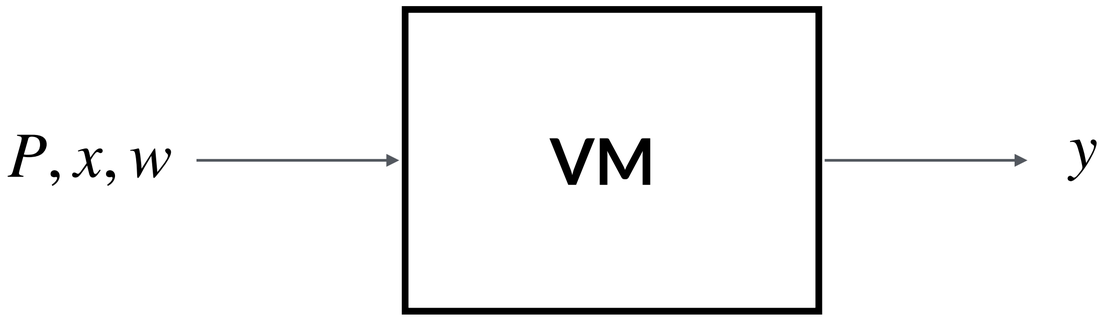
# Memory checking in IVC-based zkVMs

**ZKProof, Berlin 2024**

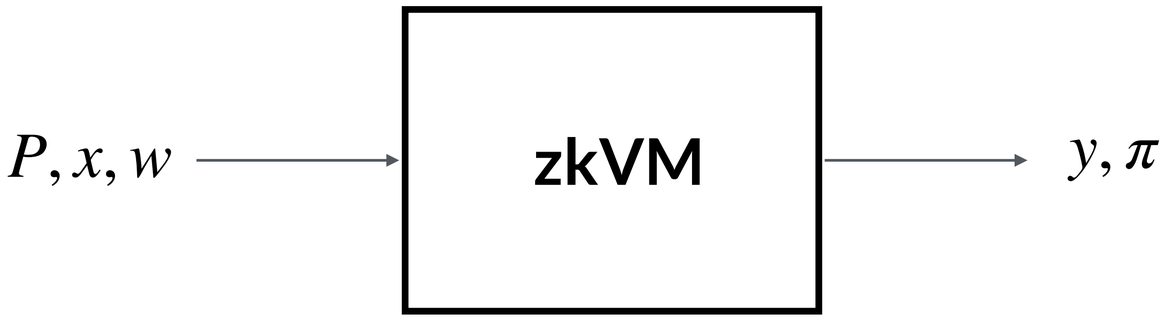
**Jens Groth, Nexus - nexus.xyz**
**based on discussions with Yinuo Zhang**

## Virtual Machine

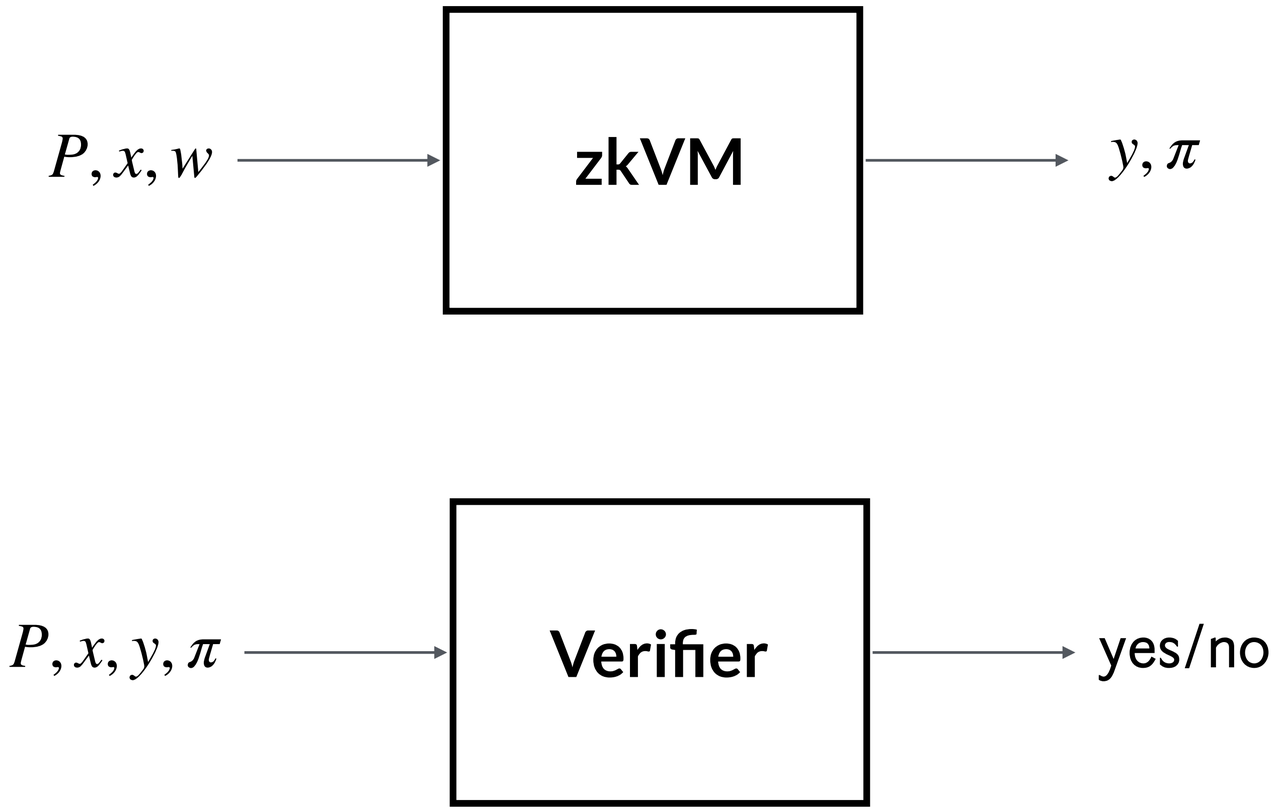



- Program $P$ specified according to an Instruction Set Architecture
- Input $x$ to the program (public)
- Auxiliary input $w$ to the program (maybe not public)
- Output $y$

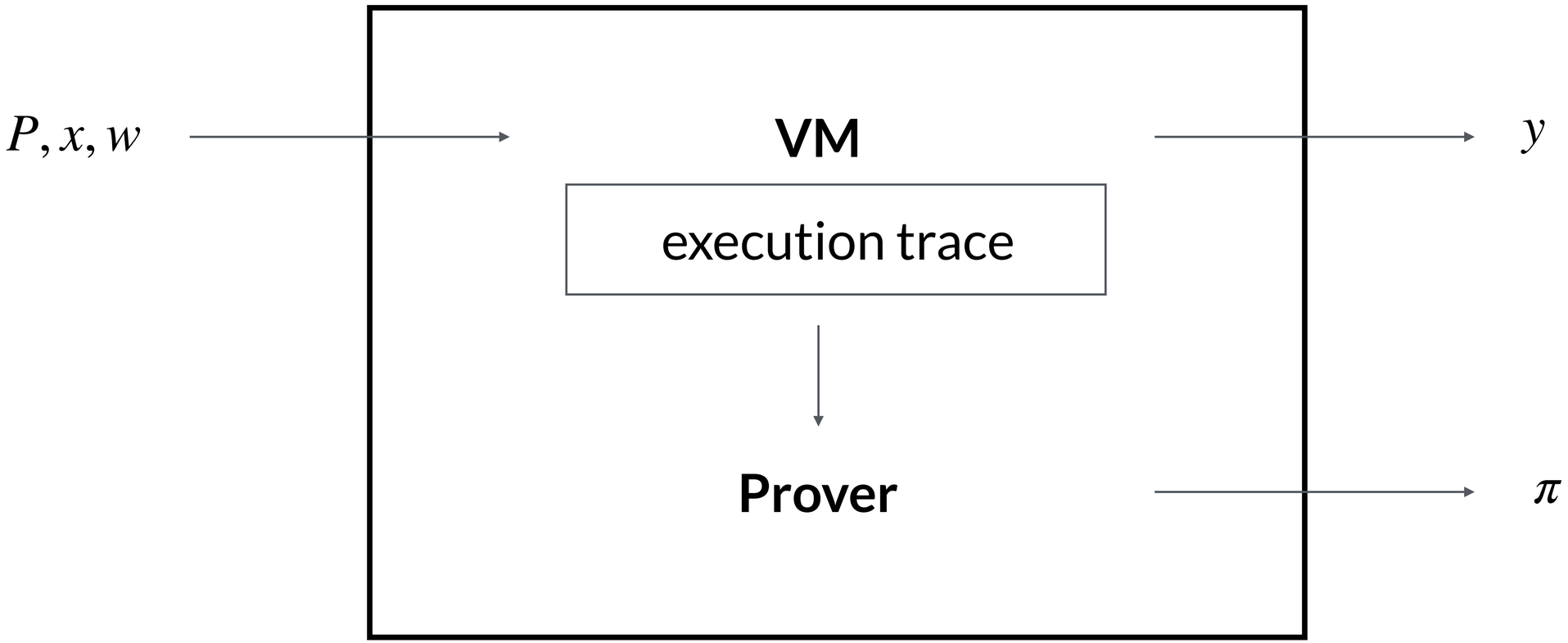
# Zero Knowledge Virtual Machine



# Zero Knowledge Virtual Machine & matching verification algorithm



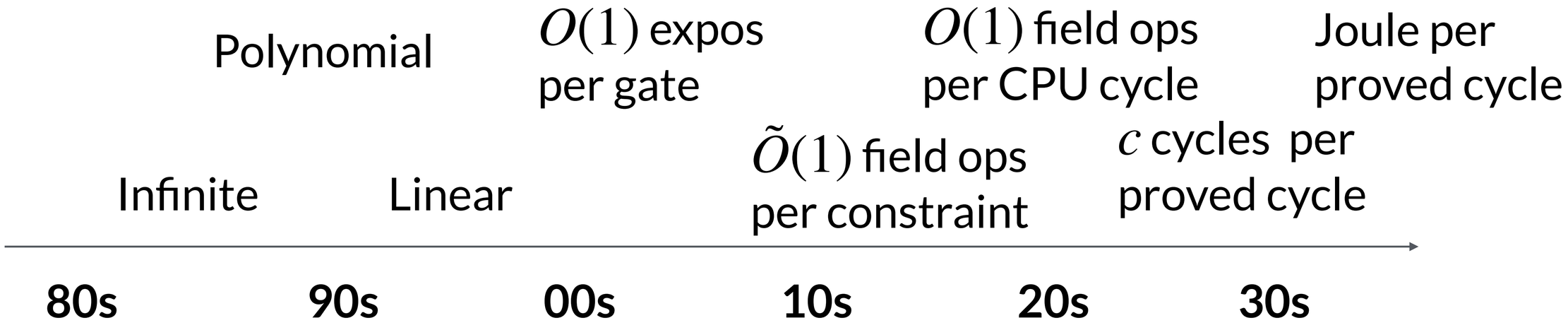
# Inside the zkVM



## Prover efficiency

- CPUs: #transistors per IC increases ~50%/year (Moore's law)
- zkVMs: #proved CPU cycles per second increases >1000%/year (in 2024)

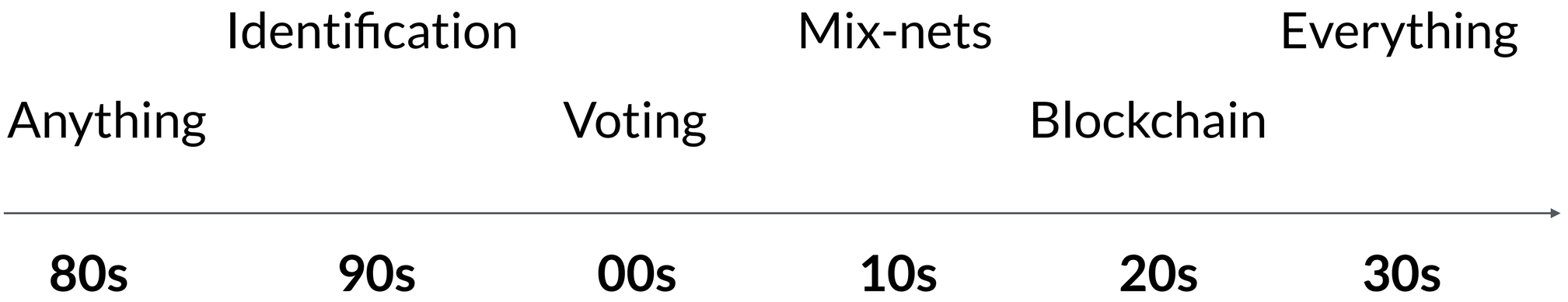
**Polite language amongst cryptographers when describing prover time**



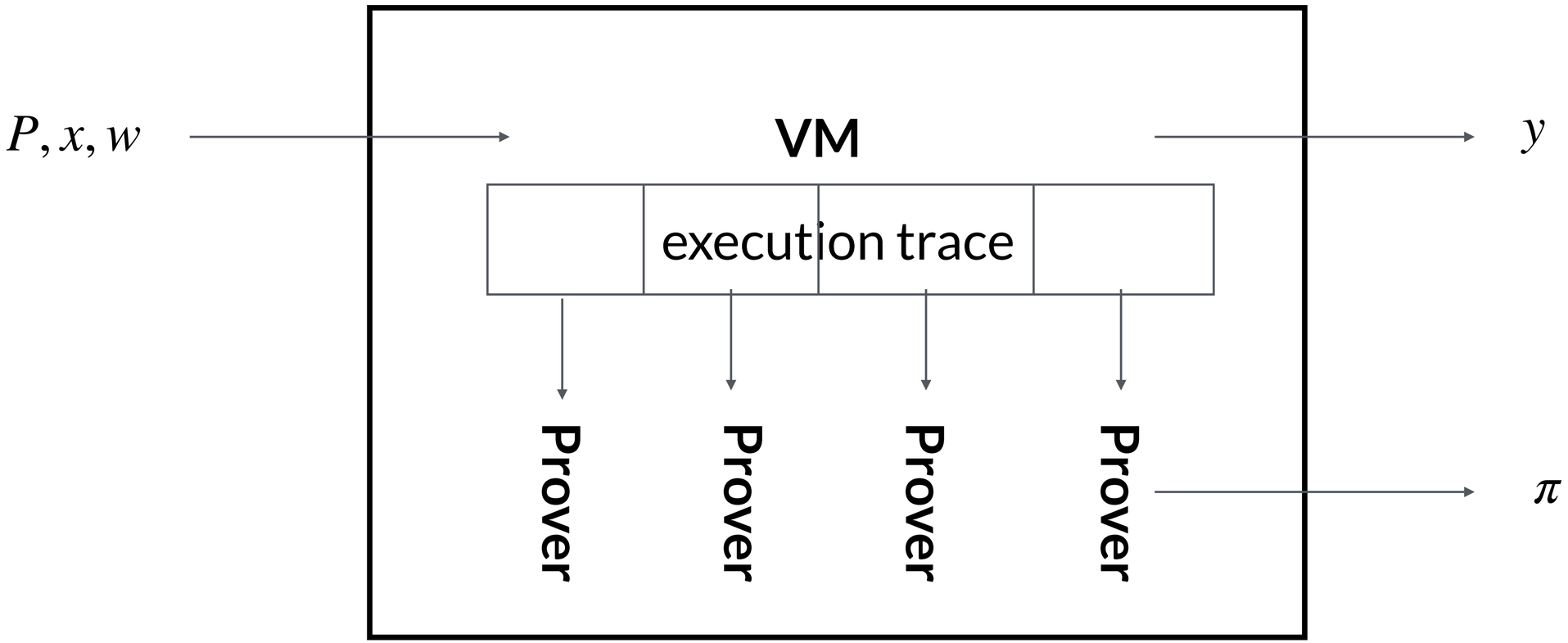
## Prover efficiency is still the bottleneck

- Proof size ✓
- Verification time ✓
- Proving time ?

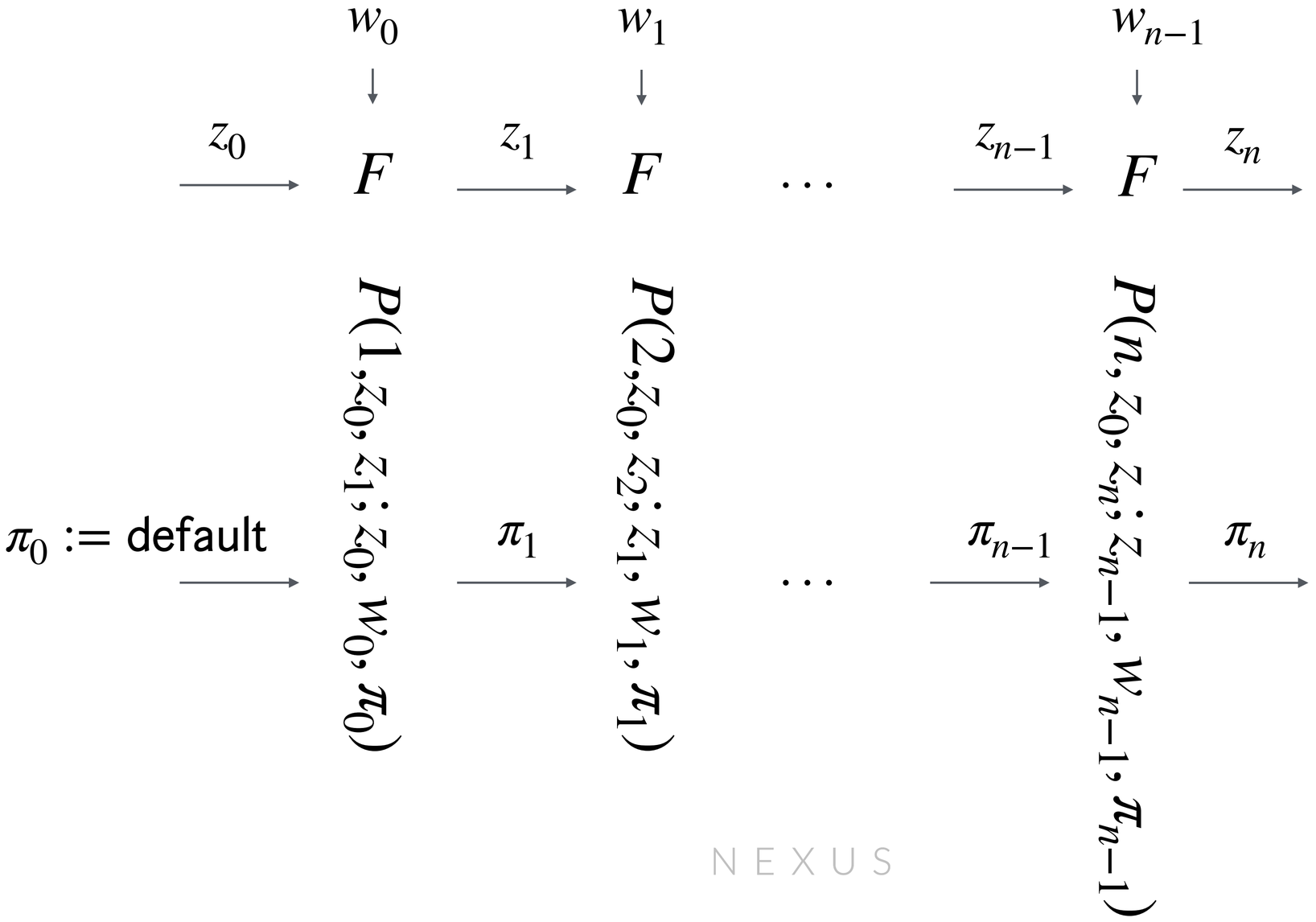
## Timeline of complicated big things to prove



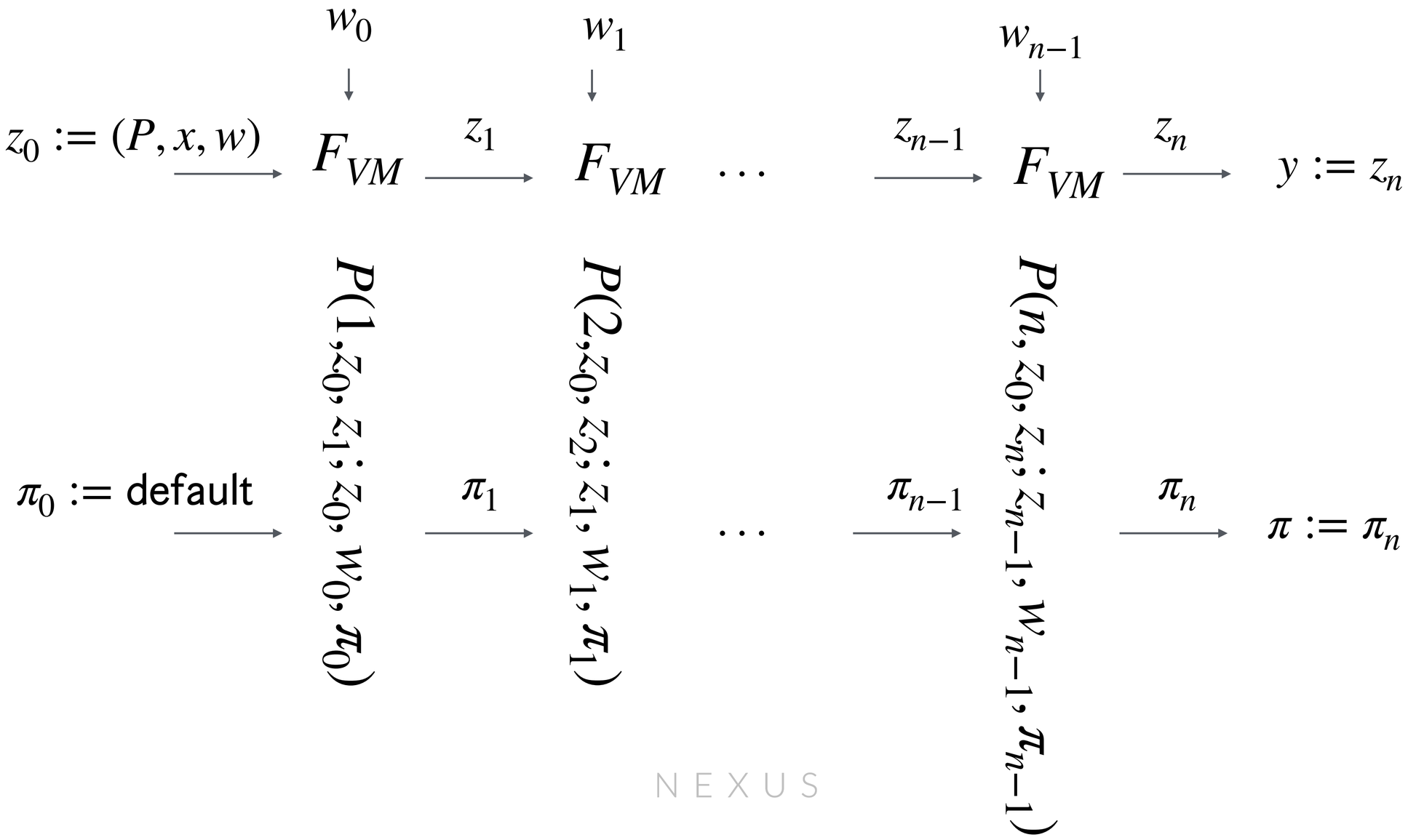
# Slicing a long computation



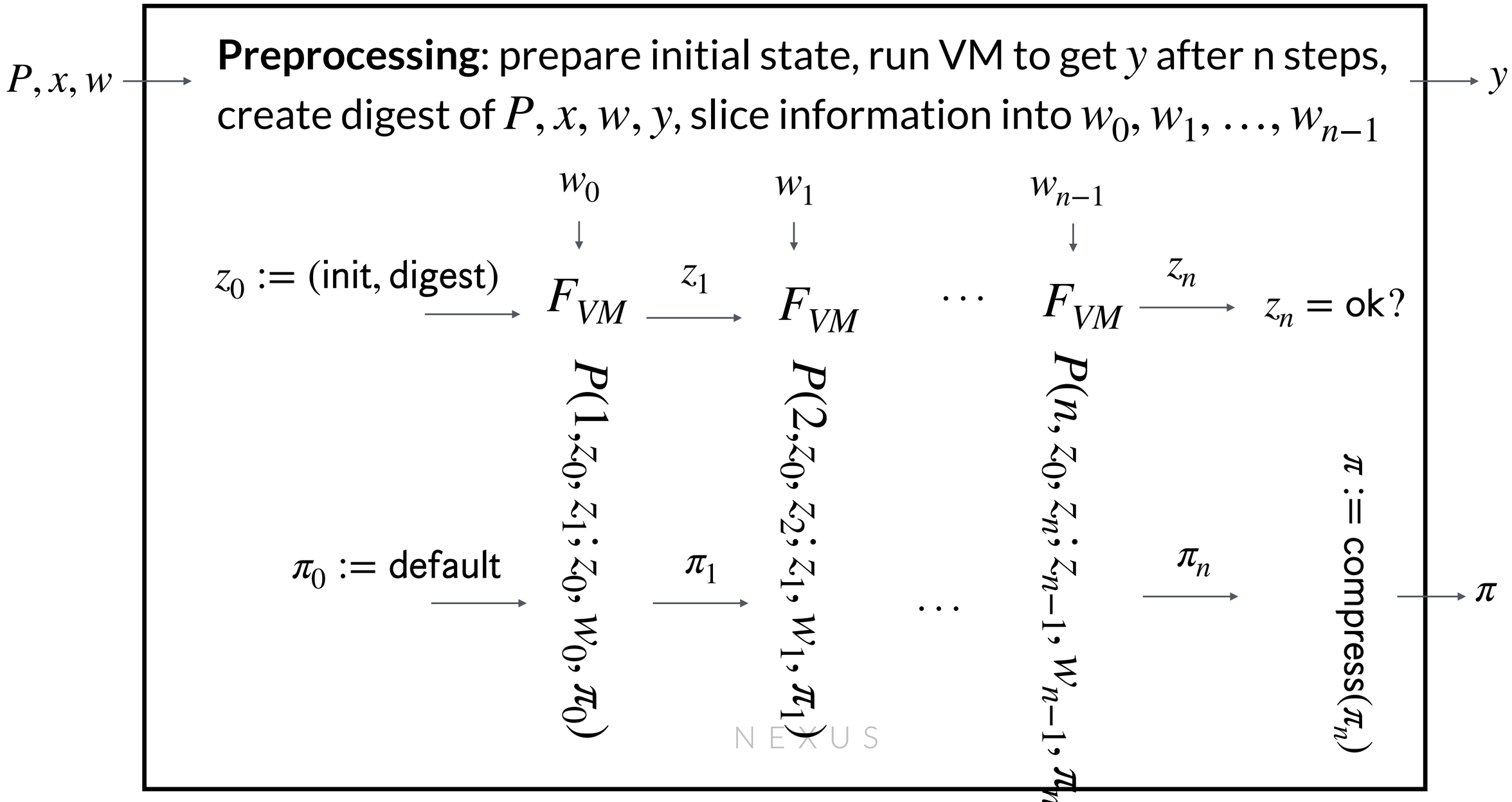
# Incrementally verifiable computation



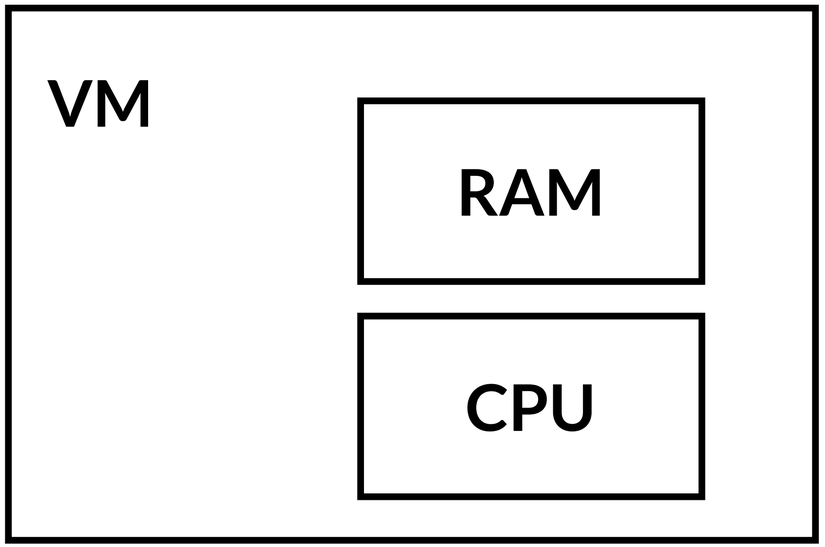
# IVC-based zkVM

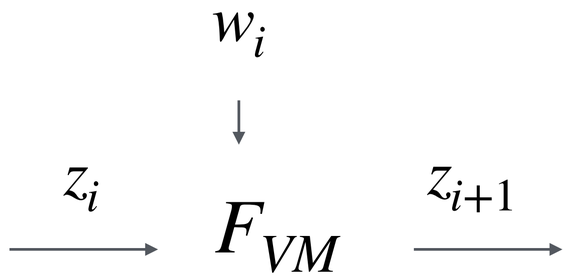


## IVC-handling by the zkVM



# The cost of maintaining memory

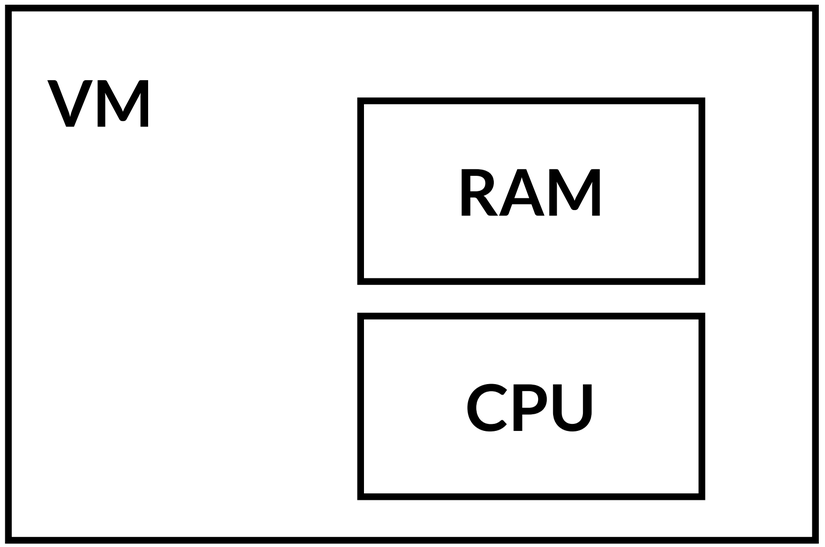



$z_i = (\text{memory}, \text{state}_{\text{CPU}}, \text{aux})$

- Prover cost: constraints for memory transition :-(

# Externalizing the memory



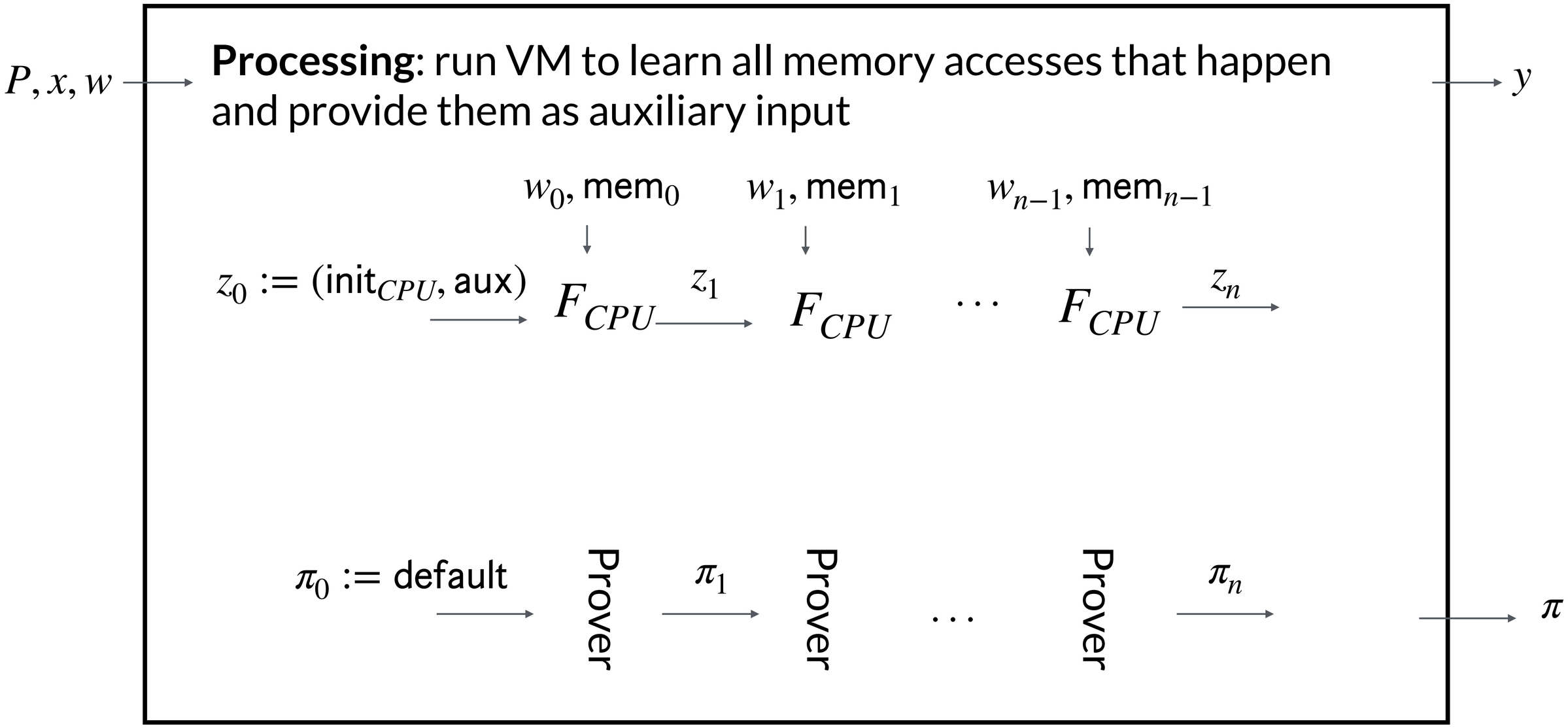

$w_i$, memory accesses

$$z_i \longrightarrow F_{CPU} \longrightarrow z_{i+1}$$

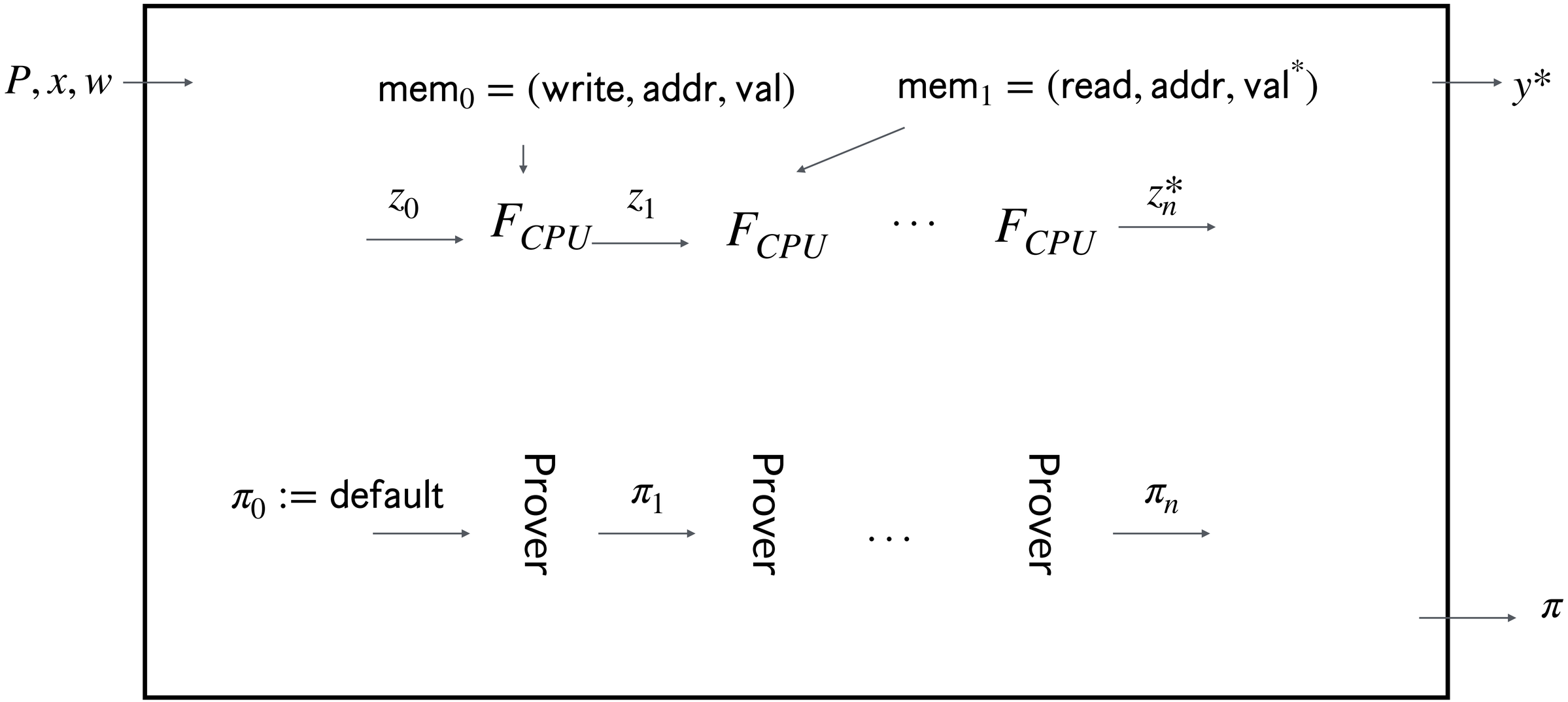
$$z_i = (\text{state}_{\text{CPU}}, \text{aux})$$

- Prover cost: only pay for constraints on CPU state changes such as program counter and registers, and the *actual* changes to memory :-)

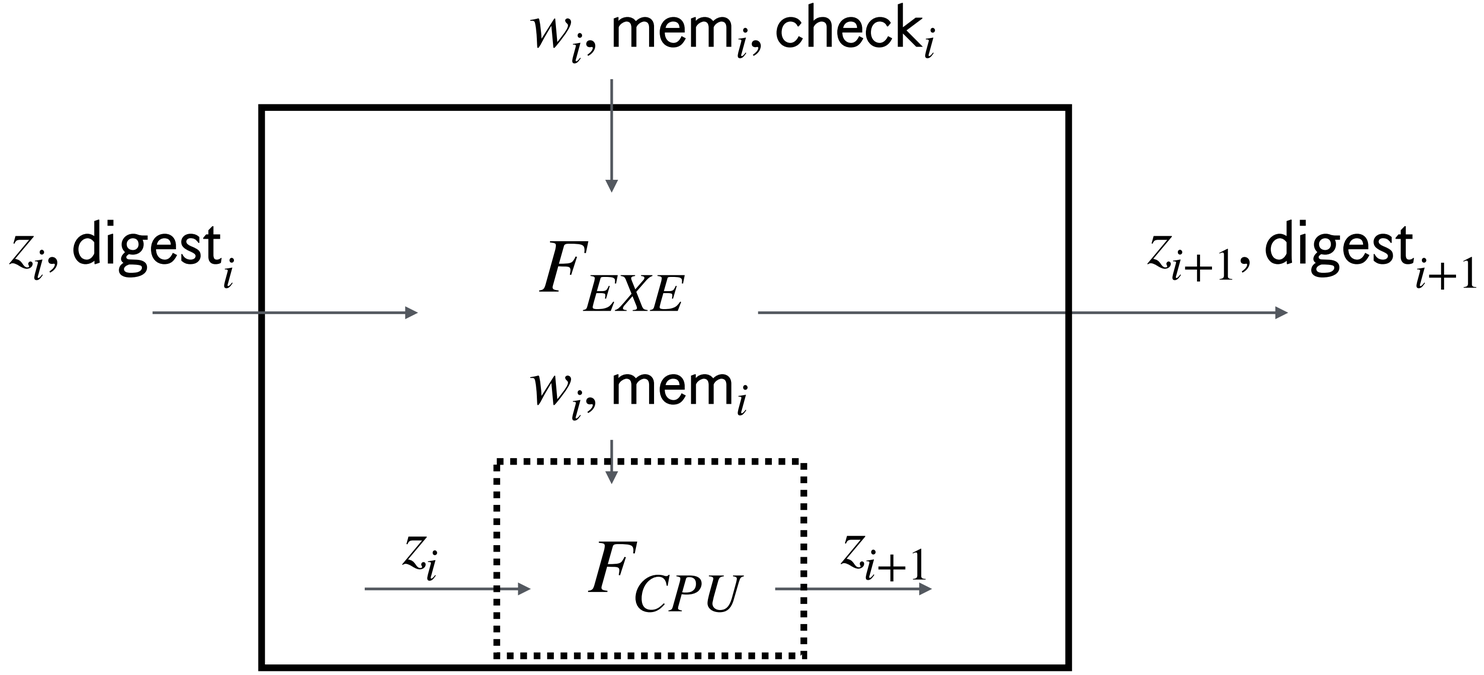
# Strawman zkVM with externalized memory



# Cheating zkVM prover exploiting inconsistent memory

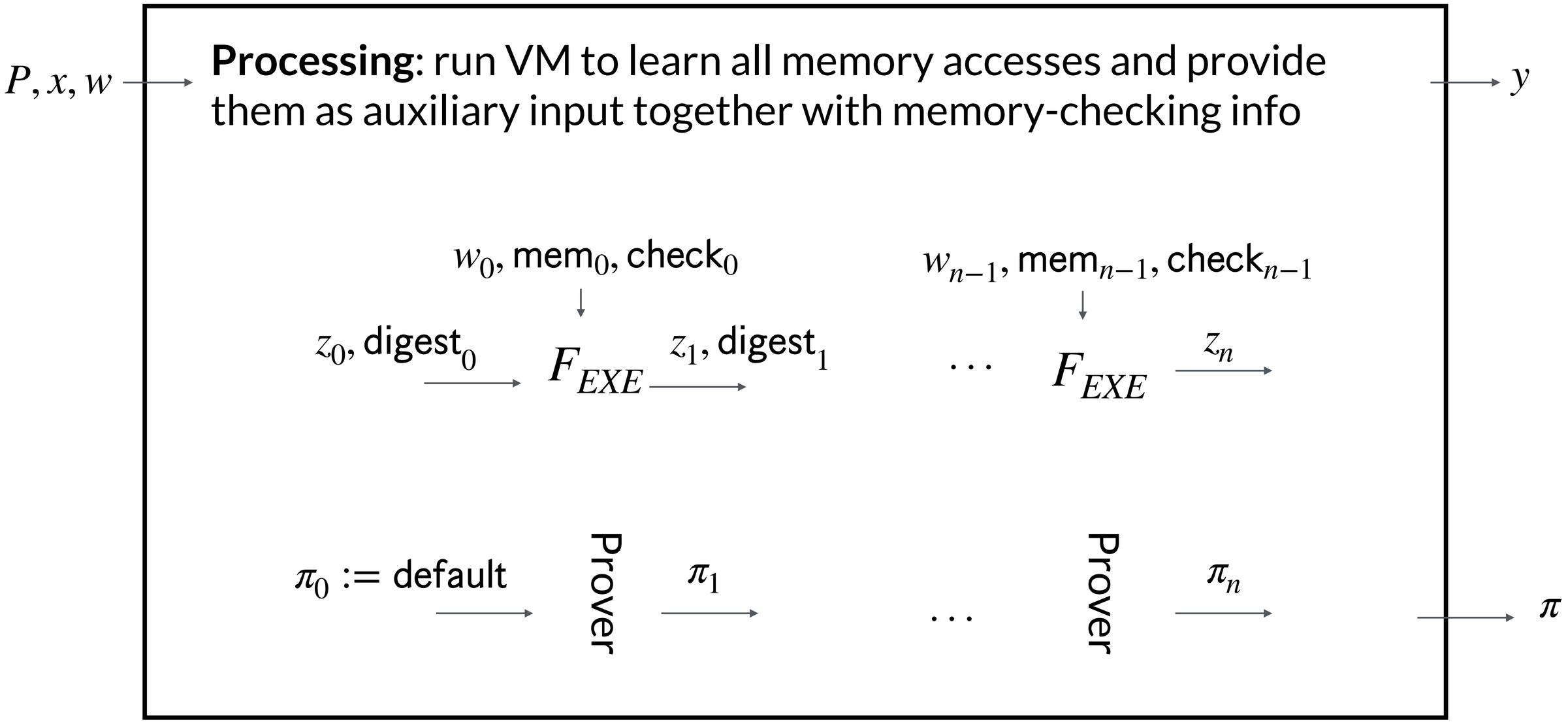


# Memory protection for the CPU

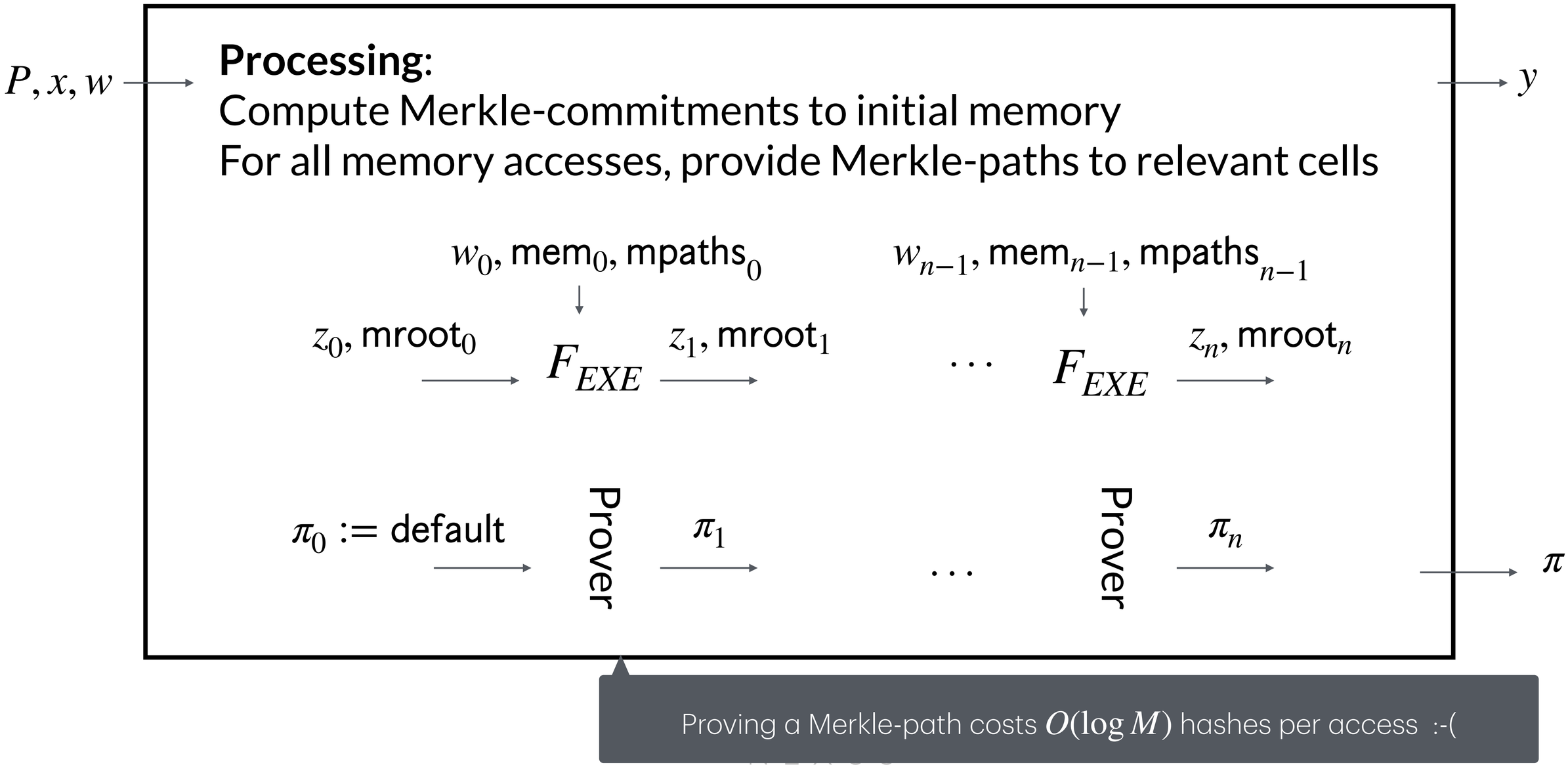



$F_{EXE}$ uses digest$_i$, check$_i$ to check the memory accesses in mem$_i$ are consistent with a global memory access pattern across the IVC computation. Then calls $F_{CPU}$ as a subroutine to get $z_{i+1}$

# zkVM with externalized memory



# Memory checking via merkle trees of the memory



## Keeping track of memory accesses

- Read and write set RS, WS used to track memory accesses
- **Invariant**: RS trails WS by the last values written to memory
- Initialize $WS$ with tuples $(a, v, t = 0)$ // starting memory
- To read/write value $v$ from/to address $a$ at time $t$
  - find $(a, v_{\text{old}}, t_{\text{old}})$ in WS\RS // if a read, check $v = v_{\text{old}}$
  - add $(a, v_{\text{old}}, t_{\text{old}})$ to RS // now RS has caught up with WS in address $a$
  - add $(a, v, t)$ to WS // now RS trails WS in address $a$ again
- When the program is done, add for each address one tuple $(a, v_{\text{old}}, t_{\text{old}})$ to RS
- **Theorem**: memory is consistent if RS = WS

## Multiset hashing

- $\mathbb{G}$ (additive) group where dlog is hard
- $H$ maps a multiset $S$ (of elements in a domain $D$) to an element in $\mathbb{G}$
- Homomorphic: $H(S_1) + H(S_2) = H(S_1 \uplus S_2)$
- Collision resistant: infeasible to find $S_1 \neq S_2$ s.t. $H(S_1) = H(S_2)$
- Instantiation: Let $h : D \to \mathbb{G}$ be a hash function (modeled as a random oracle) and define

$$H(s_1, \ldots, s_n) = h(s_1) + \ldots + h(s_n)$$

# Memory checking via multi-set hashing

$P, x, w$ →

**Processing**:
For all memory accesses, record changes to RS and WS

$w_0, \text{mem}_0, \Delta\text{RS}_0, \Delta WS_0$ ↓

$z_0, rd_0, wd_0 \rightarrow F_{EXE} \rightarrow z_1, rd_1, wd_1$

$\ldots$

$w_{n-1}, \text{mem}_{n-1}, \Delta\text{RS}_{n-1}, \Delta WS_{n-1}$ ↓

$F_{EXE} \rightarrow z_n, rd_n, wd_n$

$rd_n = wd_n ?$

→ $y$

**Memory protection**: check $\text{mem}_i, \Delta\text{RS}_i, \Delta WS_i$ consistency
Update digests

$rd_{i+1} = rd_i + H(\Delta\text{RS}_i) \qquad wd_{i+1} = wd_i + H(\Delta\text{WS}_i)$

→ $\pi$

Proving 2 hashes per access is less expensive than a Merkle path

## Should memory checking be expensive in IVC?

- Monolithic SNARKs pay $O(1)$ field op constraints per memory access
- Recall we can ensure memory is consistent by tracking reads and writes
  Write set WS tracks writes $(a, v, t)$
  The read set RS tracks reads $(a, v, t)$
  We access memory by adding $(a, v_{\text{old}}, t_{\text{old}})$ to RS and $(a, v, t)$ to WS
  We always need $t_{\text{old}} < t$ and when reading $v_{\text{old}} = v$
- After a final read of the memory, we're happy if RS = WS
- So all we need is an efficient check that RS = WS

## Memory checking via LogUp arguments [EagenKRN22, Haböck22]

- RS and WS consist of tuples $(a, v, t)$ appearing in different orders in the execution
- Given a map from tuples $(a, v, t)$ to field elements in $\mathbb{F}$, the RS tuples are represented as $r_1, \ldots, r_n \in \mathbb{F}$ and the WS tuples as $w_1, \ldots, w_n \in \mathbb{F}$
- **Theorem**: $r_1, \ldots, r_n$ is a permutation of $w_1, \ldots, w_n$ if

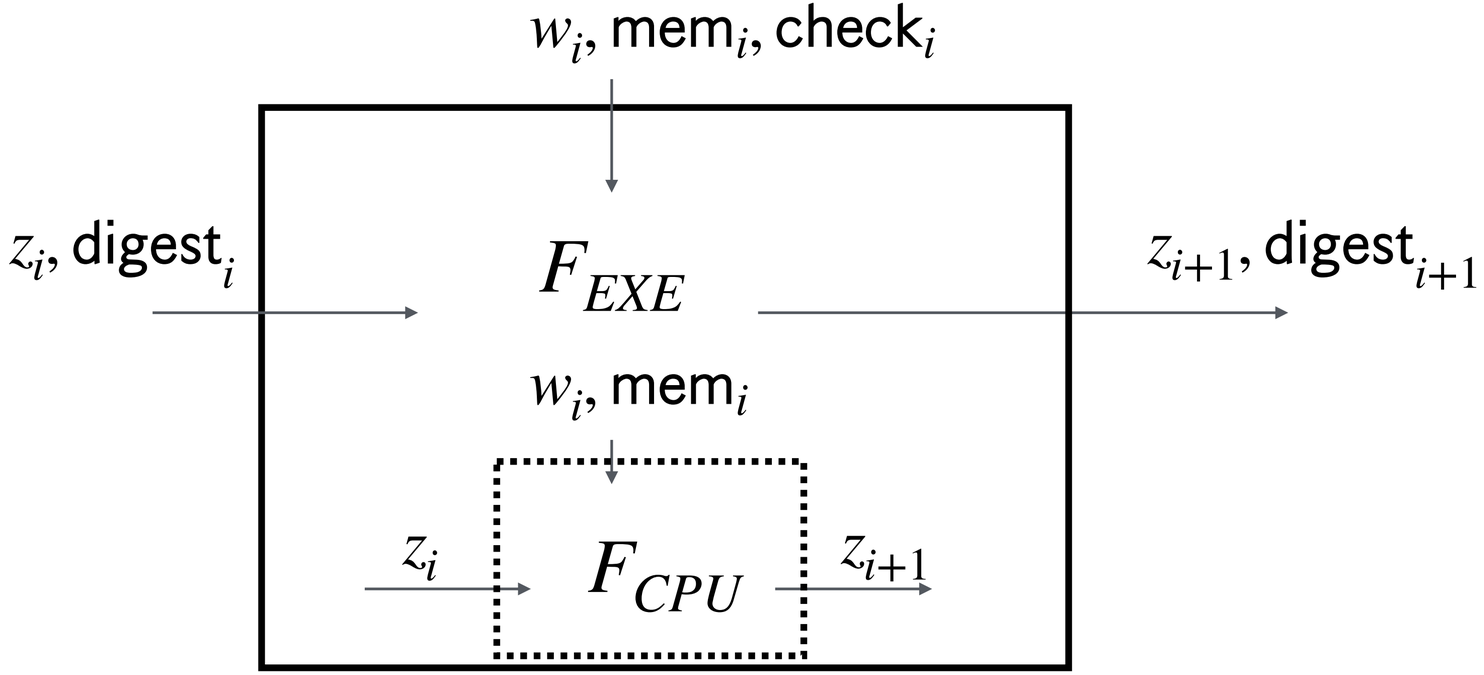
$$\sum \frac{1}{r_i + H} = \sum \frac{1}{w_j + H}$$

- Monolithic SNARKs use a Fiat-Shamir challenge $h \leftarrow \mathbb{F}$ to test this equality

## A suitable challenge

- We would like to check a LogUp test $\sum_i \frac{1}{r_i + H} = \sum_j \frac{1}{w_j + H}$ by plugging in a pseudorandom challenge $h$
- Problem 1
  - Generate $h$ early: the prover can cheat by choosing $r_i, w_j$ that depend on $h$
  - Generate $h$ late: now the IVC is concluded and we missed our chance to use it
- Problem 2
  - The IVC is incremental, no step has the full memory view to compute the sums

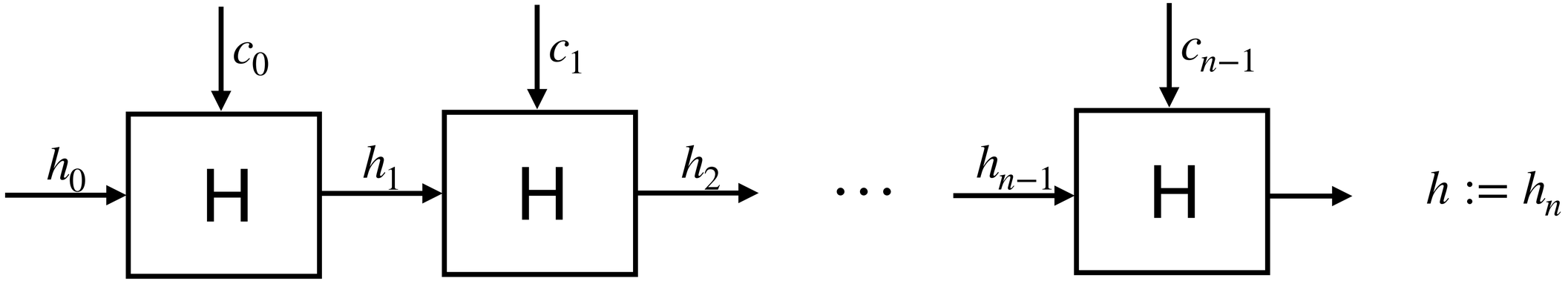
## Recall we have split the step function into memory checking and CPU checking



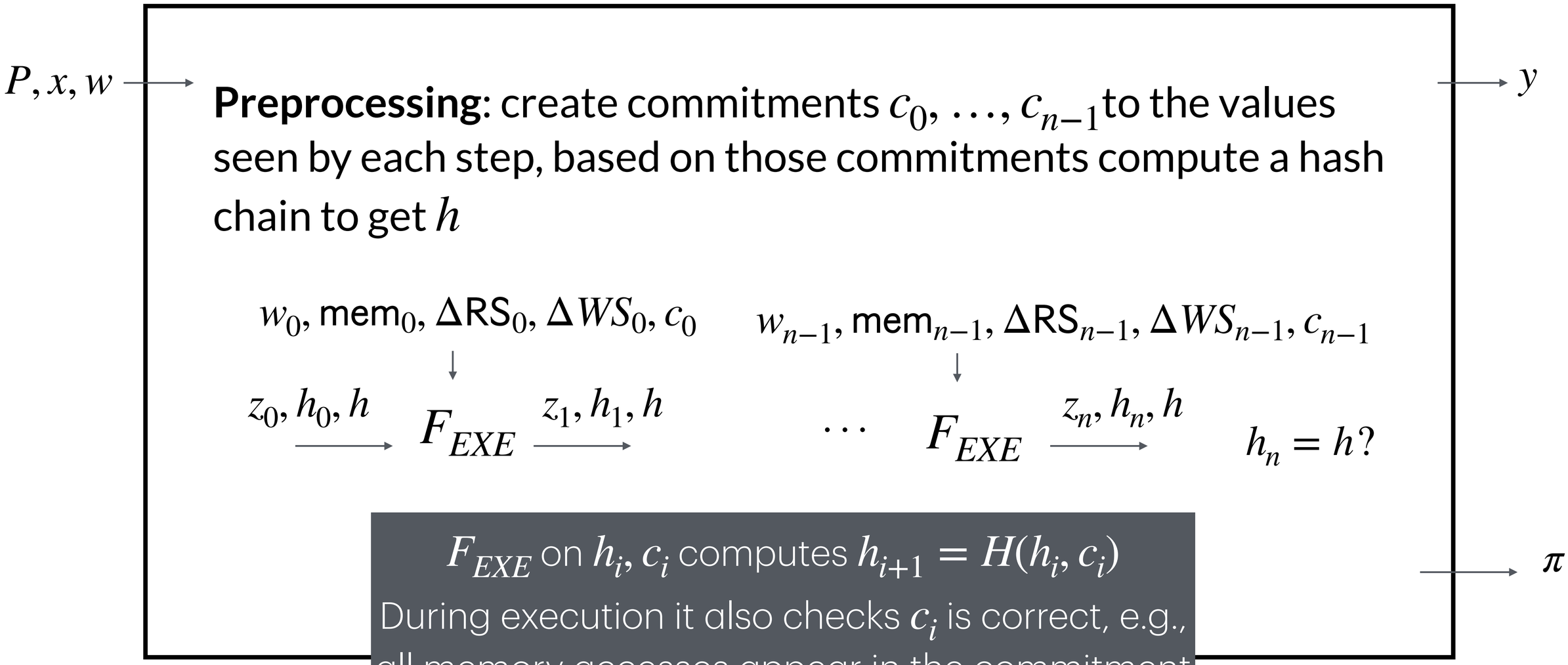

The execution trace is independent of the challenge $h$. It contains CPU state and memory accesses $(a, v, t)$ that can be determined by what $F_{\text{CPU}}$ sees.

## Incrementally building the challenge [Soukhanov23]

- Preprocessing by the zkVM prover
  - run the VM to learn the CPU's view
  - memory accesses are part of the CPU's view, it knows the tuples $(a, v, t)$
  - commit to the CPU's view in each step $c_0, \ldots, c_{n-1}$
  - compute the challenge $h$ via a hash chain (starting at some default $h_0$)



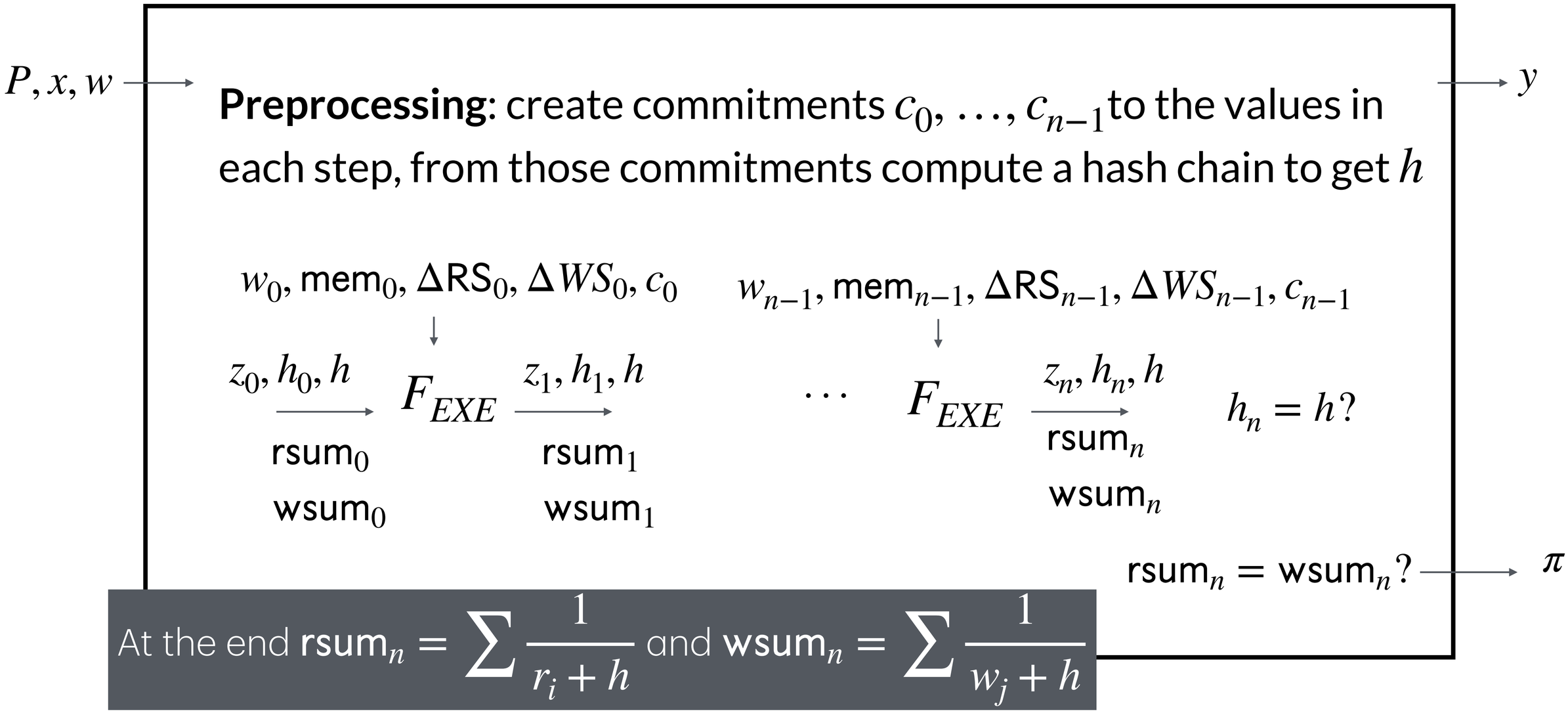
# Incrementally verifying the challenge [Soukhanov23]



## A suitable challenge

- We would like to conduct a LogUp test $\sum_i \frac{1}{r_i + H} = \sum_j \frac{1}{w_j + H}$ by plugging in a pseudorandom challenge $h$
- The zkVM uses the precomputed hash-chain challenge $h$, which it gives as input to the IVC from the start
- The IVC verifies $h$ is correctly computed
- The zkVM prover also keeps track of partial LogUp sums for the read and write sets. At each step it updates the partial LogUp sums according to the memory accesses in this step

# Checking the LogUp equality




At the end $\text{rsum}_n = \sum \frac{1}{r_i + h}$ and $\text{wsum}_n = \sum \frac{1}{w_j + h}$

# Comparing the memory-checking techniques

| Technique | Efficiency<br>(computation to verify per memory access) | Incrementaility<br>(how IVC-like) |
|---|---|---|
| **Merkle** | $O(\log M)$ hashes | High. Can get proof at any step and keep going |
| **Multiset hash** | $O(C)$ hashes | Medium. Can keep going and get proof at any step at the cost of finalizing the memory |
| **2-pass offline memory checking** | $O(1)$ field ops | Low. The first pass dictates the bound on the computation, you cannot get intermediate proofs. |

## Nexus - nexus.xyz

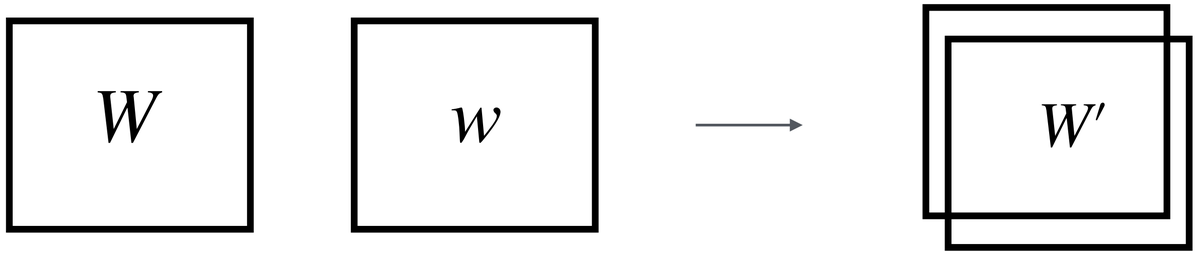
- Nexus 1.0
  - VM inspired by RISC-V
  - Can compile Rust to VM
  - Open source on GitHub, MIT and Apache licensed
  - Not so fast yet: 100 proved CPU cycles/second

- Future
  - 2024 goal: 1T proved CPU cycles/second
  - Path to get there
    a) fast core prover (single threaded)
    b) prover network (massive parallelization)

# Folding

Instance aggregation (prover and verifier)


Witness aggregation (prover)



## Relaxed R1CS

Notation: $\overline{m} \in \mathbb{G}$ is a Pedersen commitment (non-randomized) to $m \in \mathbb{F}^n$

Setup: $A, B, C \in \mathbb{F}^{m \times n}$

Instance: $(\overline{w}, x, r, \overline{e}) \in \mathbb{G} \times \mathbb{F}^{\ell} \times \mathbb{F} \times \mathbb{G}$

Witness: $(w, e) \in \mathbb{F}^{n-1-\ell} \times \mathbb{F}^m$ openings of commitments such that
$z = (x, r, w)$ satisfies $Az \circ Bz = rCz + e$

Note: R1CS (non-relaxed) has $r = 1$ and $\overline{e} = \overline{0}$

## Example: Nova folding scheme

Instance aggregation

$(\overline{W}, X, R, \overline{E})$ $(\overline{w}, x, r, \overline{e})$

Prover and verifier

- P: Let $Z = (X, R, W), z = (x, r, w)$
- P: Let $\overline{T} = \overline{AZ \circ Bz + Az \circ BZ - RCz - rCZ}$
- P&V: Fiat-Shamir challenge $\rho$

$\longrightarrow$ $(\overline{W + \rho w}, X + \rho x, R + \rho r, \overline{E + \rho T + \rho^2 e})$

Witness aggregation

$(W, E)$ $(w, e)$ $\longrightarrow$ $(W + \rho w, E + \rho e)$

## Nova modified to split pre-challenge values and post-challenge-values

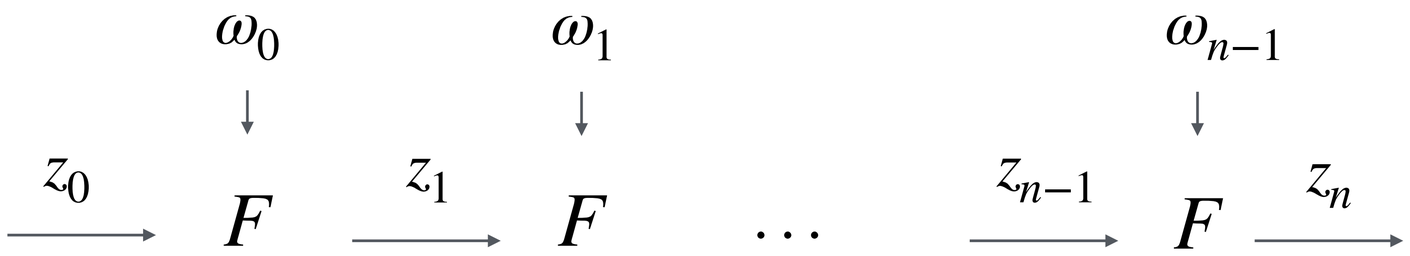
Commitments to values verified in Nova are split into two Pedersen commitments, one to values known during preprocessing before $h$ is known and before IVC has started.

The corresponding RR1CS variant is:

Instance: $(\overline{w}_{\text{pre}}, \overline{w}_{\text{post}}, x, r, \overline{e}) \in \mathbb{G}^2 \times \mathbb{F}^{\ell} \times \mathbb{F} \times \mathbb{G}$

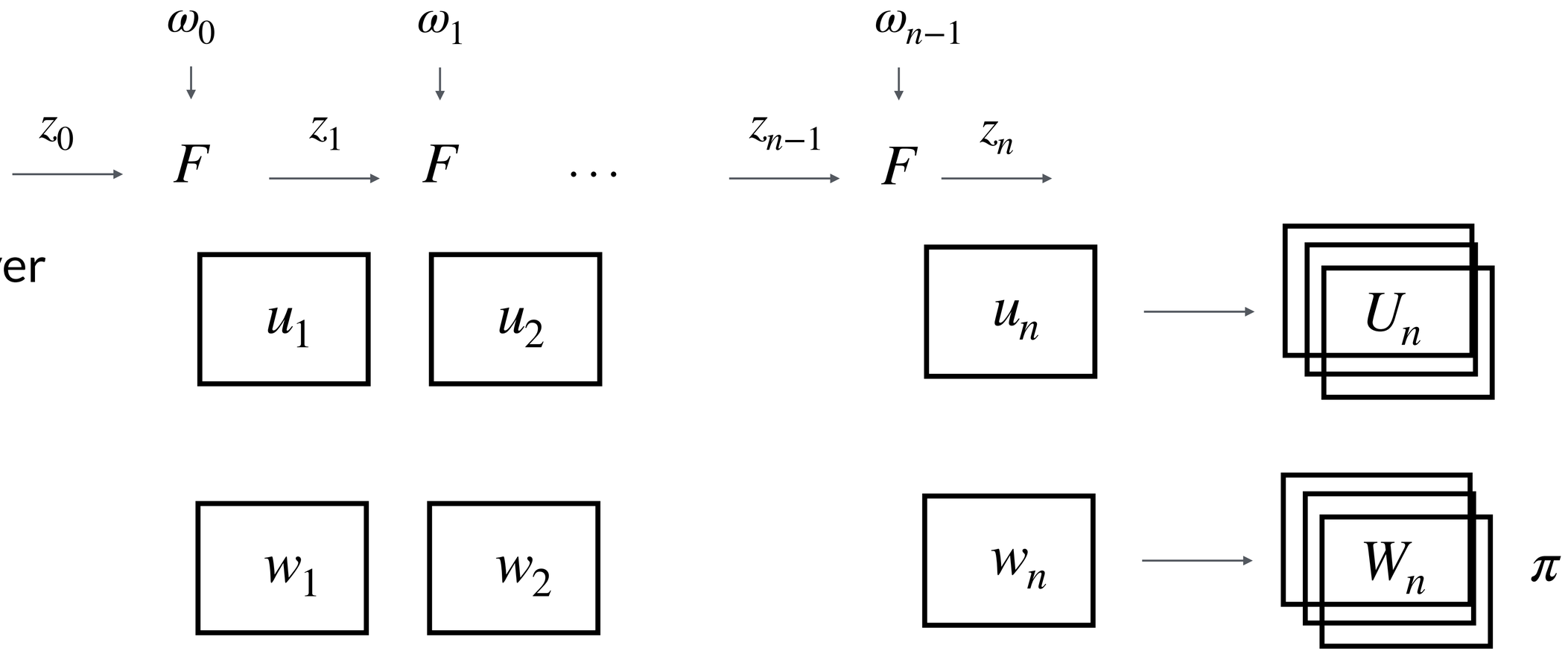
Witness: $(w_{\text{pre}}, w_{\text{post}}, e) \in \mathbb{F}^{n-1-\ell} \times \mathbb{F}^m$ openings of commitments such that $z = (x, r, w_{\text{pre}}, w_{\text{post}})$ satisfies $Az \circ Bz = rCz + e$

# Incrementally verifiable computation based on folding - intuition



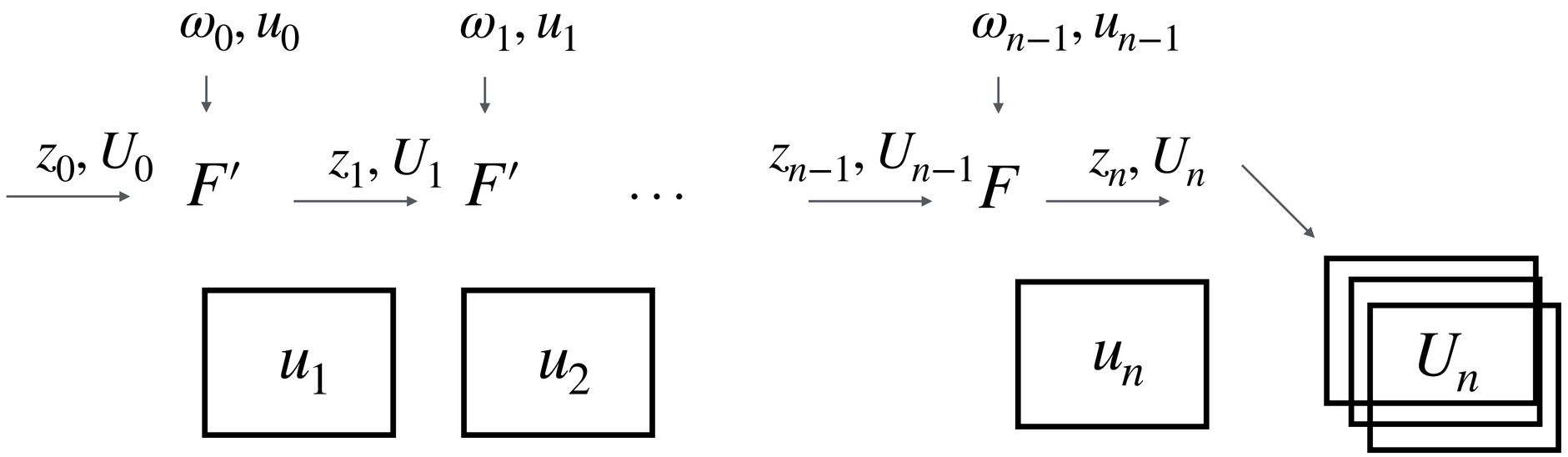

- $u_i$ R1CS instance claiming the current step $i$ gives the correct $z_i$
- $U_i$ RR1CS instance containing aggregated folding of $u_1, u_2, \ldots, u_{i-1}$
- $w_i$ witness for correct computation in step $i$
- $W_i$ aggregated RR1CS witness for $U_i$

# Incrementally verifiable computation based on folding - intuition




Does not work, the prover could cheat with $U_n$

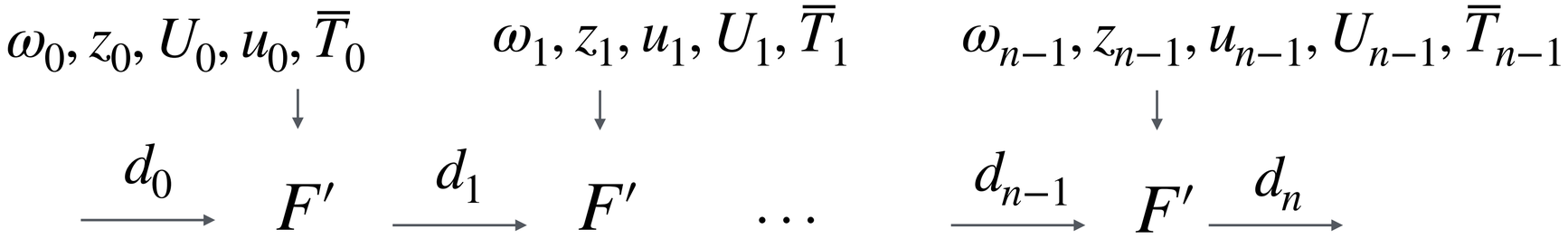
# Incrementally verifiable computation based on folding - augmented step function




Prover


Better, but still sweeping things under the rug

## Incrementally verifiable computation based on Nova folding




- $u_i$ R1CS instance claiming the current step $i$ is correctly computed and that $U_i$ was correctly computed as a fold of $U_{i-1}, u_{i-1}$
- $U_i$ RR1CS instance containing aggregated folding of $u_1, u_2, \ldots, u_{i-1}$
- $d_i$ A digest (hash) of $z_i, U_i$ // compact state, larger auxiliary input
- $F'$ Augmented step function that unpacks digest, folds $u_{i-1}, U_{i-1}$ to get $U_i$, computes $z_i = F(z_{i-1}, \omega_{i-1})$, and the digest $d_i$ of $z_i, U_i$