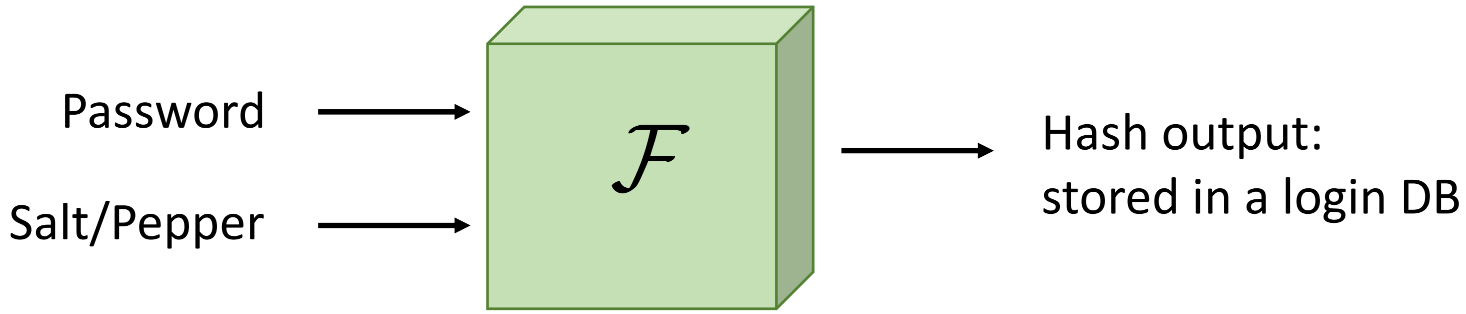
Static-Memory-Hard Functions, and Modeling the Cost of Space vs. Time

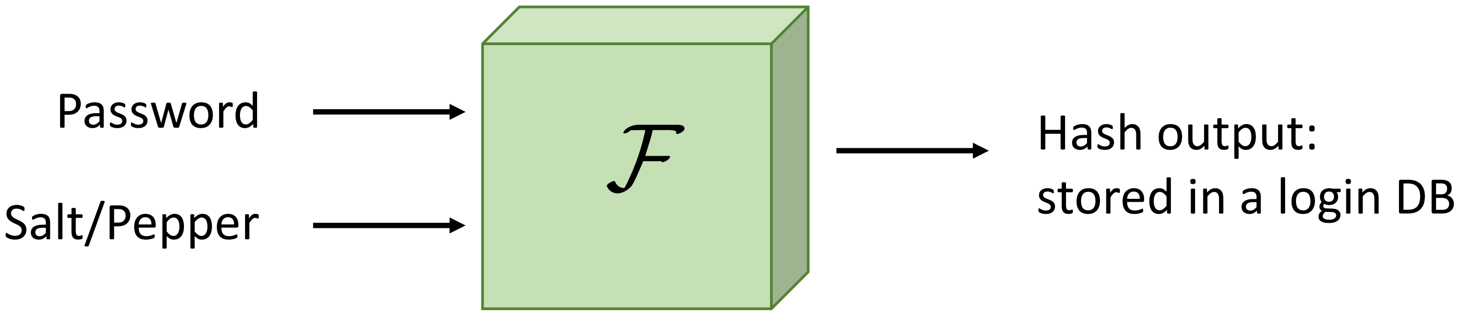
Thaddeus Dryja, Quanquan C. Liu, Sunoo Park

MIT

Password Hashing

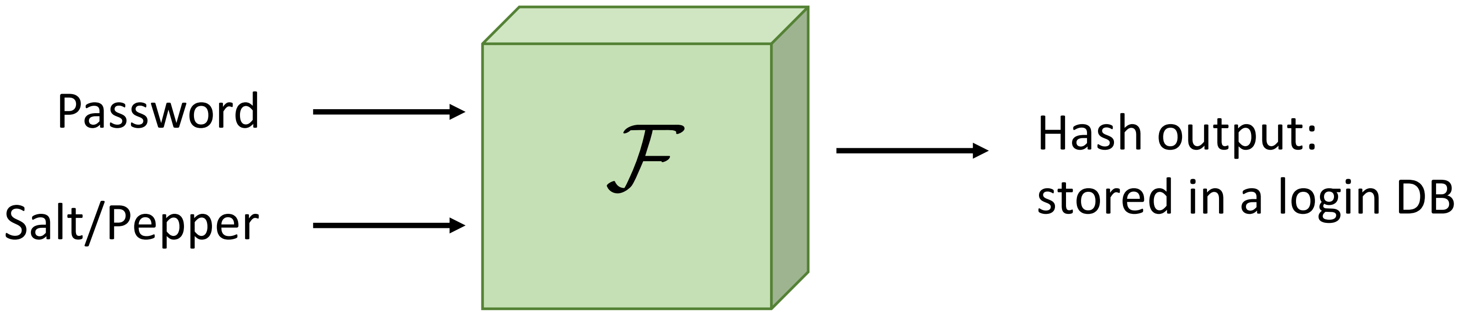


Password Hashing



Honest evaluators need only use $\mathcal{F}$ few times

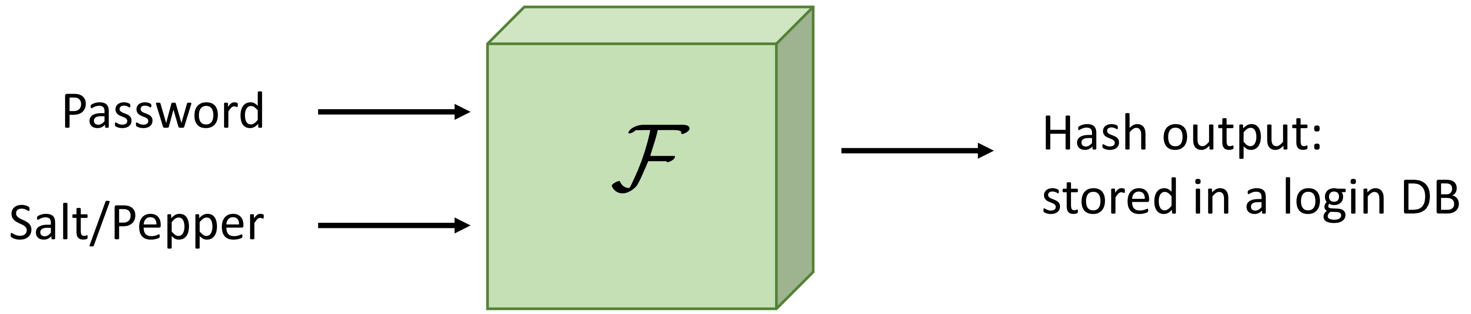
Password Hashing



Honest evaluators need only use $\mathcal{F}$ few times

Adversaries may run $\mathcal{F}$ many times (e.g. large-scale server attacks).

Password Hashing



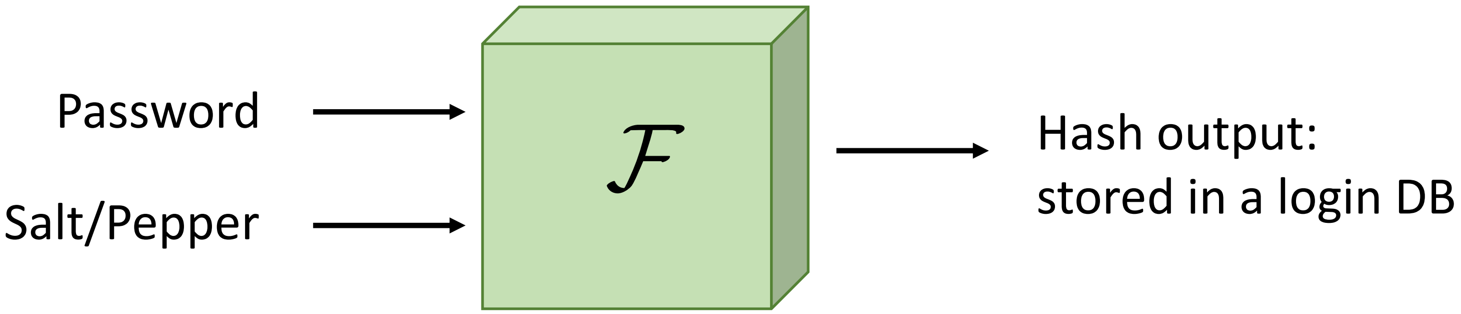
Honest evaluators need only use $\mathcal{F}$ few times

Adversaries may run $\mathcal{F}$ many times (e.g. large-scale server attacks).

Desirable goal:

Make brute-force attacks hard by making $\mathcal{F}$ hard to compute over many hashes.

Password Hashing



Honest evaluators need only use $\mathcal{F}$ few times

Adversaries may run $\mathcal{F}$ many times (e.g. large-scale server attacks).

Desirable goal:

Make brute-force attacks hard by making $\mathcal{F}$ hard to compute over many hashes.

(Not implied by traditional hash function guarantees like collision-resistance.)

Honest Evaluator vs. Adversary

Honest Evaluator	**Adversary**

• Few evaluations
  • Total cost ≈ cost per evaluation
	• Many evaluations
	  • Total cost = cost of many evaluations

Honest Evaluator vs. Adversary

Honest Evaluator

- Few evaluations
  - Total cost ≈ cost per evaluation
  - Cannot amortize costs

Adversary

- Many evaluations
  - Total cost = cost of many evaluations
  - Can use amortization / parallelization

Honest Evaluator vs. Adversary

Honest Evaluator

- Few evaluations
  - Total cost ≈ cost per evaluation
  - Cannot amortize costs
- General-purpose hardware

Adversary

- Many evaluations
  - Total cost = cost of many evaluations
  - Can use amortization / parallelization
- Special-purpose hardware
  (e.g., ASICs optimized for hash computation)

Honest Evaluator vs. Adversary

Honest Evaluator	**Adversary**

• Few evaluations
  • Total cost ≈ cost per evaluation
  • Cannot amortize costs
• General-purpose hardware
	• Many evaluations
	  • Total cost = cost of many evaluations
	  • Can use amortization / parallelization
	• Special-purpose hardware (e.g., ASICs optimized for hash computation)

Desirable goal:

Make brute-force attacks hard by making $\mathcal{F}$ hard to compute over many hashes

Honest Evaluator vs. Adversary

Honest Evaluator	**Adversary**

• Few evaluations
  • Total cost ≈ cost per evaluation
  • Cannot amortize costs
	• Many evaluations
	  • Total cost = cost of many evaluations
	  • Can use amortization / parallelization

• General-purpose hardware	• Special-purpose hardware (e.g., ASICs optimized for hash computation)

Desirable goal:

Make brute-force attacks hard by making $\mathcal{F}$ hard to compute over many hashes

even against adversaries with the advantages of **hardware** and **scale**.

Memory Complexity Measures

- Several have been proposed
    1. ST-Complexity
    2. Cumulative Complexity
    3. Sustained Space Complexity
- Each has their strengths and weaknesses

Memory doesn't just mean buying a disk, it could mean renting storage space from AWS, etc.

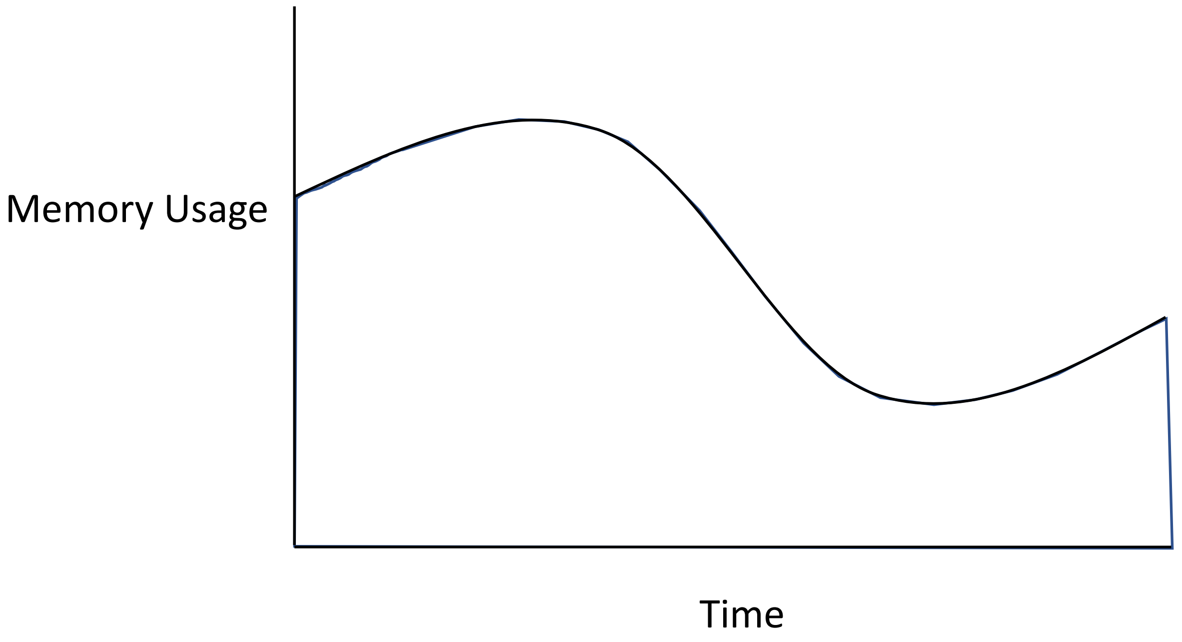
Memory Complexity Measures

- Several have been proposed
  1. ST-Complexity
  2. Cumulative Complexity
  3. Sustained Space Complexity
- Each has their strengths and weaknesses
- Next: a quick overview of prior proposed measures
  - **Then we'll get into our contributions**

Memory doesn't just mean buying a disk, it could mean renting storage space from AWS, etc.

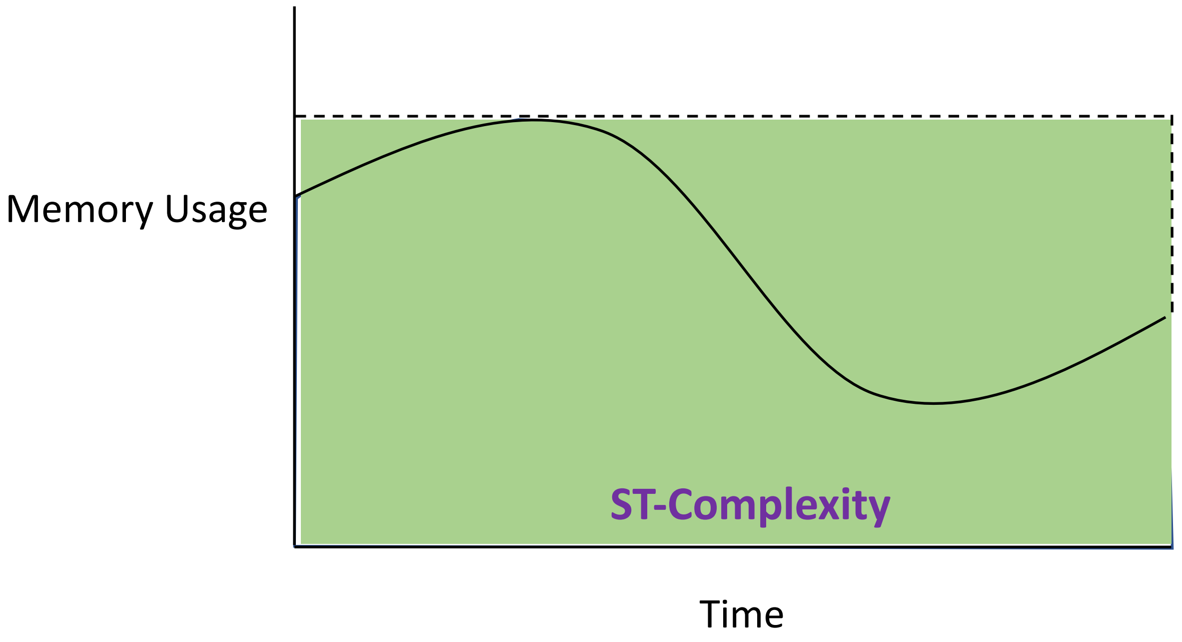
ST-Complexity

- “ST” = “Space-Time” = max (peak) memory usage x time



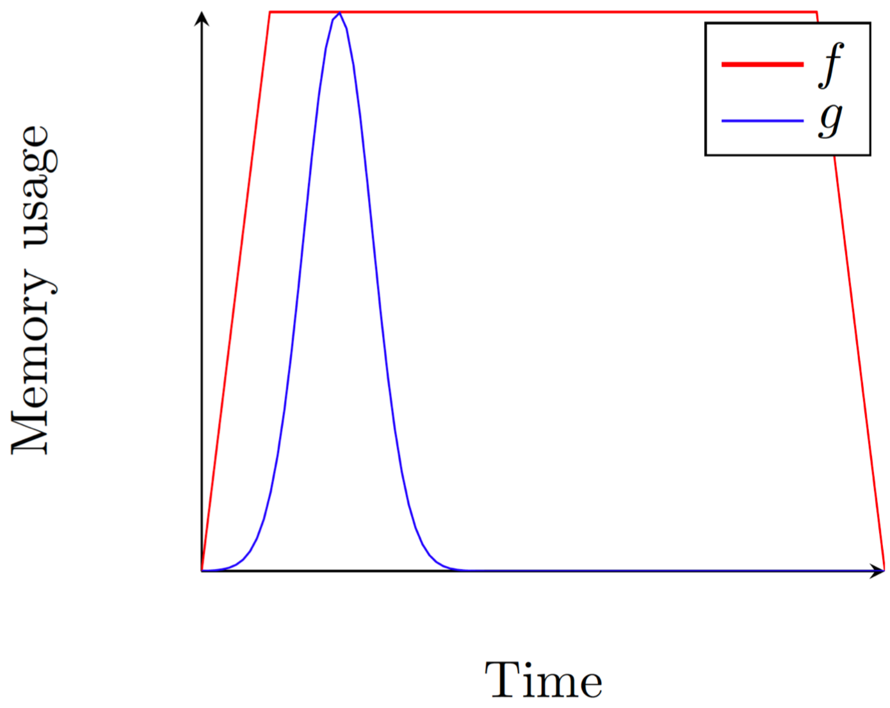
ST-Complexity

- “ST” = “Space-Time” = max (peak) memory usage x time

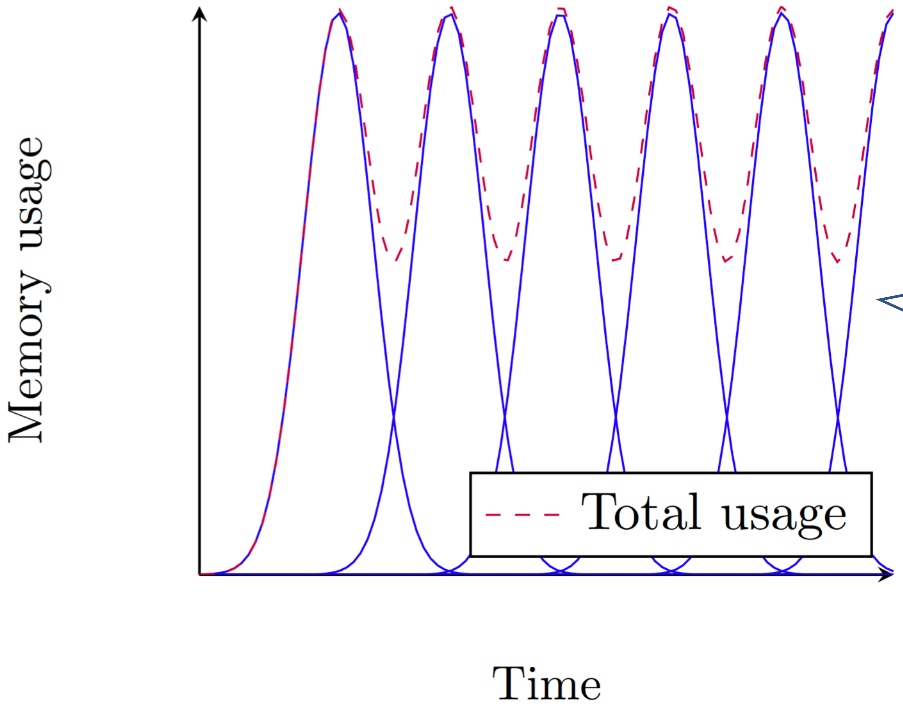


ST-Complexity

- Limitation: does not capture amortization



Honest evaluations
(f & g have same ST-complexity)

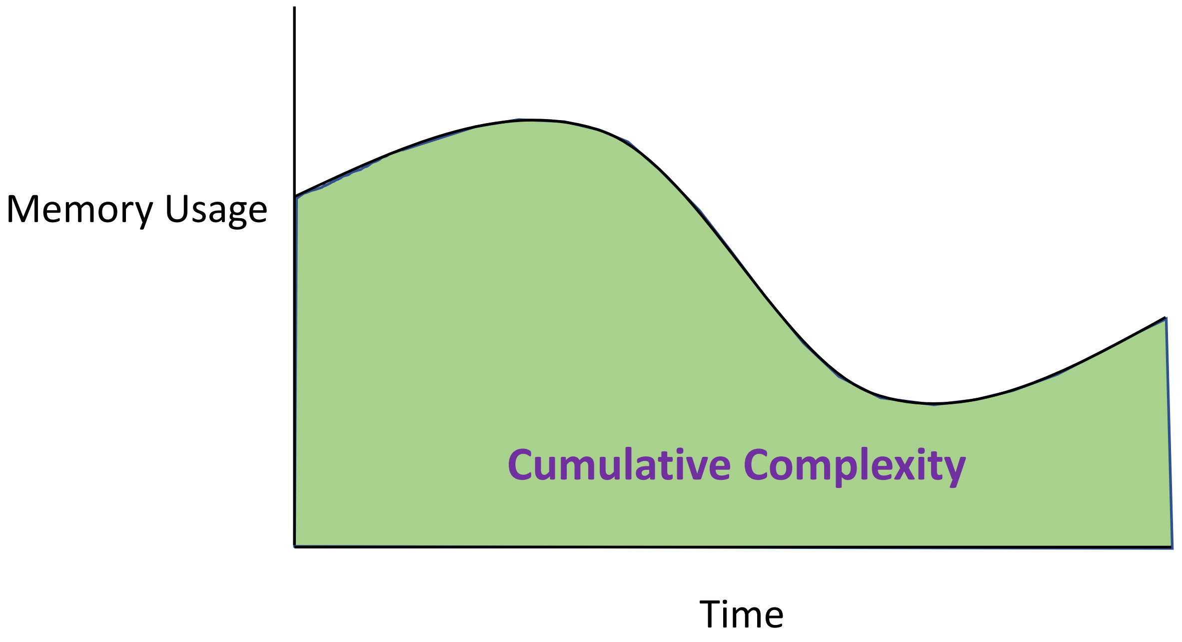


Multiple adversarial evaluations of g

Not robust to amortization attacks!

Cumulative Memory Complexity (CC) [AS15]

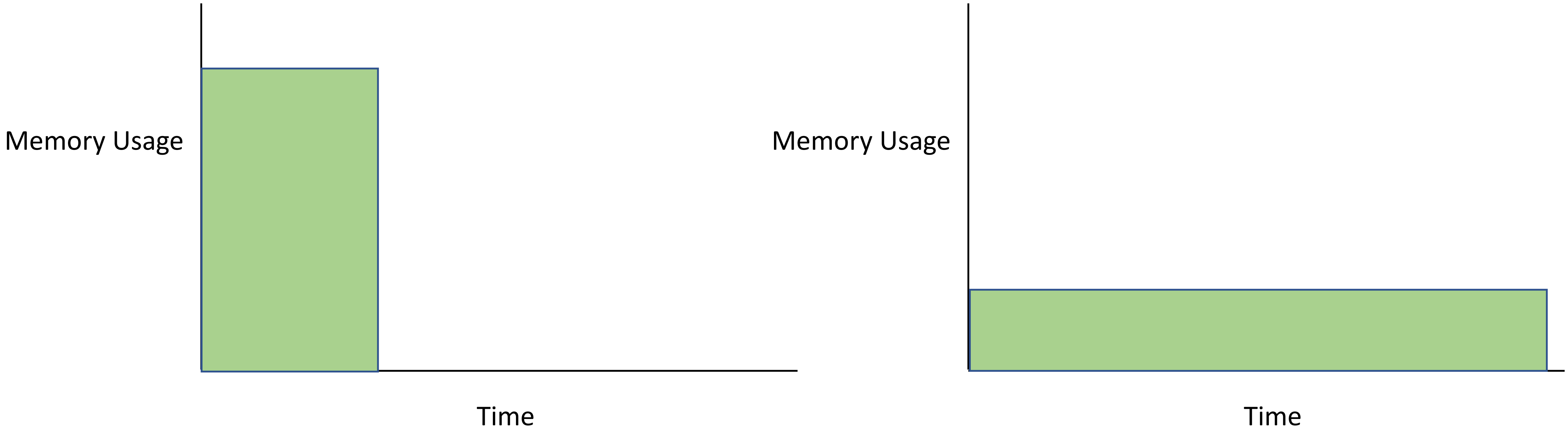
- **<u>Goal</u>**: Solves amortization of cost issue for ST-complexity
- **<u>CC</u>:** Sum of the memory used over time



Cumulative Memory Complexity (CC) [AS15]

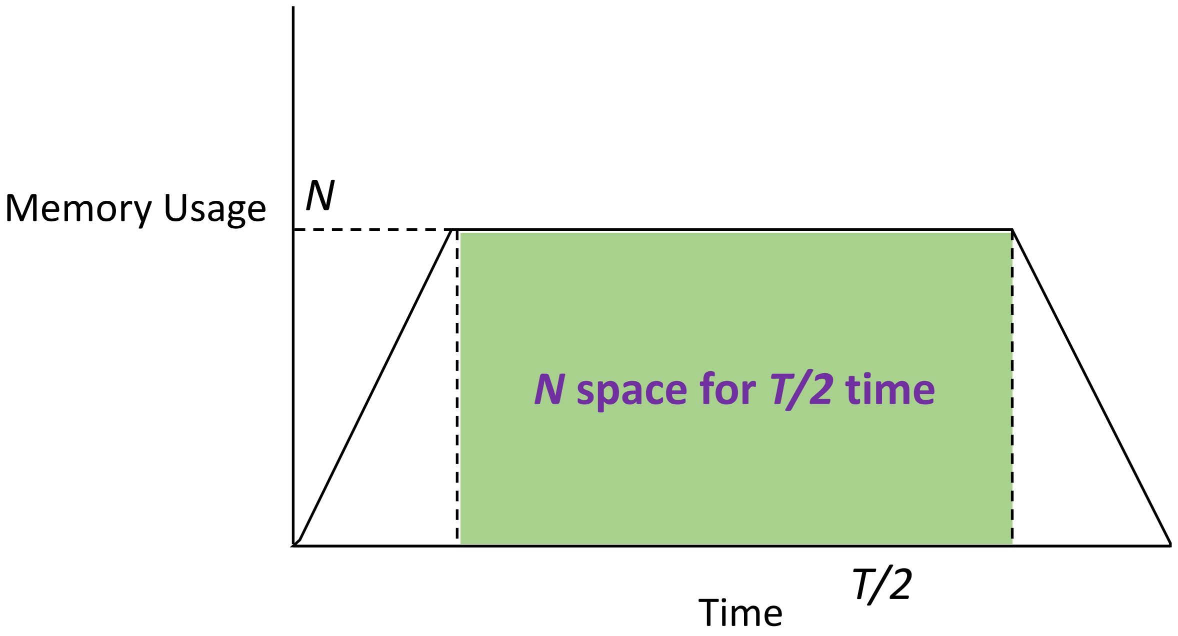
- **<u>Caveat</u>**: Could still result in different hardware costs (e.g. one time cost or cost varies with time)

Same CC but **different cost i.e. cost is not uniform w.r.t. time**



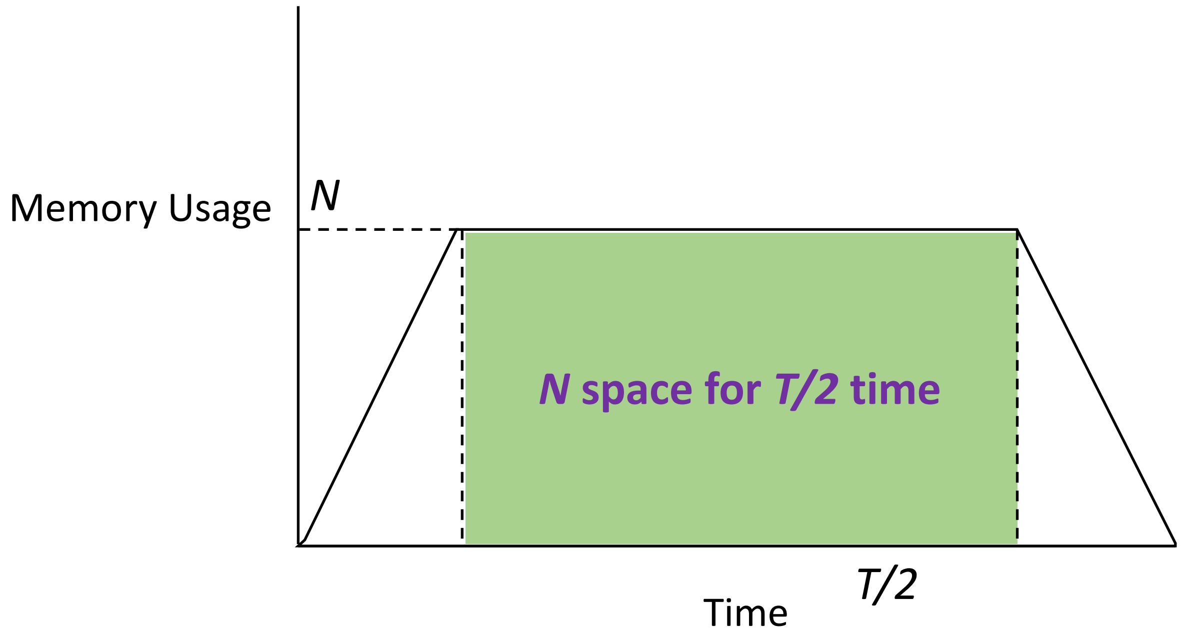
Sustained-Space Complexity (SSC) [ABP17]

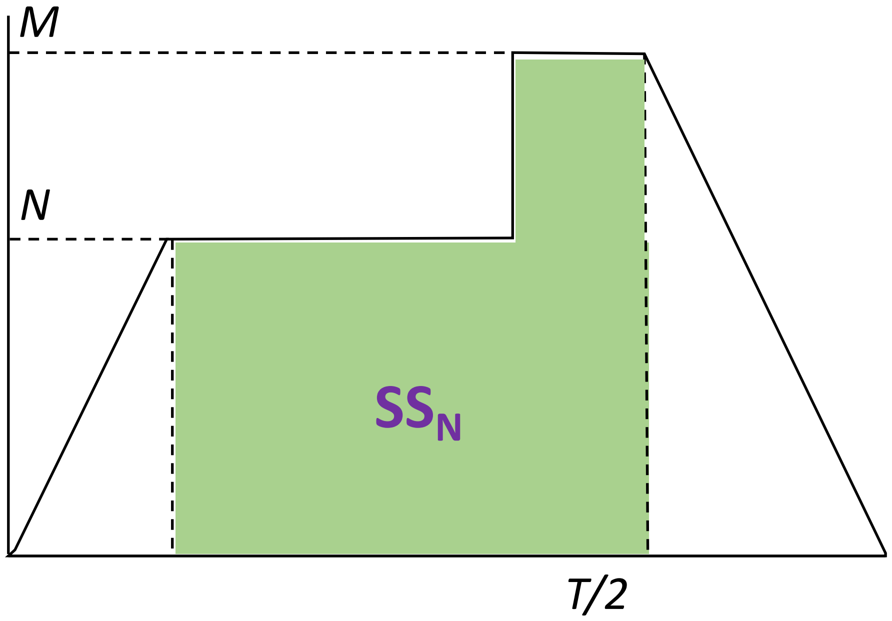
- **SSC_N:** Space usage ≥ N is **sustained** for a period of time



Sustained Space Complexity (SSC) [ABP17]

- **SSC_N:** Space usage ≥ N is **sustained** for a period of time

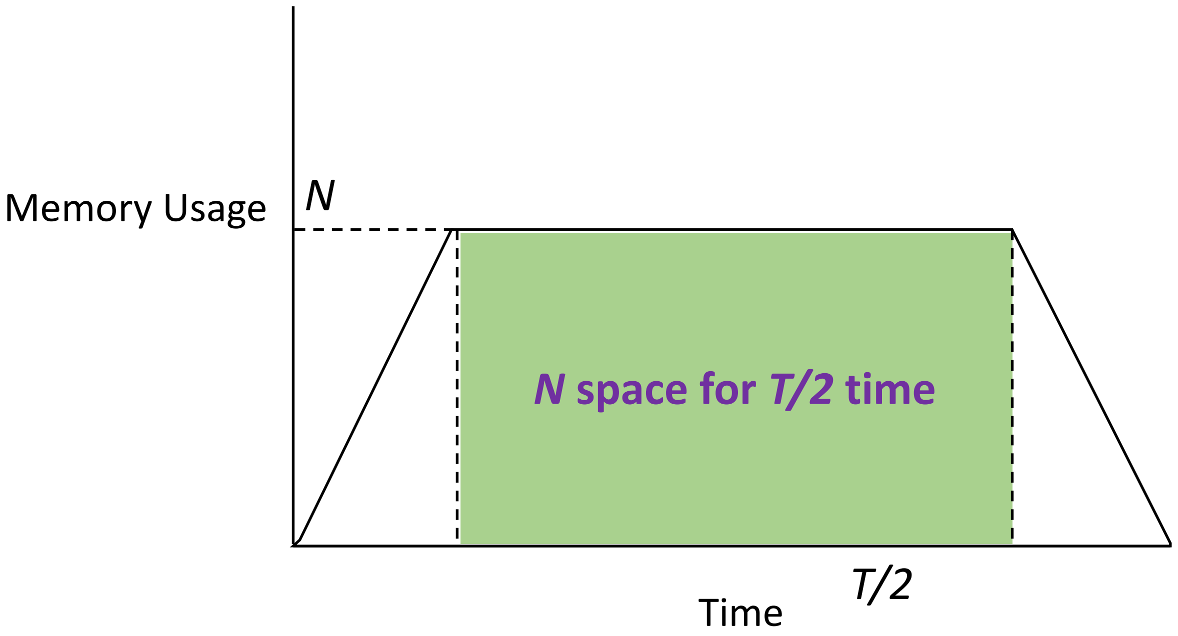


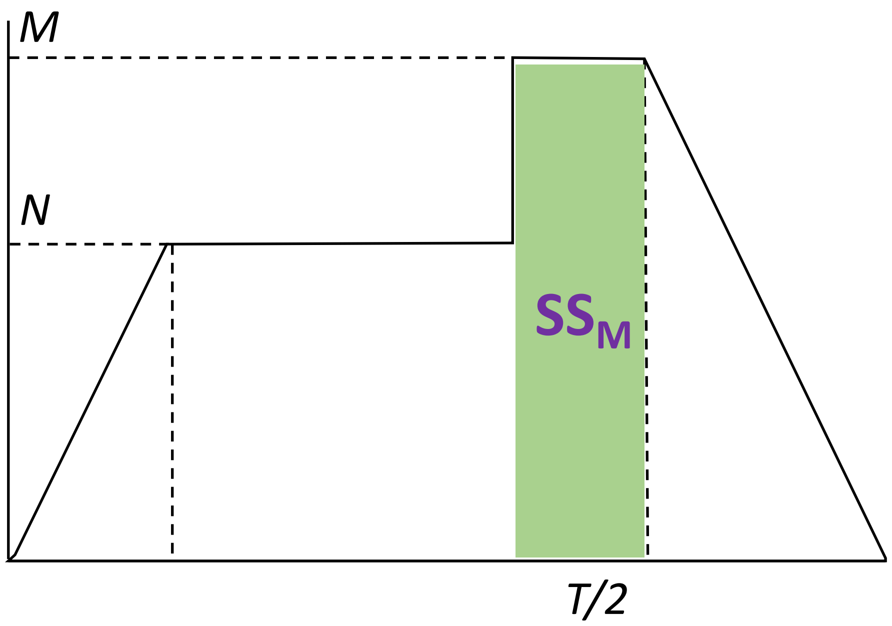


Inherently parameterized notion
(what's the most informative N for a given situation?)

Sustained Space Complexity (SSC) [ABP17]

- **SSC_N:** Space usage ≥ N is **sustained** for a period of time



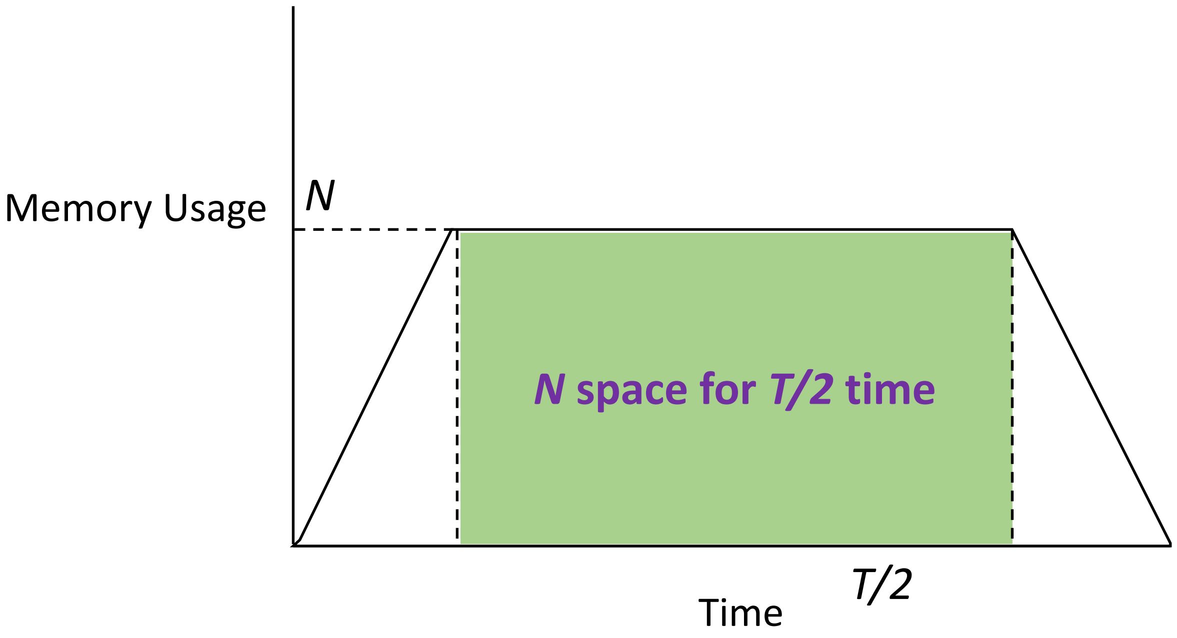


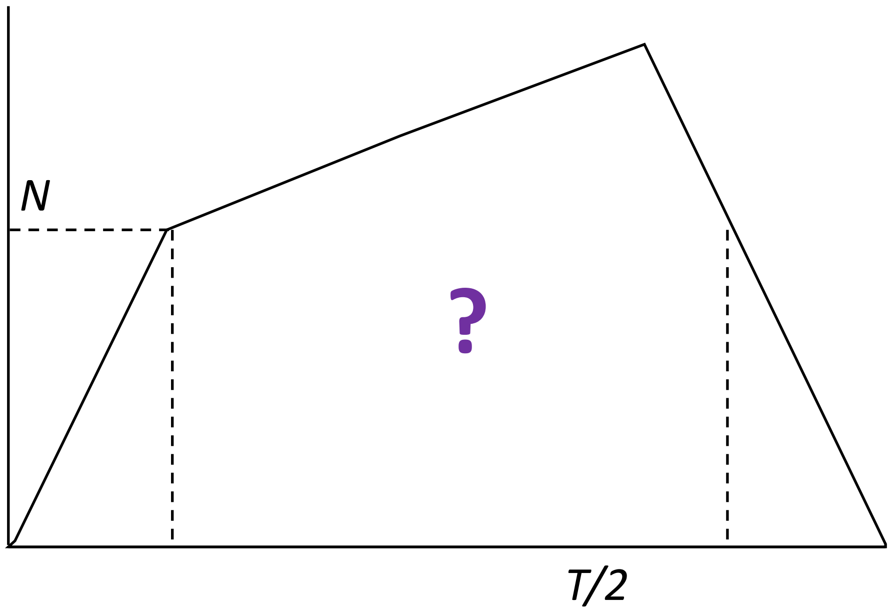
Inherently parameterized notion

(what's the most informative N for a given situation?)

Sustained Space Complexity (SSC) [ABP17]

- **SSC_N:** Space usage ≥ N is **sustained** for a period of time



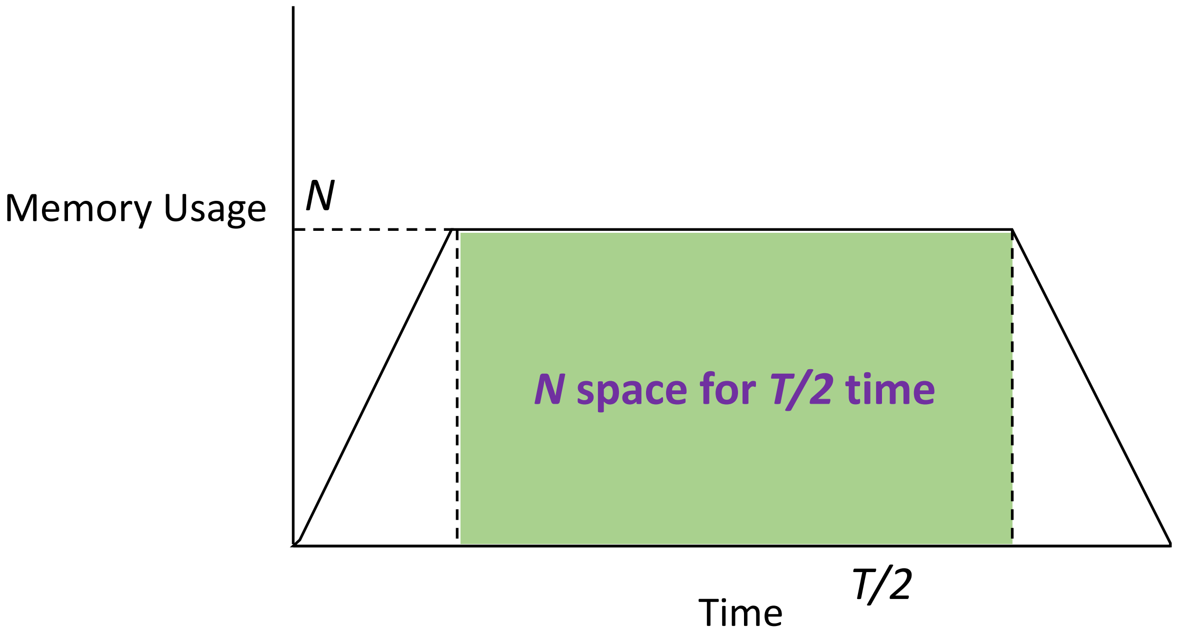


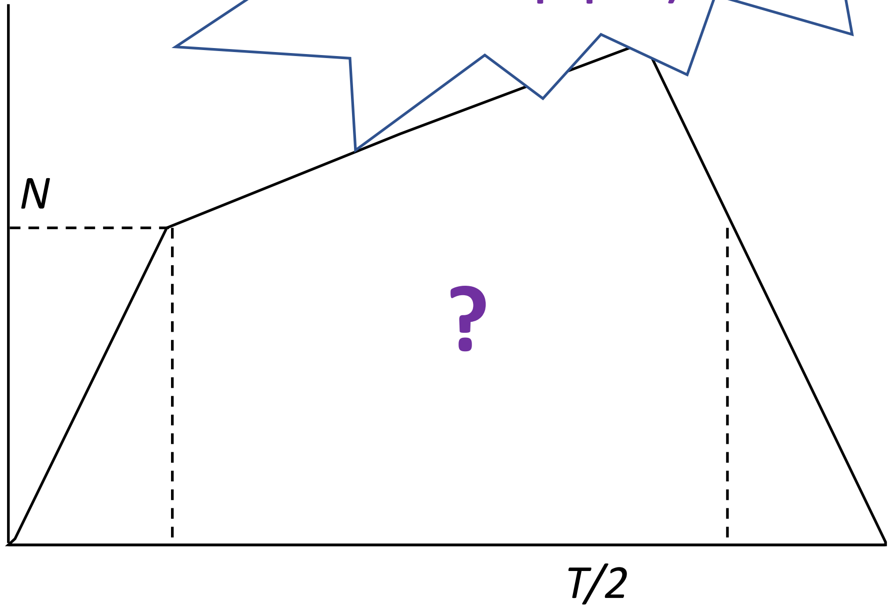
Inherently parameterized notion
(what's the most informative N for a given situation?)

Sustained Space Complexity (SSC) [ABP17]

- **SSC_N:** Space usage ≥ N is **sustained** for a period of

Sum of all memory (to some power) over total eval time: **CC-alpha (look in paper!)**





Inherently parameterized notion
(what's the most informative N for a given situation?)

Memory-Hard Functions [AS15]

- **Goal**: Protection against large-scale password-cracking attacks
  - **Resilient against**: special circuitry, parallel evaluation, amortization of cost over multiple evaluations

- **Using memory instead of time** [Percival 2009, ACPRT16]: A (data-dependent) function `scrypt` that needs a lot of ***memory*** to compute (not time)
  - Advances in memory access times incremental
  - Construction forces sequential evaluations
  - Complexity measure (**CC-complexity**)

✓ Special Circuitry
✓ Parallelism
✓ Amortization

Memory-Hard Functions (MHFs)

- **MHFs** [AS15, AB16, AB17, ABP17, RD16]: Memory is generated **dynamically** at runtime given the input to the hash function (i.e. in RAM, not on disk)
  - Existing constructions rely on combinatorial concept of ***pebbling*** **a hard-to-pebble graph** "via" random oracle queries

- **Caveats:** Size of memory requirement is **bounded by runtime needed by honest evaluator**
  - Honest evaluator needs to pebble the graph **at runtime** provided input to the function

Our Contributions

- **Static-Memory-Hard Functions (SHFs)**
  - Definition
  - Preliminaries for construction:
    graph pebbling & **parallel random oracle model** (PROM)
    - New pebbling game useful for our constructions: **black-magic pebble game**
  - Constructions
- **CC-alpha**
  (new complexity measure capturing non-linear space/time tradeoffs)
- **Optimal-CC construction** in sequential setting (up to polylog factors)

Talk Outline

- **Static-Memory-Hard Functions (SHFs)**
  - Definition
  - Preliminaries for our constructions
    - Graph pebbling & parallel random oracle model (PROM)
      - New pebbling game useful for our constructions: **black-magic pebble game**
    - Functions defined by DAGs
  - Constructions
- **CC-alpha**
  (new complexity measure capturing non-linear space/time tradeoffs)
- **Optimal-CC construction** in sequential setting (up to polylog factors)

Static-Memory-Hard Functions (SHFs)

- **Goal**: Account for **static** memory requirements
  - Static (read-only, on-disk) memory requirements can serve as deterrent to large-scale attacks, but are not captured at all by existing MHF definitions.
  - Static memory requirements may be much greater than *dynamic* memory requirements captured by MHF notions, because they could be >> runtime.
- Two-part hash function:
  - **Part 1 (setup phase):**
    One-time generation of value table (static generation of memory)
  - **Part 2 (online phase):**
    Quick online lookups, given oracle access to the output of Part 1
    (Low time complexity hash evaluation **given input**, for honest evaluator)
  - Note: Part 1 is input-independent.
- **Complementary** & incomparable to standard MHF guarantee
  - Ideally, want both! ("Dynamic-SHF" — will mention briefly later.)

Static-Memory-Hard Functions (SHFs)

- (Parallel) random oracle model
- Syntax:
  - A static-memory hash function family

$$\mathcal{H}^{\mathcal{O}} = \{h_{\kappa}^{\mathcal{O}} : \{0,1\}^{w'} \to \{0,1\}^{w}\}_{\kappa \in \mathbb{N}}$$

is described by deterministic oracle algorithms $(\mathcal{H}_1, \mathcal{H}_2)$:

Static-Memory-Hard Functions (SHFs)

- (Parallel) random oracle model
- <u>Syntax:</u>
  - A static-memory hash function family

$$\mathcal{H}^{\mathcal{O}} = \{h_\kappa^{\mathcal{O}} : \{0,1\}^{w'} \to \{0,1\}^{w}\}_{\kappa \in \mathbb{N}}$$

  is described by deterministic oracle algorithms $(\mathcal{H}_1, \mathcal{H}_2)$:

One-time setup: $\mathcal{H}_1(1^\kappa) = R$

(R is a "big string" or "lookup table" and H_1 is a succinct description of how to generate R)

Online computation: $\mathcal{H}_2^R(1^\kappa, x) = h_\kappa(x)$

(H_2 computes the correct hash output **on input x**, and has oracle access to the output of H_1)

Static-Memory-Hard Functions (SHFs)

- (Parallel) random oracle model
- <u>Syntax:</u>
  - A static-memory hash function family

$$\mathcal{H}^{\mathcal{O}} = \{h_{\kappa}^{\mathcal{O}} : \{0,1\}^{w'} \to \{0,1\}^{w}\}_{\kappa \in \mathbb{N}}$$

is described by deterministic oracle algorithms $(\mathcal{H}_1, \mathcal{H}_2)$:

One-time setup: $\mathcal{H}_1(1^{\kappa}) = R$

(R is a "big string" or "lookup table" and H_1 is a succinct description of how to generate R)

Online computation: $\mathcal{H}_2^R(1^{\kappa}, x) = h_{\kappa}(x)$

(H_2 computes the correct hash output **on input x**, and has **oracle access** to the output of H_1)

models **static** (disk) memory access

Static-Memory-Hard Functions (SHFs)

- (Parallel) random oracle model
- <u>Syntax:</u>
  - A static-memory hash function family

$$\mathcal{H}^{\mathcal{O}} = \{h_\kappa^{\mathcal{O}} : \{0,1\}^{w'} \to \{0,1\}^{w}\}_{\kappa \in \mathbb{N}}$$

is described by deterministic oracle algorithms $(\mathcal{H}_1, \mathcal{H}_2)$:

H_1 is input-independent

One-time setup: $\mathcal{H}_1(1^\kappa) = R$

(R is a "big string" or "lookup table" and H_1 is a succinct description of how to generate R)

Online computation: $\mathcal{H}_2^R(1^\kappa, x) = h_\kappa(x)$

(H_2 computes the correct hash output **on input x**, and has **oracle access** to the output of H_1)

models **static** (disk) memory access

Static-Memory-Hard Functions (SHFs)

- Adversary model: 2-part adversary $(\mathcal{A}_1, \mathcal{A}_2)$
  - A_1 outputs a "big string" R' (think of this as the adversary's static memory)
  - A_2 tries to output correct pairs (x,h(x)) given oracle access to R'
    Intuition: A_2 shouldn't be able to correctly guess more pairs than fit in R'

Static-Memory-Hard Functions (SHFs)

- Adversary model: 2-part adversary $(\mathcal{A}_1, \mathcal{A}_2)$
  - A_1 outputs a "big string" R' (think of this as the adversary's static memory)
  - A_2 tries to output correct pairs (x,h(x)) given oracle access to R'
    Intuition: A_2 shouldn't be able to correctly guess more pairs than fit in R'

- Security guarantee (informal): $(\Lambda, \Delta, \tau, q)$-hardness
  Any adversary $(\mathcal{A}_1, \mathcal{A}_2)$ that produces ≥ *q* correct input-output pairs of the hash function must ***either***
  - have A_1 produce Λ-Δ **static** memory (i.e., |R'|≥ Λ-Δ)   ***or***
  - have A_2 use Λ **dynamic** memory sustained over τ time-steps

Static-Memory-Hard Functions (SHFs)

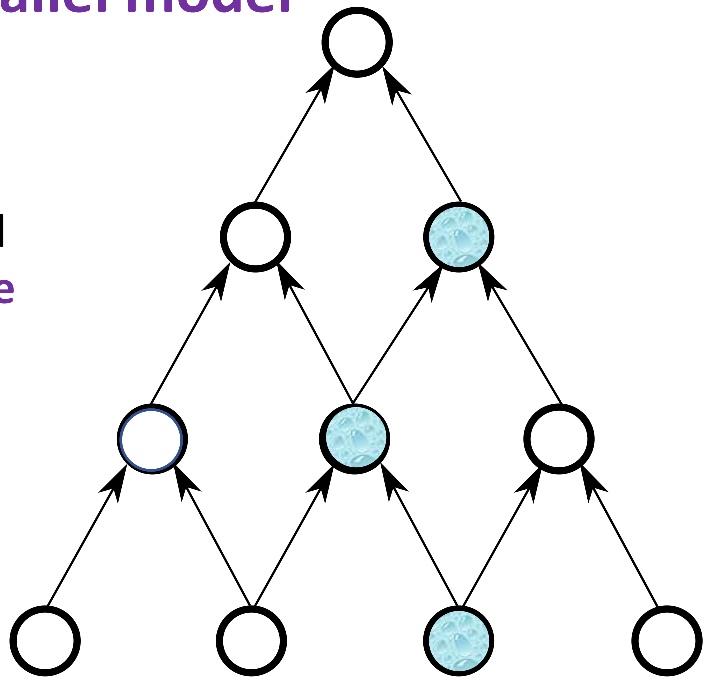
- Adversary model: 2-part adversary $(\mathcal{A}_1, \mathcal{A}_2)$
  - A_1 outputs a "big string" R' (think of this as the adversary's static memory)
  - A_2 tries to output correct pairs (x,h(x)) given oracle access to R'
    Intuition: A_2 shouldn't be able to correctly guess more pairs than fit in R'

- Security guarantee (informal): $(\Lambda, \Delta, \tau, q)$-hardness
  Any adversary $(\mathcal{A}_1, \mathcal{A}_2)$ that produces ≥ *q* correct input-output pairs of the hash function must ***either***
  - have A_1 produce Λ-Δ **static** memory (i.e., |R'|≥ Λ-Δ) ***or***
  - have A_2 use Λ **dynamic** memory sustained over τ time-steps
    ⇒ requires runtime at least Λ.
    - Recall: Λ may be gigabytes & honest evaluator only requires a few oracle accesses to their "big string", so this adversary's runtime >> honest runtime of H_2!

Graph Pebbling and PROM

- Given a DAG, computation of the hash result follows from rules of our **black-magic pebble game**
  - Computation in **parallel model**

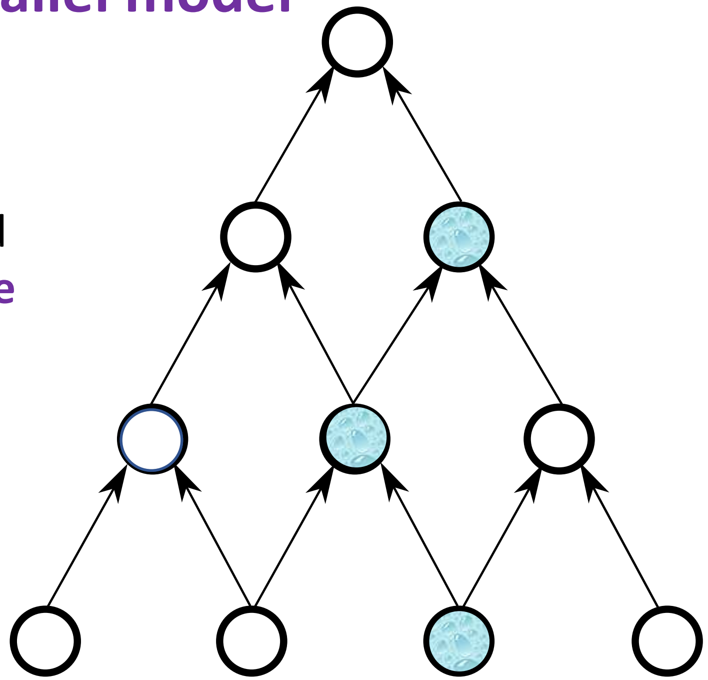
Magic pebbles placed **anywhere** at **any time**



Graph Pebbling and PROM

- Given a DAG, computation of the hash result follows from rules of our **black-magic pebble game**
  - Computation in **parallel model**

Magic pebbles placed **anywhere** at **any time**

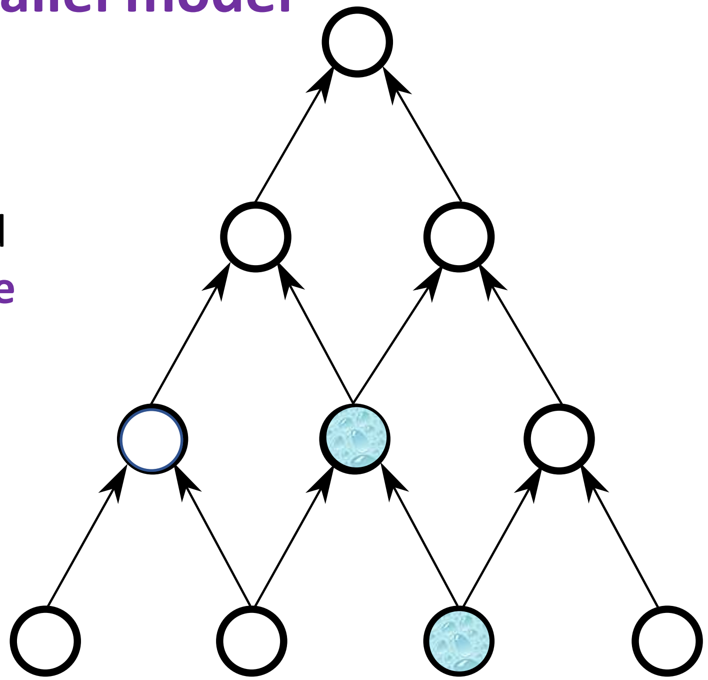


Limited number *M* of pebbles

Graph Pebbling and PROM

- Given a DAG, computation of the hash result follows from rules of our **black-magic pebble game**
  - Computation in **parallel model**

Magic pebbles placed **anywhere** at **any time**



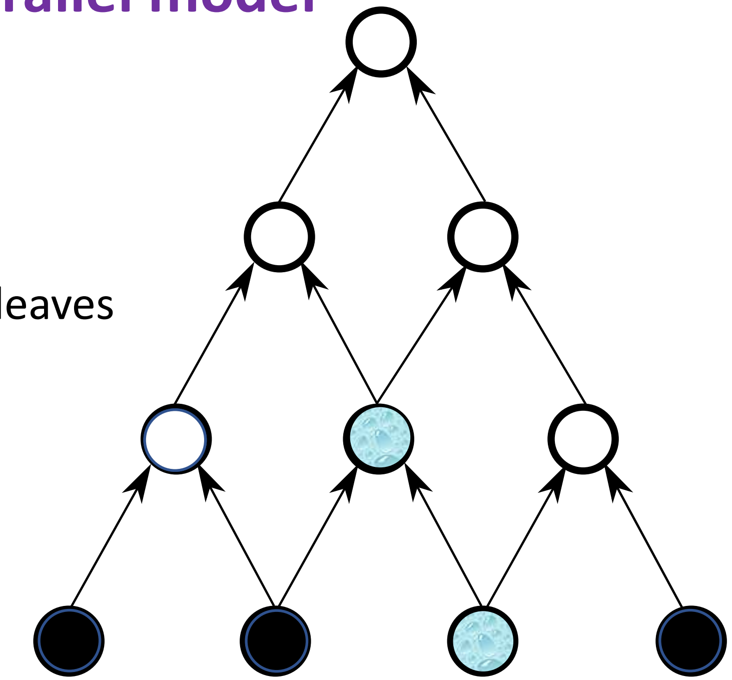
Limited number *M* of pebbles

Once you've used one and delete it, it's gone!

Graph Pebbling and PROM

- Given a DAG, computation of the hash result follows from rules of our **black-magic pebble game**
  - Computation in **parallel model**

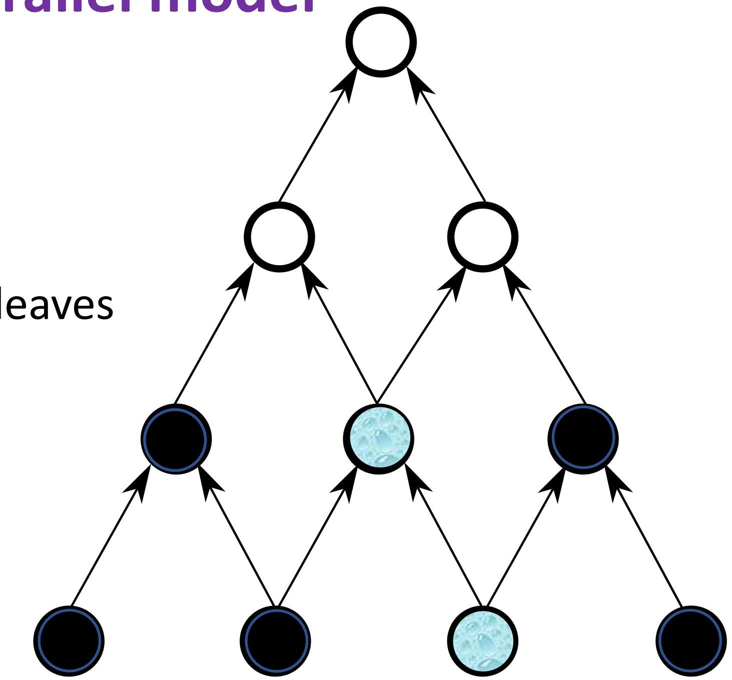
Black pebbles can be placed on leaves



Graph Pebbling and PROM

- Given a DAG, computation of the hash result follows from rules of our **black-magic pebble game**
  - Computation in **parallel model**

Black pebbles can be placed on leaves

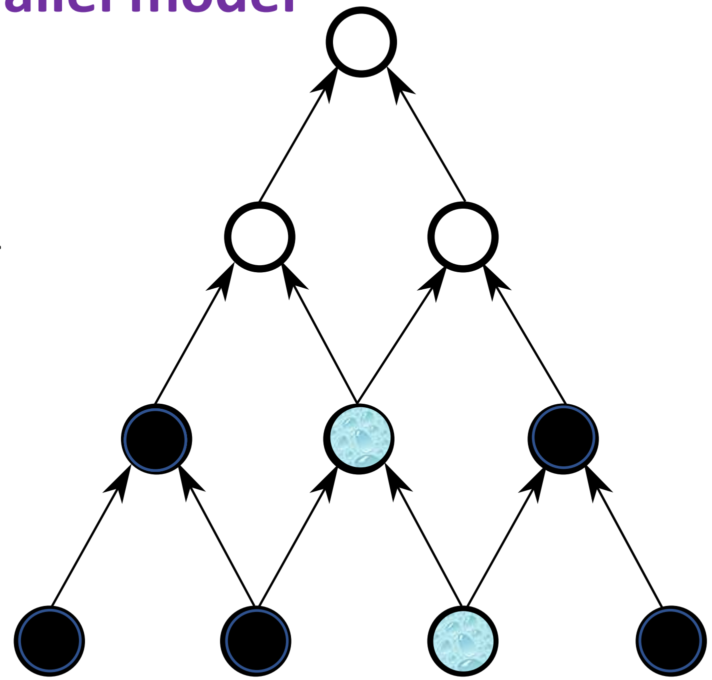


Black pebbles can be placed on nodes where all predecessors are pebbled

Graph Pebbling and PROM

- Given a DAG, computation of the hash result follows from rules of our **black-magic pebble game**
  - Computation in **parallel model**

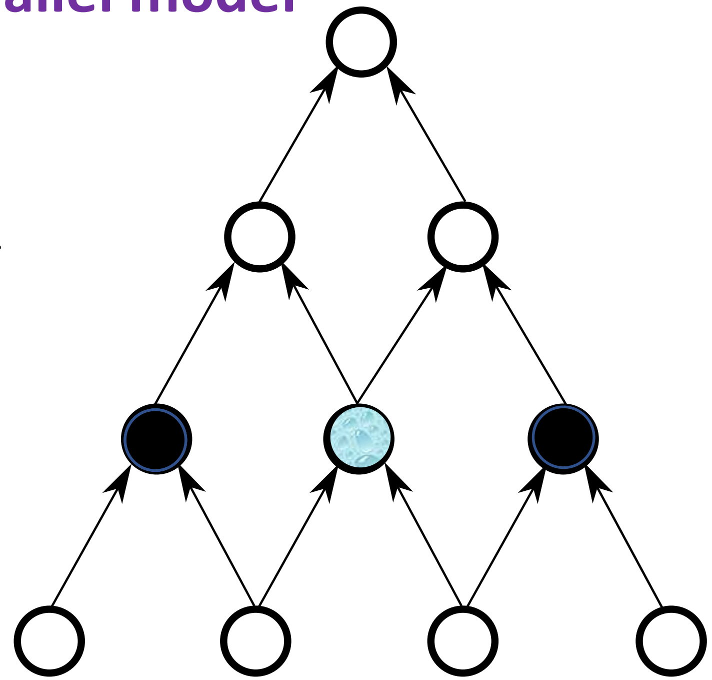
Delete any number of pebbles at any time



Graph Pebbling and PROM

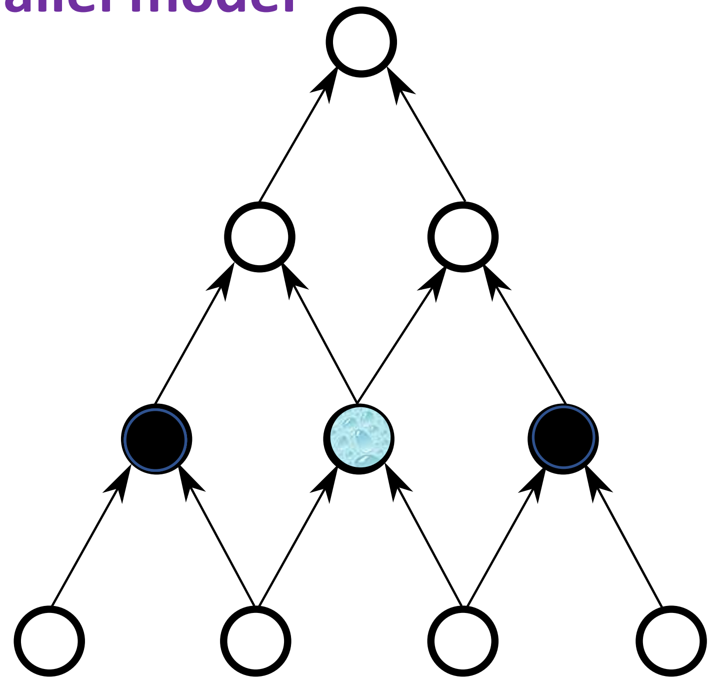
- Given a DAG, computation of the hash result follows from rules of our **black-magic pebble game**
  - Computation in **parallel model**

Delete any number of pebbles at any time



Graph Pebbling and PROM

- Given a DAG, computation of the hash result follows from rules of our **black-magic pebble game**
  - Computation in **parallel model**



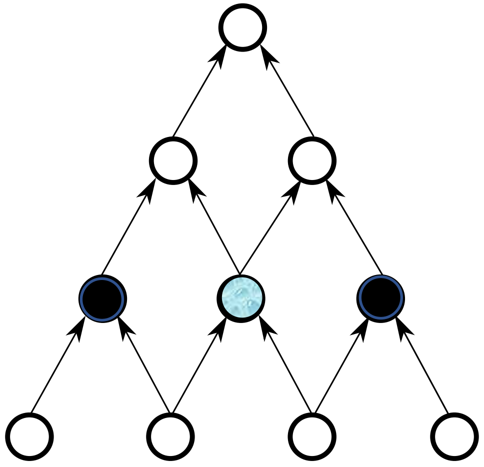
Similar to the pebble game presented in [DFKP15].

Complexity of Strategy = maximum number of pebbles on the graph at any time and total number of magic pebbles

Functions Defined by DAGs

- Function defined by DAG:
  - Magic pebbles represent stored labels
  - Label each node via recursive function where a black pebble represents computing a label:

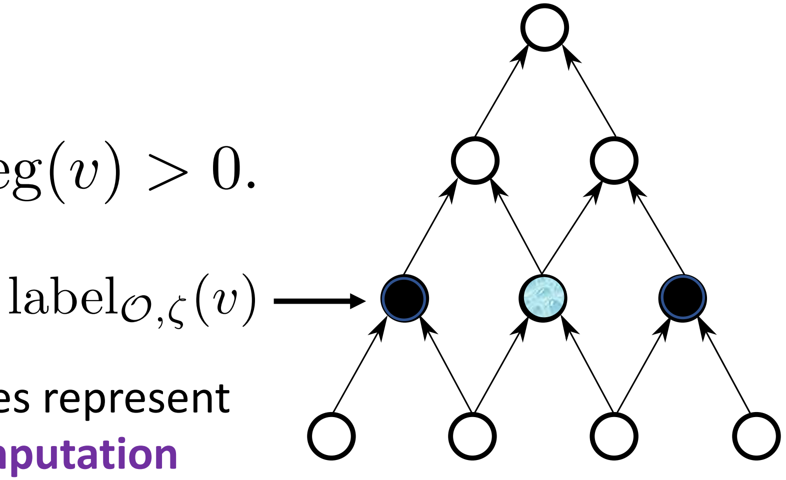
$$\mathrm{label}_{\mathcal{O},\zeta}(v) = \begin{cases} \mathcal{O}(v,\zeta) \text{ if } \mathrm{indeg}(v) = 0 \\ \mathcal{O}(v, \mathrm{label}_{\mathcal{O},\zeta}(\mathrm{pred}(v))) \text{ if } \mathrm{indeg}(v) > 0. \end{cases}$$



Functions Defined by DAGs

- Function defined by DAG:
  - Magic pebbles represent stored labels
  - Label each node via recursive function where a black pebble represents computing a label:

$$\mathrm{label}_{\mathcal{O},\zeta}(v) = \begin{cases} \mathcal{O}(v,\zeta) \text{ if } \mathrm{indeg}(v) = 0 \\ \mathcal{O}(v, \mathrm{label}_{\mathcal{O},\zeta}(\mathrm{pred}(v))) \text{ if } \mathrm{indeg}(v) > 0. \end{cases}$$

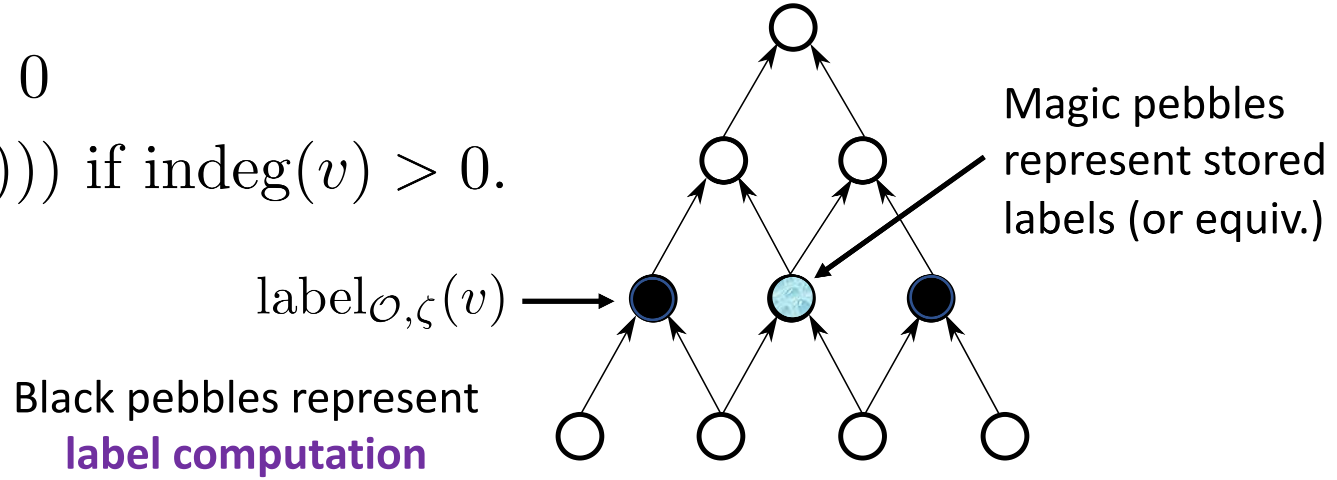


Black pebbles represent **label computation**

Functions Defined by DAGs

- Function defined by DAG:
  - Magic pebbles represent stored labels
  - Label each node via recursive function where a black pebble represents computing a label:

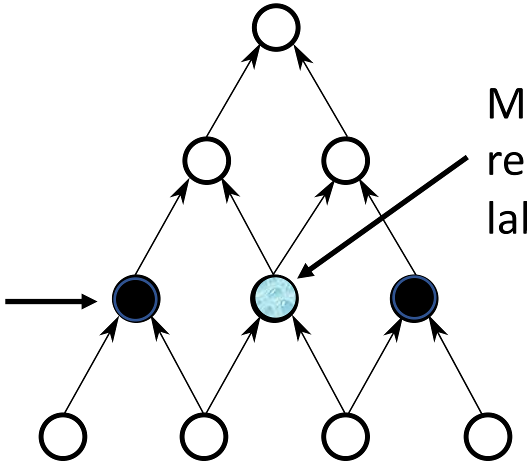
$$\mathrm{label}_{\mathcal{O},\zeta}(v) = \begin{cases} \mathcal{O}(v,\zeta) \text{ if } \mathrm{indeg}(v) = 0 \\ \mathcal{O}(v, \mathrm{label}_{\mathcal{O},\zeta}(\mathrm{pred}(v))) \text{ if } \mathrm{indeg}(v) > 0. \end{cases}$$



Functions Defined by DAGs

- Function defined by DAG:
  - Magic pebbles represent stored labels
  - Label each node via recursive function where a black pebble represents computing a label:

$$\text{label}_{\mathcal{O},\zeta}(v) = \begin{cases} \mathcal{O}(v,\zeta) \text{ if indeg}(v) = 0 \\ \mathcal{O}(v, \text{label}_{\mathcal{O},\zeta}(\text{pred}(v))) \text{ if indeg}(v) > 0. \end{cases}$$

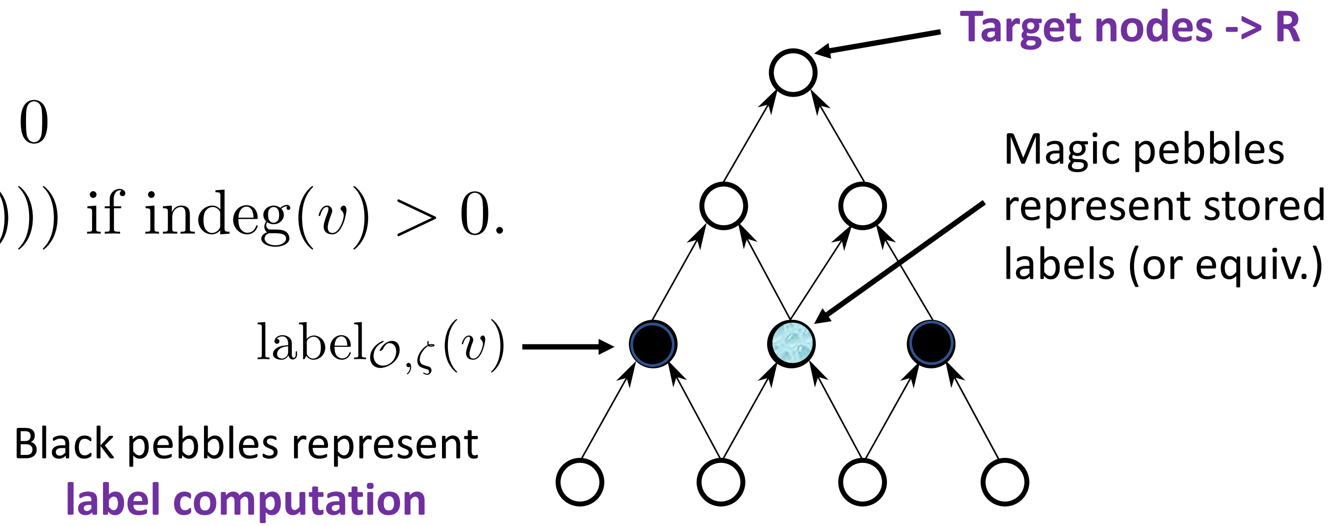


Memory complexity represented by **max number of pebbles on the graph** and **total number of magic pebbles used**

Functions Defined by DAGs

- Function defined by DAG:
  - Magic pebbles represent stored labels
  - Label each node via recursive function where a black pebble represents computing a label:

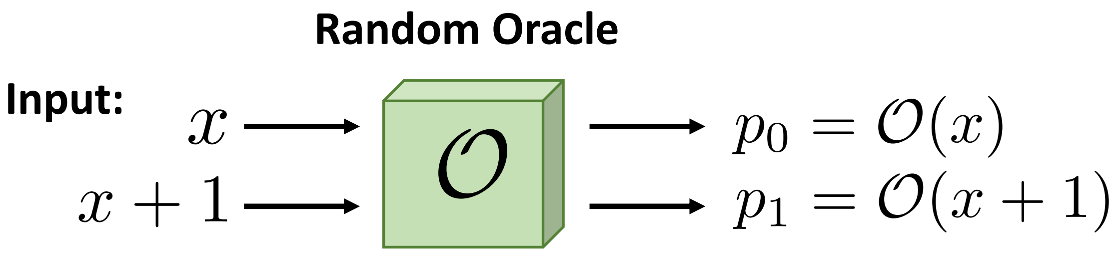
$$\text{label}_{\mathcal{O},\zeta}(v) = \begin{cases} \mathcal{O}(v,\zeta) \text{ if } \text{indeg}(v) = 0 \\ \mathcal{O}(v, \text{label}_{\mathcal{O},\zeta}(\text{pred}(v))) \text{ if } \text{indeg}(v) > 0. \end{cases}$$



Memory complexity represented by **max number of pebbles on the graph** and **total number of magic pebbles used**

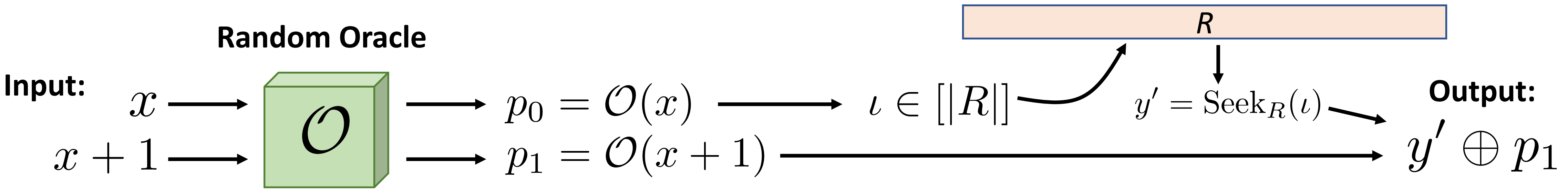
Static-Memory-Hard Function Definition

- $(\mathcal{H}_1, \mathcal{H}_2)$: $\mathcal{H}_1$ computes static table of values via black-magic pebble game
  - One-time set-up computation
- $\mathcal{H}_2$ queries for values in table provided hash function input
  - Many queries over entire period of use
- $\mathcal{H}_2$ construction:
  - On input x and given oracle access to Seek_R where R is the string output from $\mathcal{H}_1$



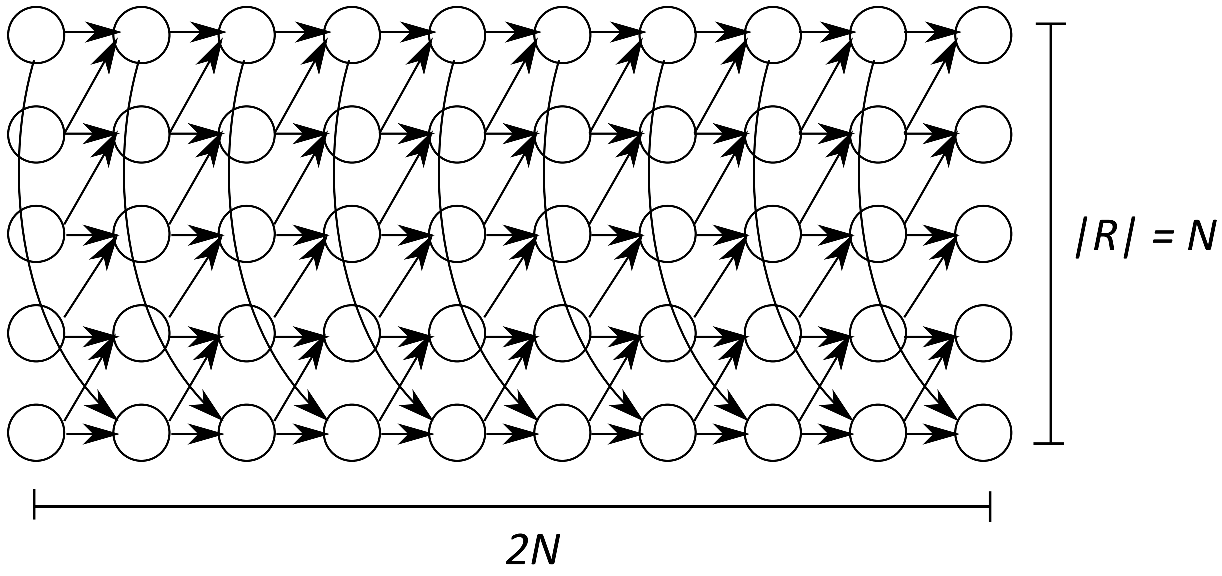
Static-Memory-Hard Function Definition

- $(\mathcal{H}_1, \mathcal{H}_2)$: $\mathcal{H}_1$ computes static table of values via black-magic pebble game
  - One-time set-up computation
- $\mathcal{H}_2$ queries for values in table provided hash function input
  - Many queries over entire period of use
- $\mathcal{H}_2$ construction:
  - On input x and given oracle access to Seek_R where R is the string output from $\mathcal{H}_1$



Candidate Constructions of $\mathcal{H}_1$

- Any graph with one target node **doesn't work**
- Need at least enough target nodes so that *R* is reasonably large
- Simple construction **cylinder graph** we implemented (n = N^2)



Our Constructions & Security Guarantees

- **Cylinder Graph SHF**: For $\Lambda \in \Theta\left(\sqrt{n}/\kappa - \xi \log(\kappa)\right)$ where n is the number of nodes in the graph, κ is the security parameter, and $\xi \in \omega(1)$, an adversary attempting to query $Q = \omega(S)$ **non-trivially more hashes than she stored** must **incur at least Λ dynamic memory** usage **for at least $\Theta(\sqrt{n})$ steps**.

Our Constructions & Security Guarantees

- **<u>Cylinder Graph SHF</u>**: For $\Lambda \in \Theta\left(\sqrt{n}/\kappa - \xi\log(\kappa)\right)$ where n is the number of nodes in the graph, κ is the security parameter, and $\xi \in \omega(1)$, an adversary attempting to query $Q = \omega(S)$ **non-trivially more hashes than she stored** must **incur at least Λ dynamic memory** usage **for at least $\Theta(\sqrt{n})$ steps**.

Best possible for layered graph constructions!

Our Constructions & Security Guarantees

- **Cylinder Graph SHF**: For $\Lambda \in \Theta\left(\sqrt{n}/\kappa - \xi\log(\kappa)\right)$ where n is the number of nodes in the graph, κ is the security parameter, and $\xi \in \omega(1)$, an adversary attempting to query $Q = \omega(S)$ **non-trivially more hashes than she stored** must **incur at least Λ dynamic memory** usage **for at least $\Theta(\sqrt{n})$ steps**.

Best possible for layered graph constructions!

- **"Shortcut-Free" SHF**: For $\Lambda \in \Theta\left(\sqrt{n}/\kappa - \xi\log(\kappa)\right)$ where definitions as above, an adversary attempting to query non-trivially more hashes than she stored must incur at least Λ dynamic memory usage for **at least $\Theta(n)$ steps**.

Dynamic-SHFs: Best of Both Worlds

- Combine with MHFs [AS15, AB16, AB17, ABP17, RD16] from previous works via simple concatenation scheme
- **Benefits:**
  - Inherits both the properties of SHFs and MHFs
  - **Dynamic memory requirement** upon input from MHF
  - Adversaries incur **large static memory requirement** from SHF

Open Questions

- SHFs:
  - Can we improve the security guarantee to have a smaller loss from the security parameter?
  - Can we have better space guarantees for SHFs in general graphs?
- CC-alpha (from paper)
  - Does there exist an example where CC-alpha differs between linear and quadratic trade-off?
- Optimal CC construction (from paper)
  - Can our optimal sequential construction be modified to obtain optimal bounds in the parallel case?

Talk Outline

- **Static-Memory-Hard Functions**
  - Definition
  - Preliminaries for our constructions
    - Graph pebbling & parallel random oracle model (PROM)
      - New pebbling game useful for our constructions: **black-magic pebble game**
    - Functions defined by DAGs
  - Constructions
- **CC-alpha**
  (new complexity measure capturing non-linear space/time tradeoffs)
- **Optimal-CC construction** in sequential setting (up to polylog factors)

CC^{α}

- **Goal:** Another complexity measure for **non-linear space-time cost tradeoffs**
- Based on the cumulative complexity measure [AS15]
- **Definition:** Given a graph $G = (V, E)$, the $CC^{\alpha}(G)$ is $\min_{\mathcal{P} \in \mathbb{P}} \left(\sum_{P_i \in \mathcal{P}} |P_i|^{\alpha} \right)$

CC^{α}

- **Goal:** Another complexity measure for **non-linear space-time cost tradeoffs**
- Based on the cumulative complexity measure [AS15]
- **Definition:** Given a graph $G = (V, E)$, the $CC^{\alpha}(G)$ is $\min_{\mathcal{P} \in \mathbb{P}} \left(\sum_{P_i \in \mathcal{P}} |P_i|^{\alpha} \right)$

Main Theorem: There exist graphs for which an adversary facing a *linear space-time* trade-off would **employ a different pebbling strategy** from one facing a *cubic trade-off*.

Talk Outline

- **Static-Memory-Hard Functions**
  - Definition
  - Preliminaries for our constructions
    - Graph pebbling & parallel random oracle model (PROM)
      - New pebbling game useful for our constructions: **black-magic pebble game**
    - Functions defined by DAGs
  - Constructions
- **CC-alpha**
  (new complexity measure capturing non-linear space/time tradeoffs)
- **Optimal-CC construction** in sequential setting (up to polylog factors)

Optimal CC Construction for Sequential Case

- Asymptotically tight sequential lower bound for $\alpha = 1$
- Using stacked superconcentrator construction of [LT82] (with slight modification)
  - Gives CC of $\Theta\left(\frac{n^2 \log\log n}{\log n}\right)$
  - Meets upper bound [AB16, ABP17] up to polylog factors