CS 594 – Advanced Crypto (Spring 2026)

Alex Block

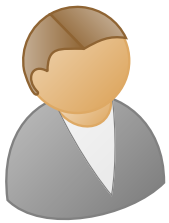
Lecture 21

April 08, 2026

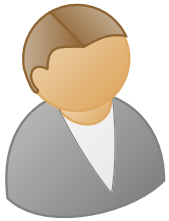
Memory-hard Functions

Motivation: Password Storage

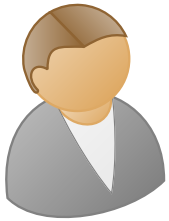
Motivation: Password Storage



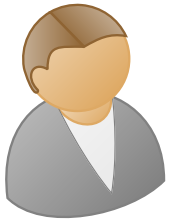
Motivation: Password Storage



Motivation: Password Storage

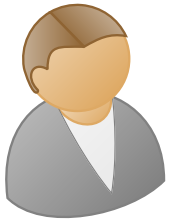


Motivation: Password Storage



(arblock, 123456)

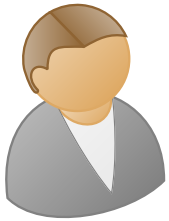
Motivation: Password Storage



(arblock, 123456)

Username	Salt	Hash
arblock	89d9 7803 4a3f6	946f f6e7 599f 342a 0a9b 9ae1 e5d3 dece 9091 52d6

Motivation: Password Storage

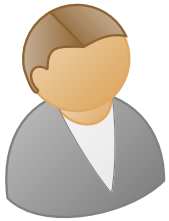


(arblock, 123456)

Username	Salt	Hash
arblock	89d9 7803 4a3f6	946f f6e7 599f 342a 0a9b 9ae1 e5d3 dece 9091 52d6

SHA1(12345689d978034a3f6)
= 946ff6e7599f342a0a9b9ae1e5d3dece909152d6

Motivation: Password Storage

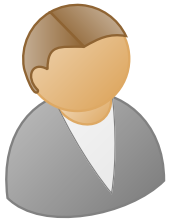


(arblock, 123456)

Username	Salt	Hash
arblock	89d9 7803 4a3f6	946f f6e7 599f 342a 0a9b 9ae1 e5d3 dece 9091 52d6

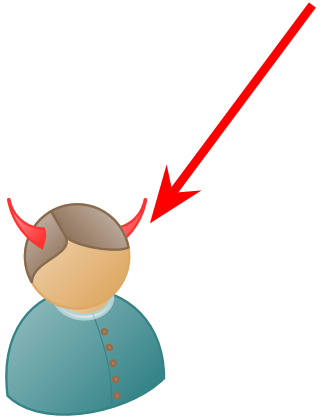
SHA1(12345689d978034a3f6)
= 946ff6e7599f342a0a9b9ae1e5d3dece909152d6

Motivation: Password Storage



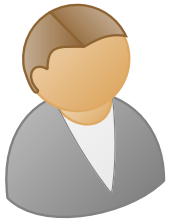
(arblock, 123456)

Username	Salt	Hash
arblock	89d9 7803 4a3f6	946f f6e7 599f 342a 0a9b 9ae1 e5d3 dece 9091 52d6



SHA1(12345689d978034a3f6)
= 946ff6e7599f342a0a9b9ae1e5d3dece909152d6

Motivation: Password Storage



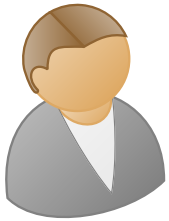
(arblock, 123456)

Username	Salt	Hash
arblock	89d9 7803 4a3f6	946f f6e7 599f 342a 0a9b 9ae1 e5d3 dece 9091 52d6

SHA1(12345689d978034a3f6)
= 946ff6e7599f342a0a9b9ae1e5d3dece909152d6

+

Motivation: Password Storage



(arblock, 123456)

Username	Salt	Hash
arblock	89d9 7803 4a3f6	946f f6e7 599f 342a 0a9b 9ae1 e5d3 dece 9091 52d6

SHA1(12345689d978034a3f6)
= 946ff6e7599f342a0a9b9ae1e5d3dece909152d6

+

$H(H(H(\cdots H(\text{uname}\|\text{salt})\cdots$

$x \xrightarrow{H(x)} \underbrace{\circ \xrightarrow{H(\cdot)} \circ \xrightarrow{H(\cdot)} \cdots \circ}_{32 \text{ times}}$

Motivation: Offline Attacks are a Common Problem

Motivation: Offline Attacks are a Common Problem

- Password breaches at major companies have affected **billions** of user accounts.

Motivation: Offline Attacks are a Common Problem

- Password breaches at major companies have affected **billions** of user accounts.

Motivation: Offline Attacks are a Common Problem

- Password breaches at major companies have affected **billions** of user accounts.

https://haveibeenpwned.com/

Goal: Moderately Expensive Hash Functions

Goal: Moderately Expensive Hash Functions

Can we design a *moderately expensive* hash function?

Goal: Moderately Expensive Hash Functions

Can we design a *moderately expensive* hash function?

- *Fast* to compute on regular hardware (e.g., desktop, laptop, or phone); and

Goal: Moderately Expensive Hash Functions

Can we design a *moderately expensive* hash function?

- *Fast* to compute on regular hardware (e.g., desktop, laptop, or phone); and
- *Expensive* to compute on an ASIC?

Goal: Moderately Expensive Hash Functions

Can we design a *moderately expensive* hash function?

- *Fast* to compute on regular hardware (e.g., desktop, laptop, or phone); and
- *Expensive* to compute on an ASIC?
  - *Application-specific Integrated circuits.*

Goal: Moderately Expensive Hash Functions

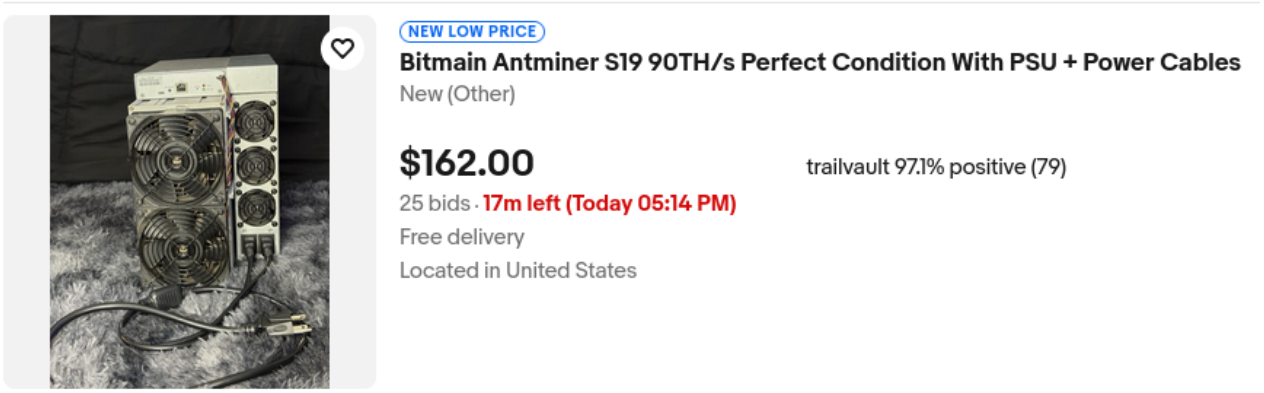
Can we design a *moderately expensive* hash function?

- *Fast* to compute on regular hardware (e.g., desktop, laptop, or phone); and
- *Expensive* to compute on an ASIC?
  - *Application-specific Integrated circuits.*
  - Became more widely available for computing hashes like SHA-3 or SHA-256 because of Bitcoin and friends.

Goal: Moderately Expensive Hash Functions

Can we design a *moderately expensive* hash function?

- *Fast* to compute on regular hardware (e.g., desktop, laptop, or phone); and
- *Expensive* to compute on an ASIC?
  - *Application-specific Integrated circuits.*
  - Became more widely available for computing hashes like SHA-3 or SHA-256 because of Bitcoin and friends.



Memory-hard Functions

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Space cost

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Space cost

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Space cost

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Need 2× the time?
Leave computer on!

Space cost

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Need 2× the time?
Leave computer on!

Space cost

Need 2× the space?
Download more RAM!

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Need 2× the time?
Leave computer on!

Space cost

Need 2× the space?
Download more RAM!

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Need 2× the time?
Leave computer on!

Space cost

Need 2× the space?
Download more RAM!

Goal

Memory-hard Functions

- Intuition: force computational cost to be dominated by *memory*.
  - CPU time is *cheap*; adding more RAM is *expensive*.

Time cost

Need 2× the time?
Leave computer on!

Space cost

Need 2× the space?
Download more RAM!

Goal

Force attacker to use *large amounts of memory* for duration of computation, even on customized hardware.

Memory-hard Functions: Two Flavors

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Data-dependent MHF (dMHF)

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Data-dependent MHF (dMHF)

Data-independent MHF (iMHF)

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Data-dependent MHF (dMHF)

- The structure and output of a dMHF f *depends* on the input x being evaluated.

Data-independent MHF (iMHF)

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Data-dependent MHF (dMHF)

- The structure and output of a dMHF f *depends* on the input x being evaluated.

Data-independent MHF (iMHF)

- The structure and output of a iMHF f is *independent* of *any* input x being evaluated.

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Data-dependent MHF (dMHF)

- The structure and output of a dMHF f *depends* on the input x being evaluated.

Pros

- Best "memory-hard" guarantees

Data-independent MHF (iMHF)

- The structure and output of a iMHF f is *independent* of *any* input x being evaluated.

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Data-dependent MHF (dMHF)

- The structure and output of a dMHF f *depends* on the input x being evaluated.

Pros

- Best "memory-hard" guarantees

Cons

- Susceptible to side-channel attacks!

Data-independent MHF (iMHF)

- The structure and output of a iMHF f is *independent* of *any* input x being evaluated.

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Data-dependent MHF (dMHF)

- The structure and output of a dMHF f *depends* on the input x being evaluated.

Pros

- Best "memory-hard" guarantees

Cons

- Susceptible to side-channel attacks!

Data-independent MHF (iMHF)

- The structure and output of a iMHF f is *independent* of *any* input x being evaluated.

Pros

- Highly resistant to side-channel attacks

Memory-hard Functions: Two Flavors

- MHFs come in two flavors:
  - data-*dependent*; and
  - data-*independent*

Data-dependent MHF (dMHF)

- The structure and output of a dMHF f *depends* on the input x being evaluated.

Pros

- Best "memory-hard" guarantees

Cons

- Susceptible to side-channel attacks!

Data-independent MHF (iMHF)

- The structure and output of a iMHF f is *independent* of *any* input x being evaluated.

Pros

- Highly resistant to side-channel attacks

Cons

- Provably worse "memory-hardness"

MHFs from Directed Graphs

MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.

↳ "Fancy" hash-chaining

MHFs from Directed Graphs

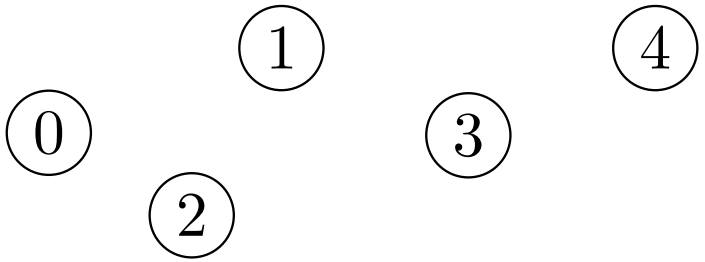
- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.

MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.

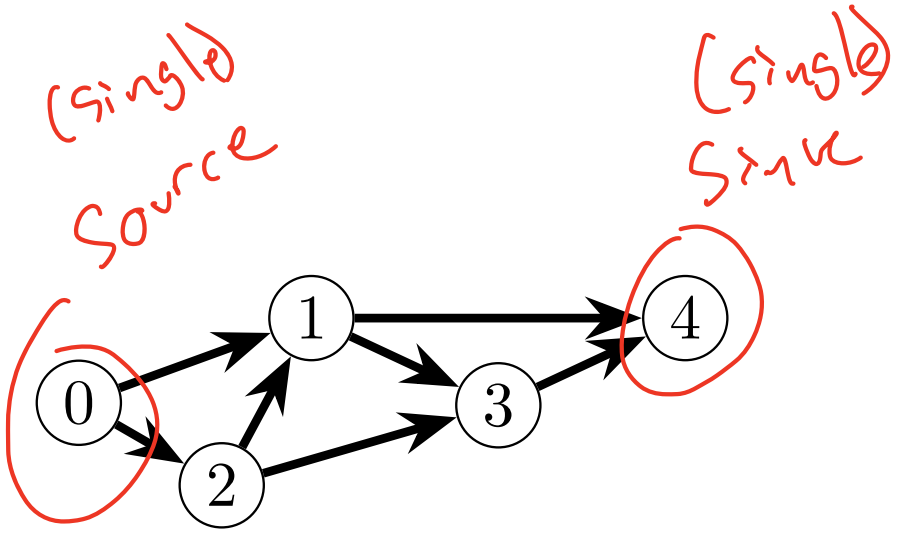
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



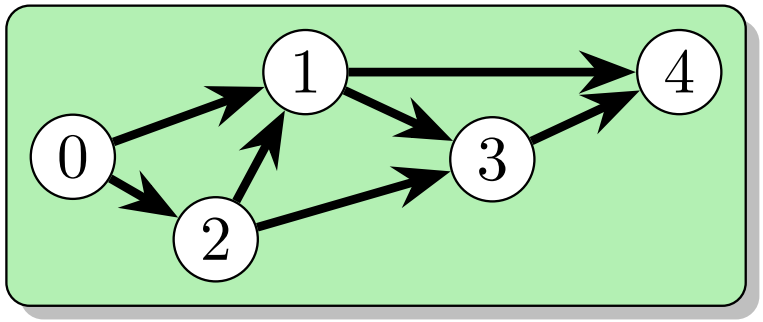
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



MHFs from Directed Graphs

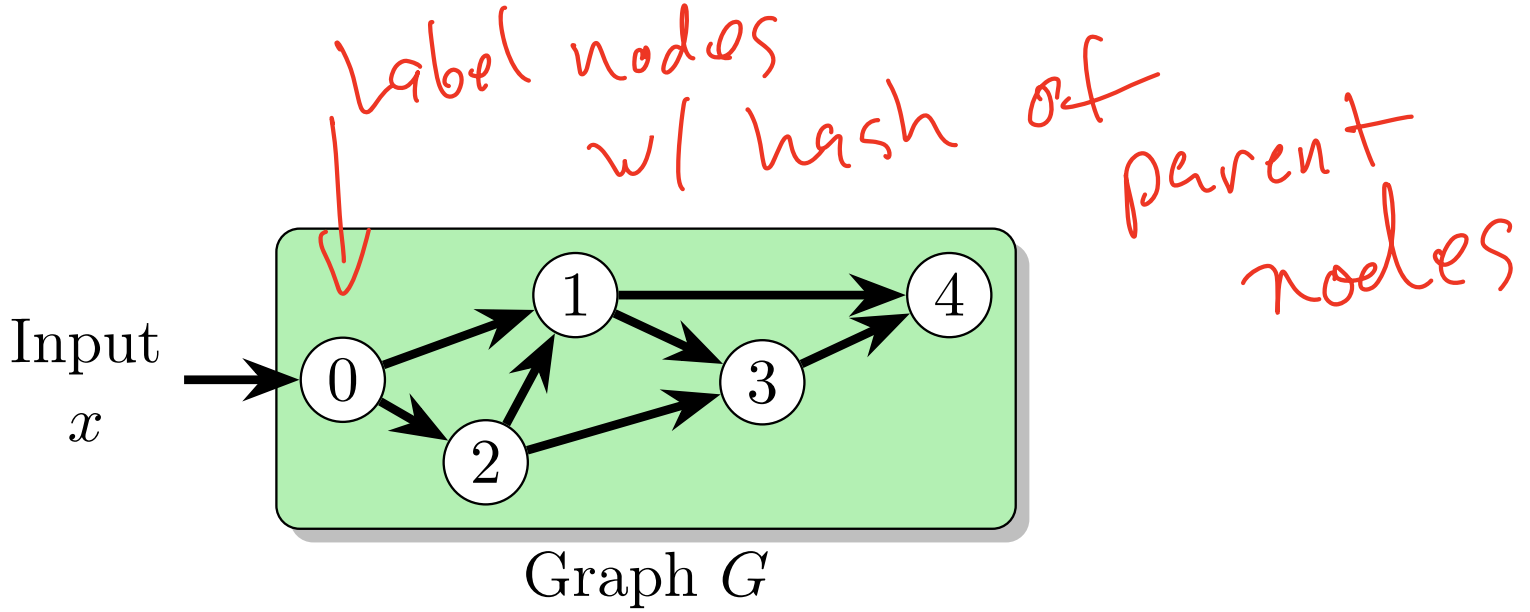
- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



Graph G

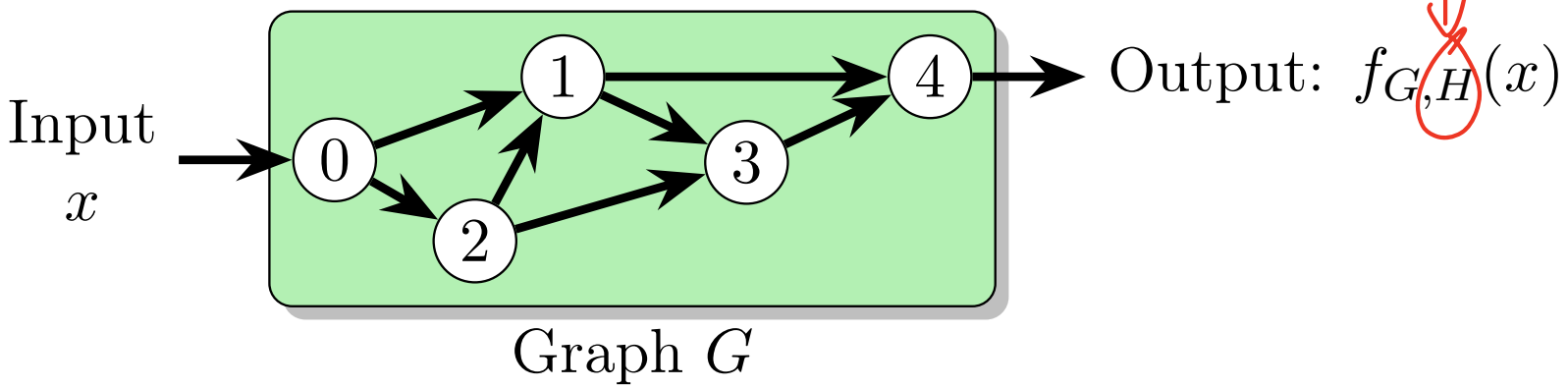
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



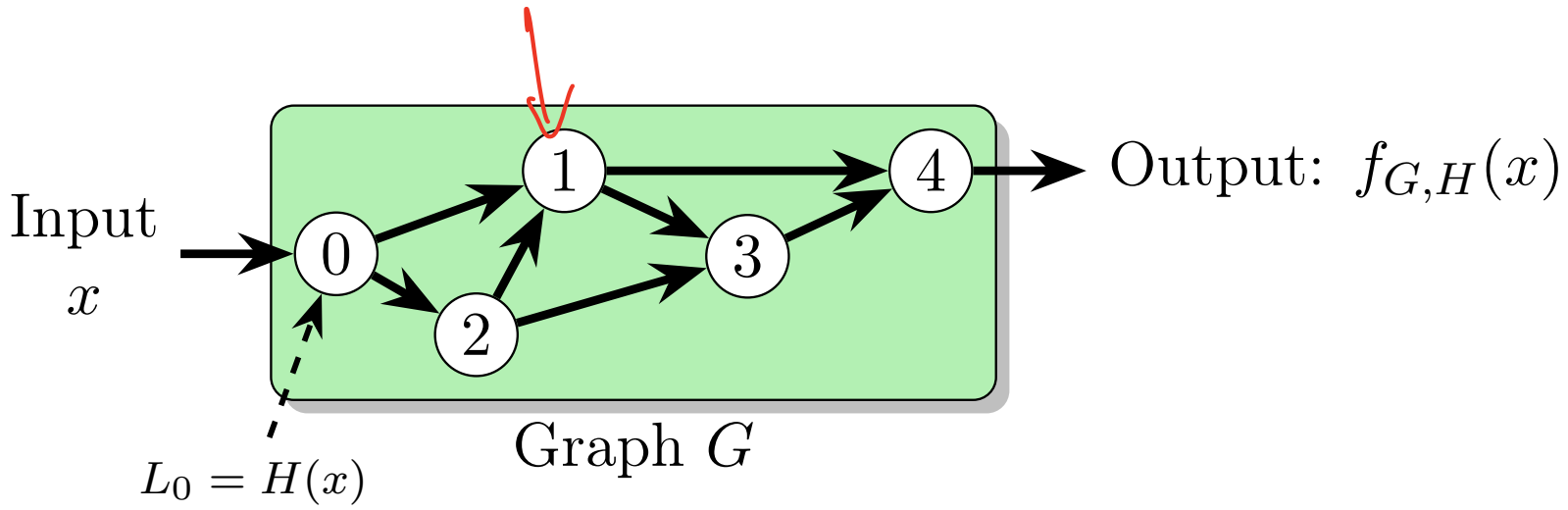
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



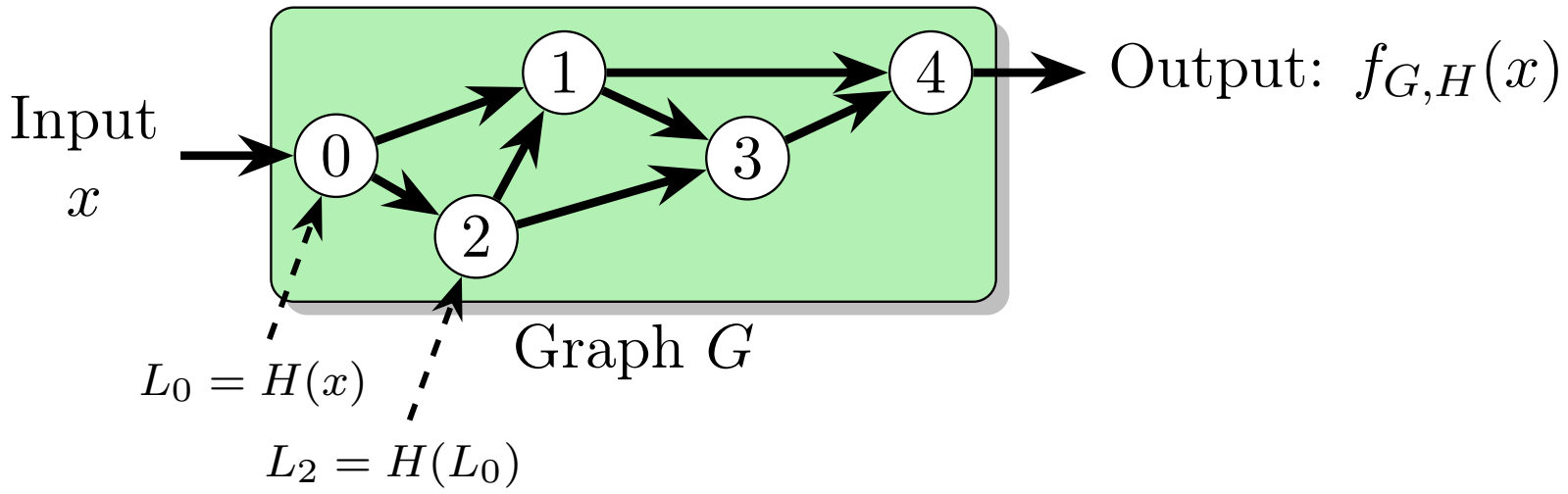
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



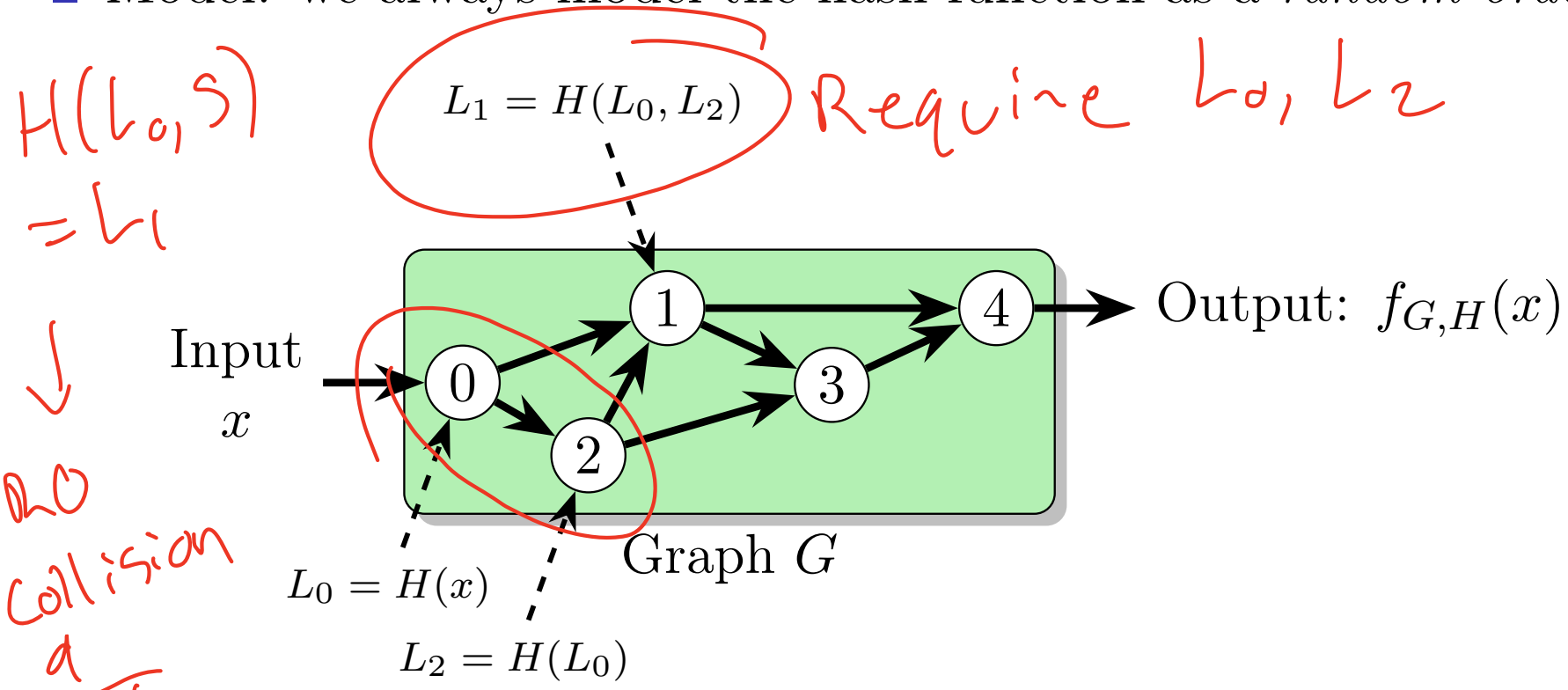
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



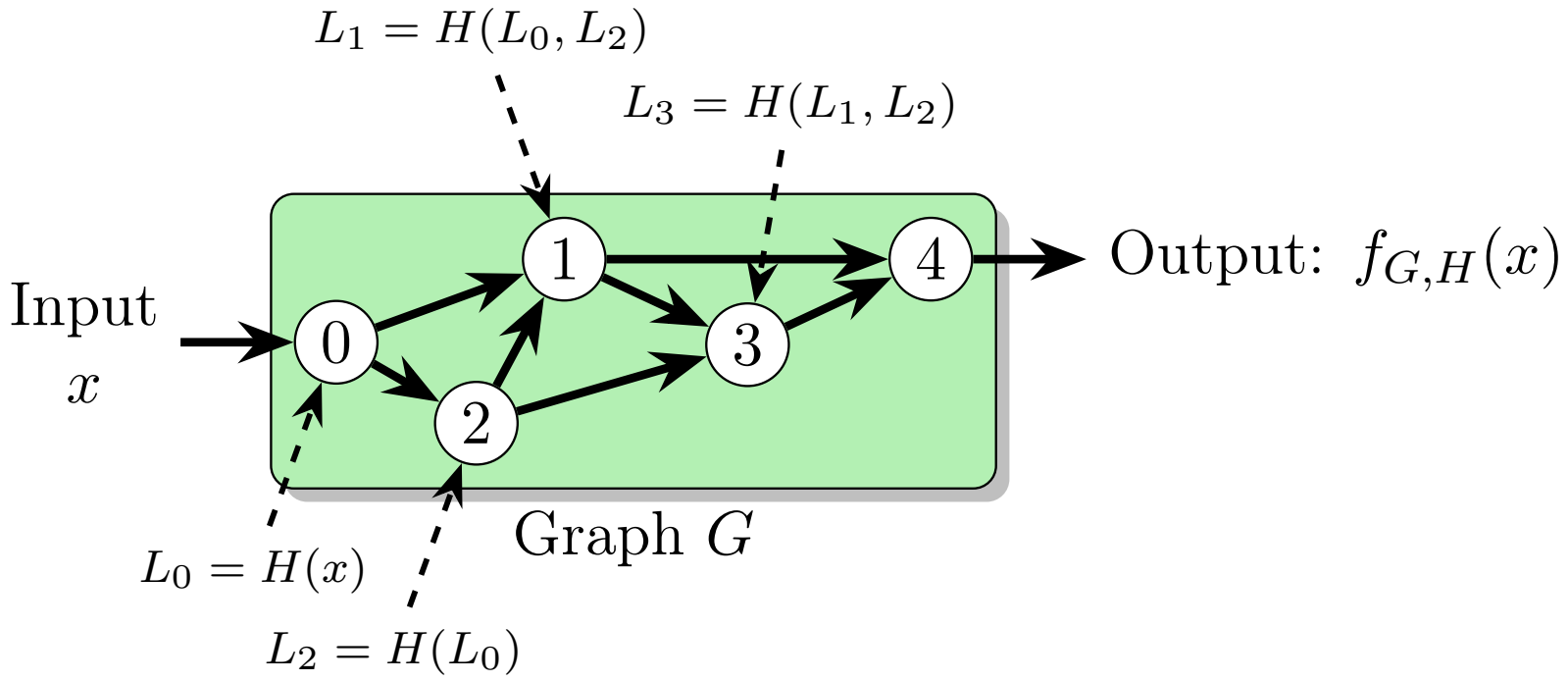
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



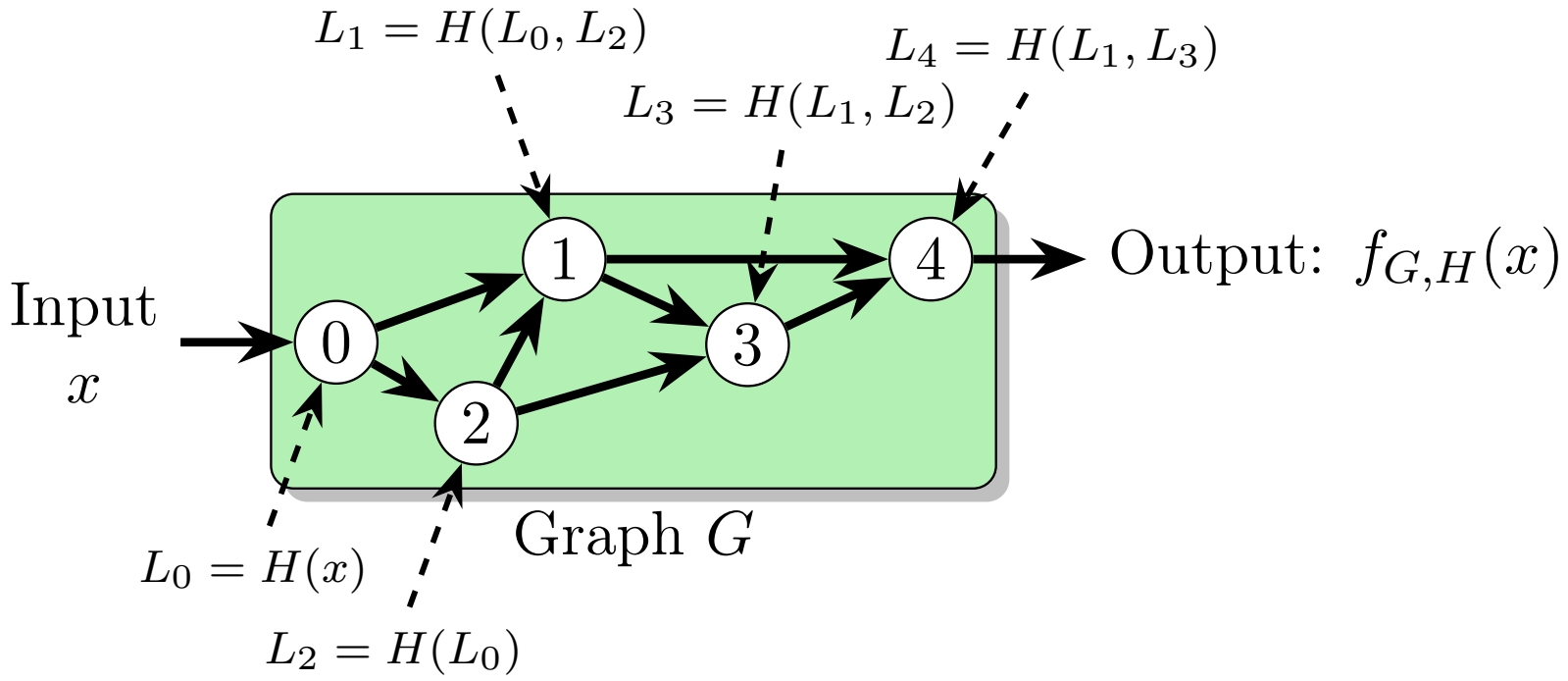
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



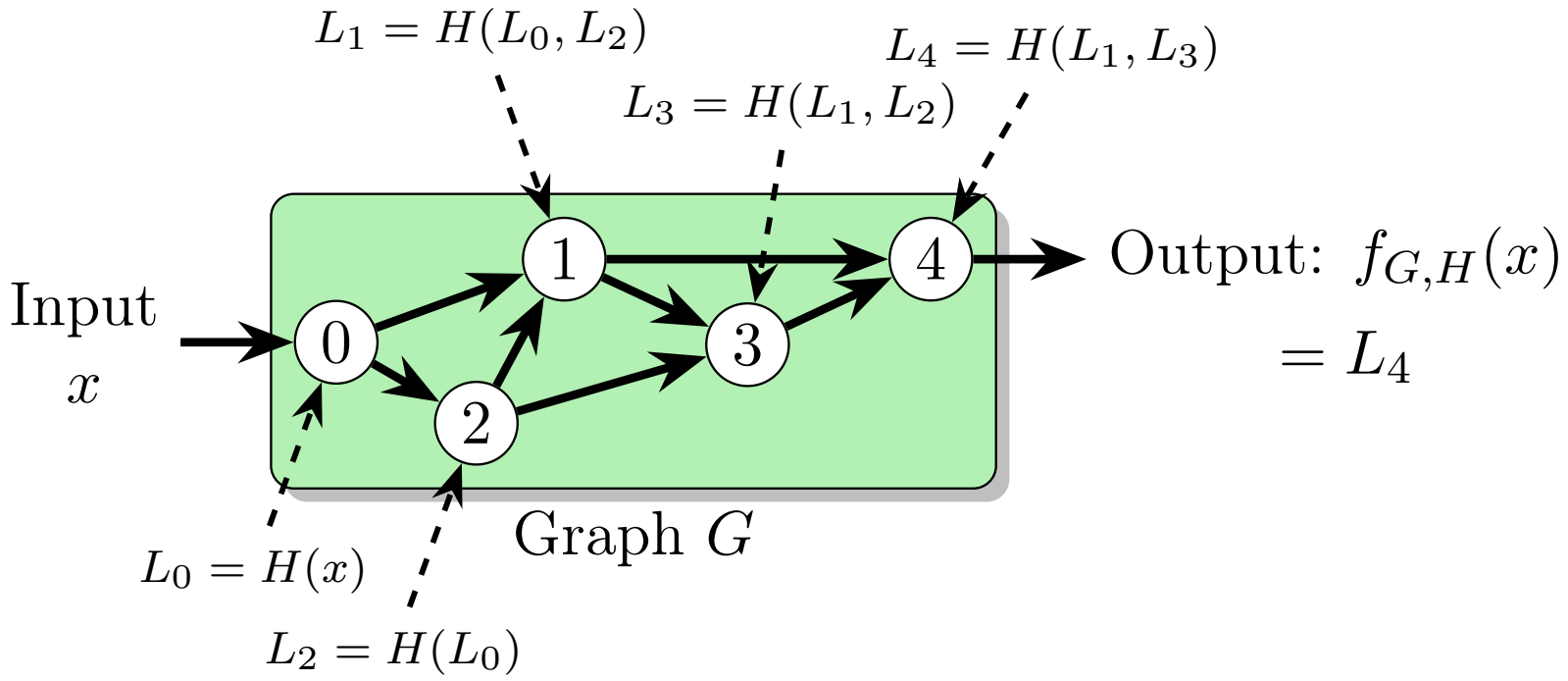
MHFs from Directed Graphs

- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.

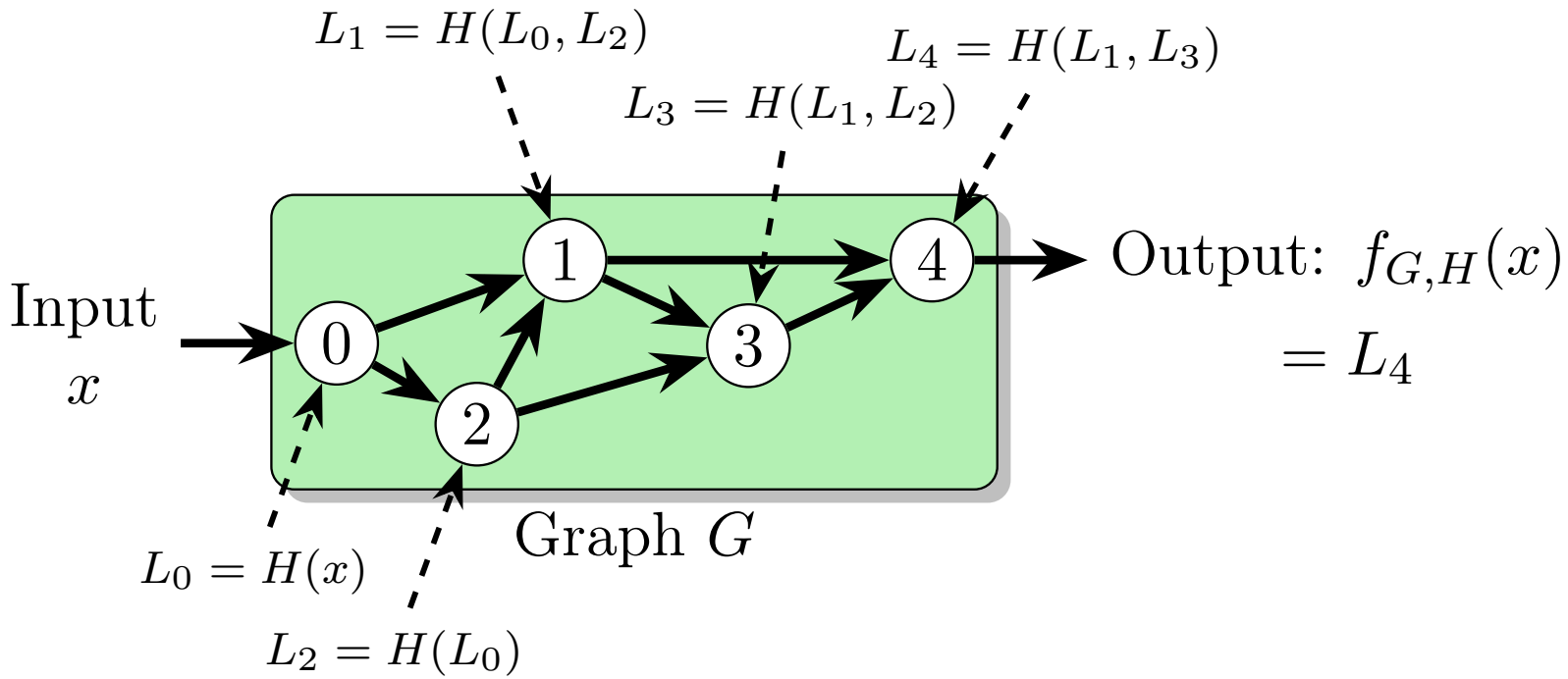


MHFs from Directed Graphs

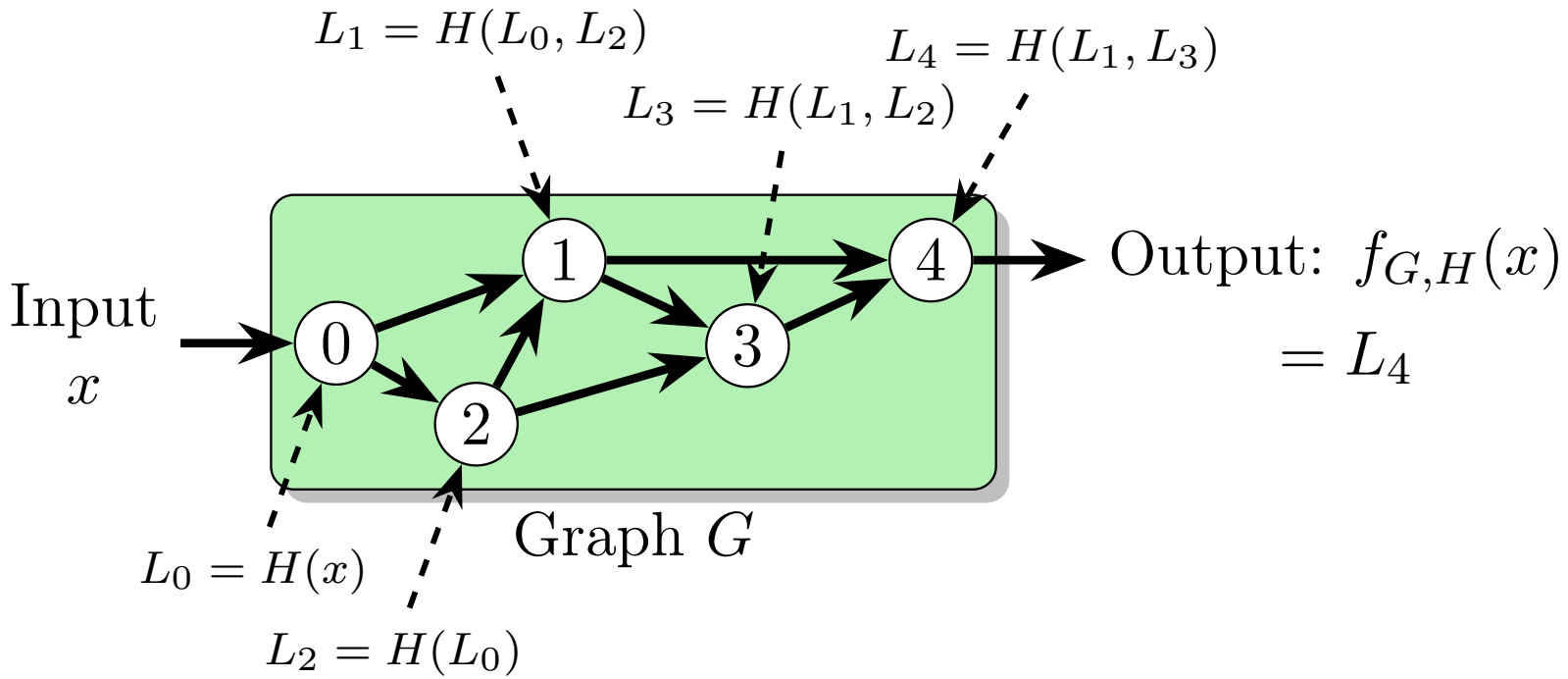
- Key idea of basically all MHFs: force hash to be computed as labels of a *directed acyclic graph*.
- Idea: use DAG to encode data dependencies to force large amounts of memory to be stored.
- Model: we always model the hash function as a *random oracle*.



MHFs from Directed Graphs

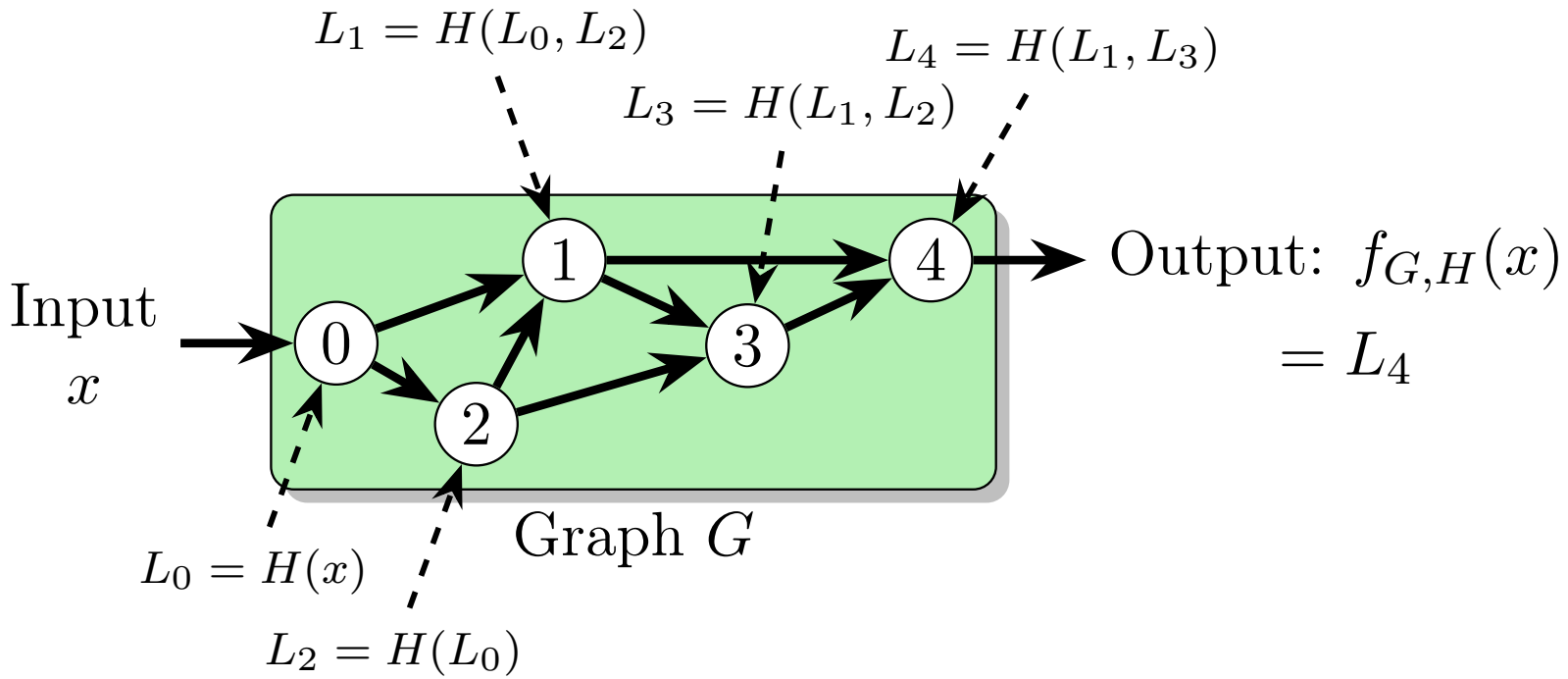


MHFs from Directed Graphs



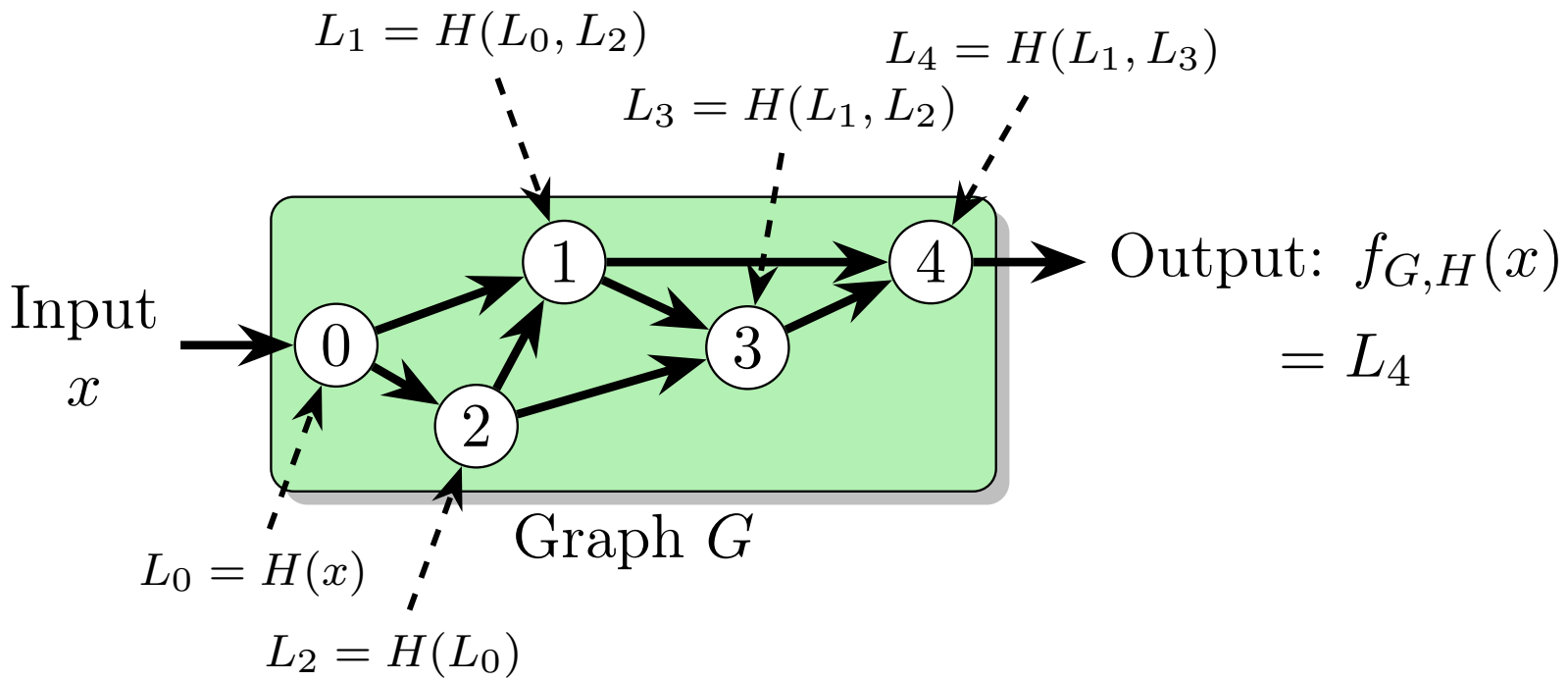
- $H\colon \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.

MHFs from Directed Graphs



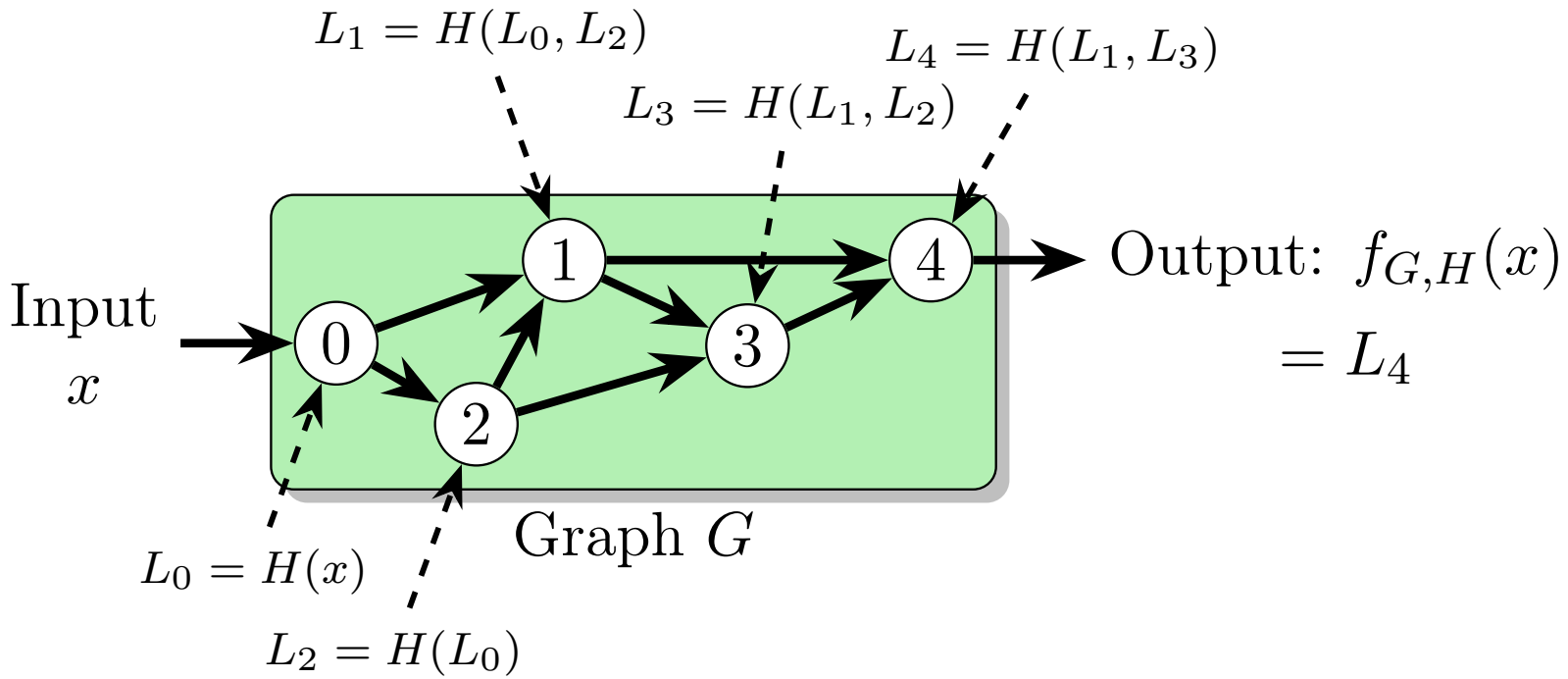
- $H\colon \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.
- DAG G encodes data dependencies.

MHFs from Directed Graphs



- $H : \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.
- DAG G encodes data dependencies.
  - Maximum in-degree $\delta = O(1)$.

MHFs from Directed Graphs



- $H\colon \{0,1\}^* \to \{0,1\}^\lambda$ is a random oracle.
- DAG G encodes data dependencies.
  - Maximum in-degree $\delta = O(1)$.
  - $|V| = N = 2^n$ nodes/vertices (not in this example).

dMHFs vs. iMHFs

dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.

dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.

dMHFs vs. iMHFs

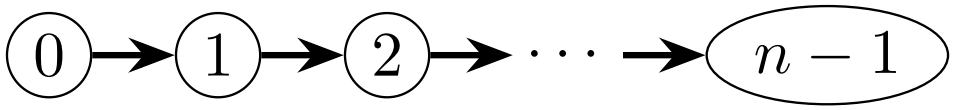
- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.

dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`

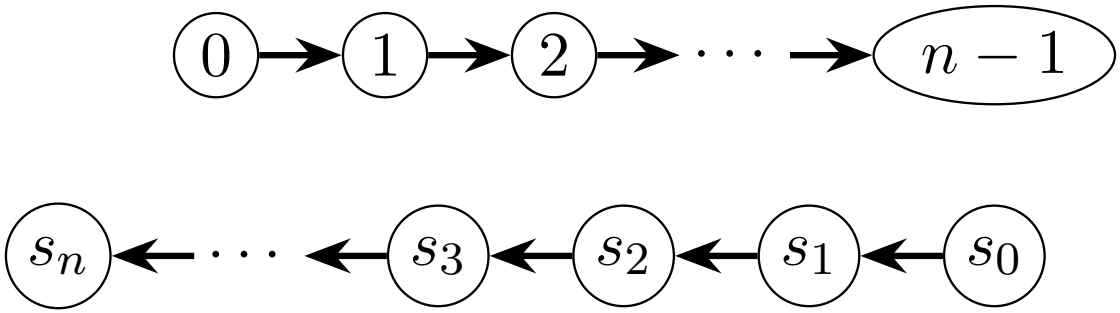
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



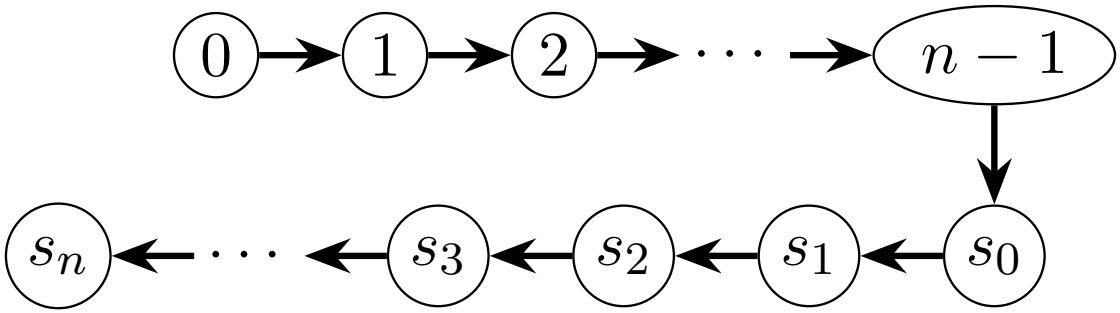
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



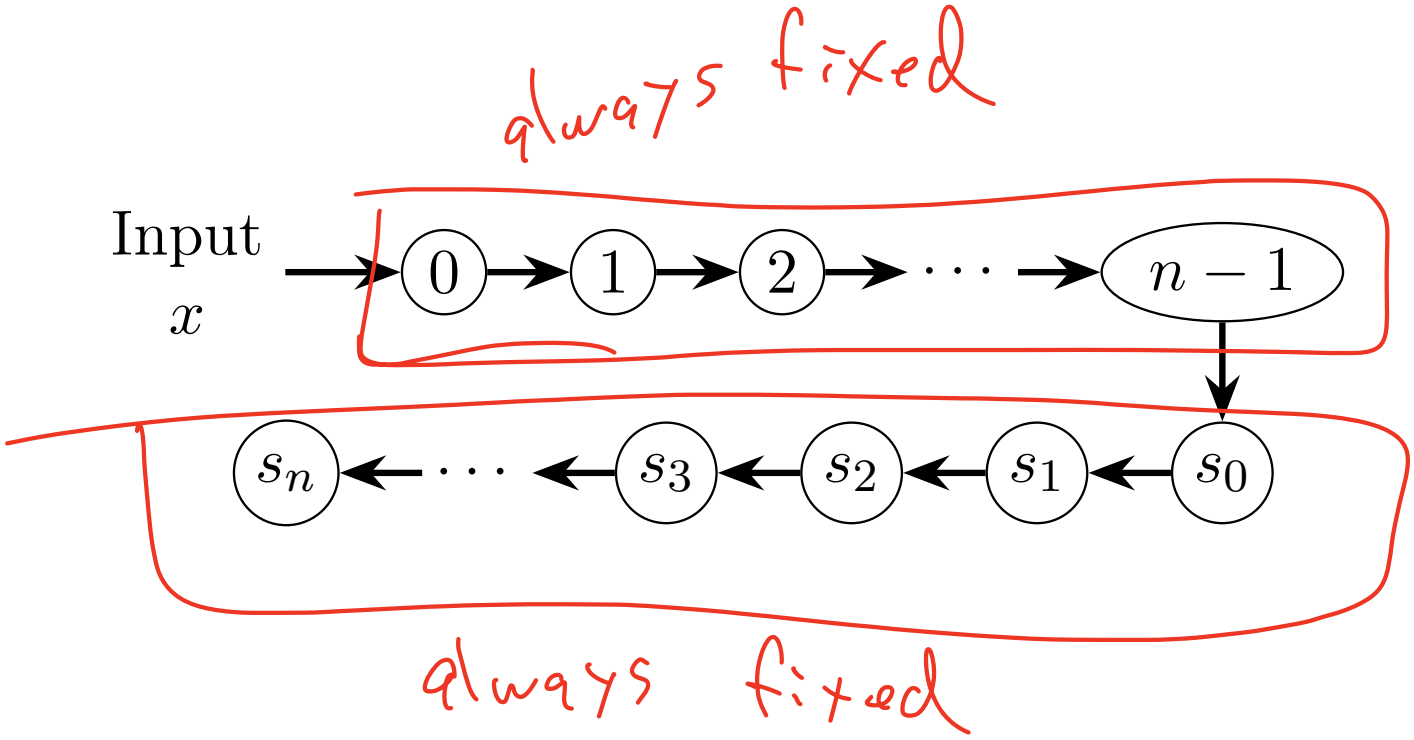
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



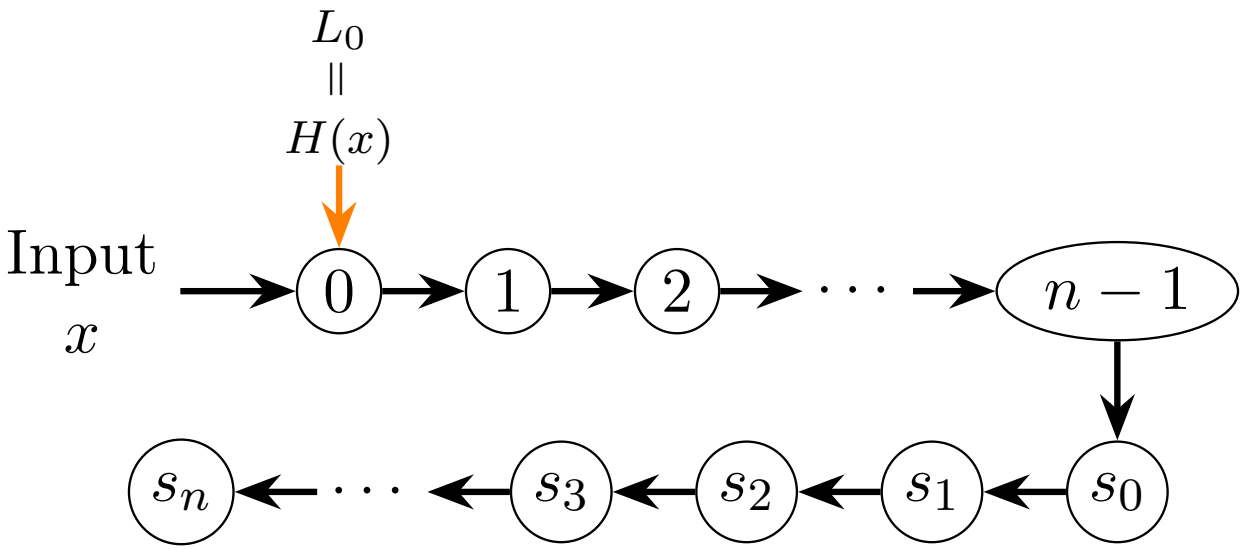
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



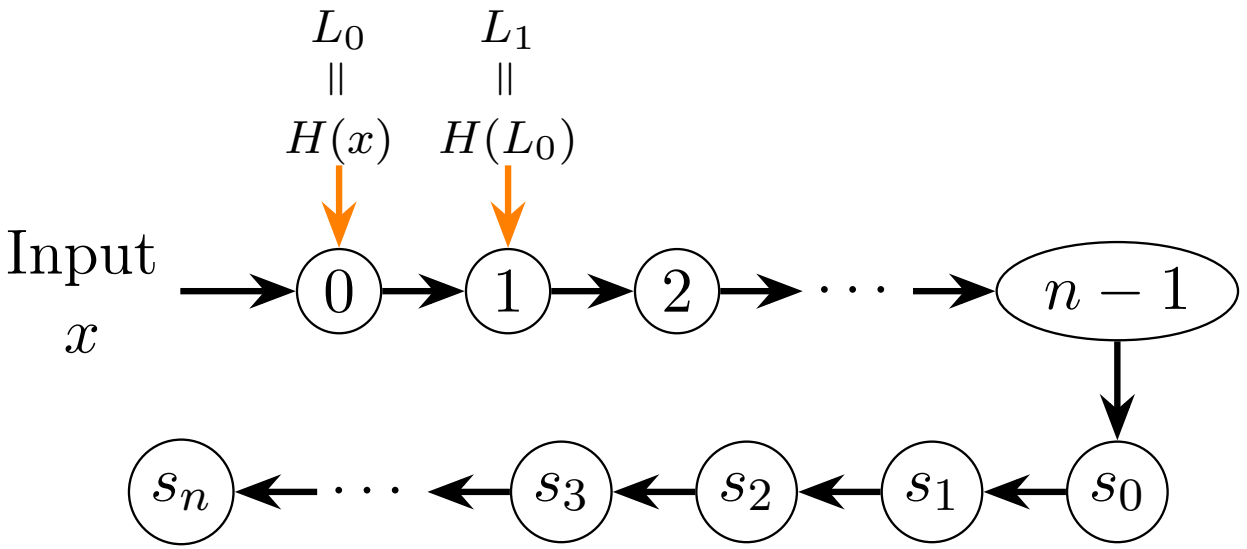
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



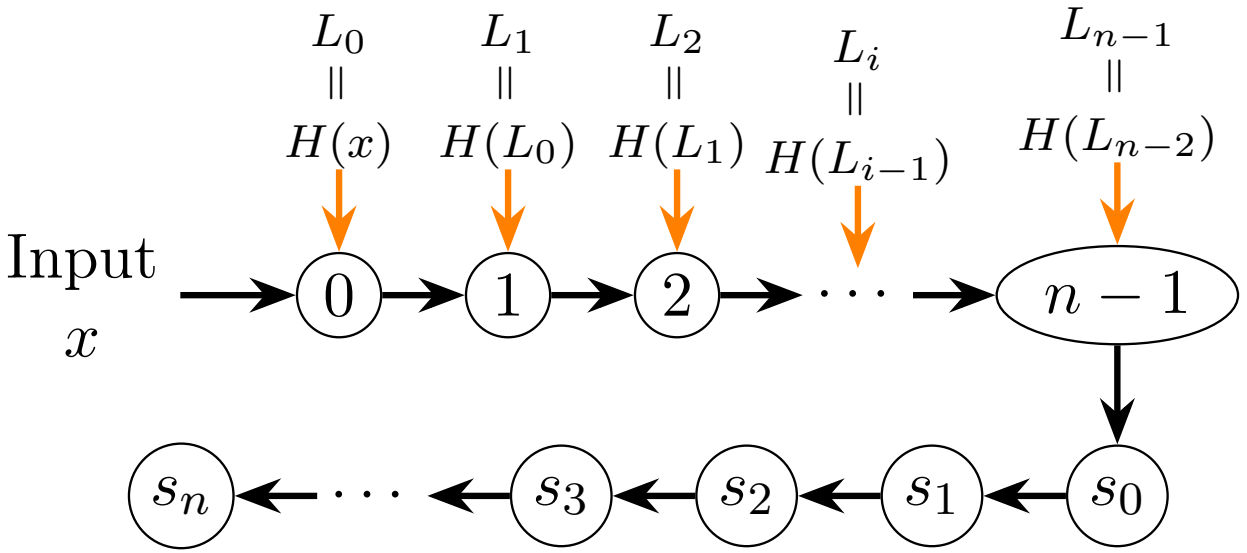
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



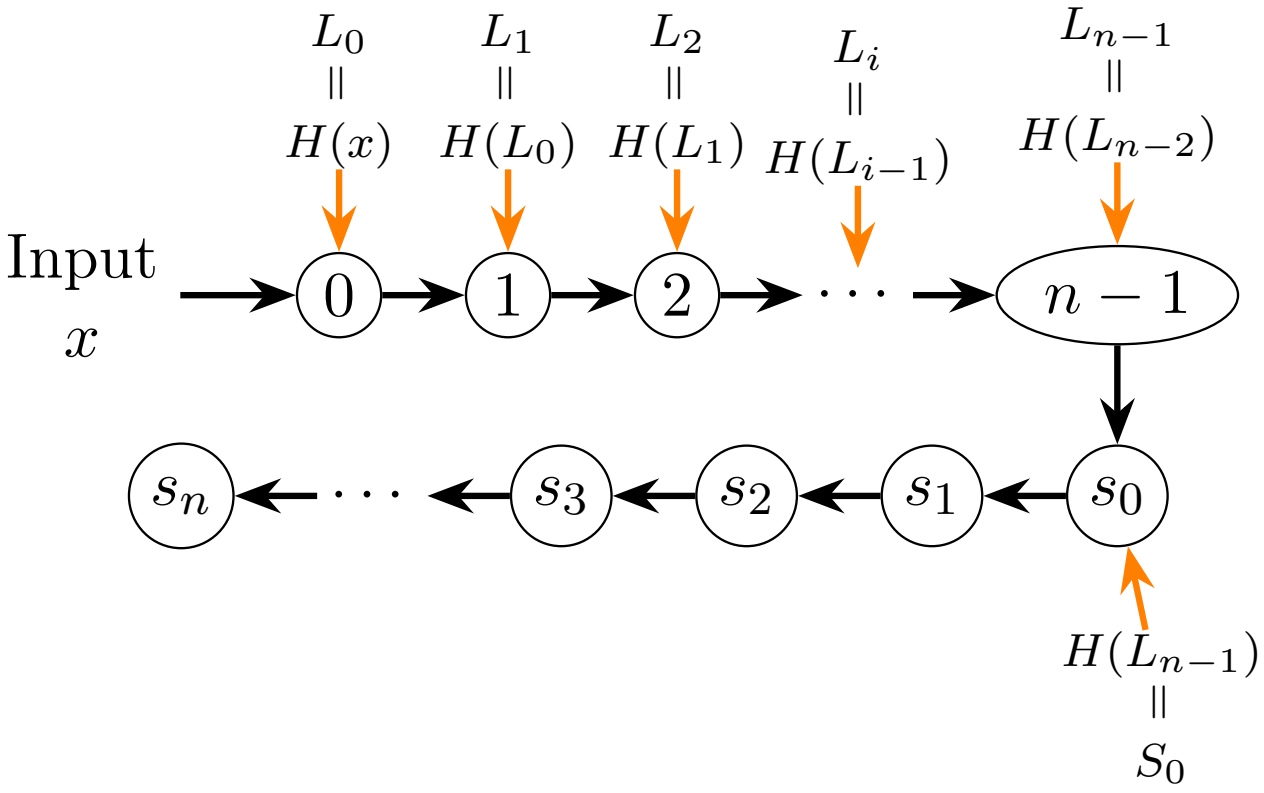
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



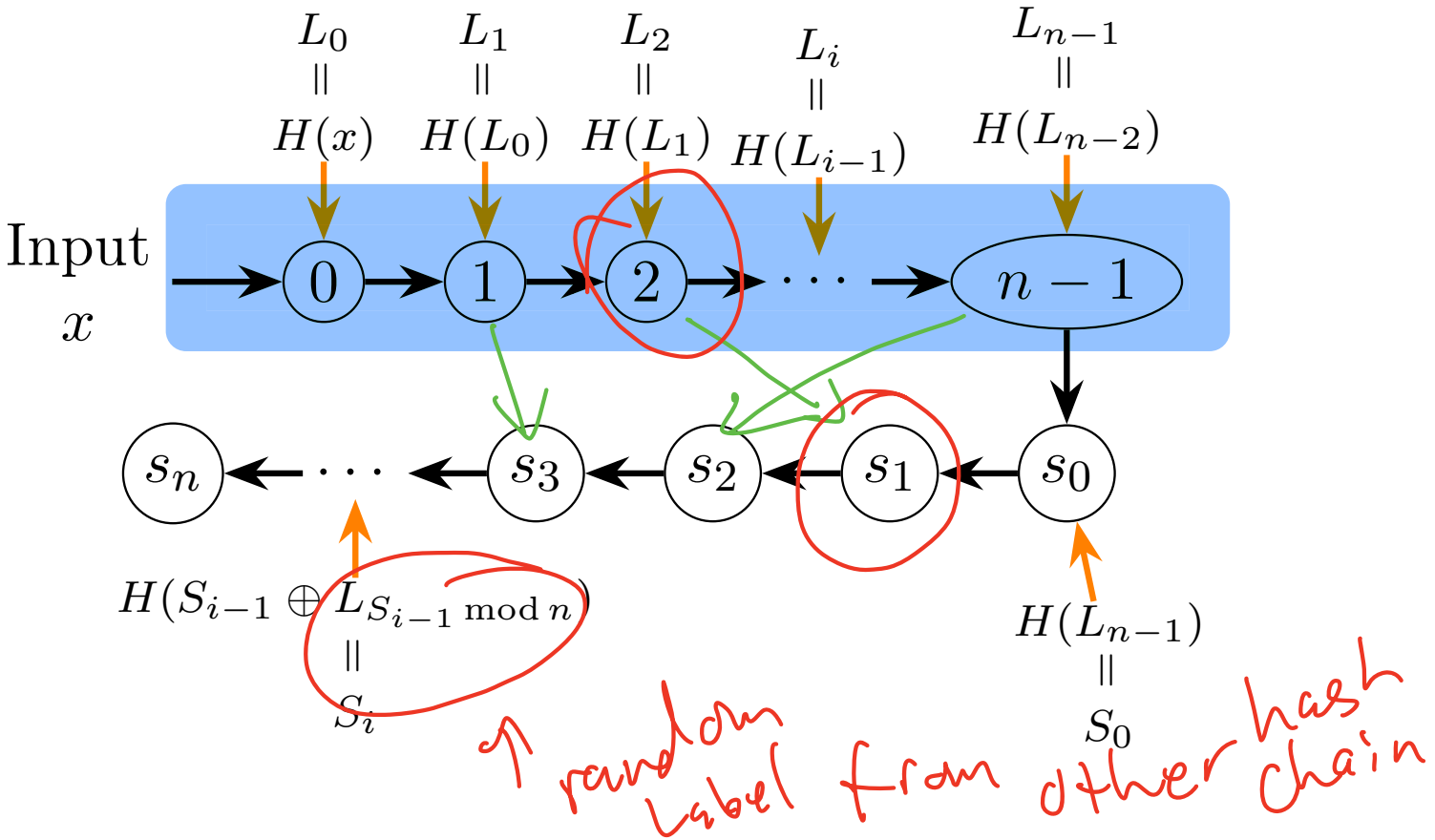
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



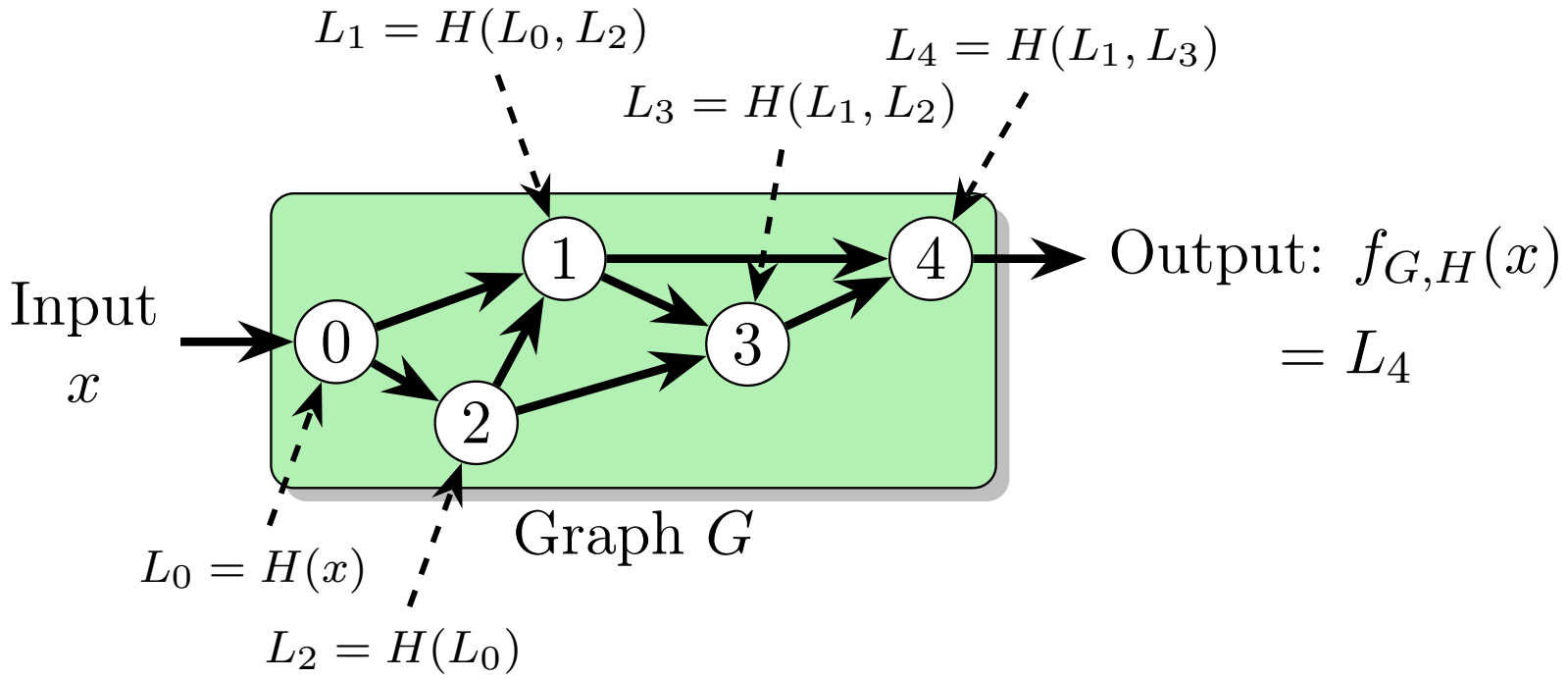
dMHFs vs. iMHFs

- For an iMHF, the graph G is randomly sampled *independently* of any input.
  - I.e., randomly sample a graph before defining the function.
- For dMHFs, the graph G is computed *on-the-fly* using some function of the hash function/RO and the input.
  - Key example: `sCrypt`



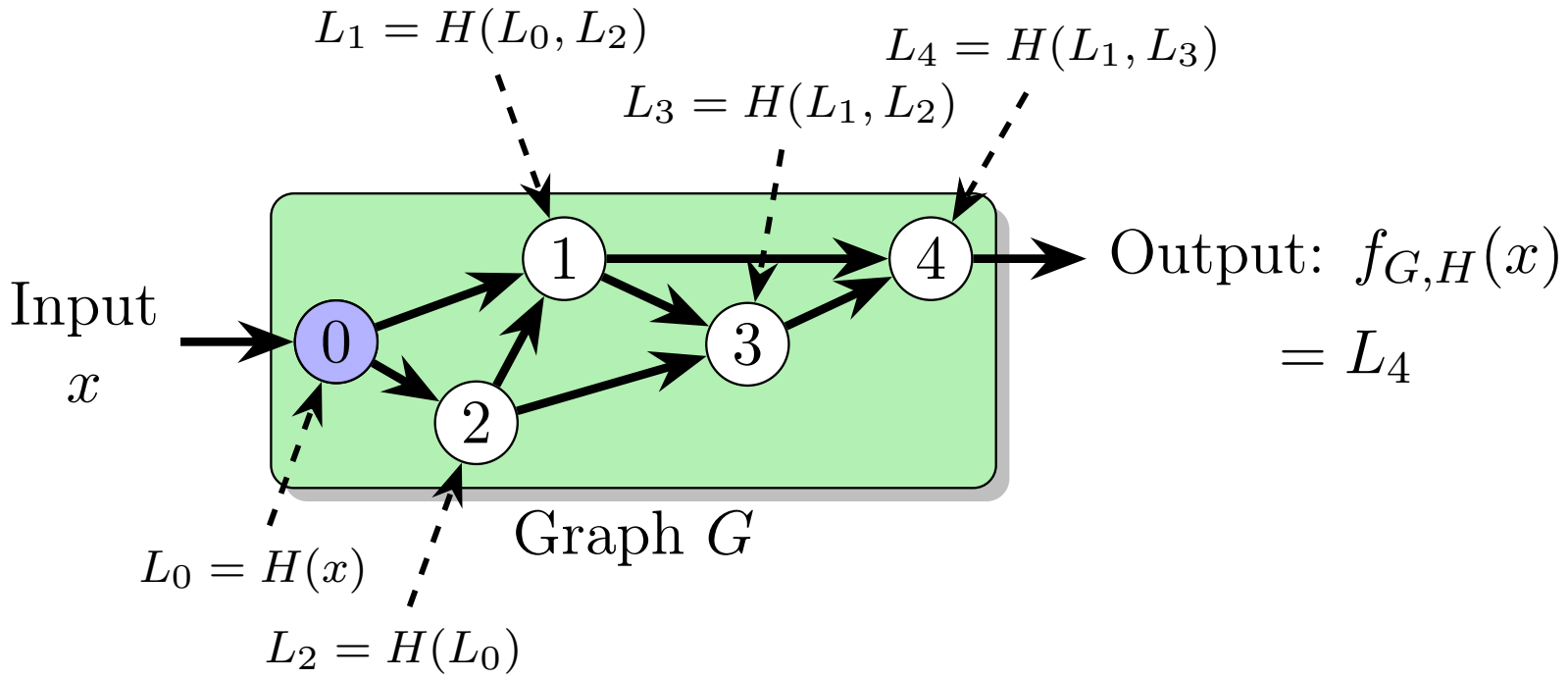
Evaluating a MHF

- Idea: *pebble the graph.*



Evaluating a MHF

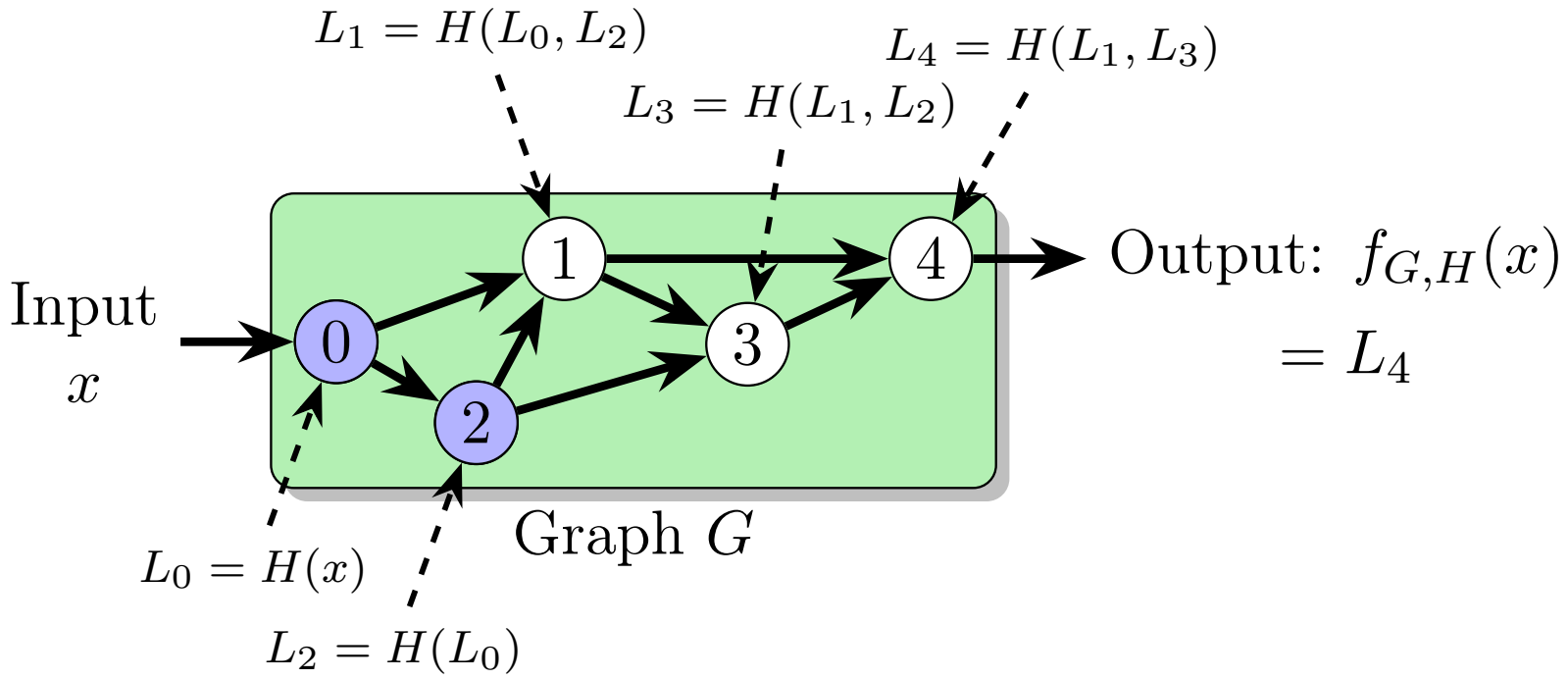
- Idea: *pebble the graph.*



- $P_0 = \{L_0\}$.

Evaluating a MHF

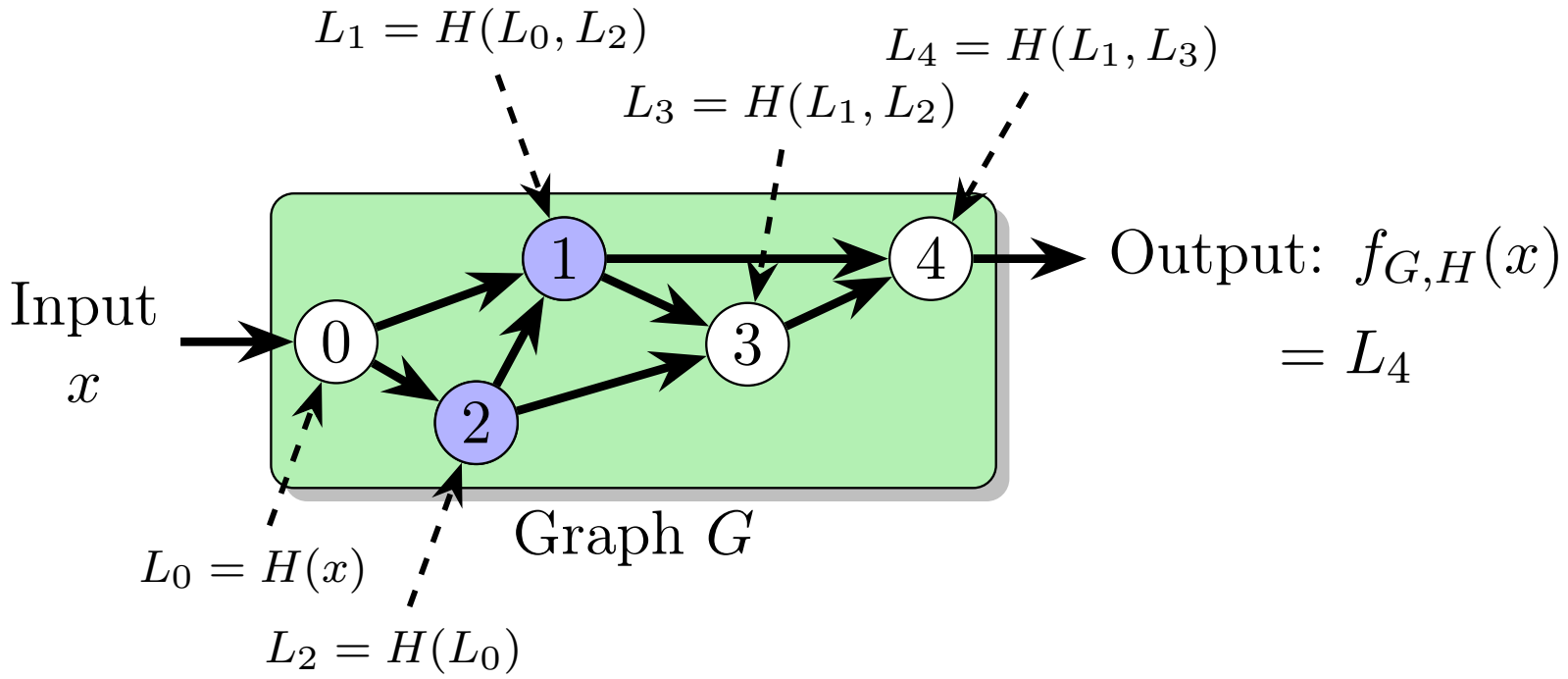
- Idea: *pebble the graph.*



- $P_0 = \{L_0\}$.
- $P_1 = \{L_0, L_2\}$.

Evaluating a MHF

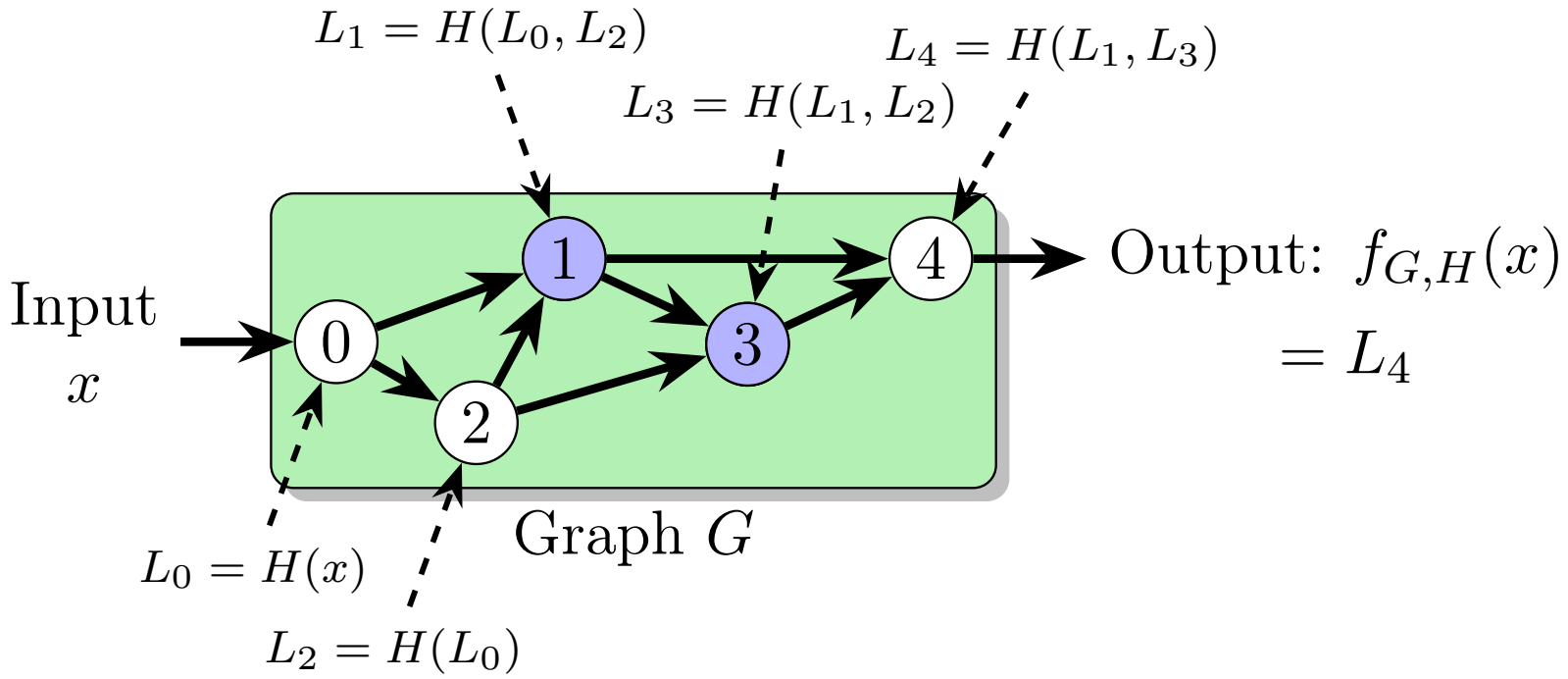
- Idea: *pebble the graph.*



- $P_0 = \{L_0\}$.
- $P_1 = \{L_0, L_2\}$.
- $P_2 = \{L_2, L_1\}$.

Evaluating a MHF

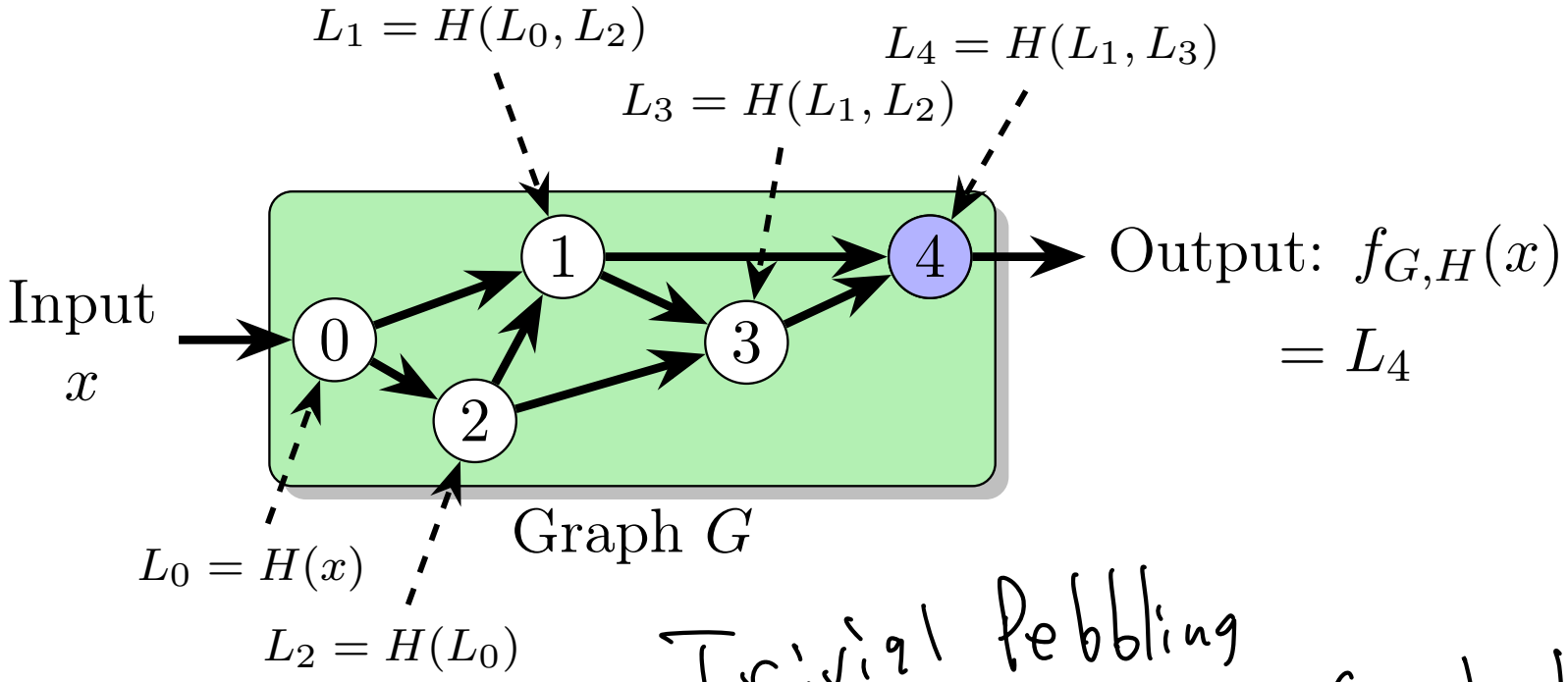
- Idea: *pebble the graph.*



- $P_0 = \{L_0\}$.
- $P_1 = \{L_0, L_2\}$.
- $P_2 = \{L_2, L_1\}$.
- $P_3 = \{L_1, L_3\}$.

Evaluating a MHF

- Idea: *pebble the graph.*



$\vec{P}$

- $P_0 = \{L_0\}$.
- $P_1 = \{L_0, L_2\}$.
- $P_2 = \{L_2, L_1\}$.
- $P_3 = \{L_1, L_3\}$.
- $P_4 = \{L_4\}$.

Trivial Pebbling

$P_0 = \{L_0\}$

$P_1 = \{L_0, L_2\}$

$P_2 = \{L_0, L_2, L_1\}$

$P_3 = \{L_0, L_1, L_2, L_3\}$

$P_4 = \{L_0, \ldots, L_4\}$

How to Measure Memory-hardness?

How to Measure Memory-hardness?

- Natural question: what does it actually mean to be *memory-hard*?

How to Measure Memory-hardness?

- Natural question: what does it actually mean to be *memory-hard*?
- There are several possible definitions of "memory-hardness."

How to Measure Memory-hardness?

- Natural question: what does it actually mean to be *memory-hard*?
- There are several possible definitions of "memory-hardness."
- Intuitively, we want to capture the idea that an adversary must lock up large amounts of memory for the *duration* of a computation.

How to Measure Memory-hardness?

- Natural question: what does it actually mean to be *memory-hard*?
- There are several possible definitions of "memory-hardness."
- Intuitively, we want to capture the idea that an adversary must lock up large amounts of memory for the *duration* of a computation.
- Idea: translate memory costs to the cost of *pebbling* the underlying DAG of a MHF.

ST Complexity: Attempt 1

ST Complexity: Attempt 1

- First natural attempt: *space-time complexity*.

ST Complexity: Attempt 1

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

ST Complexity: Attempt 1

- First natural attempt: *space-time complexity*.

- For a DAG G, we define the *ST-Complexity* of G as

# of pebbling states

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

ST Complexity: Attempt 1

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.

ST Complexity: Attempt 1

- First natural attempt: *space-time complexity*.

- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.

ST Complexity: Attempt 1

- First natural attempt: *space-time complexity*.

- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.
  - $|\mathbf{P}_i|$ is the size of the i^{th} pebbling state.

ST Complexity: Attempt 1

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.
  - $|\mathbf{P}_i|$ is the size of the i^{th} pebbling state.
- ST-Complexity has a rich theory and is studied in many works.

ST Complexity: Attempt 1

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.
  - $|\mathbf{P}_i|$ is the size of the i^{th} pebbling state.
- ST-Complexity has a rich theory and is studied in many works.

ST-Complexity is not appropriate for password hashing!

Amortization and Parallelism

Amortization and Parallelism

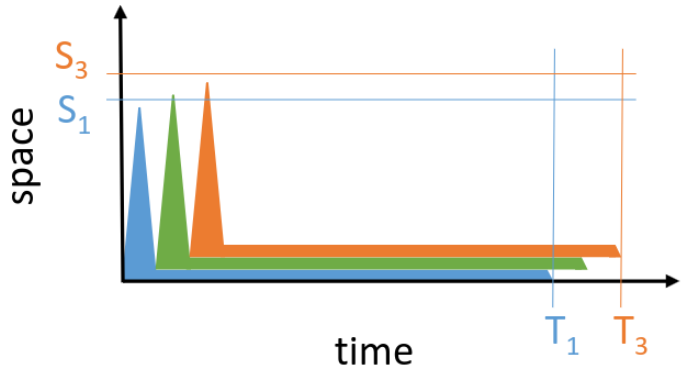
- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation*.

Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation*.
  - Password hashing is a very parallel computation!

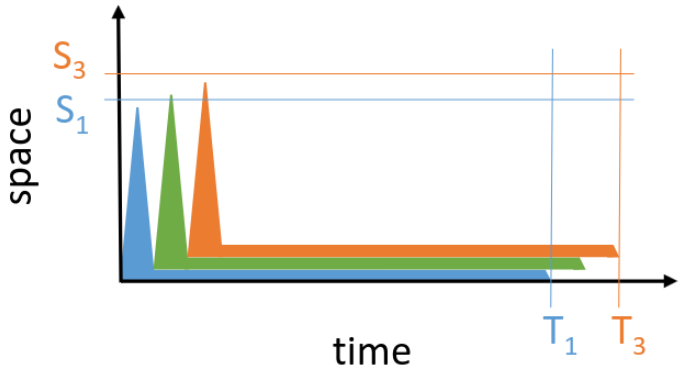
Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation.*
  - Password hashing is a very parallel computation!



Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation.*
  - Password hashing is a very parallel computation!



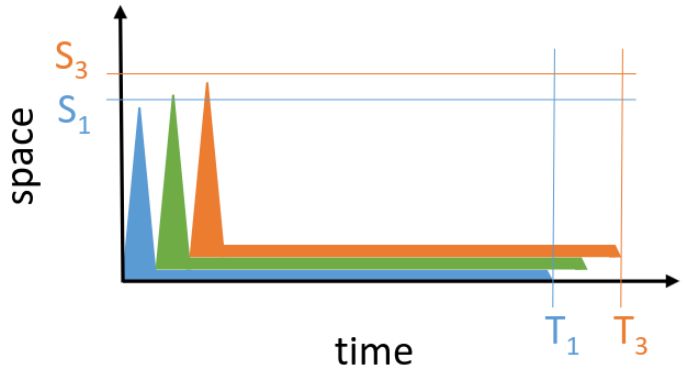
$$ST_1 = S_1 \times T_1 \approx S_3 \times T_3 = ST_3$$

cost of computing f once

cost of computing f three times

Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation.*
  - Password hashing is a very parallel computation!



$$\mathsf{ST}_1 = S_1 \times T_1 \approx S_3 \times T_3 = \mathsf{ST}_3$$

cost of computing f once

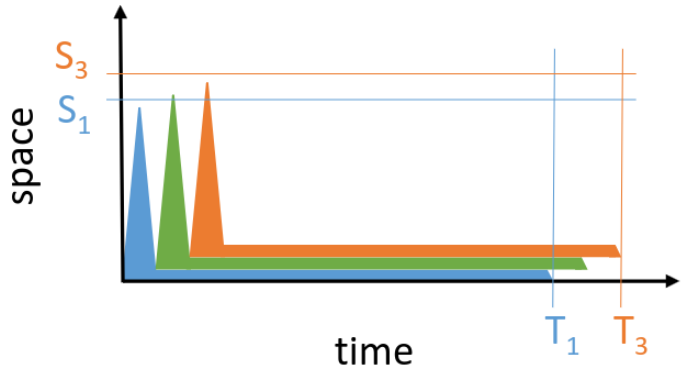
cost of computing f three times

Theorem 1

There exists a function f_n (making n calls to a random oracle) such that the ST-Complexity of $\sqrt{n}$ parallel calls to f_n is $O(\mathsf{ST}(f))$.

Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation.*
  - Password hashing is a very parallel computation!



$$\mathsf{ST}_1 = S_1 \times T_1 \approx S_3 \times T_3 = \mathsf{ST}_3$$

cost of computing f once

cost of computing f three times

Theorem 1

There exists a function f_n (making n calls to a random oracle) such that the ST-Complexity of $\sqrt{n}$ parallel calls to f_n is $O(\mathsf{ST}(f))$.

For measuring memory-hardness, we would like an *amortized* memory cost (amortized area-time complexity)

Cumulative (Memory) Complexity

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

- $|\mathbf{P}_i|$ is the memory used at step i of the computation.

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

- $|\mathbf{P}_i|$ is the memory used at step i of the computation.
- Guessing two passwords now doubles an attacker's cost!

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

- $|\mathbf{P}_i|$ is the memory used at step i of the computation.
- Guessing two passwords now doubles an attacker's cost!

$$\mathsf{CC}(G, G) = 2 \cdot \mathsf{CC}(G)$$

Next Time: CC, the Parallel ROM, and Depth-Robustness