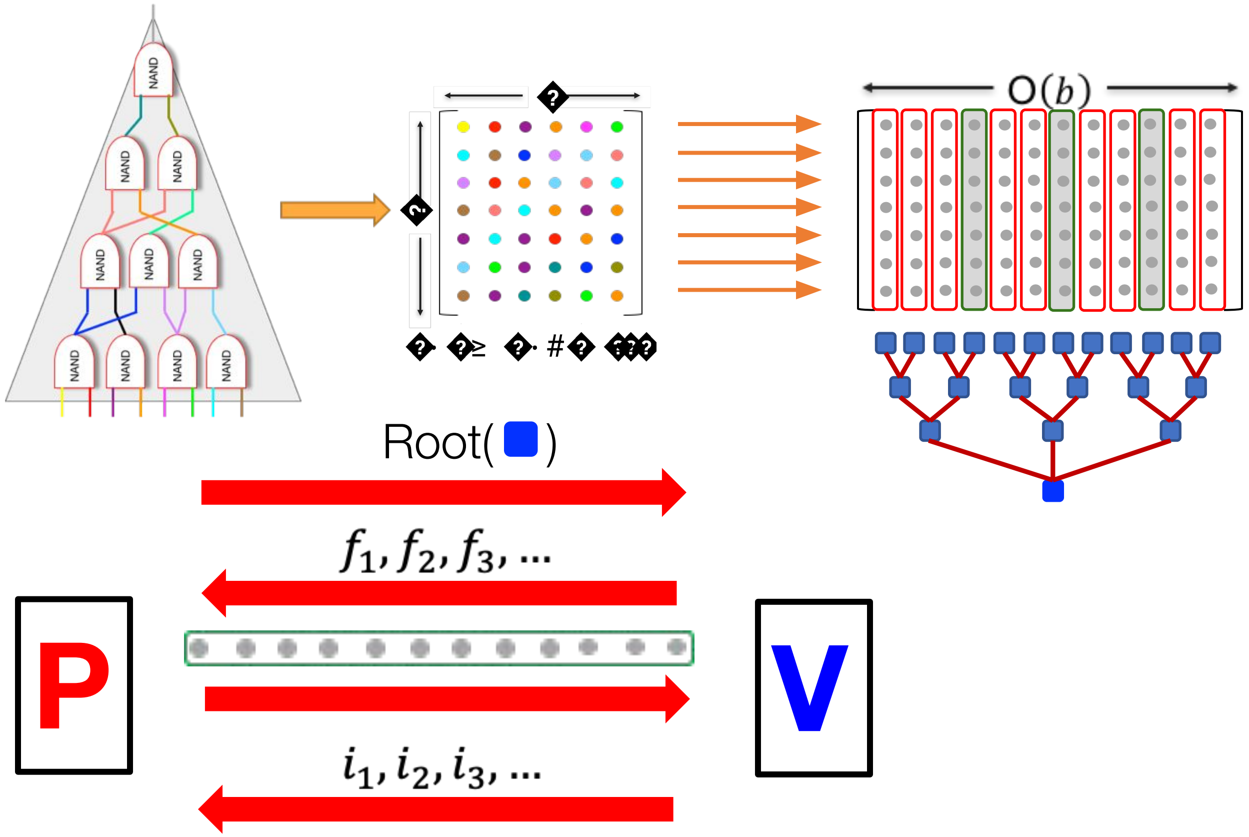
P

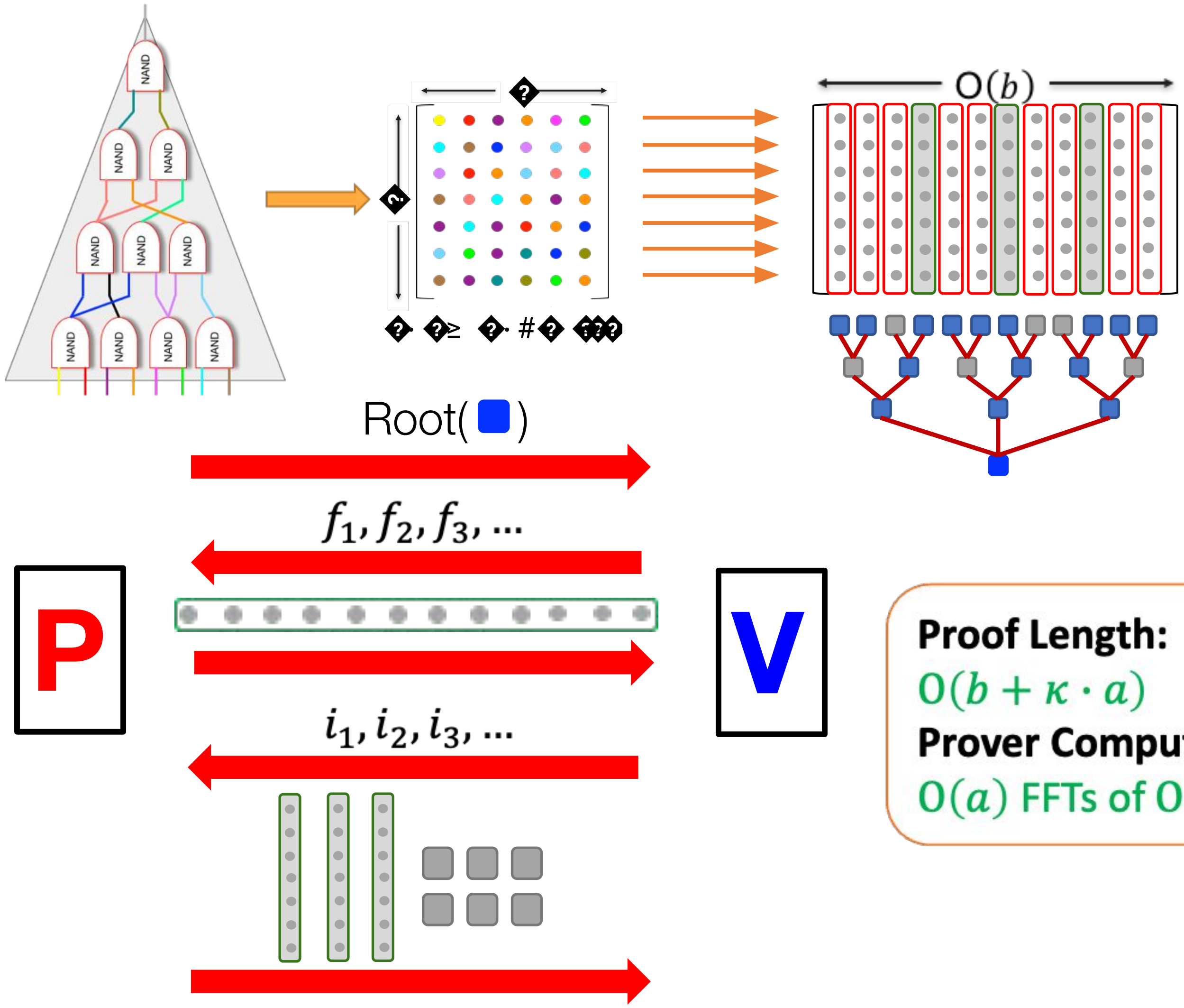
V


NAND
O(b)
Root( )
$f_1, f_2, f_3, \ldots$
P
V


NAND
O(b)
Row-wise
Root(■)
$f_1, f_2, f_3, \ldots$
P
V


NAND
O(b)
Row-wise
Root(■)
f_1, f_2, f_3, ...
P
V
i_1, i_2, i_3, ...


NAND
O(b)
Root( )
f₁, f₂, f₃, ...
i₁, i₂, i₃, ...
P
V



Proof Length:

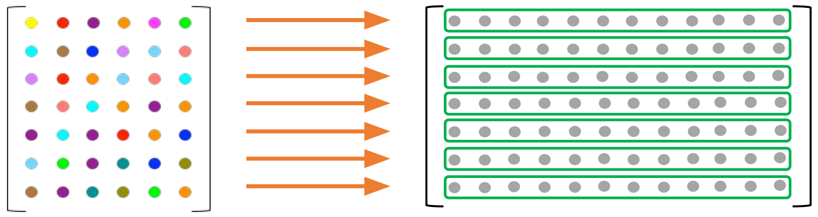
$O(b + \kappa \cdot a)$

Prover Computation:

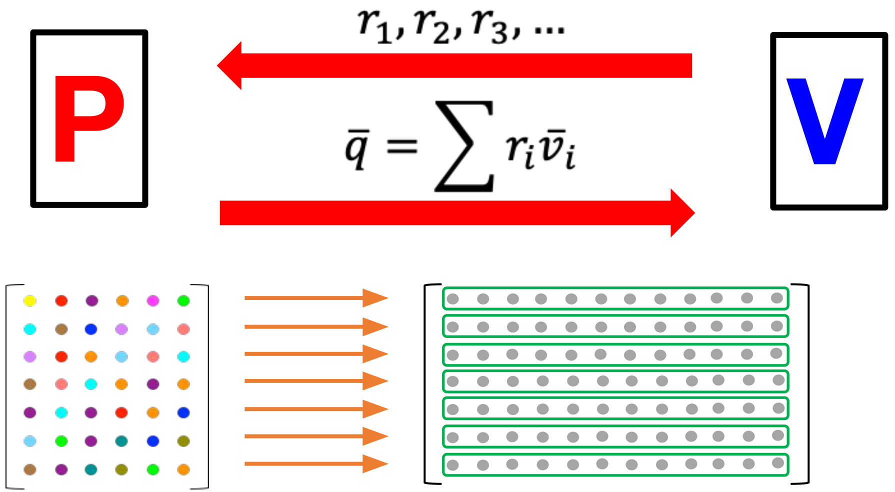
$O(a)$ FFTs of $O(b)$

What about row aggregates?

1. Code test – the prover encoded each row correctly
2. Quadratic test – the prover computed multiplication gates correctly
3. Linear test – the prover computed “linear” gates correctly



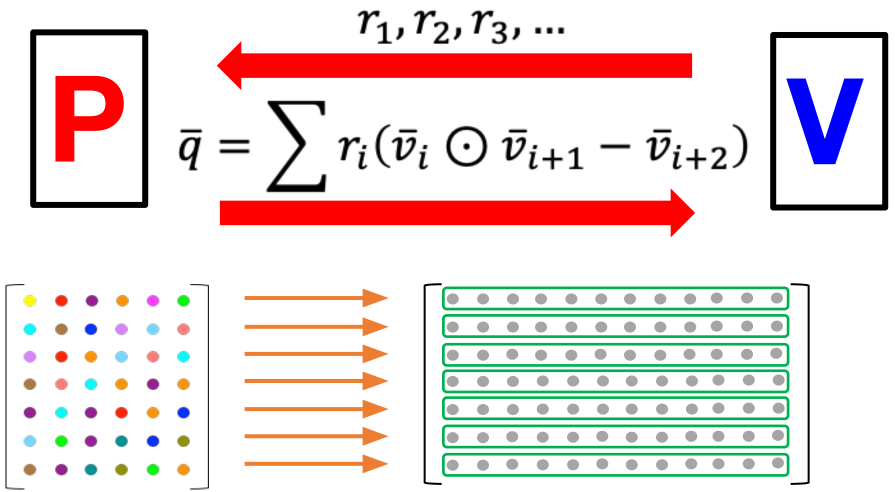
1. Code test – prover encoded correctly



Check if $\bar{q}$ is a valid codeword and agrees on random columns

2. Multiplication gates were correct

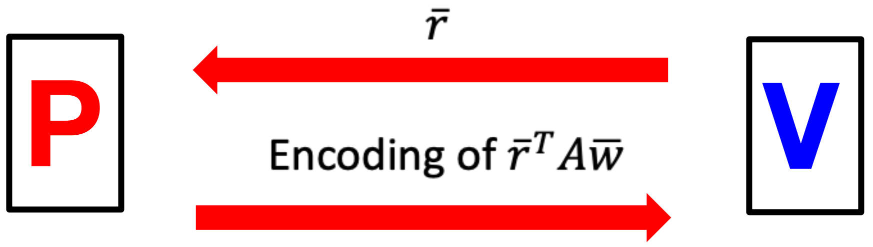
Arrange values so that all constraints for multiplication gates are so that $\overline{w}_i \odot \overline{w}_{i+1} = \overline{w}_{i+2}$ for some set of values i


$r_1, r_2, r_3, \ldots$
P
$\bar{q} = \sum r_i(\bar{v}_i \odot \bar{v}_{i+1} - \bar{v}_{i+2})$
V

Check if $\bar{q}$ is a valid codeword encoding 0s and agrees on revealed columns

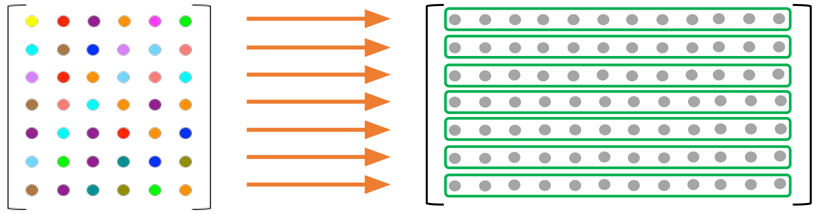
2. Linear gates were correct

Linear constraints can be expressed as $A\bar{w} = \bar{b}$



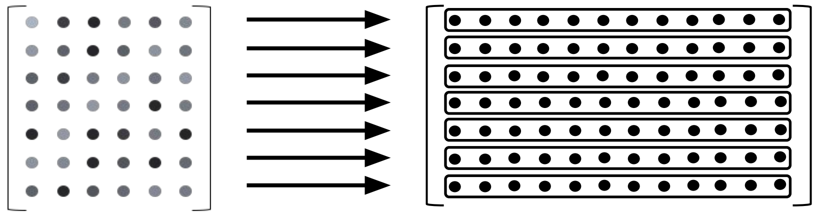
Check if response encodes $\bar{r}^T\bar{b}$

ith row $\bar{w}_i$ is ith chunk of $\bar{w}$



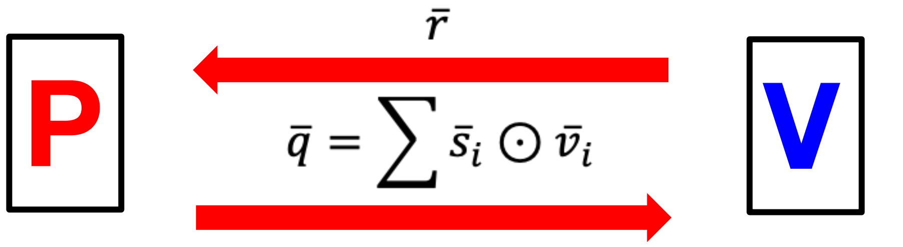
$\bar{s} = \bar{r}^T A$

ith row $\bar{s}_i$ is ith chunk of $\bar{s}$



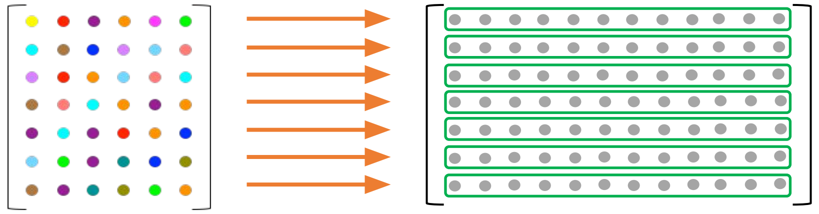
3. Linear gates were correct

Linear constraints can be expressed as $A\bar{w} = \bar{b}$



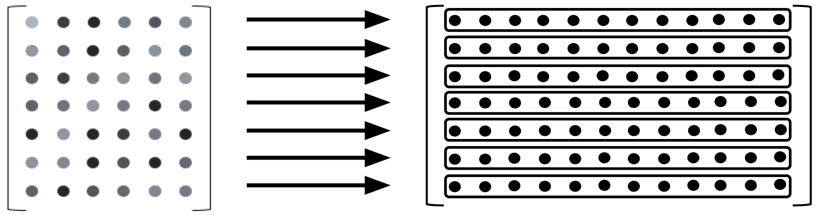
Check if response encodes $\bar{r}^T\bar{b}$

ith row $\bar{w}_i$ is ith chunk of $\bar{w}$



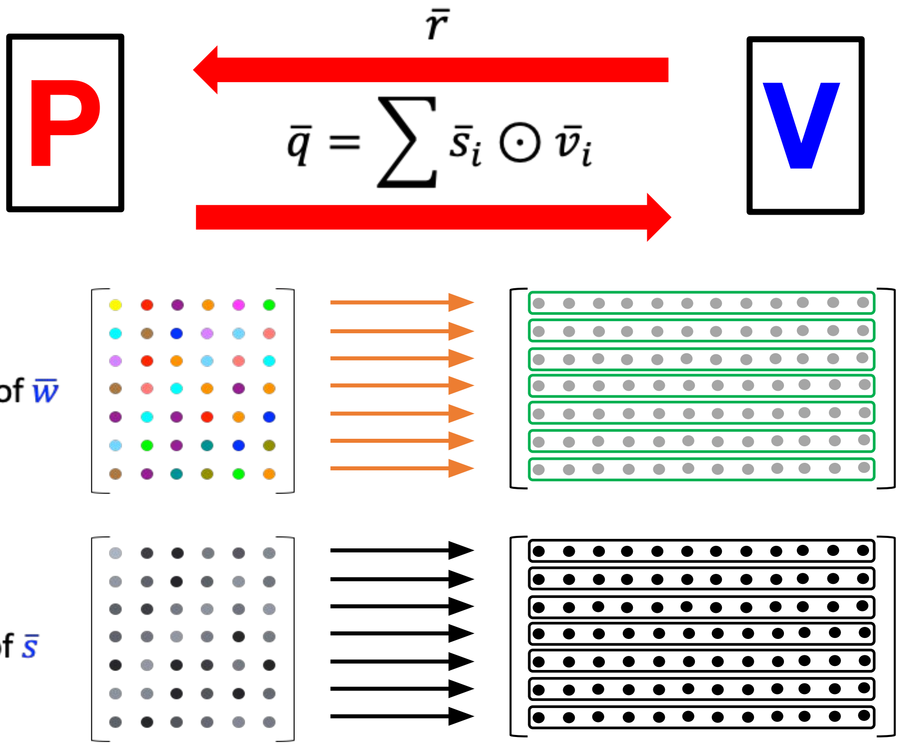
$\bar{s} = \bar{r}^T A$

ith row $\bar{s}_i$ is ith chunk of $\bar{s}$



3. Linear gates were correct

Linear constraints can be expressed as $A\bar{w} = \bar{b}$

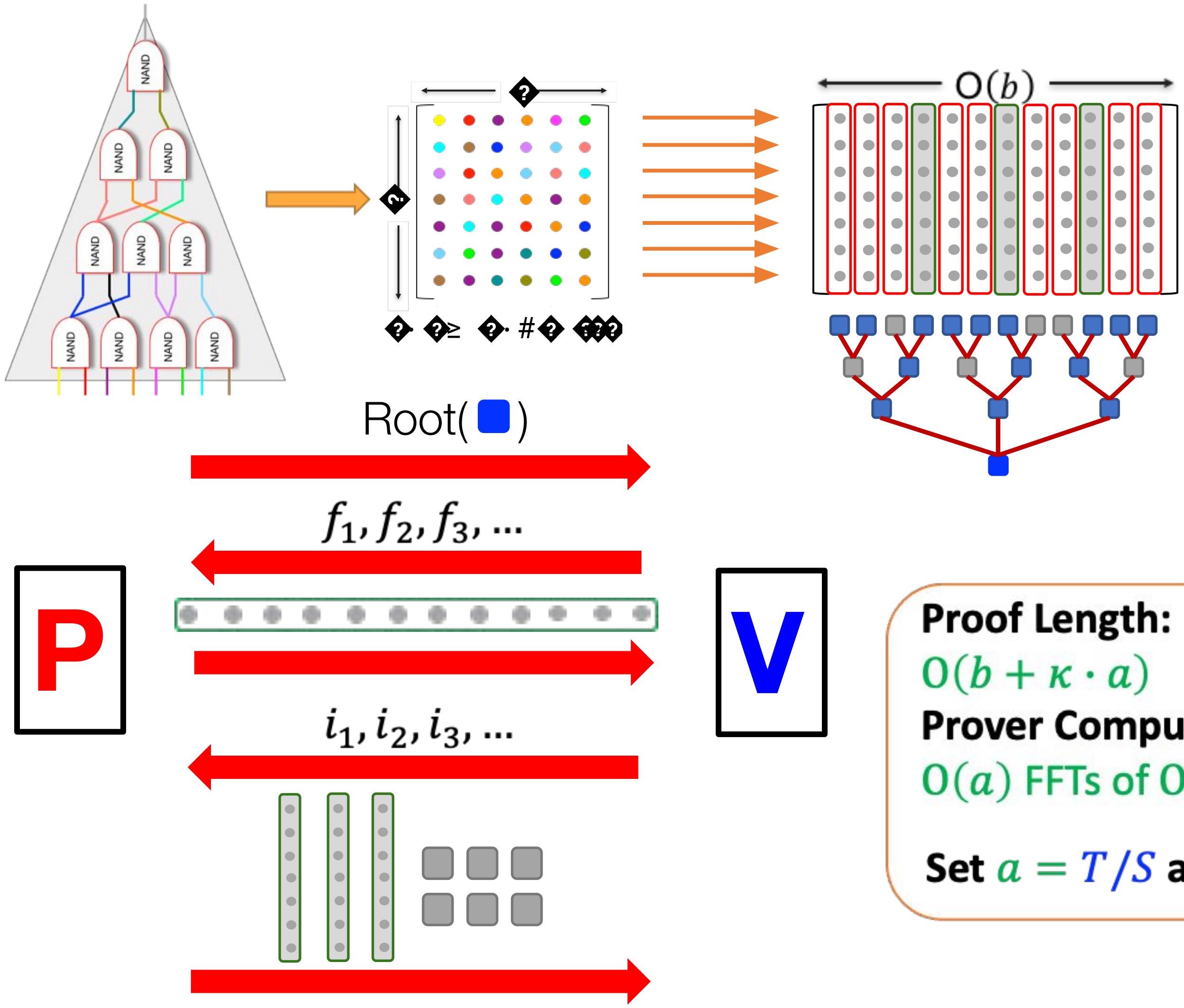


Check if $\bar{q}$ is a valid codeword encoding $\bar{z}$ s.t. $\mathbf{1}^T \bar{z} = \bar{r}^T \bar{b}$ and agrees on revealed columns

ith row $\bar{w}_i$ is ith chunk of $\bar{w}$

$\bar{s} = \bar{r}^T A$

ith row $\bar{s}_i$ is ith chunk of $\bar{s}$



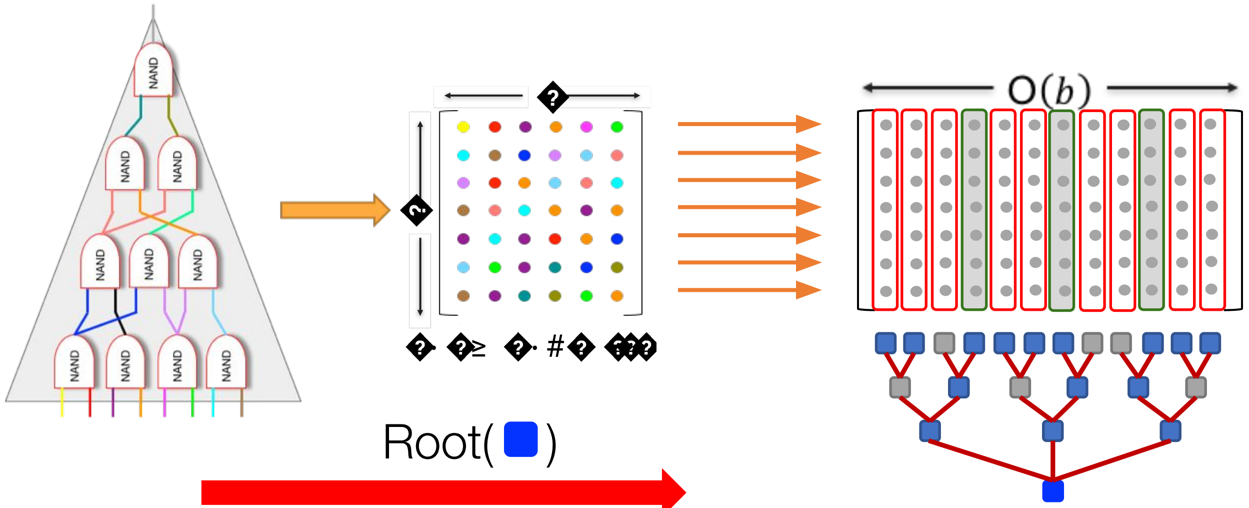
Proof Length:

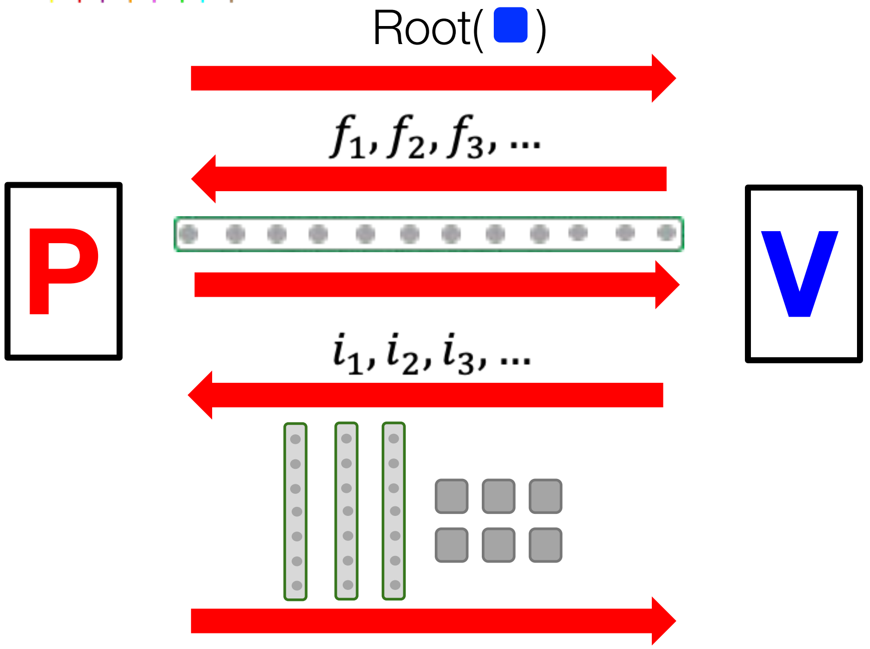
$O(b + \kappa \cdot a)$

Prover Computation:

$O(a)$ FFTs of $O(b)$

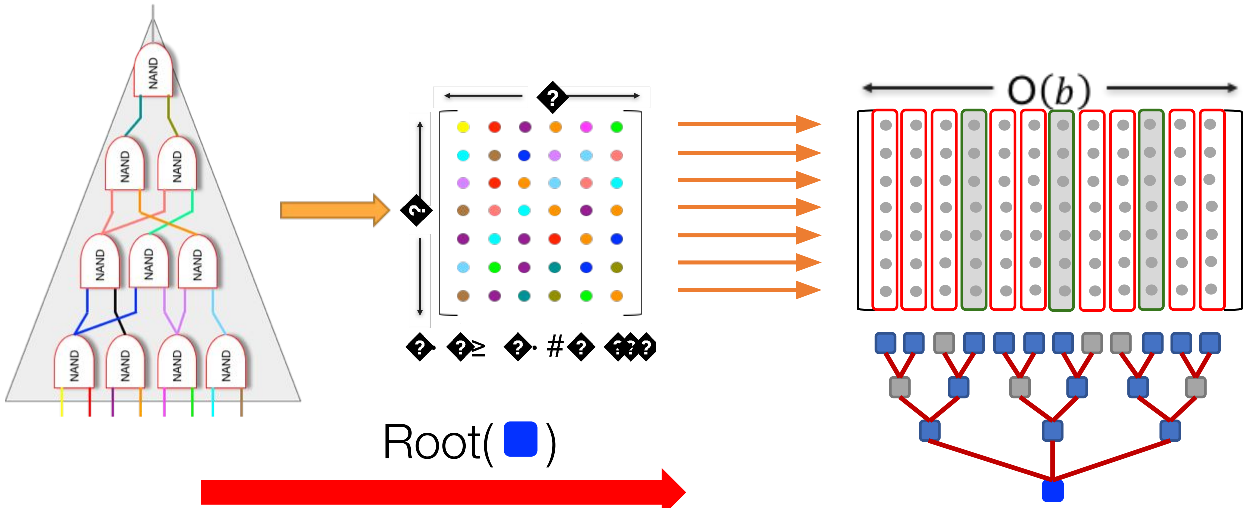
Set $a = T/S$ **and** $b = S$

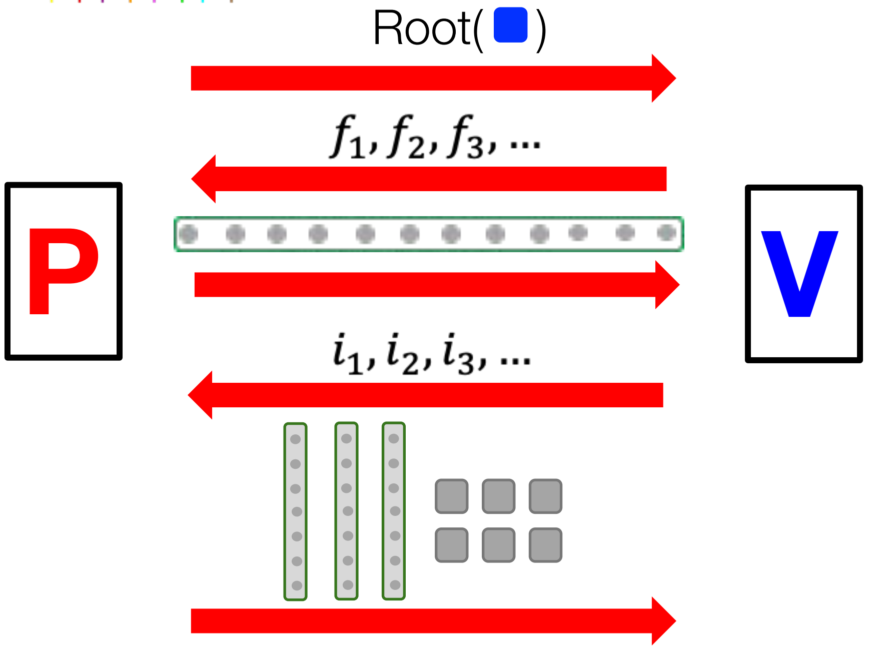




Proof Length:
$\tilde{O}(T/S)$
Prover Computation:
$\tilde{O}(T)$

Set $a = T/S$ **and** $b = S$





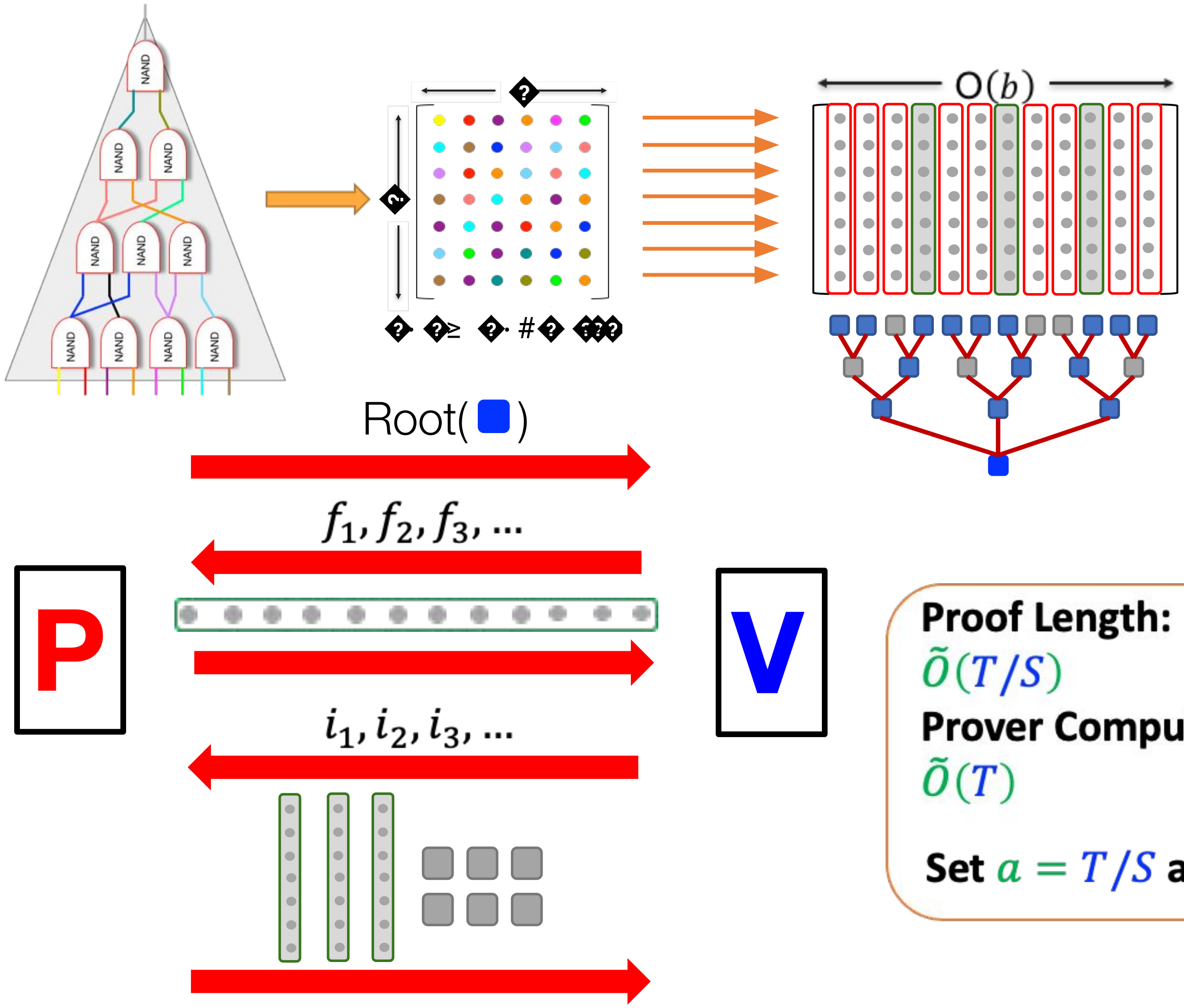
Proof Length:

$O(T/S + \kappa \cdot S)$

Prover Computation:

$O(T/S)$ FFTs of $O(S)$

Set $a = T/S$ **and** $b = S$



Proof Length:
$\tilde{O}(T/S)$
Prover Computation:
$\tilde{O}(T)$

Set $a = T/S$ **and** $b = S$

Build your ZK app at ligetron.com

Scan this to verify proof on your device