Parallelizable Delegation from LWE

Cody Freitag[1], Rafael Pass[1,2], Naomi Sirkin[1]

[1]Cornell Tech
[2]Tel-Aviv University

Delegating RAM Computation

Delegating RAM Computation

Verifier **V**

Delegating RAM Computation

Prover **P**

M, x

Verifier **V**

Delegating RAM Computation

Prover **P**

y = M(x)

M, x

Verifier **V**

Delegating RAM Computation

Prover **P**

$y = M(x)$

M, x

$\longleftarrow$

$\longrightarrow$

y, π

Verifier **V**

Delegating RAM Computation

Prover **P**

y = M(x)

M, x

y, π

Verifier **V**

Completeness:

- If y = M(x),
  π convinces **V**.

Delegating RAM Computation

Prover **P**

y = M(x)

M, x

y, π

Verifier **V**

Completeness:

- If y = M(x), π convinces **V**.

Soundness:

- If y != M(x), PPT **P*** **cannot generate** a convincing π.

Delegating RAM Computation

Prover **P**

$y = M(x)$

M, x

y, π

Verifier **V**

Completeness:

- If $y = M(x)$, π convinces **V**.

Soundness:

- If $y \ne M(x)$, PPT **P*** **cannot generate** a convincing π.

Succinctness:

- $|\pi|$ and runtime of **V** are at most **polylog(T)** where $T = \mathrm{Time}_M(x)$.

Delegating RAM Computation

Prover **P**

y = M(x)

M, x

y, π

Verifier **V**

Ignoring poly(λ) terms

Completeness:

- If y = M(x), π convinces **V**.

Soundness:

- If y != M(x), PPT **P*** **cannot generate** a convincing π.

Succinctness:

- |π| and runtime of **V** are at most **polylog(T)** where T = $Time_M(x)$.

Delegating RAM Computation

Prover **P**

$y = M(x)$

Verifier **V**

M, x ←

→ y, π

Completeness:

- If $y = M(x)$, π convinces **V**.

Soundness:

- If $y \mathrel{!}= M(x)$, PPT **P*** **cannot generate** a convincing π.

Succinctness:

- $|\pi|$ and runtime of **V** are at most **polylog(T)** where $T = \mathrm{Time}_M(x)$.

Delegating RAM Computation

CRS

Prover **P**

y = M(x)

M, x

←

→

y, π

Verifier **V**

Completeness:

- If y = M(x), π convinces **V**.

Soundness:

- If y != M(x), PPT **P*** **cannot generate** a convincing π.

Succinctness:

- |π| and runtime of **V** are at most **polylog(T)** where T = Time_M(x).

Delegating RAM Computatio

|CRS| <= polylog(T)

CRS

Prover **P**

$y = M(x)$

M, x

y, π

Verifier **V**

Completeness:

- If $y = M(x)$, π convinces **V**.

Soundness:

- If $y \mathrel{!}= M(x)$, PPT **P*** **cannot generate** a convincing π.

Succinctness:

- |π| and runtime of **V** are at most **polylog(T)** where $T = \text{Time}_M(x)$.

What does the prover do?

What does the prover do?

Wall-clock time

What does the prover do?

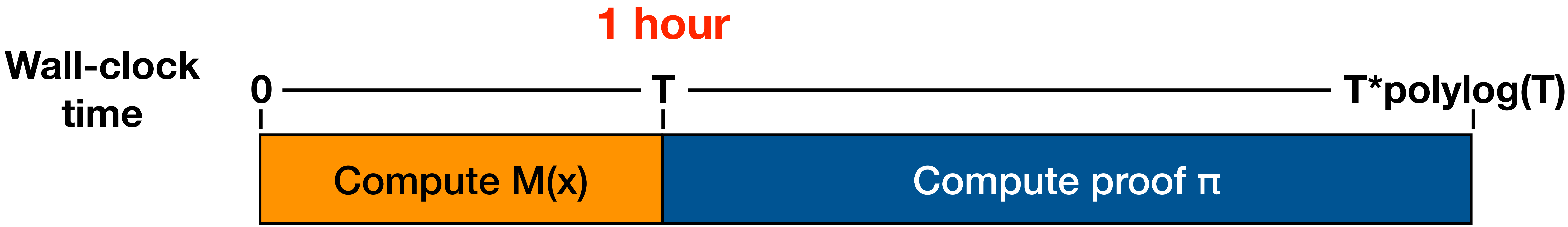
Wall-clock time

0 —————————— **T**

Compute M(x)

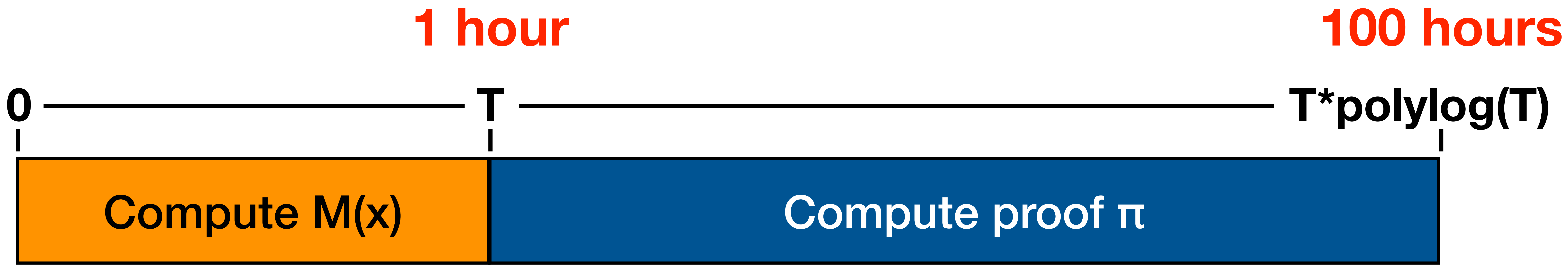
What does the prover do?

What does the prover do?

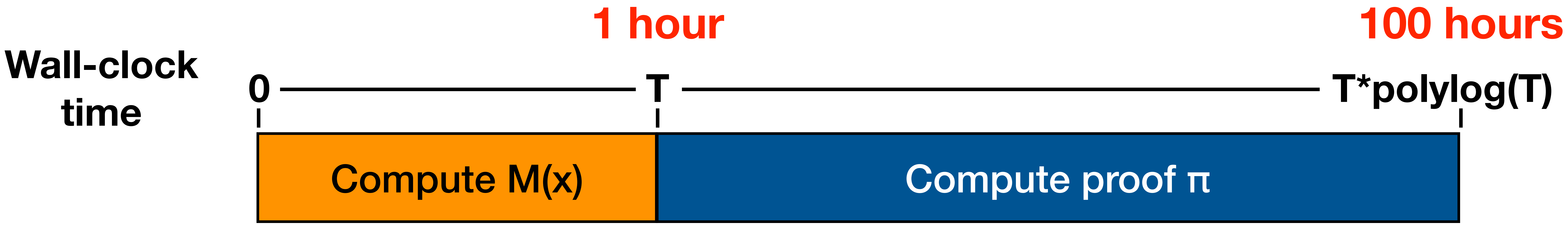


What does the prover do?

Wall-clock time

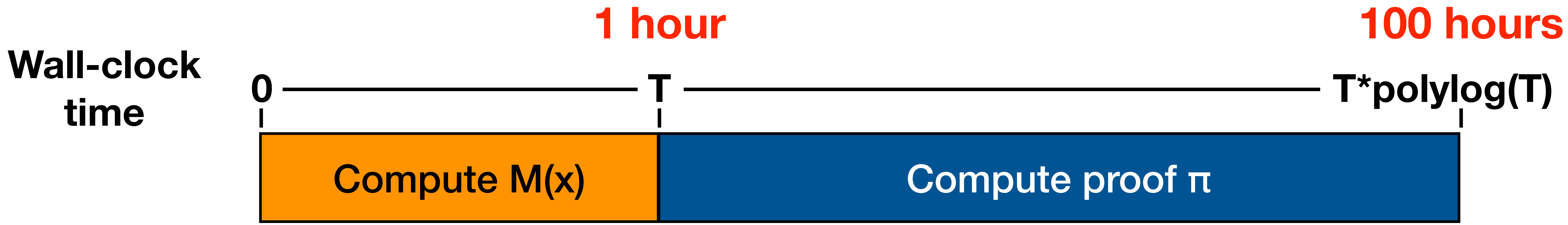


What does the prover do?



Quasi-linear T*polylog(T) prover efficiency:

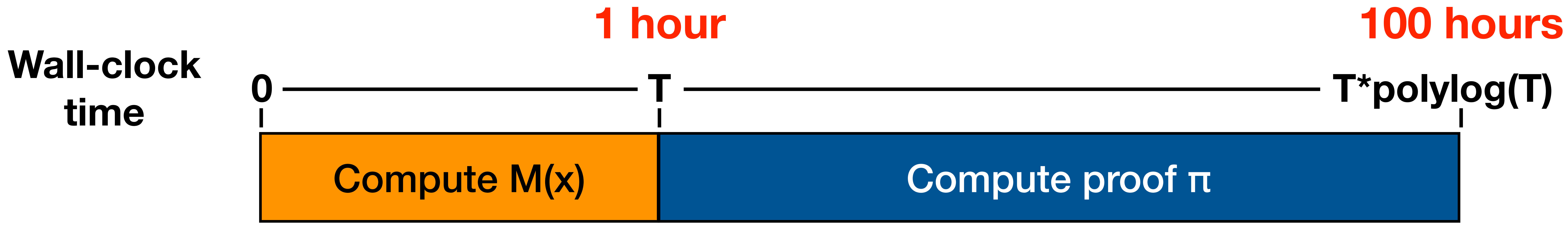
What does the prover do?



Quasi-linear T*polylog(T) prover efficiency:

- from **ROM** or **SNARKs** [M94, BS05, BCCT13].

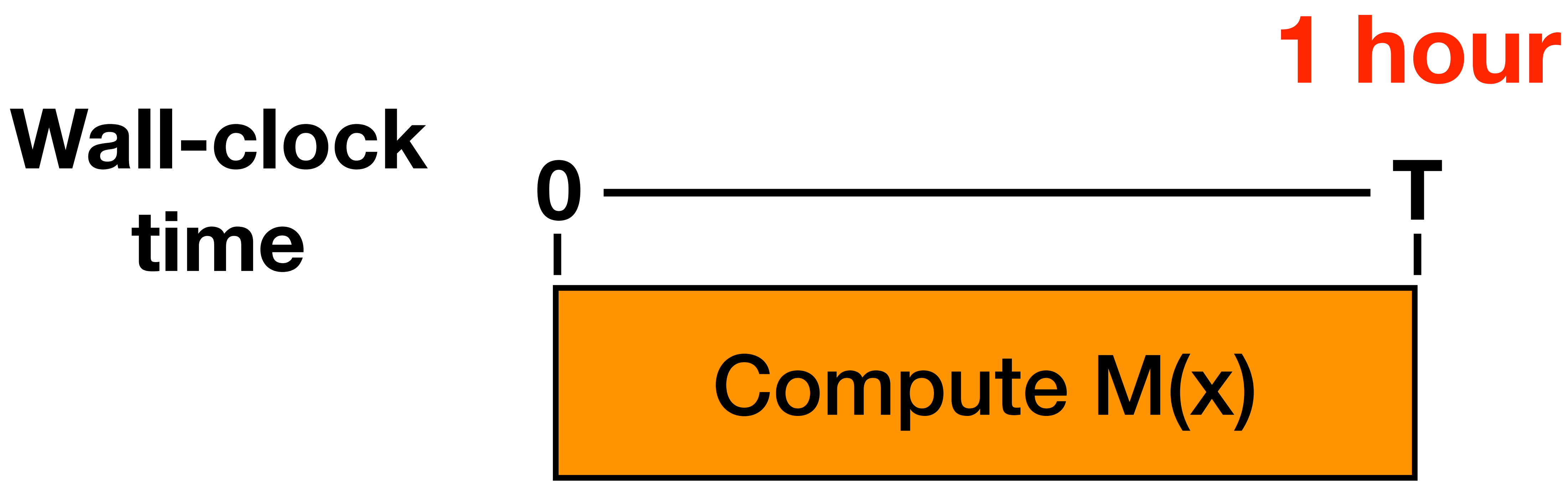
What does the prover do?



Quasi-linear T*polylog(T) prover efficiency:

- from **ROM** or **SNARKs** [M94, BS05, BCCT13].
- from **LWE** [CJJ21].

SPARG:
Parallelizable Delegation for RAM



SPARG:
Parallelizable Delegation for RAM

Succinct Parallelizable Argument

Wall-clock time

1 hour

0 ——— **T**

Compute M(x)

SPARG:
Parallelizable Delegation for RAM

1 hour

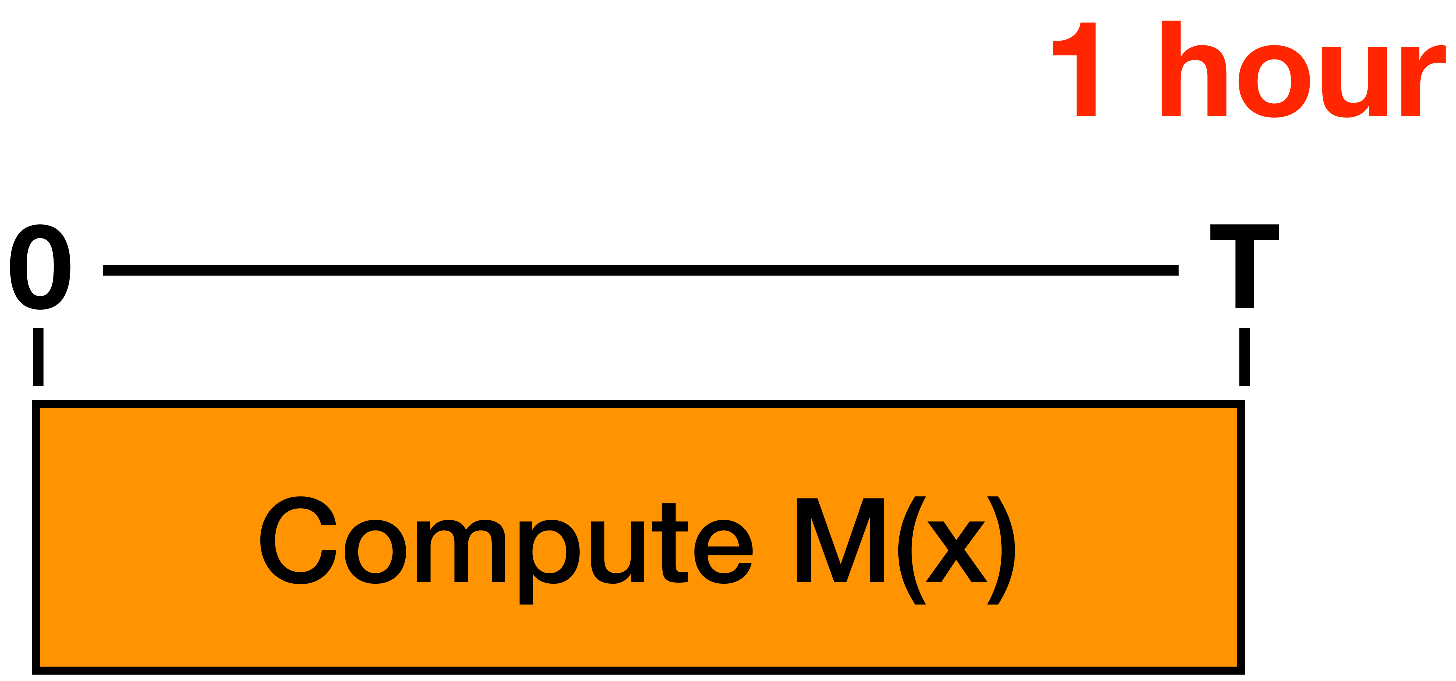
Wall-clock time

0 —————— **T**

Compute M(x)

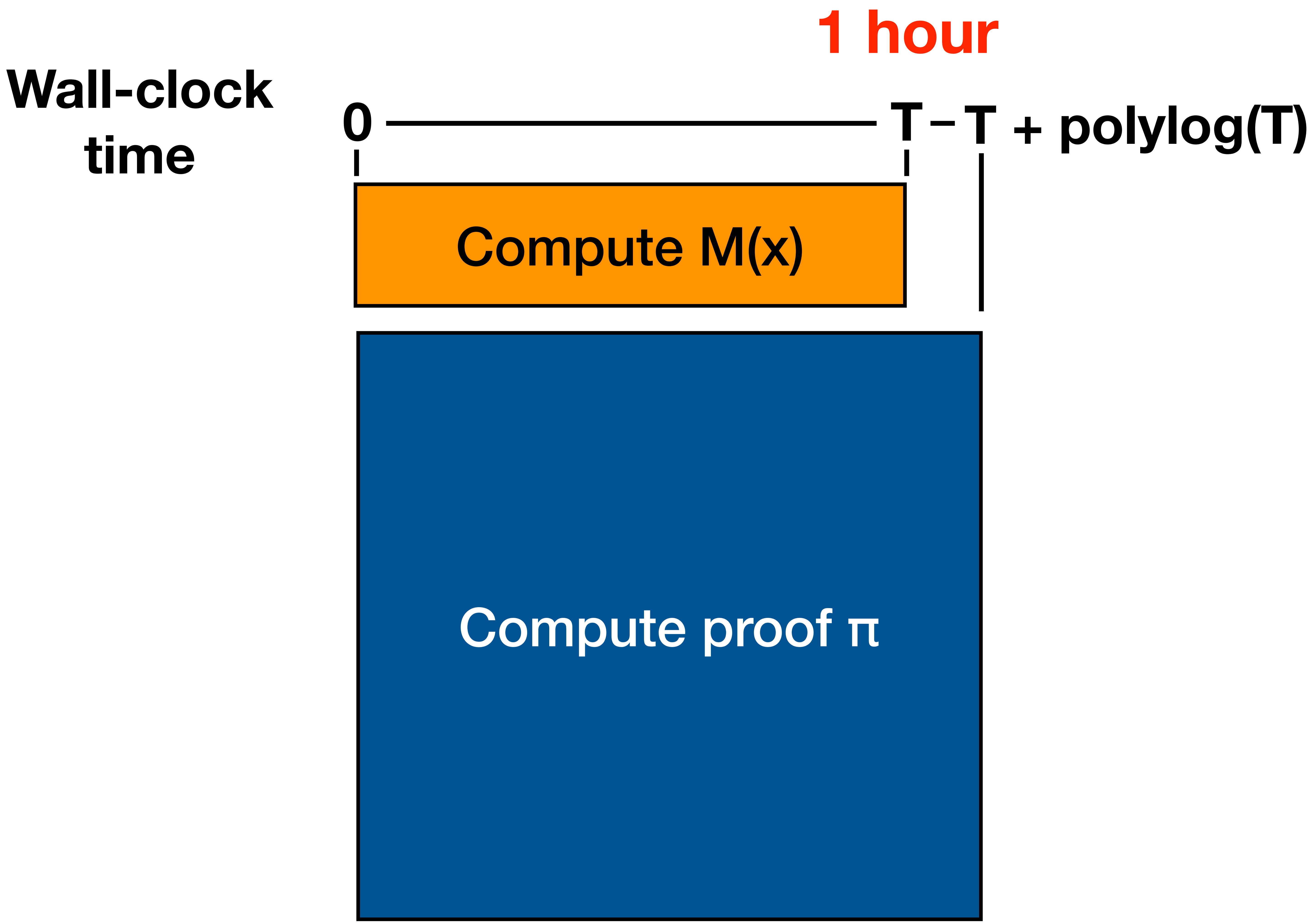
SPARG:
Parallelizable Delegation for RAM

Wall-clock time



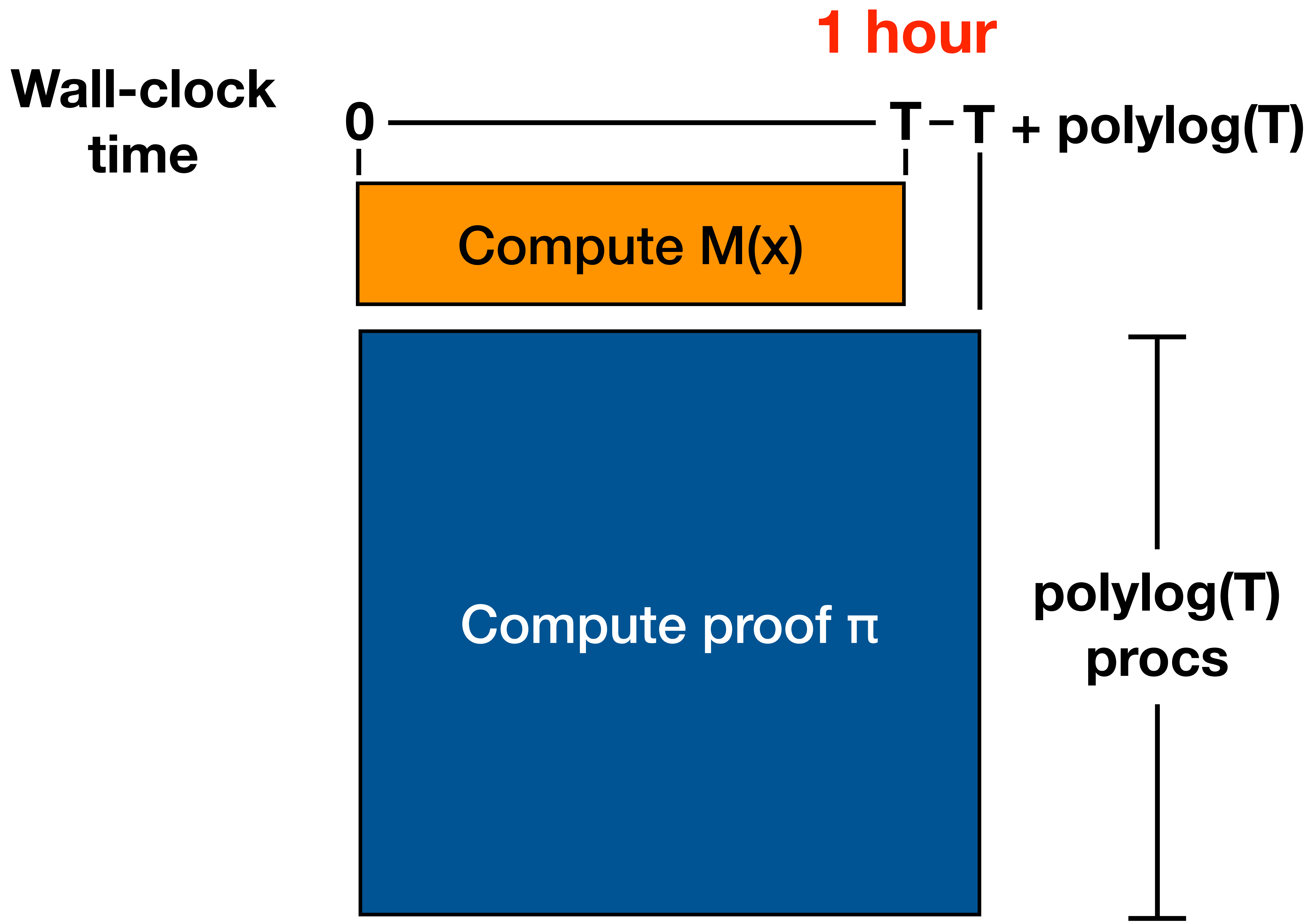
Compute the proof
in parallel
to the computation

SPARG:
Parallelizable Delegation for RAM



Compute the proof
in parallel
to the computation

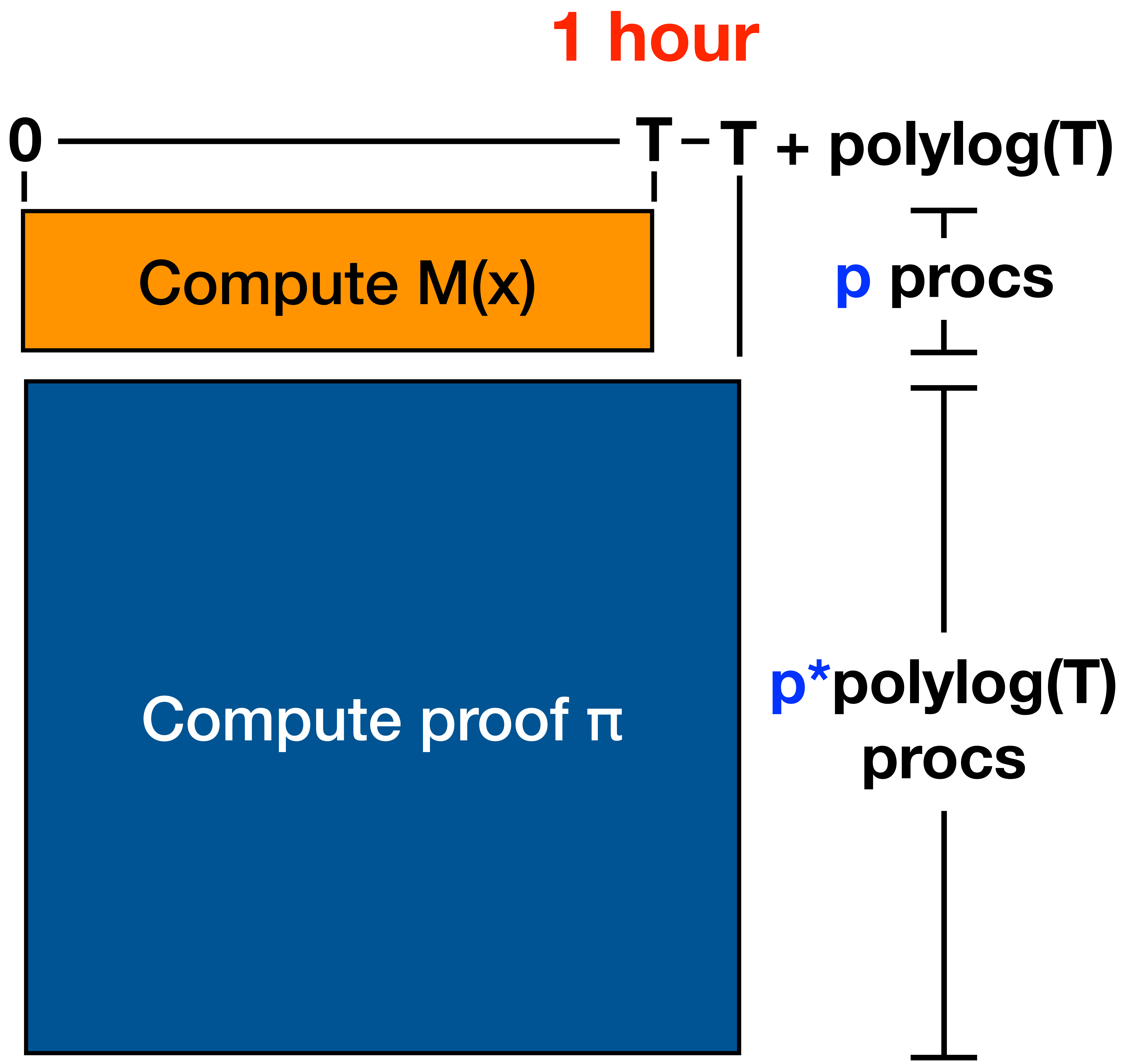
SPARG: Parallelizable Delegation for RAM



Compute the proof
in parallel
to the computation

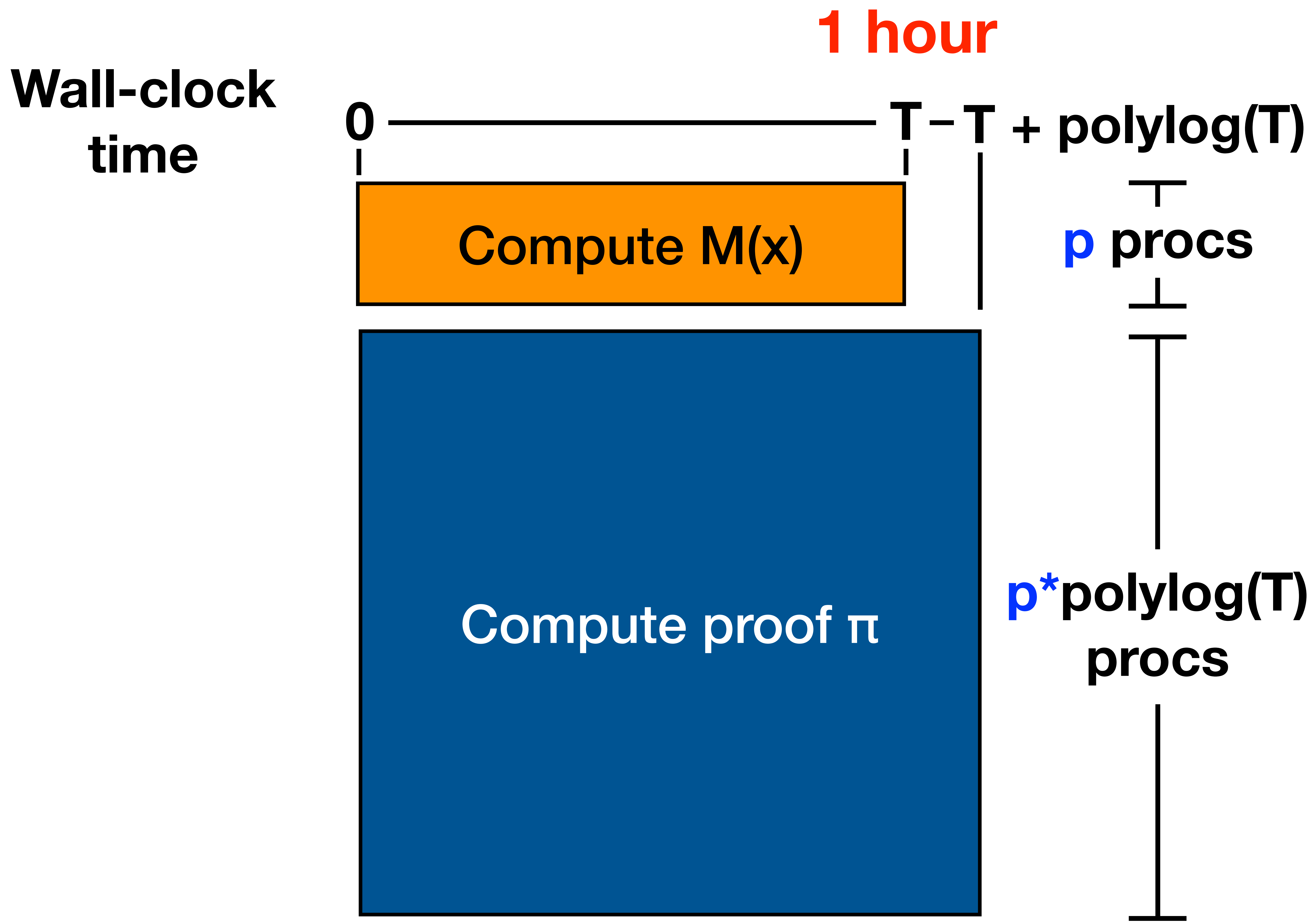
SPARG:
Parallelizable Delegation for PRAM

Wall-clock time



Compute the proof
in parallel
to the computation

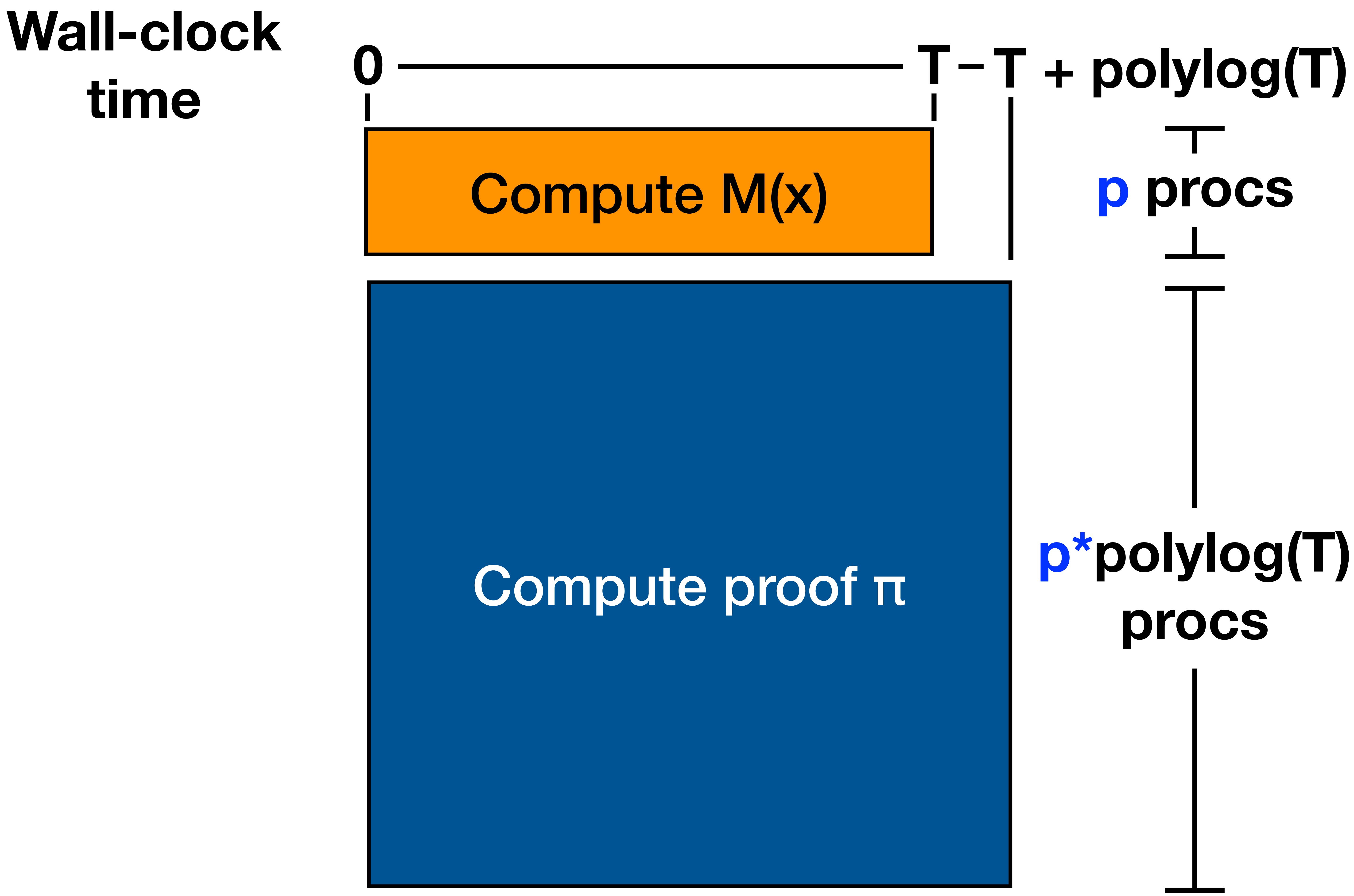
SPARG:
Parallelizable Delegation for PRAM



Compute the proof ***in parallel*** *to the computation*

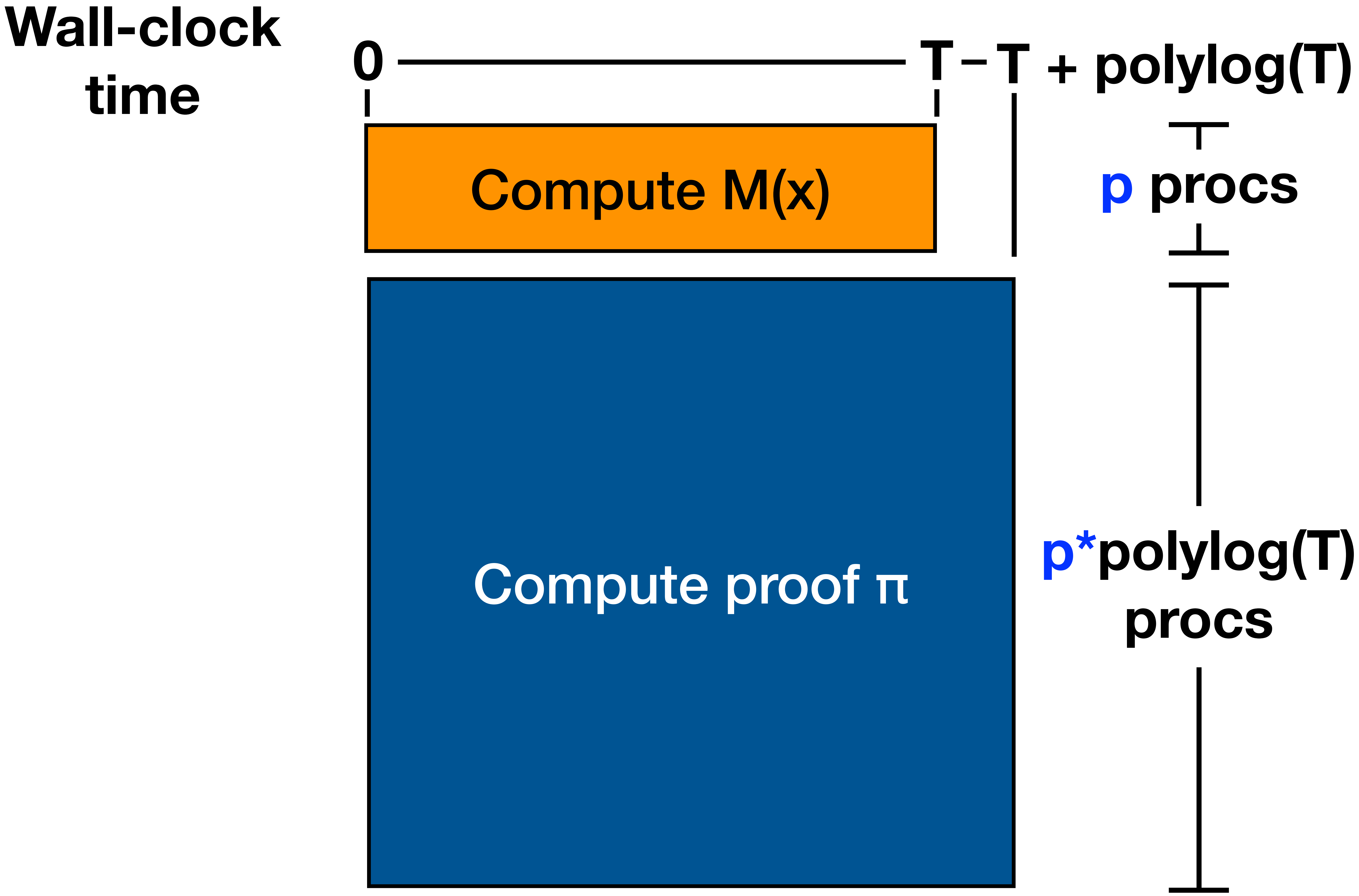
- Only known from **SNARKs** [EFKP20]

Main Result



Wall-clock time
0
T
T + polylog(T)
Compute M(x)
p procs
Compute proof π
p*polylog(T) procs

Main Result



Theorem:

Assuming **LWE**, there exists parallelizable delegation for *any* PRAM computation.

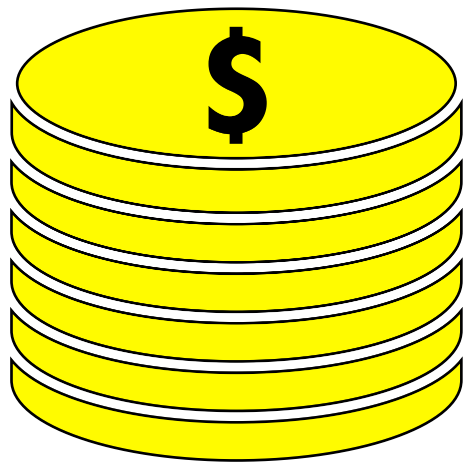
Application:
Verifiable Delay Functions [BBBF18]

Application:
Verifiable Delay Functions [BBBF18]

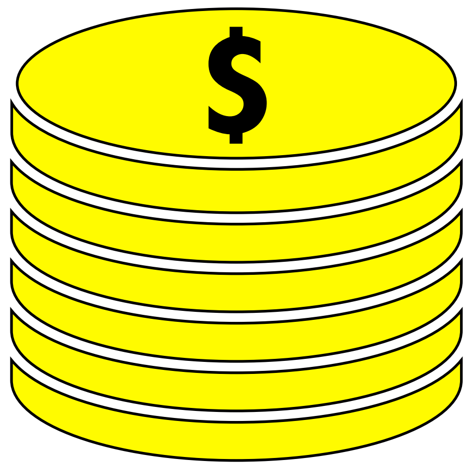
- Verifiable function that **cannot be sped up** with **many processors**

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**



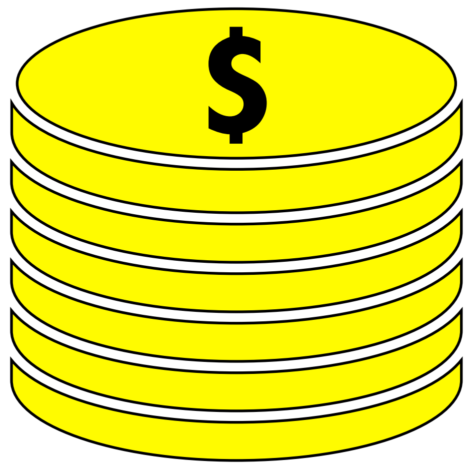
Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors** 

Plain model constructions:

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**

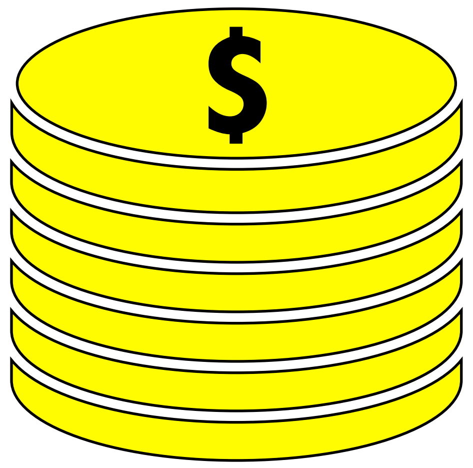


Plain model constructions:

Iterated Sequential Function
+ SNARGs for P
[BBBF18]

Application:
Verifiable Delay Functions [BBBF18]

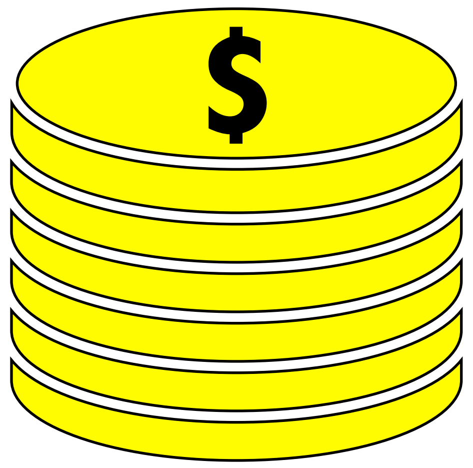
- Verifiable function that **cannot be sped up** with **many processors**



Plain model constructions:

Iterated Sequential Function + SNARGs for P [BBBF18]	*Any* Sequential Function + SNARKs for NP [EFKP20]

Application:
Verifiable Delay Functions [BBBF18]

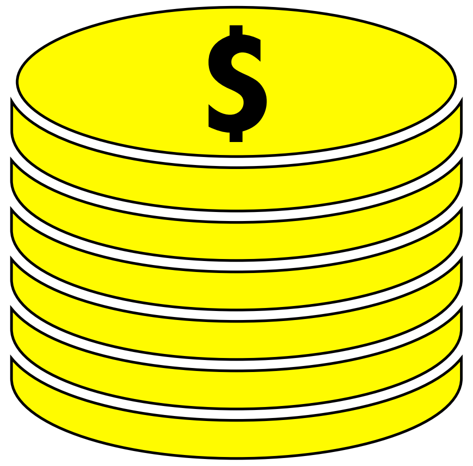
- Verifiable function that **cannot be sped up** with **many processors** 

Plain model constructions:

Iterated Sequential Function + SNARGs for P [BBBF18]	*Any* Sequential Function + SNARKs for NP [EFKP20]	Repeated Squaring + LWE [BCHKLPR22] (Previous talk)

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**



Plain model constructions:

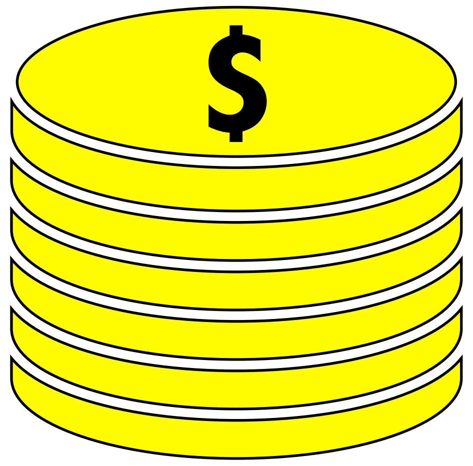
Iterated Sequential Function + SNARGs for P [BBBF18]	*Any* Sequential Function + SNARKs for NP [EFKP20]	Repeated Squaring + LWE [BCHKLPR22] (Previous talk)

SPARG Paradigm:

Any Function + SPARG => Verifiable Function

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**



Plain model constructions:

- Iterated Sequential Function + SNARGs for P [BBBF18]
- *Any* Sequential Function + SNARKs for NP [EFKP20]
- Repeated Squaring + LWE [BCHKLPR22] (Previous talk)

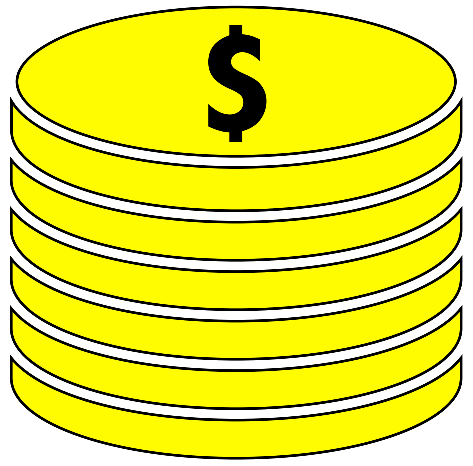
SPARG Paradigm:

Any Function + SPARG => Verifiable Function

preserves parallel running time!

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**

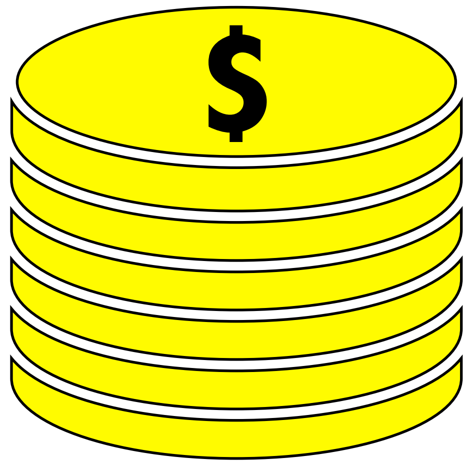


Plain model constructions:

Iterated Sequential Function + SNARGs for P [BBBF18]	*Any* Sequential Function + SNARKs for NP [EFKP20]	Repeated Squaring + LWE [BCHKLPR22] (Previous talk)

Application: Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**



Plain model constructions:

Iterated Sequential Function
+ SNARGs for P
[BBBF18]

Any Sequential Function
+ SNARKs for NP
[EFKP20]

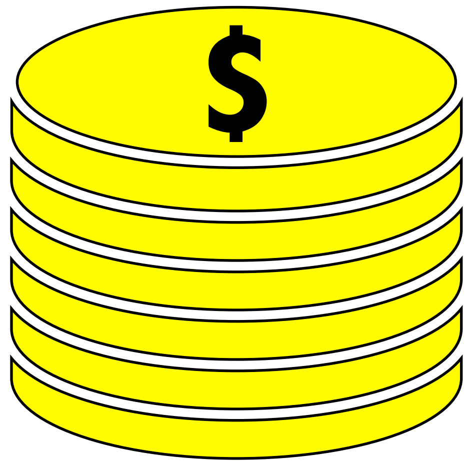
Repeated Squaring
+ LWE [BCHKLPR22]
(Previous talk)

Theorem:

Assuming **LWE** and *any* **sequential function**, there exists a VDF.

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**



Plain model constructions:

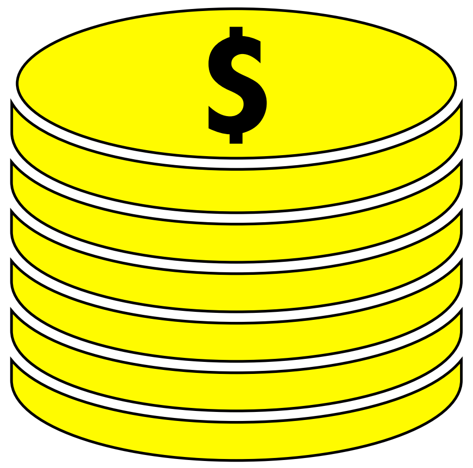
Iterated Sequential Function + SNARGs for P [BBBF18]	*Any* Sequential Function + SNARKs for NP [EFKP20]	Repeated Squaring + LWE [BCHKLPR22] (Previous talk)

Minimal assumption

Theorem:

Assuming **LWE** and *any* **sequential function**, there exists a VDF.

Application: Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors** 

Plain model constructions:

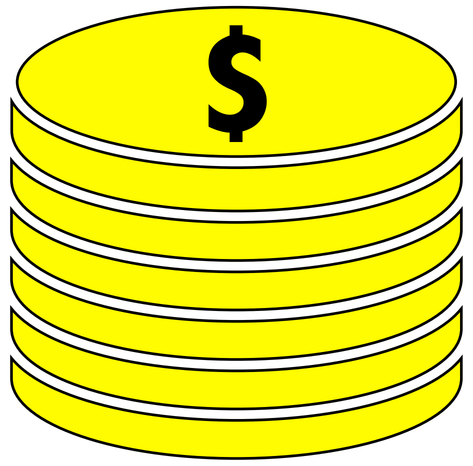
Iterated Sequential Function + SNARGs for P [BBBF18]	*Any* Sequential Function + SNARKs for NP [EFKP20]	Repeated Squaring + LWE [BCHKLPR22] (Previous talk)

Theorem:

Assuming **LWE** and *any* **sequential function**, there exists a VDF.

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**



Plain model constructions:

Iterated Sequential Function + SNARGs for P [BBBF18]	*Any* Sequential Function + SNARKs for NP [EFKP20]	Repeated Squaring + LWE [BCHKLPR22] (Previous talk)

Theorem:

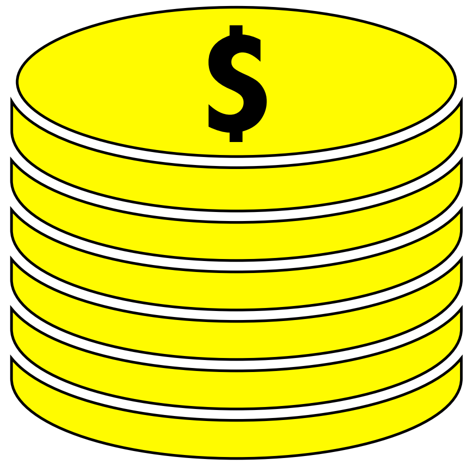
Assuming **LWE** and *any* **sequential function**, there exists a VDF.

Theorem:

Assuming **LWE** and *any* **memory-hard sequential function**, there exists a **memory-hard** VDF.

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**



Plain model constructions:

Iterated Sequential Function + SNARGs for P [BBBF18]	***Any* Sequential Function + SNARKs for [EFKP20]**	**g 22]**

First construction without "knowledge" assumptions

Theorem:

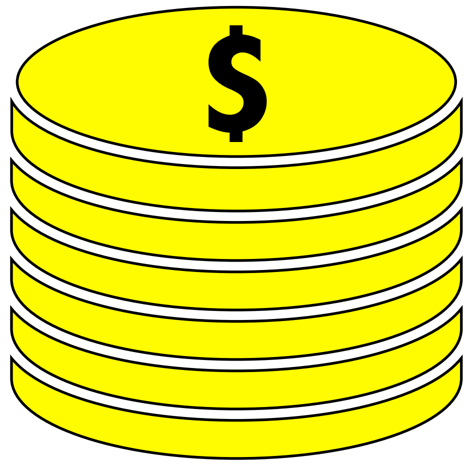
Assuming **LWE** and *any* **sequential function**, there exists a VDF.

Theorem:

Assuming **LWE** and *any* **memory-hard sequential function**, there exists a **memory-hard** VDF.

Application:
Verifiable Delay Functions [BBBF18]

- Verifiable function that **cannot be sped up** with **many processors**



Plain model constructions:

Iterated Sequential Function + SNARGs for P [BBBF18]	*Any* Sequential Function + SNARKs for NP [EFKP20]	Repeated Squaring + LWE [BCHKLPR22] (Previous talk)

Theorem:

Assuming **LWE** and *any* **sequential function**, there exists a VDF.

Theorem:

Assuming **LWE** and *any* **memory-hard sequential function**, there exists a **memory-hard** VDF.

Additional Result: Time-Independent SPARGs

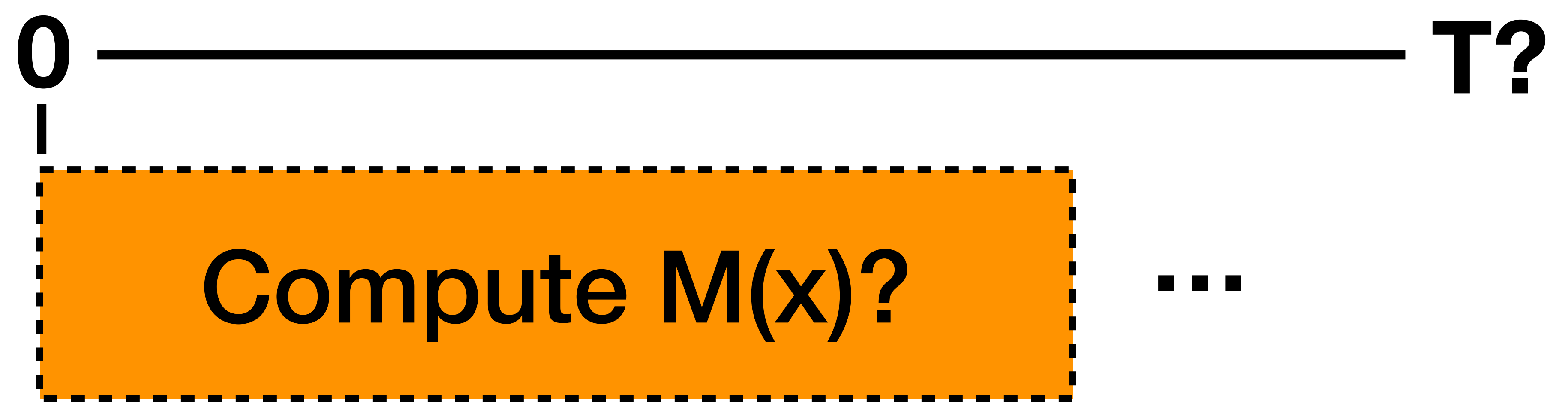
Additional Result:
Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

Additional Result: Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

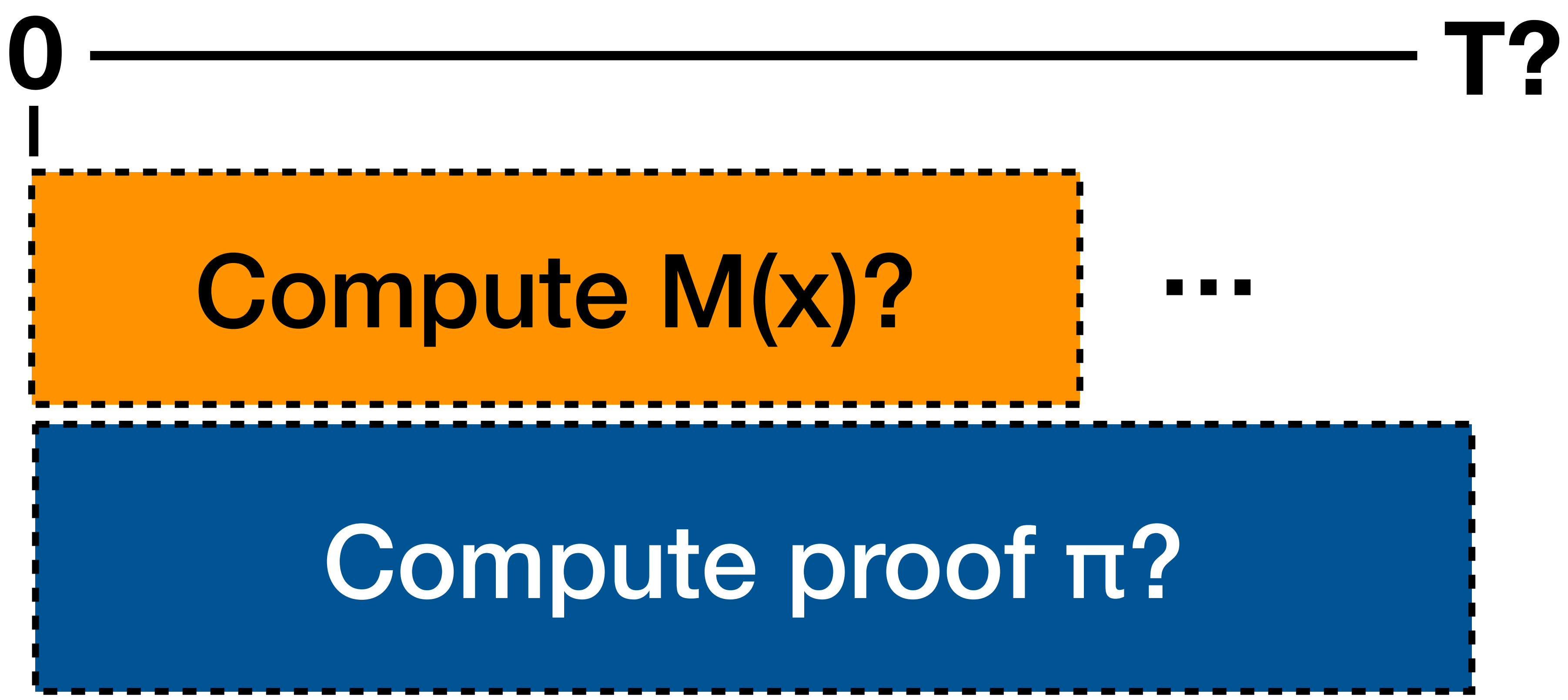
For SPARGs:



Additional Result:
Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

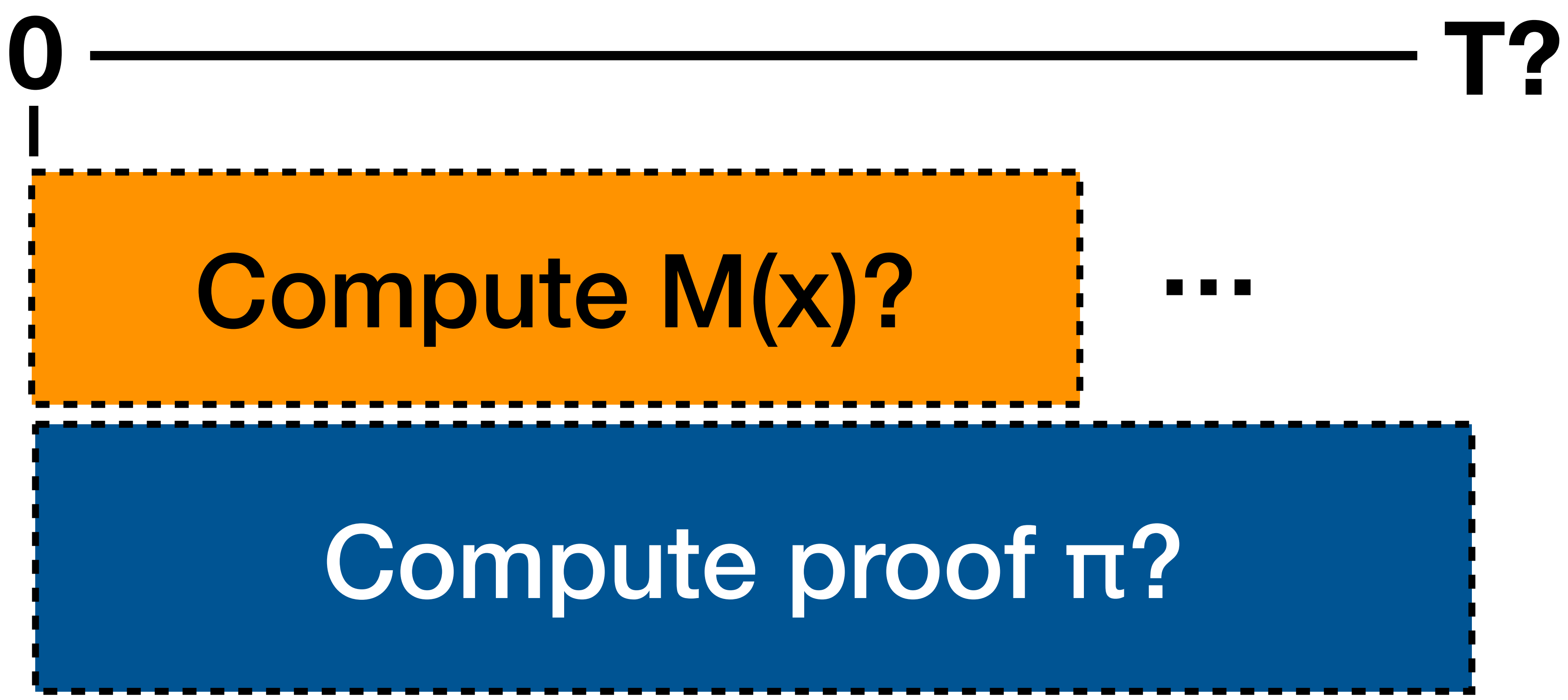
For SPARGs:



Additional Result: Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

For SPARGs:

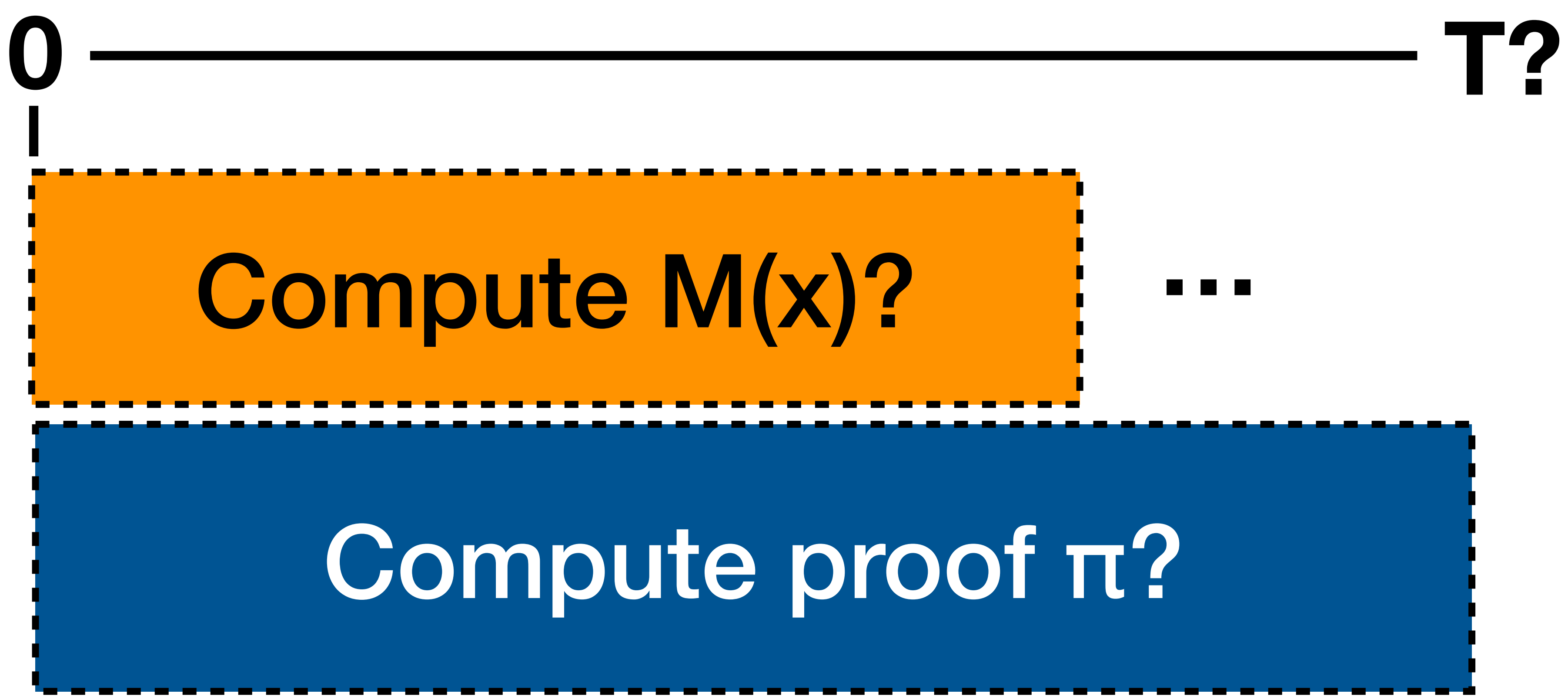


- [E**F**KP20] relied on knowing T in advance

Additional Result: Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

For SPARGs:



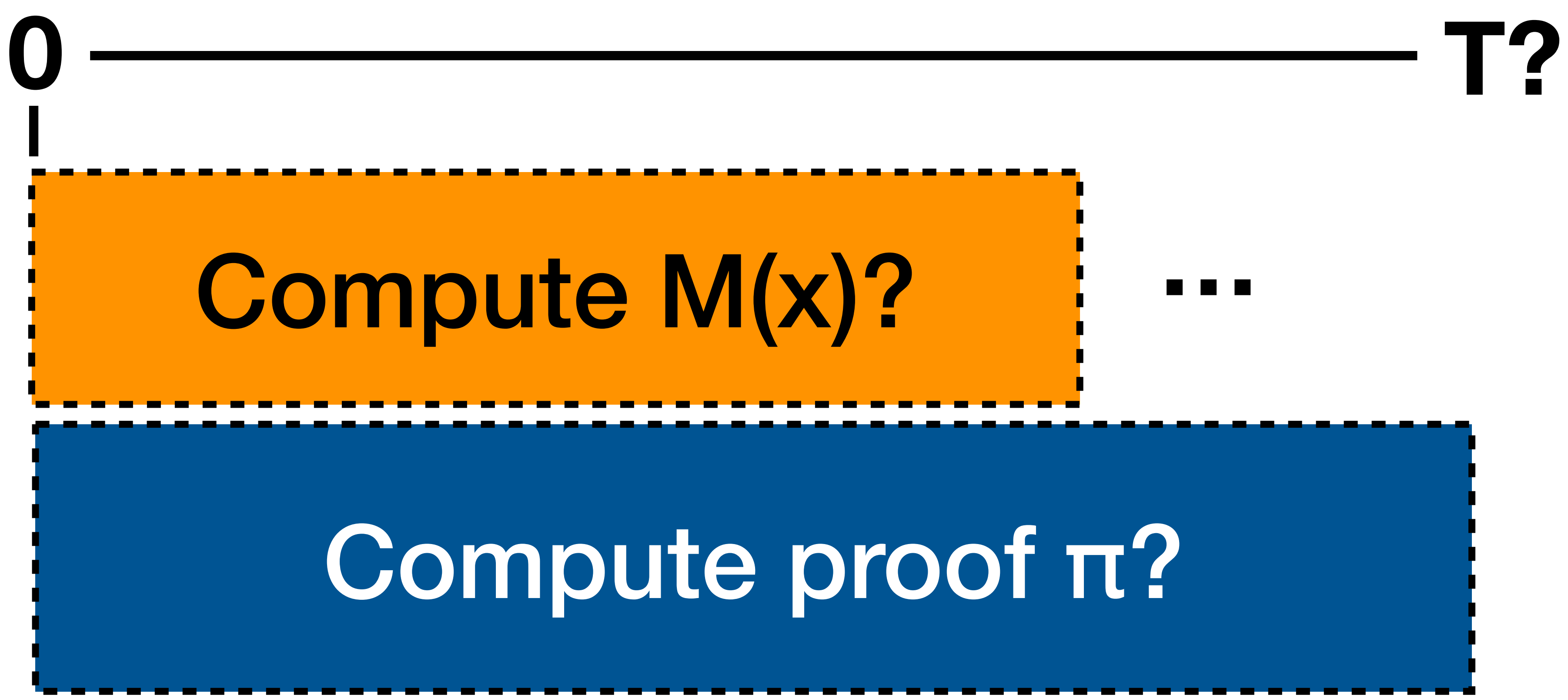
- [E**F**KP20] relied on knowing T in advance

Is this necessary?

Additional Result:
Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

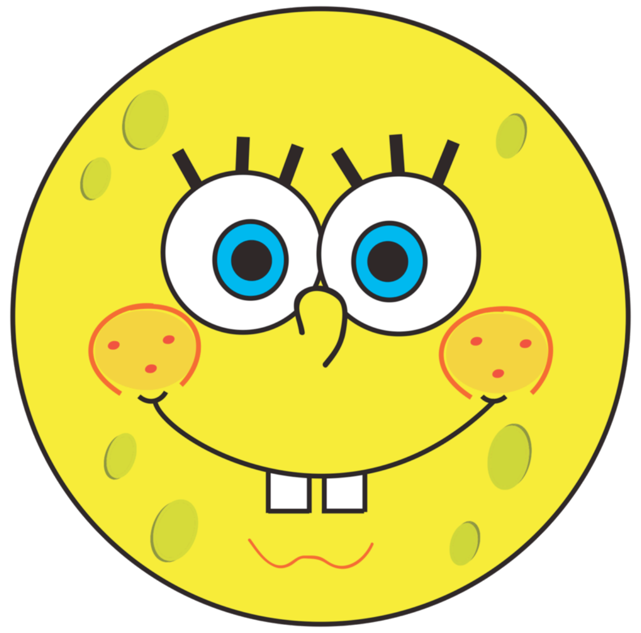
For SPARGs:



- [E**F**KP20] relied on knowing T in advance

Is this necessary?

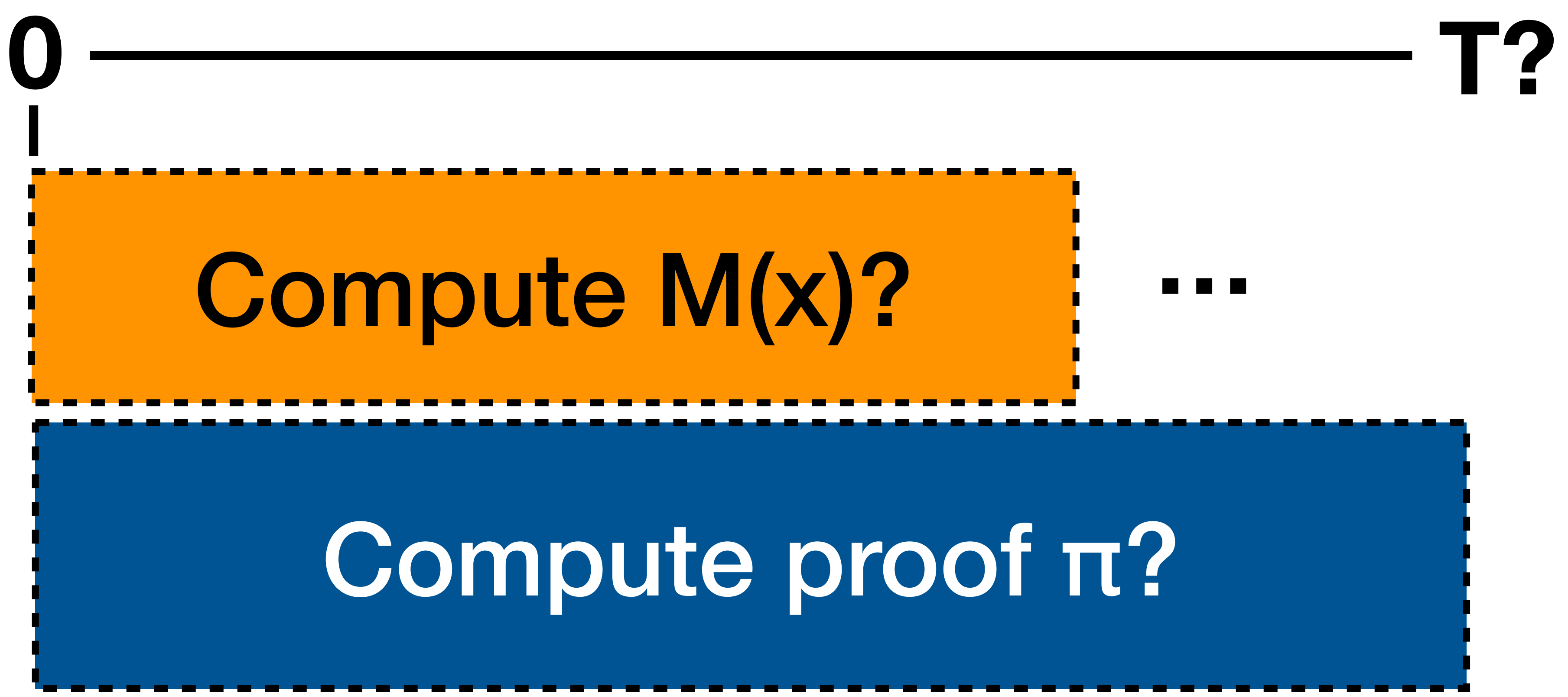
We show:
No!



Additional Result:
Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

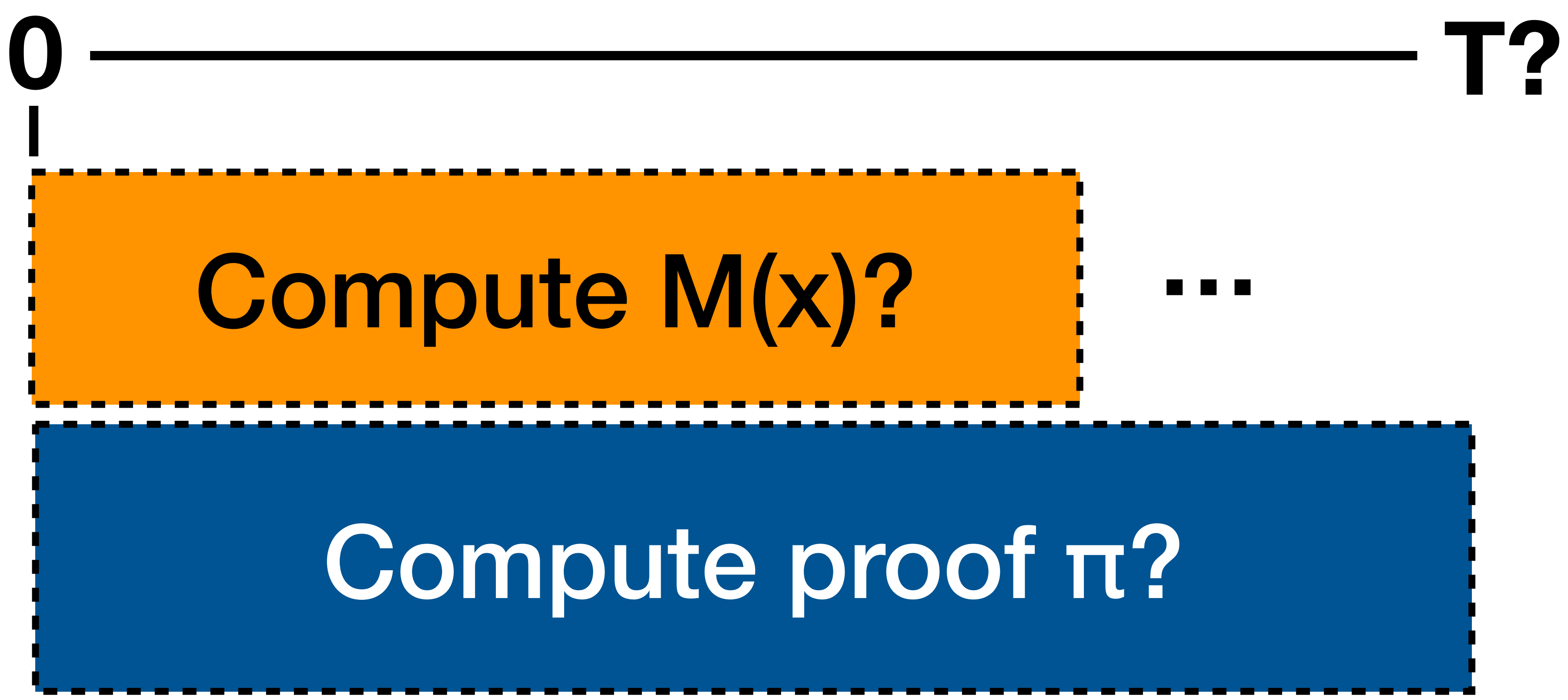
For SPARGs:



Additional Result:
Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

For SPARGs:



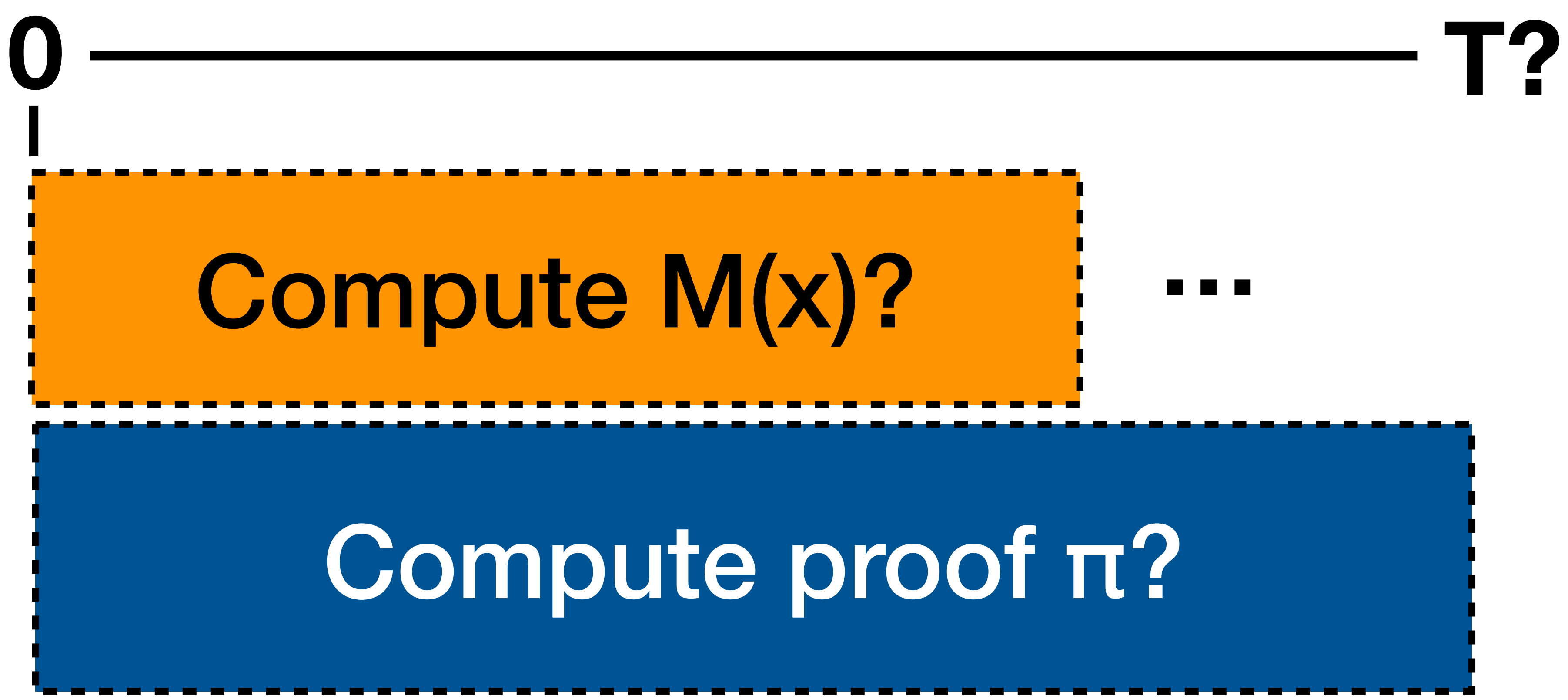
Theorem (informal):

Given *any* SPARG, can construct a **time-independent SPARG**.

Additional Result: Time-Independent SPARGs

Observation:
For standard arguments, can know the time bound **T** when you compute the proof.

For SPARGs:



Theorem (informal):

Given *any* SPARG, can construct a **time-independent SPARG**.

Key idea:
Tree-based construction to "non-deterministically guess" binary representation of **T**.

Big Picture for
Our Main Result

Big Picture for Our Main Result

[EFKP20]

Big Picture for Our Main Result

[EFKP20]

Any SNARK for NP => [BCCT13] Quasi-linear SNARK for NP

Big Picture for Our Main Result

[EFKP20]

Any SNARK for NP **=>** [BCCT13] **Quasi-linear SNARK for NP** $\xrightarrow{\text{generic transformation}}$ **SPARKs for NP**

Big Picture for Our Main Result

[EFKP20]

Any SNARK for NP **=>** [BCCT13] **Quasi-linear SNARK for NP** $\xrightarrow{\text{generic transformation}}$ **SPARKs for NP**

Our Work

Big Picture for Our Main Result

[EFKP20]

Any SNARK for NP $\underset{\text{[BCCT13]}}{=>}$ Quasi-linear SNARK for NP $\xrightarrow{\text{generic transformation}}$ SPARKs for NP

Our Work

Specific SNARG for P from LWE [CJJ21] => **Quasi-linear updatable RAM delegation**

Big Picture for Our Main Result

[EFKP20]

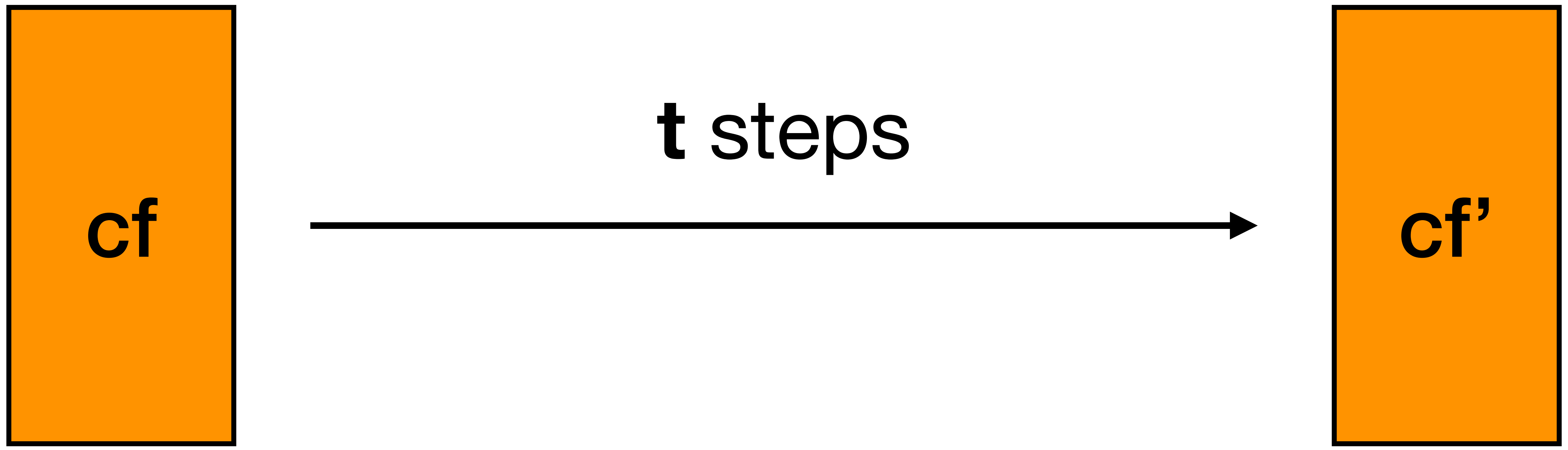
Any SNARK for NP => [BCCT13] Quasi-linear SNARK for NP $\xrightarrow{\text{generic transformation}}$ **SPARKs for NP**

Our Work

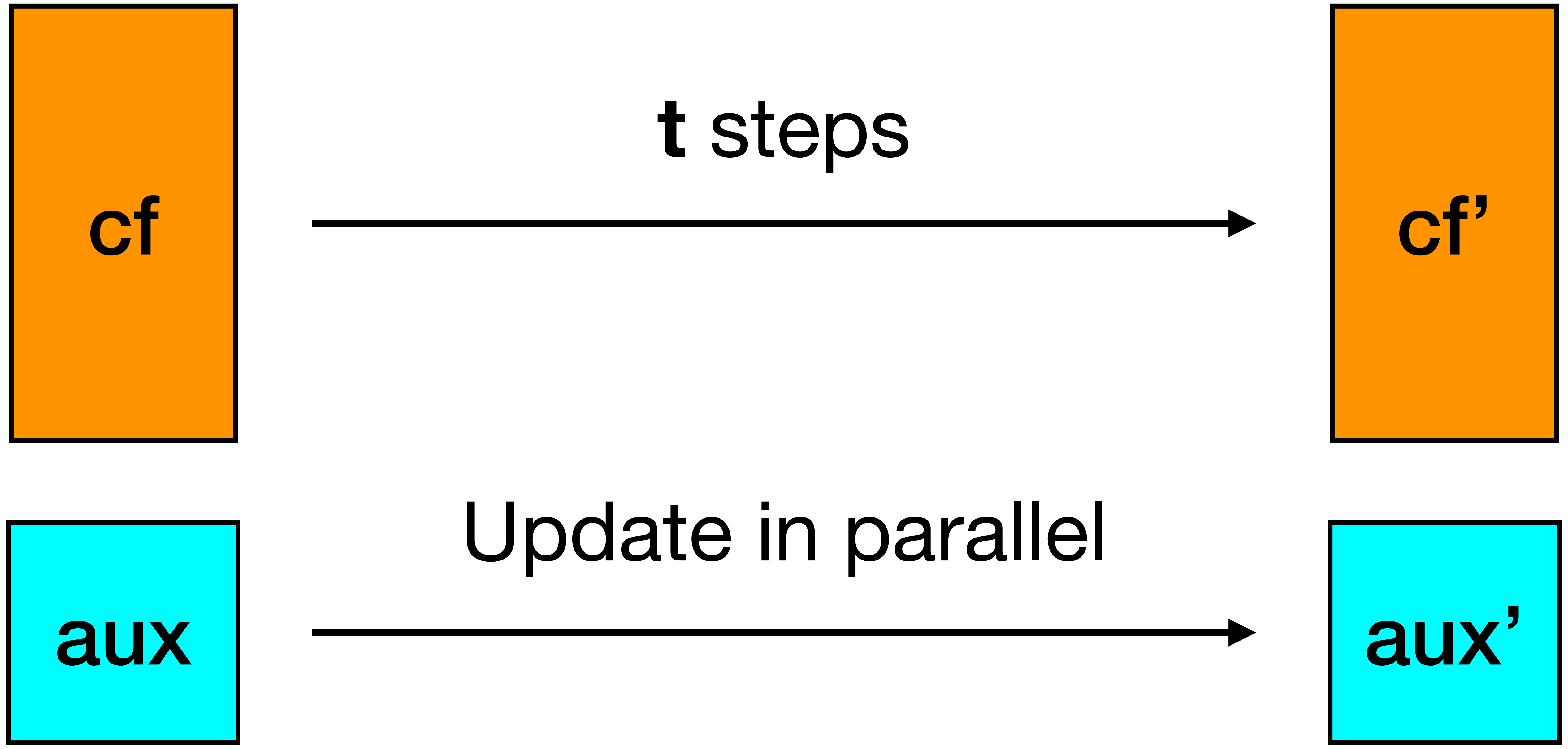
Specific SNARG for P from LWE [CJJ21] => **Quasi-linear updatable RAM delegation** $\xrightarrow{\text{generic transformation}}$ **SPARGs for P**

Quasi-linear Updatable RAM Delegation

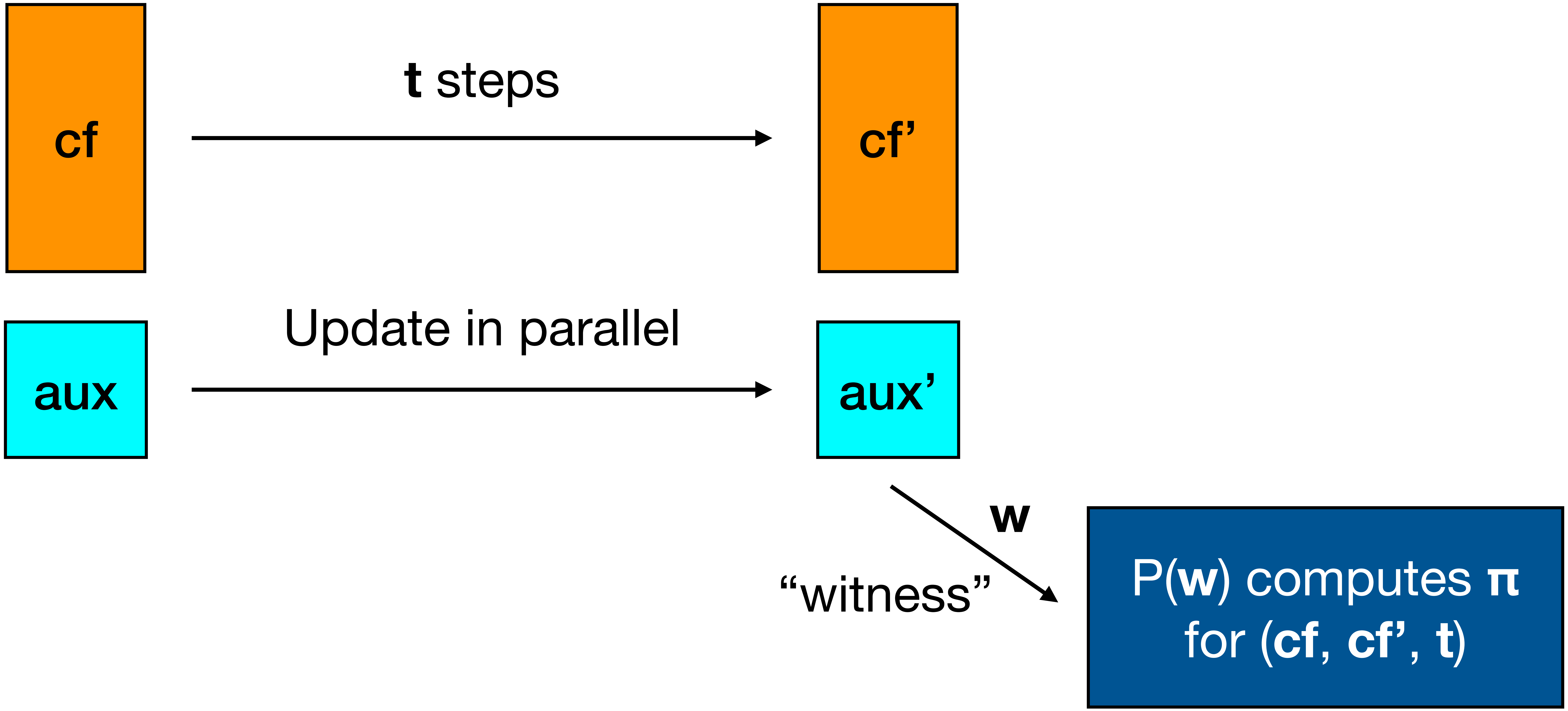
Quasi-linear Updatable RAM Delegation



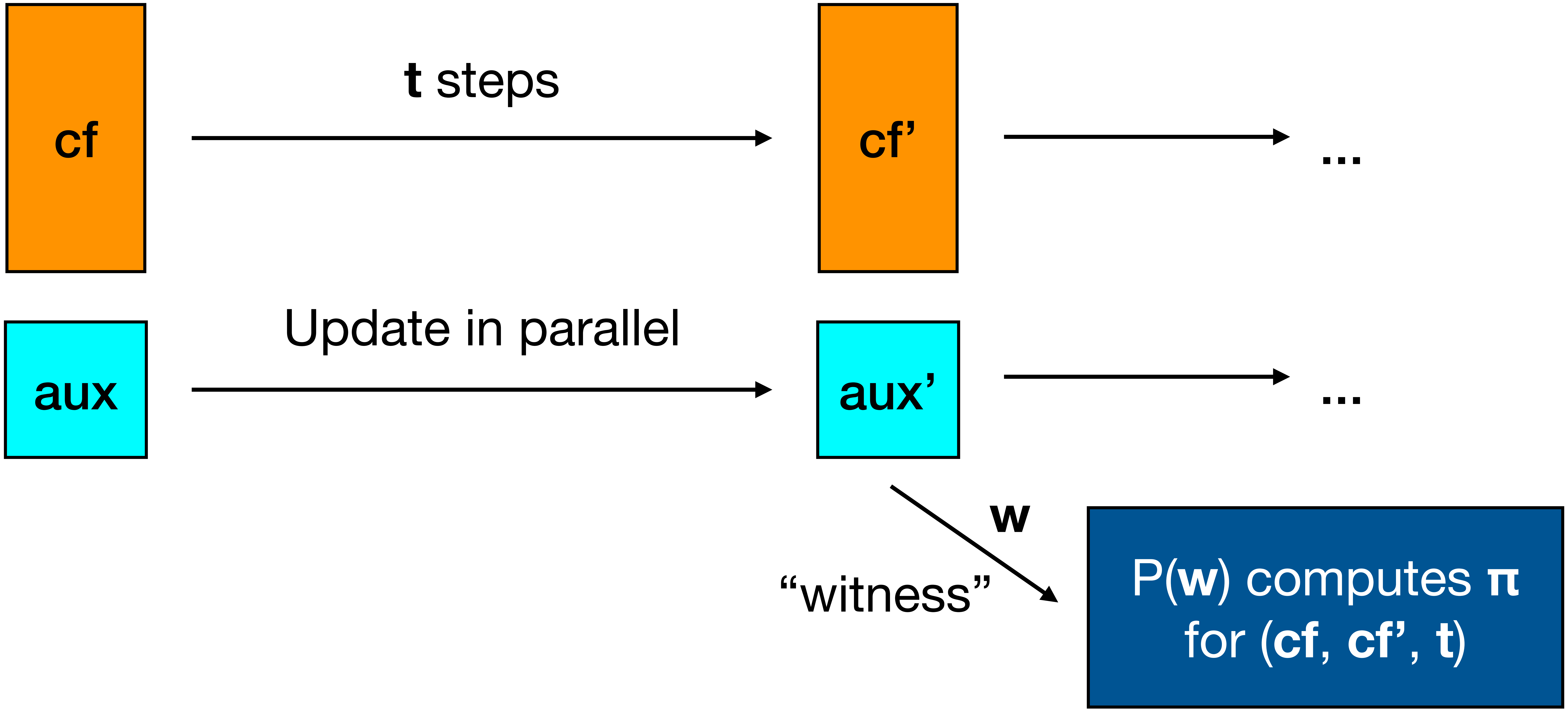
Quasi-linear Updatable RAM Delegation



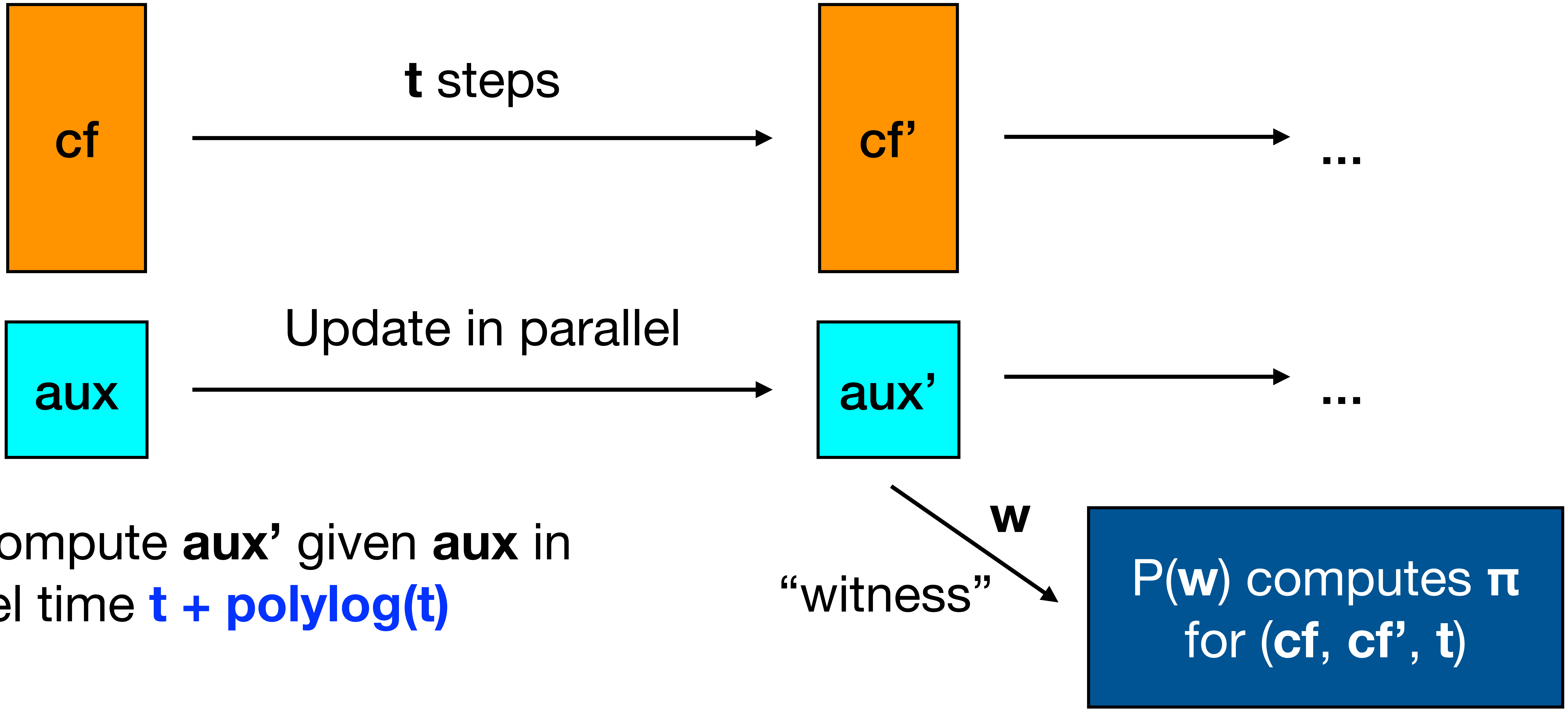
Quasi-linear Updatable RAM Delegation



Quasi-linear Updatable RAM Delegation

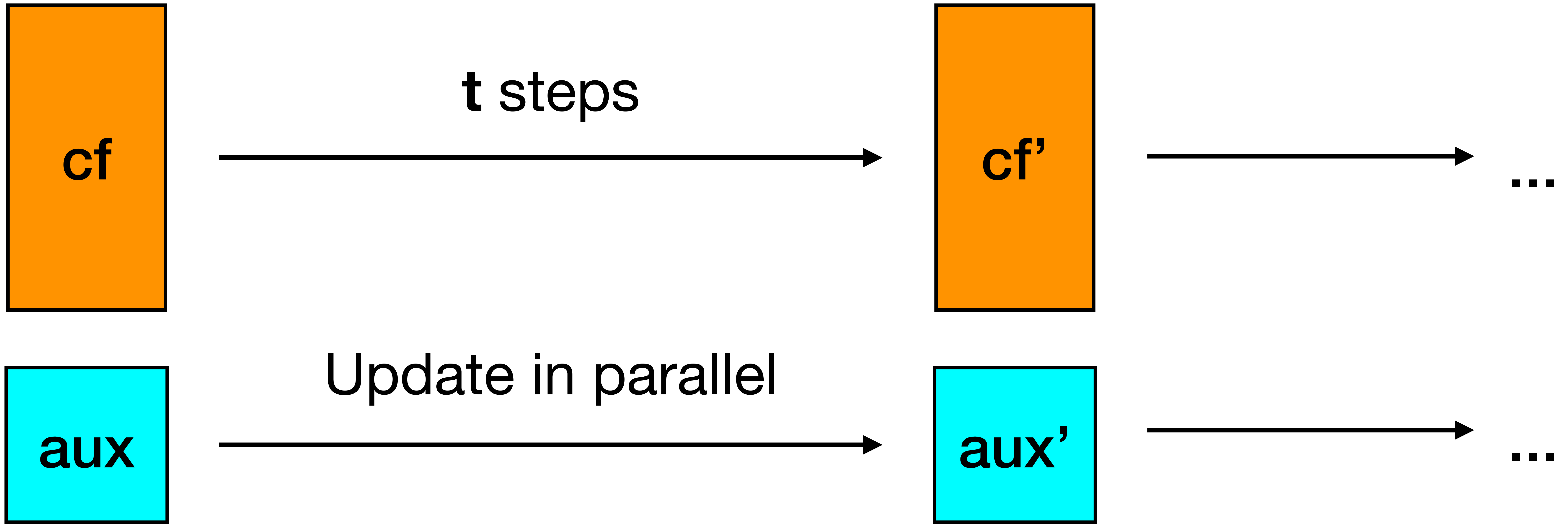


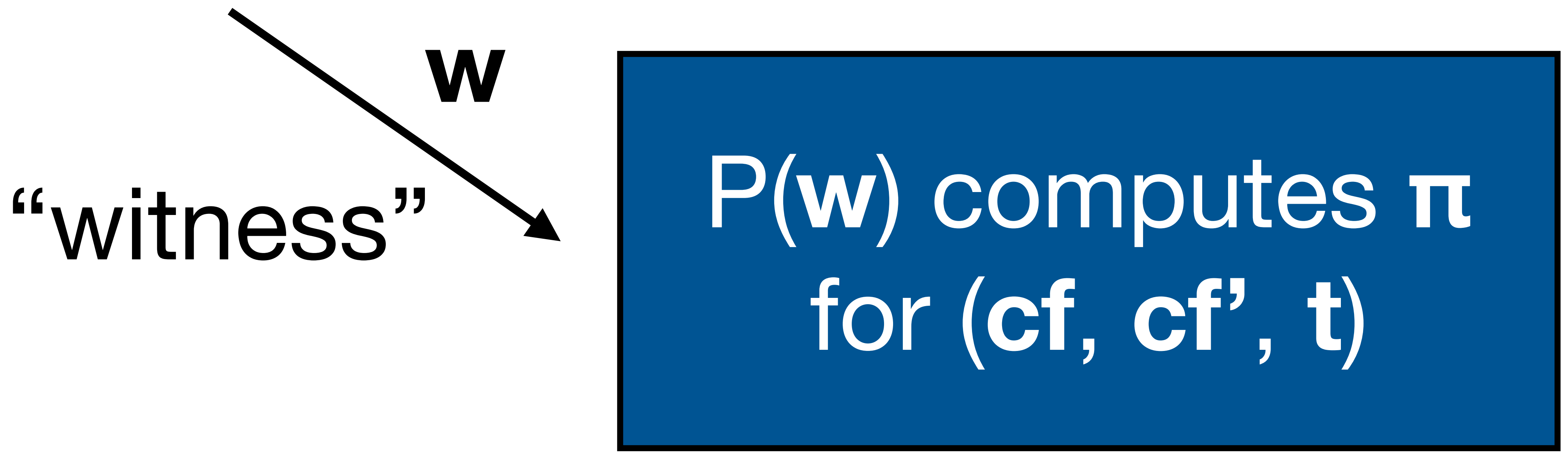
Quasi-linear Updatable RAM Delegation



- Can compute **aux’** given **aux** in parallel time **t + polylog(t)**

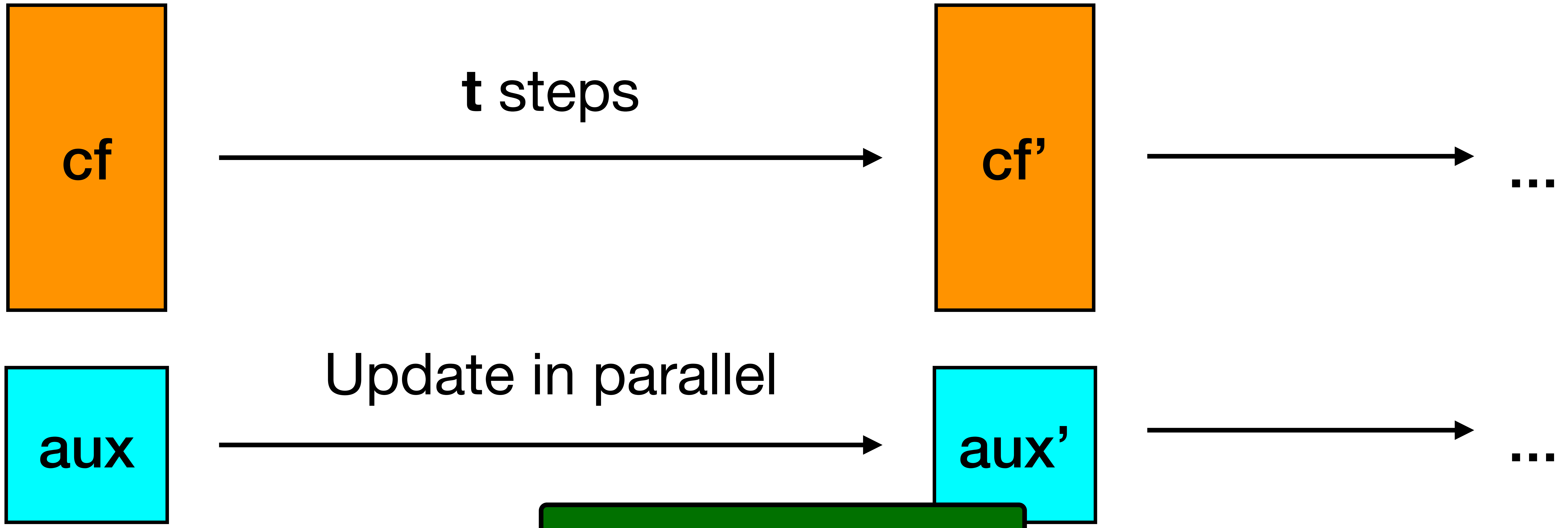
Quasi-linear Updatable RAM Delegation





- Can compute **aux’** given **aux** in parallel time **t + polylog(t)**
- Given **w**, can compute a proof **π** for **cf->cf’** in time **t *polylog(t)**

Quasi-linear Updatable RAM Delegation



- Can compute **aux’** given **au** parallel time **t + polylog(t)**
- Given **w**, can compute a proof **π** for **cf->cf’** in time **t *polylog(t)**

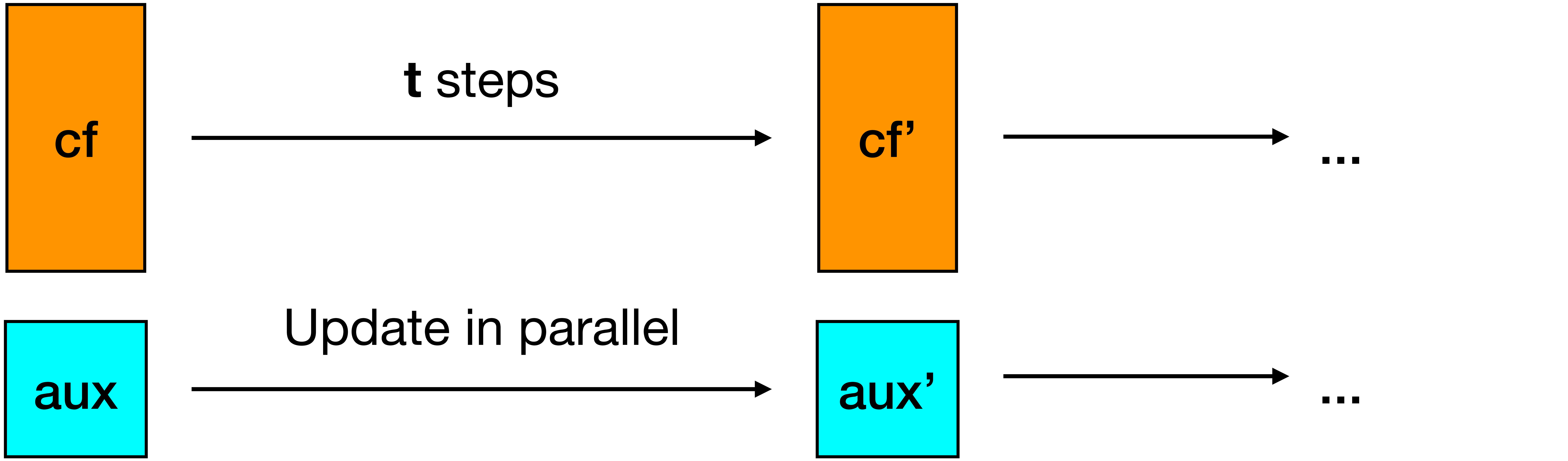
Without w, depends on |cf|.

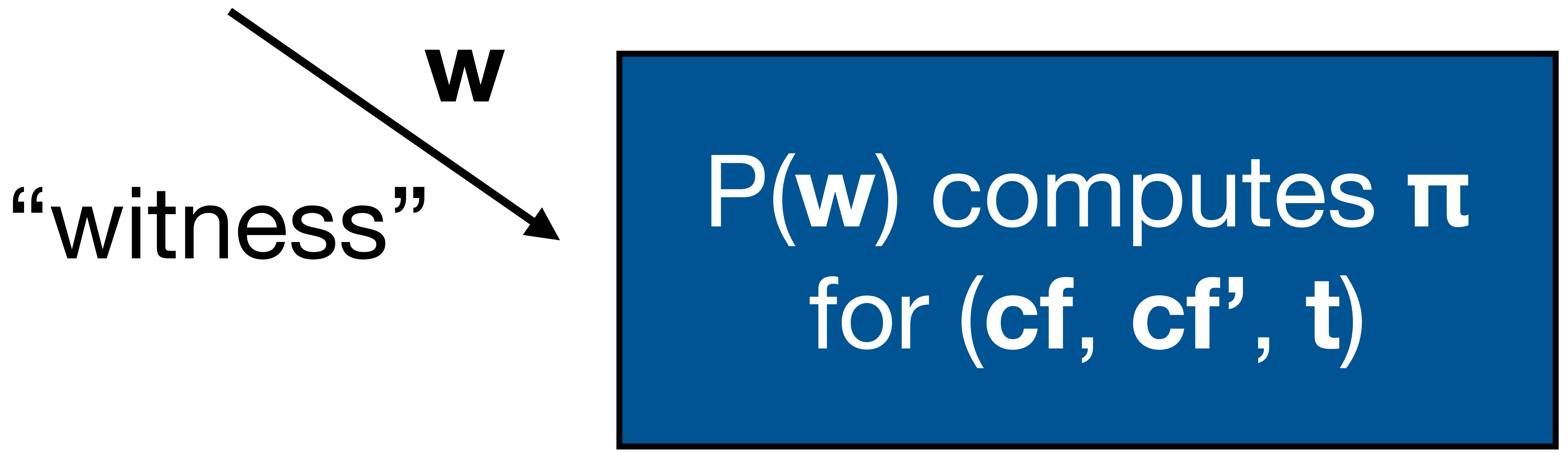
w

ess”

P(**w**) computes **π** for (**cf**, **cf’**, **t**)

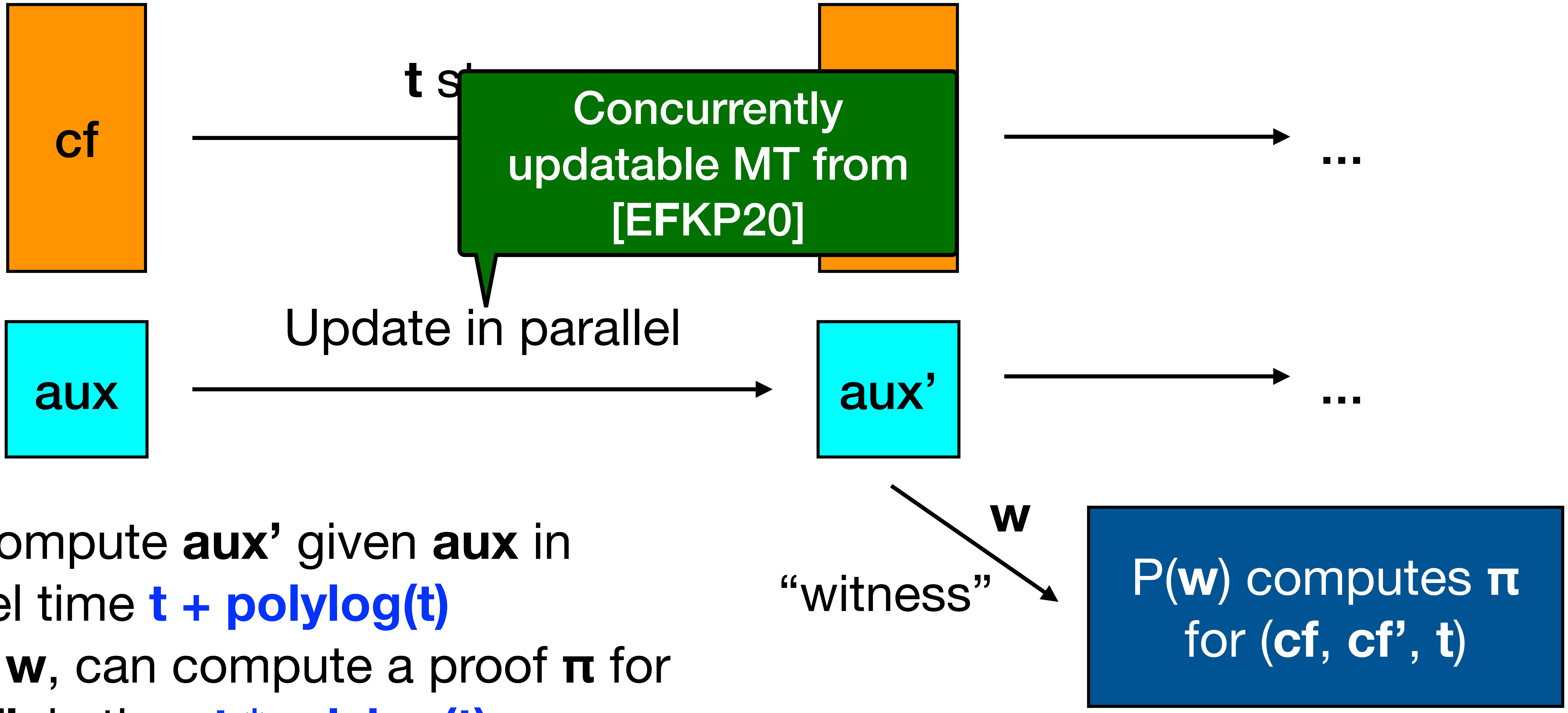
Instantiating from LWE





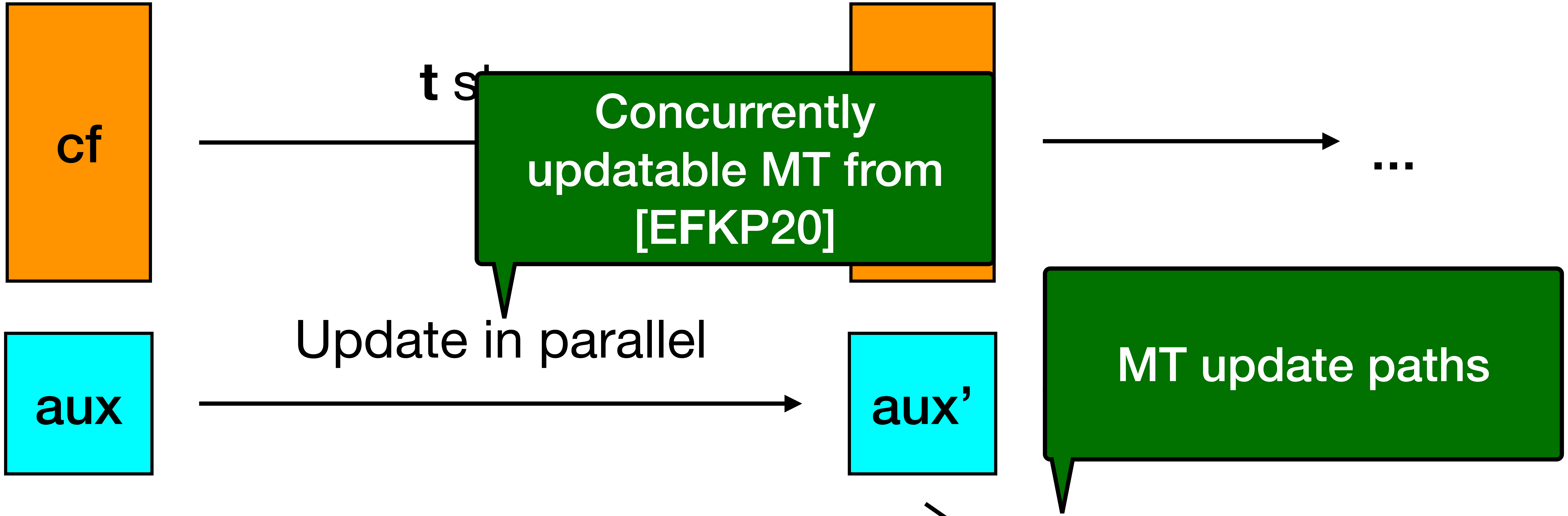
- Can compute **aux'** given **aux** in parallel time **t + polylog(t)**
- Given **w**, can compute a proof **π** for **cf->cf'** in time **t *polylog(t)**

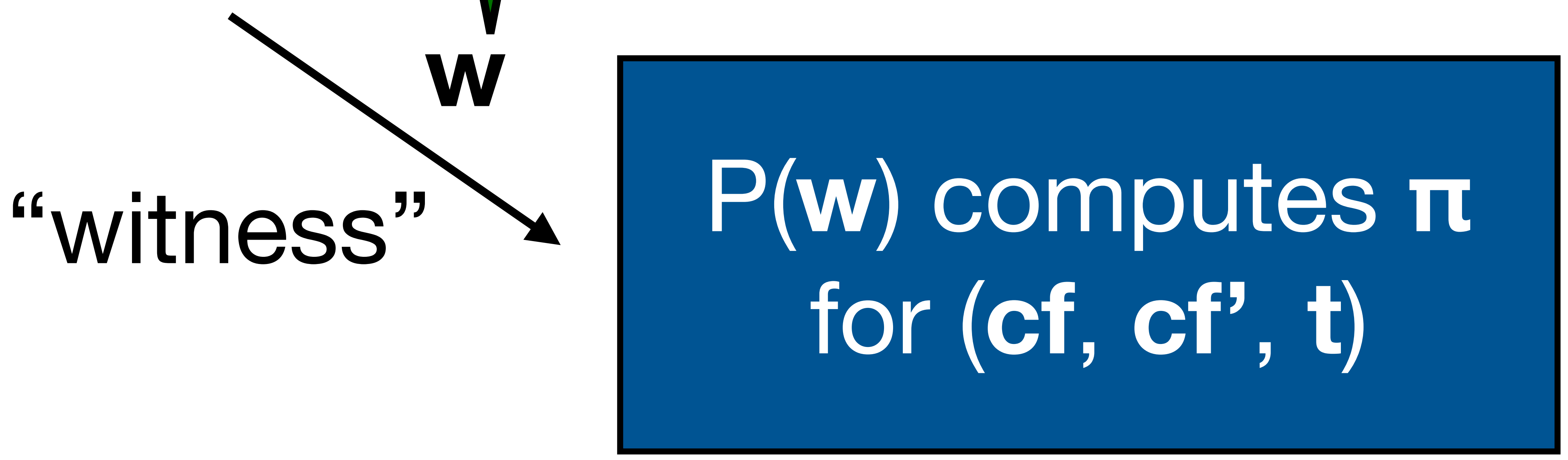
Instantiating from LWE



- Can compute **aux'** given **aux** in parallel time **t + polylog(t)**
- Given **w**, can compute a proof **π** for **cf->cf'** in time **t *polylog(t)**

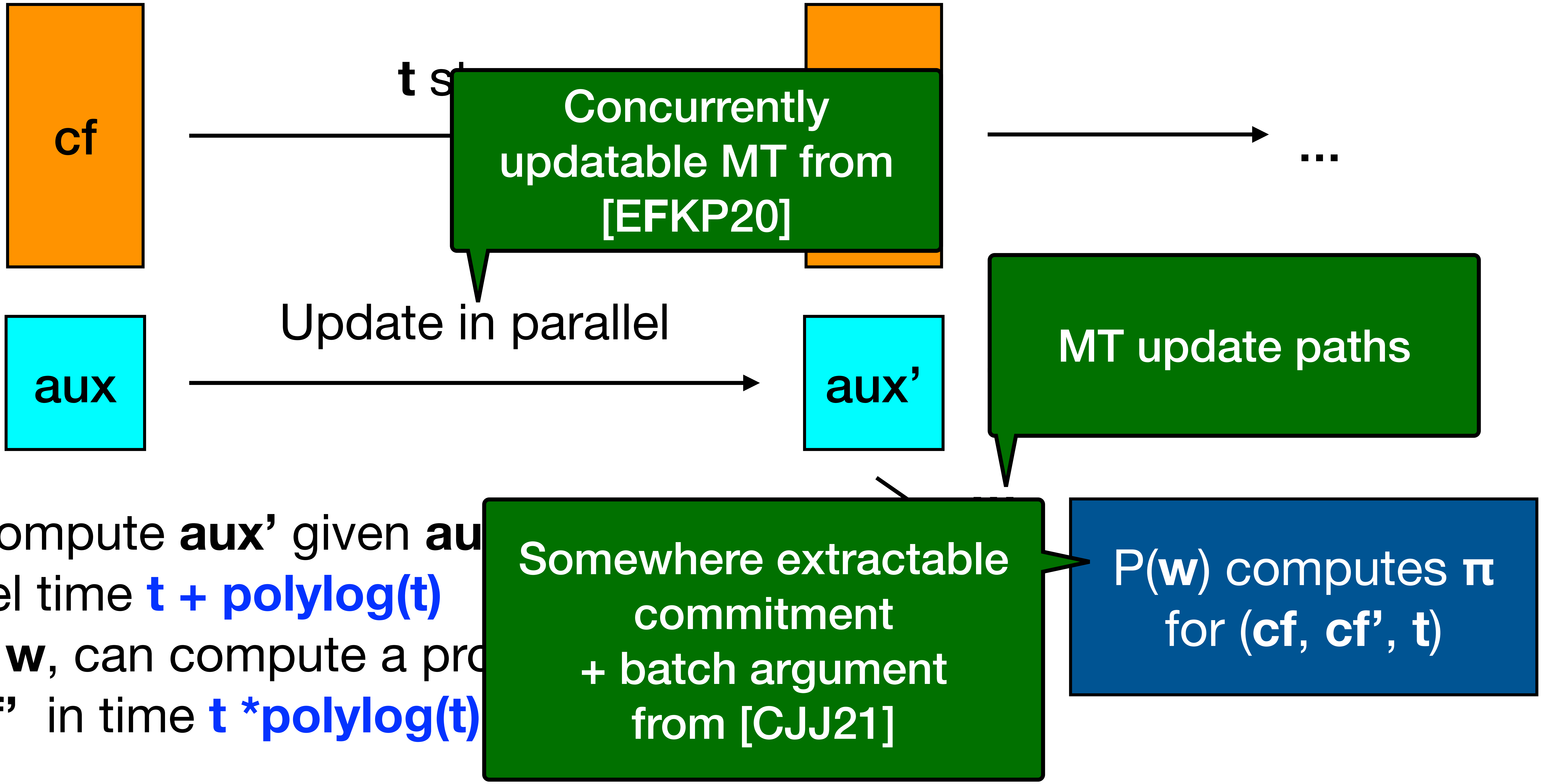
Instantiating from LWE





- Can compute **aux’** given **aux** in parallel time **t + polylog(t)**
- Given **w**, can compute a proof **π** for **cf->cf’** in time **t *polylog(t)**

Instantiating from LWE



- Can compute **aux'** given **au**
  parallel time **t + polylog(t)**
- Given **w**, can compute a pr
  cf->cf' in time **t *polylog(t)**

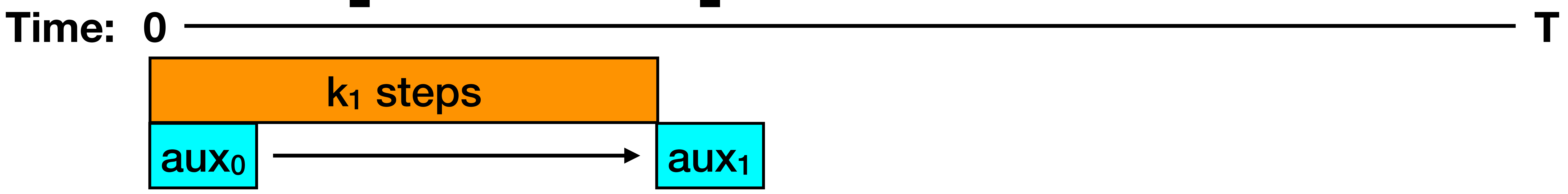
Putting it all together with [EFKP20] transformation

Putting it all together with [EFKP20] transformation

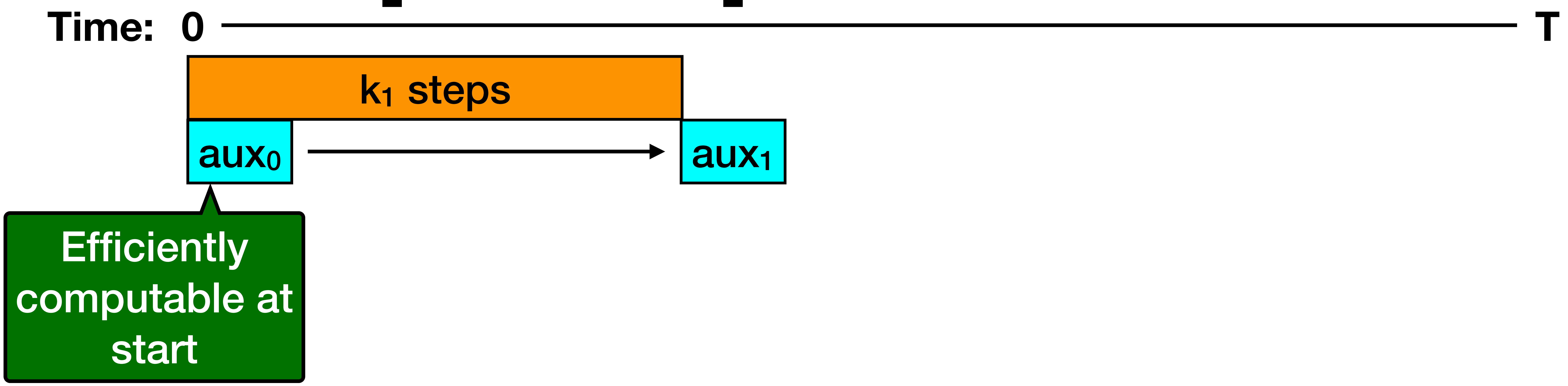
Time: **0** —— **T**

Putting it all together with [EFKP20] transformation

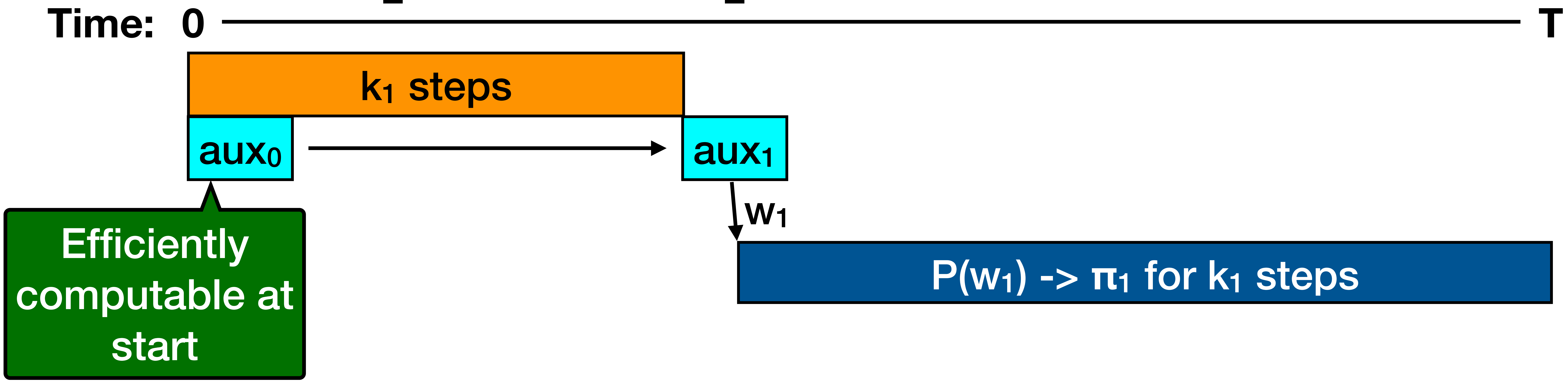
Putting it all together with [EFKP20] transformation



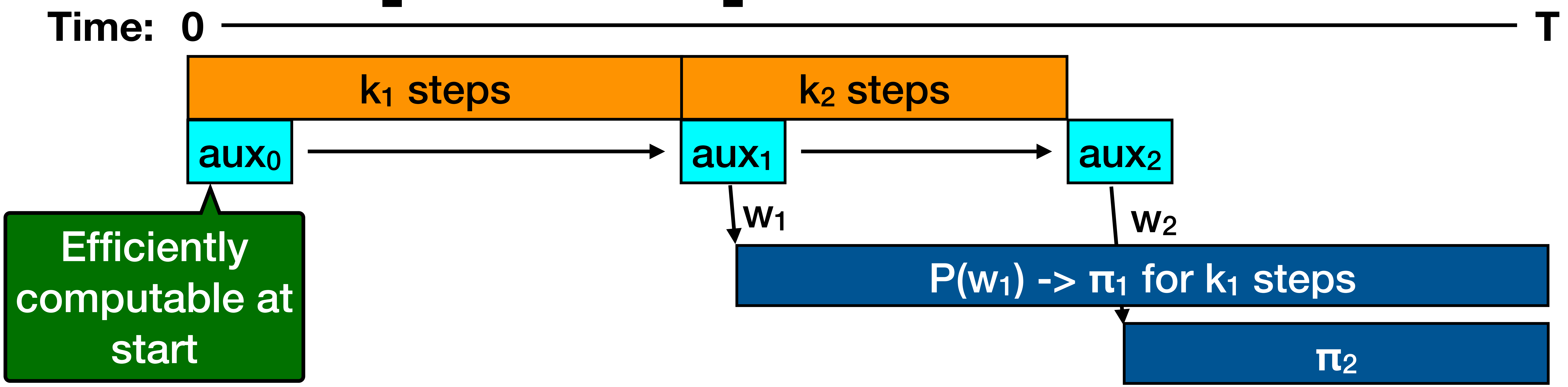
Putting it all together with [EFKP20] transformation



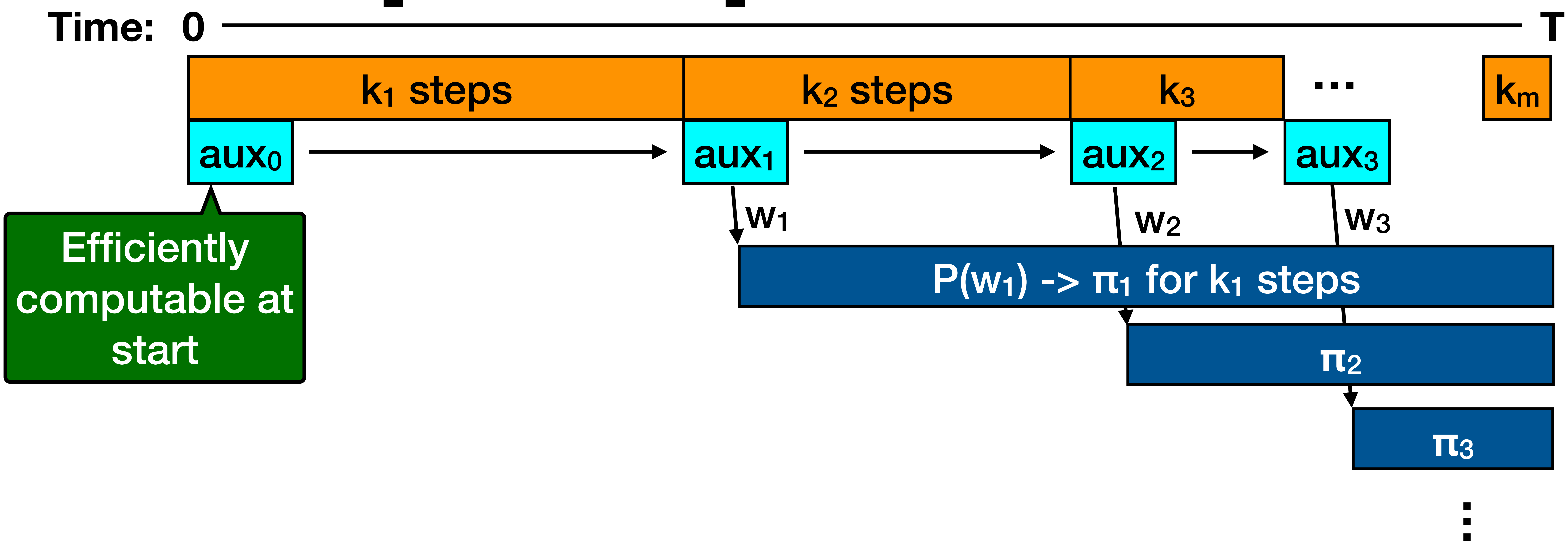
Putting it all together with [EFKP20] transformation



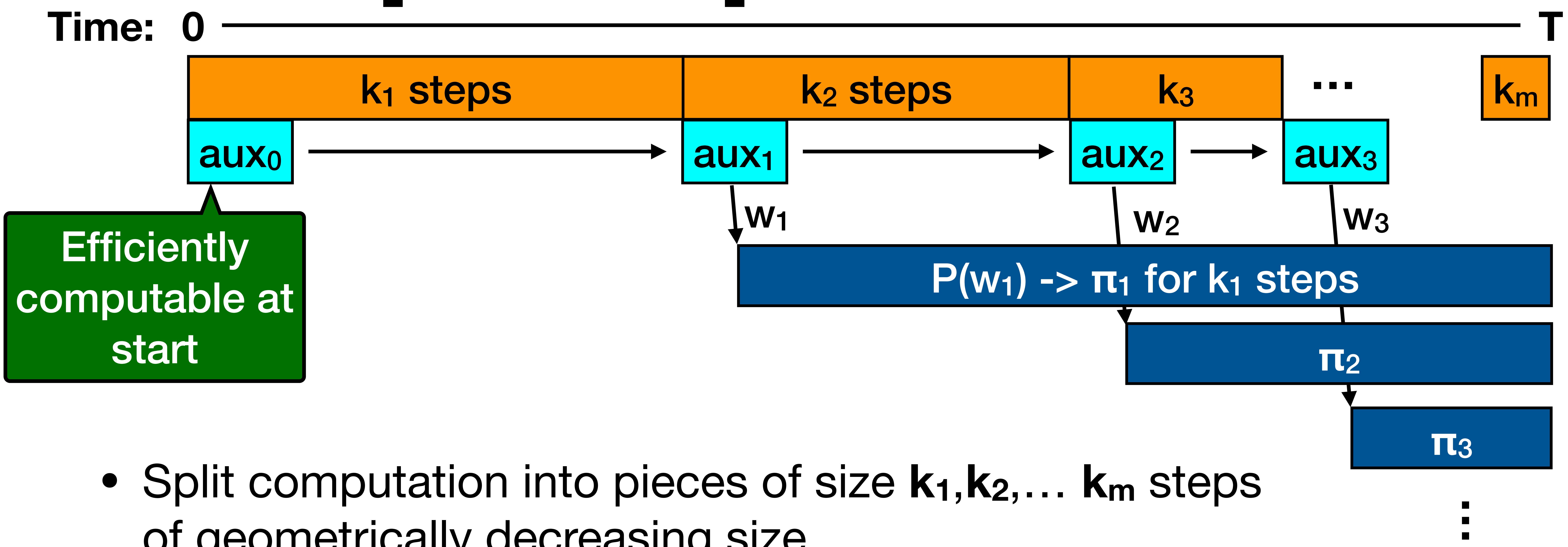
Putting it all together with [EFKP20] transformation



Putting it all together with [EFKP20] transformation

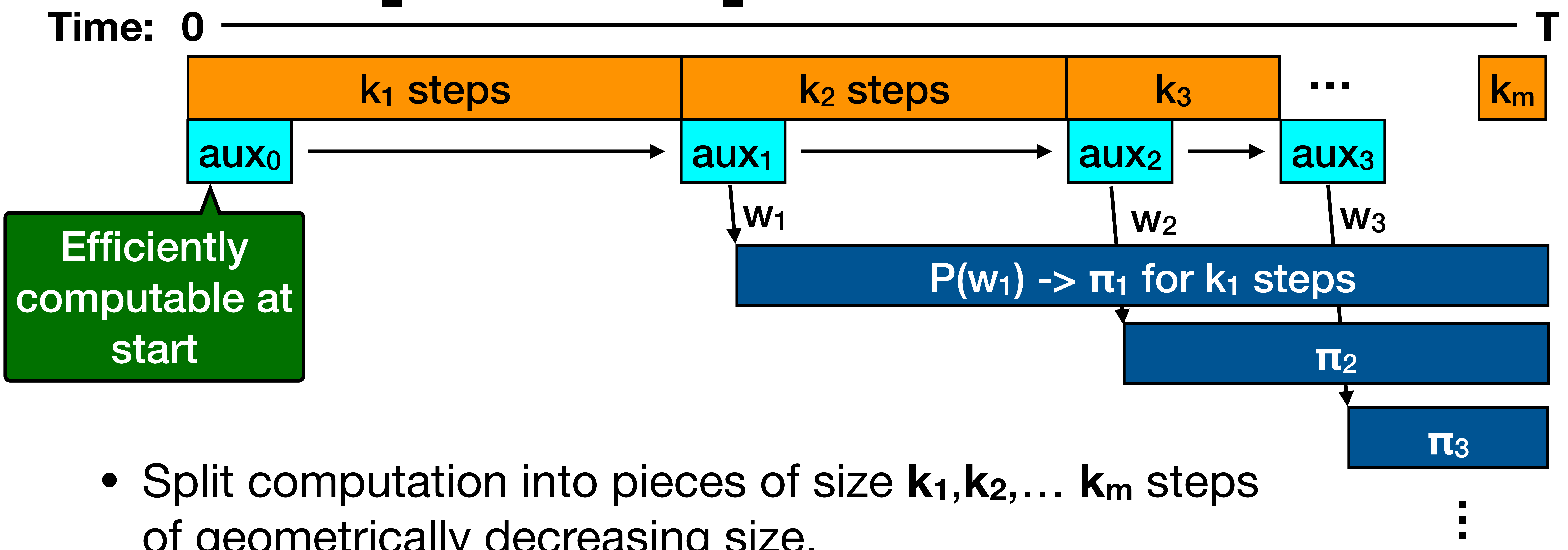


Putting it all together with [EFKP20] transformation



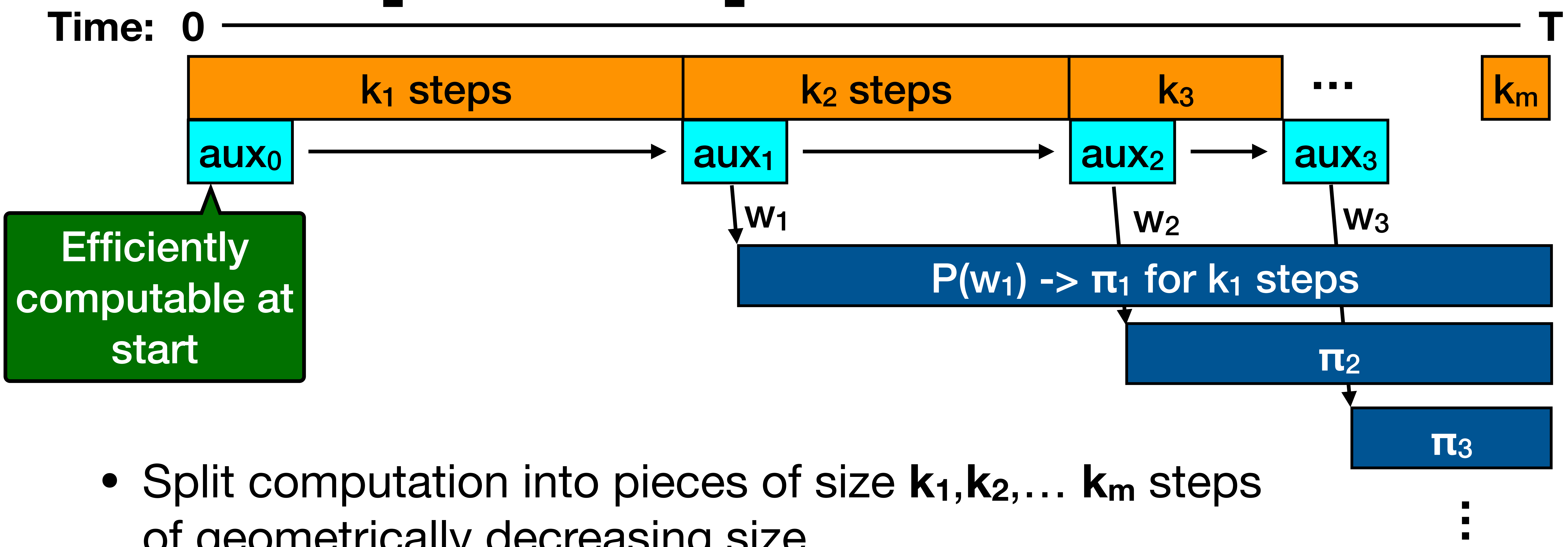
- Split computation into pieces of size $\mathbf{k_1}$,$\mathbf{k_2}$,… $\mathbf{k_m}$ steps of geometrically decreasing size.

Putting it all together with [EFKP20] transformation



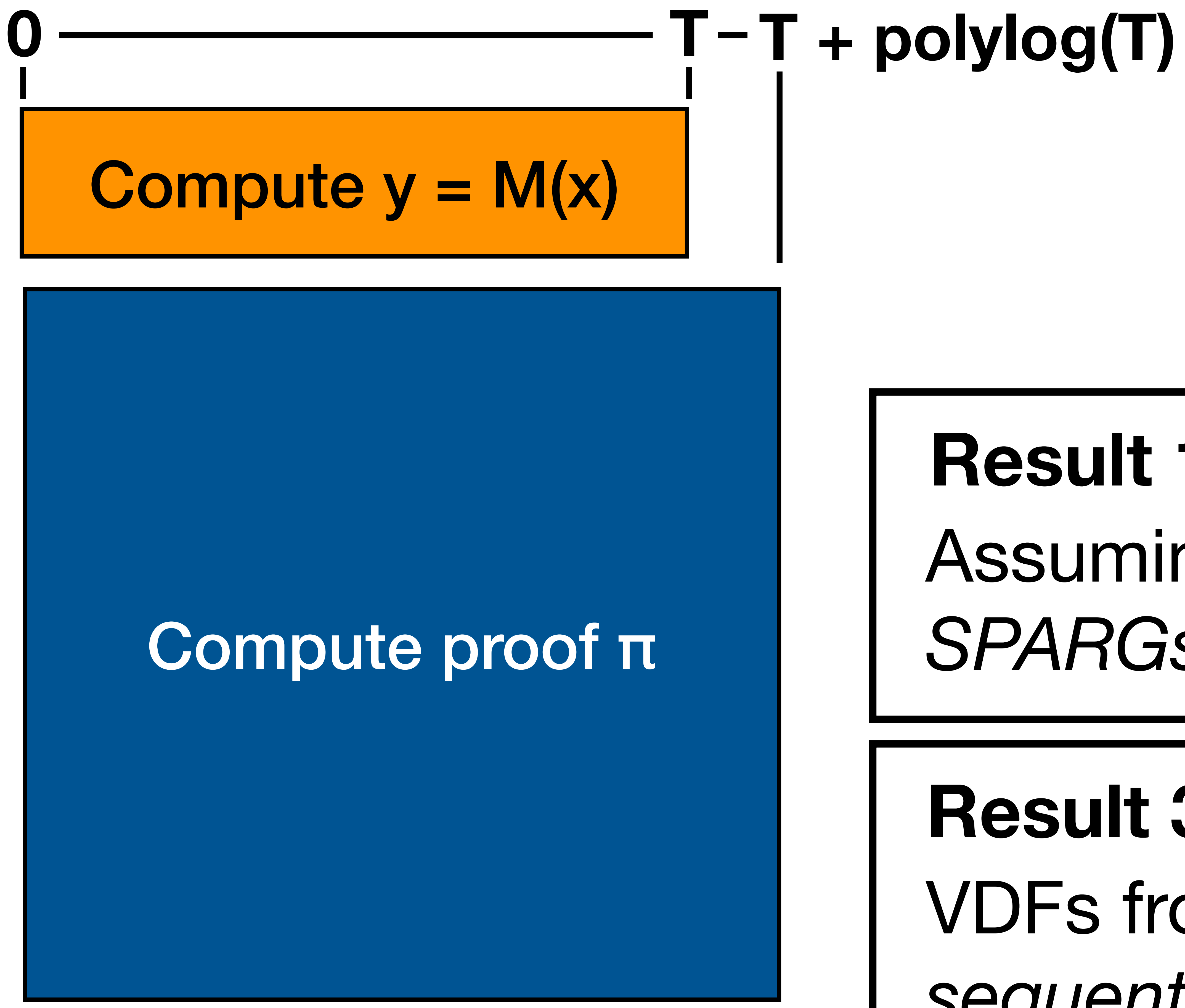
- Split computation into pieces of size $\mathbf{k_1},\mathbf{k_2},\ldots\ \mathbf{k_m}$ steps of geometrically decreasing size.
- Set $\mathbf{k_i}$ so that "i"th proof finished by time **T**.

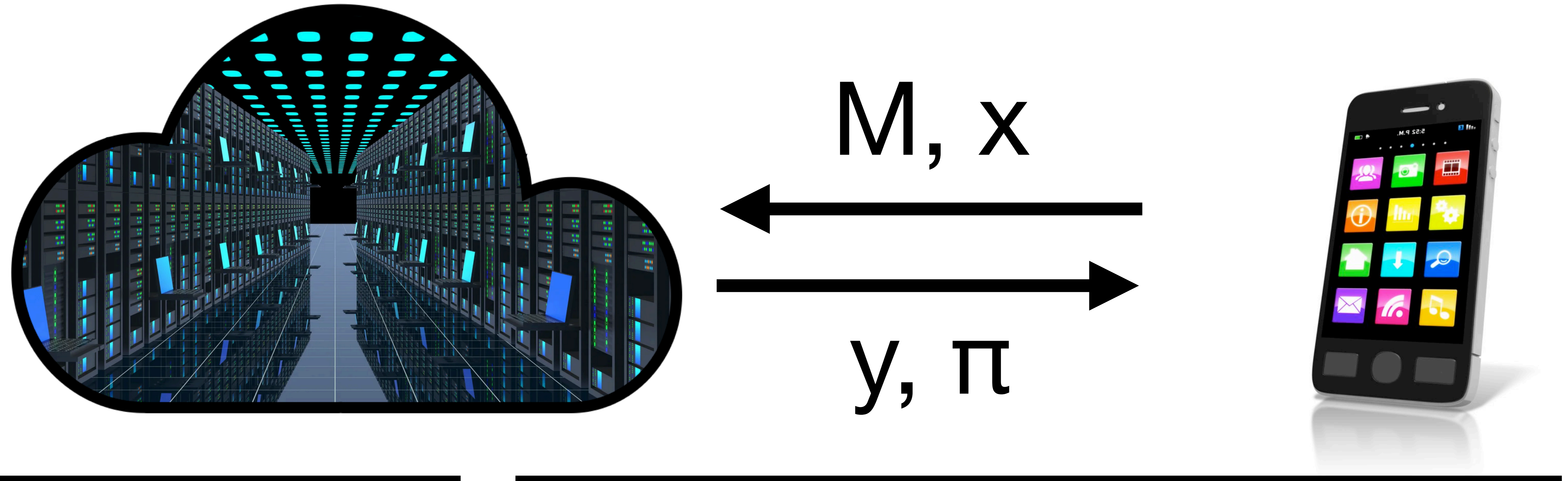
Putting it all together with [EFKP20] transformation



- Split computation into pieces of size $\mathbf{k_1}$,$\mathbf{k_2}$,… $\mathbf{k_m}$ steps of geometrically decreasing size.
- Set $\mathbf{k_i}$ so that "i"th proof finished by time **T**.
- Final proof consists of **m = polylog(T)** sub-proofs.

Questions?





Result 1:
Assuming *LWE*, exist *SPARGs for PRAM*.

Result 2:
SPARGs =>
time-independent SPARGs

Result 3:
VDFs from *LWE* + *any sequential function*

Result 4:
Mem-hard VDFs without "knowledge" assumptions