Zero-Knowledge Proofs And ZK-SNARKs

Foundations Seminar

Helger Lipmaa, April, 2025

Brief Introduction

- Introduction to zero-knowledge and zk-SNARKs

Brief Introduction

- Introduction to zero-knowledge and zk-SNARKs
- **Goal:** introduction, trying to sell the hype, collaboration

Brief Introduction

- Introduction to zero-knowledge and zk-SNARKs
- **Goal:** introduction, trying to sell the hype, collaboration
  - Give an up-to-date overview of the area

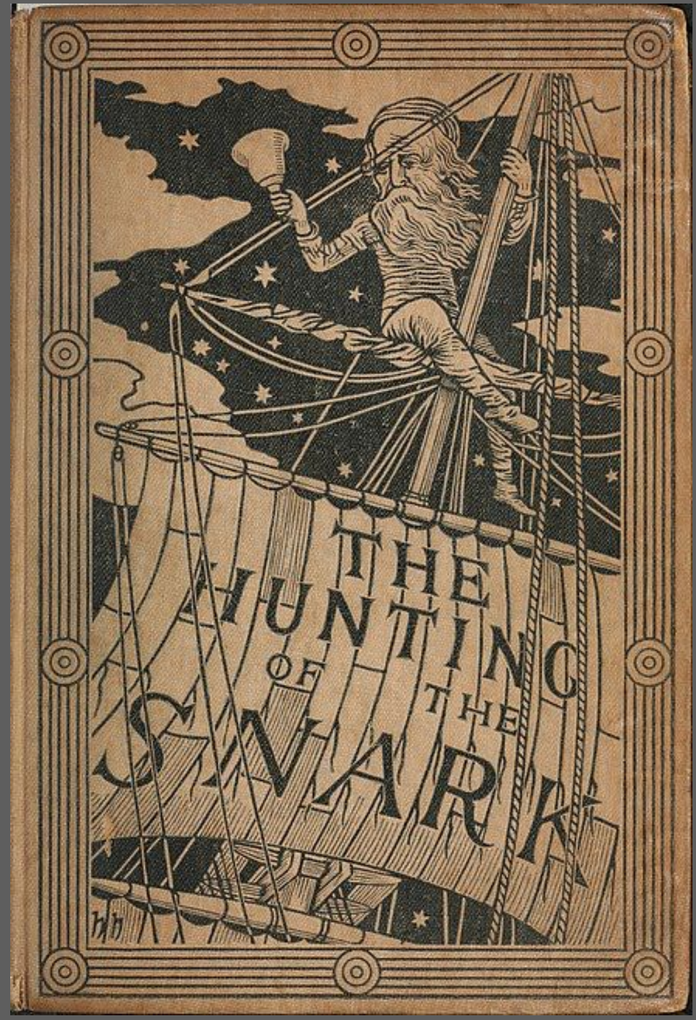
Brief Introduction

- Introduction to zero-knowledge and zk-SNARKs
- **Goal:** introduction, trying to sell the hype, collaboration
  - Give an up-to-date overview of the area
  - ZK field is wide, and there is a lot of collaborations possible

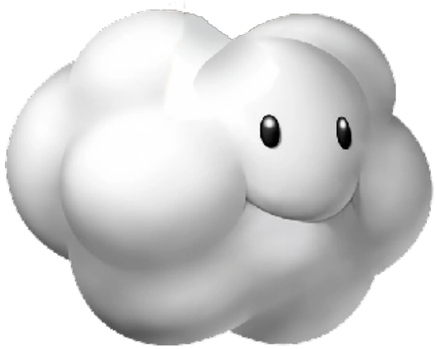
Brief Introduction

- Introduction to zero-knowledge and zk-SNARKs
- **Goal:** introduction, trying to sell the hype, collaboration
  - Give an up-to-date overview of the area
  - ZK field is wide, and there is a lot of collaborations possible
  - Coding theory, ML, formal verification, …

ZK-SNARKs: Motivations



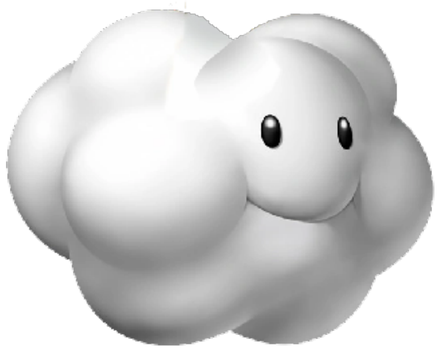
Motivation: Verifiable Computation



Motivation: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

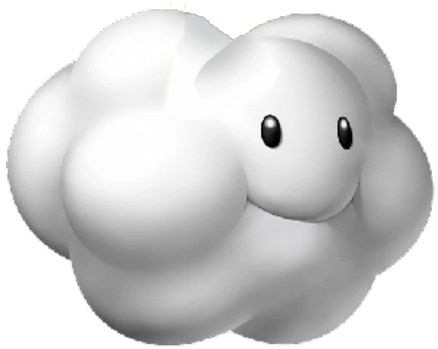


Motivation: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w

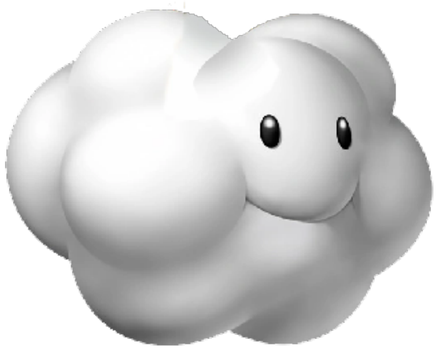


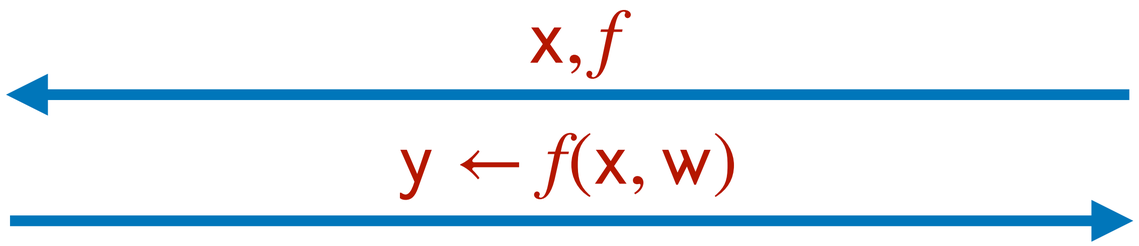
Motivation: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w



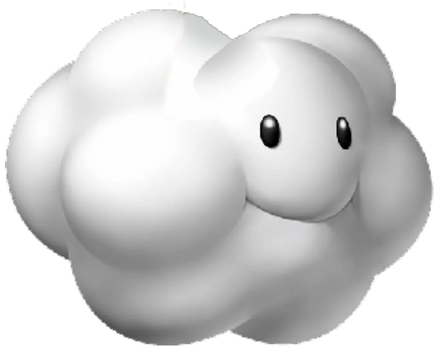


Motivation: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w



x, f

$y \leftarrow f(x, w)$

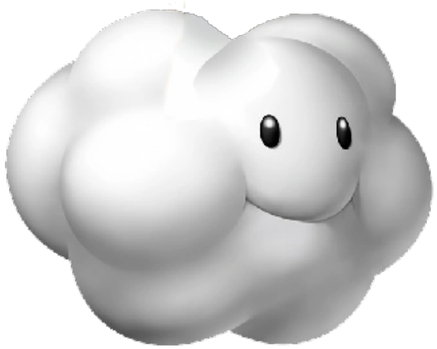
Thank you! Here's your $1000

Motivation: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w



x, f

$y \leftarrow f(x, w)$

Thank you! Here's your $1000

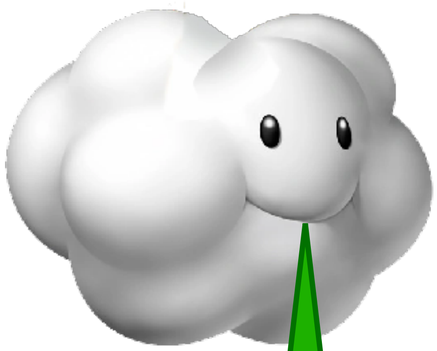
I don't trust this guy! Is the output really correct or I was scammed?

Motivation: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w



x, f

$y \leftarrow f(x, w)$

Thank you! Here's your $1000

I don't trust this guy! I don't want the computation to leak my private data

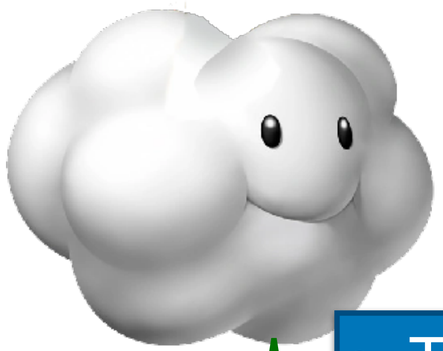
I don't trust this guy! Is the output really correct or I was scammed?

Motivation: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w



x, f

$y \leftarrow f(x, w)$

I don't trust this … want the com… leak my priv…

… this guy! Is the …lly correct or I …cammed?

- The whole **currently** popular ZK field could also be called "verifiable computation" since that is the driving application
- ZK is stuck as the "sexy" name
- ZK has other, more classical applications, like authentication, that currently get less attention
  - Less money ☹…

Solution: zk-SNARK

Solution: zk-SNARK

Computation: f
Public input (statement) x
Private input (witness) w

Solution: zk-SNARK

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

Solution: zk-SNARK

Computation: f
Public input (statement) x
Private input (witness) w

$y \leftarrow f(x, w)$

Computation: f
Public input (statement) x

Solution: zk-SNARK

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

$y \leftarrow f(x, w)$

Proof π that $f(x, w) = y$

Solution: zk-SNARK

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

$y \leftarrow f(x, w)$

Proof π that $f(x, w) = y$

Proof can be interactive:
- Consist of several message back and forth

Security

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

$y \leftarrow f(x, w)$

Proof π that $f(x, w) = y$

- **Completeness:** honest verifier accepts honest prover

Security

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

$y \leftarrow f(x, w)$

Proof π that $f(x, w) = y$

- **Completeness:** honest verifier accepts honest prover
- **Knowledge Soundness:** if honest verifier accepts, prover “knows" w

Security

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

$y \leftarrow f(x, w)$

Proof π that $f(x, w) = y$

- **Completeness:** honest verifier accepts honest prover
- **Knowledge Soundness:** if honest verifier accepts, prover “knows" w
  - **Proof** = knowledge-sound even if prover is **omnipotent**

Security

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

$y \leftarrow f(x, w)$

Proof π that $f(x, w) = y$

- **Completeness:** honest verifier accepts honest prover
- **Knowledge Soundness:** if honest verifier accepts, prover “knows" w
  - **Proof** = knowledge-sound even if prover is **omnipotent**
  - **Argument** = knowledge-sound only against **polynomial-time** provers

Security

Computation: f
Public input (statement) x
Private input (witness) w

Computation: f
Public input (statement) x

$y \leftarrow f(x, w)$

Proof π that $f(x, w) = y$

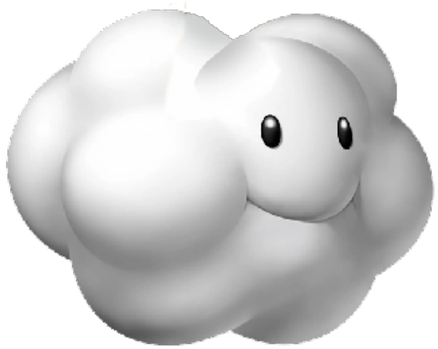
- **Completeness:** honest verifier accepts honest prover
- **Knowledge Soundness:** if honest verifier accepts, prover “knows" w
  - **Proof** = knowledge-sound even if prover is **omnipotent**
  - **Argument** = knowledge-sound only against **polynomial-time** provers
- **Zero-Knowledge:** nothing about the private input of honest prover is leaked

Efficiency

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w



x, f

$y \leftarrow f(x, w)$

Argument π that $f(x, w) = y$

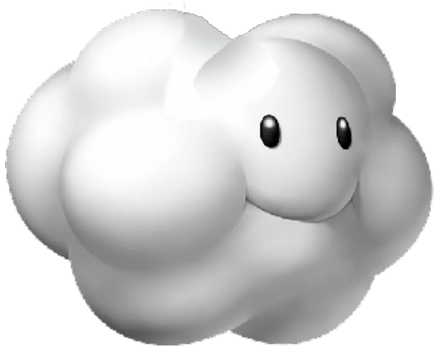
Thank you! Here's your $1000

Efficiency

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w



x, f

$y \leftarrow f(x, w)$

Argument π that $f(x, w) = y$

Thank you! Here's your $1000

If verification of a zk-SNARK takes as much time as recomputing, the application is less interesting!

The cloud still preserves her privacy

Zk-SNARKs

- Zero-knowledge **succinct** non-interactive arguments of knowledge

Zk-SNARKs

- Zero-knowledge **succinct** non-interactive arguments of knowledge
- Adds efficiency requirements to “just" ZK arguments:

Zk-SNARKs

- Zero-knowledge **succinct** non-interactive arguments of knowledge
- Adds efficiency requirements to “just" ZK arguments:
  - Argument should be **much shorter** than the computation description

Zk-SNARKs

- Zero-knowledge **succinct** non-interactive arguments of knowledge
- Adds efficiency requirements to “just" ZK arguments:
  - Argument should be **much shorter** than the computation description
    - Ideally: constant or logarithmic in computation length n

Zk-SNARKs

- Zero-knowledge **succinct** non-interactive arguments of knowledge
- Adds efficiency requirements to “just" ZK arguments:
  - Argument should be **much shorter** than the computation description
    - Ideally: constant or logarithmic in computation length n
  - Verification should be **much faster** than the computation

Zk-SNARKs

- Zero-knowledge **succinct** non-interactive arguments of knowledge
- Adds efficiency requirements to “just" ZK arguments:
  - Argument should be **much shorter** than the computation description
    - Ideally: constant or logarithmic in computation length n
  - Verification should be **much faster** than the computation
    - Ideally: constant or logarithmic

Zk-SNARKs

- Zero-knowledge **succinct** non-interactive arguments of knowledge
- Adds efficiency requirements to “just" ZK arguments:
  - Argument should be **much shorter** than the computation description
    - Ideally: constant or logarithmic in computation length n
  - Verification should be **much faster** than the computation
    - Ideally: constant or logarithmic
- Note: proofs cannot be succinct, but arguments can

Zk-SNARKs

- Zero-knowledge **succinct** non-interactive arguments of knowledge
- Adds efficiency requirements to “just" ZK arguments:
  - Argument should be **much shorter** than the computation description
    - Ideally: constant or logarithmic in computation length n
  - Verification should be **much faster** than the computation
    - Ideally: constant or logarithmic
- Note: proofs cannot be succinct, but arguments can

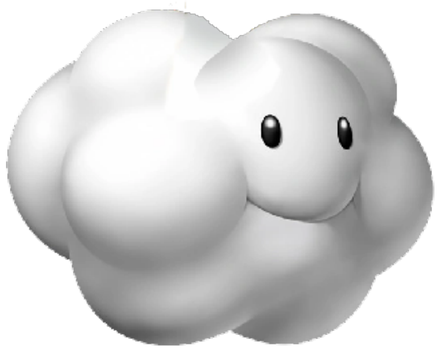
- **Scalability**: verifying argument is significantly faster than recomputation
- Important in many applications
  - even **without** privacy… — a lot of what is called ZK is actually **not** ZK but VC

Recall: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w

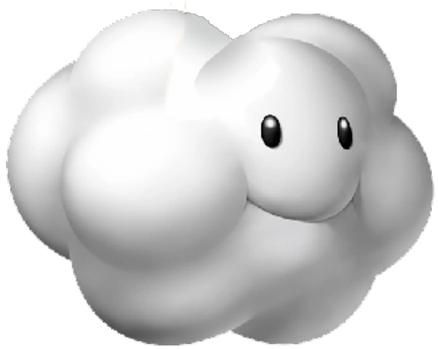


Recall: Verifiable Computation

Computation f: **arbitrary** computation of $\leq 2^{30}$ steps

Public input: x

Private input: w



x, f

$y \leftarrow f(x, w)$

Argument π that $f(x, w) = y$

- **Completeness:** honest verifier accepts if prover **knows** w such that $f(x, w) = y$
- **Knowledge Soundness:** if honest verifier accepts, prover **knows** w
- **Zero-Knowledge: nothing** about w is leaked
- **Efficiency:** verifying π should be **much faster** than redoing the computation

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography
  - **But:** sometimes can be only 10x (GKR with regular circuits)

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography
  - **But:** sometimes can be only 10x (GKR with regular circuits)
- **Memory cost ☹️:** usually linear in computation size

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography
  - **But:** sometimes can be only 10x (GKR with regular circuits)
- **Memory cost ☹️:** usually linear in computation size
  - Often **the** limitation

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography
  - **But:** sometimes can be only 10x (GKR with regular circuits)
- **Memory cost ☹️:** usually linear in computation size
  - Often **the** limitation
  - Recent research improves on this — streaming zk-SNARKs

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography
  - **But:** sometimes can be only 10x (GKR with regular circuits)
- **Memory cost ☹️:** usually linear in computation size
  - Often **the** limitation
  - Recent research improves on this — streaming zk-SNARKs
- **Verifier time:** milliseconds for arbitrary computation

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography
  - **But:** sometimes can be only 10x (GKR with regular circuits)
- **Memory cost ☹️:** usually linear in computation size
  - Often **the** limitation
  - Recent research improves on this — streaming zk-SNARKs
- **Verifier time:** milliseconds for arbitrary computation
- Concrete numbers depend on the construction

State-of-the-Art

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography
  - **But:** sometimes can be only 10x (GKR with regular circuits)
- **Memory cost ☹️:** usually linear in computation size
  - Often **the** limitation
  - Recent research improves on this — streaming zk-SNARKs
- **Verifier time:** milliseconds for arbitrary computation
- Concrete numbers depend on the construction
- There is often an explicit trade-off between prover's and verifier's time

State-of-the-Art

This number might be outdated

- **Prover time ☹️:** up to 10000x overhead
  - Depends on application and underlying cryptography
  - **But:** sometimes can be only 10x (GKR with regular circuits)
- **Memory cost ☹️:** usually linear in computation size
  - Often **the** limitation
  - Recent research improves on this — streaming zk-SNARKs
- **Verifier time:** milliseconds for arbitrary computation
- Concrete numbers depend on the construction
- There is often an explicit trade-off between prover's and verifier's time
- **Very active** research topic — prover overhead decreases each year

Application: Cryptocurrencies

Application: Cryptocurrencies

Computation f: computing transaction y from public info

Public input: x (public information on blockchain)

Private input: w

- transaction amount, payer account, payee account, …

Computation: f

Public input: x

$y \leftarrow f(x, w)$

Argument π that $f(x, w) = y$

- **Completeness**
- **Knowledge Soundness**
- **Zero-Knowledge**
- **Efficiency**

Application: Cryptocurrencies

- The main source of R&D at this moment
- Spending $50B on research can secure $2T money

Computation f: computing transaction y from public info

Public input: x (public information on blockchain)

Private input: w

- transaction amount, payer account, payee account, …

Computation: f

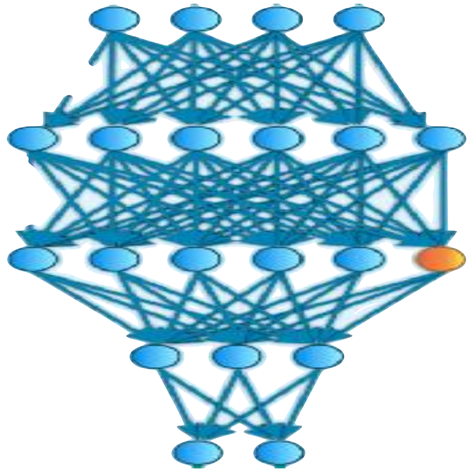
Public input: x

$y \leftarrow f(x, w)$

Argument π that $f(x, w) = y$

- **Completeness**
- **Knowledge Soundness**
- **Zero-Knowledge**
- **Efficiency**

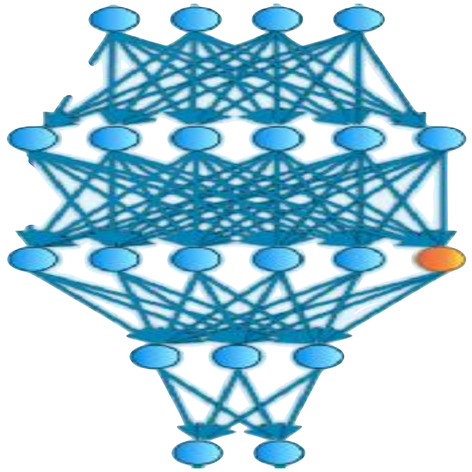
Application: zkML



Application: zkML

Or: model was trained correctly

Computation f: inference was correct
Public input: x (public input)
Private input: w (model)



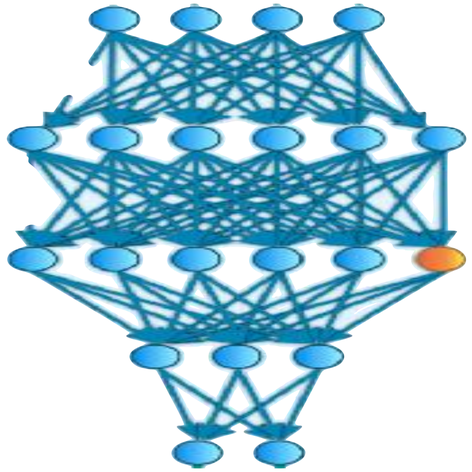
Computation: f
Public input: x

Application: zkML

Or: model was trained correctly

Computation f: inference was correct
Public input: x (public input)
Private input: w (model)

Computation: f
Public input: x



$y \leftarrow f(x, w)$

Argument π that $f(x, w) = y$

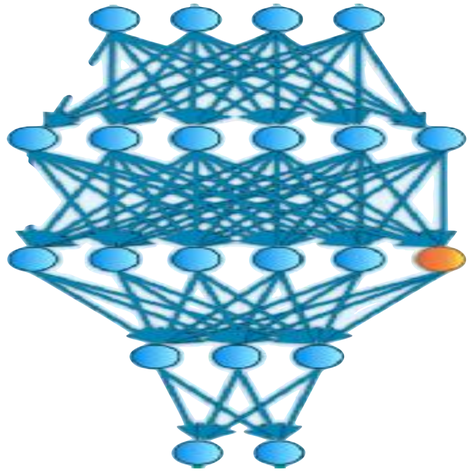
- **Completeness**
- **Knowledge Soundness**
- **Zero-Knowledge**
- **Efficiency**

Application: zkML

Or: model was trained correctly

Computation f: inference was correct
Public input: x (public input)
Private input: w (model)

Computation: f
Public input: x



$y \leftarrow f(x, w)$

Argument π that $f(x, w) = y$

- **Completeness**
- **Knowledge Soundness**
- **Zero-Knowledge**
- **Efficiency**

- **Fairness: the same model was used in all cases**
  - **Property loans, …**

Application: zkML

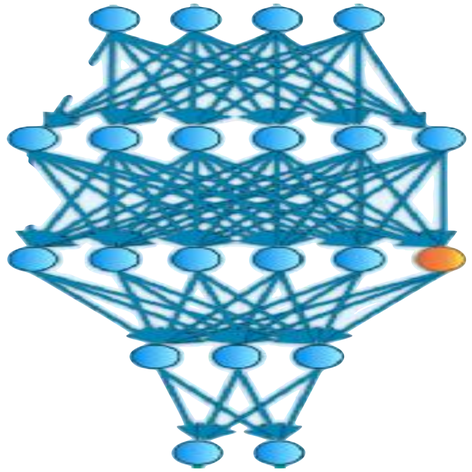
Or: model was trained correctly

Computation f: inference was corre

Public input: x (public input)

Private input: w (model)

- **Collaboration questions:**
- **In which ML-related questions, ZK can help?**
- **How does the low-level ML computation look like, can it be made more "ZK-friendly"?**



$y \leftarrow f(x, w)$

Argument π that $f(x, w) = y$

- **Completeness**
- **Knowledge Soundness**
- **Zero-Knowledge**
- **Efficiency**

- **Fairness: the same model was used in all cases**
  - **Property loans, …**

Application: E-voting

Application: E-voting

Computation f: tallying was correct

Public input: x (all incoming signed encrypted ballots)

Private input: w; who voted for who

Computation: f

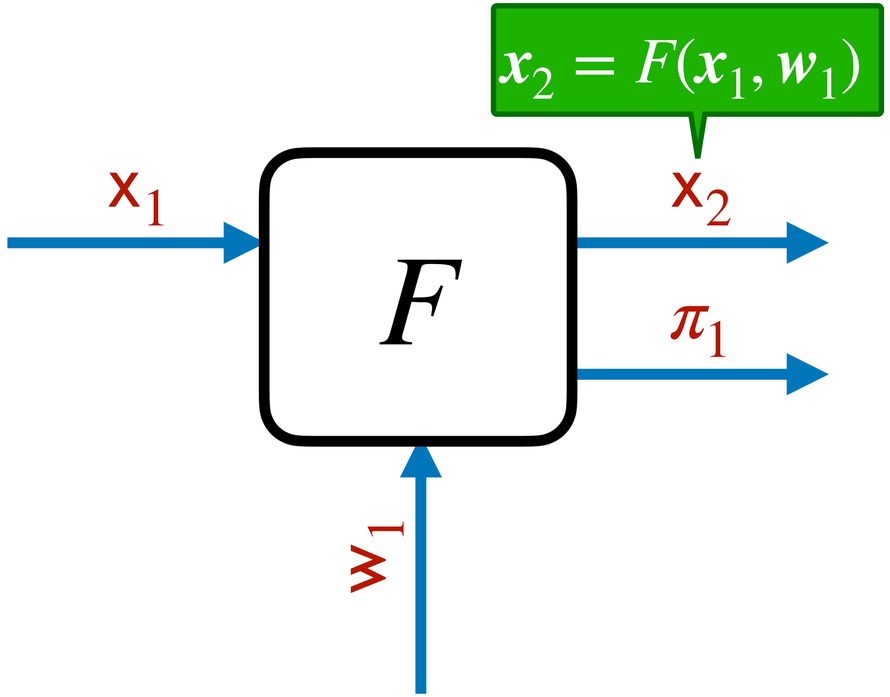
Public input: x

$\text{true} \leftarrow f(\text{x}, \text{w})$

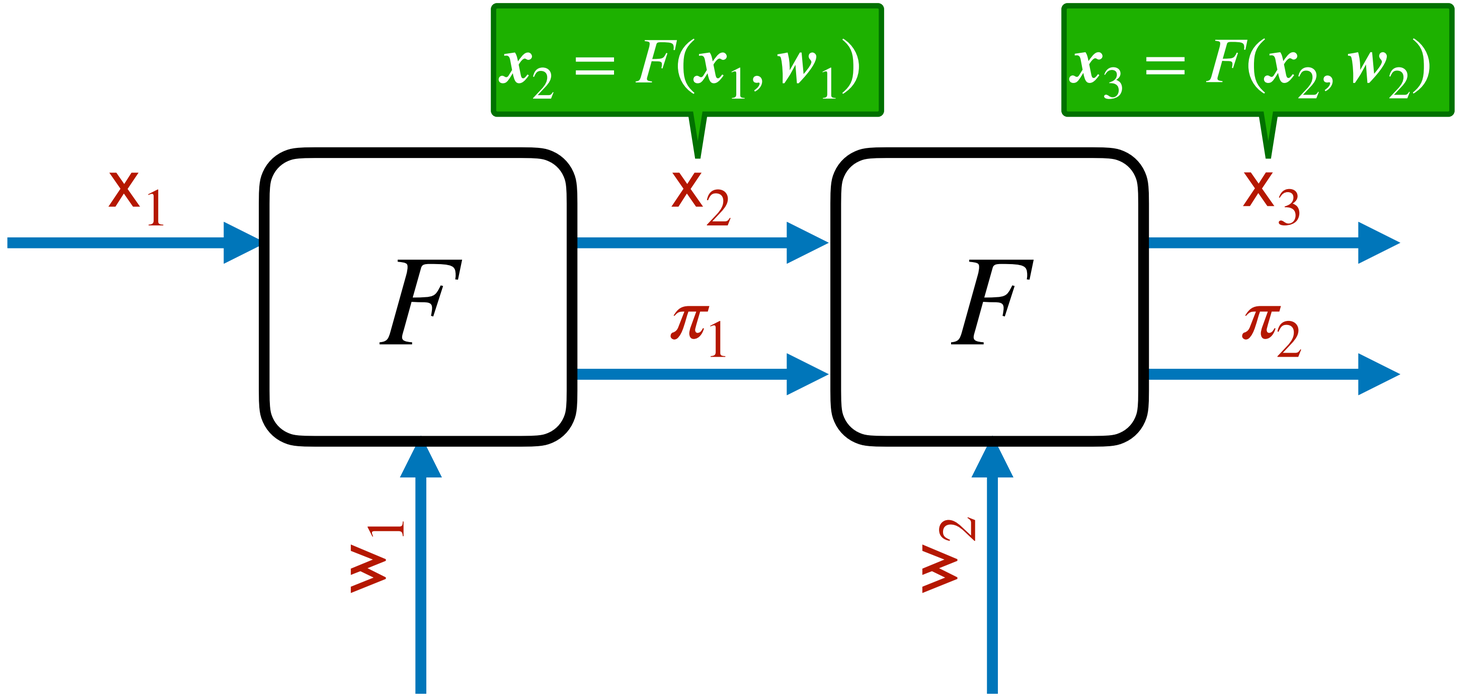
Argument π that $f(\text{x}, \text{w}) = \text{true}$

- **Completeness**
- **Knowledge Soundness**
- **Zero-Knowledge**
- **Efficiency**

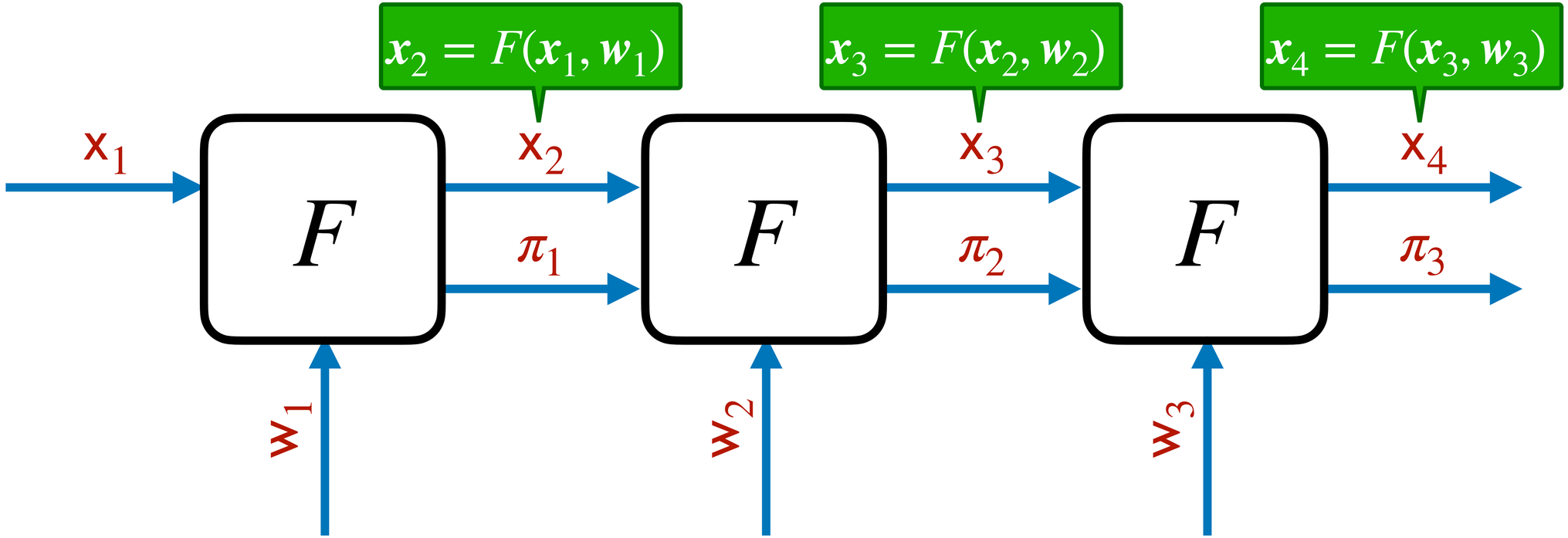
Incrementally Verifiable Computation



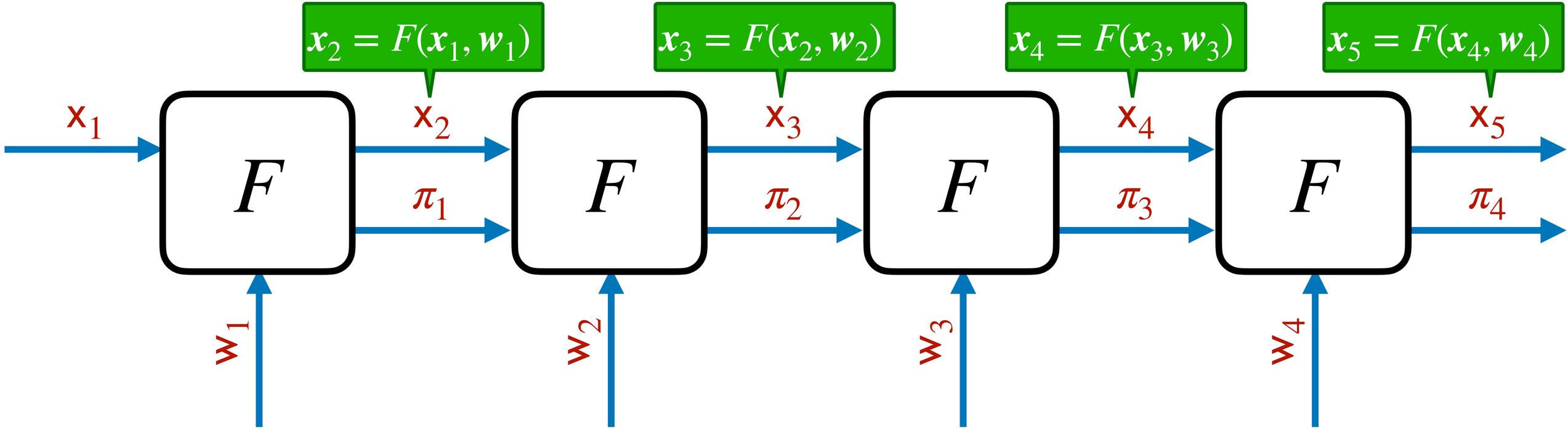
Incrementally Verifiable Computation



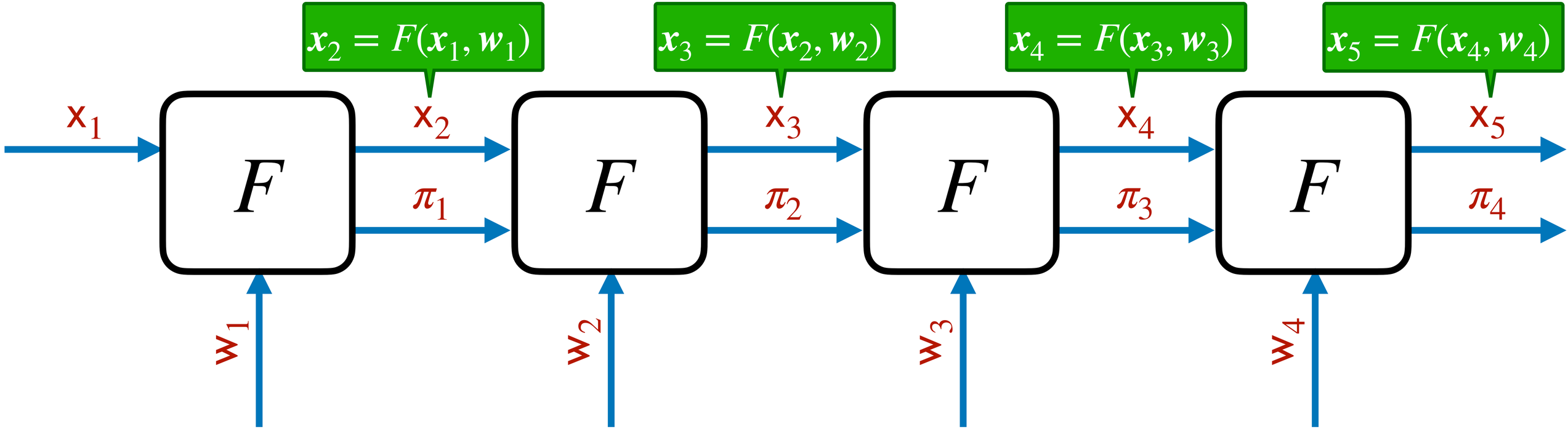
Incrementally Verifiable Computation



Incrementally Verifiable Computation

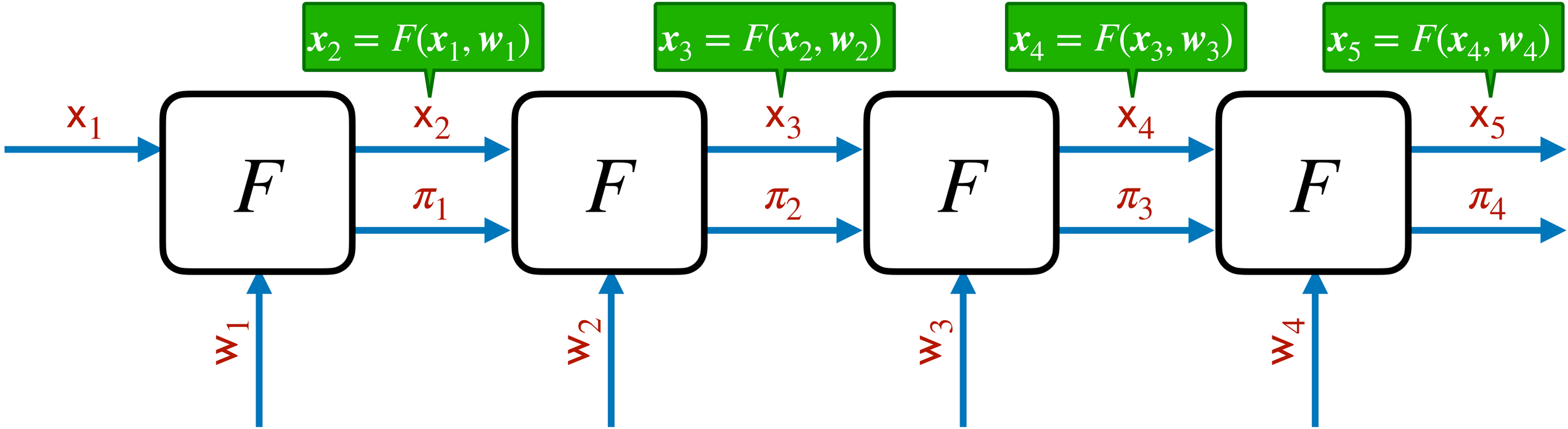


Incrementally Verifiable Computation



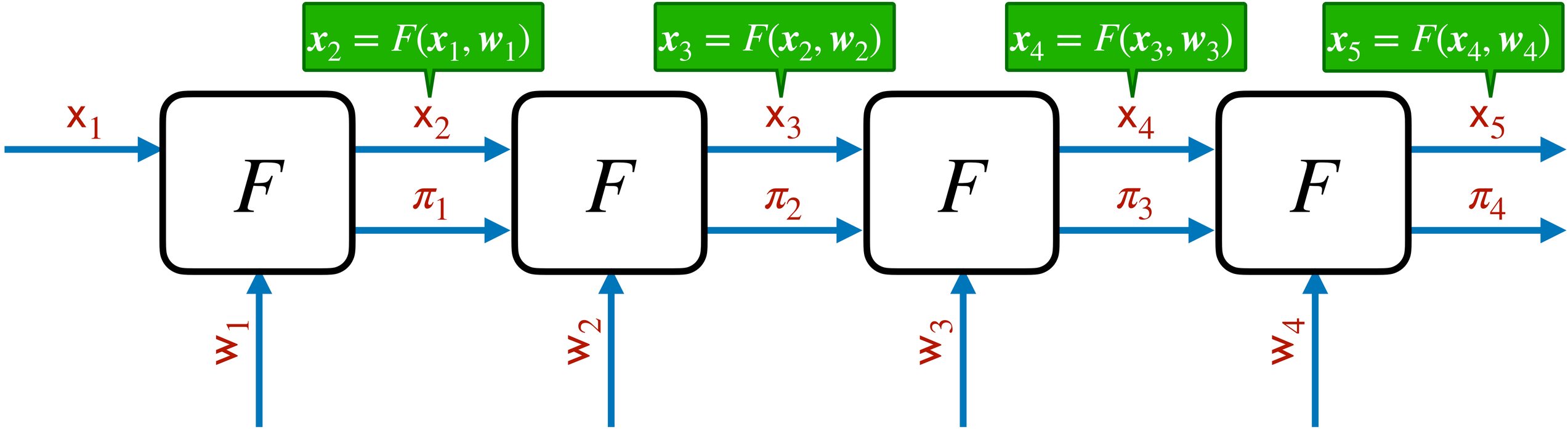
- Allows to perform proving **piecewise**, spreading the costs over time

Incrementally Verifiable Computation



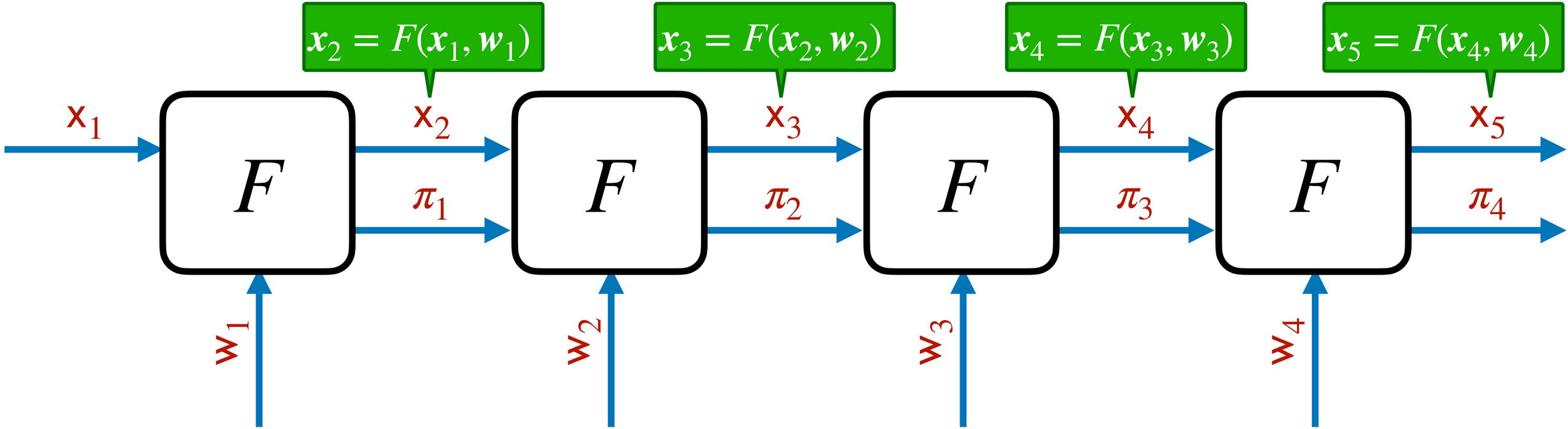
- Allows to perform proving **piecewise**, spreading the costs over time
- Example application 1: **EVM** (Ethereum Virtual Machine)

Incrementally Verifiable Computation



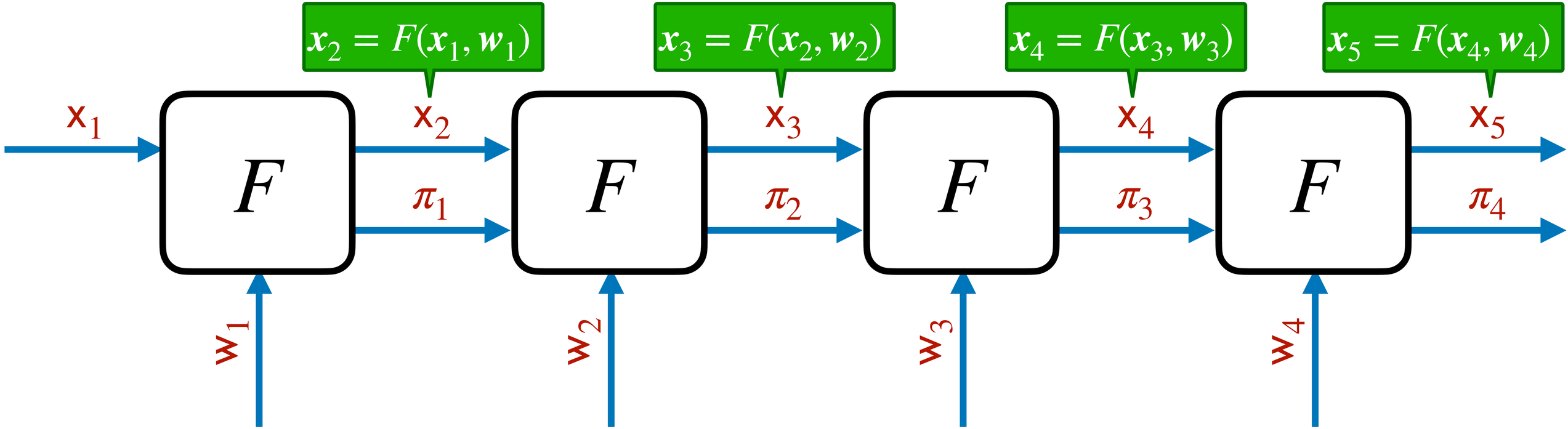
- Allows to perform proving **piecewise**, spreading the costs over time
- Example application 1: **EVM** (Ethereum Virtual Machine)
  - https://ethereum.org/en/developers/docs/evm/

Incrementally Verifiable Computation



- Allows to perform proving **piecewise**, spreading the costs over time
- Example application 1: **EVM** (Ethereum Virtual Machine)
  - https://ethereum.org/en/developers/docs/evm/
- Example application 2: **RISC Zero** (proving RISC V execution correctness)

Incrementally Verifiable Computation



- Allows to perform proving **piecewise**, spreading the costs over time
- Example application 1: **EVM** (Ethereum Virtual Machine)
  - https://ethereum.org/en/developers/docs/evm/
- Example application 2: **RISC Zero** (proving RISC V execution correctness)
  - https://www.risczero.com/

Current Machinery

The Current Machinery of ZK-SNARKs

Computation f

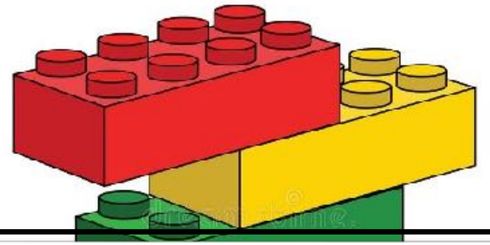
```rust
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

The Current Machinery of ZK-SNARKs

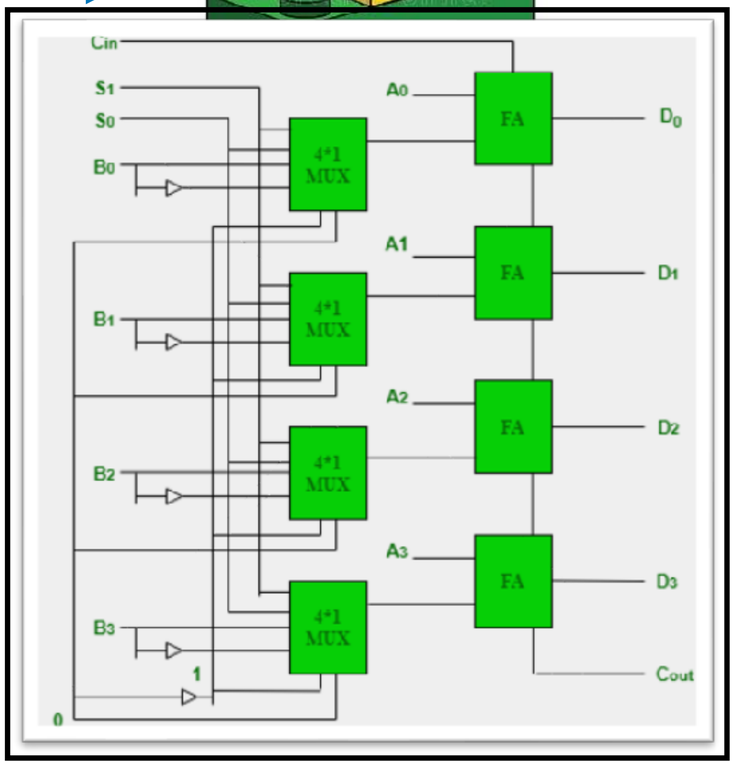
Computation f

Intermediate Representation

DSL



```rust
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```



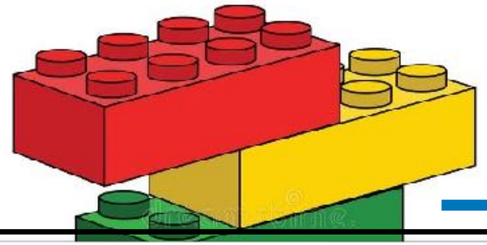
The Current Machinery of ZK-SNARKs

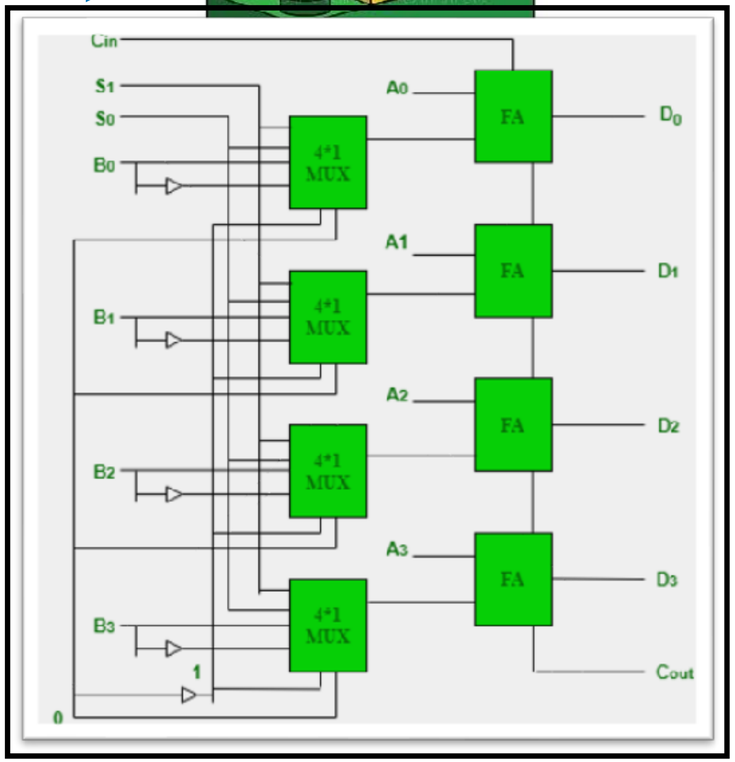
Computation f

```rust
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

DSL

Intermediate Representation





Cryptography

ZK-SNARKs

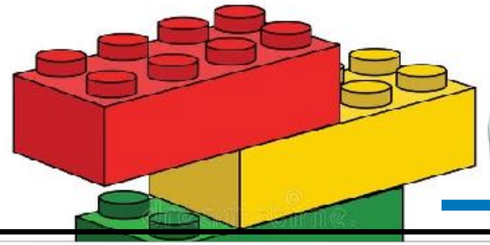
The Current Machinery of ZK-SNARKs

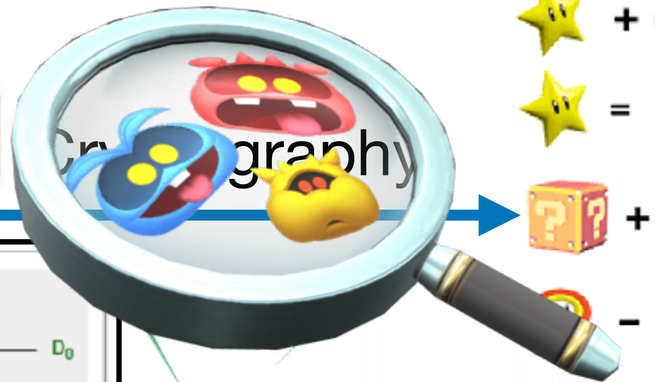
Computation f

Intermediate Representation

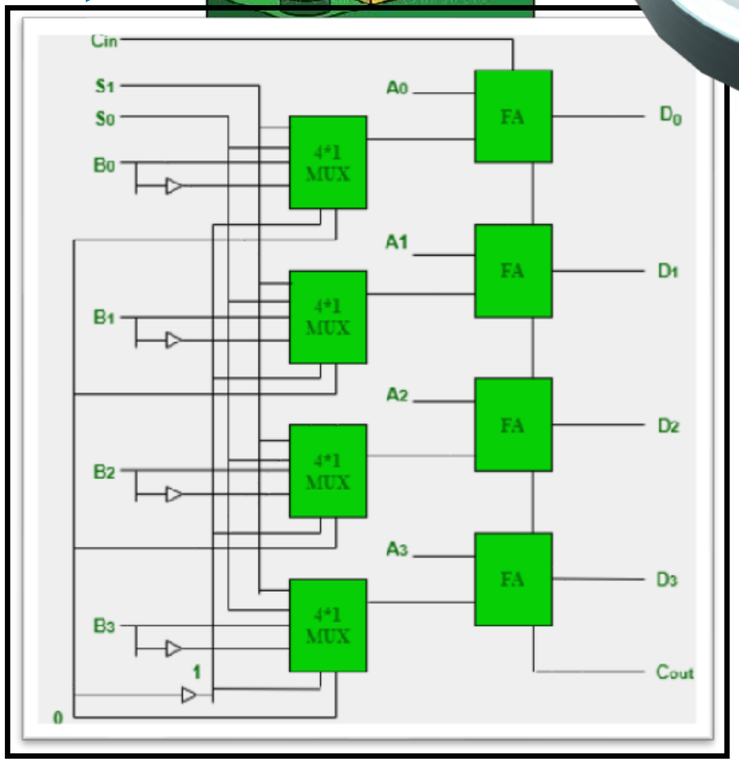
ZK-SNARKs

DSL






```rust
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```



Intermediate Representation

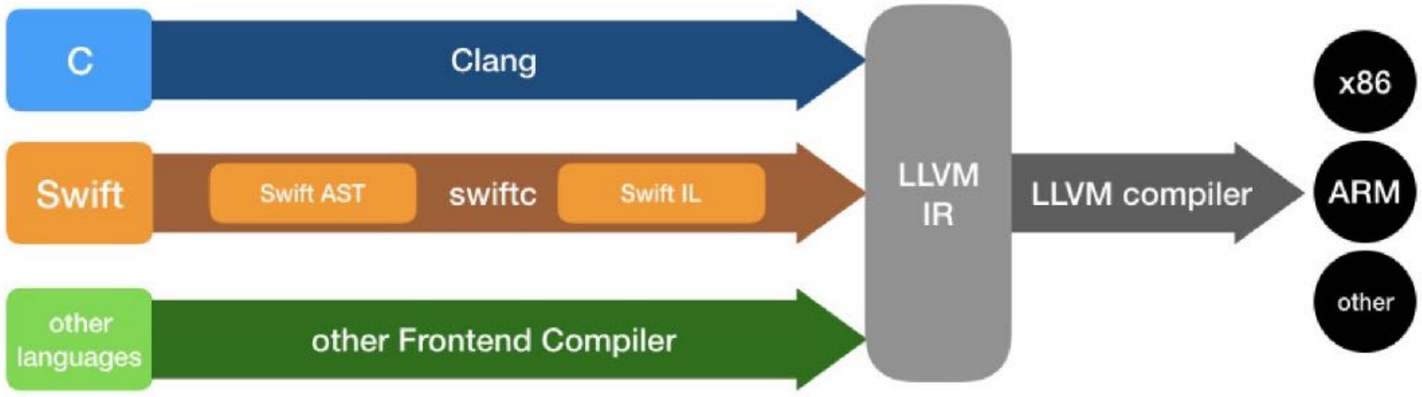
- Main philosophical question: we want to verify a computation is correct

Intermediate Representation

- Main philosophical question: we want to verify a computation is correct
- But **what** is a computation?

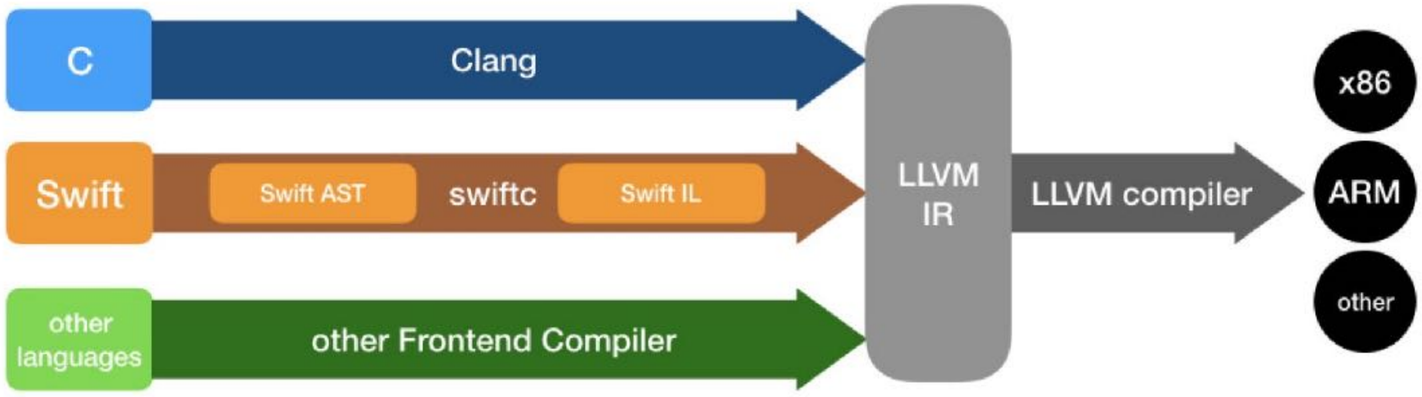
Intermediate Representation

- Main philosophical question: we want to verify a computation is correct
- But **what** is a computation?
- IR in **other contexts**: bytecode, LLVM (language-independent IR), …



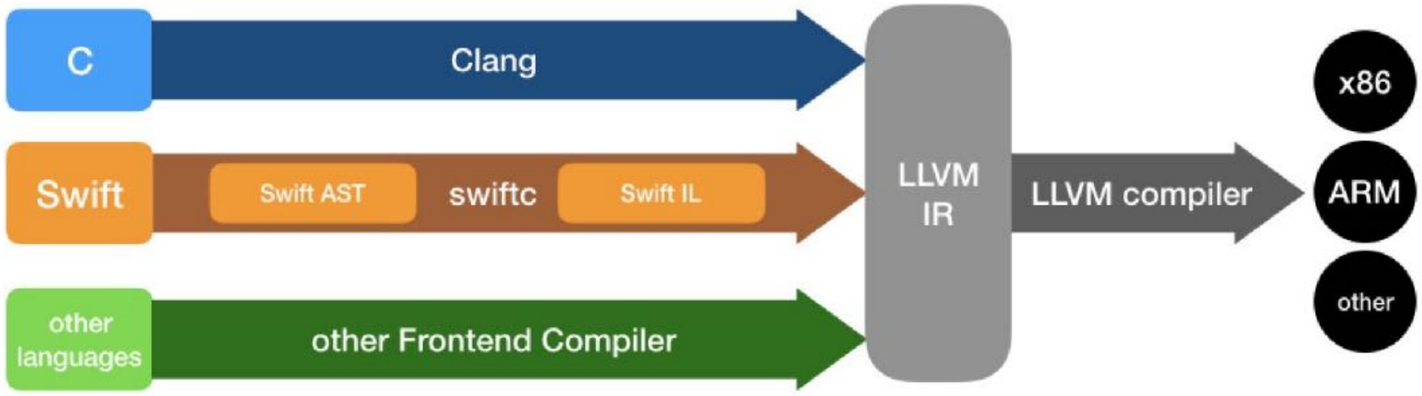
Intermediate Representation

- Main philosophical question: we want to verify a computation is correct
- But **what** is a computation?
- IR in **other contexts**: bytecode, LLVM (language-independent IR), …
- IR in **ZK**: The goal is to **verify** a function, **not** to compute it



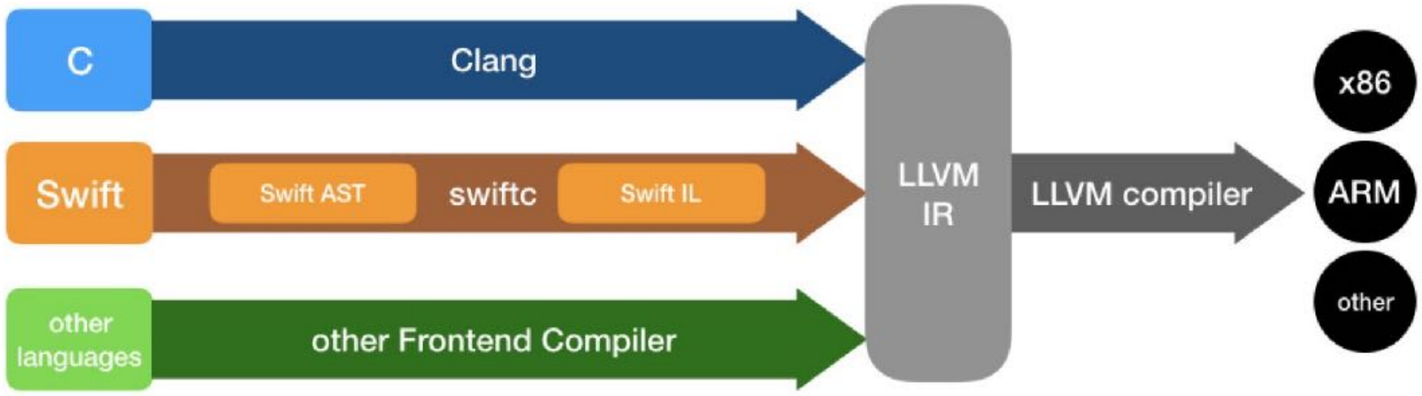
Intermediate Representation

- Main philosophical question: we want to verify a computation is correct
- But **what** is a computation?
- IR in **other contexts**: bytecode, LLVM (language-independent IR), …
- IR in **ZK**: The goal is to **verify** a function, **not** to compute it
- IR: (1) a machine model + (2) how to verify its computation



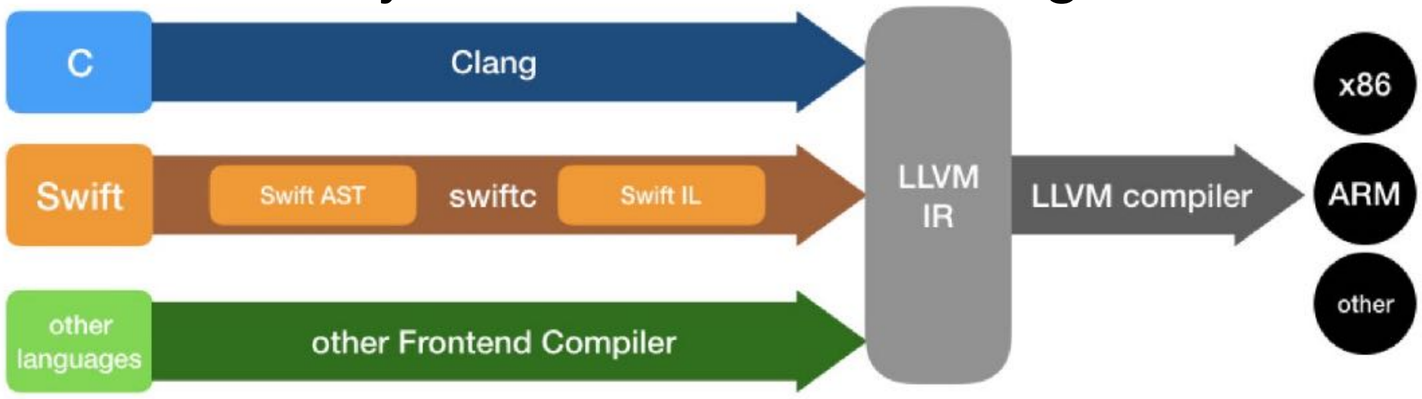
Intermediate Representation

- Main philosophical question: we want to verify a computation is correct
- But **what** is a computation?
- IR in **other contexts**: bytecode, LLVM (language-independent IR), …
- IR in **ZK**: The goal is to **verify** a function, **not** to compute it
- IR: (1) a machine model + (2) how to verify its computation
    1. Machine model: Turing machine, random access machine, circuits



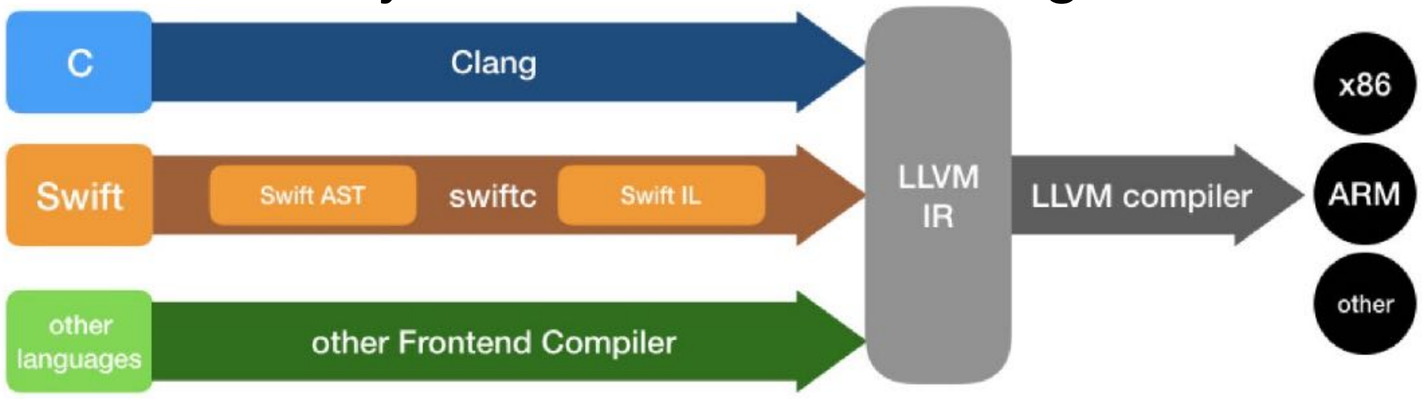
Intermediate Representation

- Main philosophical question: we want to verify a computation is correct
- But **what** is a computation?
- IR in **other contexts**: bytecode, LLVM (language-independent IR), …
- IR in **ZK**: The goal is to **verify** a function, **not** to compute it
- IR: (1) a machine model + (2) how to verify its computation
  1. Machine model: Turing machine, random access machine, circuits
  2. Formalise its verification as a few efficiently verifiable checks using a suitable “proof system”



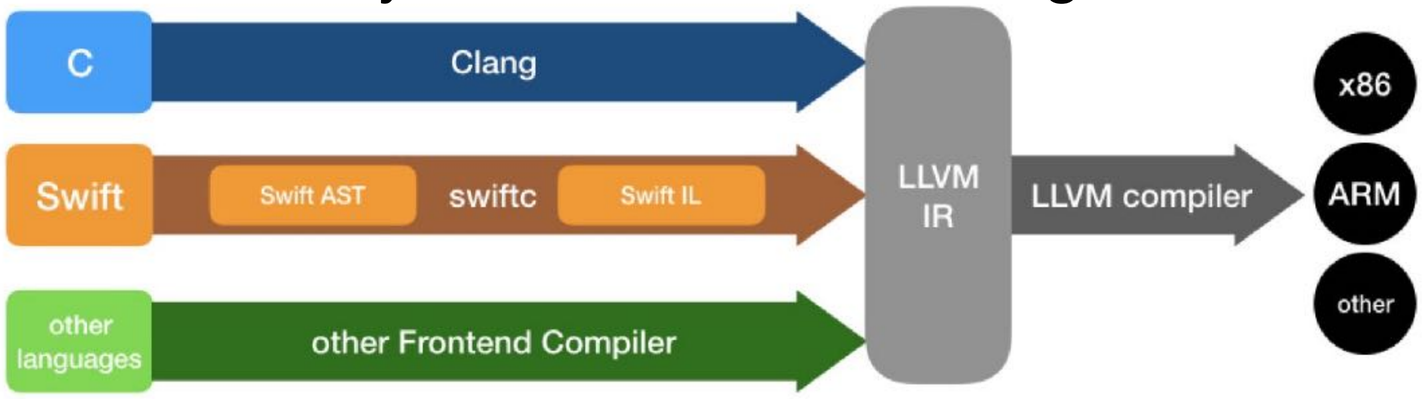
Intermediate Representation

- Main philosophical question: we want to verify a computation is correct
- But **what** is a computation?
- IR in **other contexts**: bytecode, LLVM (language-independent IR), …
- IR in **ZK**: The goal is to **verify** a function, **not** to compute it
- IR: (1) a machine model + (2) how to verify its computation
  1. Machine model: Turing machine, random access machine, circuits
  2. Formalise its verification as a few efficiently verifiable checks using a suitable “proof system”
- Crypto part:



Intermediate Representation

- Main philosophical question: we want to verify a computation is correct
- But **what** is a computation?
- IR in **other contexts**: bytecode, LLVM (language-independent IR), …
- IR in **ZK**: The goal is to **verify** a function, **not** to compute it
- IR: (1) a machine model + (2) how to verify its computation
  1. Machine model: Turing machine, random access machine, circuits
  2. Formalise its verification as a few efficiently verifiable checks using a suitable “proof system”
- Crypto part:
  - implement the checks

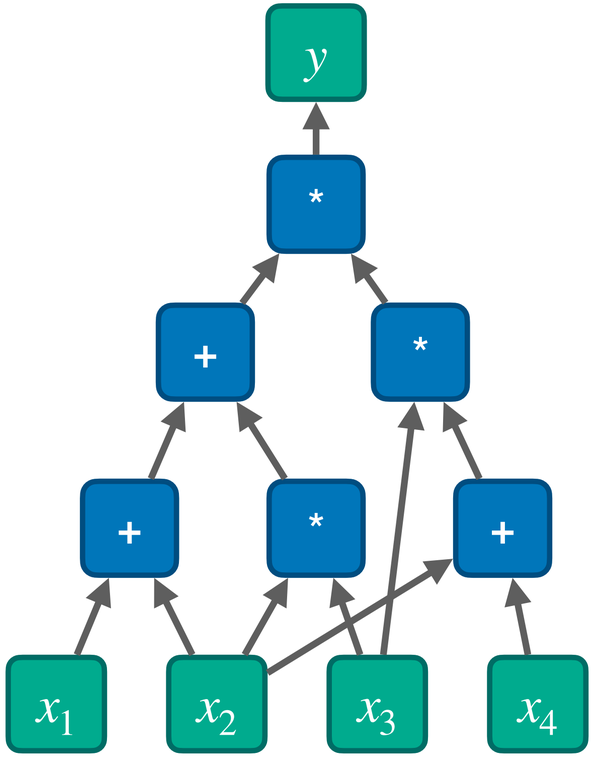


Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**

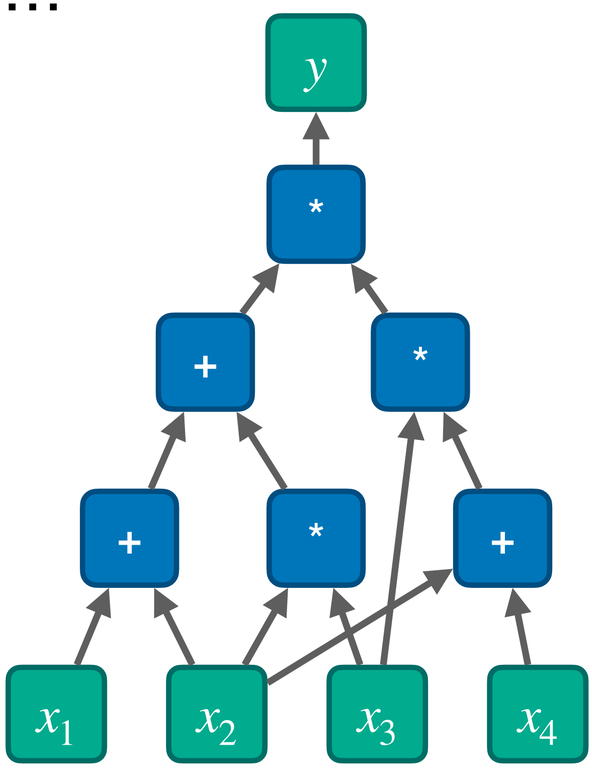
Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**



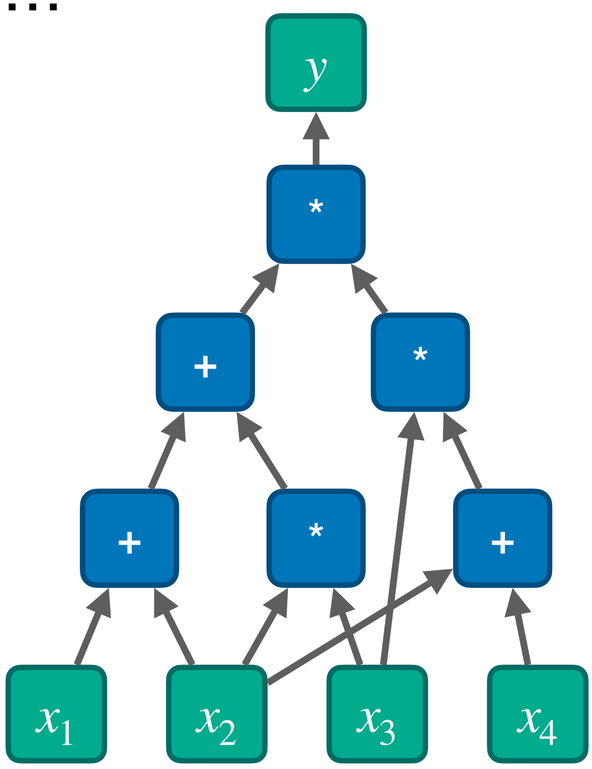
Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**
  - AIR (Arithmetic Intermediate Representation), Plonkish, R1CS, …



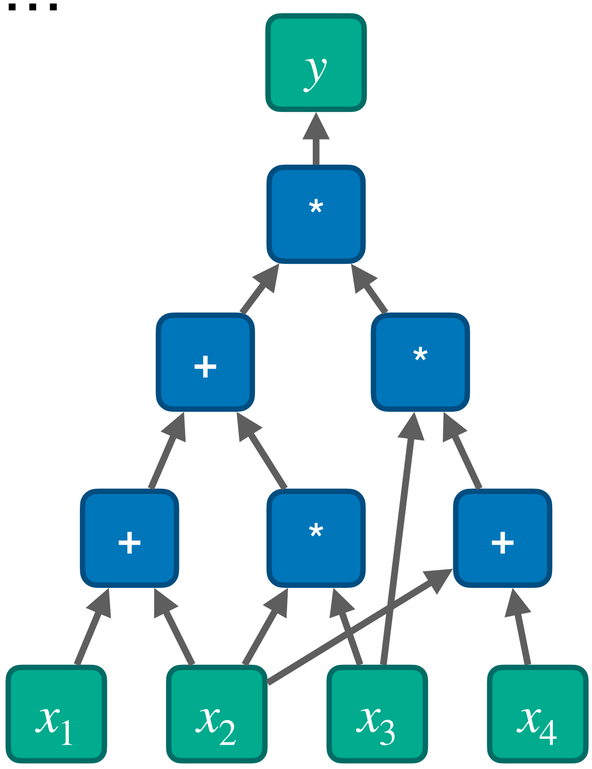
Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**
  - AIR (Arithmetic Intermediate Representation), Plonkish, R1CS, …
  - Model is less efficient ☹️



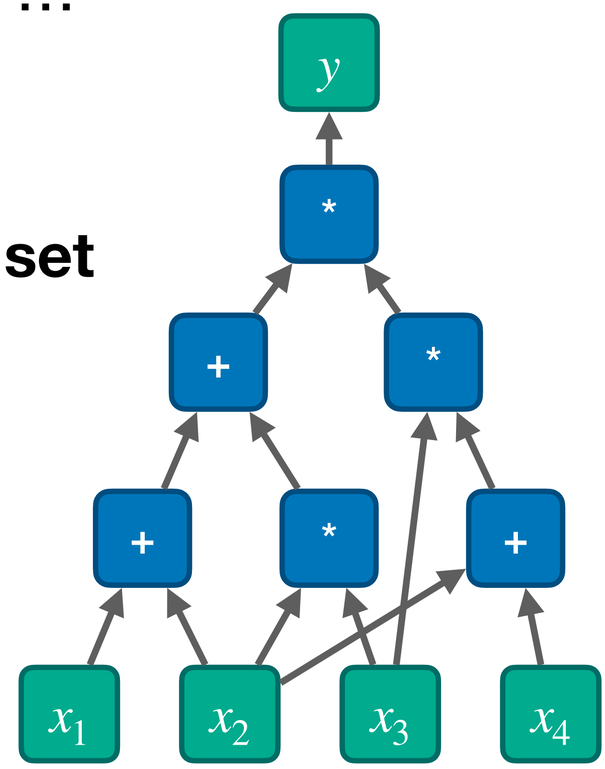
Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**
  - AIR (Arithmetic Intermediate Representation), Plonkish, R1CS, …
  - Model is less efficient ☹️
  - Efficient verification 😀: verify each gate is correctly computed



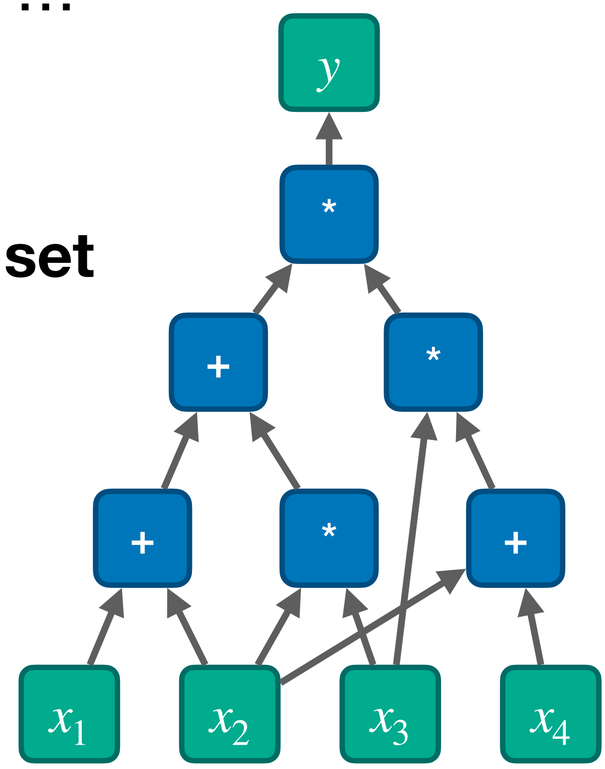
Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**
  - AIR (Arithmetic Intermediate Representation), Plonkish, R1CS, …
  - Model is less efficient ☹️
  - Efficient verification 😀: verify each gate is correctly computed
- **Common solution 2:** implement a **RISC processor instruction set**



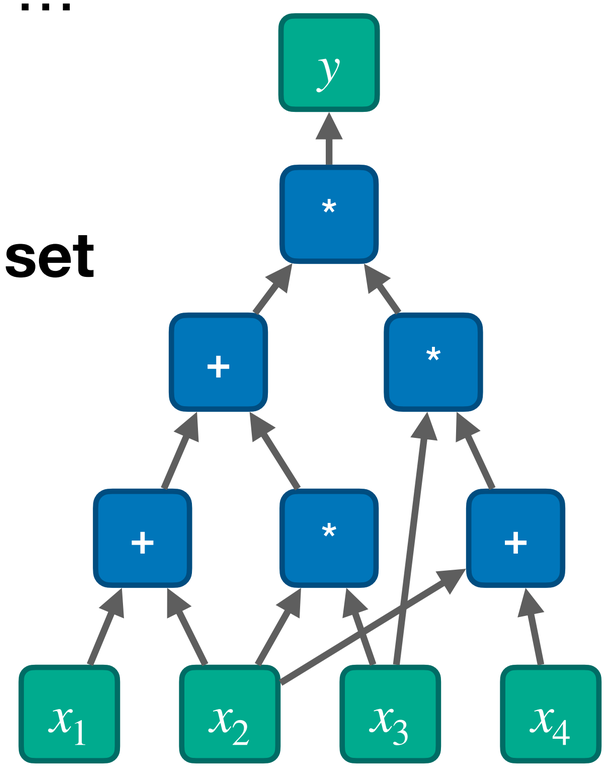
Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**
  - AIR (Arithmetic Intermediate Representation), Plonkish, R1CS, …
  - Model is less efficient ☹️
  - Efficient verification 😀: verify each gate is correctly computed
- **Common solution 2:** implement a **RISC processor instruction set**
  - Model much more efficient 😀



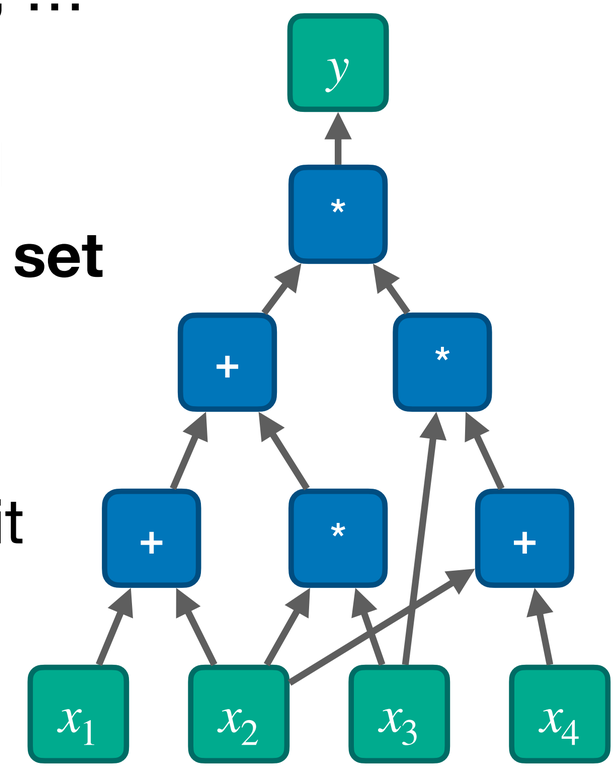
Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**
  - AIR (Arithmetic Intermediate Representation), Plonkish, R1CS, …
  - Model is less efficient ☹️
  - Efficient verification 😀: verify each gate is correctly computed
- **Common solution 2:** implement a **RISC processor instruction set**
  - Model much more efficient 😀
  - Adding verification more costly ☹️ (many ops are global)



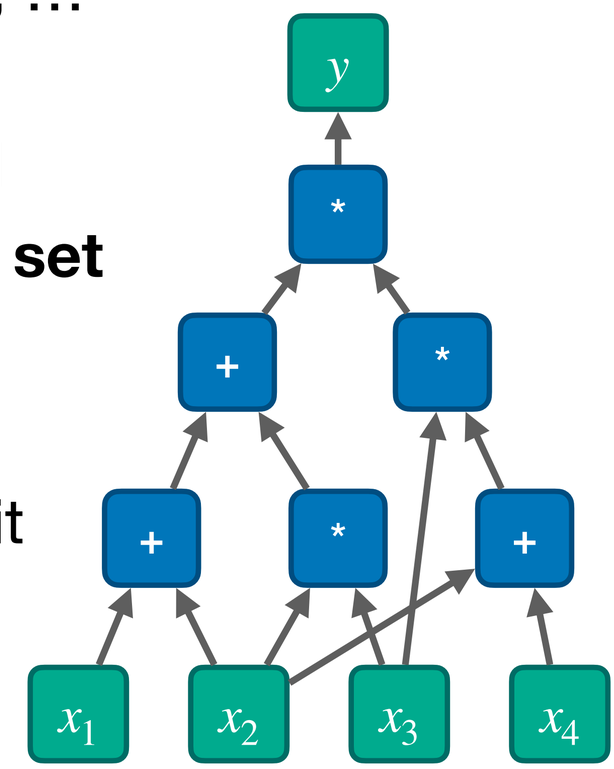
Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**
  - AIR (Arithmetic Intermediate Representation), Plonkish, R1CS, …
  - Model is less efficient ☹️
  - Efficient verification 😀: verify each gate is correctly computed
- **Common solution 2:** implement a **RISC processor instruction set**
  - Model much more efficient 😀
  - Adding verification more costly ☹️ (many ops are global)
- Verifying r**andom memory access** is costly, so many IRs avoid it



Intermediate Representation

- **Tradeoff** between efficiencies of **computation model** and **cryptographic checks**
- **Common solution 1:** some variant of **arithmetic circuits**
  - AIR (Arithmetic Intermediate Representation), Plonkish, R1CS, …
  - Model is less efficient ☹️
  - Efficient verification 😀: verify each gate is correctly computed
- **Common solution 2:** implement a **RISC processor instruction set**
  - Model much more efficient 😀
  - Adding verification more costly ☹️ (many ops are global)
- Verifying r**andom memory access** is costly, so many IRs avoid it
  - However, model is efficient without such access



On Domain-Specific Languages

```rust
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

On Domain-Specific Languages

- The frontend DSL compiles source code to the chosen IR

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

On Domain-Specific Languages

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

On Domain-Specific Languages

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

On Domain-Specific Languages

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

On Domain-Specific Languages

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

On Domain-Specific Languages

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:
  - “**Verification**", not "**computation**" languages

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

On Domain-Specific Languages

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:
  - “**Verification**", not "**computation**" languages
  - **Optimization** depends on IR; certain operations are “unexpectedly” costly

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

On Domain-Specific Languages

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:
  - “**Verification**", not "**computation**" languages
  - **Optimization** depends on IR; certain operations are “unexpectedly” costly
- DSLs change a lot, but are fun

On Domain-Specific Languages

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:
  - “**Verification**", not "**computation**" languages
  - **Optimization** depends on IR; certain operations are “unexpectedly” costly
- DSLs change a lot, but are fun
  - Learn one: then you see how big the intermediate interpretations can be

On Domain-Specific Languages

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:
  - “**Verification**", not "**computation**" languages
  - **Optimization** depends on IR; certain operations are “unexpectedly” costly
- DSLs change a lot, but are fun
  - Learn one: then you see how big the intermediate interpretations can be
    - And how efficient a SNARK you need

On Domain-Specific Languages

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:
  - “**Verification**", not "**computation**" languages
  - **Optimization** depends on IR; certain operations are “unexpectedly” costly
- DSLs change a lot, but are fun
  - Learn one: then you see how big the intermediate interpretations can be
    - And how efficient a SNARK you need
- Importantly, you do not have to sit down and write an IR for your task

On Domain-Specific Languages

```
#![no_std]
#![no_main]

fn fib(n: u32) -> u32 {
    match n {
        0 => 0,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}

#[nexus::main]
fn main() {
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

- The frontend DSL compiles source code to the chosen IR
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:
  - “**Verification**", not "**computation**" languages
  - **Optimization** depends on IR; certain operations are “unexpectedly” costly
- DSLs change a lot, but are fun
  - Learn one: then you see how big the intermediate interpretations can be
    - And how efficient a SNARK you need
- Importantly, you do not have to sit down and write an IR for your task
  - There are tools for it if you know a high-level programming language!

On Domain-Specific Languag

- Collaboration questions:
- How to compile high-level code efficiently to ZK IRs?
- How to formally verify that compilation was correct?
- Design etc of DSLs

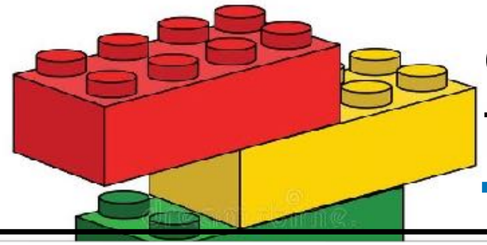
- The frontend DSL compiles source code to the chose
- Many DSLs by now
  - A short list: Cairo, Noir, gsnark, Halo2, Leo, Nexus, …
  - Each has their own limitations
- Different from general purpose languages:
  - “**Verification**", not "**computation**" languages
  - **Optimization** depends on IR; certain operations are “unexpectedly” costly
- DSLs change a lot, but are fun
  - Learn one: then you see how big the intermediate interpretations can be
    - And how efficient a SNARK you need
- Importantly, you do not have to sit down and write an IR for your task
  - There are tools for it if you know a high-level programming language!

```
    let n = 7;
    let result = fib(n);
    assert_eq!(result, 21);
}
```

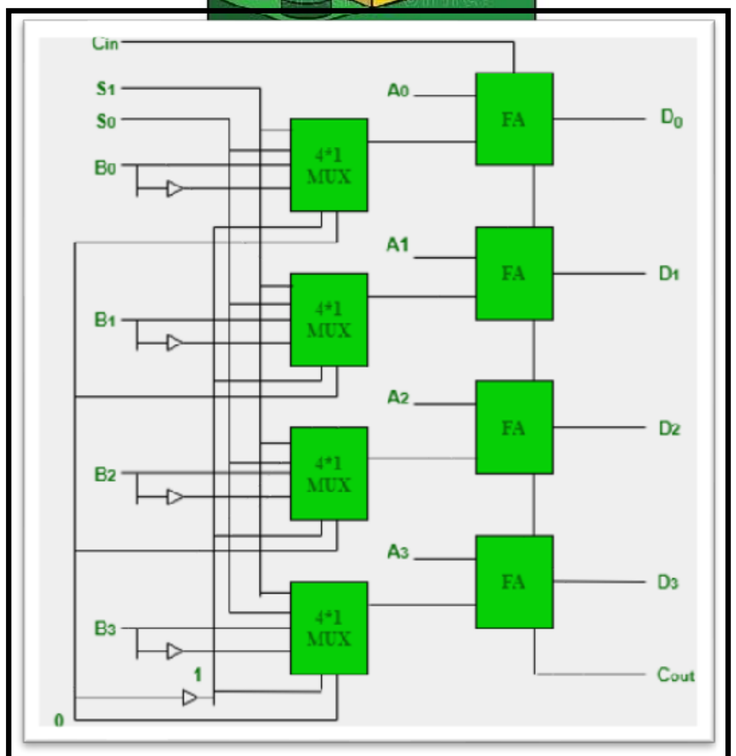
Backend

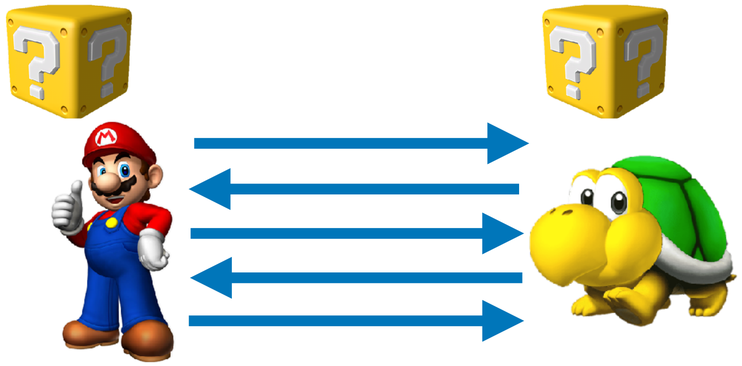
Intermediate Representation

(Polynomial) Interactive Oracle Proof



"Non-cryptographic" techniques



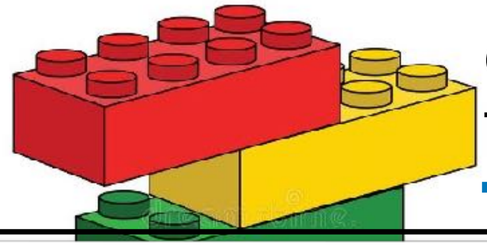


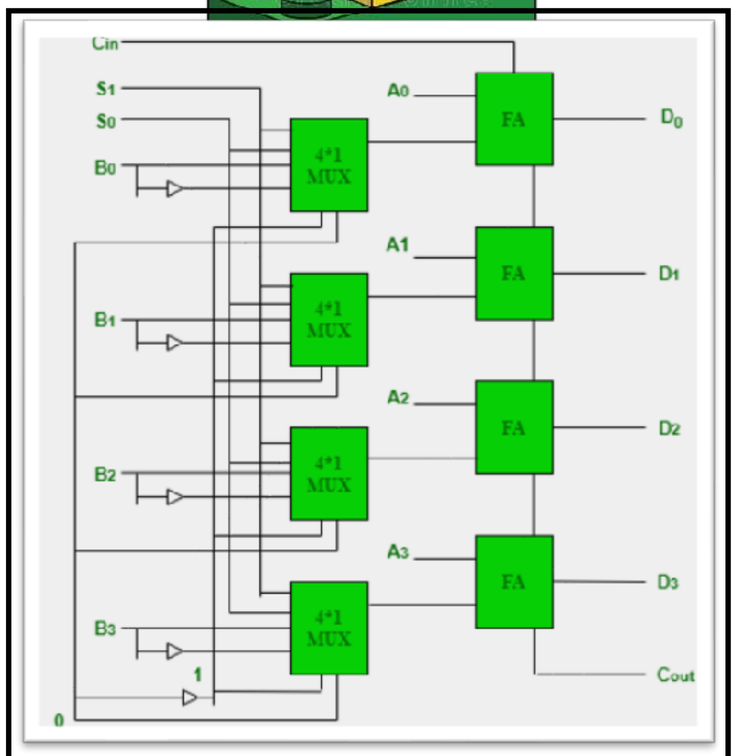
Backend

Intermediate Representation

(Polynomial) Interactive Oracle Proof

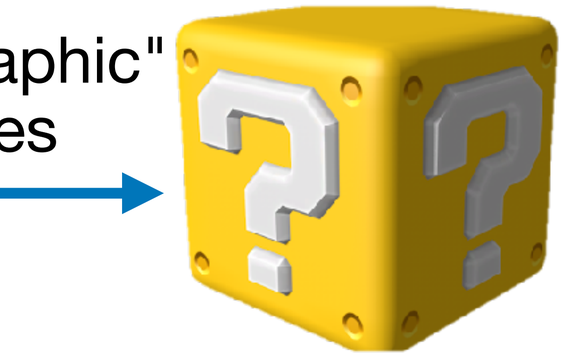
ZK-SNARKs

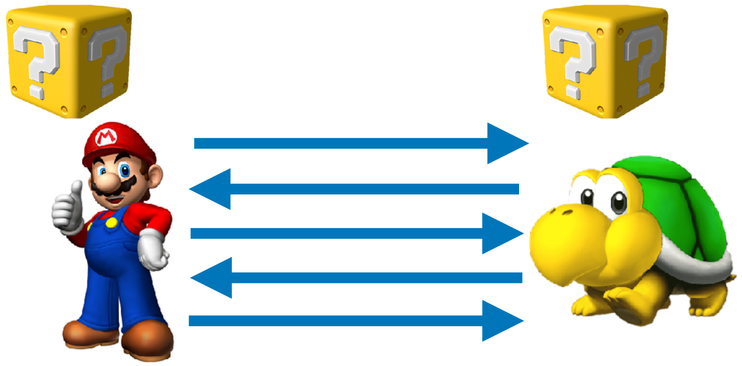




"Non-cryptographic" techniques

Cryptography



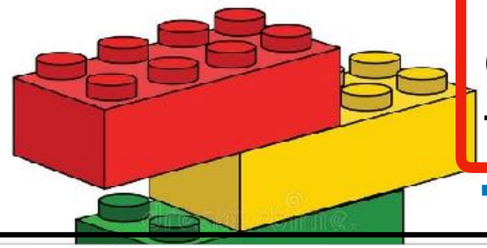


Backend

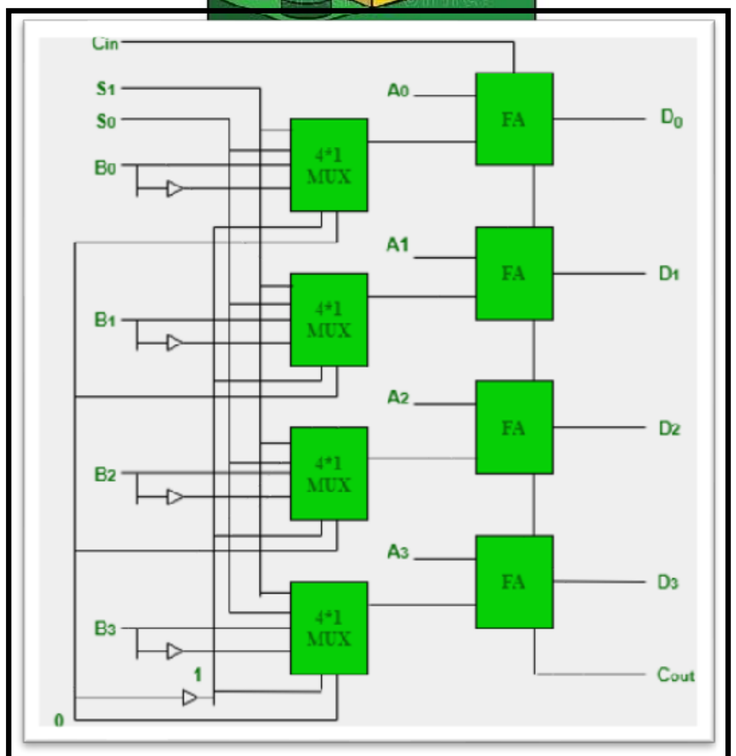
Intermediate Representation

(Polynomial) Interactive Oracle Proof

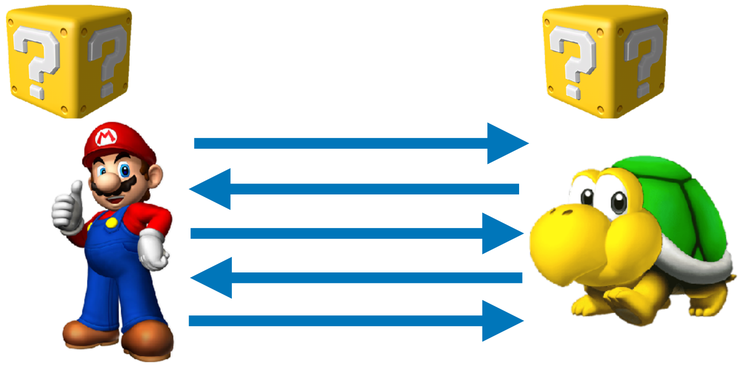
ZK-SNARKs



“Non-cryptographic" techniques



Cryptography



Usual Proof

$x, w = \boldsymbol{a} \in \mathbb{F}^n$

x

Usual Proof

$x, w = \boldsymbol{a} \in \mathbb{F}^n$

$\boldsymbol{a} \in \mathbb{F}^n$

x

Usual Proof

$x, w = a \in \mathbb{F}^n$

$a \in \mathbb{F}^n$

x

Read every bit
Accept/reject

Usual Proof

$x, w = a \in \mathbb{F}^n$

$a \in \mathbb{F}^n$

x

NP: class of languages that have proofs w which can be verified in polynomial time

Read every bit
Accept/reject

Usual Proof

$x, w = a \in \mathbb{F}^n$

$a \in \mathbb{F}^n$

x

NP: class of languages that have proofs w which can be verified in polynomial time

Think of $n = 2^{30}$

Even prover time $n^2 = 2^{60}$ is impenetrable

We want verifier to be much faster than 2^{30}!

Read every bit
Accept/reject

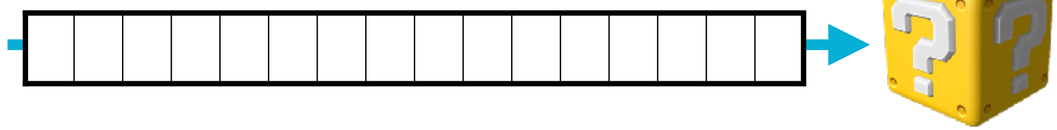
Probabilistically Checkable Proof (~1992)

$x, w = \boldsymbol{a} \in \mathbb{F}^n$

Probabilistically Checkable Proof (~1992)

$x, w = \boldsymbol{a} \in \mathbb{F}^n$

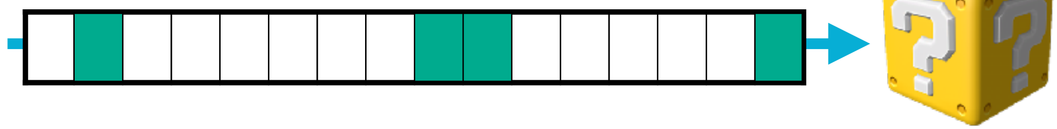
$\boldsymbol{b} = \mathrm{Enc}(\boldsymbol{a}) \in \mathbb{F}^{\ell(n)}$



Probabilistically Checkable Proof (~1992)

$x, w = \boldsymbol{a} \in \mathbb{F}^n$

$\boldsymbol{b} = \mathrm{Enc}(\boldsymbol{a}) \in \mathbb{F}^{\ell(n)}$



V can toss random coins (not secure if V is deterministic)

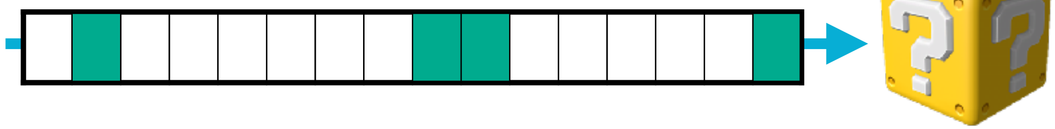
$i \in [1, \ell(n)]$

$\boldsymbol{b}[i]$

Probabilistically Checkable Proof (~1992)

$x, w = \boldsymbol{a} \in \mathbb{F}^n$

$\boldsymbol{b} = \mathrm{Enc}(\boldsymbol{a}) \in \mathbb{F}^{\ell(n)}$



V can toss random coins (not secure if V is deterministic)

$i \in [1, \ell(n)]$

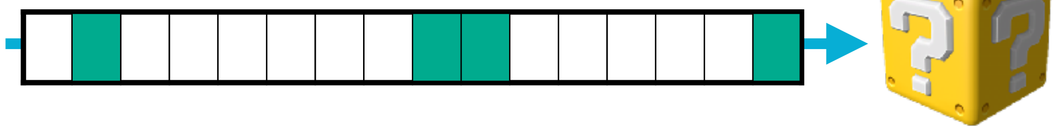
$\boldsymbol{b}[i]$

Accept/reject based on queried bits

Probabilistically Checkable Proof (~1992)

$x, w = \boldsymbol{a} \in \mathbb{F}^n$

$\boldsymbol{b} = \mathrm{Enc}(\boldsymbol{a}) \in \mathbb{F}^{\ell(n)}$



V can toss random coins (not secure if V is deterministic)

$i \in [1, \ell(n)]$

$\boldsymbol{b}[i]$

Accept/reject based on queried bits

Key insight: allowing <u>randomness</u> makes it much more efficient to <u>verify</u>!

PCP Theorem

$x, w = a \in \mathbb{F}^n$

$b = \mathrm{Enc}(a) \in \mathbb{F}^{\ell(n)}$

PCP$[r(n), q(n)]$ - class of languages where proofs can be verified by using $r(n)$ verifier's random bits and $q(n)$ queries

PCP theorem: PCP$[O(\log n), O(1)] = \mathrm{NP}$

- Celebrated as one of the most central theorems in complexity theory

ins (not
inistic)

Accept/reject
based on queried bits

PCP Theorem

$x, w = a \in \mathbb{F}^n$

$b = \mathrm{Enc}(a) \in \mathbb{F}^{\ell(n)}$

ins (not
inistic)

PCP$[r(n), q(n)]$ - class of languages where proofs can be verified by using $r(n)$ verifier's random bits and $q(n)$ queries

PCP theorem: PCP$[O(\log n), O(1)] = \mathrm{NP}$

- Celebrated as one of the most central theorems in complexity theory

😭😭😭😭 Known PCPs are quite inefficient for the prover (proof length $O(n \log^4 n)$)

Accept/reject based on queried bits

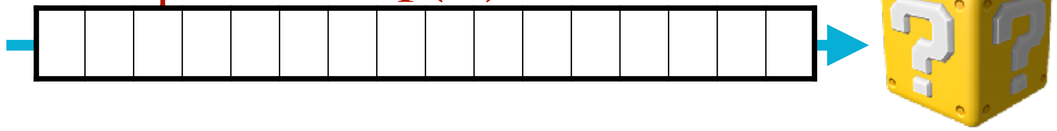
Interactive Oracle Proof (2016)

$x, w = \boldsymbol{a} \in \mathbb{F}^n$

Interactive Oracle Proof (2016)

$x, w = a \in \mathbb{F}^n$

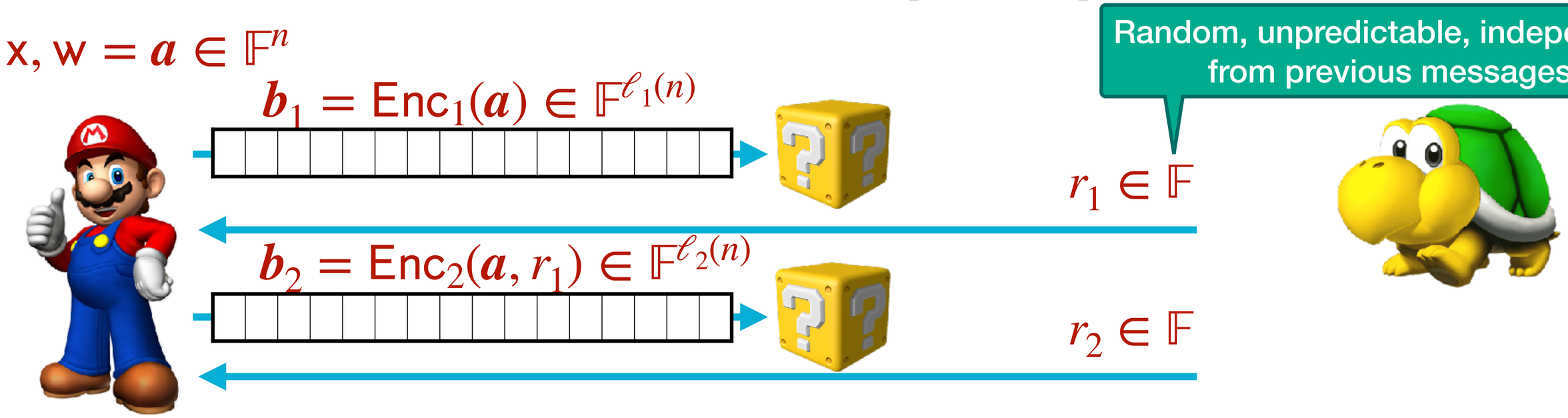
$b_1 = \mathsf{Enc}_1(a) \in \mathbb{F}^{\ell_1(n)}$



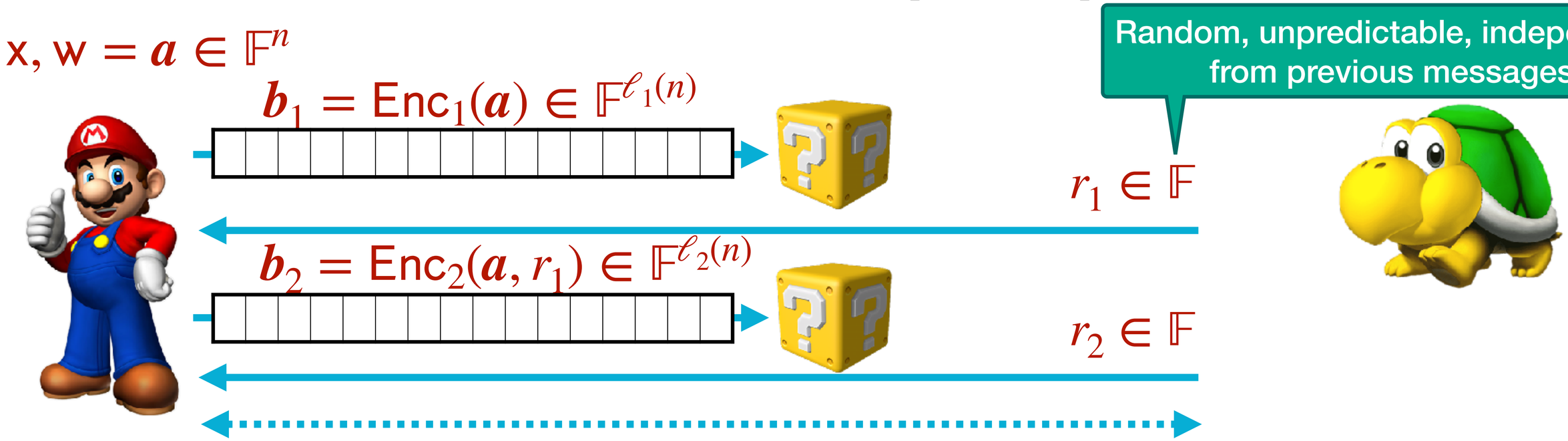
$r_1 \in \mathbb{F}$

Random, unpredictable, independent from previous messages

Interactive Oracle Proof (2016)



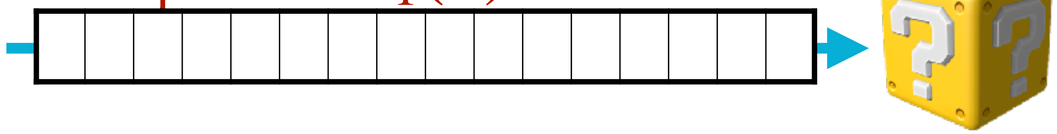
Interactive Oracle Proof (2016)



Interactive Oracle Proof (2016)

$x, w = a \in \mathbb{F}^n$

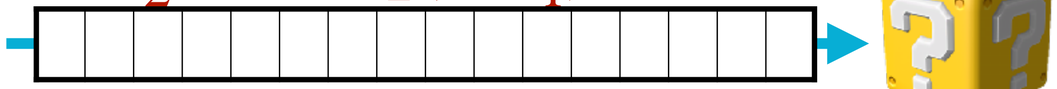
$b_1 = \mathsf{Enc}_1(a) \in \mathbb{F}^{\ell_1(n)}$



Random, unpredictable, independent from previous messages

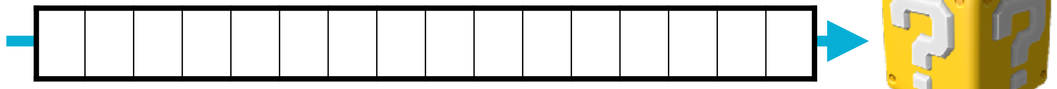
$r_1 \in \mathbb{F}$

$b_2 = \mathsf{Enc}_2(a, r_1) \in \mathbb{F}^{\ell_2(n)}$



$r_2 \in \mathbb{F}$

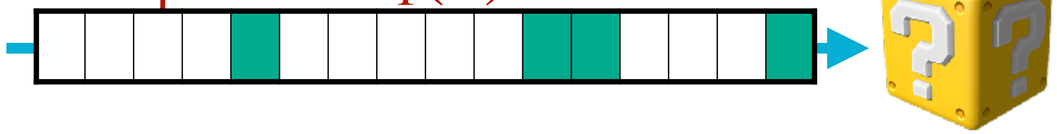
$b_K = \mathsf{Enc}_K(a, r_1, \ldots, r_{K-1}) \in \mathbb{F}^{\ell_K(n)}$



Interactive Oracle Proof (2016)

$x, w = a \in \mathbb{F}^n$

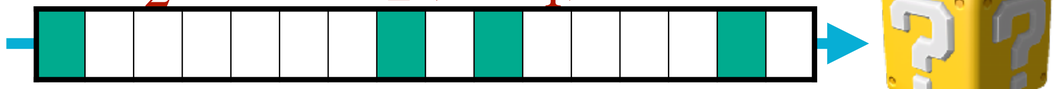
$b_1 = \mathsf{Enc}_1(a) \in \mathbb{F}^{\ell_1(n)}$



Random, unpredictable, independent from previous messages

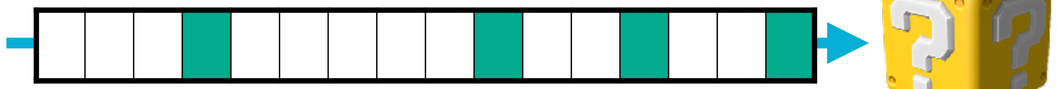
$r_1 \in \mathbb{F}$

$b_2 = \mathsf{Enc}_2(a, r_1) \in \mathbb{F}^{\ell_2(n)}$



$r_2 \in \mathbb{F}$

$b_K = \mathsf{Enc}_K(a, r_1, \ldots, r_{K-1}) \in \mathbb{F}^{\ell_K(n)}$



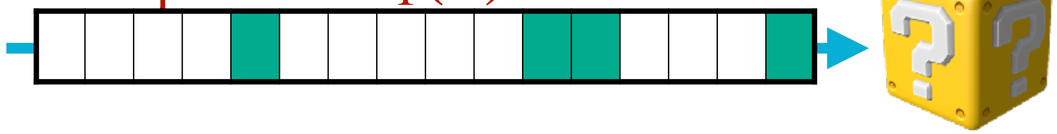
i, j

$b_j[i]$

Interactive Oracle Proof (2016)

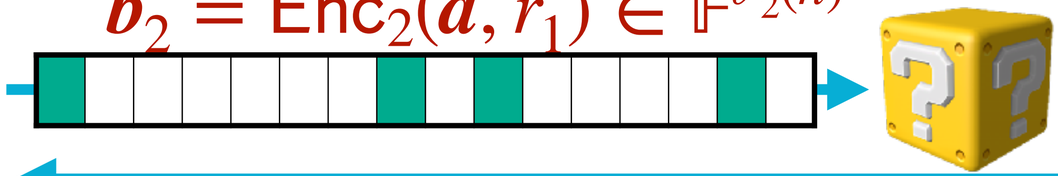
$x, w = a \in \mathbb{F}^n$

$b_1 = \mathsf{Enc}_1(a) \in \mathbb{F}^{\ell_1(n)}$



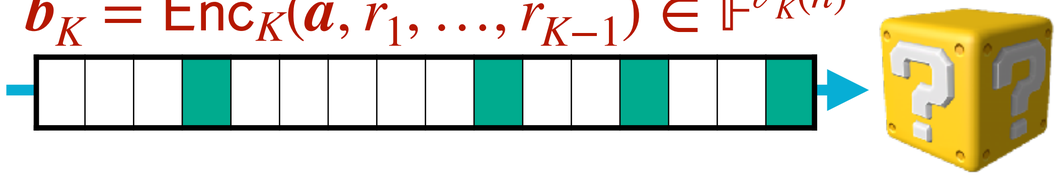
$r_1 \in \mathbb{F}$

$b_2 = \mathsf{Enc}_2(a, r_1) \in \mathbb{F}^{\ell_2(n)}$



$r_2 \in \mathbb{F}$

$b_K = \mathsf{Enc}_K(a, r_1, \ldots, r_{K-1}) \in \mathbb{F}^{\ell_K(n)}$



i, j

$b_j[i]$

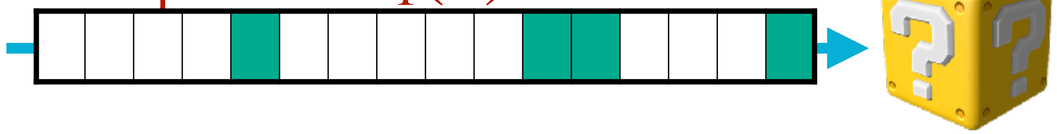
Random, unpredictable, independent from previous messages

Accept/reject based on randomizers and queried bits

Interactive Oracle Proof (2016)

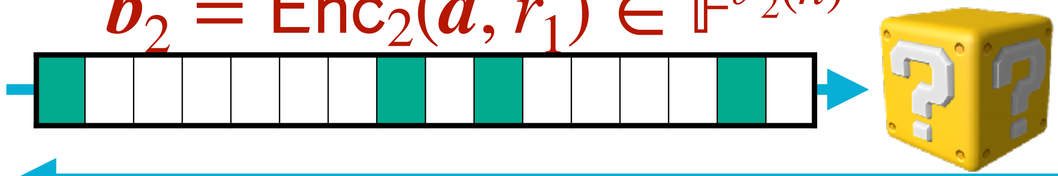
$x, w = a \in \mathbb{F}^n$

$b_1 = \mathsf{Enc}_1(a) \in \mathbb{F}^{\ell_1(n)}$



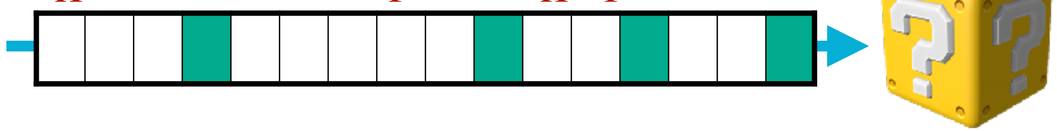
$r_1 \in \mathbb{F}$

$b_2 = \mathsf{Enc}_2(a, r_1) \in \mathbb{F}^{\ell_2(n)}$



$r_2 \in \mathbb{F}$

$b_K = \mathsf{Enc}_K(a, r_1, \ldots, r_{K-1}) \in \mathbb{F}^{\ell_K(n)}$



Random, unpredictable, independent from previous messages

Key insight (1980+): allowing interaction makes it much more efficient to prove and verify

i, j

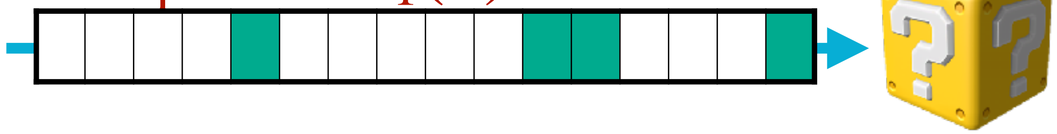
$b_j[i]$

Accept/reject based on randomizers and queried bits

Interactive Oracle Proof (2016)

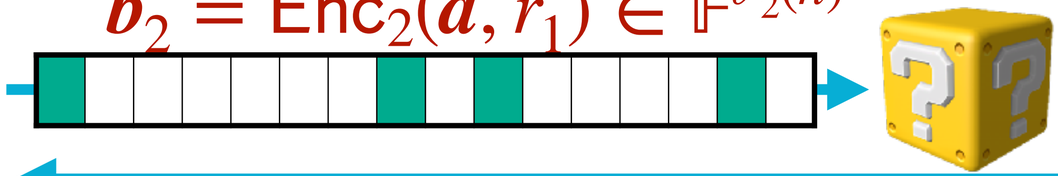
$x, w = a \in \mathbb{F}^n$

$b_1 = \mathsf{Enc}_1(a) \in \mathbb{F}^{\ell_1(n)}$



$r_1 \in \mathbb{F}$

$b_2 = \mathsf{Enc}_2(a, r_1) \in \mathbb{F}^{\ell_2(n)}$



$r_2 \in \mathbb{F}$

$b_K = \mathsf{E}$

Random, unpredictable, independent from previous messages

Key ideas:

- V can spot-check certain coordinates of the encodings
- For efficiency, need a tool to "**smear**" errors
- => Use error-correcting codes

Key insight: <u>interaction</u> makes it much more efficient to <u>prove</u> and <u>verify</u>

$b_j[i]$

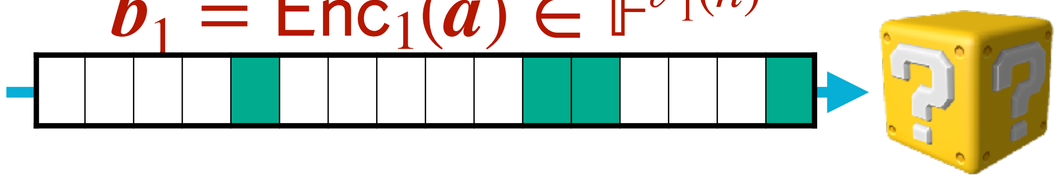
Accept/reject based on randomizers and queried bits

Interactive Oracle Proof (

- Collaboration questions:
- Find good error-correcting codes
- Find out which codes are needed…
- Give input to cryptographers about known coding theory facts

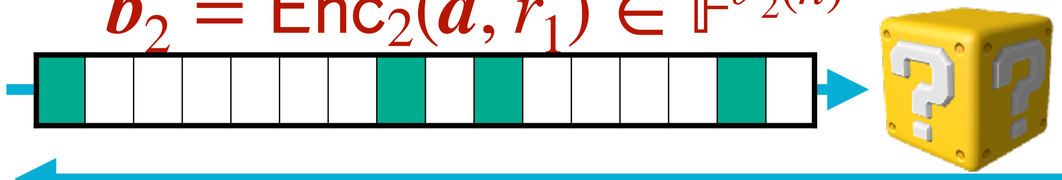
$x, w = a \in \mathbb{F}^n$

$b_1 = \mathrm{Enc}_1(a) \in \mathbb{F}^{\ell_1(n)}$



$r_1 \in \mathbb{F}$

$b_2 = \mathrm{Enc}_2(a, r_1) \in \mathbb{F}^{\ell_2(n)}$



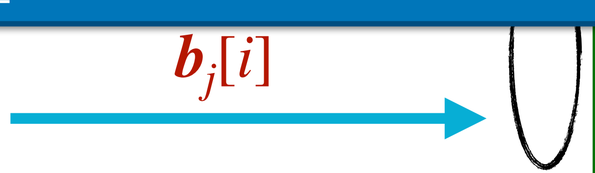
$r_2 \in \mathbb{F}$

$b_K = \mathrm{E}$

Key ideas:
- V can spot-check certain coordinates of the encodings
- For efficiency, need a tool to "**smear**" errors
- => Use error-correcting codes

Key insight: **interaction** makes it much more efficient to **prove** and **verify**

$b_j[i]$

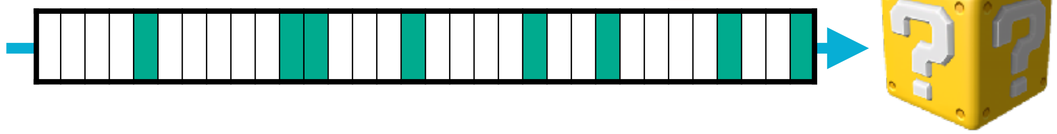


Accept/reject based on randomizers and queried bits

Polynomial Interactive Oracle Proof (2020)

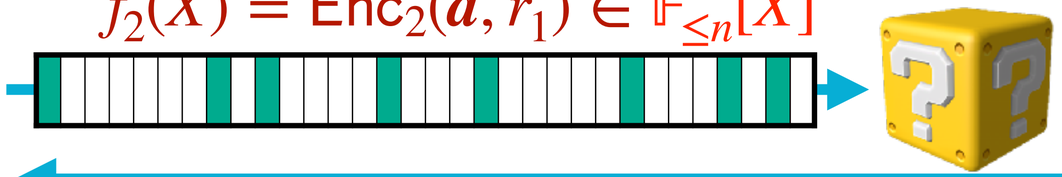
$x, w = \boldsymbol{a} \in \mathbb{F}^n$

$f_1(X) = \mathsf{Enc}_1(\boldsymbol{a}) \in \mathbb{F}_{\leq n}[X]$



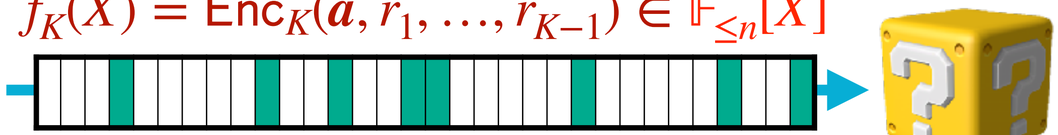
$r_1 \in \mathbb{F}$

$f_2(X) = \mathsf{Enc}_2(\boldsymbol{a}, r_1) \in \mathbb{F}_{\leq n}[X]$



$r_2 \in \mathbb{F}$

$f_K(X) = \mathsf{Enc}_K(\boldsymbol{a}, r_1, \ldots, r_{K-1}) \in \mathbb{F}_{\leq n}[X]$



i, j

$f_j(i)$

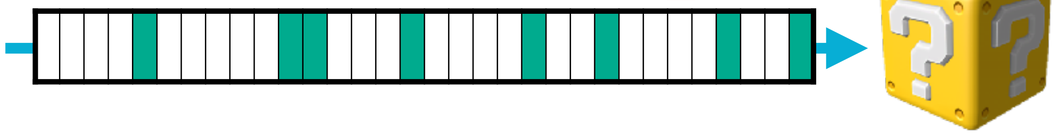
Random, unpredictable, independent from previous messages

Accept/reject based on randomizers and queried evaluations

Polynomial Interactive Oracle Proof (2020)

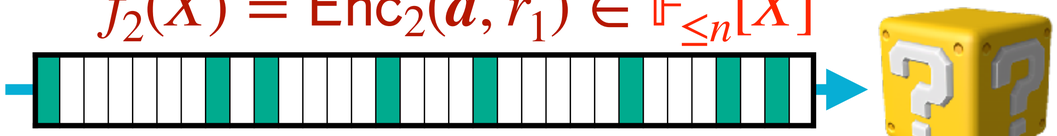
$x, w = \boldsymbol{a} \in \mathbb{F}^n$

$f_1(X) = \mathsf{Enc}_1(\boldsymbol{a}) \in \mathbb{F}_{\leq n}[X]$



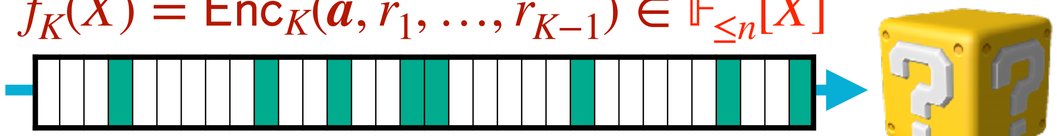
$r_1 \in \mathbb{F}$

$f_2(X) = \mathsf{Enc}_2(\boldsymbol{a}, r_1) \in \mathbb{F}_{\leq n}[X]$



$r_2 \in \mathbb{F}$

$f_K(X) = \mathsf{Enc}_K(\boldsymbol{a}, r_1, \ldots, r_{K-1}) \in \mathbb{F}_{\leq n}[X]$



Random, unpredictable, independent from previous messages

$|\mathbb{F}|$ possible queries i instead of just n

i, j

$f_j(i)$

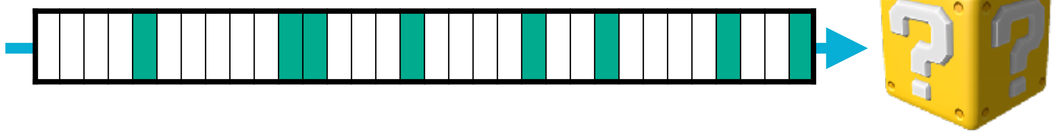
Accept/reject based on randomizers and queried evaluations

Key insight (trade-off): allowing a more powerful oracle makes IOP more efficient but implementing the oracle is more costly

Polynomial Interactive Oracle Proof (2020)

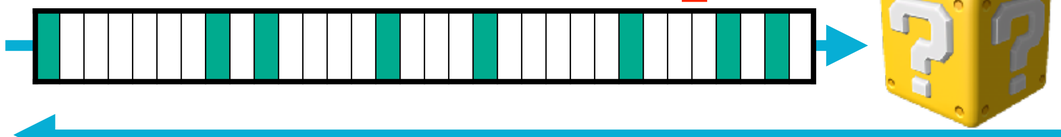
$x, w = a \in \mathbb{F}^n$

$f_1(X) = \mathsf{Enc}_1(a) \in \mathbb{F}_{\leq n}[X]$



$r_1 \in \mathbb{F}$

$f_2(X) = \mathsf{Enc}_2(a, r_1) \in \mathbb{F}_{\leq n}[X]$

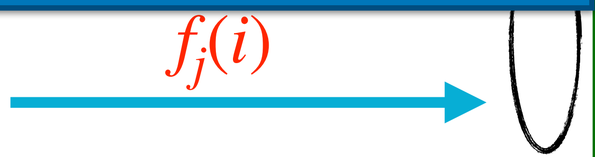


$r_2 \in \mathbb{F}$

$f_K(X) = \mathsf{Enc}_K(\ldots) \in \mathbb{F}_{\leq n}[X]$

Random, unpredictable, independent from previous messages

$f_j(i)$



s i instead of just n

Accept/reject based on randomizers and queried evaluations

Key ideas:

- V can query polynomials at random points
- Since low-degree polynomials have few roots, random evaluation = 0 $\Longrightarrow$ polynomial = 0 w.h.p

Key insight (tra… more powerful … more efficient but implementing the oracle is more costly

Insufficiency

- PIOPs are **idealized** protocols

Insufficiency

- PIOPs are **idealized** protocols
- You need to **trust** that the oracles function correctly

Insufficiency

- PIOPs are **idealized** protocols
- You need to **trust** that the oracles function correctly
- The next step in zk-SNARK design: **instantiating** the boxes

Insufficiency

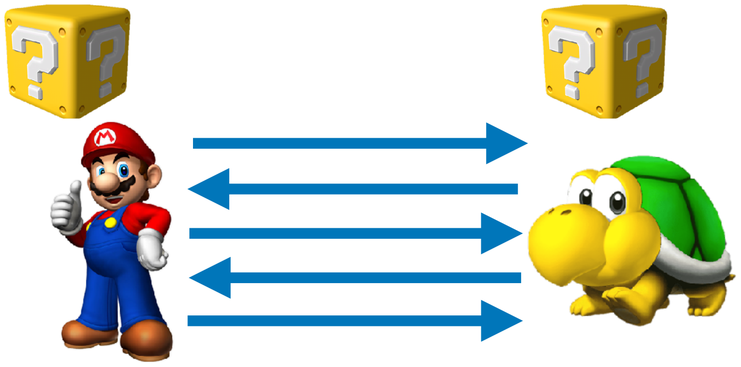
- PIOPs are **idealized** protocols
- You need to **trust** that the oracles function correctly
- The next step in zk-SNARK design: **instantiating** the boxes
  - Instantiating makes protocols less efficient

Insufficiency

- PIOPs are **idealized** protocols
- You need to **trust** that the oracles function correctly
- The next step in zk-SNARK design: **instantiating** the boxes
  - Instantiating makes protocols less efficient
  - … and also **only** computationally secure

Backend

(Polynomial) Interactive Oracle Proof

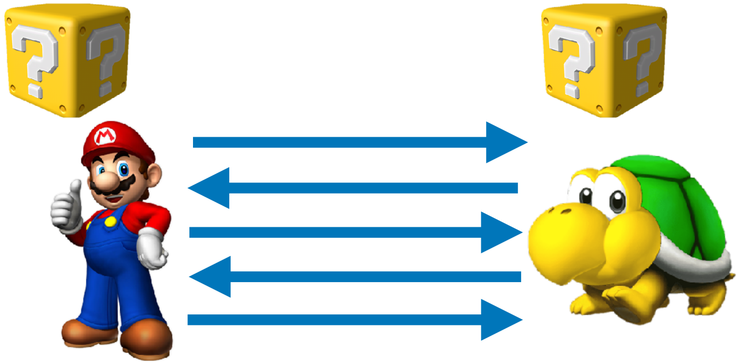


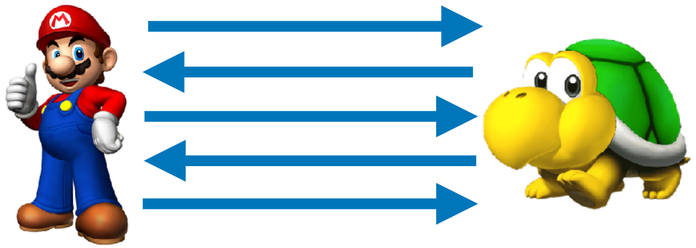
Backend

(Polynomial) Interactive Oracle Proof

Interactive Protocols

Commitment





Backend

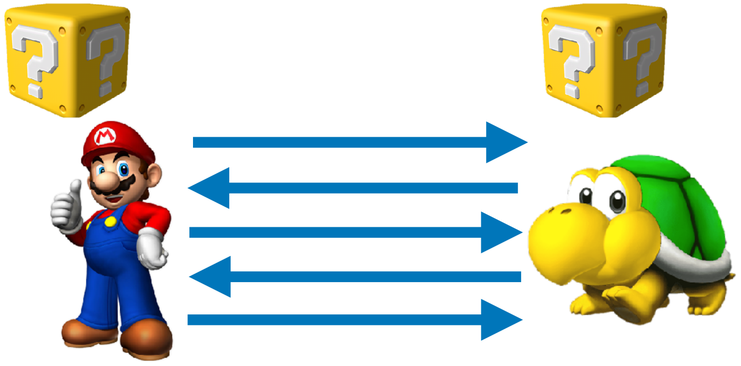
(Polynomial) Interactive Oracle Proof

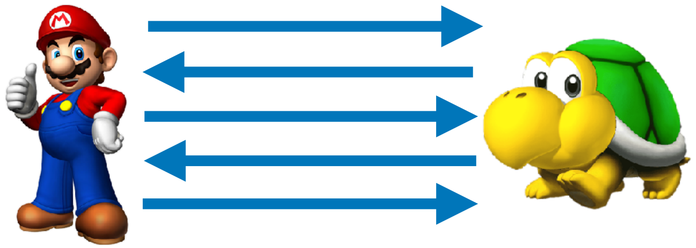
Commitment

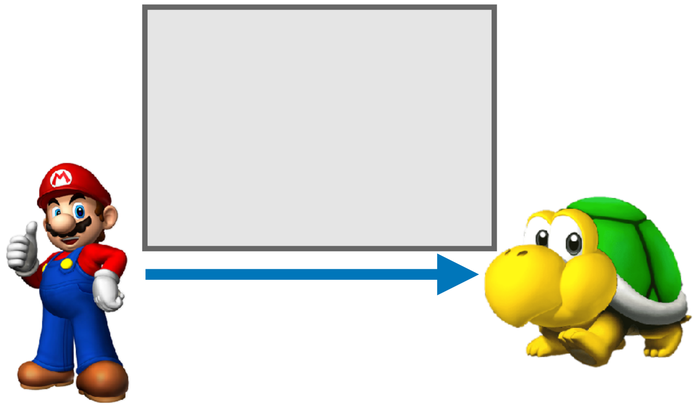
Interactive Protocols

Fiat-Shamir

ZK-SNARKs







Backend

(Polynomial) Interactive Oracle Proof

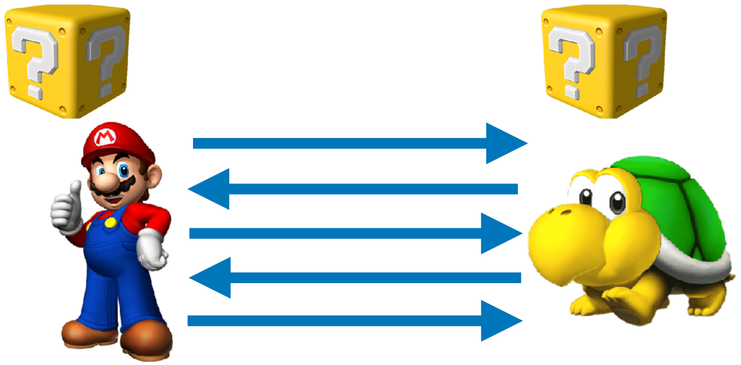
Commitment

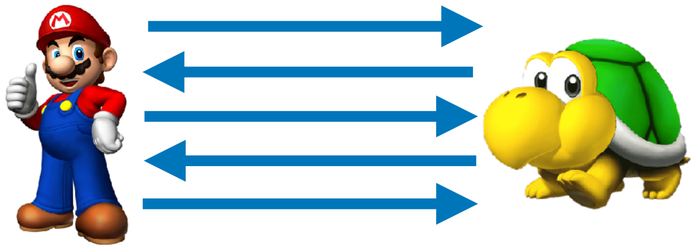
Interactive Protocols

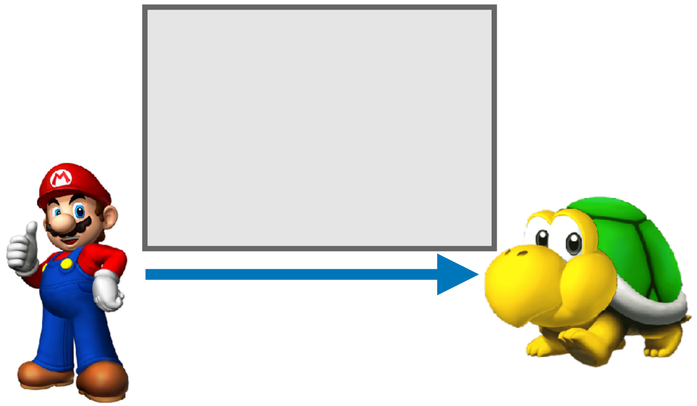
We will not discuss FS today

Fiat-Shamir

ZK-SNARKs







Vector And Polynomial Commitment Scheme

- **VCS:** Real-life instantiation of the **IOP** black box

Vector And Polynomial Commitment Scheme

- **VCS:** Real-life instantiation of the **IOP** black box
- Security definitions of **VCS**, **IOP** are chosen so that combining a secure **VCS** with a secure **IOP** results in a secure zk-SNARK

Vector And Polynomial Commitment Scheme

- **VCS:** Real-life instantiation of the **IOP** black box
- Security definitions of **VCS**, **IOP** are chosen so that combining a secure **VCS** with a secure **IOP** results in a secure zk-SNARK
- Efficiency depends both on the efficiency of the **IOP** and the **VCS**

Vector And Polynomial Commitment Scheme

- **VCS:** Real-life instantiation of the **IOP** black box
- Security definitions of **VCS**, **IOP** are chosen so that combining a secure **VCS** with a secure **IOP** results in a secure zk-SNARK
- Efficiency depends both on the efficiency of the **IOP** and the **VCS**
- **VCS** gives most of the structural flavour to resulting zk-SNARKs:

Vector And Polynomial Commitment Scheme

- **VCS:** Real-life instantiation of the **IOP** black box
- Security definitions of **VCS**, **IOP** are chosen so that combining a secure **VCS** with a secure **IOP** results in a secure zk-SNARK
- Efficiency depends both on the efficiency of the **IOP** and the **VCS**
- **VCS** gives most of the structural flavour to resulting zk-SNARKs:
  - Security assumptions (post-quantum?)

Vector And Polynomial Commitment Scheme

- **VCS:** Real-life instantiation of the **IOP** black box
- Security definitions of **VCS**, **IOP** are chosen so that combining a secure **VCS** with a secure **IOP** results in a secure zk-SNARK
- Efficiency depends both on the efficiency of the **IOP** and the **VCS**
- **VCS** gives most of the structural flavour to resulting zk-SNARKs:
  - Security assumptions (post-quantum?)
  - Trusted parameters

Vector And Polynomial Commitment Scheme

- **VCS:** Real-life instantiation of the **IOP** black box
- Security definitions of **VCS**, **IOP** are chosen so that combining a secure **VCS** with a secure **IOP** results in a secure zk-SNARK
- Efficiency depends both on the efficiency of the **IOP** and the **VCS**
- **VCS** gives most of the structural flavour to resulting zk-SNARKs:
  - Security assumptions (post-quantum?)
  - Trusted parameters

- **PCS:** Real-life instantiation of the **PIOP** black box
- Security definitions of **PCS**, **PIOP** are chosen so that combining a secure **PCS** with a secure **PIOP** results in a secure zk-SNARK
- Efficiency depends both on the efficiency of the **PIOP** and the **PCS**
- **PCS** gives most of the structural flavour to resulting zk-SNARKs:
  - Security assumptions (post-quantum?)
  - Trusted parameters

Vector And Polynomial Commitment Scheme

- **VCS:** Real-life instantiation of the **IOP** black box
- Security definitions of **VCS**, **IOP** are chosen so that combining a secure **VCS** with a secure **IOP** results in a secure zk-SNARK
- Efficiency depends both on the efficiency of the **IOP** and the **VCS**
- **VCS** gives most of the structural flavour to resulting zk-SNARKs:
  - Security assumptions (post-quantum?)
  - Trusted parameters

- **PCS:** Real-life instantiation of the **PIOP** black box
- Security definitions of **PCS**, **PIOP** are chosen so that combining a secure **PCS** with a secure **PIOP** results in a secure zk-SNARK
- Efficiency depends both on the efficiency of the **PIOP** and the **PCS**
- **PCS** gives most of the structural flavour to resulting zk-SNARKs:
  - Security assumptions (post-quantum?)
  - Trusted parameters

Security assumptions and trusted parameters depend only on the VCS/PCS

Current Main "Cryptographic" Approaches

	Assumption	Post-Quantum	Trusted para-meters	Prover speed	Verifier speed	Argument length	Examples
Pairing-based PCS + PIOP	Various pairing-based (elliptic curve) assumptions	No	Yes	O(n log n) heavy operations	Small constant number of heavy operations (a few milliseconds)	Very short (<600 B for any computation)	Plonk, Marlin, Groth16, Polymath, Lunar, Basilisk, Vampire, Spartan, HyperPlonk, BabySpartan, ….
Hash-based VCS + IOP	A secure hash function (CRHF)	Yes	Minimal	O(n log n) or even O(n) simple operations	Large number of simple operations New schemes (WHIR) have an efficient verifier	Long (50 KB-500KB)	FRI, STIR, WHIR, Brakedown, Binius, Orion, Ligero, BaseFold

Current Main "Cryptographic" Approaches

More efficient IOP part, a lot of algebra

	Assumption	Post-Quantum	Trusted para-meters	Prover speed	Verifier speed	Argument length	Examples
Pairing-based PCS + PIOP	Various pairing-based (elliptic curve) assumptions	No	Yes	$O(n \log n)$ heavy operations	Small constant number of heavy operations (a few milliseconds)	Very short (<600 B for **any** computation)	**Plonk**, Marlin, **Groth16**, Polymath, Lunar, Basilisk, Vampire, Spartan, HyperPlonk, **BabySpartan**, ….
Hash-based VCS + IOP	A secure hash function (CRHF)	Yes	Minimal	$O(n \log n)$ or even $O(n)$ simple operations	Large number of simple operations **New schemes (WHIR) have an efficient verifier**	Long (50 KB-500KB)	**FRI**, STIR, **WHIR**, Brakedown, **Binius**, Orion, Ligero, BaseFold

Current Main "Cryptographic" Approaches

More efficient IOP part, a lot of algebra

	Assumption	Post-Quantum	Trusted para-meters	Prover speed	Verifier speed	Argument length	Examples
Pairing-based PCS + PIOP	Various pairing-based (elliptic curve) assumptions	No	Yes	$O(n \log n)$ heavy operations	Small constant number of heavy operations (a few milliseconds)	Very short (<600 B for **any** computation)	**Plonk**, Marlin, **Groth16**, Polymath, Lunar, Basilisk, Vampire, Spartan, HyperPlonk, **BabySpartan**, ….
Hash-based VCS + IOP	A secure hash function (CRHF)	Yes	Minimal	$O(n \log n)$ or even $O(n)$ simple operations	Large number of simple operations **New schemes (WHIR) have an efficient verifier**	Long (50 KB-500KB)	**FRI**, STIR, **WHIR**, Brakedown, **Binius**, Orion, Ligero, BaseFold

More efficient CS part, no algebra at all

Current Main "Cryptographic" Approaches

Other approaches exist but are **currently** more experimental (lattice-based, …)

More efficient IOP part, a lot of algebra

	Assumption	Post-Quantum	Trusted para-meters	Prover speed	Verifier speed	Argument length	Examples
Pairing-based PCS + PIOP	Various pairing-based (elliptic curve) assumptions	No	Yes	$O(n \log n)$ heavy operations	Small constant number of heavy operations (a few milliseconds)	Very short (<600 B for **any** computation)	**Plonk**, Marlin, **Groth16**, Polymath, Lunar, Basilisk, Vampire, Spartan, HyperPlonk, **BabySpartan**, ….
Hash-based VCS + IOP	A secure hash function (CRHF)	Yes	Minimal	$O(n \log n)$ or even $O(n)$ simple operations	Large number of simple operations **New schemes (WHIR) have an efficient verifier**	Long (50 KB-500KB)	**FRI**, STIR, **WHIR**, Brakedown, **Binius**, Orion, Ligero, BaseFold

More efficient CS part, no algebra at all

Current Main "Cryptographic" Approaches

Other approaches exist but are **currently** more experimental (lattice-based, …)

More efficient IOP part, a lot of algebra

Inherent trade-off between efficiency and trusted parameters/assumptions

	Assumption	Post-Quantum	Trusted para-meters	Prover speed	Verifier speed	Argument length	Examples
Pairing-based PCS + PIOP	Various pairing-based (elliptic curve) assumptions	No	Yes	*O*(*n* log *n*) heavy operations	Small constant number of heavy operations (a few milliseconds)	Very short (<600 B for **any** computation)	**Plonk**, Marlin, **Groth16**, Polymath, Lunar, Basilisk, Vampire, Spartan, HyperPlonk, **BabySpartan**, ….
Hash-based VCS + IOP	A secure hash function (CRHF)	Yes	Minimal	*O*(*n* log *n*) or even *O*(*n*) simple operations	Large number of simple operations **New schemes (WHIR) have an efficient verifier**	Long (50 KB-500KB)	**FRI**, STIR, **WHIR**, Brakedown, **Binius**, Orion, Ligero, BaseFold

More efficient CS part, no algebra at all

Background Material

- **ZK MOOC:** https://zk-learning.org/

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC’s Discord server (many from industry)

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**
  - https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**
  - https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html
  - Freelly available

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**
  - https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html
  - Freelly available
  - 2023 Spring: uptodate at that moment

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**
  - https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html
  - Freelly available
  - 2023 Spring: uptodate at that moment
  - Meant for non-cryptographers (but related background helps)

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**
  - https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html
  - Freelly available
  - 2023 Spring: uptodate at that moment
  - Meant for non-cryptographers (but related background helps)
- **Alessandro Chiesa and Elon Yogev's book:**

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**
  - https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html
  - Freelly available
  - 2023 Spring: uptodate at that moment
  - Meant for non-cryptographers (but related background helps)
- **Alessandro Chiesa and Elon Yogev's book:**
  - https://snargsbook.org/

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**
  - https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html
  - Freelly available
  - 2023 Spring: uptodate at that moment
  - Meant for non-cryptographers (but related background helps)
- **Alessandro Chiesa and Elon Yogev's book:**
  - https://snargsbook.org/
  - 2024, modern, and very technical

Background Material

- **ZK MOOC:** https://zk-learning.org/
  - Top presenters
  - 2023 Spring: uptodate at that moment
  - More than 2000 people on MOOC's Discord server (many from industry)
- **Justin Thaler's book:**
  - https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html
  - Freelly available
  - 2023 Spring: uptodate at that moment
  - Meant for non-cryptographers (but related background helps)
- **Alessandro Chiesa and Elon Yogev's book:**
  - https://snargsbook.org/
  - 2024, modern, and very technical
  - Specialized to cover hash-based zk-SNARKs and only their foundations

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/
- **Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra**

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/
- **Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra**
  - Top book about efficient computer algebra

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/
- **Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra**
  - Top book about efficient computer algebra
  - Various algorithms for fast arithmetic, polynomials, …, ending with factorisation

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/
- **Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra**
  - Top book about efficient computer algebra
  - Various algorithms for fast arithmetic, polynomials, …, ending with factorisation
  - Relevant for this course: polynomial multiplication and division, FFT, …

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/
- **Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra**
  - Top book about efficient computer algebra
  - Various algorithms for fast arithmetic, polynomials, …, ending with factorisation
  - Relevant for this course: polynomial multiplication and division, FFT, …
- **Nadia El Mrabet, Marc Joye: Guide to Pairing-Based Cryptography**

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/
- **Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra**
  - Top book about efficient computer algebra
  - Various algorithms for fast arithmetic, polynomials, …, ending with factorisation
  - Relevant for this course: polynomial multiplication and division, FFT, …
- **Nadia El Mrabet, Marc Joye: Guide to Pairing-Based Cryptography**
  - If you need to know more about elliptic curves & pairings

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/
- **Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra**
  - Top book about efficient computer algebra
  - Various algorithms for fast arithmetic, polynomials, …, ending with factorisation
  - Relevant for this course: polynomial multiplication and division, FFT, …
- **Nadia El Mrabet, Marc Joye: Guide to Pairing-Based Cryptography**
  - If you need to know more about elliptic curves & pairings
- **ZK Podcast: https://zeroknowledge.fm/**

Background Material

- **MoonMath Manual: // background math for zk-SNARKs**
  - https://leastauthority.com/community-matters/moonmath-manual/
- **Joachim von zur Gathen, Jürgen Gerhard: Modern Computer Algebra**
  - Top book about efficient computer algebra
  - Various algorithms for fast arithmetic, polynomials, …, ending with factorisation
  - Relevant for this course: polynomial multiplication and division, FFT, …
- **Nadia El Mrabet, Marc Joye: Guide to Pairing-Based Cryptography**
  - If you need to know more about elliptic curves & pairings
- **ZK Podcast: https://zeroknowledge.fm/**
  - Often top researchers talking about the most recent work

Background Material

Background Material

- **Thousands of YouTube videos**

Background Material

- **Thousands of YouTube videos**
  - Just search for anything (for example, “GKR")

Background Material

- **Thousands of YouTube videos**
  - Just search for anything (for example, “GKR")
  - But ask me for who the good presenters are

Important References

- (Usual proofs:) Euclid. *Elements* (~300 BC)
- PCP:
  - Sanjeev Arora, Shmuel Safra. **Probabilistic checking of proofs: A new characterization of NP** (1998)
  - Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, Mario Szegedy. **Proof verification and the hardness of approximation problems** (1998)
- IOP:
  - Eli Ben-Sasson, Alessandro Chiesa, Nicholas Spooner. **Interactive Oracle Proofs** (2016)
  - Omer Reingold, Guy N. Rothblum, Ron D. Rothblum. **Constant-round interactive proofs for delegating computation** (2016)

Important References

- PIOP:
  - Benedikt Bünz, Ben Fisch, Alan Szepieniec. **Transparent SNARKs from DARK compilers** (2020)
  - Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, Nicholas Ward. **Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS** (2020)
- PCS:
  - Aniket Kate, Gregory M. Zaverucha, Ian Goldberg. **Constant-Size Commitments to Polynomials and Their Applications** (2010)
- (Definition of interactive protocols and ZK) Shafi Goldwasser, Silvio Micali, Charles Rackoff: **The Knowledge Complexity of Interactive Proof-Systems.** STOC 1985: 291-304
- Noir: https://noir-lang.org/