Advancing Scalability in Decentralized Storage:

a Novel Approach to Proof-of-Replication via Polynomial Evaluation

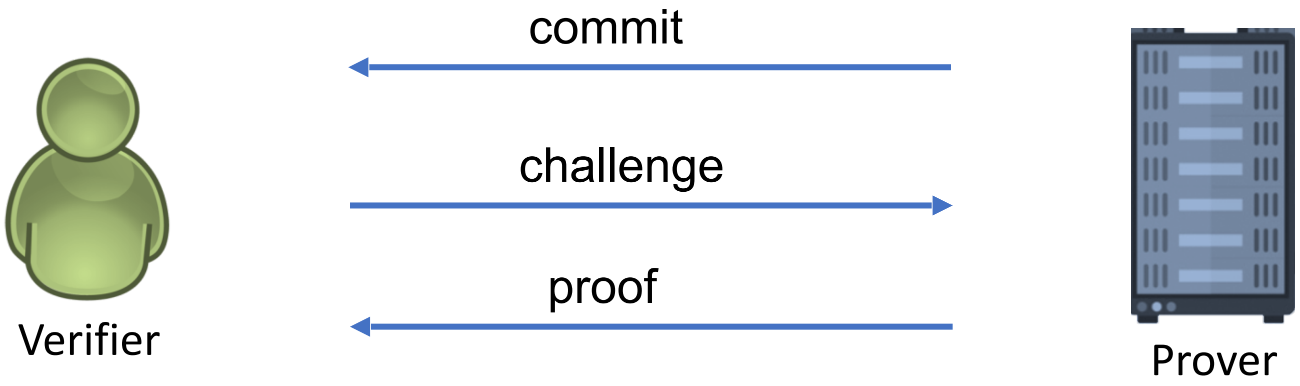
Giuseppe Ateniese[1], Foteini Baldimtsi[1],

Matteo Campanelli[2], **Danilo Francati**[1], Ioanna Karantaidou[1]

[1]George Mason University, [2]Matter Labs

Crypto ‘24

Proof of Space (PoS)



commit

challenge

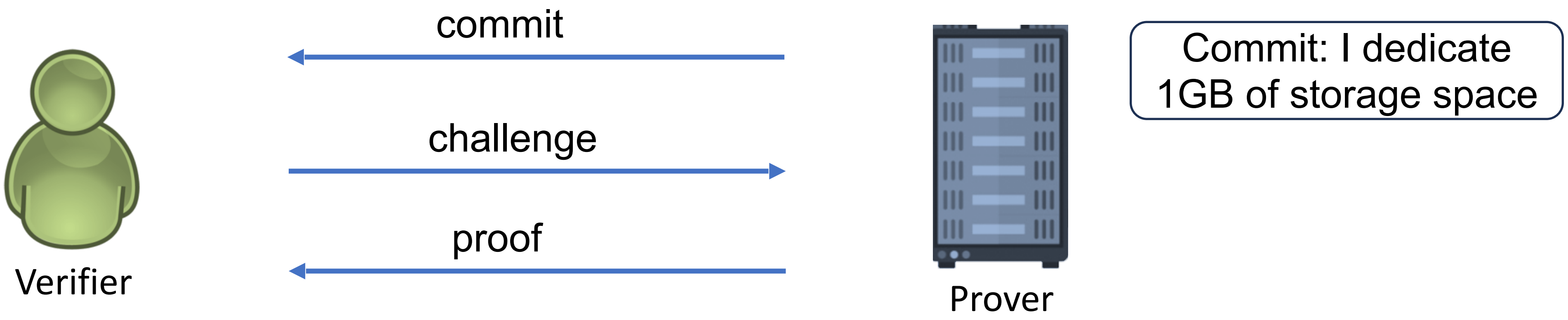
proof

Verifier

Prover

Commit: I dedicate 1GB of storage space

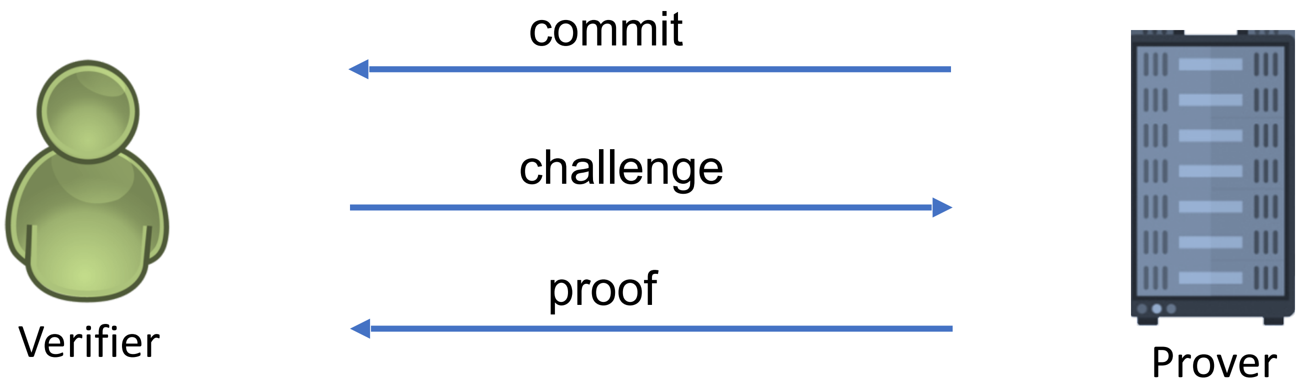
Proof of Space (PoS)



(eco-friendly) Alternative to Proof-of-Work

Applications: spam prevention, DDoS attack resistance, Sybil-resistant blockchain consensus

Proof of Space (PoS)



commit

challenge

proof

Verifier

Prover

Commit: I dedicate 1GB of storage space

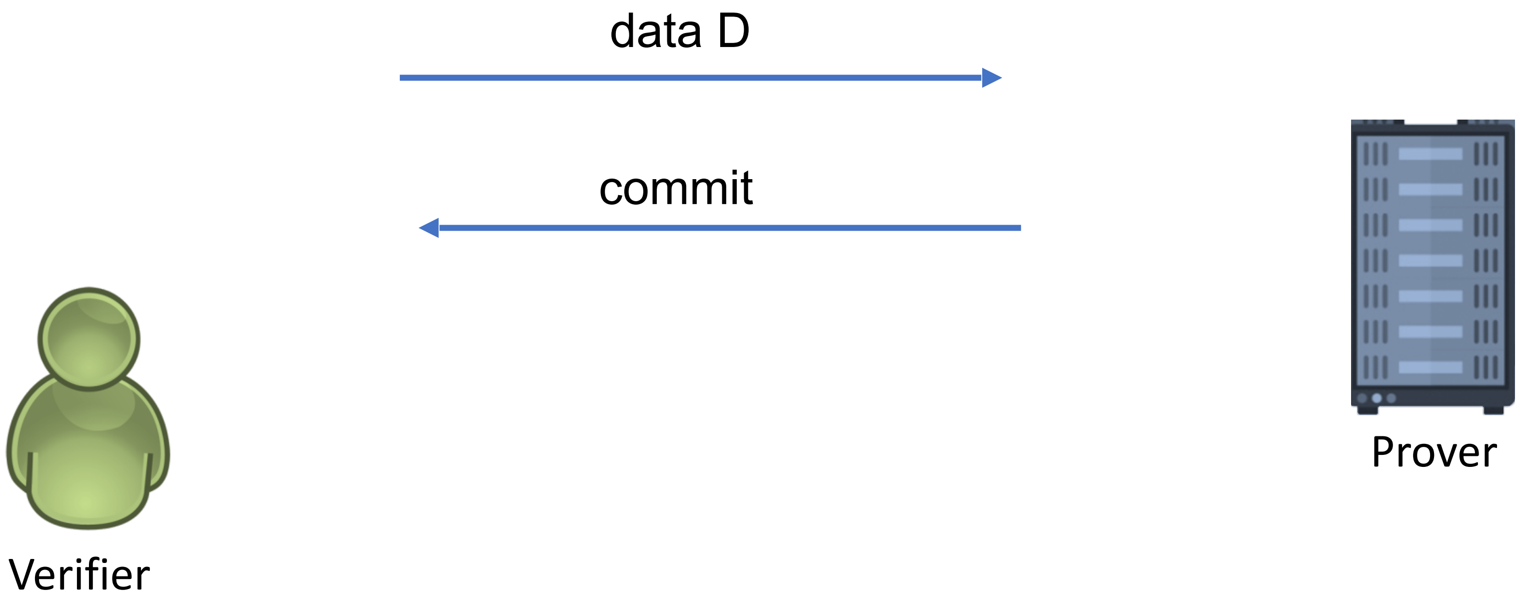
⚠ no specific/useful data are being stored by the prover

(eco-friendly) Alternative to Proof-of-Work

Applications: spam prevention, DDoS attack resistance, Sybil-resistant blockchain consensus

Proof of Replication (PoRep)

Setup Phase



data D

commit

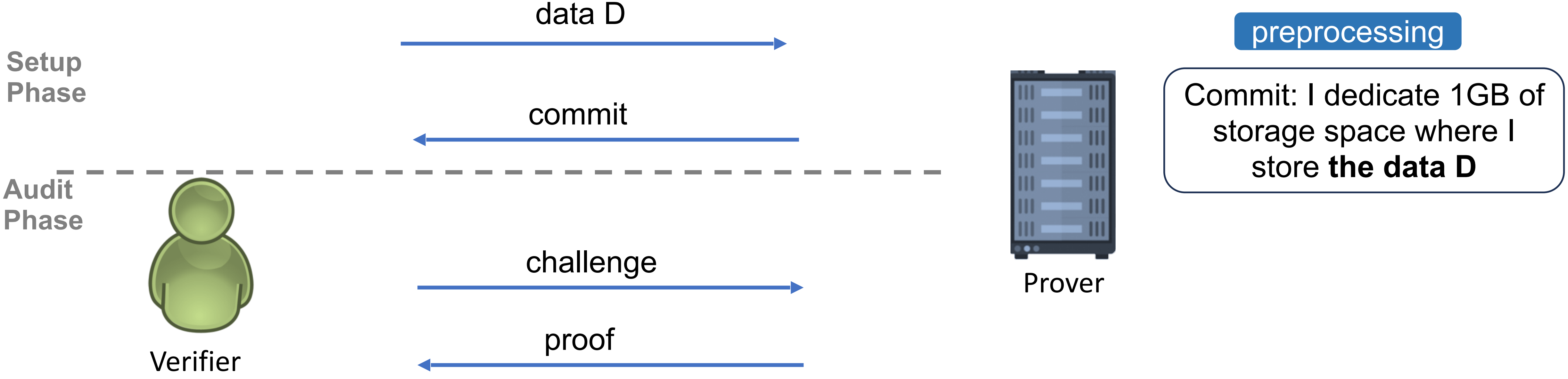
Verifier

Prover

preprocessing

Commit: I dedicate 1GB of storage space where I store **the data D**

Proof of Replication (PoRep)



Setup Phase

data D

commit

Audit Phase

challenge

proof

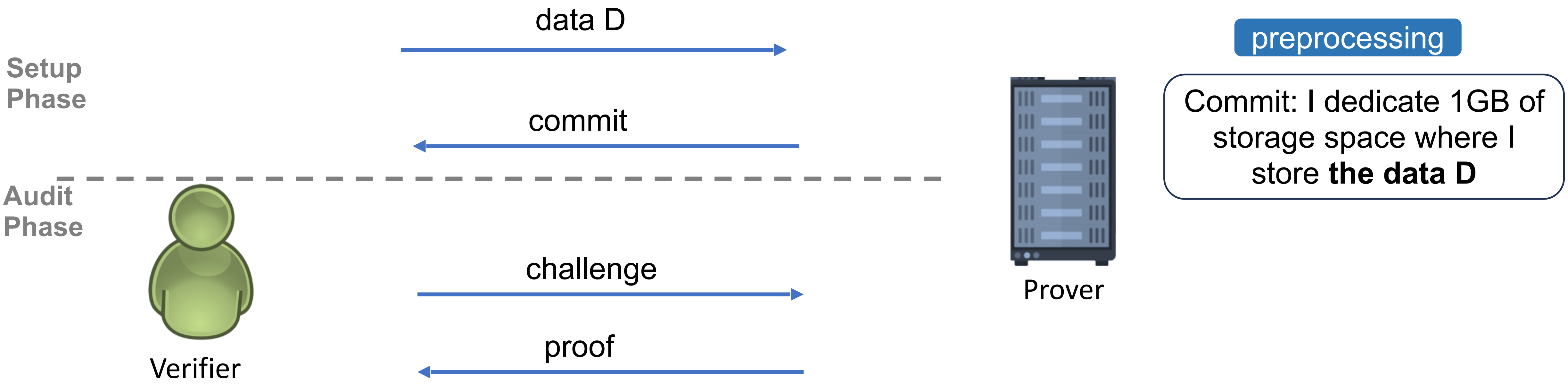
Verifier

Prover

preprocessing

Commit: I dedicate 1GB of storage space where I store **the data D**

Proof of Replication (PoRep)



Setup Phase

data D

commit

Audit Phase

challenge

proof

Verifier

Prover

preprocessing

Commit: I dedicate 1GB of storage space where I store **the data D**

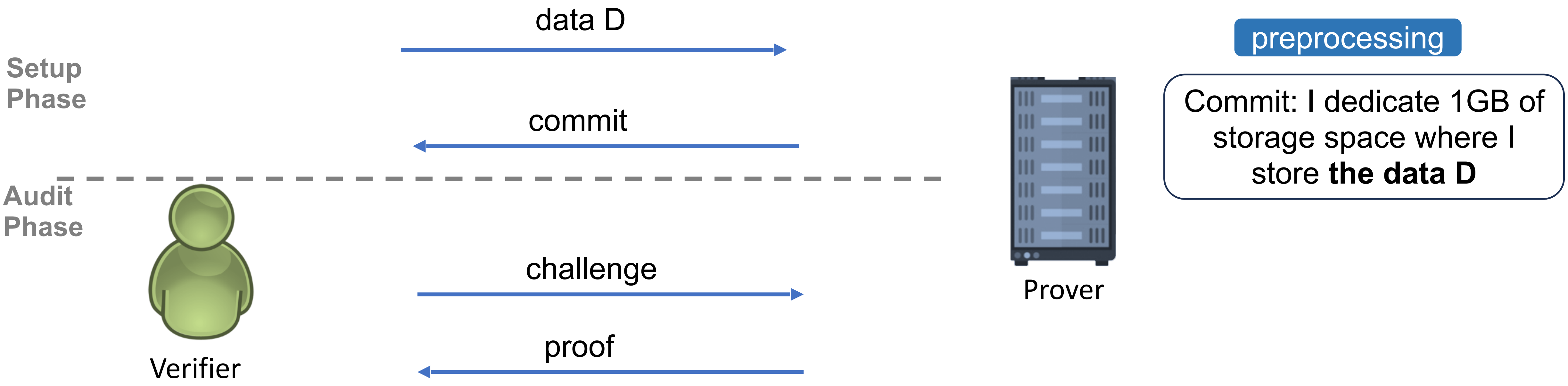
No waste of the dedicated space!

PoRep guarantees that the prover is dedicating unique storage resources per replica of the data

↓

<u>decentralized and verifiable file storage</u>

Proof of Replication (PoRep)



Setup Phase

data D

commit

Audit Phase

challenge

proof

Verifier

Prover

preprocessing

Commit: I dedicate 1GB of storage space where I store **the data D**

No waste of the dedicated space!

PoRep guarantees that the prover is dedicating unique storage resources per replica of the data

<u>decentralized and verifiable file storage</u>

“Filecoin (FIL) is an open-source, public cryptocurrency and digital payment system intended to be a blockchain-based cooperative digital storage and data retrieval method.” [wikipedia]

PoRep Definition

PoRep Definition

preprocessing

audit phase

PoRep Definition

public keys

- Setup(1^λ,1^t,1^n) → ek,pk,vk
- Encode(m,ek,id) → encoding c and digest h

Publish h

preprocessing

---

audit phase

PoRep Definition

public keys

- Setup($1^\lambda, 1^t, 1^n$) → ek,pk,vk
- Encode(m,ek,id) → encoding c and digest h

Publish h

preprocessing

- Prove(pk, challenge, c) → π

audit phase

PoRep Definition

public keys

- Setup(1^λ,1^t,1^n) → ek,pk,vk
- Encode(m,ek,id) → encoding c and digest h

preprocessing

Publish h

- Prove(pk, challenge, c) → π
- Verify(vk,h,π) → 0/1

audit phase

PoRep Definition

public keys

- Setup($1^{\lambda}, 1^{t}, 1^{n}$) → ek,pk,vk
- Encode(m,ek,id) → encoding c and digest h

Publish h

preprocessing

- Prove(pk, challenge, c) → π
- Verify(vk,h,π) → 0/1
- Decode(ek,id,c) → m

audit phase

PoRep Definition

public keys

- Setup($1^\lambda, 1^t, 1^n$) → ek,pk,vk
- Encode(m,ek,id) → encoding c and digest h

preprocessing

Publish h

- Prove(pk, challenge, c) → π
- Verify(vk,h,π) → 0/1

audit phase

- Decode(ek,id,c) → m

Properties: encoding correctness, proof completeness, replication and extraction

Proof of Replication (PoRep)

Proof of Storage
PoS

+

Proof of Data Possession
PDP

+

Proof of Retrievability
PoR

Proof of Replication (PoRep)

Proof of Storage
PoS

+

Proof of Data Possession
PDP

+

Proof of Retrievability
PoR

Memory usage

File size n then
memory usage is n

Even if file is compressible

The data/file is stored

Data/file can be retrieved

PoRep Definition

public keys

- Setup(1^λ,1^t,1^n) → ek,pk,vk
- Encode(m,ek,id) → encoding c and digest h

Publish h

In some settings, messages might be highly compressible, i.e. m=($F_k(1)$, $F_k(2)$,…). There, a prover could avoid storing m and generate as needed when challenged.

Slow Encode()

+

Run time of computing proof < run time of Encode()

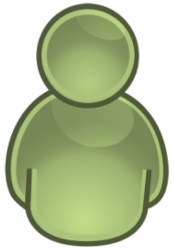
- Prove(pk, challenge, c) → π
- Verify(vk,h,π) → 0/1
- Decode(ek,id,c) → m

Properties: encoding correctness, proof completeness, replication and extraction

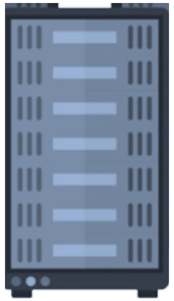
Proof of Replication (PoRep) – [Fisch'18]

preprocessing

Setup Phase



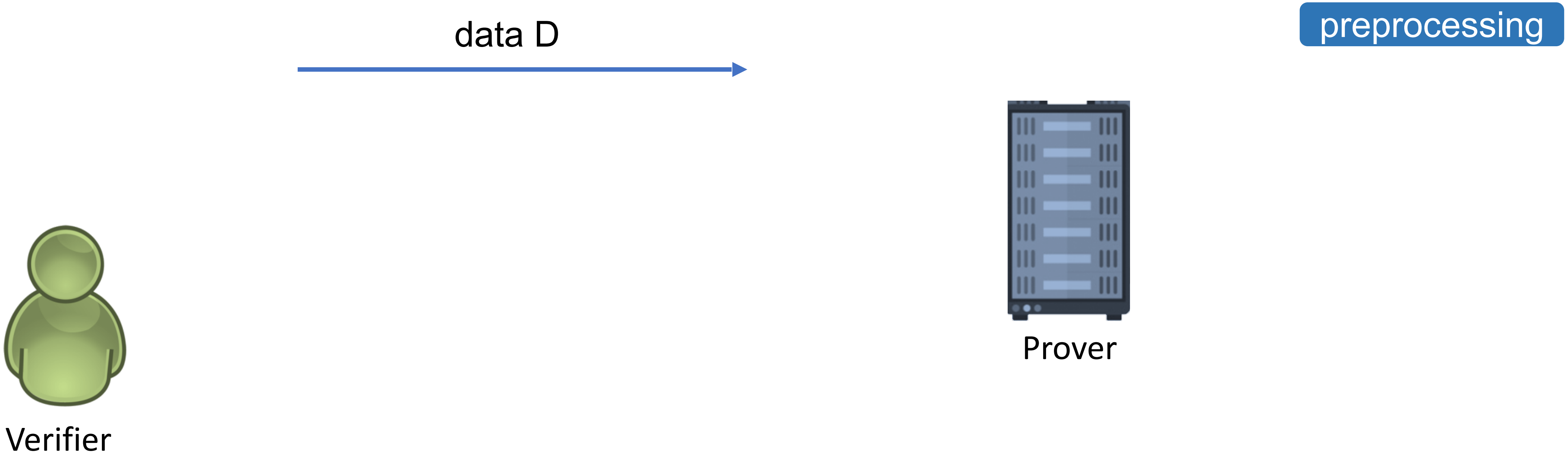
Verifier



Prover

Proof of Replication (PoRep) – [Fisch'18]

Setup Phase



data D

preprocessing

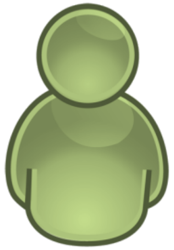
Prover

Verifier

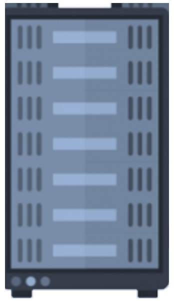
Proof of Replication (PoRep) – [Fisch'18]

Setup Phase

data D



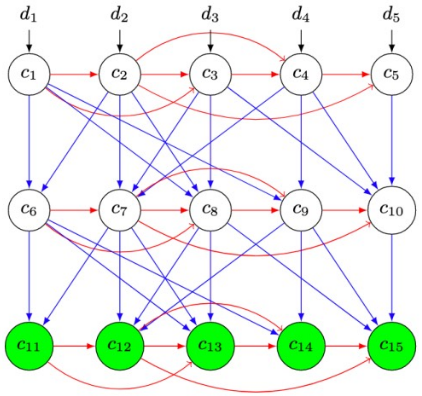
Verifier



Prover

preprocessing

c_i : data encodings



xor D

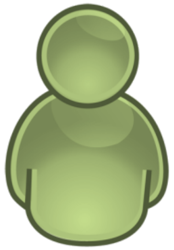
Depth Robust Graphs (DRGs)

Proof of Replication (PoRep) – [Fisch'18]

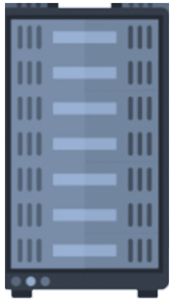
Setup Phase

data D

commit to the DRG (MT on last layer)



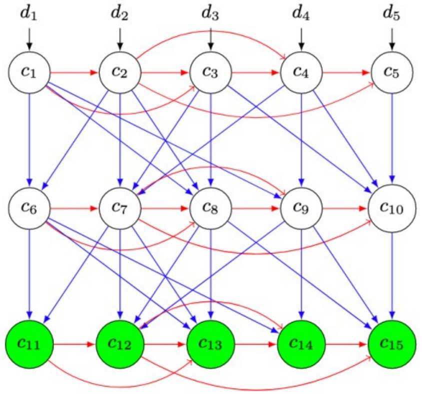
Verifier



Prover

preprocessing

c_i : data encodings



xor D

Depth Robust Graphs (DRGs)

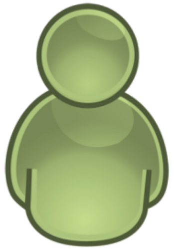
Proof of Replication (PoRep) – [Fisch'18]

Setup Phase

data D

commit to the DRG (MT on last layer)

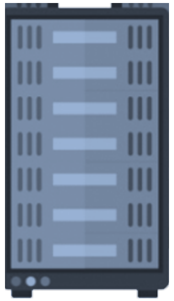
Audit Phase



Verifier

challenge on a subset of C_i's

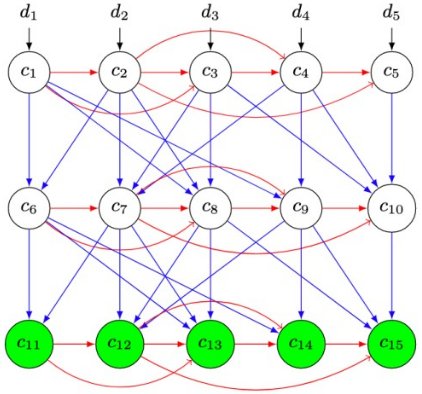
proofs (MT openings)



Prover

preprocessing

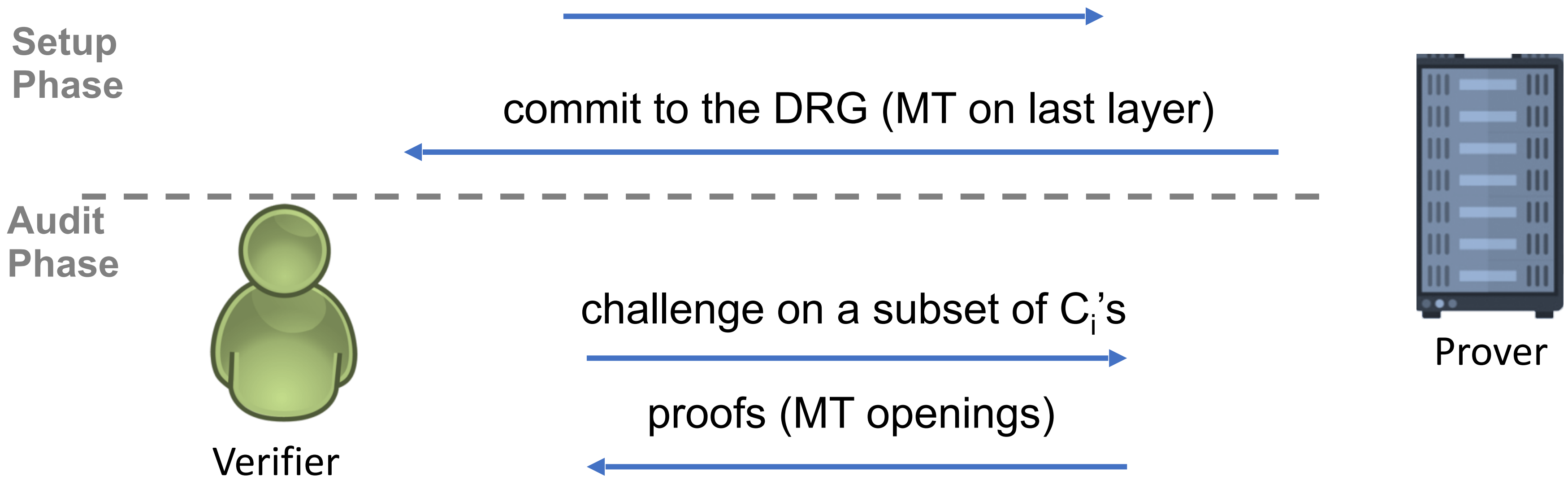
c_i : data encodings



xor D

Depth Robust Graphs (DRGs)

Proof of Replication (PoRep) – [Fisch’18]



Setup Phase

data D

commit to the DRG (MT on last layer)

Audit Phase

challenge on a subset of C_i’s

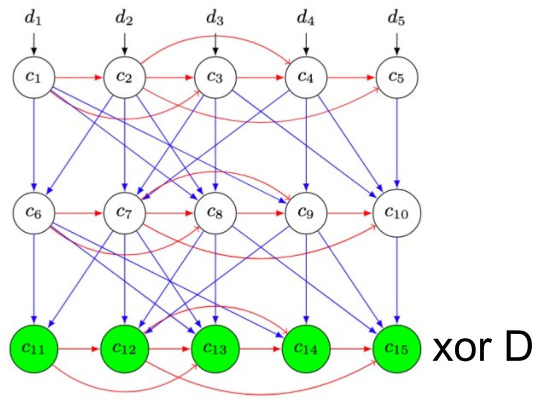
proofs (MT openings)

Verifier

Prover

preprocessing

c_i : data encodings



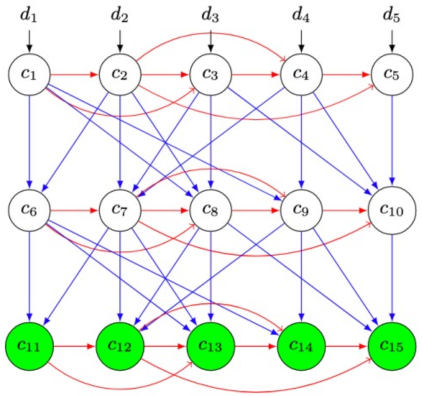
Depth Robust Graphs (DRGs)

⚠ The auditing phase is probabilistic – a large set of challenges is needed to get a good level of security.

File of n blocks - > Memory usage is n(1-ϵ) where ϵ is constant

Proof of Replication (PoRep) – [Fisch’18]

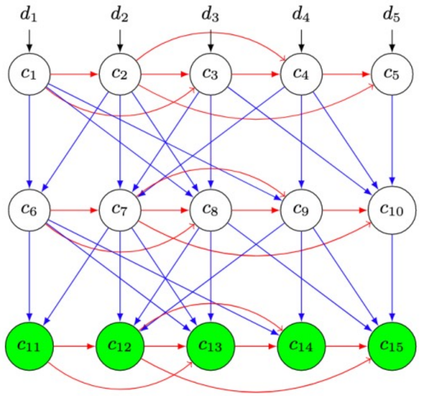
c_i : data encodings



xor D_1

Depth Robust Graphs (DRGs)

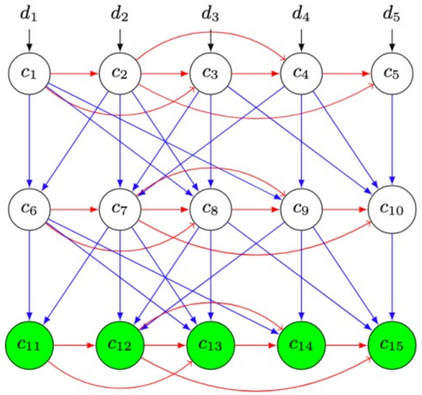
c_i : data encodings



xor D_2

Depth Robust Graphs (DRGs)

c_i : data encodings

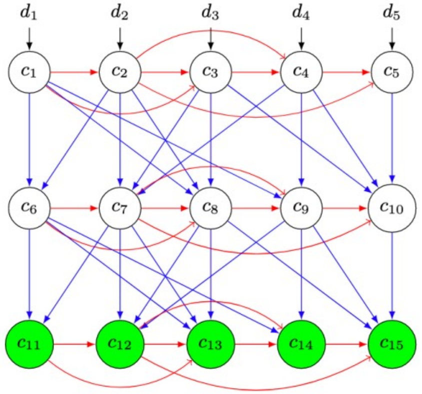


xor D_3

Depth Robust Graphs (DRGs)

Proof of Replication (PoRep) – [Fisch'18]

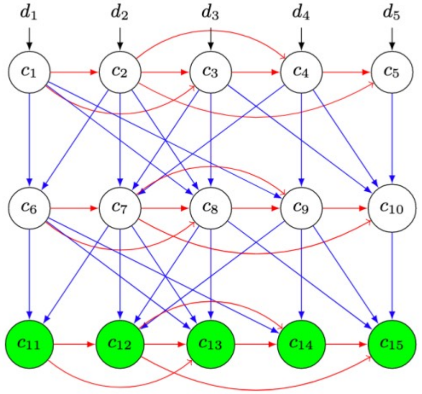
c_i : data encodings



xor D_1

Depth Robust Graphs (DRGs)

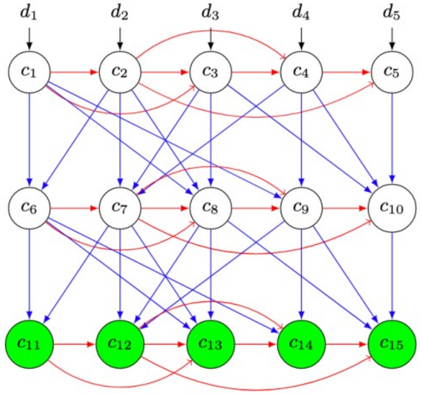
c_i : data encodings



xor D_2

Depth Robust Graphs (DRGs)

c_i : data encodings



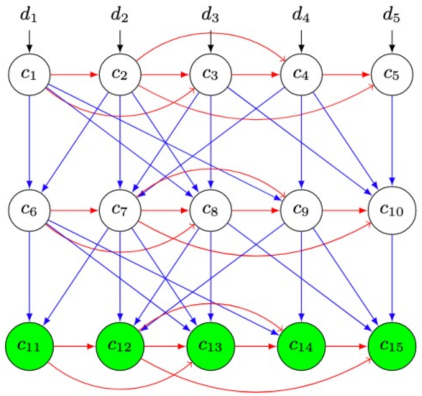
xor D_3

Depth Robust Graphs (DRGs)

u file of n blocks - > Memory usage is u*n(1-ϵ) where ϵ is constant.

Proof of Replication (PoRep) – [Fisch'18]

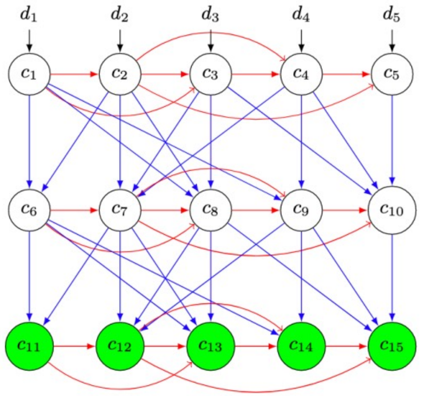
c_i : data encodings



xor D_1

Depth Robust Graphs (DRGs)

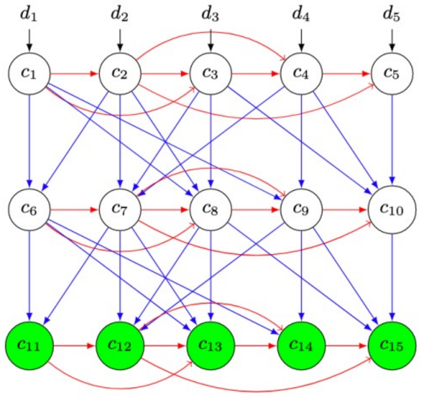
c_i : data encodings



xor D_2

Depth Robust Graphs (DRGs)

c_i : data encodings



xor D_3

Depth Robust Graphs (DRGs)

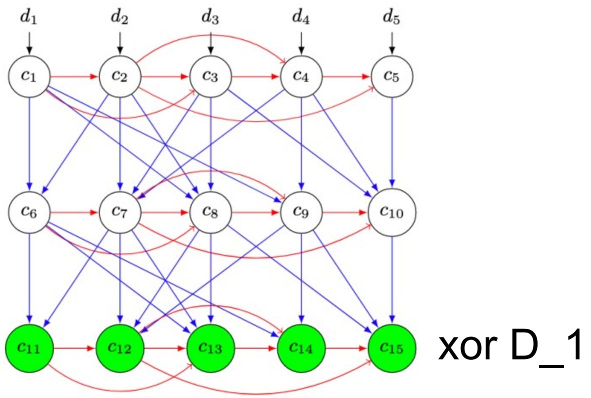
u file of n blocks - > Memory usage is u*n(1-ϵ) where ϵ is constant.

Loss scales with u (the number of files stored), i.e., loss is u*n*ϵ

The problem is the probabilistic check on the last layer (the encoding)

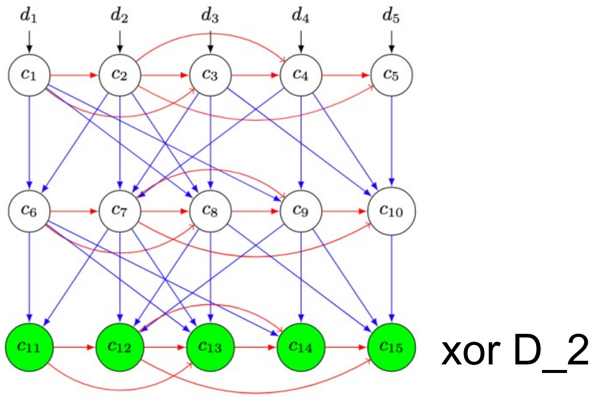
Proof of Replication (PoRep) – [Fisch'18]

c_i : data encodings



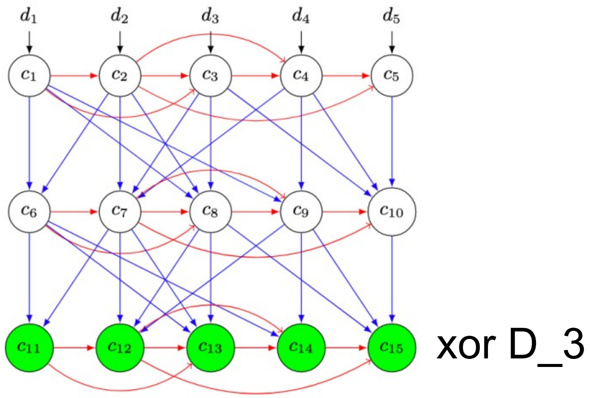
Depth Robust Graphs (DRGs)

c_i : data encodings



Depth Robust Graphs (DRGs)

c_i : data encodings



Depth Robust Graphs (DRGs)

u file of n blocks - > Memory usage is u*n(1-ϵ) where ϵ is constant.

Loss scales with u (the number of files stored), i.e., loss is u*n*ϵ

The problem is the probabilistic check on the last layer (the encoding)

⚠ Remove the loss proportional to u -> Scales the challenges proportional to u ⚠

Our work

public keys

- Setup($1^\lambda,1^t,1^n$) → ek,pk,vk

preprocessing

- Encode(m,ek,id) → encoding c and digest h

Publish h

- Prove(pk, challenge, c) → π

audit phase

- Verify(vk,h,π) → 0/1
- Decode(ek,c) → m

Properties: encoding correctness, proof completeness, replication and extraction

Our work

In this work we assume **the encoding is honestly executed** (i.e., digest h is honest). Worst case: Use a SNARK to verify h

public keys

- Setup(1^λ,1^t,1^n) → ek,pk,vk

preprocessing

- Encode(m,ek,id) → encoding c and digest h

Publish h

- Prove(pk, challenge, c) → π

audit phase

- Verify(vk,h,π) → 0/1
- Decode(ek,c) → m

Properties: encoding correctness, proof completeness, replication and extraction

Our work

In this work we assume **the encoding is honestly executed** (i.e., digest h is honest). Worst case: Use a SNARK to verify h

preprocessing

public keys

- Setup(1^λ,1^t,1^n) → ek,pk,vk
- Encode(m,ek,id) → encoding c and digest h

Publish h

---

audit phase

Our focus is to **reduce complexity** and **increase security** of this phase

- Prove(pk, challenge, c) → π
- Verify(vk,h,π) → 0/1
- Decode(ek,c) → m

Properties: encoding correctness, proof completeness, replication and extraction

Our construction

Polynomial Evaluation

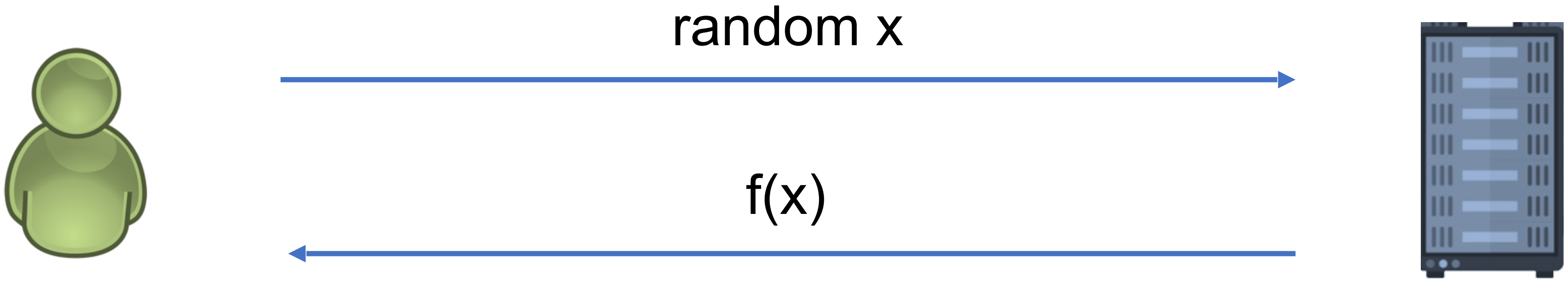
degree d polynomial $f(X) \xleftarrow{\$} \mathbb{Z}_p[X]$

$$f(X) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_d \cdot x^d$$

Polynomial Evaluation

degree d polynomial $f(X) \leftarrow\$ \mathbb{Z}_p[X]$

$$f(X) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_d \cdot x^d$$



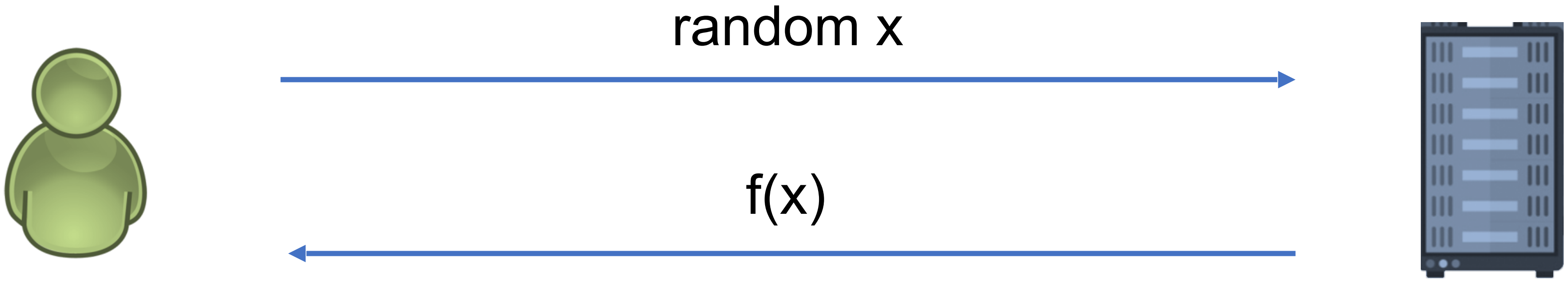
random x

f(x)

Polynomial Evaluation

degree d polynomial $f(X) \xleftarrow{\$} \mathbb{Z}_p[X]$

$$f(X) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_d \cdot x^d$$



random x

f(x)

space requirements?

Incompressibility of Random Polynomials

randomly sampled
polynomial of degree d

$$f(X) \leftarrow_{\$} \mathbb{Z}_p[X]$$

The goal: evaluating f(X) should require memory close to |f(X)|

Incompressibility of Random Polynomials

randomly sampled
polynomial of degree d

$$f(X) \leftarrow_{\$} \mathbb{Z}_p[X]$$

The goal: evaluating f(X) should require memory close to |f(X)|

😈

- pre-processes f(X) to compute a memory α smaller than |f(X)|
- shouldn’t be possible to evaluate f(x) on a random point x, unless |α| ≈ |f(X)|

Incompressibility of Random Polynomials

randomly sampled
polynomial of degree d

$$f(X) \leftarrow_{\$} \mathbb{Z}_p[X]$$

The goal: evaluating f(X) should require memory close to |f(X)|

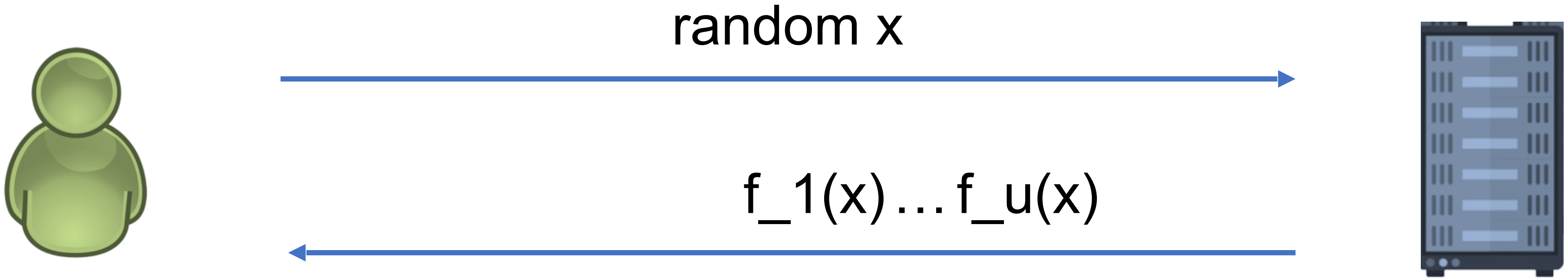
😈

- pre-processes f(X) to compute a memory α smaller than |f(X)|
- shouldn’t be possible to evaluate f(x) on a random point x, unless |α| ≈ |f(X)|

[ACFPT’23]: |α| ≈ |f(X)| − d*|x|

Multiple polynomials?

$f_1(X) \leftarrow_{\$} \mathbb{Z}_p[X]$ … $f(X)_u \leftarrow_{\$} \mathbb{Z}_p[X]$

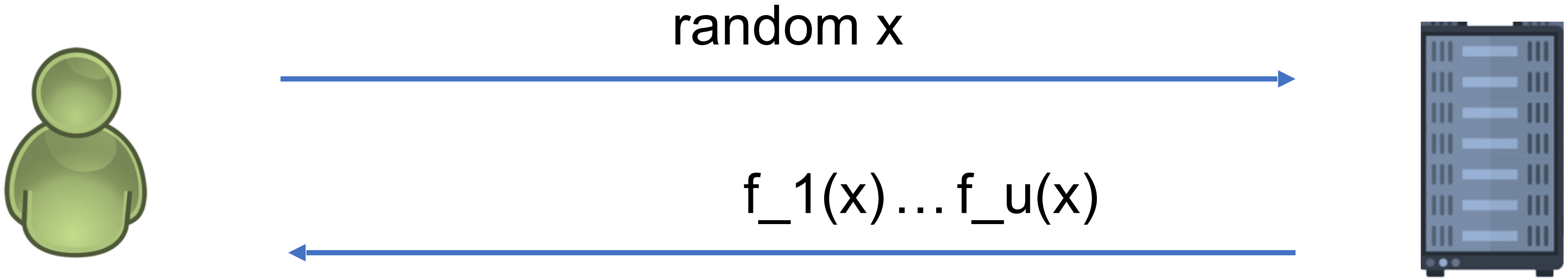


random x

f_1(x) … f_u(x)

Multiple polynomials?

$f_1(X) \leftarrow_{\$} \mathbb{Z}_p[X]$ … $f(X)_u \leftarrow_{\$} \mathbb{Z}_p[X]$



random x

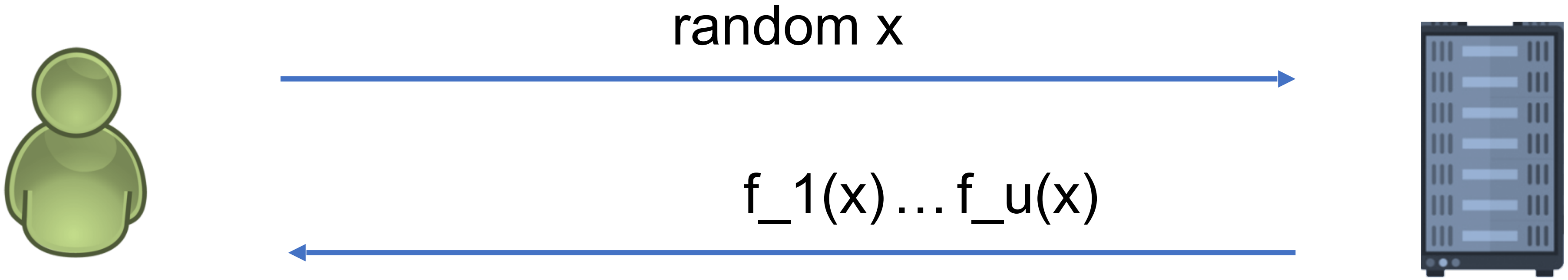
f_1(x) … f_u(x)

single polynomial

[ACFPT’23]: |α| ≈ |f(X)| − d*|x|

Multiple polynomials?

$f_1(X) \leftarrow_{\$} \mathbb{Z}_p[X]$ … $f(X)_u \leftarrow_{\$} \mathbb{Z}_p[X]$



random x

f_1(x) … f_u(x)

single polynomial

[ACFPT’23]: |α| ≈ |f(X)| − d*|x|

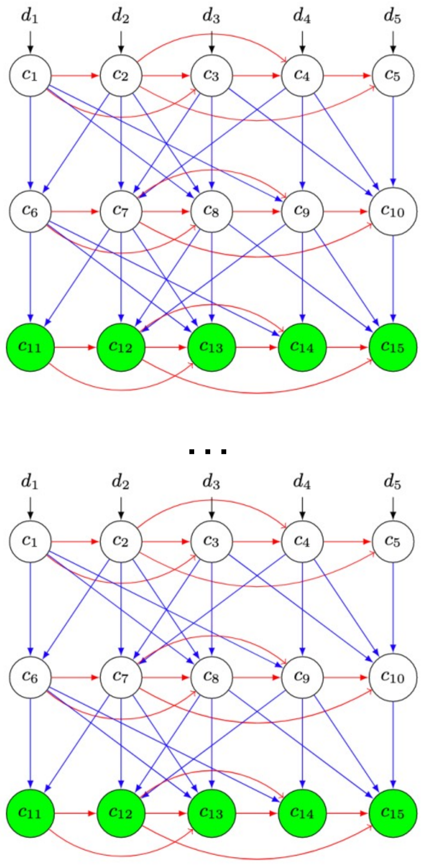
multiple polynomials

|α| ≈ u*|f(X)| − d*|x|

Partial construction

Setup Phase

preprocessing

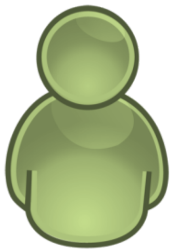


Partial construction

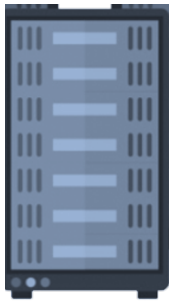
Setup Phase

data D

Commit on ???

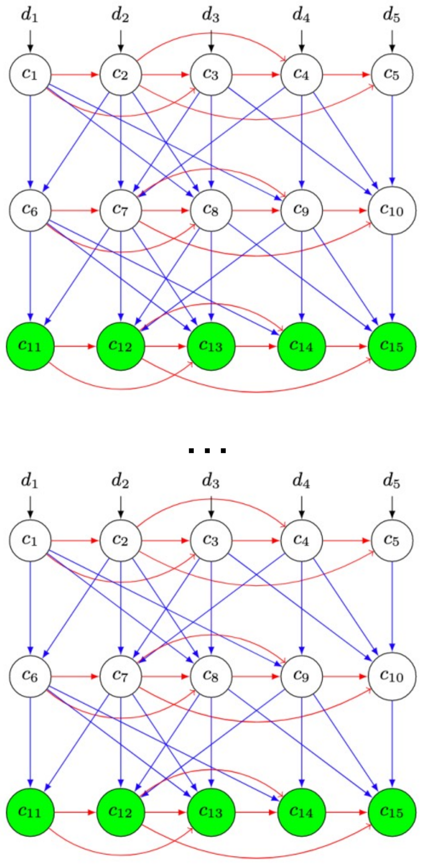


Verifier



Prover

preprocessing



xor D_1= f_1(X)

xor D_u = f_u(X)

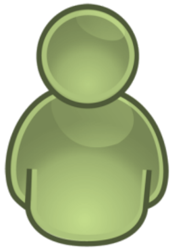
Partial construction

Setup Phase

data D

Commit on ???

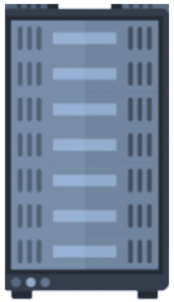
Audit Phase



Verifier

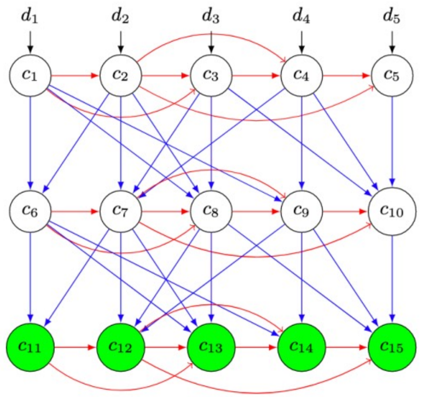
x

f_1(x), …, f_u(x)



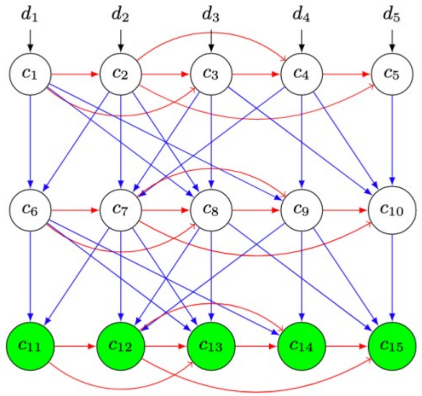
Prover

preprocessing



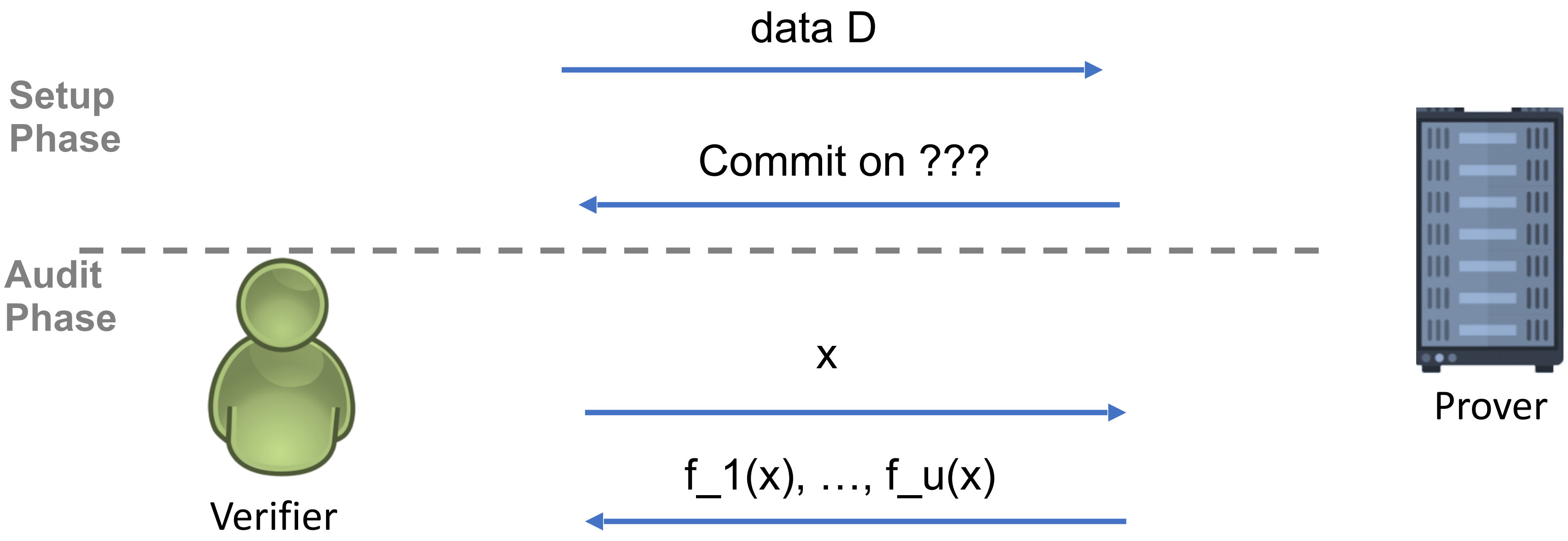
xor D_1= f_1(X)

…

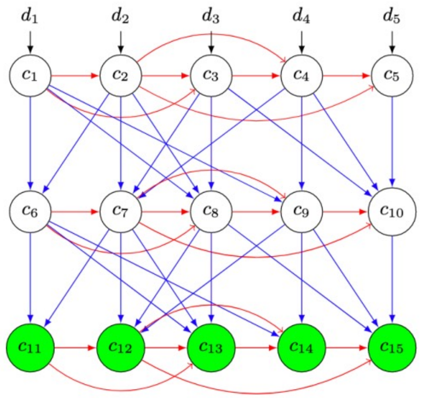


xor D_u = f_u(X)

Partial construction

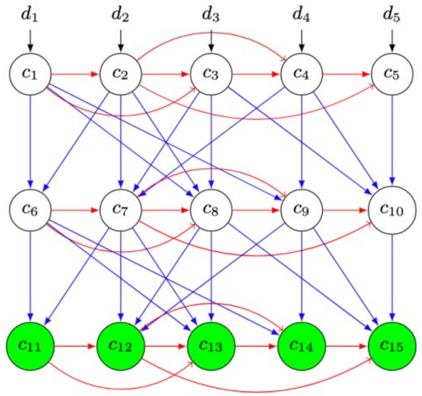


preprocessing



xor D_1= f_1(X)

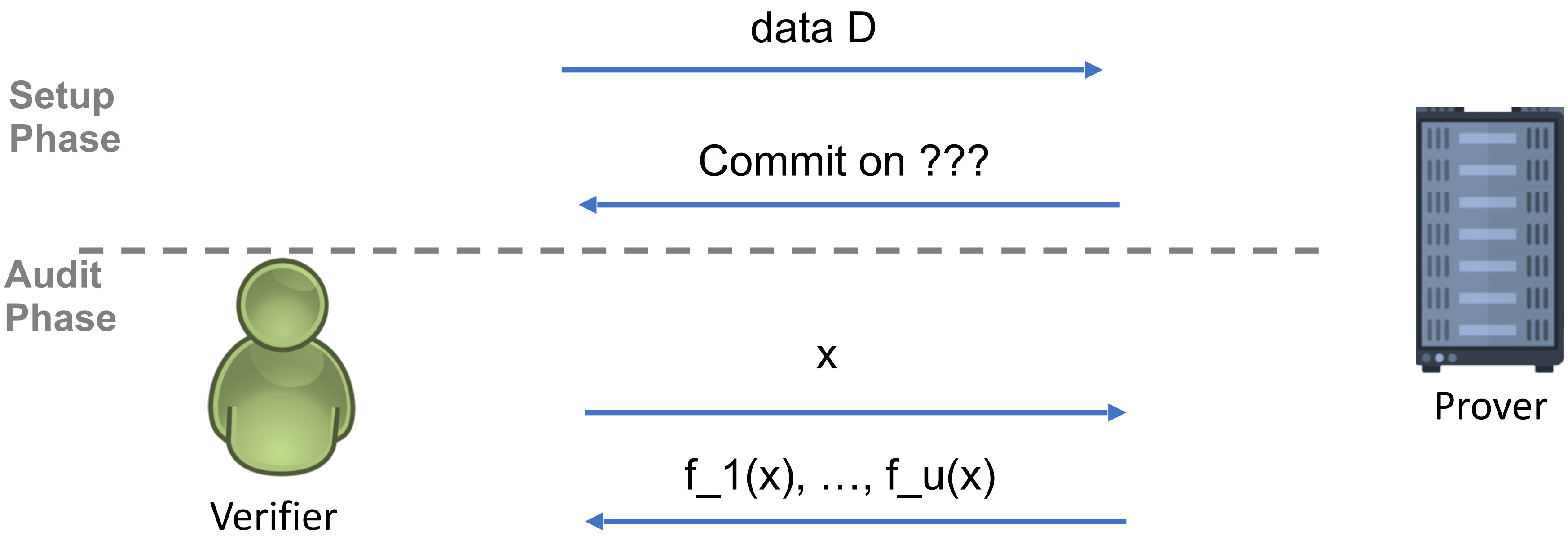
…



xor D_u = f_u(X)

Memory usage: u*|f(x)| - d*|x| ≈u*|f(x)| for large values of u

Partial construction



Setup Phase

data D

Commit on ???

Audit Phase

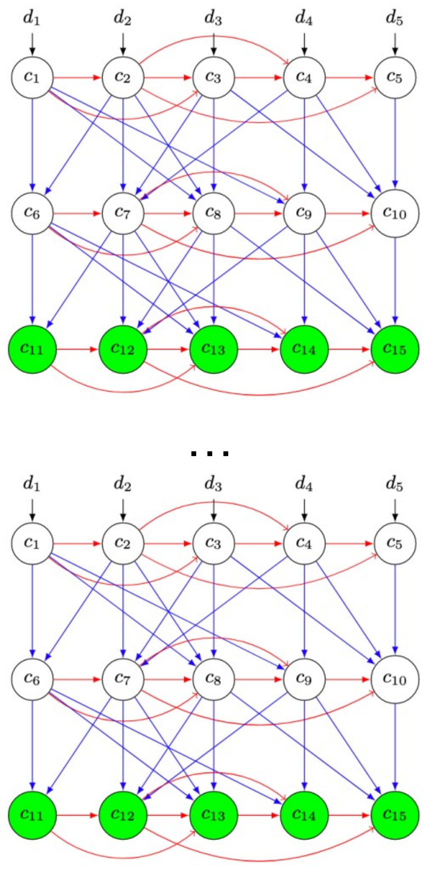
x

f_1(x), …, f_u(x)

Verifier

Prover

preprocessing



xor D_1= f_1(X)

xor D_u = f_u(X)

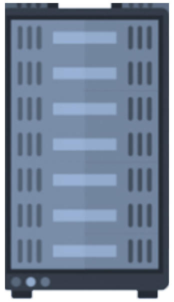
Memory usage: u*|f(x)| - d*|x| ≈u*|f(x)| for large values of u

⚠ Prover needs to read f_1(X), … f_u(X) entirely - Problem for efficiency

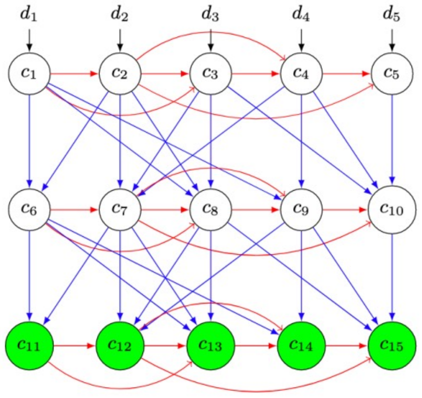
⚠ How verify that f_1(x), … f_u(x) are correct evaluations (efficiently)?

Poly-logarithmic prover’s running time

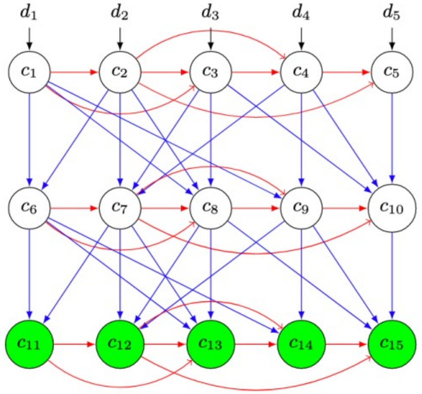
Kedlaya and Uman 2011: “Fast polynomial factorization and modular composition”



Prover



…

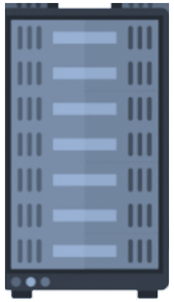


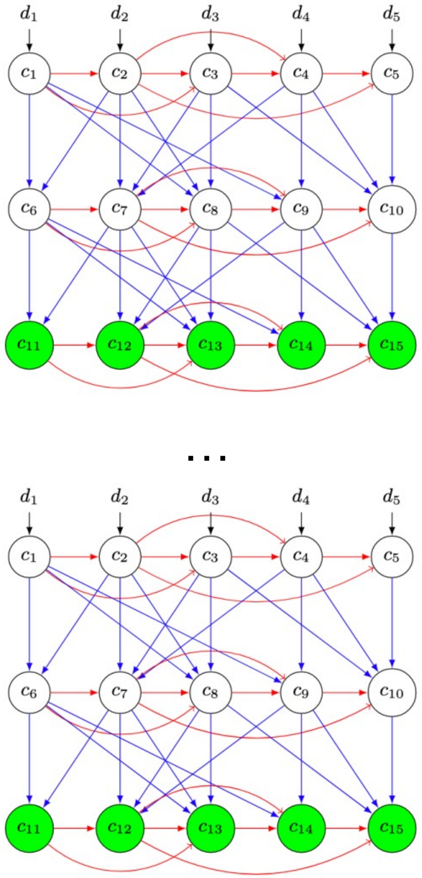
Poly-logarithmic prover's running time

Kedlaya and Uman 2011: "Fast polynomial factorization and modular composition"

✓ uses a RAM data structure to expedite polynomial evaluations

(computes f(x) in poly-log time in d)


Prover

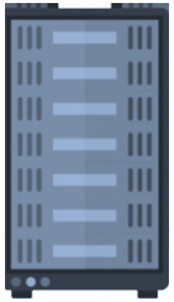


Poly-logarithmic prover’s running time

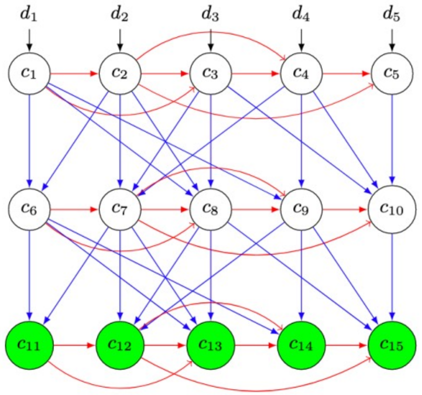
Kedlaya and Uman 2011: “Fast polynomial factorization and modular composition”

✓ uses a RAM data structure to expedite polynomial evaluations

(computes f(x) in poly-log time in d)

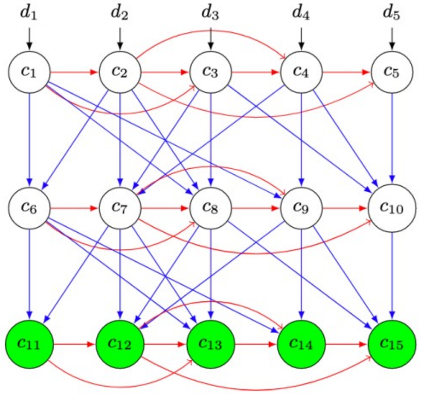


Prover



xor D_1= f_1(X) -> Compute data structure D_1

…



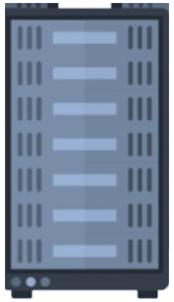
xor D_u = f_u(X) -> Compute data structure D_u

Poly-logarithmic prover’s running time

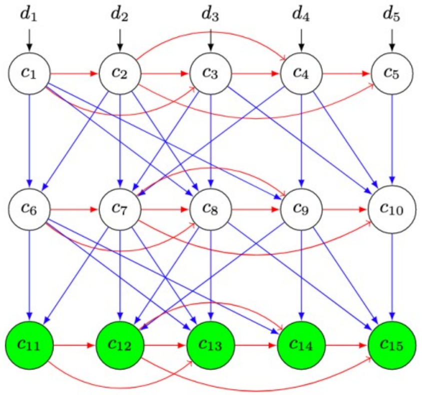
Kedlaya and Uman 2011: “Fast polynomial factorization and modular composition”

✓ uses a RAM data structure to expedite polynomial evaluations

(computes f(x) in poly-log time in d)

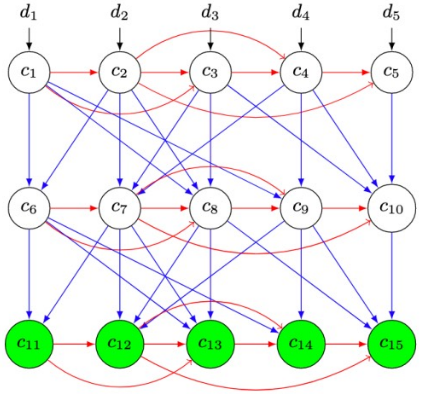


Prover



xor D_1= f_1(X) -> Compute data structure D_1

…

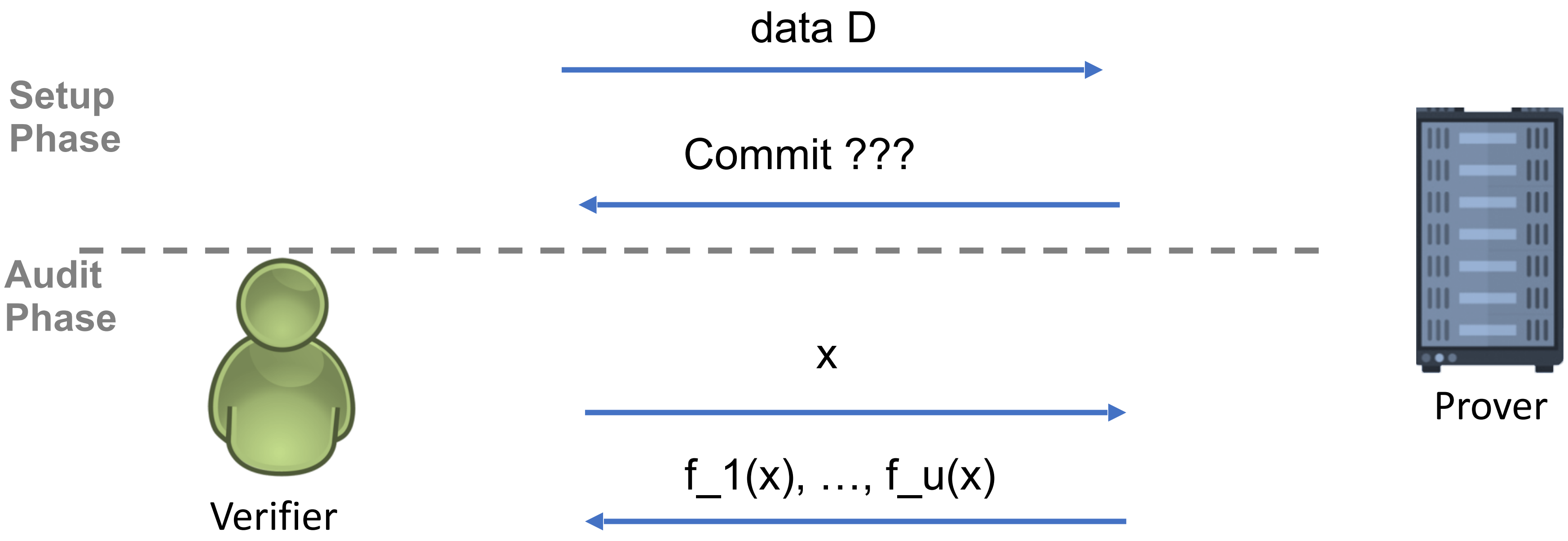


f_i(X) can be evaluated in time polylog(d)

xor D_u = f_u(X) -> Compute data structure D_u

Partial construction

preprocessing



Setup Phase

data D

Commit ???

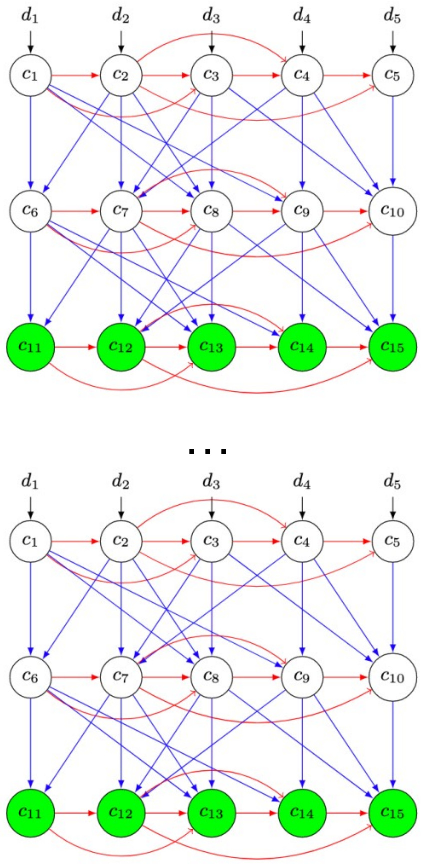
Audit Phase

x

f_1(x), …, f_u(x)

Verifier

Prover



xor D_1= f_1(X)

xor D_u = f_u(X)

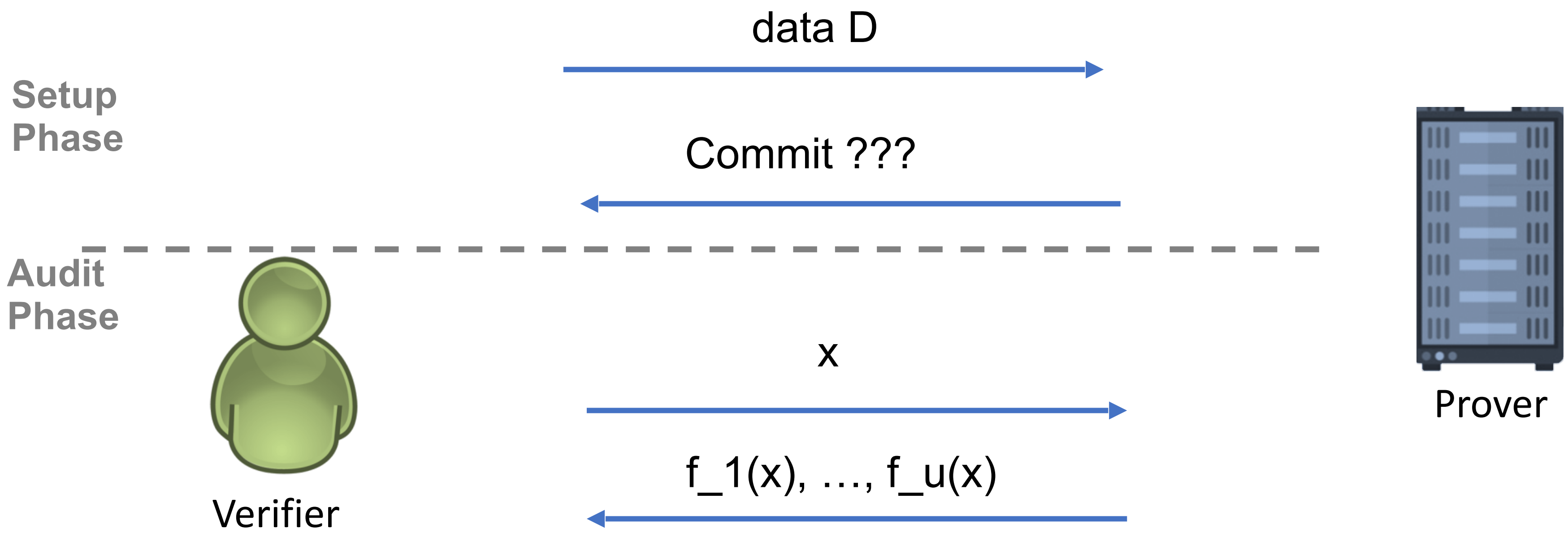
Memory usage: u*|f(x)| - d*|x| ≈u*|f(x)| for large values of u

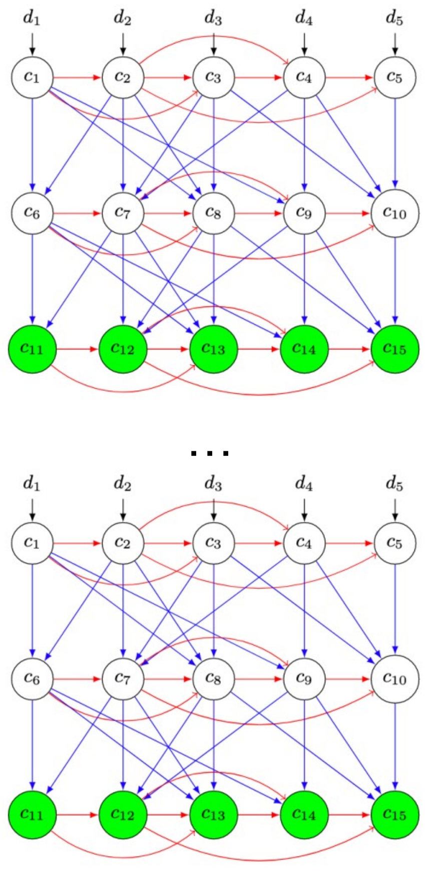
⚠ Prover needs to read f_1(X), … f_u(X) entirely - Problem for efficiency

⚠ How verify that f_1(x), … f_u(x) are correct evaluations?

Partial construction

preprocessing





xor D_1= f_1(X)

xor D_u = f_u(X)

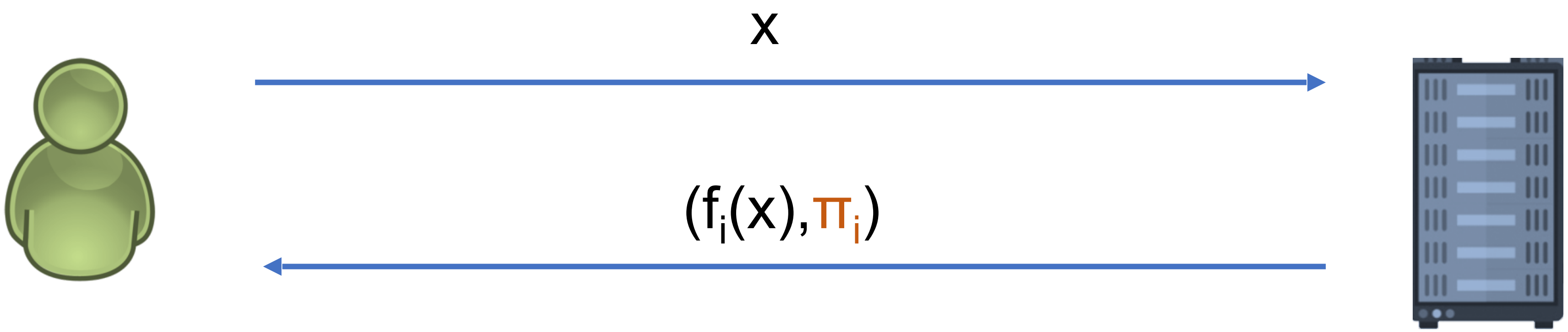
Memory usage: u*|f(x)| - d*|x| ≈u*|f(x)| for large values of u

~~Prover needs to read f_1(X), … f_u(X) entirely - Problem for efficiency~~

How verify that f_1(x), … f_u(x) are correct evaluations?

Poly-log verification

Let’s use SNARKs!



x

$(f_i(x), \pi_i)$

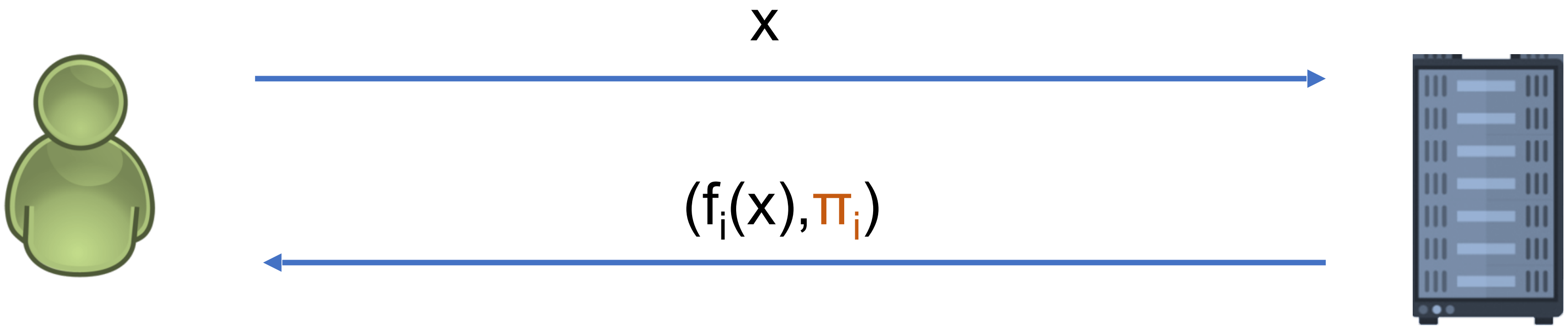
Compute $f_i(x)$

SNARK π_i

“$f_i(x)$ was honestly computed using the data structure D”

Poly-log verification

~~Let's use SNARKs!~~



x

$(f_i(x), \pi_i)$

Compute $f_i(x)$

SNARK π_i

"$f_i(x)$ was honestly computed using the data structure D"

✗ prover time becomes linear to D

Poly-log verification

What if the verifier also computes f(x)?

Poly-log verification

What if the verifier also computes f(x)?

❌ <u>no access to D!</u>

Poly-log verification

What if the verifier also computes f(x)?

❌ no access to D!

Prover only needs to access a poly-log number of blocks $D' \subset D$

Poly-log verification

What if the verifier also computes f(x)?

✗ no access to D!

💡 Prover only needs to access a poly-log number of blocks $D' \subset D$

preprocessing

Poly-log verification

What if the verifier also computes f(x)?

✗ no access to D!

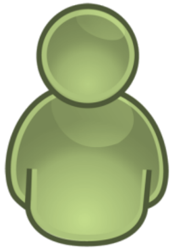
Prover only needs to access a poly-log number of blocks $D' \subset D$

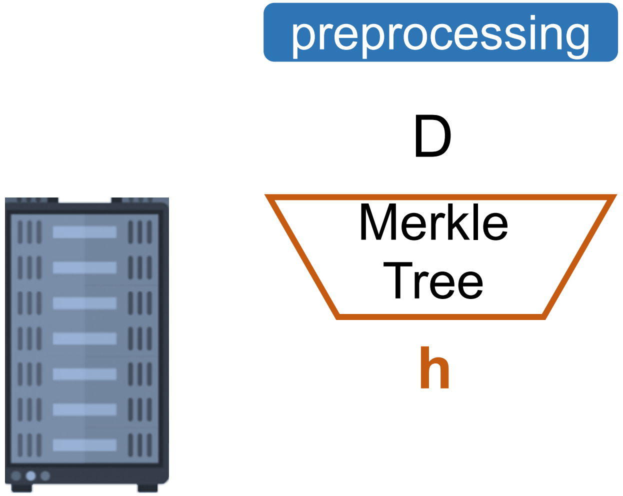
Poly-log verification

What if the verifier also computes f(x)?

❌ no access to D!

Prover only needs to access a poly-log number of blocks $D' \subset D$



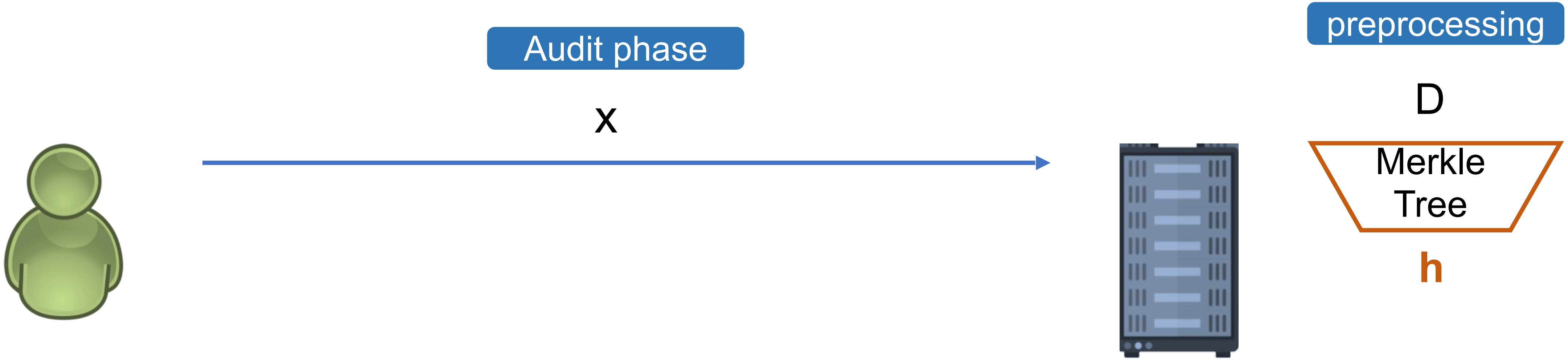


Poly-log verification

What if the verifier also computes f(x)?

✗ <u>no access to D!</u>

Prover only needs to access a poly-log number of blocks $D' \subset D$



Audit phase

x

preprocessing

D

Merkle Tree

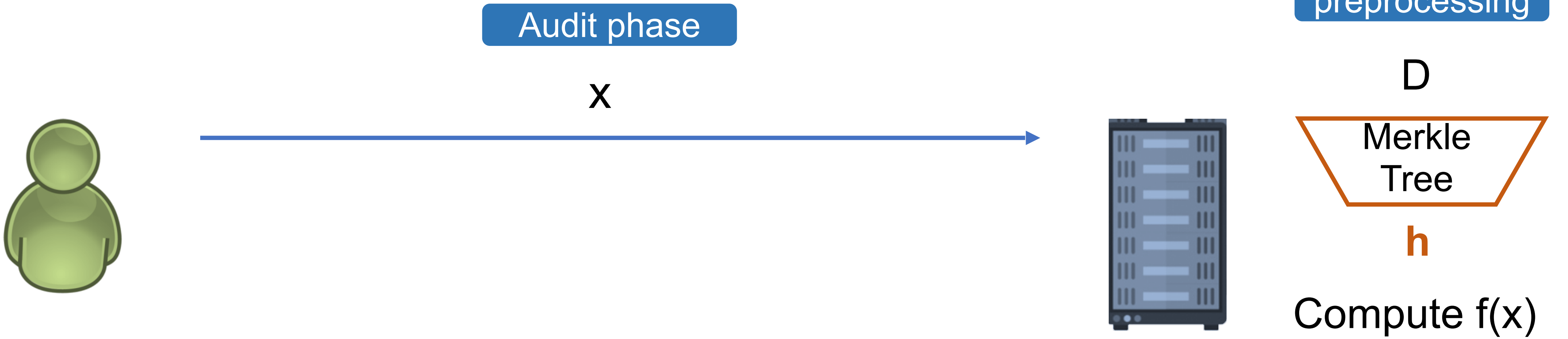
h

Poly-log verification

What if the verifier also computes f(x)?

☒ <u>no access to D!</u>

✓ Prover only needs to access a poly-log number of blocks D ′ ⊂ D



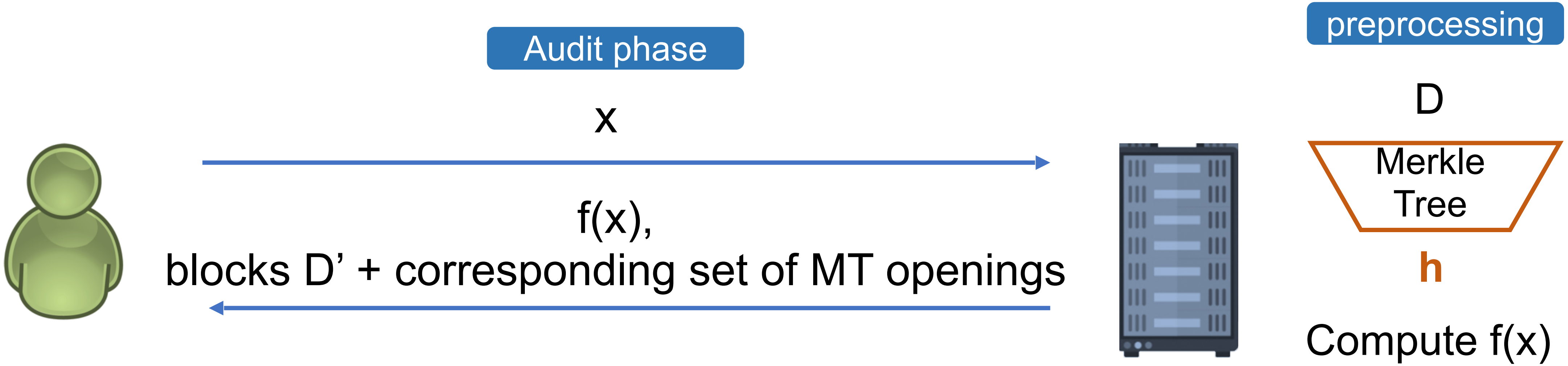
Compute f(x) from D.
Let D’ the blocks read.

Poly-log verification

What if the verifier also computes f(x)?

✗ <u>no access to D!</u>

Prover only needs to access a poly-log number of blocks $D' \subset D$



Audit phase

x

f(x),
blocks D’ + corresponding set of MT openings

preprocessing

D

Merkle Tree

h

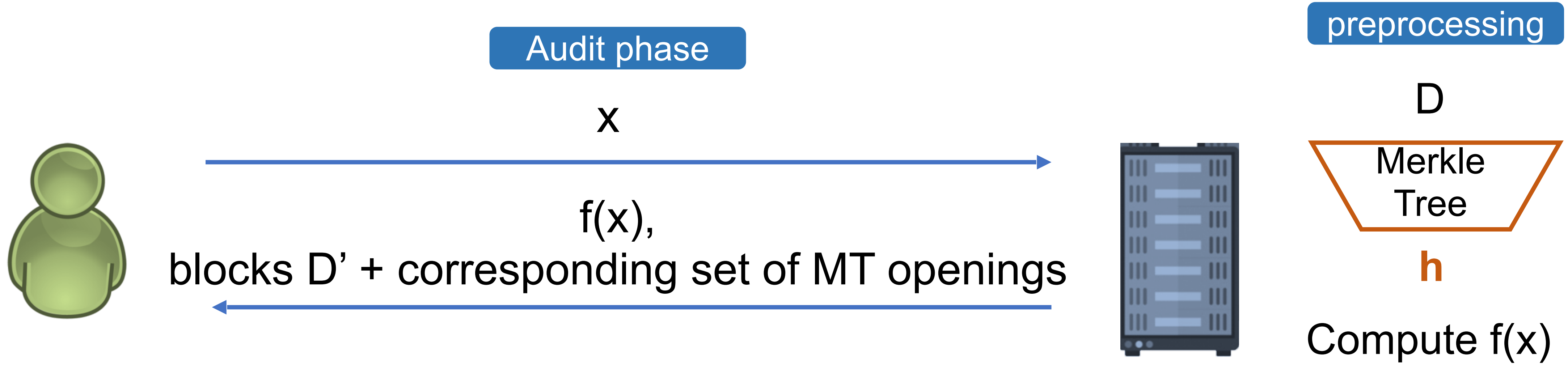
Compute f(x)
from D.
Let D’ the blocks
read.

Poly-log verification

What if the verifier also computes f(x)?

✖ <u>no access to D!</u>

Prover only needs to access a poly-log number of blocks $D' \subset D$



Audit phase

x

f(x),
blocks D' + corresponding set of MT openings

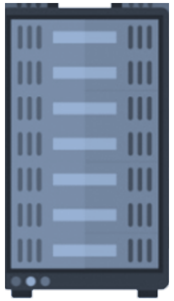
preprocessing

Compute f(x) from D.
Let D' the blocks read.

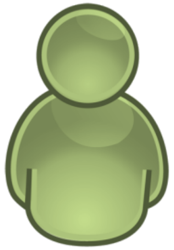
- check proofs
- compute y from D'
- check y ?= f(x)

poly-log verification

Final Construction (Example: 1 File)

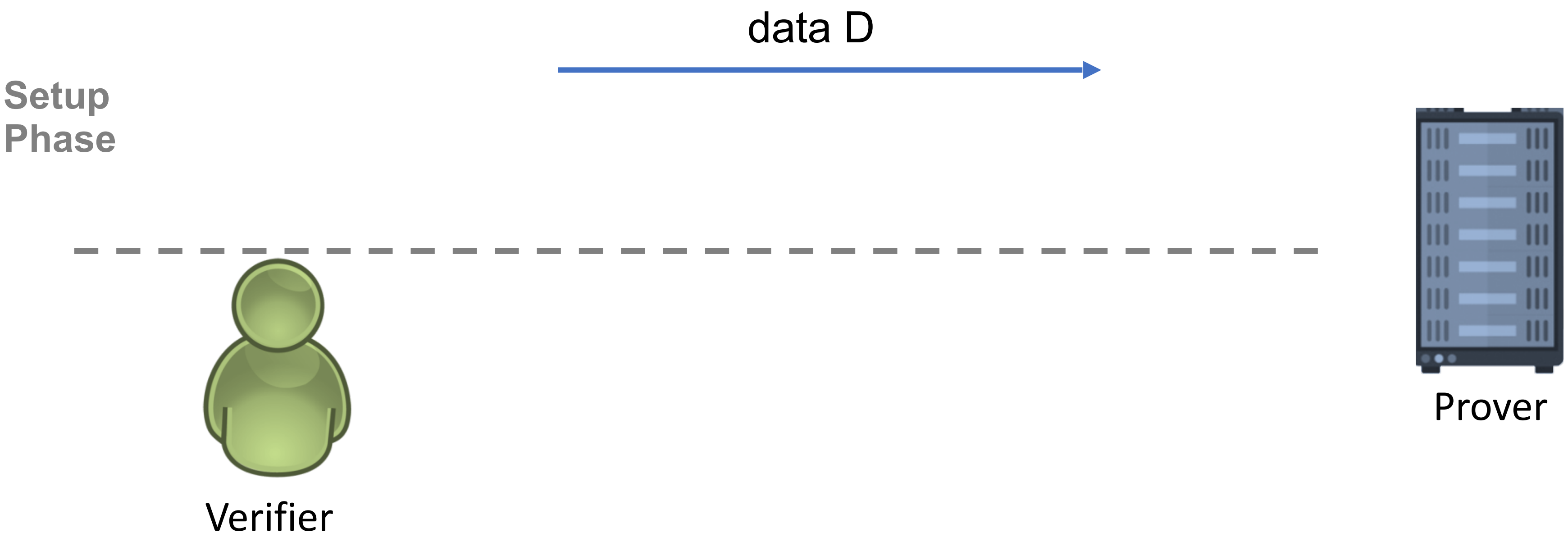


Prover



Verifier

Final Construction (Example: 1 File)



data D

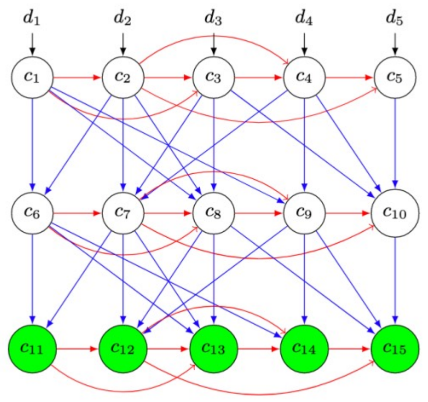
Setup Phase

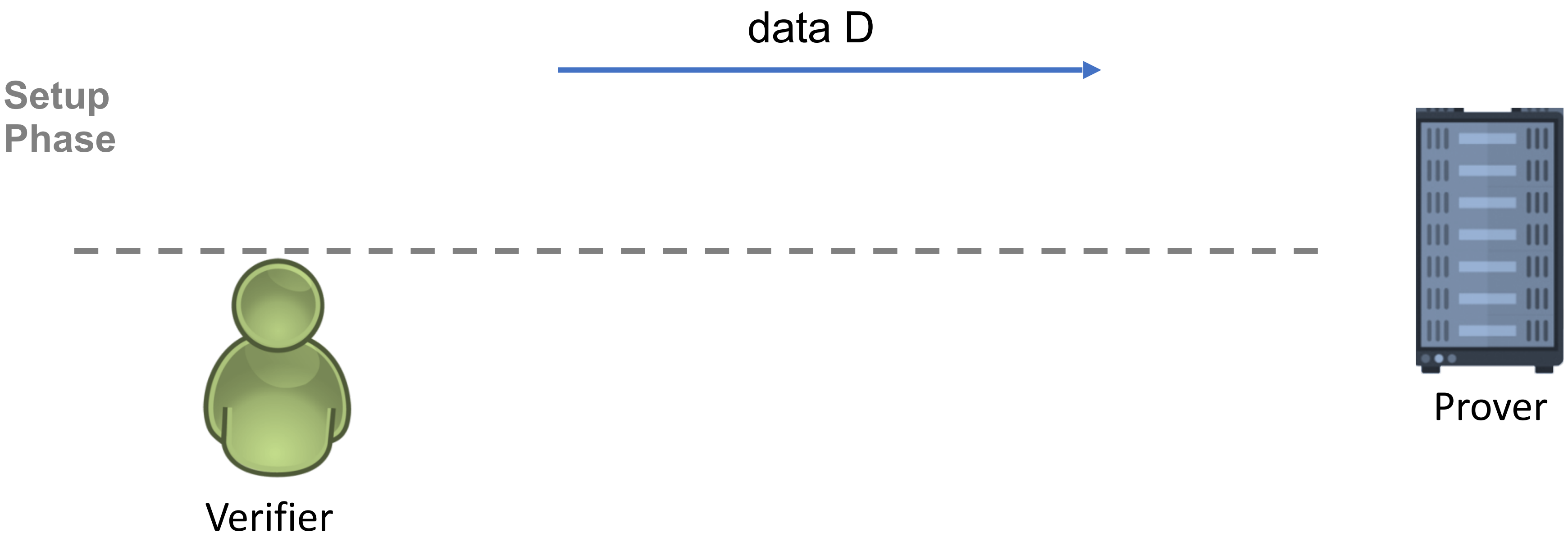
Verifier

Prover

Final Construction (Example: 1 File)

preprocessing





data D

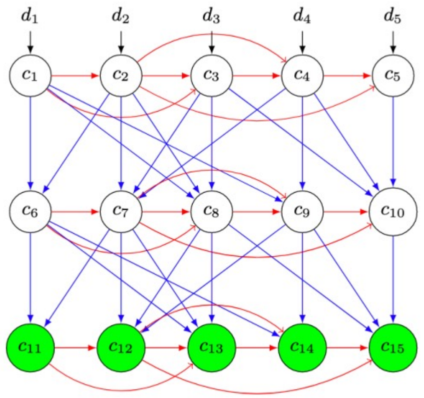
Setup Phase

Verifier

Prover

Final Construction (Example: 1 File)

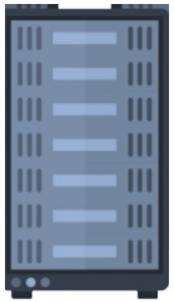
preprocessing



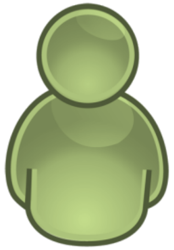
xor D = f(X)

data D

Setup Phase



Prover



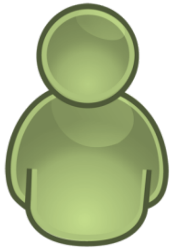
Verifier

Final Construction (Example: 1 File)

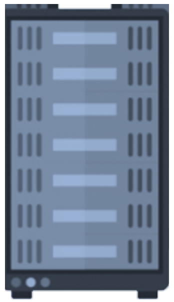
preprocessing

data D

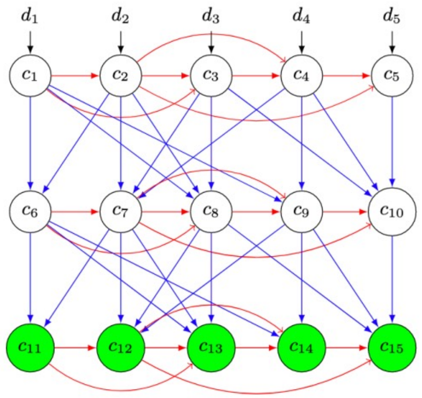
Setup Phase



Verifier



Prover



xor D = f(X)

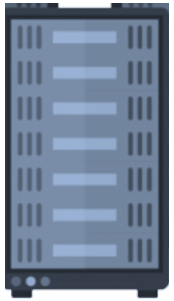
Compute from f(X) data structure D

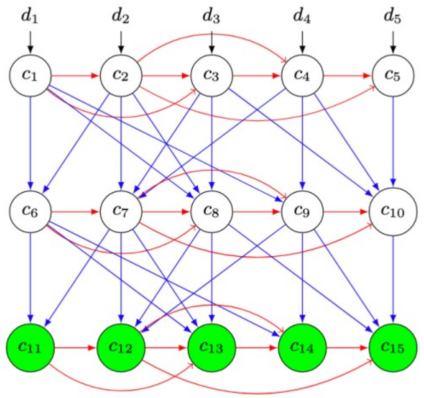
Final Construction (Example: 1 File)

preprocessing

Setup Phase

data D

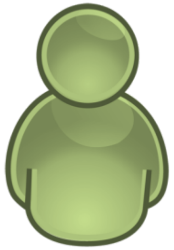

Prover



xor D = f(X)

Compute from f(X) data structure D

Merkle-tree on D’s blocks


Verifier

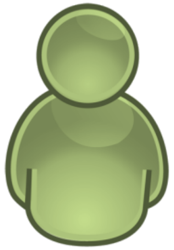
Final Construction (Example: 1 File)

preprocessing

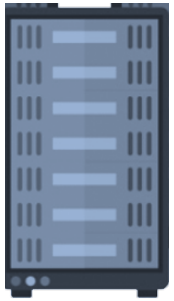
Setup Phase

data D

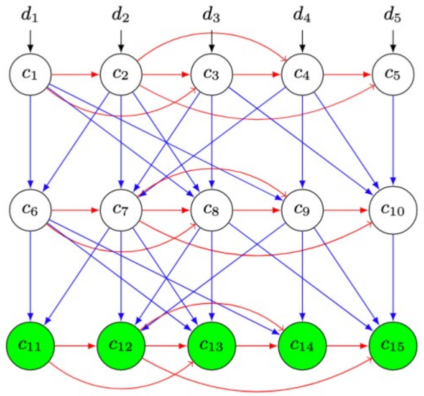
Publish merle-tree root h (digest)



Verifier



Prover



xor D = f(X)

Compute from f(X) data structure D

Merkle-tree on D’s blocks

Final Construction (Example: 1 File)

preprocessing

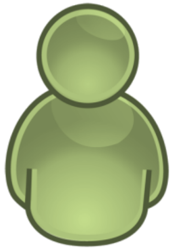
Setup Phase

data D

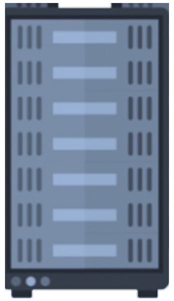
Publish merle-tree root h (digest)

Audit Phase

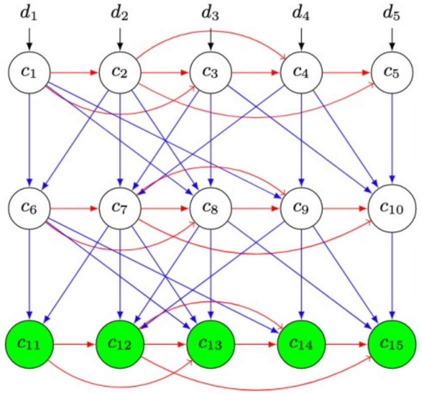
x



Verifier



Prover



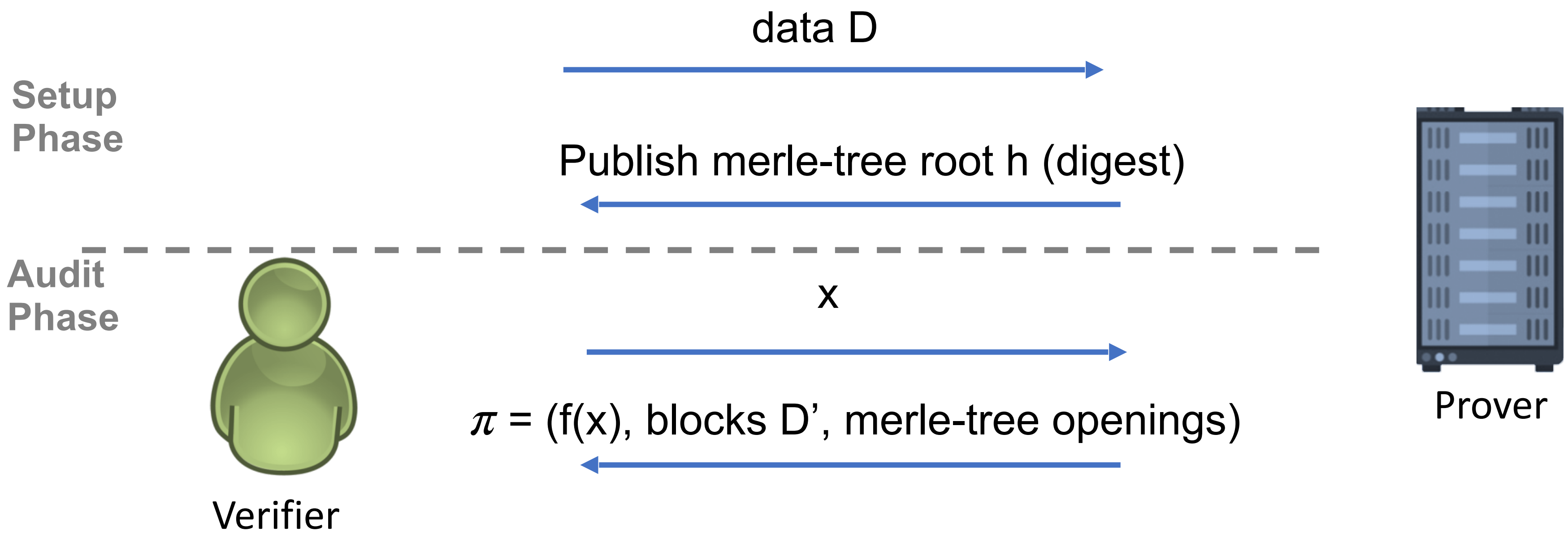
xor D = f(X)

Compute from f(X) data structure D

Merkle-tree on D’s blocks

Final Construction (Example: 1 File)

preprocessing



Setup Phase

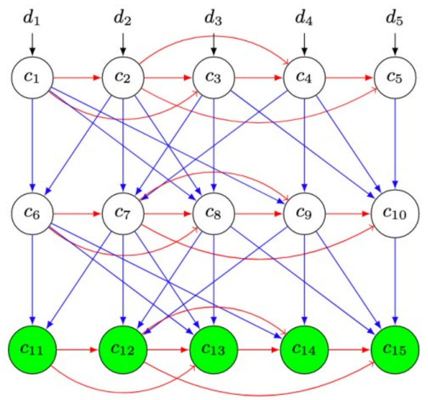
data D

Publish merle-tree root h (digest)

Audit Phase

x

π = (f(x), blocks D’, merle-tree openings)



xor D = f(X)

↓

Compute from f(X) data structure D

↓

Merkle-tree on D’s blocks

Final Construction (Example: 1 File)

preprocessing

Setup Phase

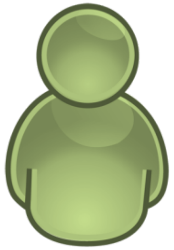
data D

Publish merle-tree root h (digest)

Audit Phase

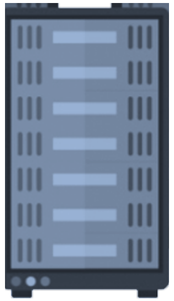
x

π = (f(x), blocks D', merle-tree openings)

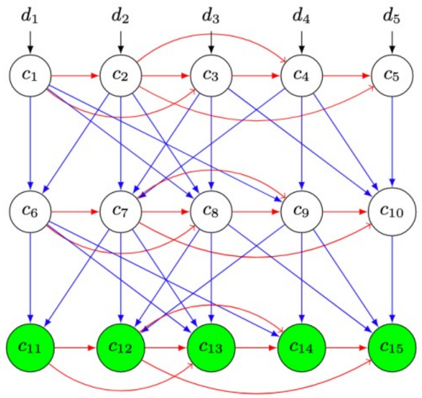


Verifier

- check proofs
- compute y from D'
- check y ?= f(x)



Prover



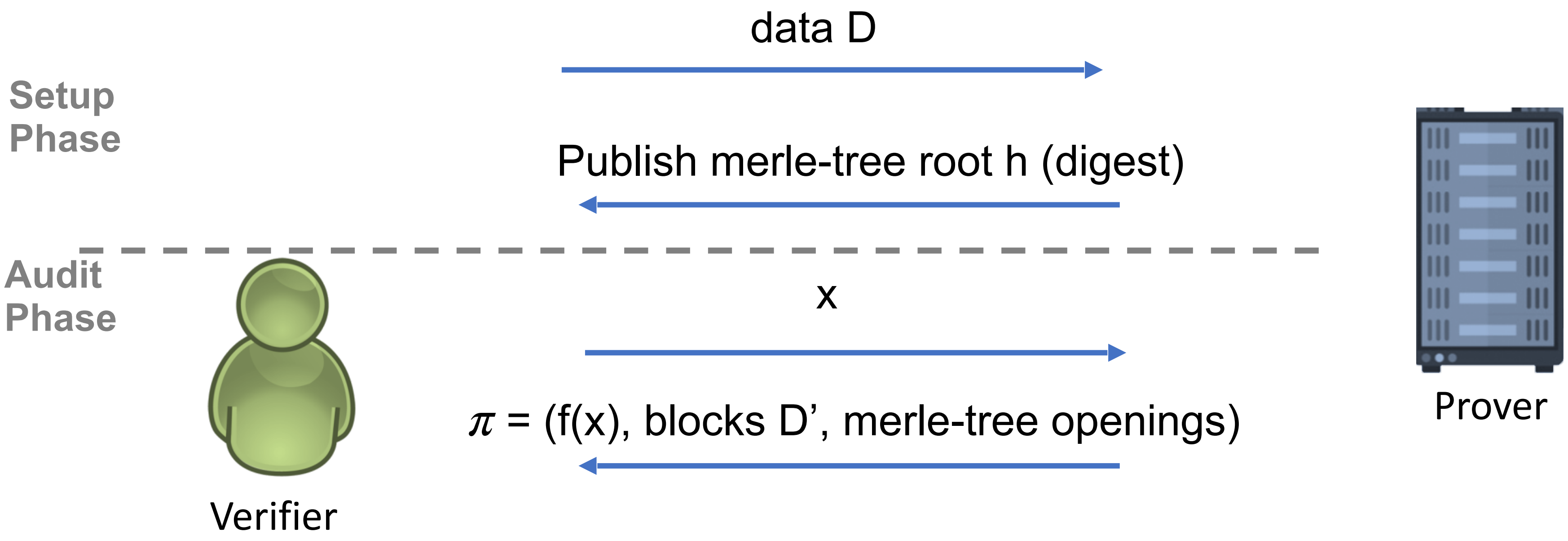
xor D = f(X)

Compute from f(X) data structure D

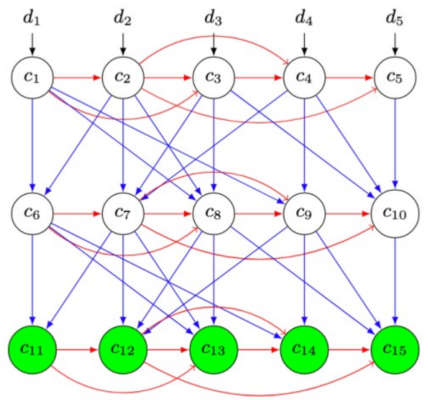
Merkle-tree on D's blocks

Final Construction (Example: 1 File)

preprocessing



- check proofs
- compute y from D'
- check y ?= f(x)



xor D = f(X)

Compute from f(X) data structure D

Merkle-tree on D's blocks

✓ Memory usage ≈ u*|f(X)| for large number of files

✓ Efficient proving and verification time

Absolute memory usage

[Fisch'18]

Ours

Absolute memory usage

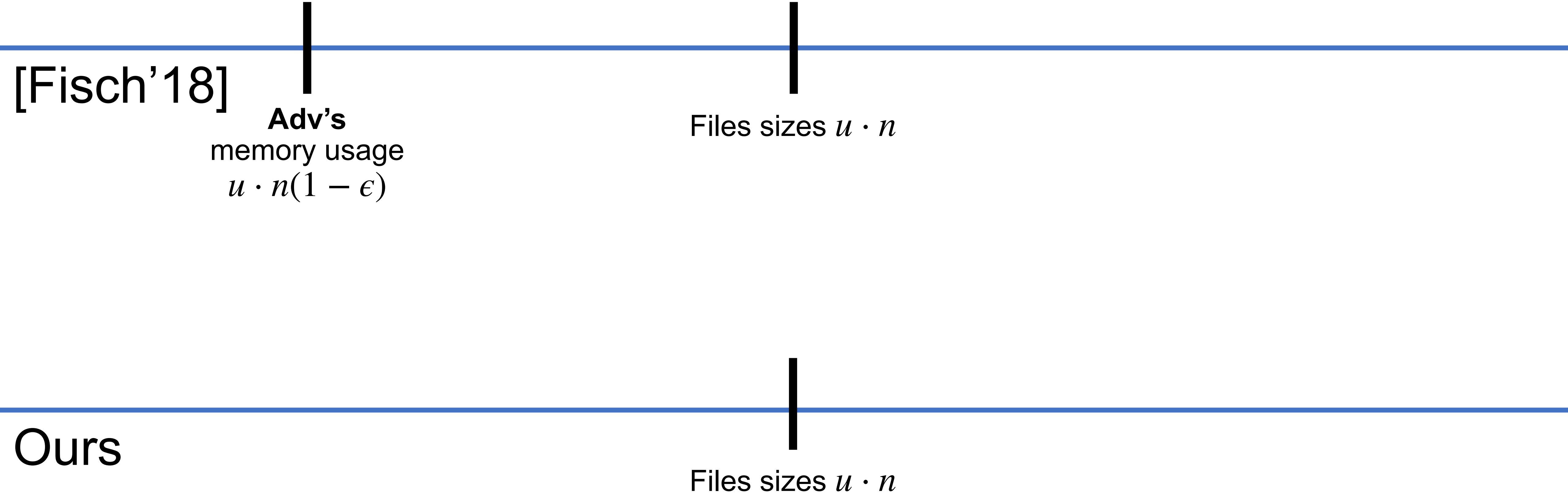
[Fisch'18]

Files sizes $u \cdot n$

Ours

Files sizes $u \cdot n$

Absolute memory usage



[Fisch'18]

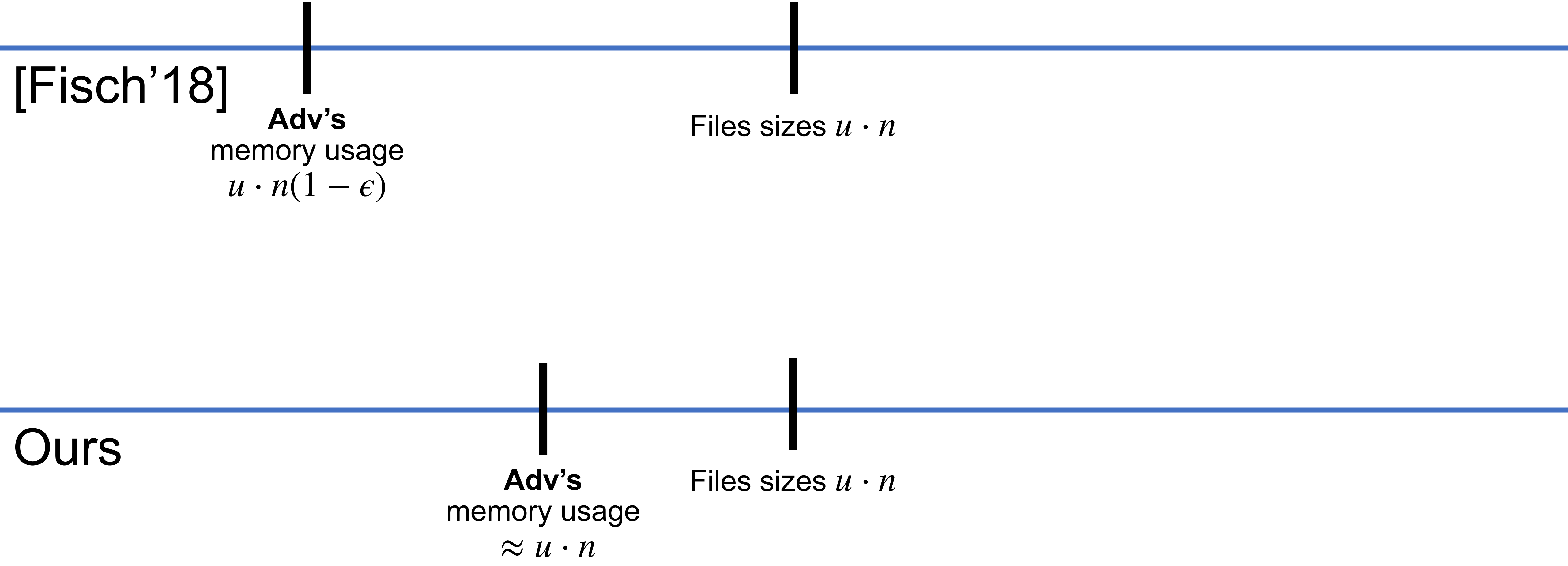
Adv's
memory usage
$u \cdot n(1-\epsilon)$

Files sizes $u \cdot n$

Ours

Files sizes $u \cdot n$

Absolute memory usage



[Fisch'18]

Adv's memory usage $u \cdot n(1-\epsilon)$

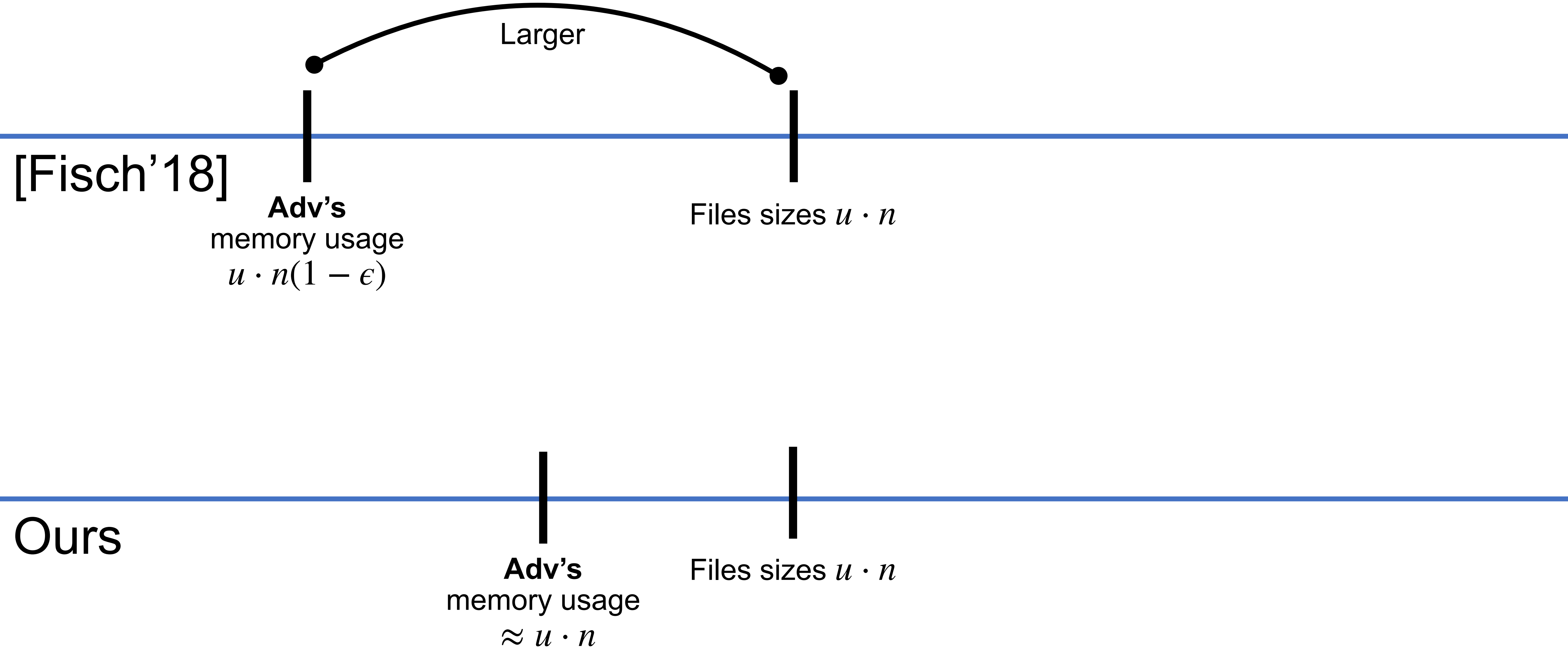
Files sizes $u \cdot n$

Ours

Adv's memory usage $\approx u \cdot n$

Files sizes $u \cdot n$

Absolute memory usage



Larger

[Fisch’18]

Adv’s
memory usage
$u \cdot n(1-\epsilon)$

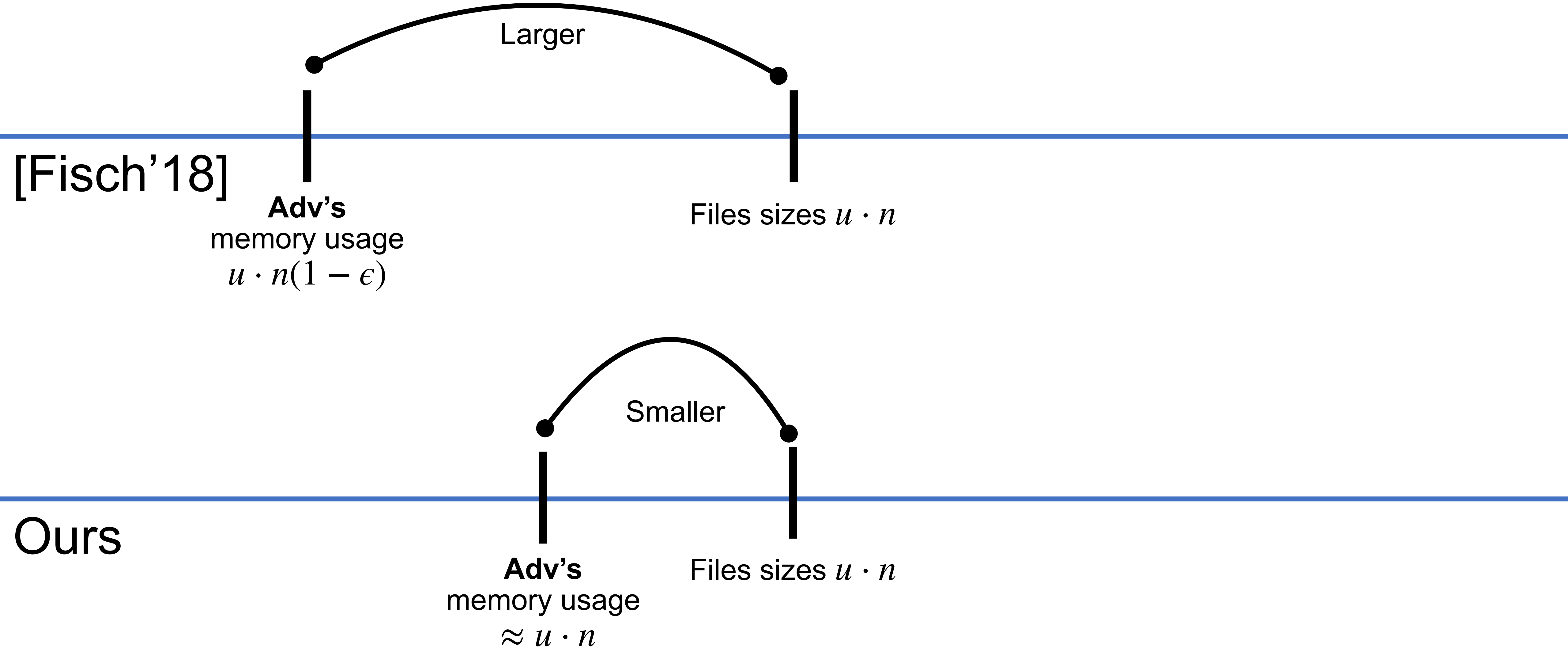
Files sizes $u \cdot n$

Ours

Adv’s
memory usage
$\approx u \cdot n$

Files sizes $u \cdot n$

Absolute memory usage



Larger

[Fisch'18]

Adv's
memory usage
$u \cdot n(1-\epsilon)$

Files sizes $u \cdot n$

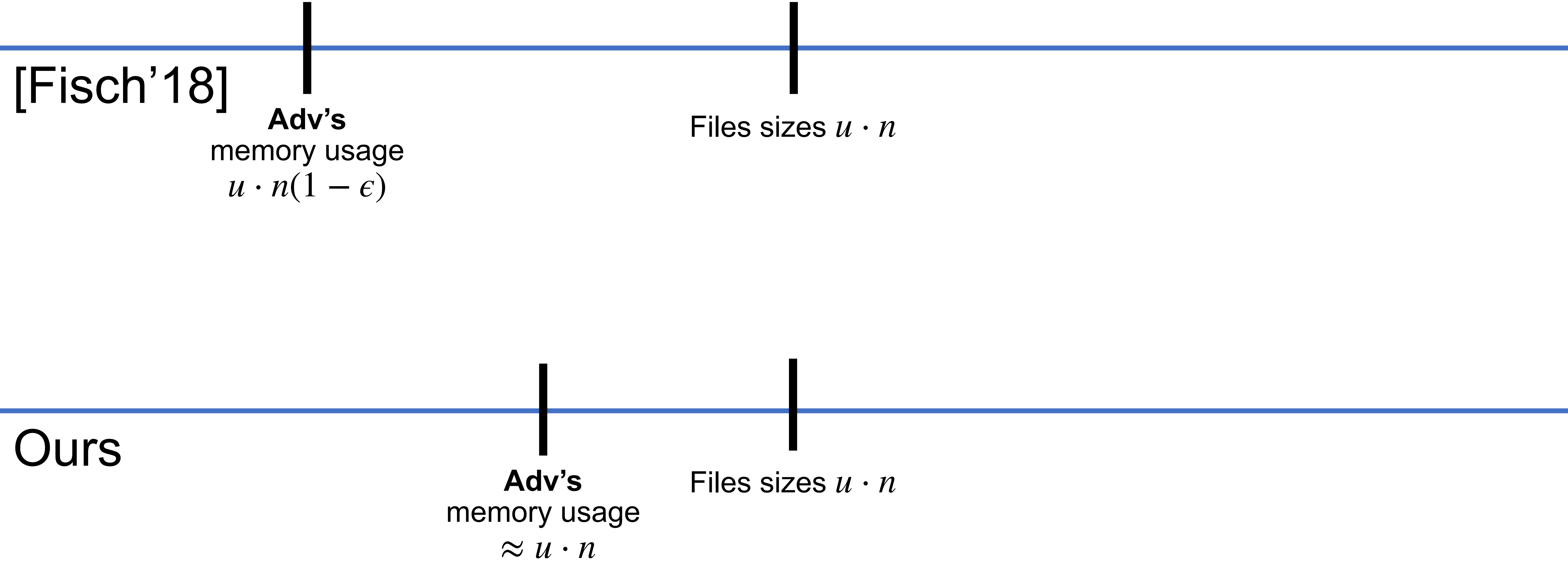
Smaller

Ours

Adv's
memory usage
$\approx u \cdot n$

Files sizes $u \cdot n$

Limitation = Space gap



[Fisch’18]

Adv’s memory usage $u \cdot n(1-\epsilon)$

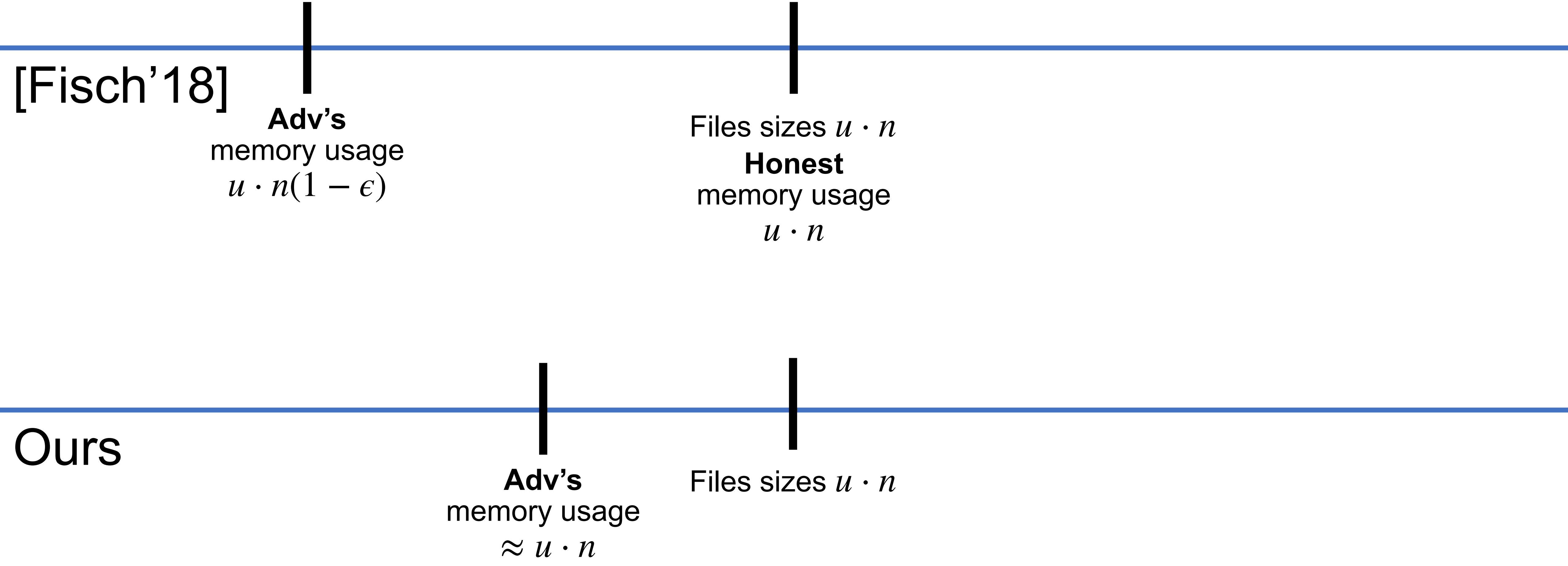
Files sizes $u \cdot n$

Ours

Adv’s memory usage $\approx u \cdot n$

Files sizes $u \cdot n$

Limitation = Space gap



[Fisch’18]

Adv’s memory usage $u \cdot n(1-\epsilon)$

Files sizes $u \cdot n$

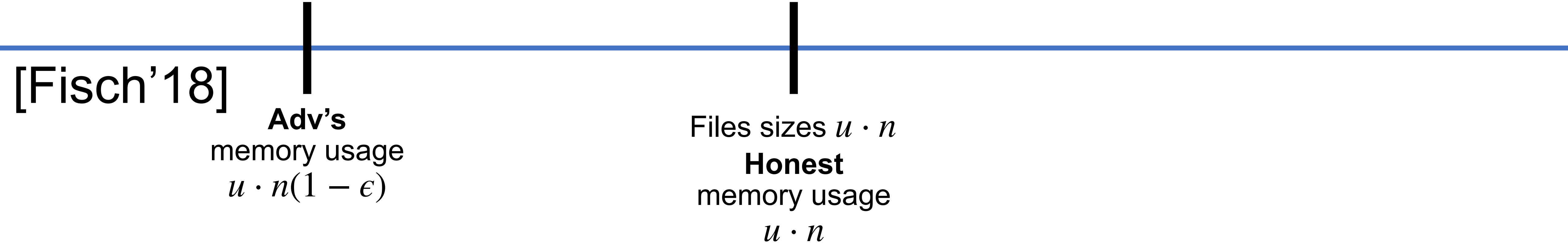
Honest memory usage $u \cdot n$

Ours

Adv’s memory usage $\approx u \cdot n$

Files sizes $u \cdot n$

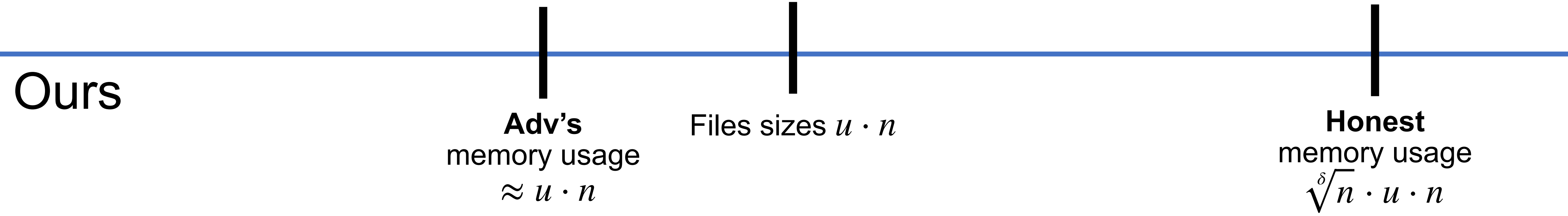
Limitation = Space gap



[Fisch’18]

Adv’s
memory usage
$u \cdot n(1 - \epsilon)$

Files sizes $u \cdot n$
Honest
memory usage
$u \cdot n$



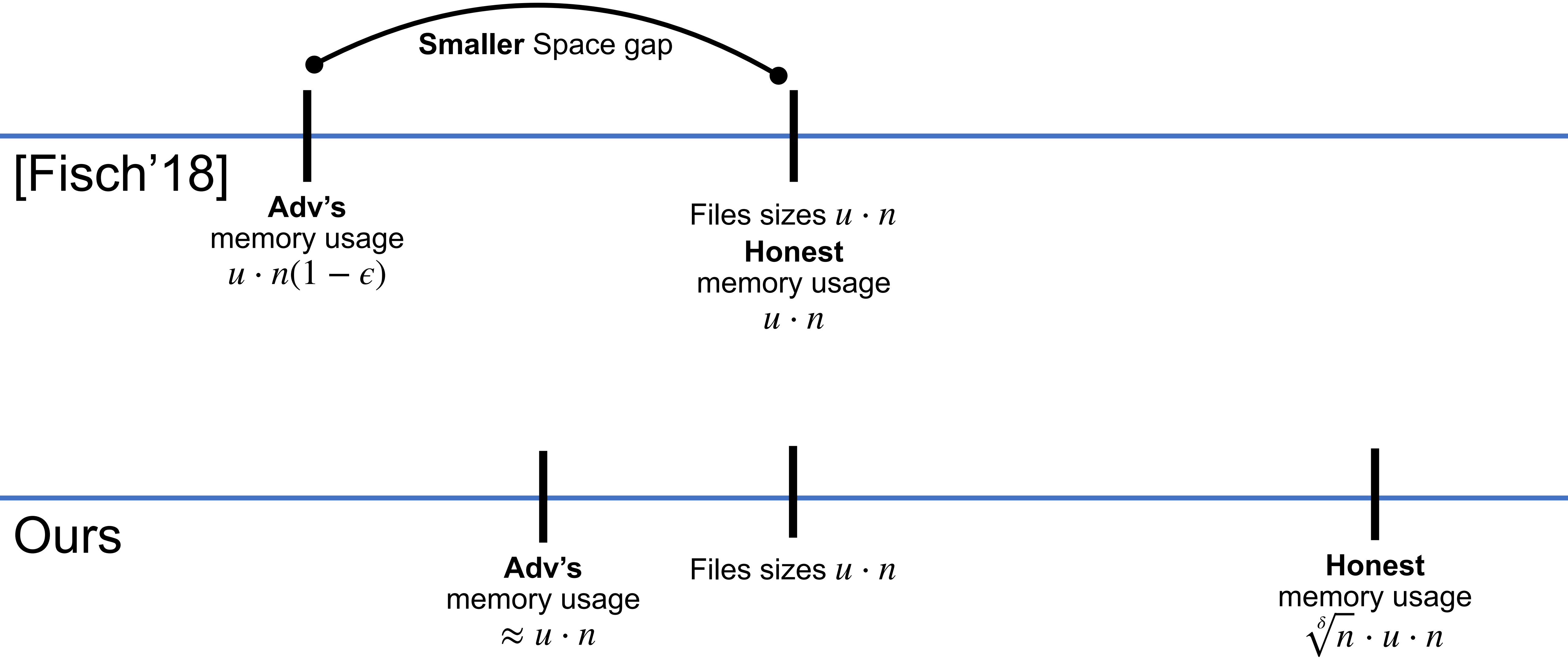
Ours

Adv’s
memory usage
$\approx u \cdot n$

Files sizes $u \cdot n$

Honest
memory usage
$\sqrt[\delta]{n} \cdot u \cdot n$

Limitation = Space gap



Smaller Space gap

[Fisch'18]

Adv's
memory usage
$u \cdot n(1 - \epsilon)$

Files sizes $u \cdot n$
Honest
memory usage
$u \cdot n$

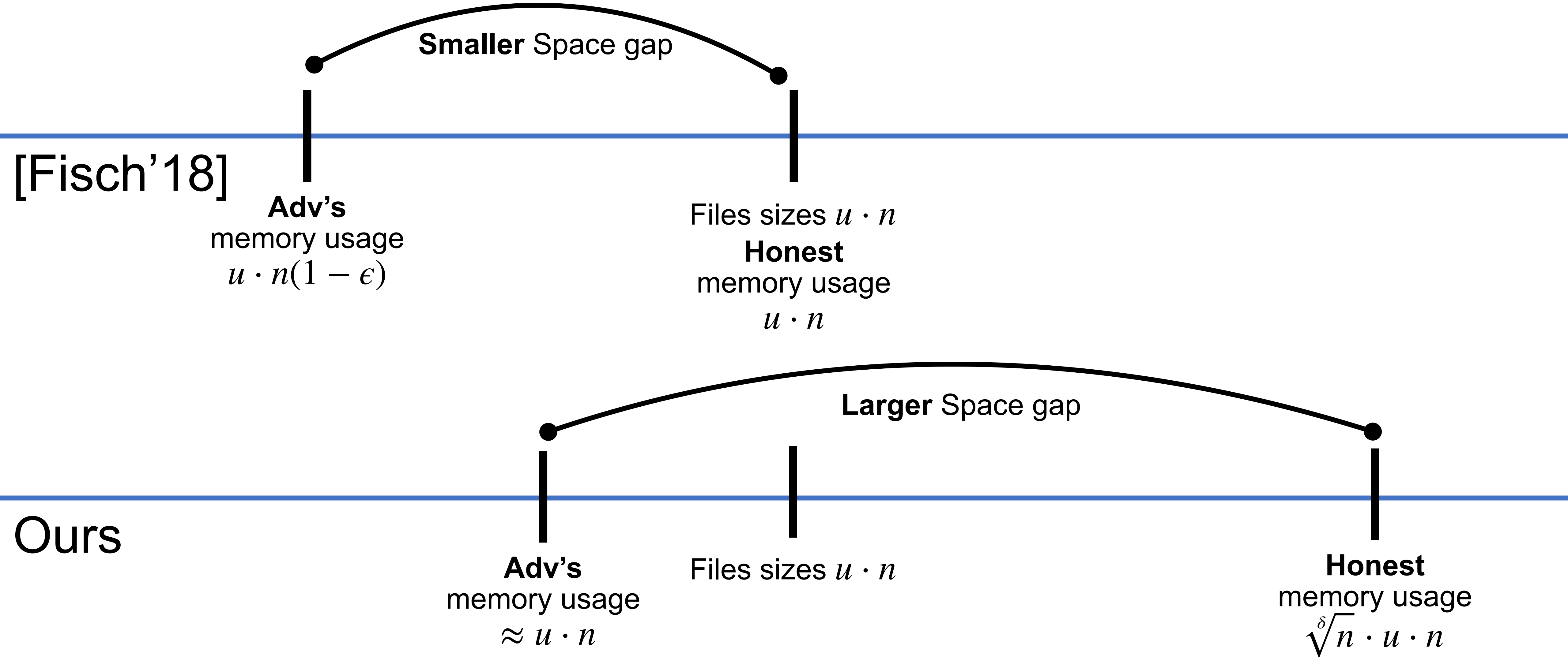
Ours

Adv's
memory usage
$\approx u \cdot n$

Files sizes $u \cdot n$

Honest
memory usage
$\sqrt[\delta]{n} \cdot u \cdot n$

Limitation = Space gap



Smaller Space gap

[Fisch’18]

Adv’s memory usage $u \cdot n(1-\epsilon)$

Files sizes $u \cdot n$

Honest memory usage $u \cdot n$

Larger Space gap

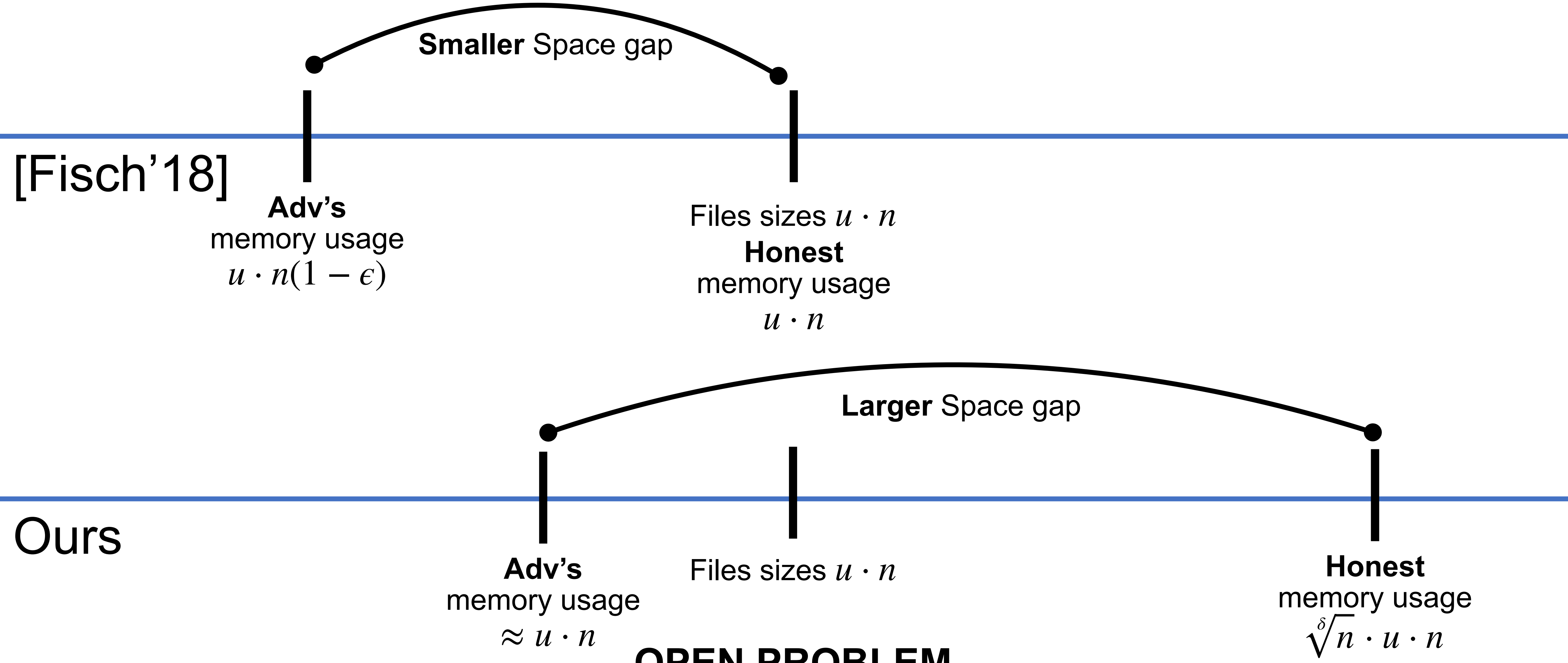
Ours

Adv’s memory usage $\approx u \cdot n$

Files sizes $u \cdot n$

Honest memory usage $\sqrt[\delta]{n} \cdot u \cdot n$

Limitation = Space gap



Smaller Space gap

[Fisch’18]

Adv’s memory usage $u \cdot n(1 - \epsilon)$

Files sizes $u \cdot n$

Honest memory usage $u \cdot n$

Larger Space gap

Ours

Adv’s memory usage $\approx u \cdot n$

Files sizes $u \cdot n$

Honest memory usage $\sqrt[\delta]{n} \cdot u \cdot n$

OPEN PROBLEM

Keep **efficiency** and **high (absolute) memory usage** while **minimising space gap**

THANK YOU!