

MIT 6.5620/6.875/18.425

Foundations of Cryptography

Lecture 14: Secure Storage

October 28, 2026

Today's Setting: Storing Large Data

Today's Setting: Storing Large Data

- **Your goal:** Store lots of data.

Today's Setting: Storing Large Data

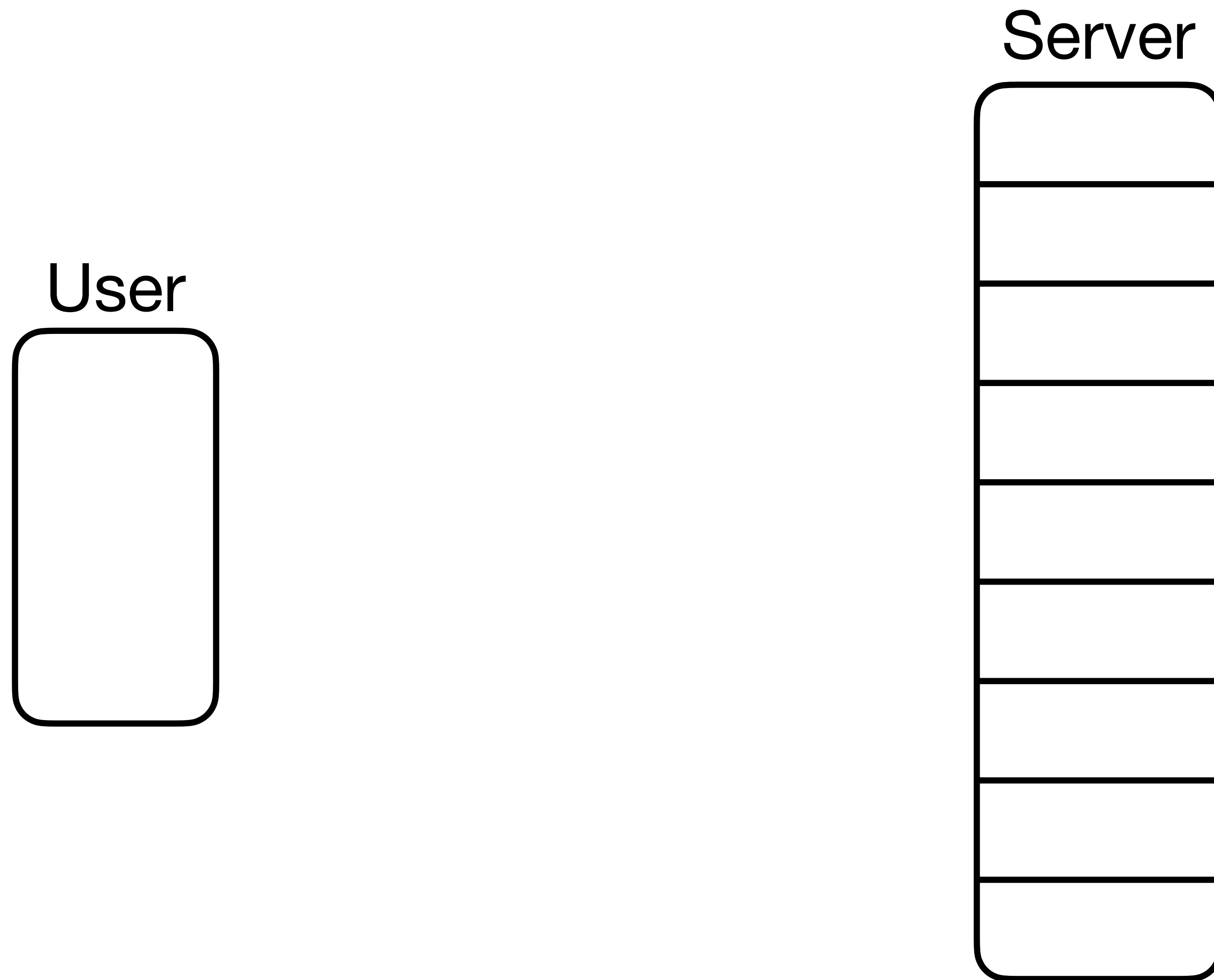
- **Your goal:** Store lots of data.
- **Problem:** You don't have enough storage.

Today's Setting: Storing Large Data

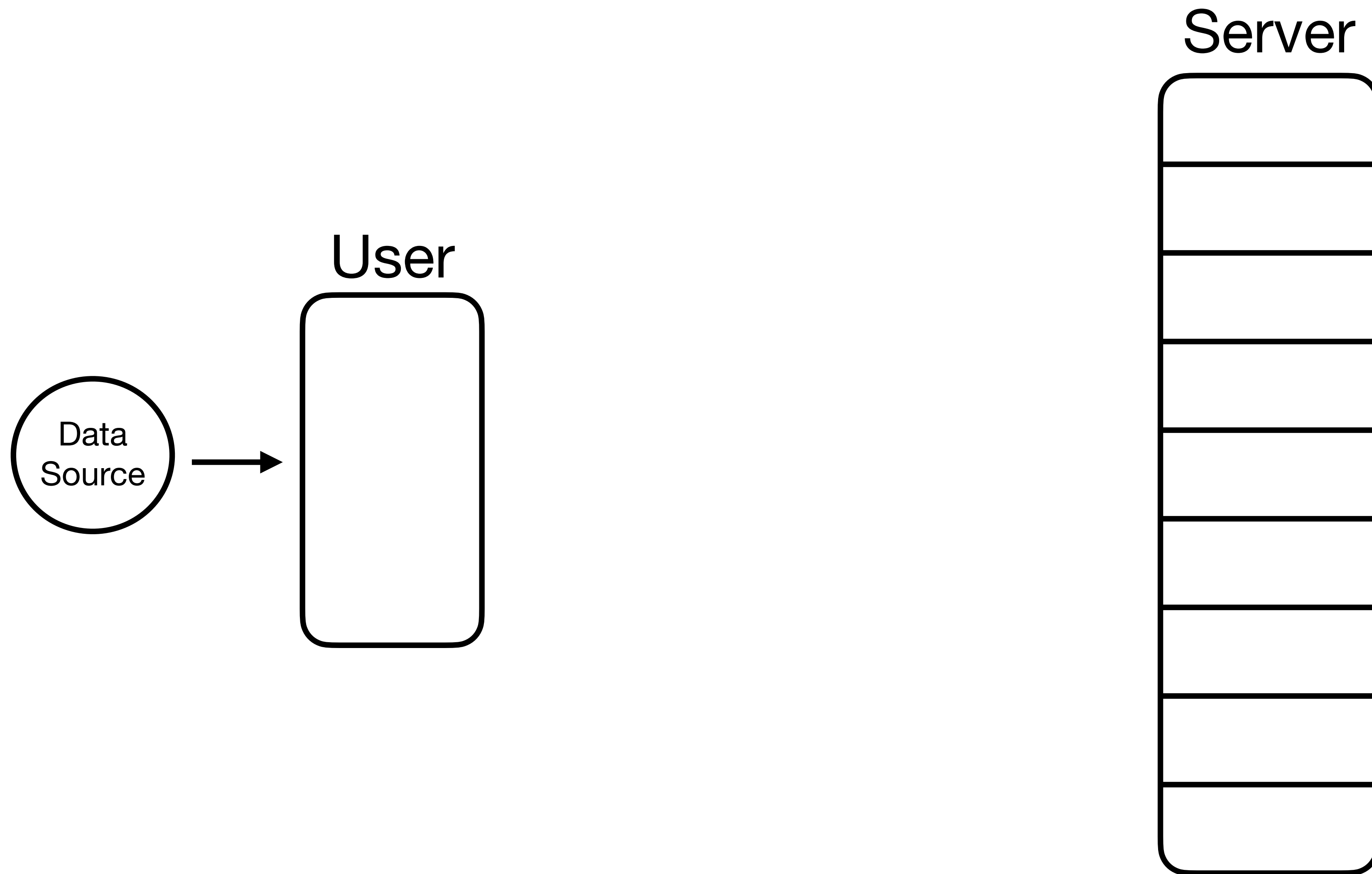
- **Your goal:** Store lots of data.
- **Problem:** You don't have enough storage.
- **Common solution:** Store your data on a **remote server**.

Basic Setup

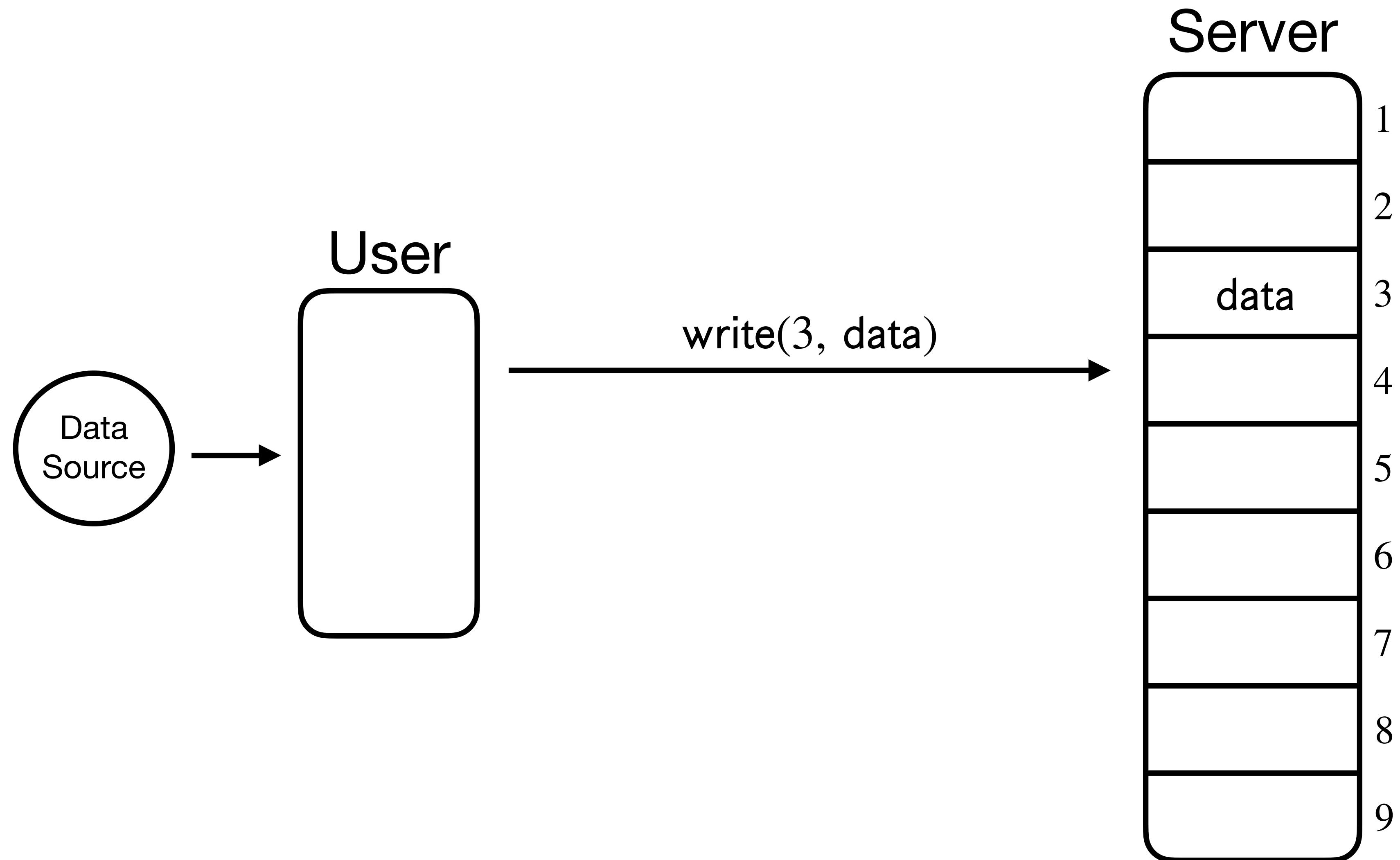
Basic Setup



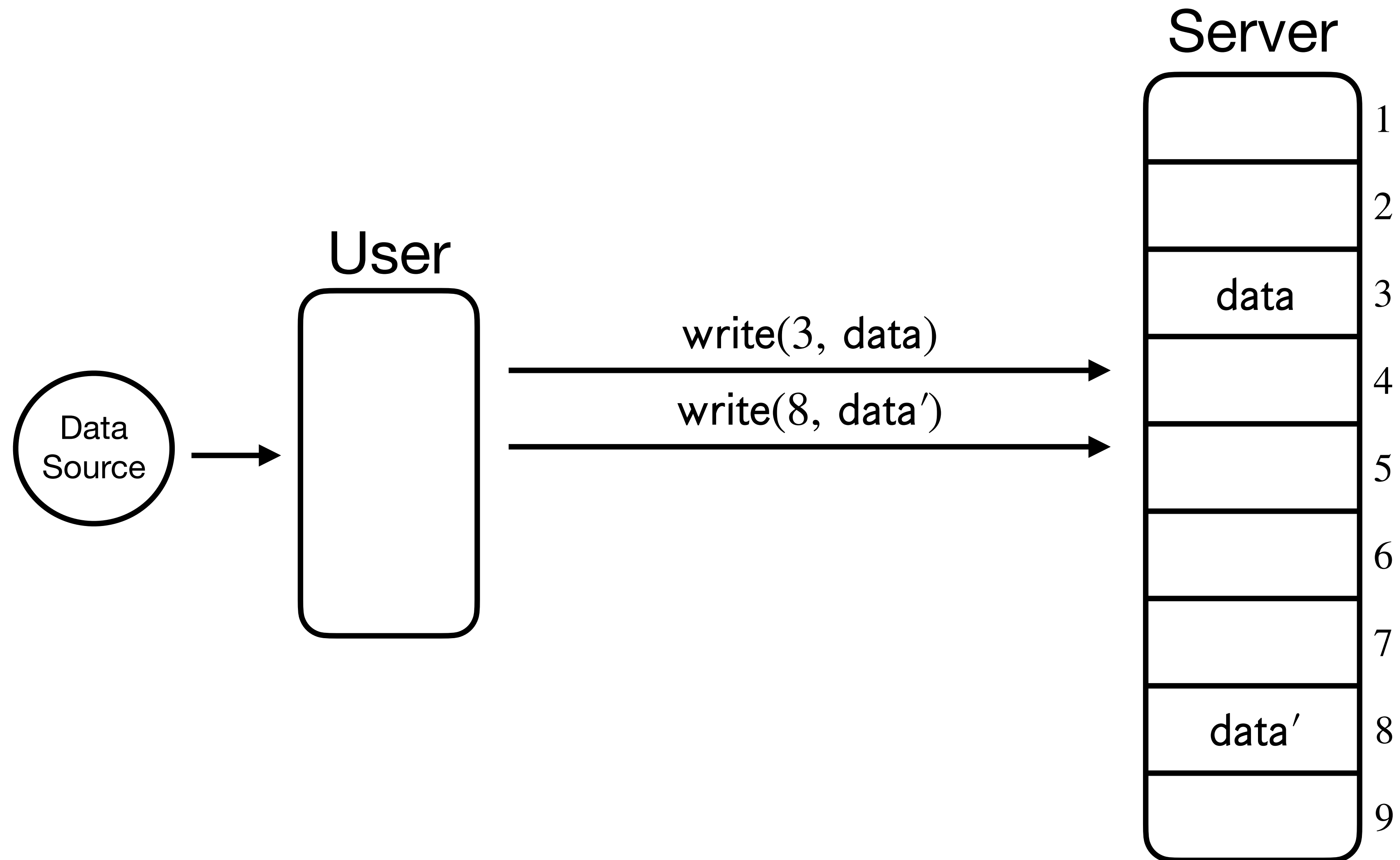
Basic Setup



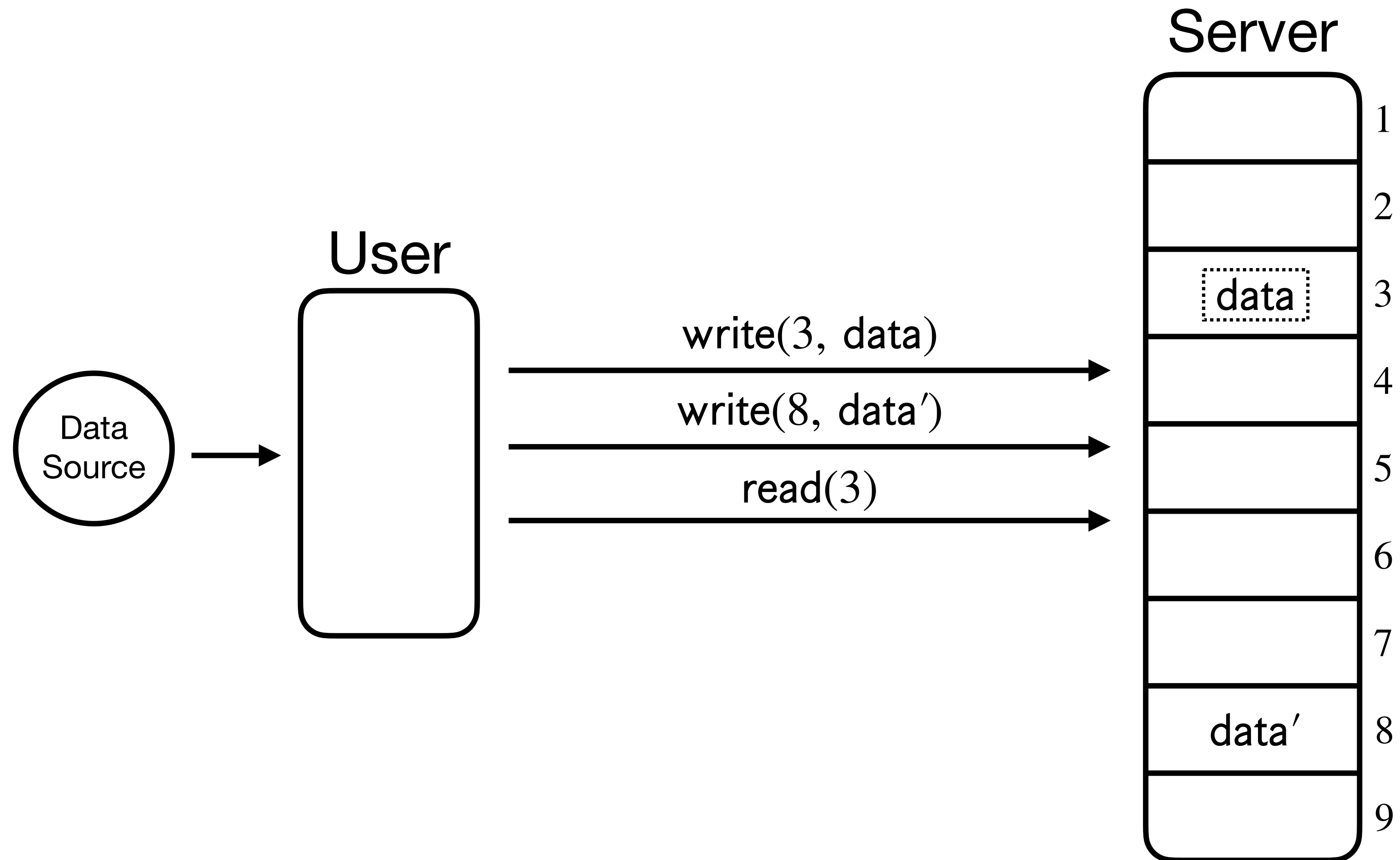
Basic Setup



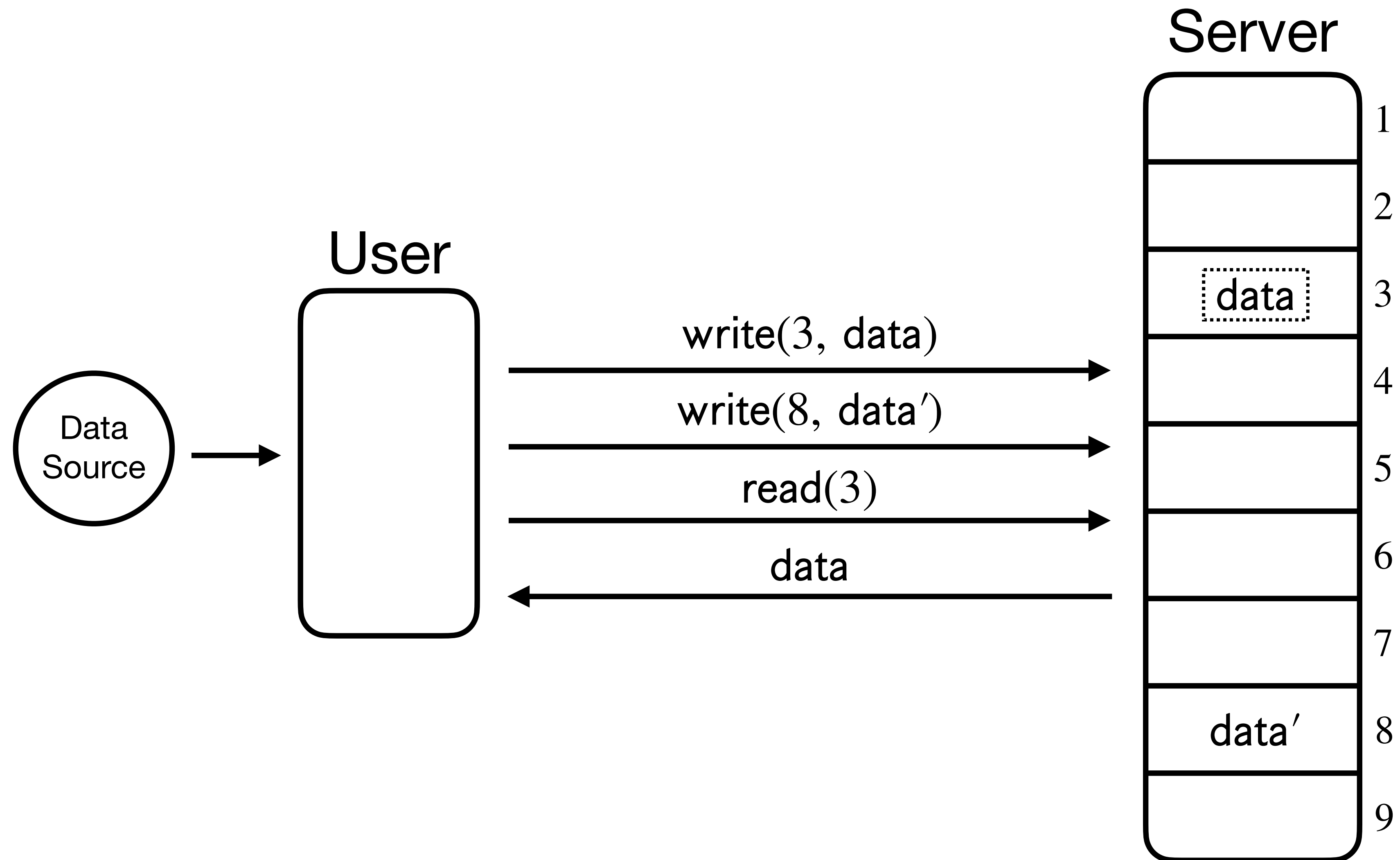
Basic Setup



Basic Setup



Basic Setup



Trusting the Remote Server

- **Common solution:** Store data on a remote server.

Trusting the Remote Server

- **Common solution:** Store data on a remote server.



OpenAI



Google Cloud



Microsoft Azure

Trusting the Remote Server

- **Common solution:** Store data on a remote server.
- Great!



OpenAI



Google Cloud



Microsoft Azure

Trusting the Remote Server

- **Common solution:** Store data on a remote server.

- Great!

- ...right? Do you trust them?



OpenAI



Google Cloud

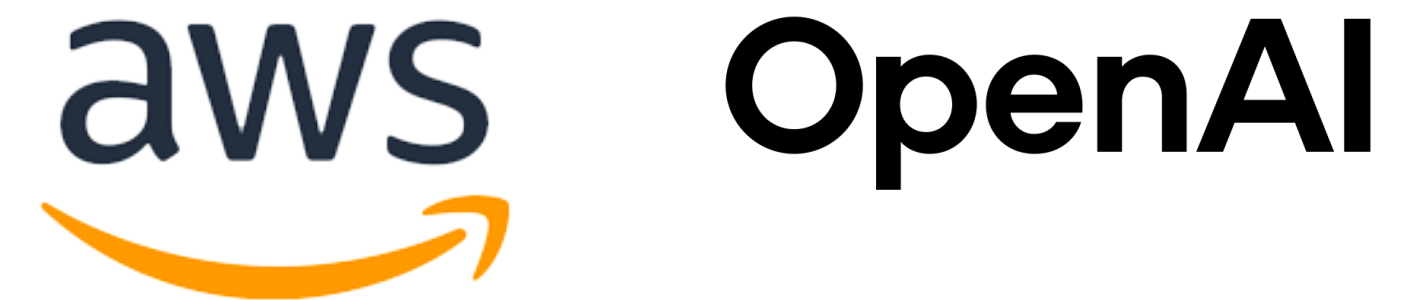


Microsoft Azure

Trusting the Remote Server

- **Common solution:** Store data on a remote server.

- Great!



- ...right? Do you trust them?



- Why shouldn't we trust the server?



Trusting the Remote Server

- **Common solution:** Store data on a remote server.

- Great!

- ...right? Do you trust them?

- Why shouldn't we trust the server?

- What are we trying to prevent?



OpenAI



Google Cloud



Microsoft Azure

Trust Issue 1: Integrity

Trust Issue 1: Integrity

- What if an adversarial server corrupts your data?

Trust Issue 1: Integrity

- What if an adversarial server corrupts your data?
- Can we prevent adversary from erasing your data?

Trust Issue 1: Integrity

- What if an adversarial server corrupts your data?
- Can we prevent adversary from erasing your data?
- **Unavoidable...** but at least you can detect this.

Trust Issue 1: Integrity

- What if an adversarial server corrupts your data?
- Can we prevent adversary from erasing your data?
 - **Unavoidable...** but at least you can detect this.
- Can we prevent adversary from modifying data *undetectably*?

Trust Issue 1: Integrity

- What if an adversarial server corrupts your data?
- Can we prevent adversary from erasing your data?
 - **Unavoidable...** but at least you can detect this.
- Can we prevent adversary from modifying data *undetectably*?
 - **Yes!** (At some cost – we'll see.)

Trust Issue 2: Privacy (Obliviousness)

Trust Issue 2: Privacy (Obliviousness)

- What if the server wants to see your data?

Trust Issue 2: Privacy (Obliviousness)

- What if the server wants to see your data?
- Can we prevent a curious adversary from learning anything about your data?

Trust Issue 2: Privacy (Obliviousness)

- What if the server wants to see your data?
- Can we prevent a curious adversary from learning anything about your data?
 - **Yes!** (At some cost – we'll see.)

Trust Issue 2: Privacy (Obliviousness)

- What if the server wants to see your data?
- Can we prevent a curious adversary from learning anything about your data?
 - **Yes!** (At some cost – we'll see.)
 - (Adversary will learn amount of data, but that's it.)

Trust Issue 3: Both simultaneously?

Trust Issue 3: Both simultaneously?

- What if the server tries **tampering** your data with the goal of **learning** something about your data?

Trust Issue 3: Both simultaneously?

- What if the server tries **tampering** your data with the goal of **learning** something about your data?
 - This is subtle!

Trust Issue 3: Both simultaneously?

- What if the server tries **tampering** your data with the goal of **learning** something about your data?
- This is subtle!
- **Still doable!** (At some cost – we'll see if time permits.)

Solutions to These Issues: Terminology

Solutions to These Issues: Terminology

1. Integrity issue: **Memory Checking** [Blum et al. '91]

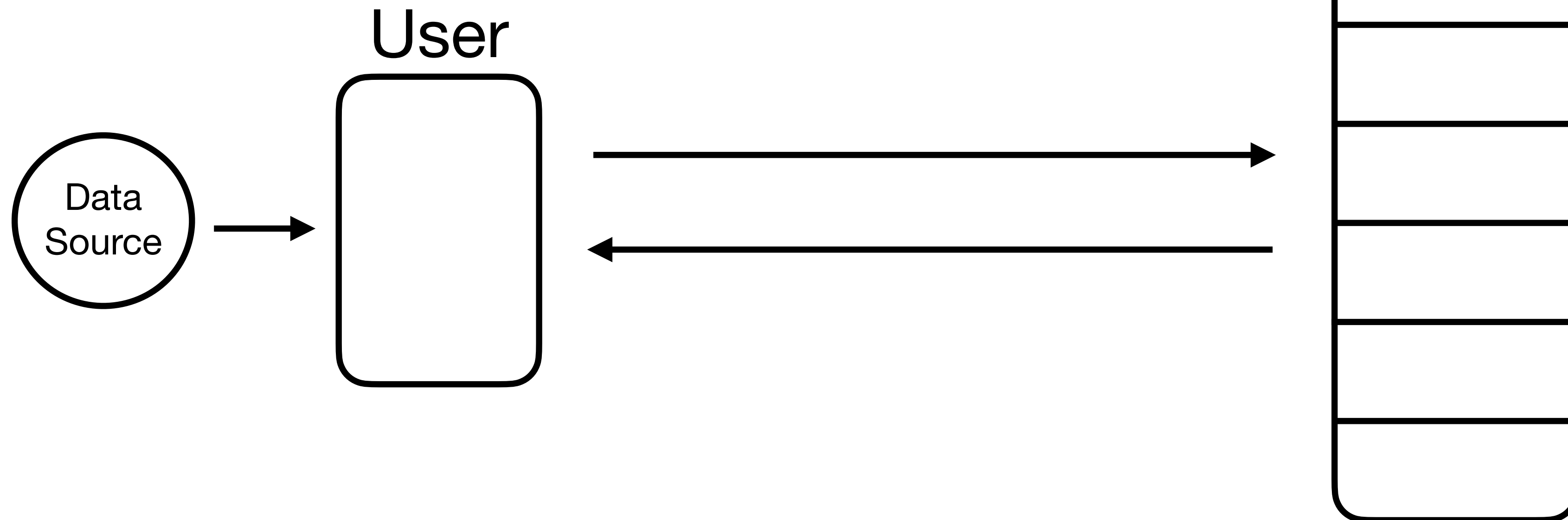
Solutions to These Issues: Terminology

1. Integrity issue: **Memory Checking** [Blum et al. '91]
[Goldreich '87,
Ostrovsky '90,
Goldreich-
Ostrovsky '96]
2. Privacy issue: (honest-but-curious) **Oblivious RAM (ORAM)**

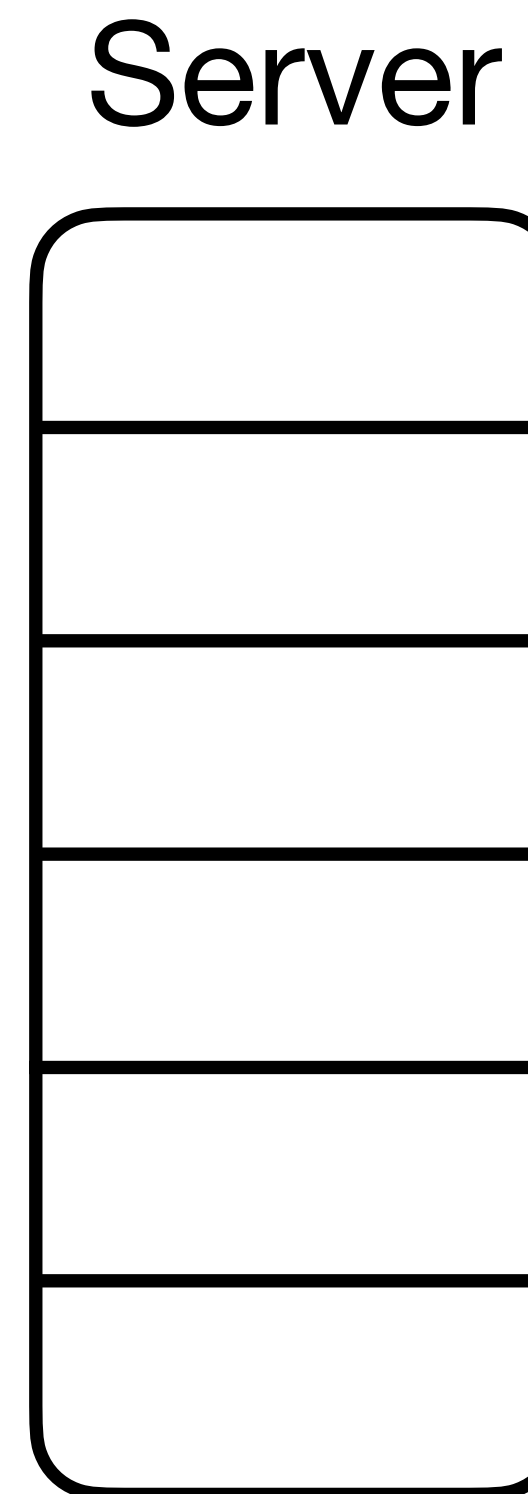
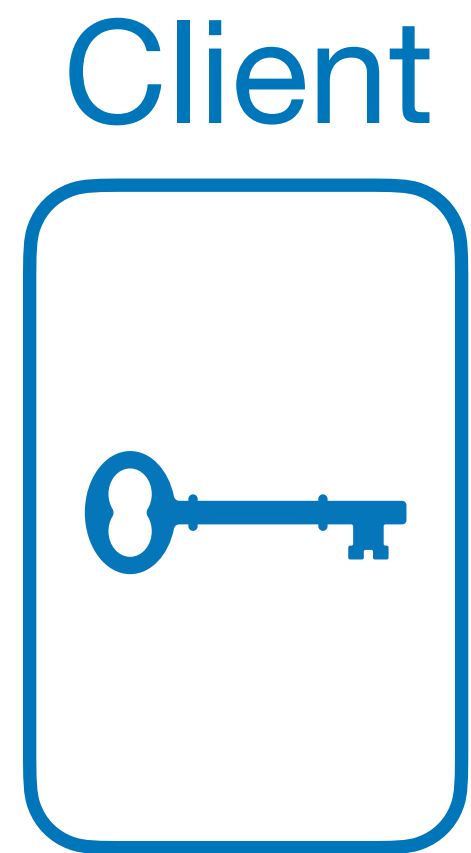
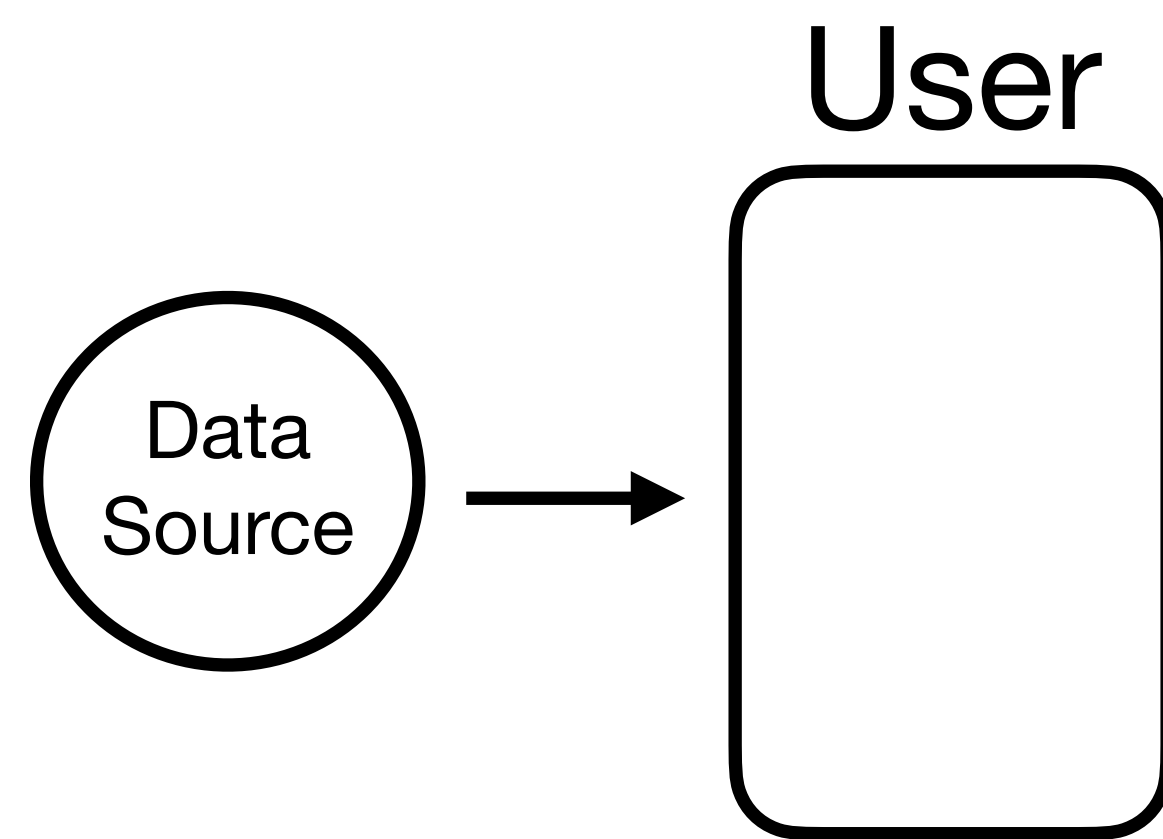
Solutions to These Issues: Terminology

1. Integrity issue: **Memory Checking** [Blum et al. '91]
[Goldreich '87,
Ostrovsky '90,
Goldreich-
Ostrovsky '96]
2. Privacy issue: (honest-but-curious) **Oblivious RAM (ORAM)**
3. Privacy and integrity issue: **Maliciously Secure ORAM**

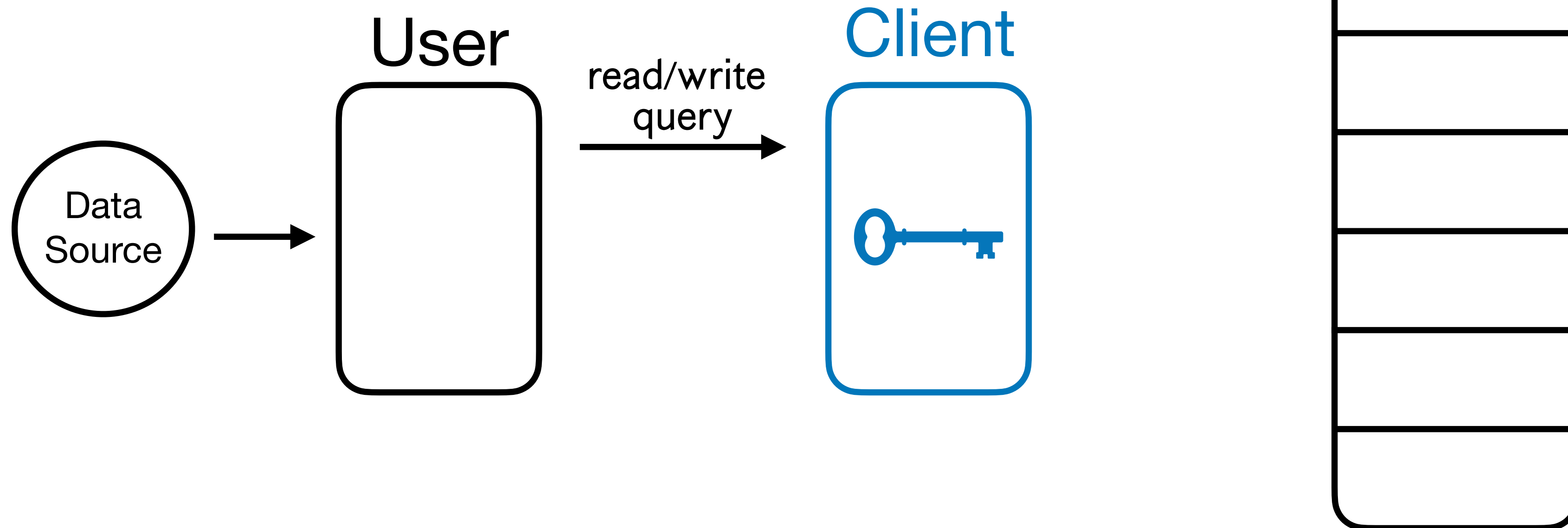
Setup



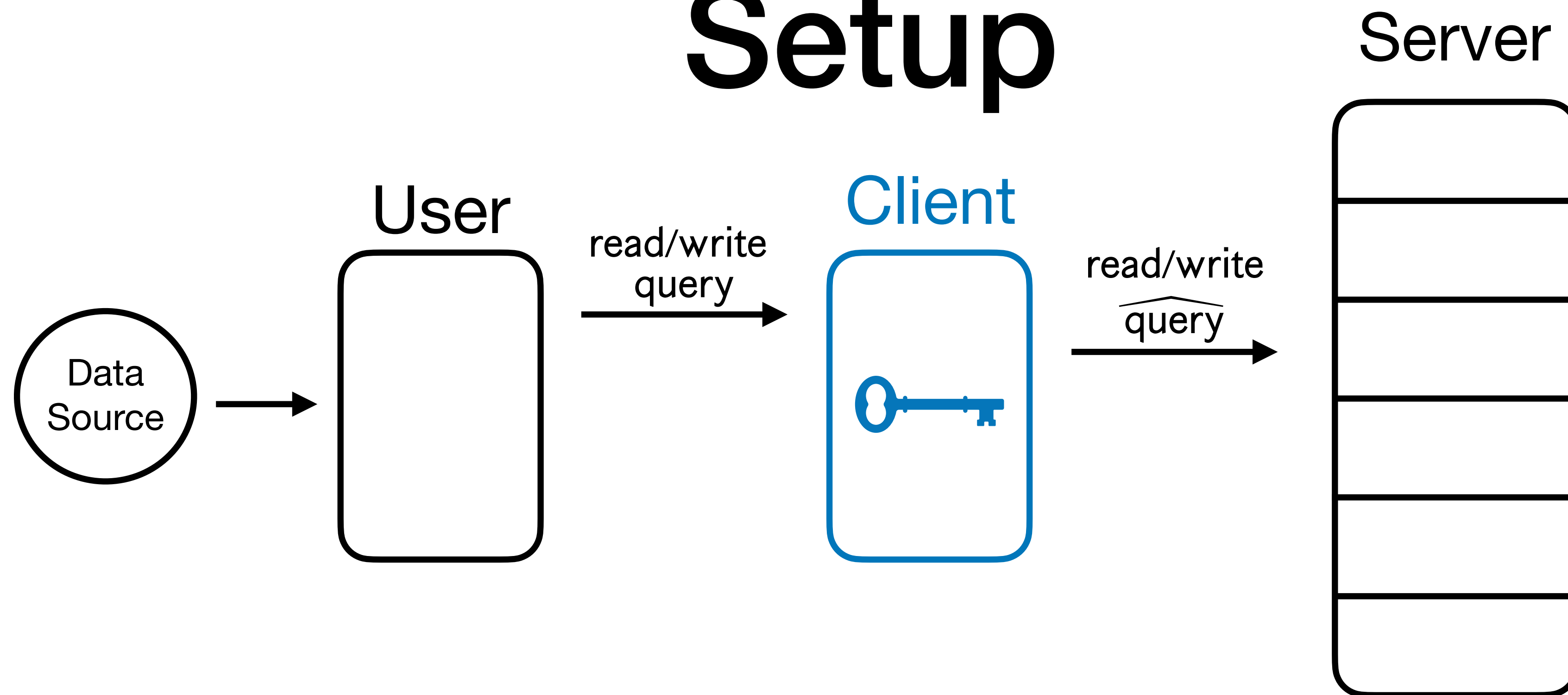
Setup



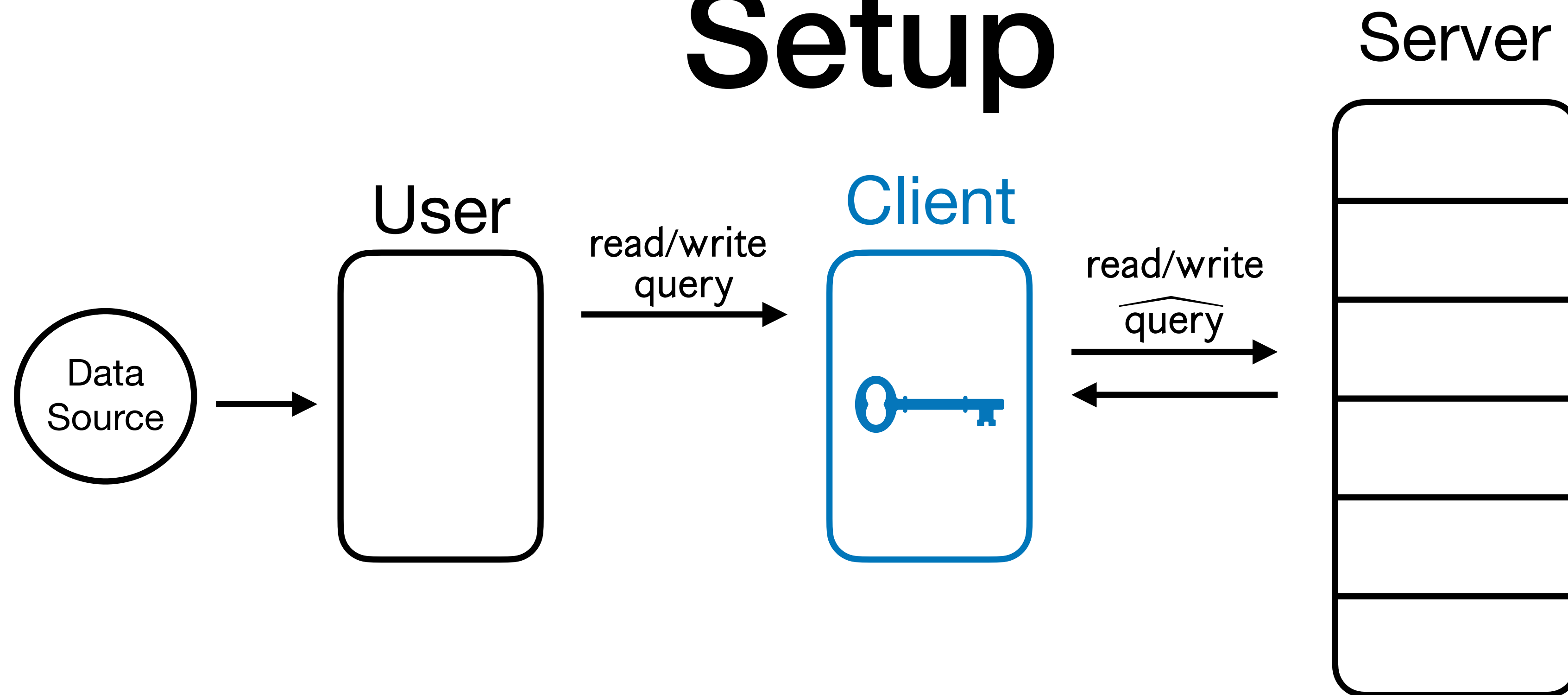
Setup



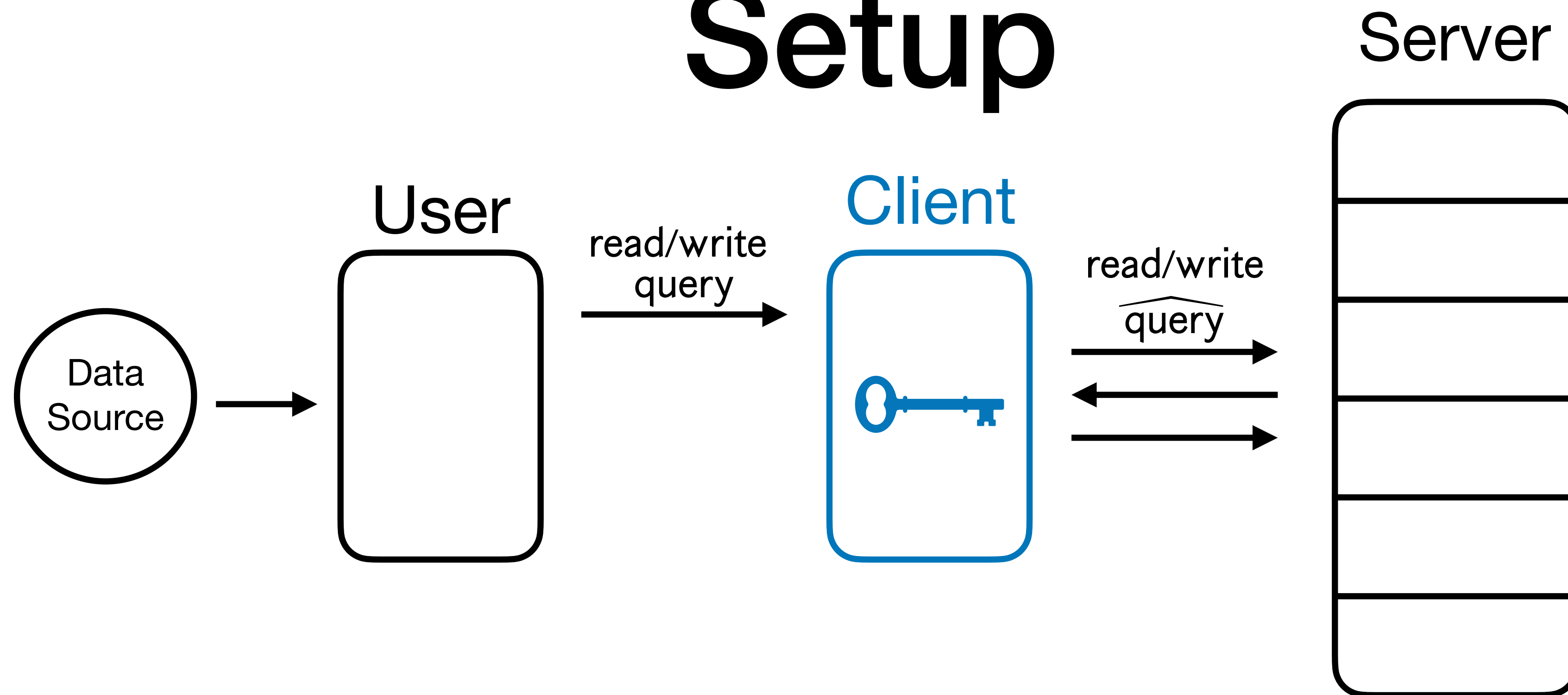
Setup



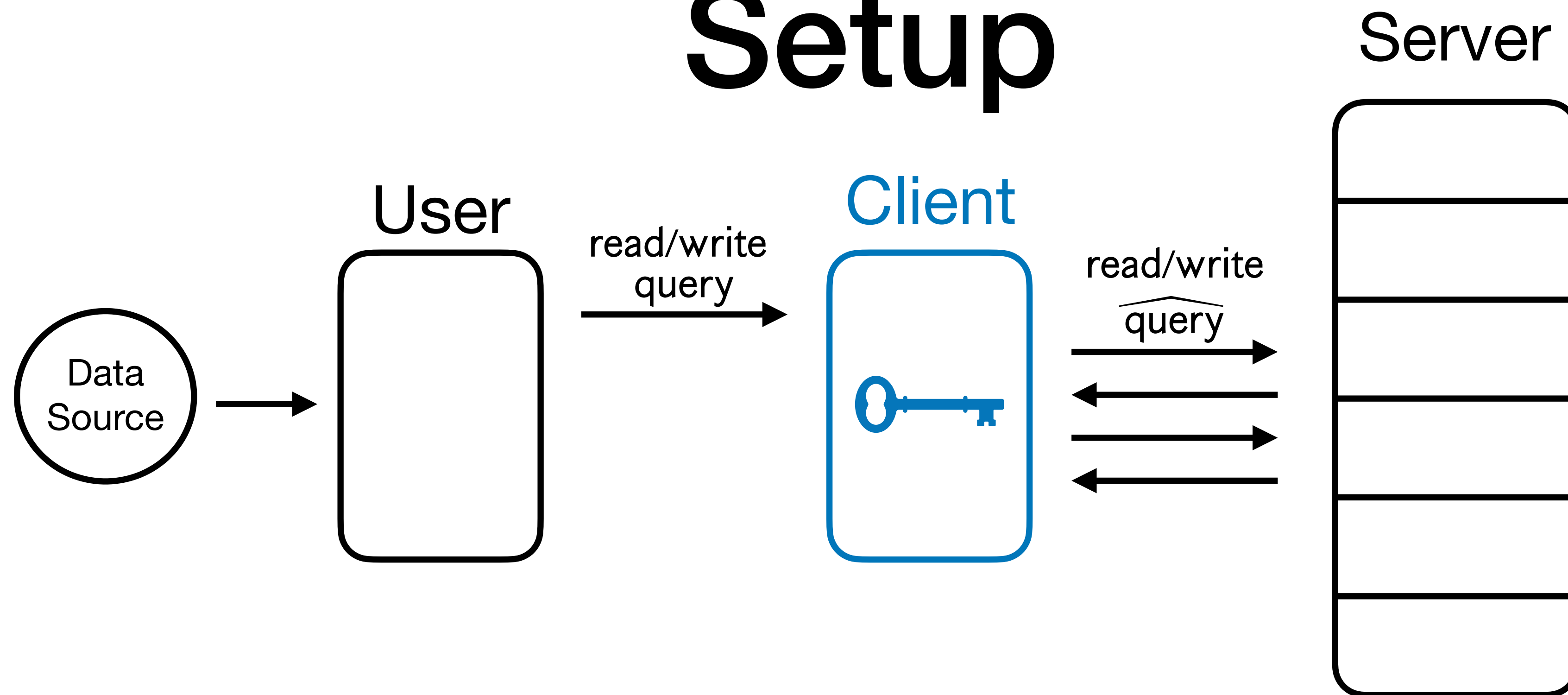
Setup



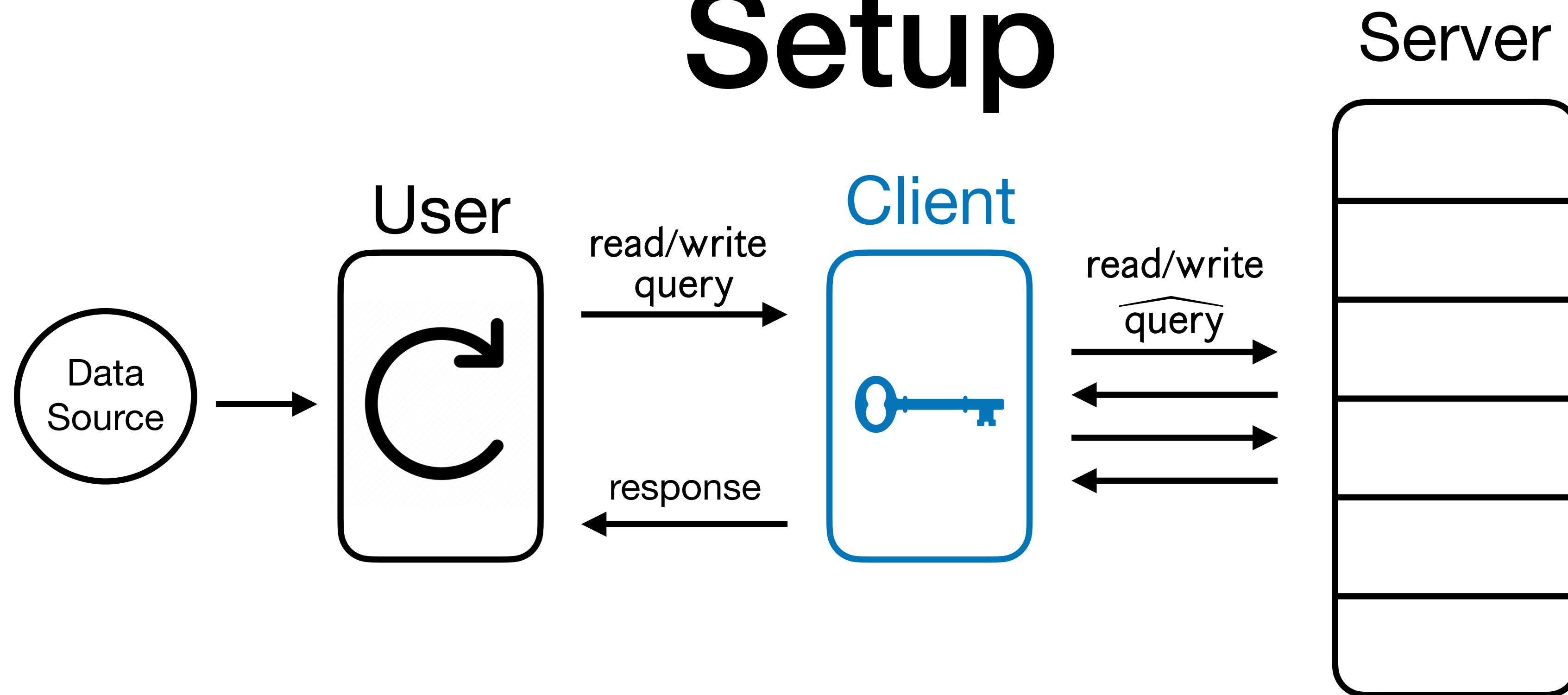
Setup



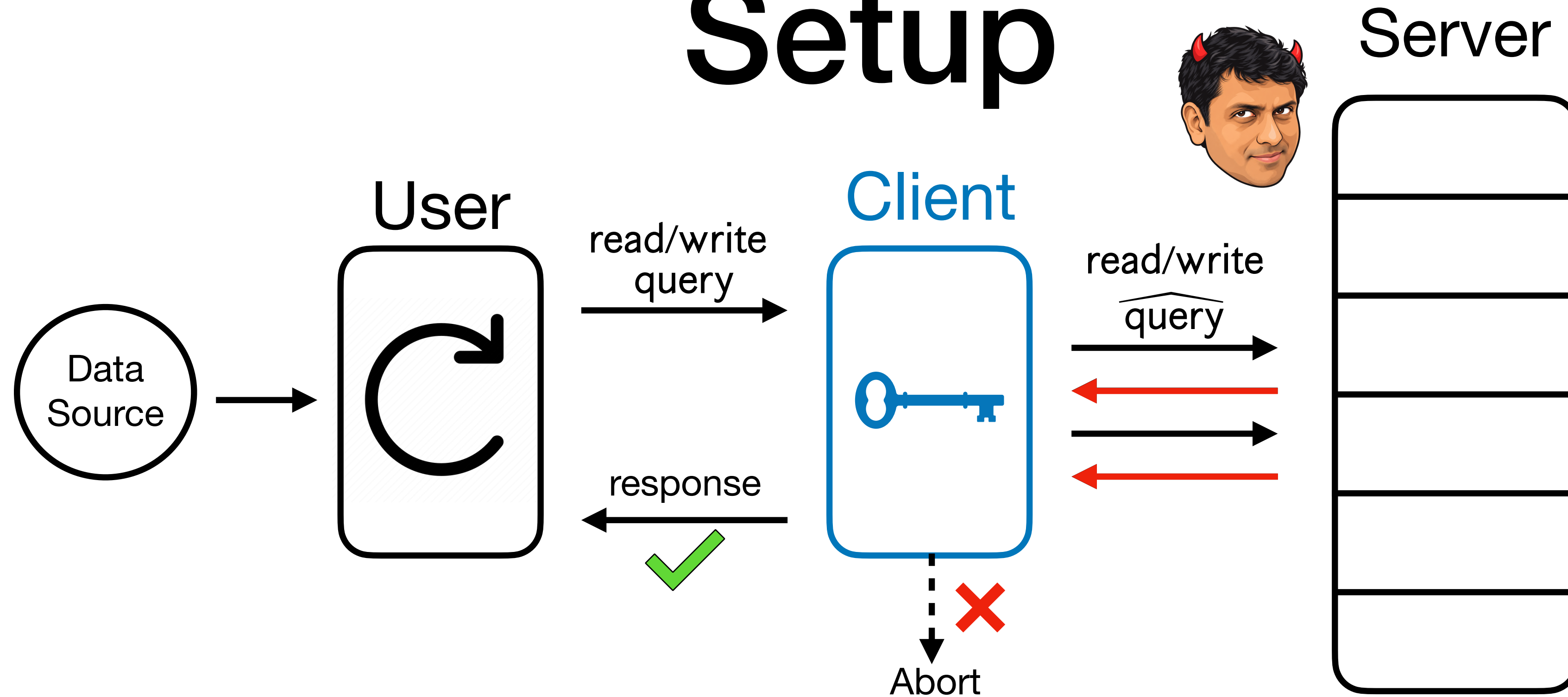
Setup



Setup

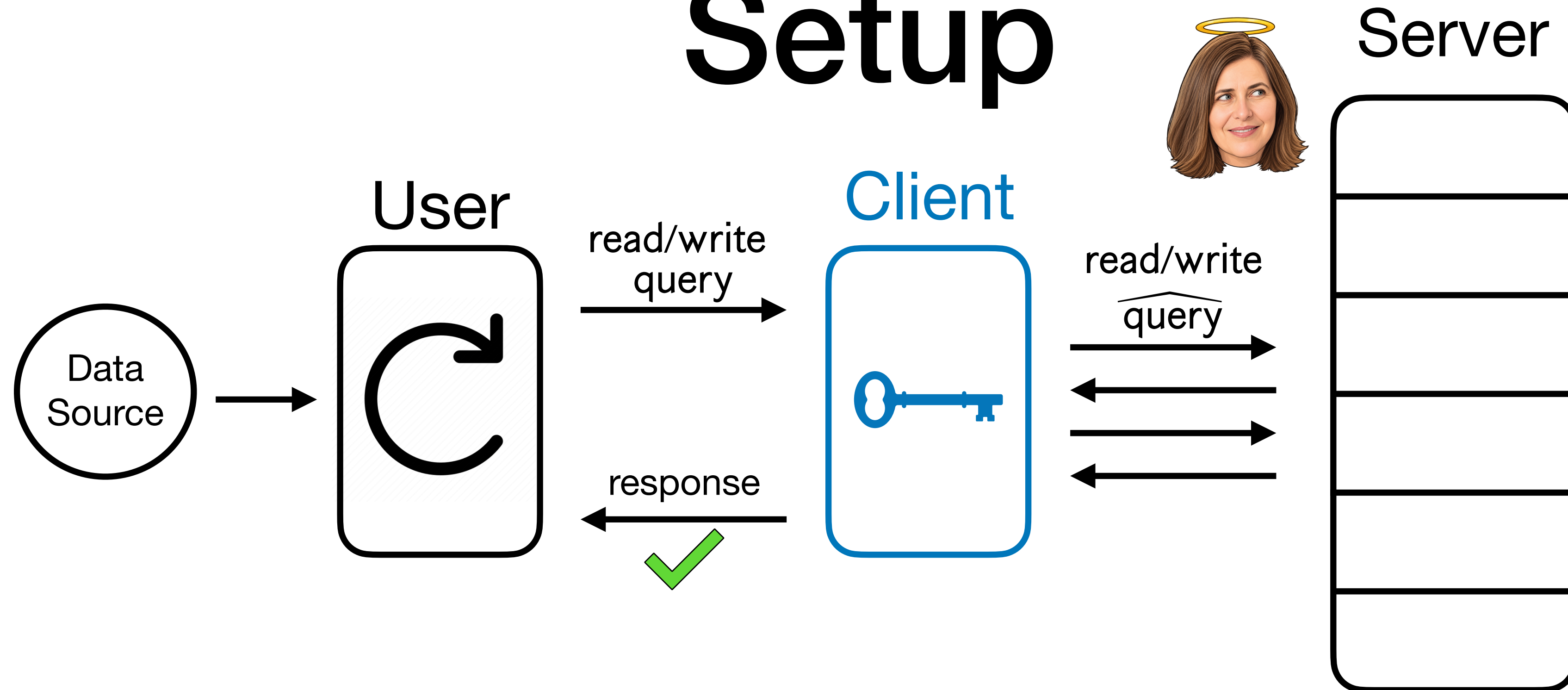


Setup



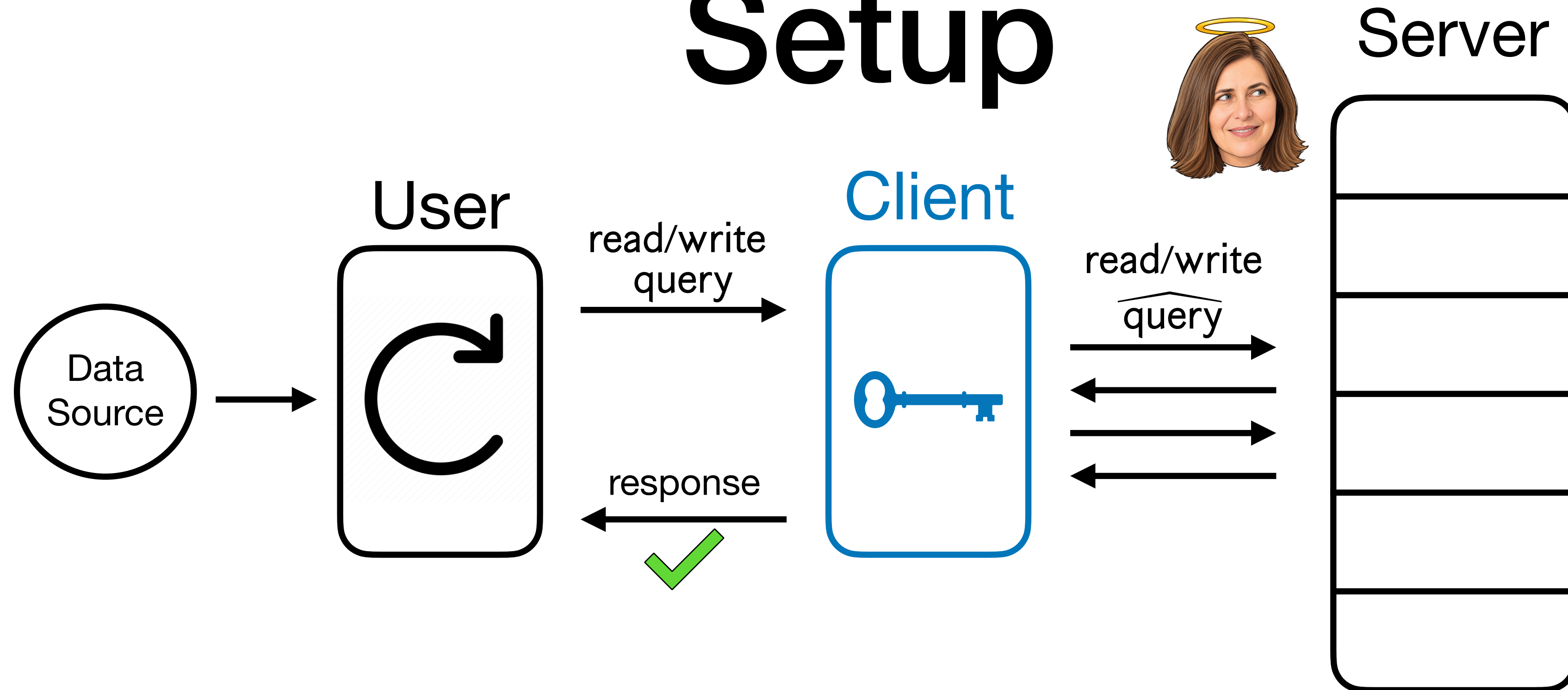
- **(Ensuring integrity) Memory Checking:** For any user queries and all PPT servers, the responses to the user are correct.

Setup



- **(Ensuring integrity) Memory Checking:** For any user queries and all PPT servers, the responses to the user are correct.
- **(Ensuring privacy) Obliviousness:** For an honest server, compiled queries leak *nothing* about the user queries (except for the number of queries):

Setup



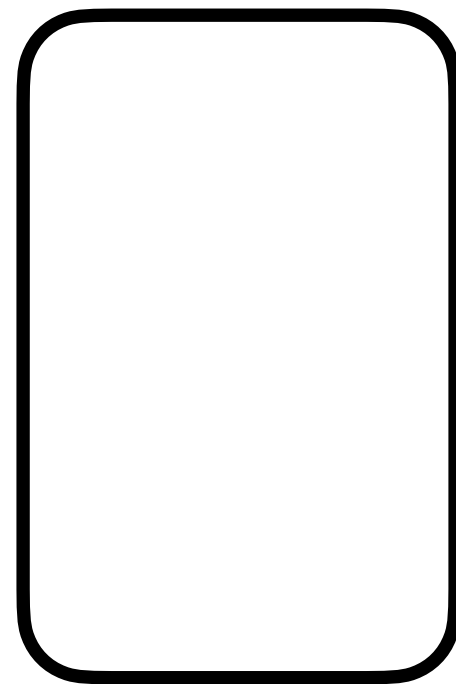
- **(Ensuring integrity) Memory Checking:** For any user queries and all PPT servers, the responses to the user are correct.
- **(Ensuring privacy) Obliviousness:** For an honest server, compiled queries leak *nothing* about the user queries (except for the number of queries):

$$\left\{ \widehat{\text{query}}_i \right\}_{i \in [m]} \approx_{\text{comp}} \left\{ \widehat{\text{query}}'_i \right\}_{i \in [m]}$$

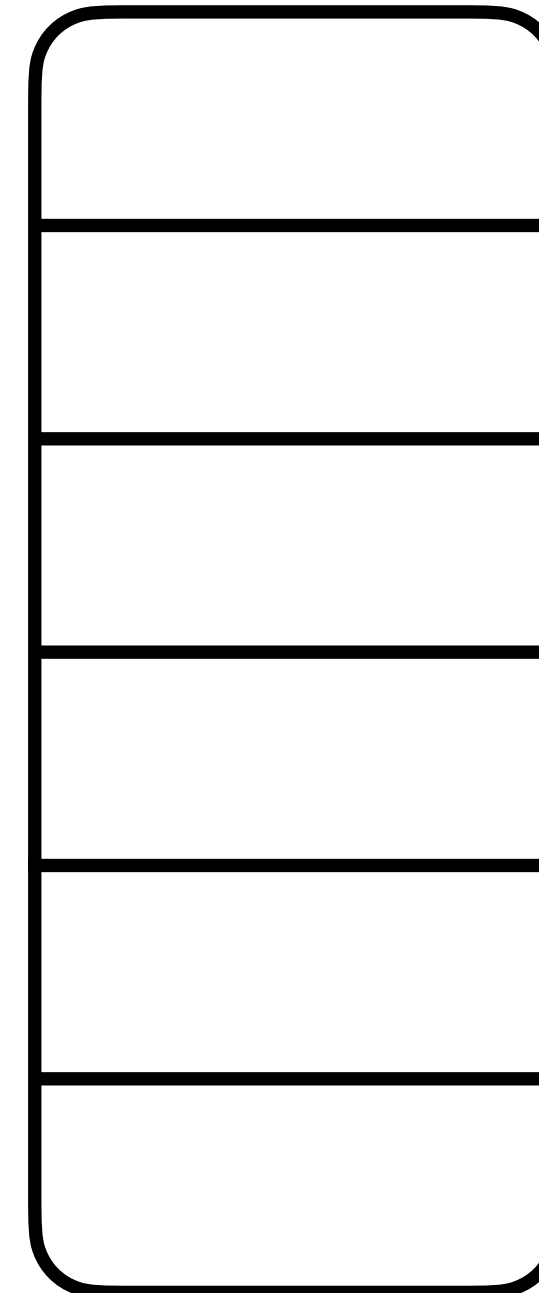
Application: File Storage Platforms

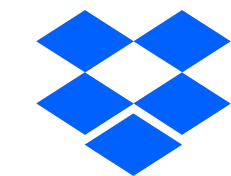


User



Server

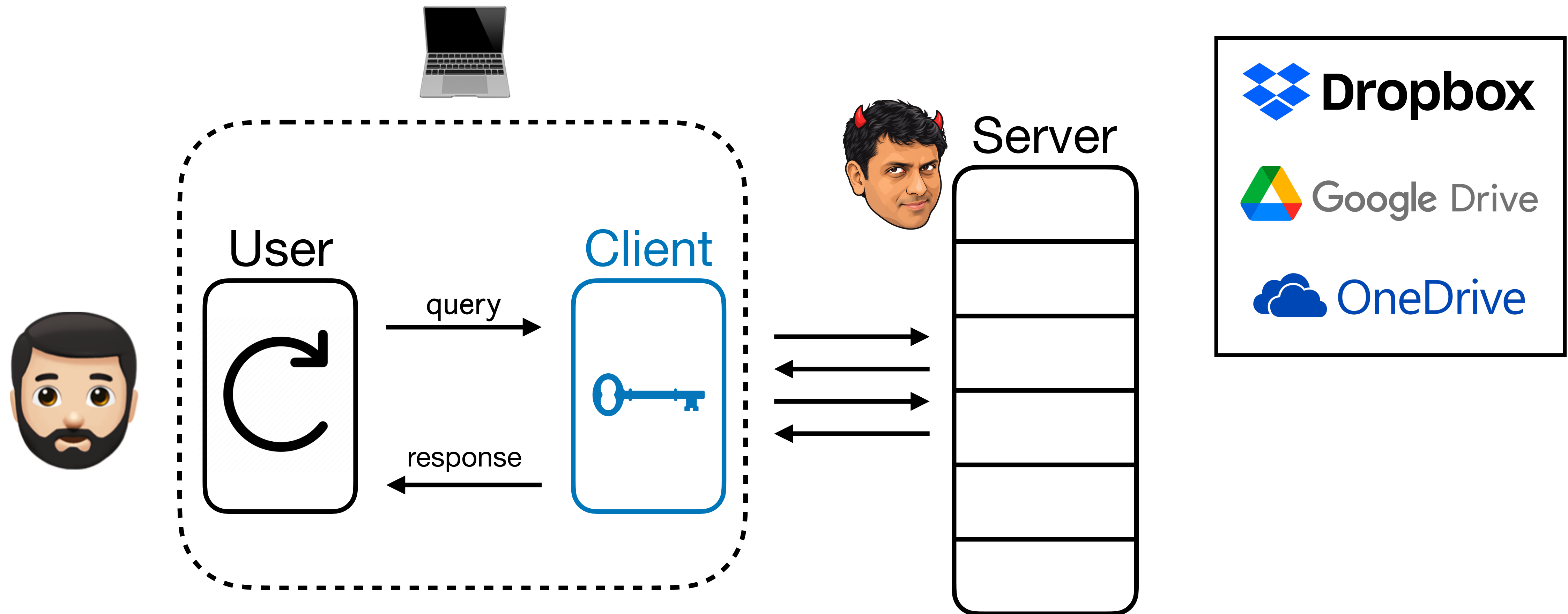


 **Dropbox**

 Google Drive

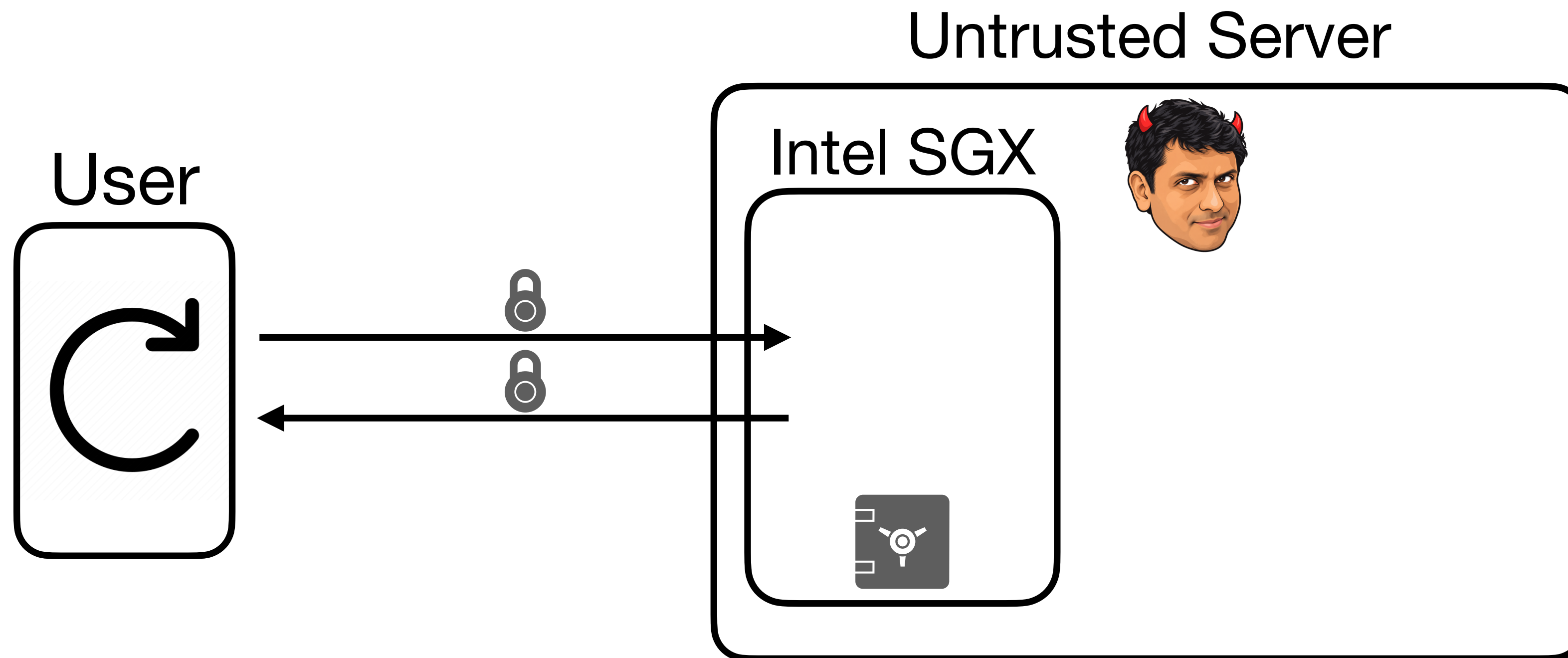
 OneDrive

Application: File Storage Platforms



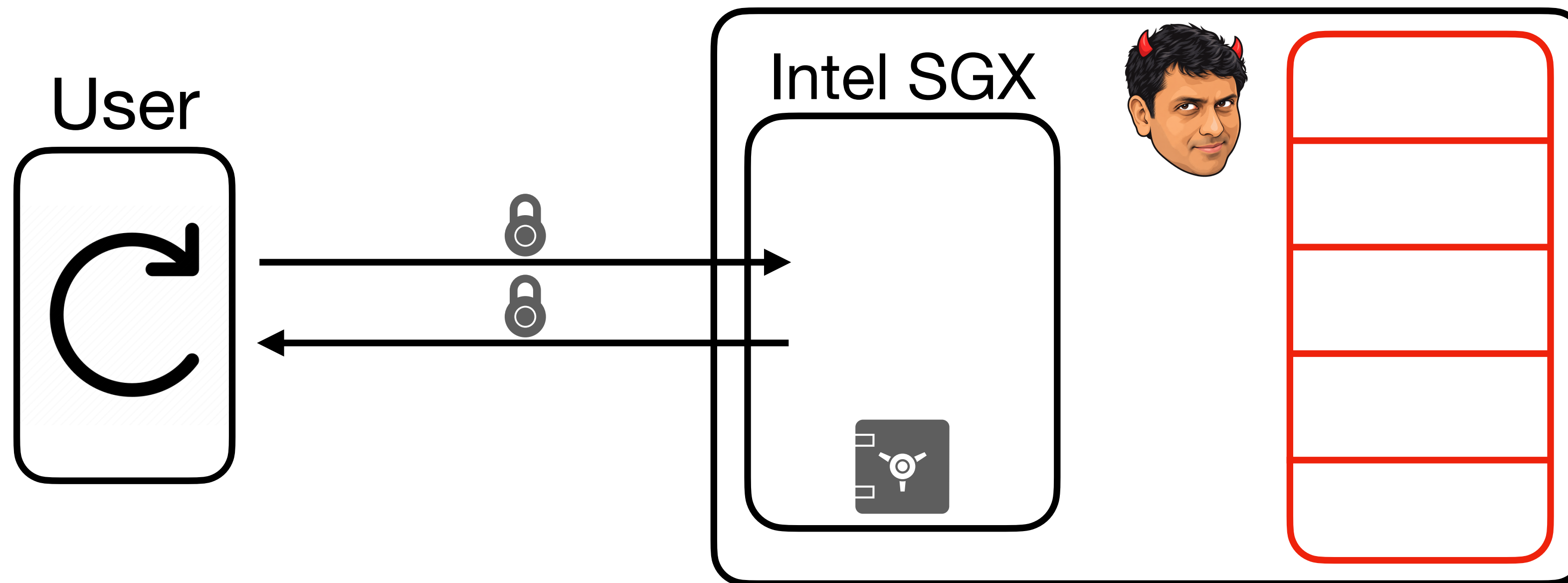
Application: Trusted Execution Environments

- **Trusted Execution Environments** (e.g., Intel SGX) allow users to execute programs securely on untrusted remote servers.



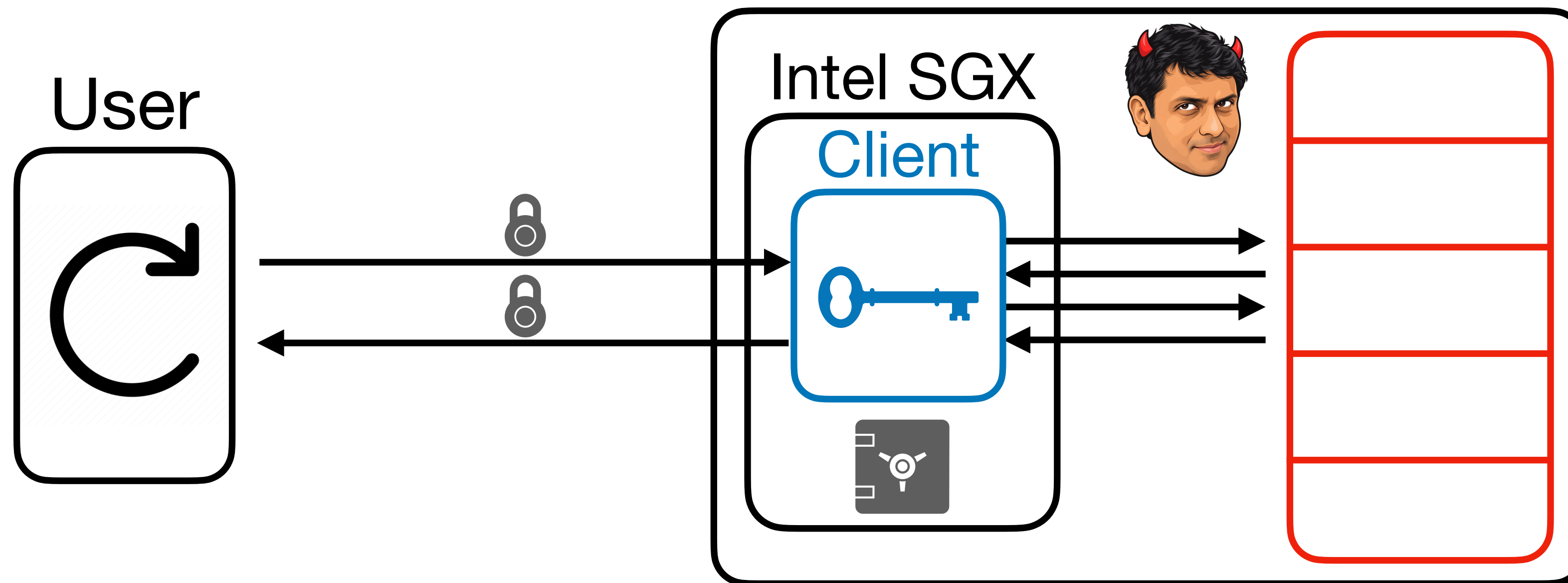
Application: Trusted Execution Environments

- **Trusted Execution Environments** (e.g., Intel SGX) allow users to execute programs securely on untrusted remote servers.
- Some TEEs have tiny internal space. Use untrusted memory within the server!



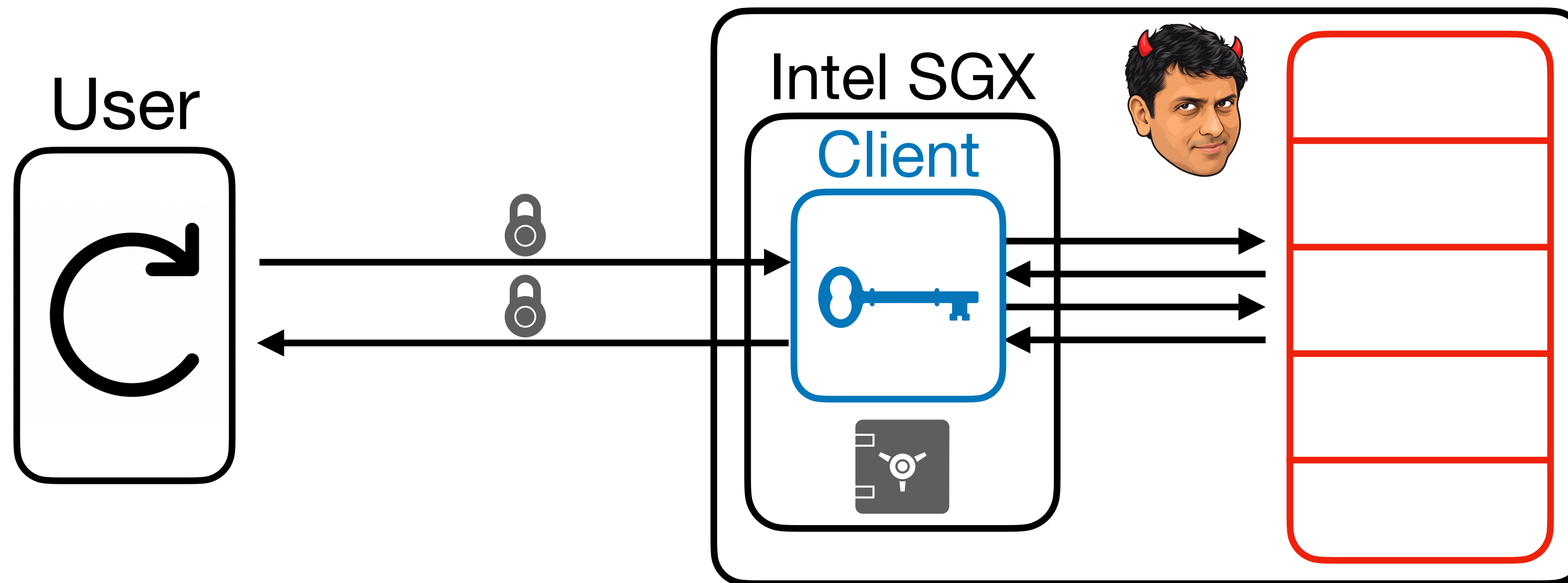
Application: Trusted Execution Environments

- **Trusted Execution Environments** (e.g., Intel SGX) allow users to execute programs securely on untrusted remote servers.
- Some TEEs have tiny internal space. Use untrusted memory within the server!



Application: Trusted Execution Environments

- **Trusted Execution Environments** (e.g., Intel SGX) allow users to execute programs securely on untrusted remote servers.
- Some TEEs have tiny internal space. Use untrusted memory within the server!



- **Real World:** Signal implemented ORAM for private contact discovery!



Efficiency

Efficiency

Two main complexity measures:

Efficiency

Two main complexity measures:

1. **Local Space:** Amount of space the client can store locally (trusted & private).

Efficiency

Two main complexity measures:

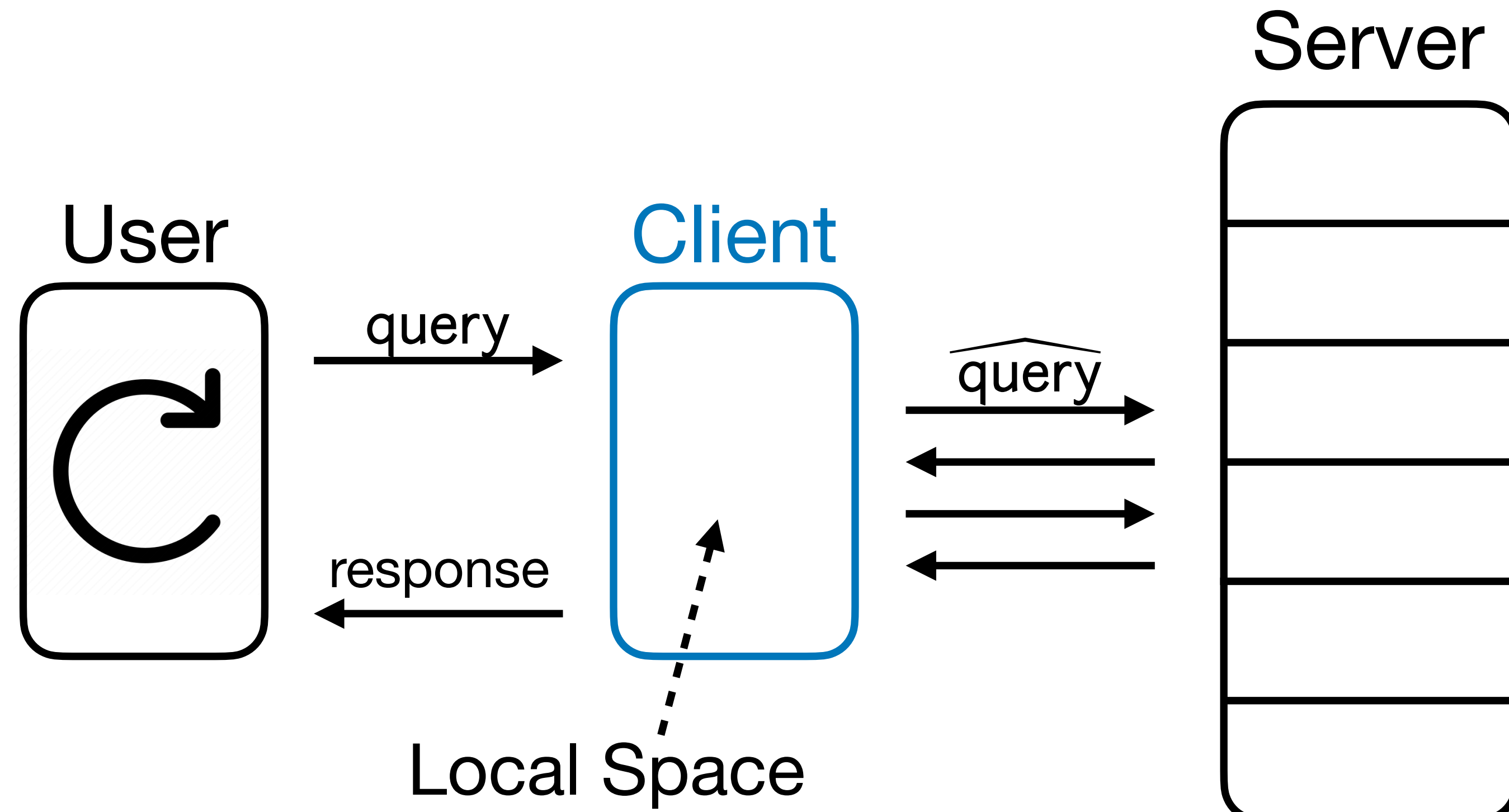
1. **Local Space:** Amount of space the client can store locally (trusted & private).
 - For a RAM with N entries, space N is trivial (can store the full RAM itself).

Efficiency

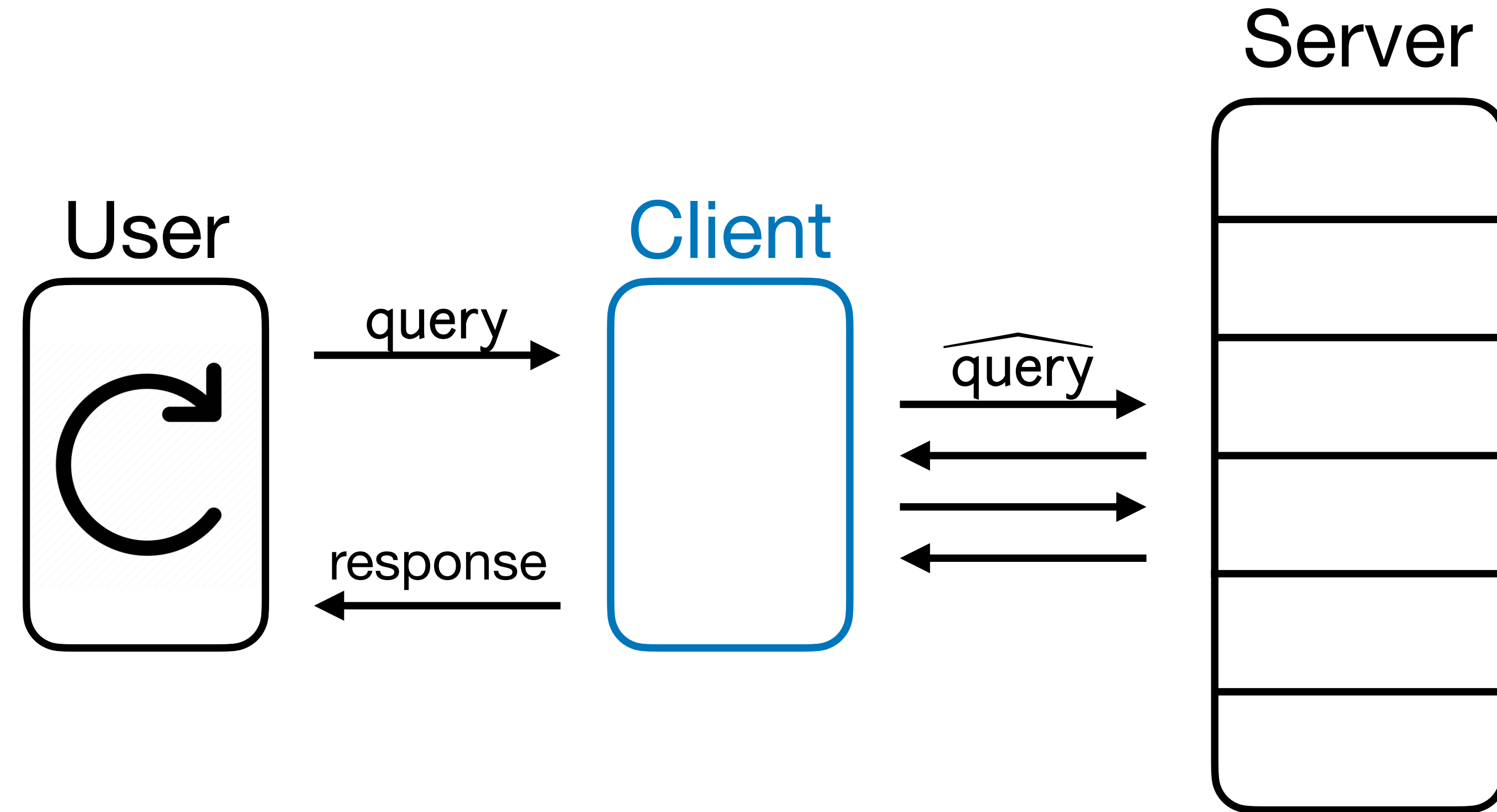
Two main complexity measures:

1. **Local Space:** Amount of space the client can store locally (trusted & private).
 - For a RAM with N entries, space N is trivial (can store the full RAM itself).
 - For the rest of lecture, think space N^ϵ or $\text{polylog}(N)$.

Efficiency

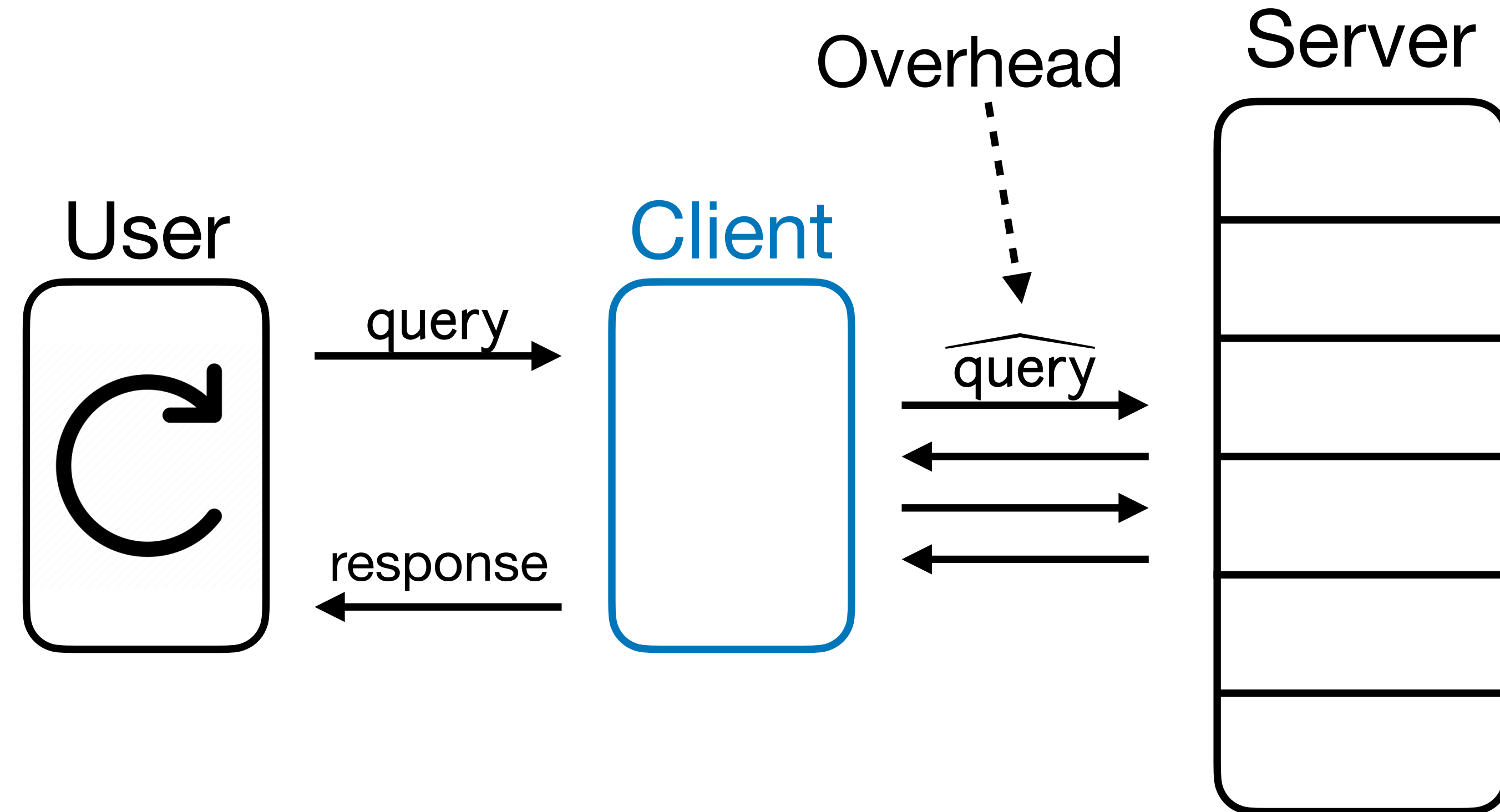


Efficiency



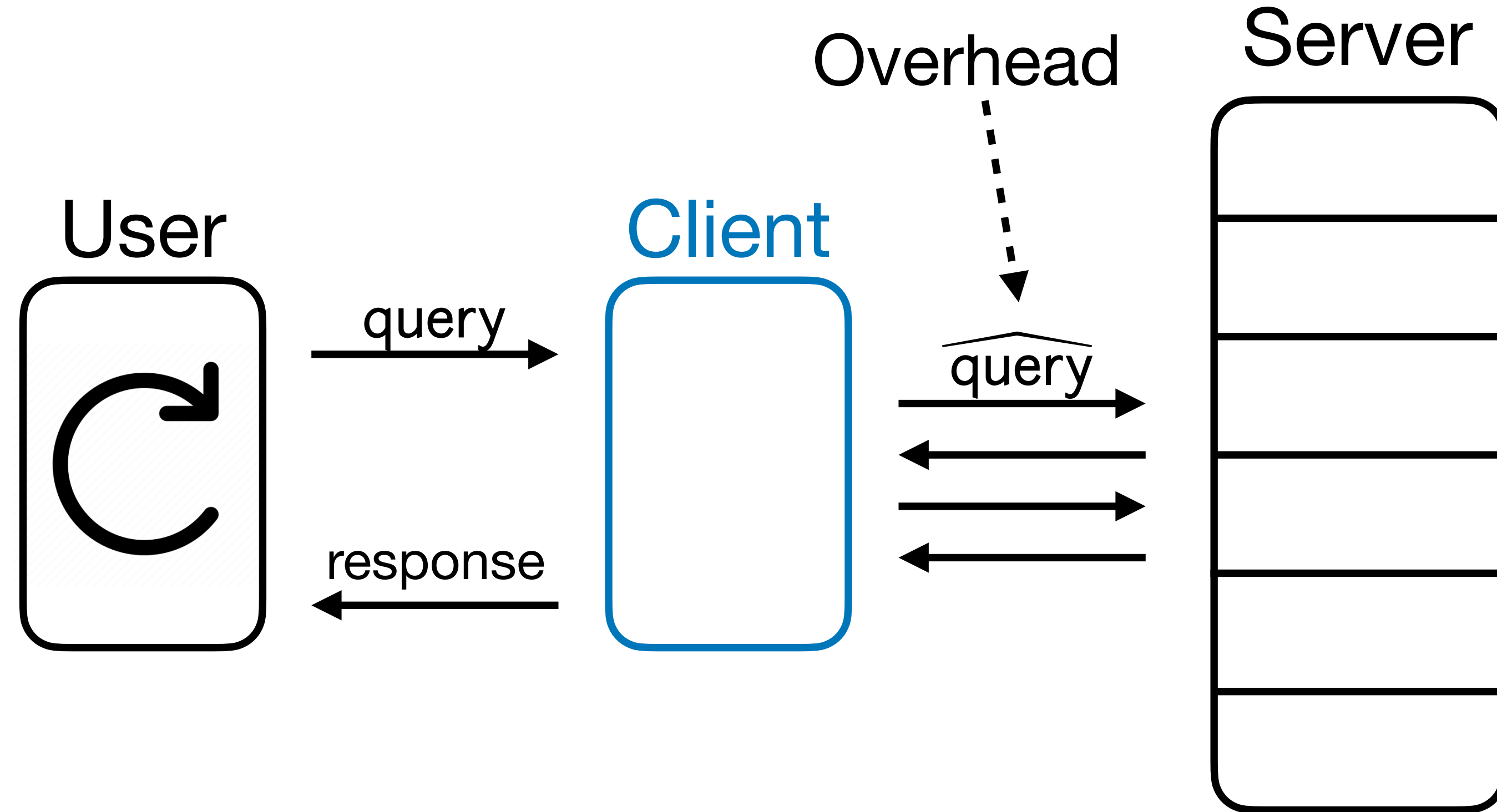
2. **Overhead:** Number of queries made to the server per user query.

Efficiency



2. **Overhead**: Number of queries made to the server per user query.

Efficiency



2. **Overhead**: Number of queries made to the server per user query.

- We want this to be as small as possible!

What's known

What's known

- Memory checker with $O(\log N / \log \log N)$ overhead. [BEGKN '91]
[PT '12]

What's known

- Memory checker with $O(\log N / \log \log N)$ overhead. [BEGKN '91]
[PT '12]
- **Matching lower bound (unconditional!)** [DNRV '09] [BKV '23]

What's known

- Memory checker with $O(\log N / \log \log N)$ overhead. [BEGKN '91] [PT '12]
- **Matching lower bound (unconditional!)** [DNRV '09] [BKV '23]
- ORAM construction with $O(\log N)$ overhead. [AKLNPS '20]

What's known

- Memory checker with $O(\log N / \log \log N)$ overhead. [BEGKN '91] [PT '12]
- **Matching lower bound (unconditional!)** [DNRV '09] [BKV '23]
- ORAM construction with $O(\log N)$ overhead. [AKLNPS '20]
- **Matching lower bound (unconditional!)** [Goldreich '87] [LN '18]

Today

Today

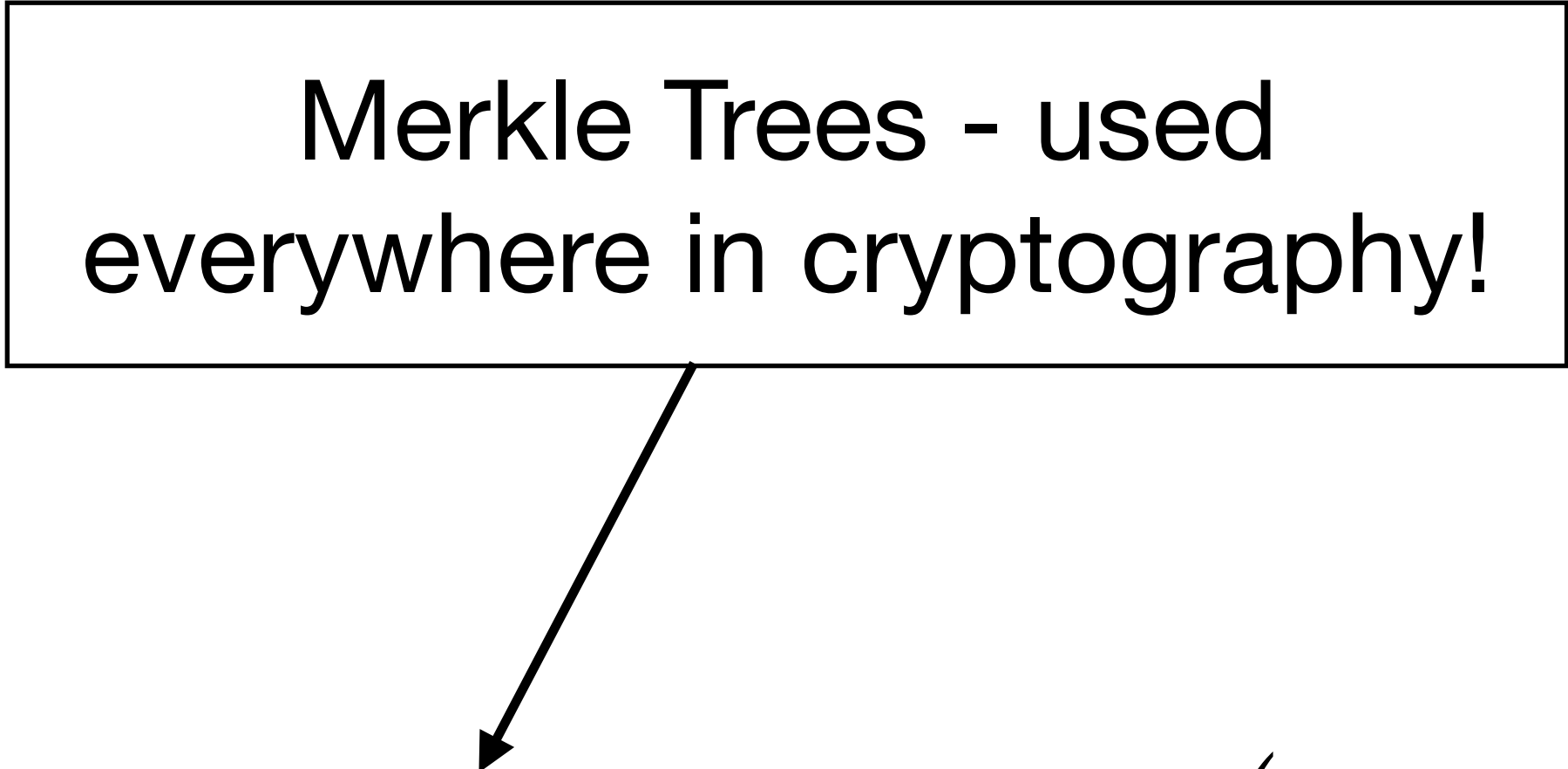
- Today, we'll see:

Today

- Today, we'll see:
 - Memory checker construction with $O(\log N)$ overhead.

Today

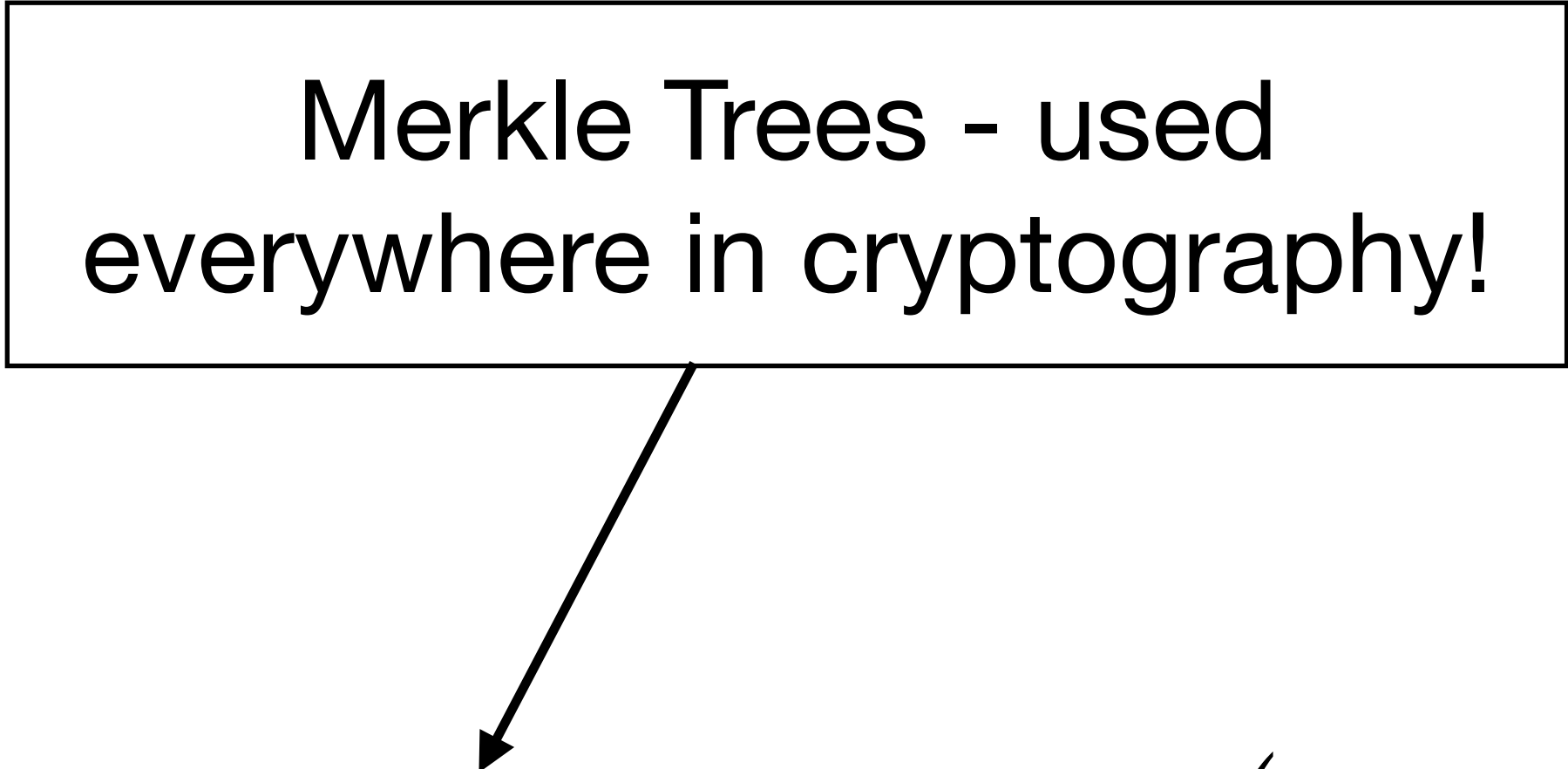
Merkle Trees - used
everywhere in cryptography!



- Today, we'll see:
- Memory checker construction with $O(\log N)$ overhead.

Today

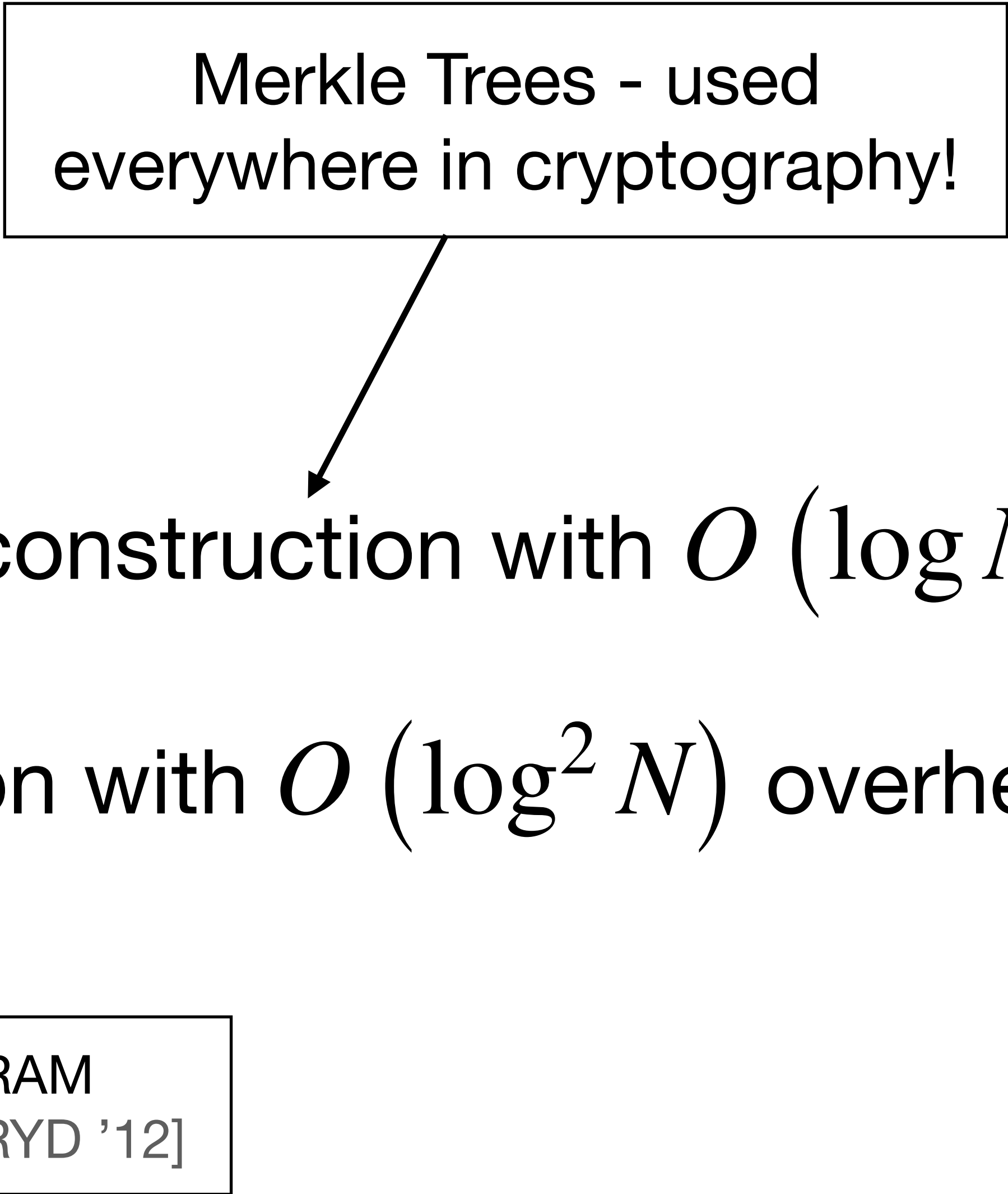
Merkle Trees - used
everywhere in cryptography!



- Today, we'll see:
 - Memory checker construction with $O(\log N)$ overhead.
 - ORAM construction with $O(\log^2 N)$ overhead.

Today

Merkle Trees - used
everywhere in cryptography!



```
graph TD; A[Merkle Trees - used everywhere in cryptography!] --> B[Memory checker construction with O(log N) overhead.]; A --> C[ORAM construction with O(log^2 N) overhead.]; D[Path ORAM [SvDSHCFRYD '12]] --> C;
```

- Today, we'll see:
 - Memory checker construction with $O(\log N)$ overhead.
 - ORAM construction with $O(\log^2 N)$ overhead.

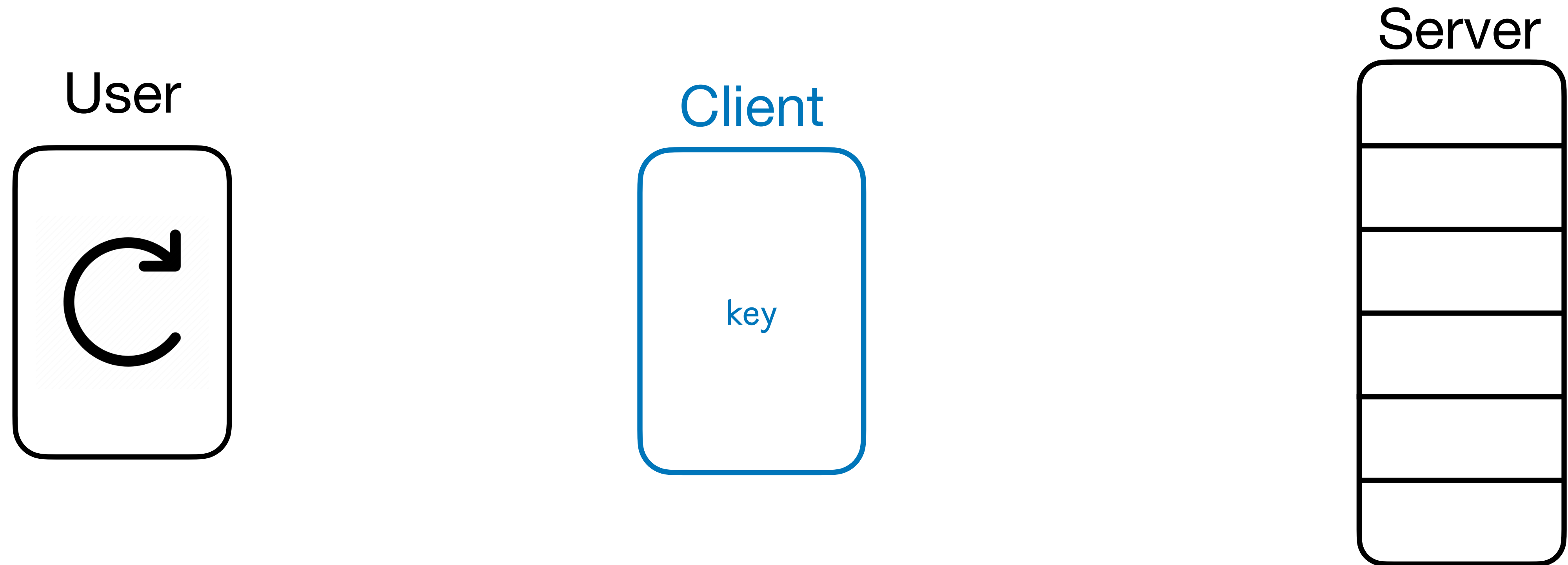
Path ORAM
[SvDSHCFRYD '12]

Memory Checking

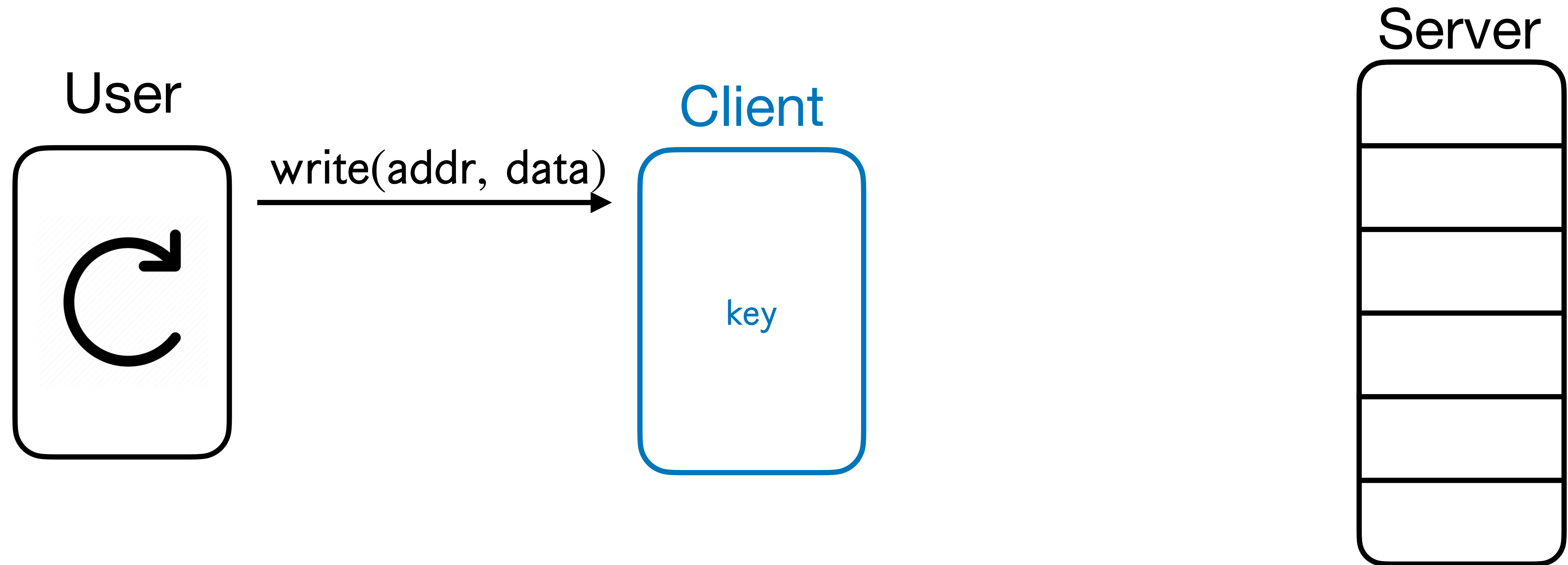
Memory Checking

Wait, does authentication solve the integrity issue?
(e.g., MACs, digital signatures)

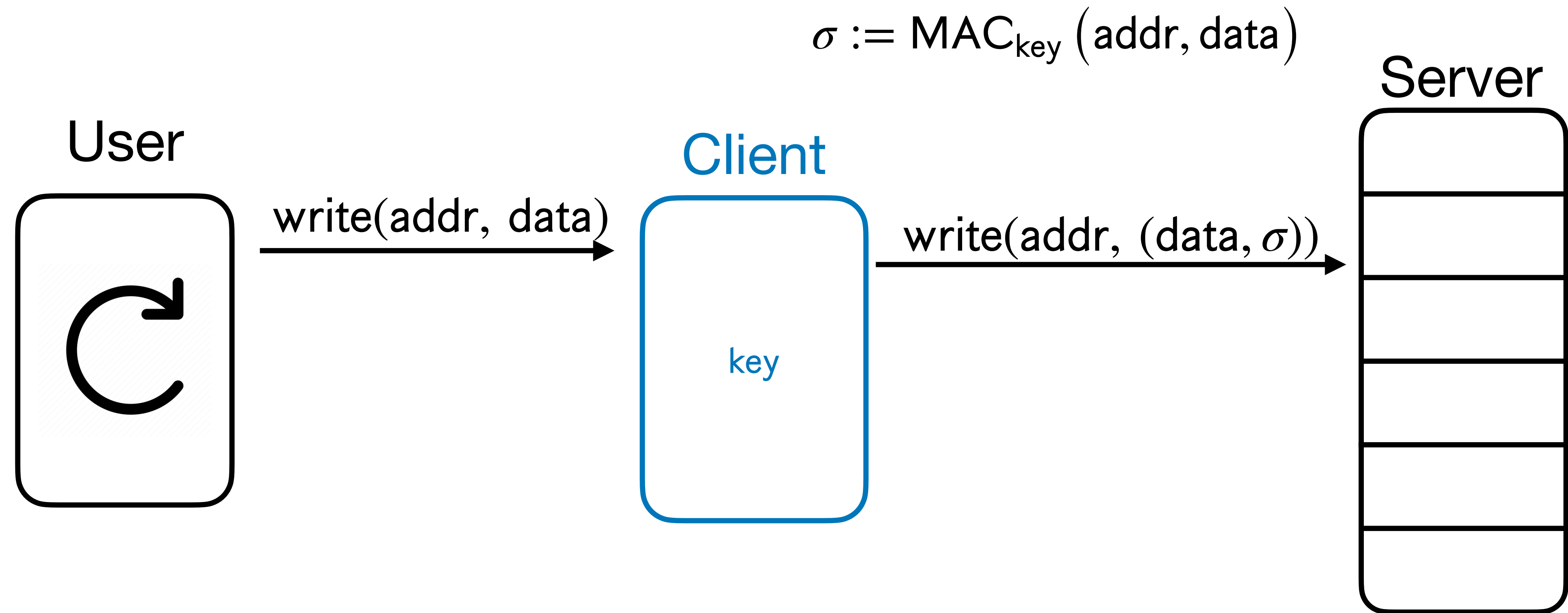
MACs for Memory Checking?



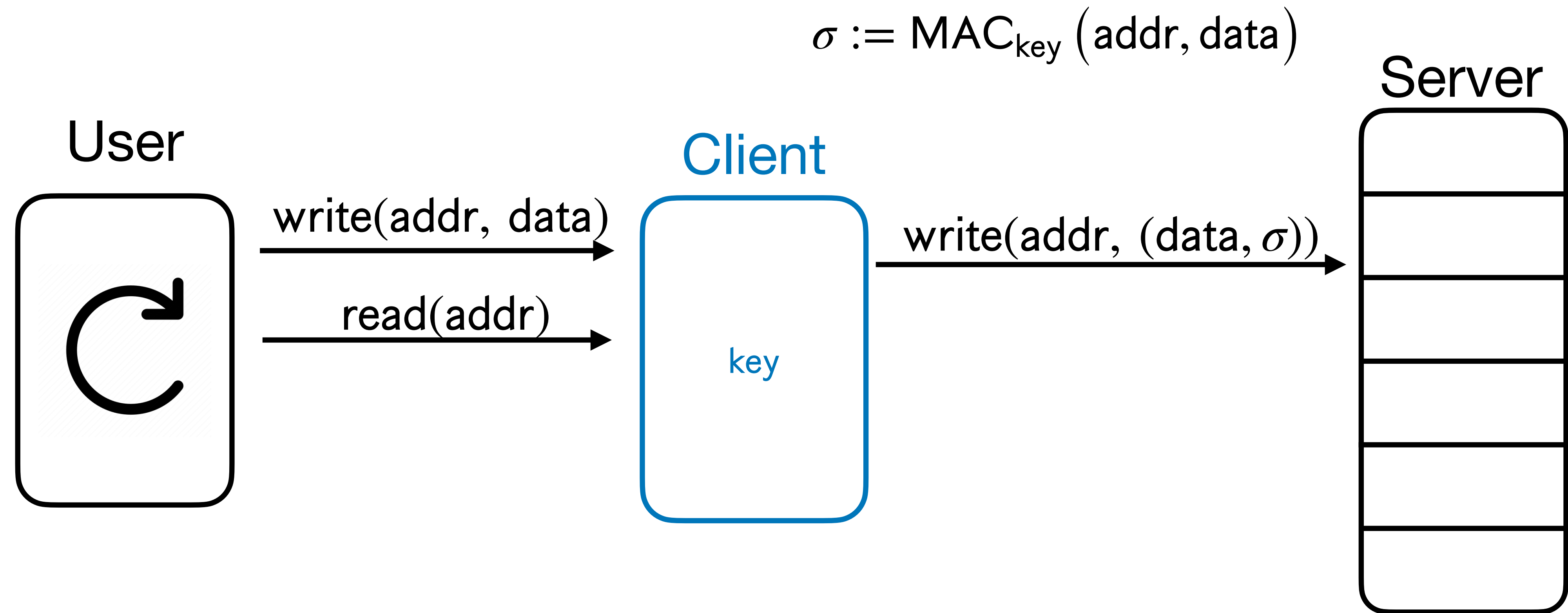
MACs for Memory Checking?



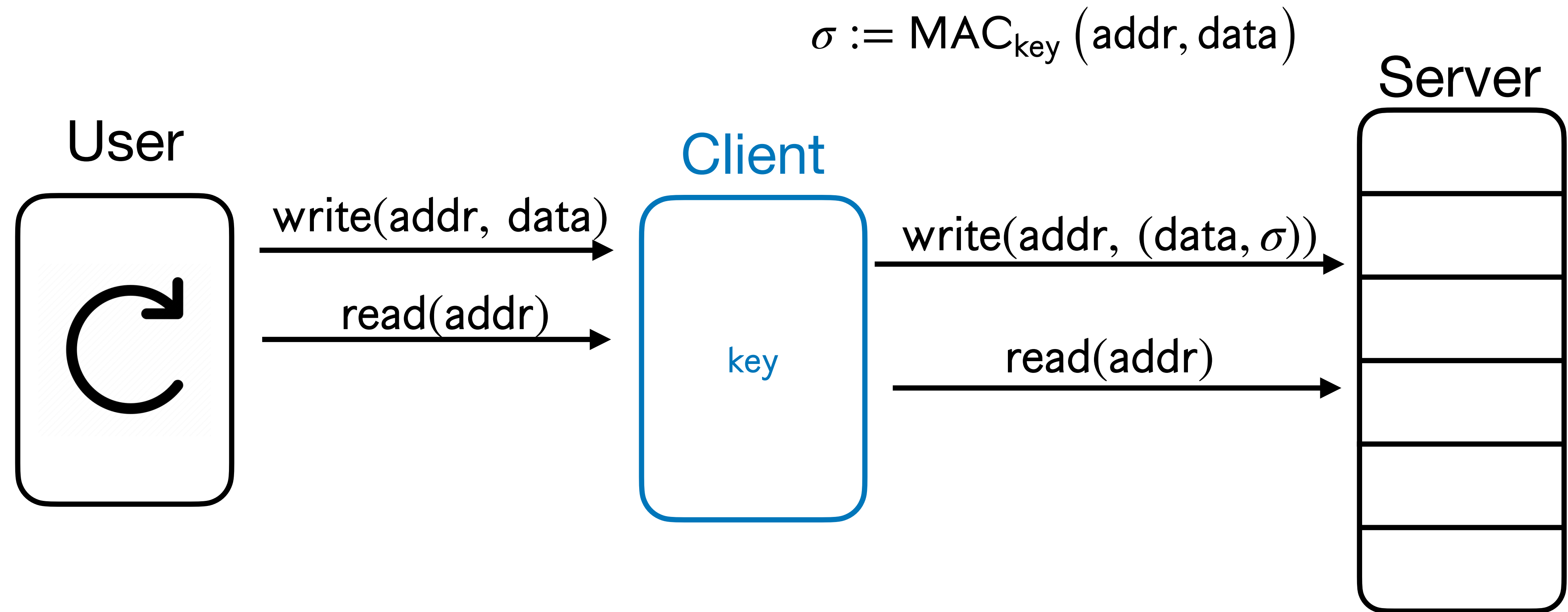
MACs for Memory Checking?



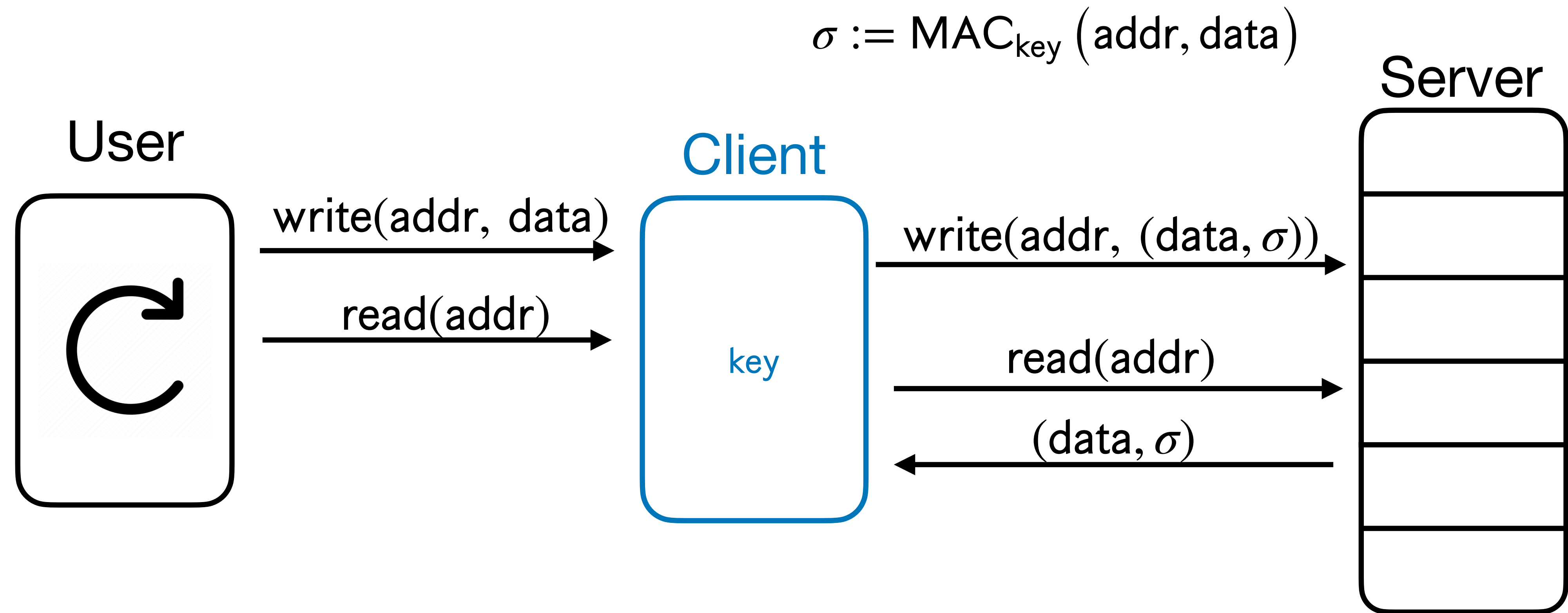
MACs for Memory Checking?



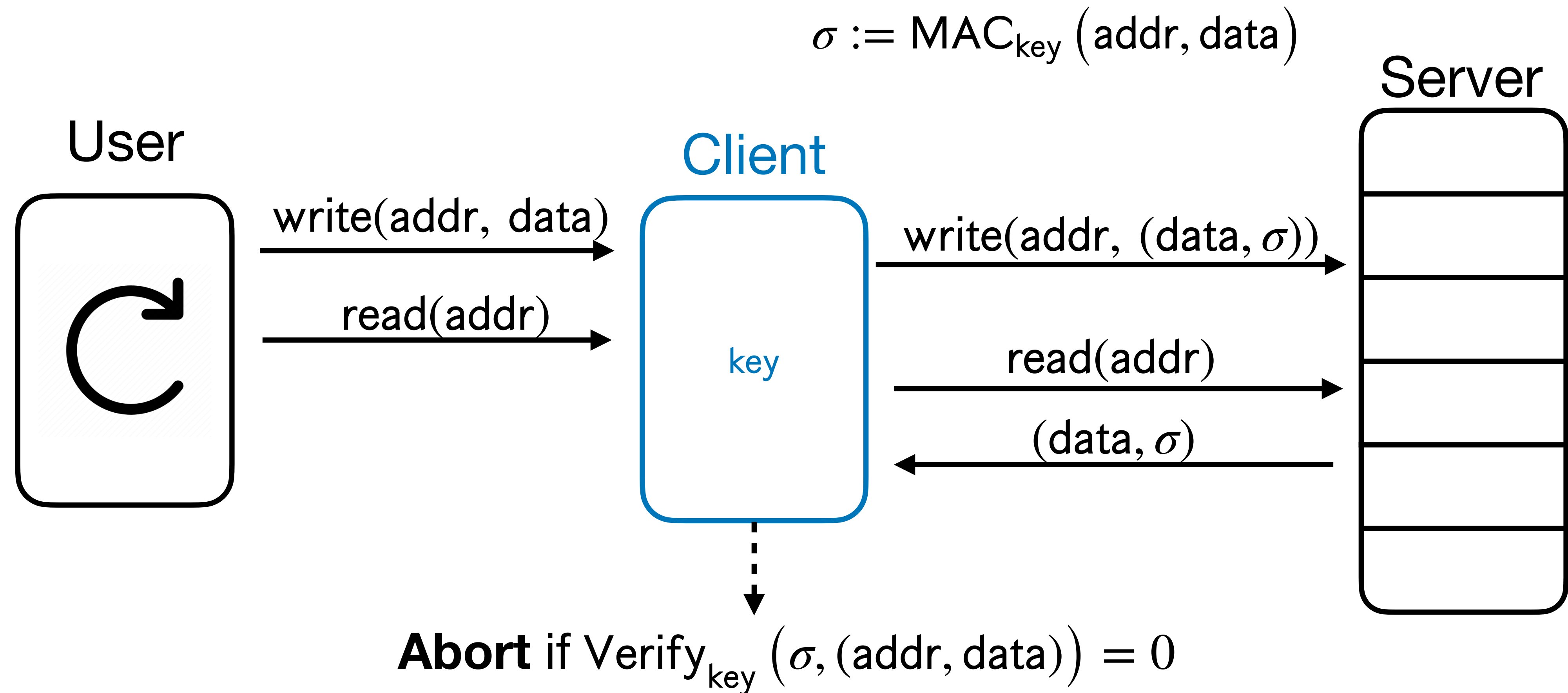
MACs for Memory Checking?



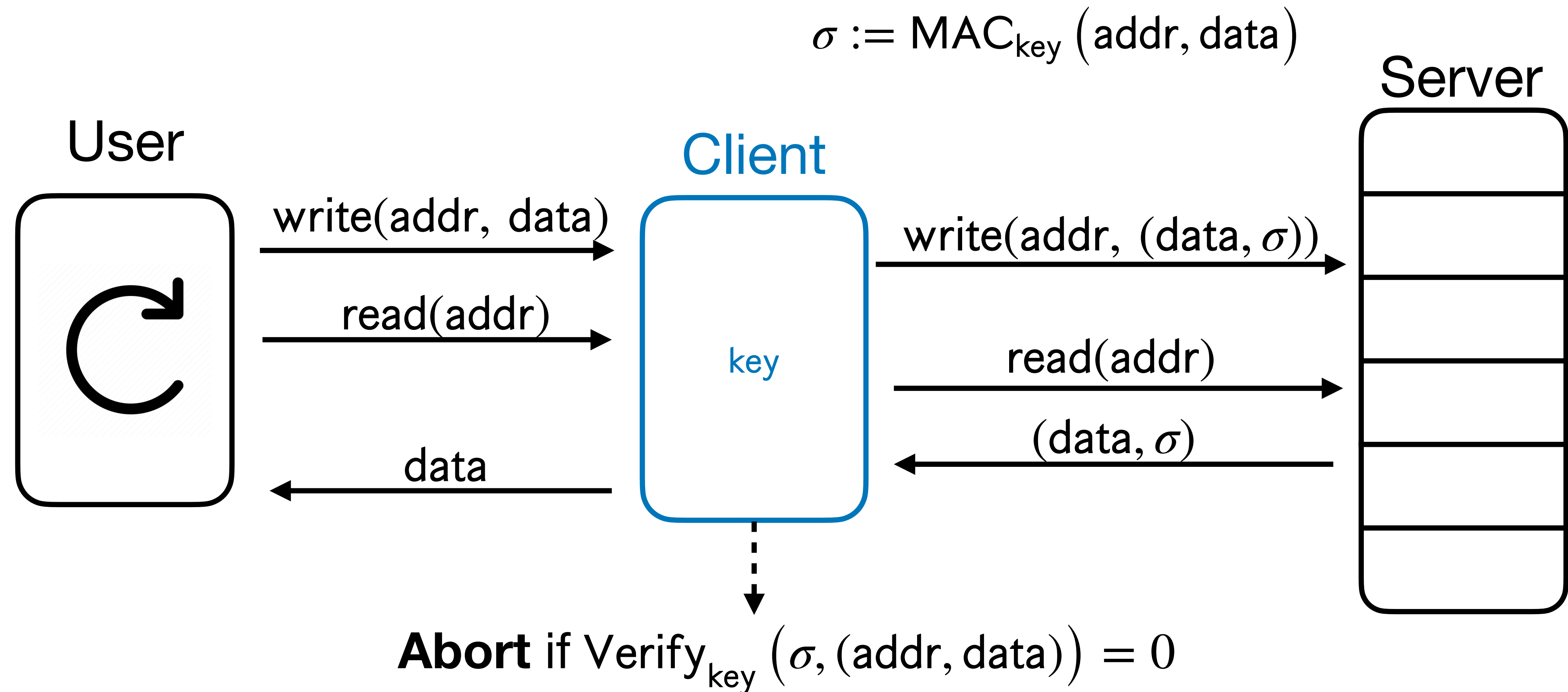
MACs for Memory Checking?



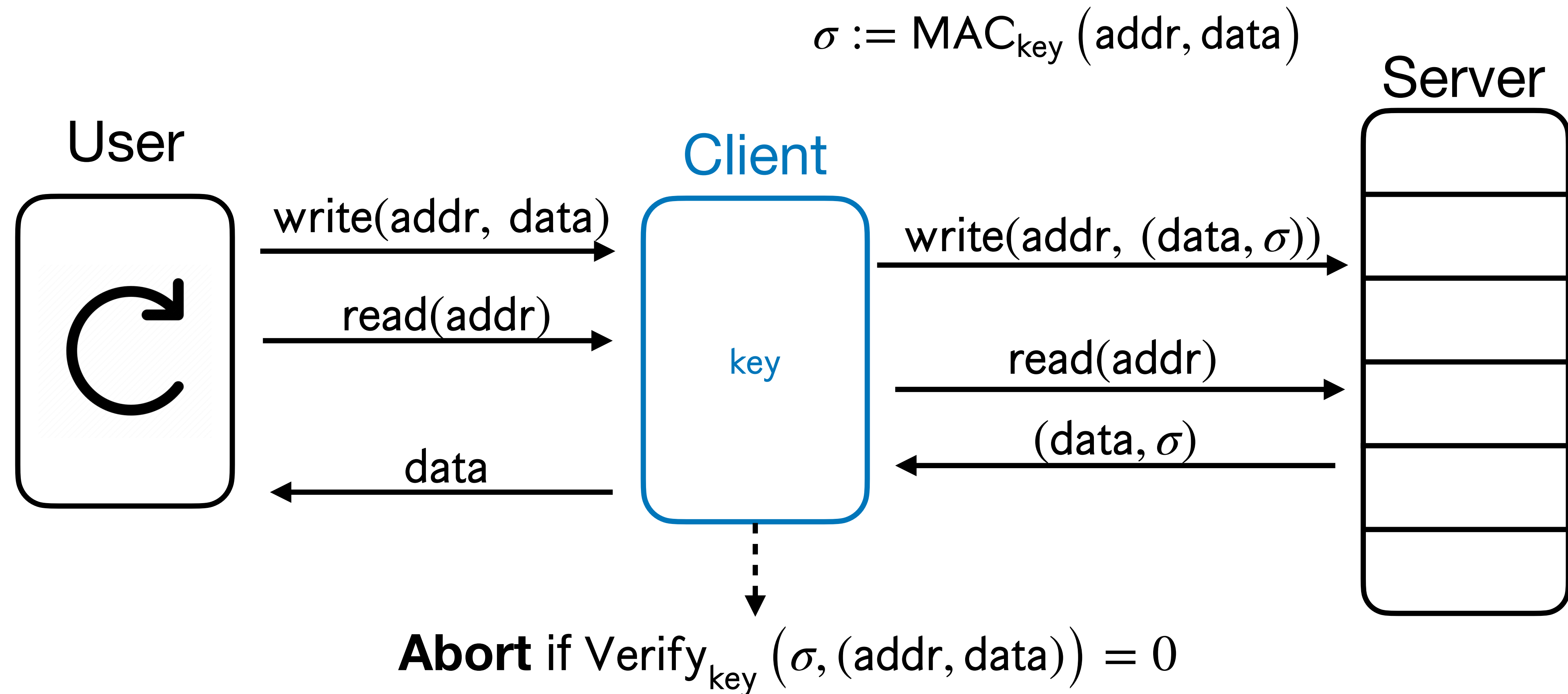
MACs for Memory Checking?



MACs for Memory Checking?



MACs for Memory Checking?



Does this work? What does it prevent?

Replay Attacks

Replay Attacks

- MACs prevent all (efficient) adversarial attacks except for **replay attacks**.

Replay Attacks

- MACs prevent all (efficient) adversarial attacks except for **replay attacks**.
- **Stale** values of (data, σ) will still pass MAC verification check.

Replay Attacks

- MACs prevent all (efficient) adversarial attacks except for **replay attacks**.
 - **Stale** values of (data, σ) will still pass MAC verification check.
- Natural idea: add counters/time-stamps inside MACs.

Replay Attacks

- MACs prevent all (efficient) adversarial attacks except for **replay attacks**.
 - **Stale** values of (data, σ) will still pass MAC verification check.
- Natural idea: add counters/time-stamps inside MACs.
- **Fatal issue**: No way to check counters/time-stamps in low space.

Merkle Trees

Merkle Trees

- Totally different approach.

Merkle Trees

- Totally different approach.
- How can we “compress” the memory and save that locally?

Merkle Trees

- Totally different approach.
- How can we “compress” the memory and save that locally?
- **Natural idea:** Collision-Resistant Hash Functions (CRHFs)

Merkle Trees

- Totally different approach.
- How can we “compress” the memory and save that locally?
- **Natural idea:** Collision-Resistant Hash Functions (CRHFs)
- **Hope:** Store hash locally, and check correctness of the hash.

Merkle Trees

- Totally different approach.
- How can we “compress” the memory and save that locally?
- **Natural idea:** Collision-Resistant Hash Functions (CRHFs)
- **Hope:** Store hash locally, and check correctness of the hash.
- Throughout, let $H : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be a CRHF with $\lambda \ll N$.

Merkle Trees

Merkle Trees

- **Option 1:** For all $i \in [N]$, locally store $H(\text{data}_i)$.

Merkle Trees

- **Option 1:** For all $i \in [N]$, locally store $H(\text{data}_i)$.
 - Reads and writes have overhead 1!

Merkle Trees

- **Option 1:** For all $i \in [N]$, locally store $H(\text{data}_i)$.
 - Reads and writes have overhead 1!
 - Local storage is smaller than database, but still $\Omega(N)$.

Merkle Trees

- **Option 1:** For all $i \in [N]$, locally store $H(\text{data}_i)$.
 - Reads and writes have overhead 1!
 - Local storage is smaller than database, but still $\Omega(N)$.
- **Option 2:** Locally store $H(\text{data}_1, \dots, \text{data}_N)$.

Merkle Trees

- **Option 1:** For all $i \in [N]$, locally store $H(\text{data}_i)$.
 - Reads and writes have overhead 1!
 - Local storage is smaller than database, but still $\Omega(N)$.
- **Option 2:** Locally store $H(\text{data}_1, \dots, \text{data}_N)$.
 - Local storage is now very small!

Merkle Trees

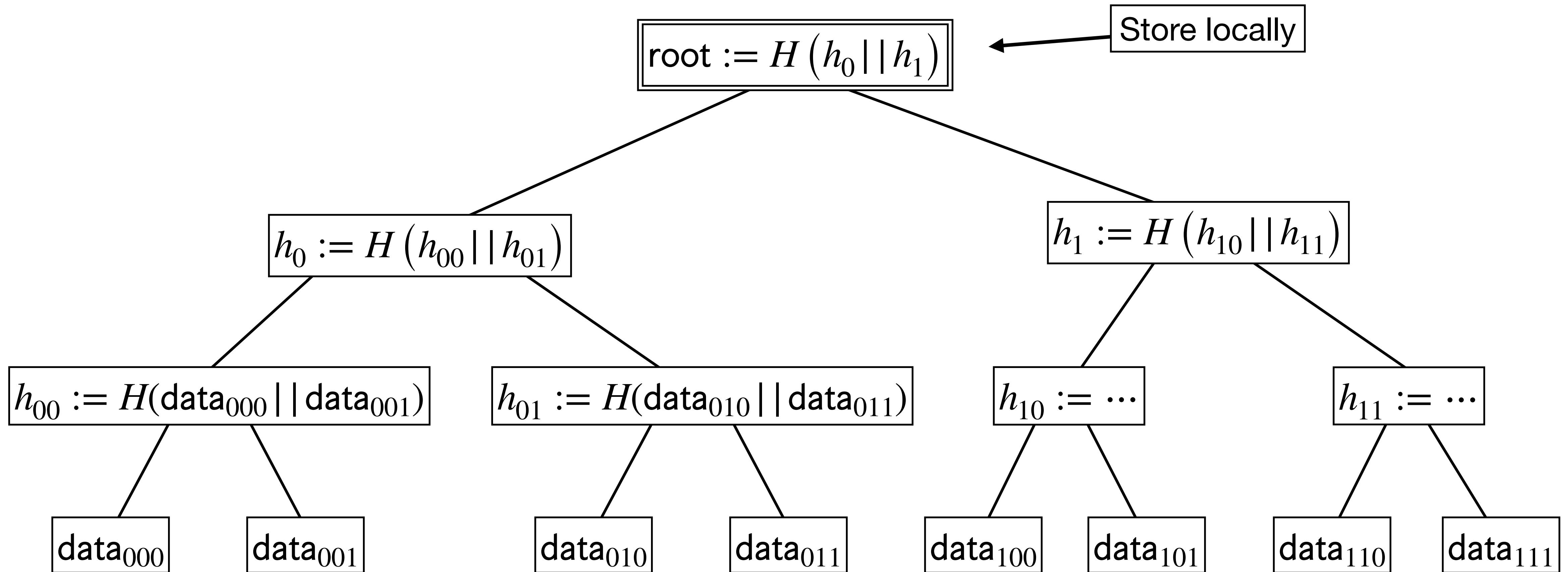
- **Option 1:** For all $i \in [N]$, locally store $H(\text{data}_i)$.
 - Reads and writes have overhead 1!
 - Local storage is smaller than database, but still $\Omega(N)$.
- **Option 2:** Locally store $H(\text{data}_1, \dots, \text{data}_N)$.
 - Local storage is now very small!
 - Verifying reads and writes are expensive, overhead $\Theta(N)$.

Merkle Trees

- **Option 1:** For all $i \in [N]$, locally store $H(\text{data}_i)$.
 - Reads and writes have overhead 1!
 - Local storage is smaller than database, but still $\Omega(N)$.
- **Option 2:** Locally store $H(\text{data}_1, \dots, \text{data}_N)$.
 - Local storage is now very small!
 - Verifying reads and writes are expensive, overhead $\Theta(N)$.
- **Option 3:** Trade off between the two options with a binary tree!

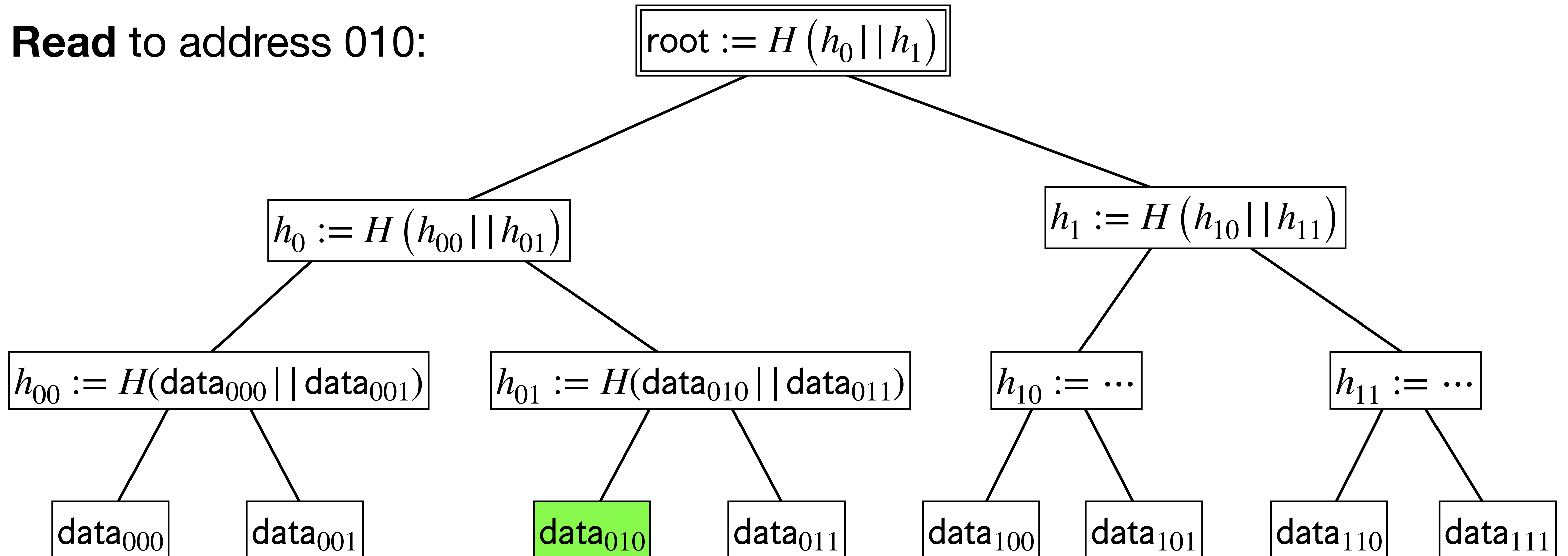
Merkle Trees

(Here, $N = 8$.)



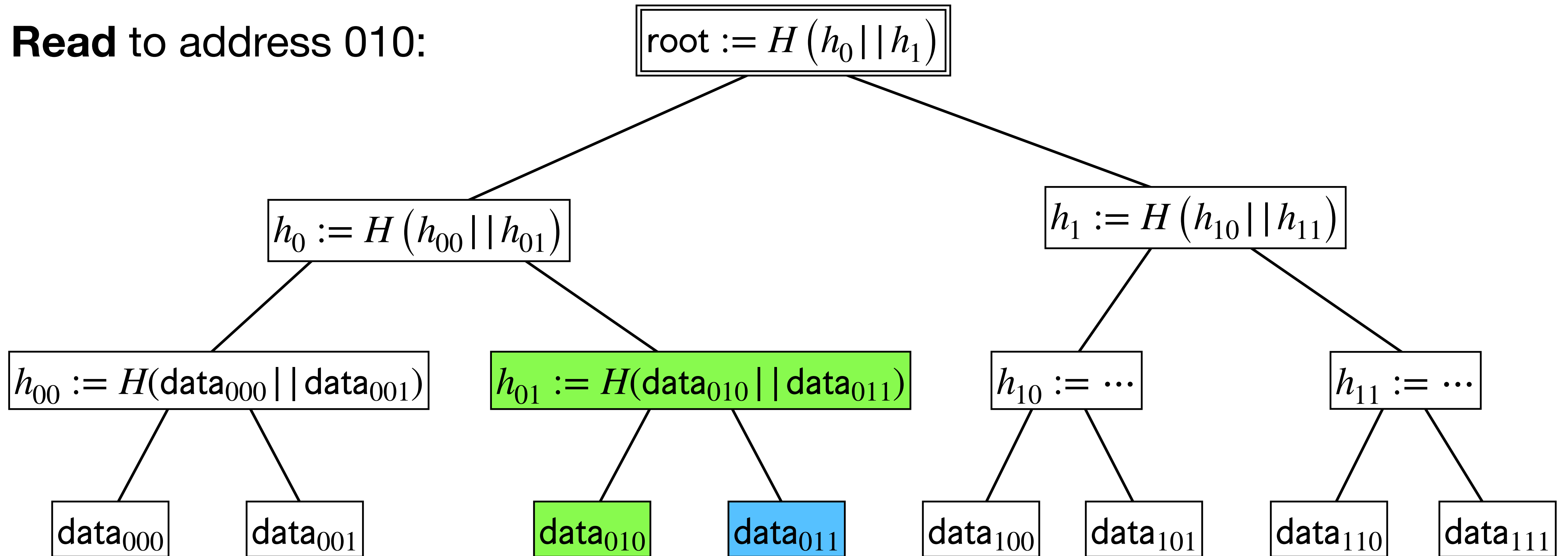
Merkle Trees

Read to address 010:



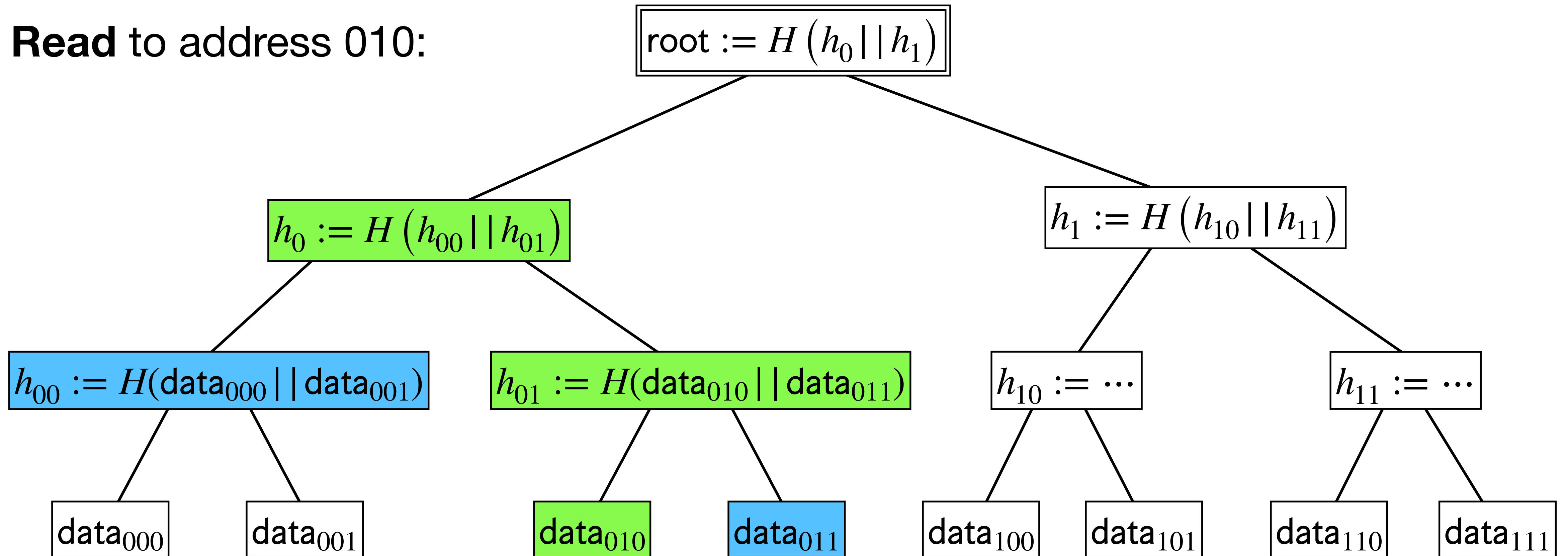
Merkle Trees

Read to address 010:



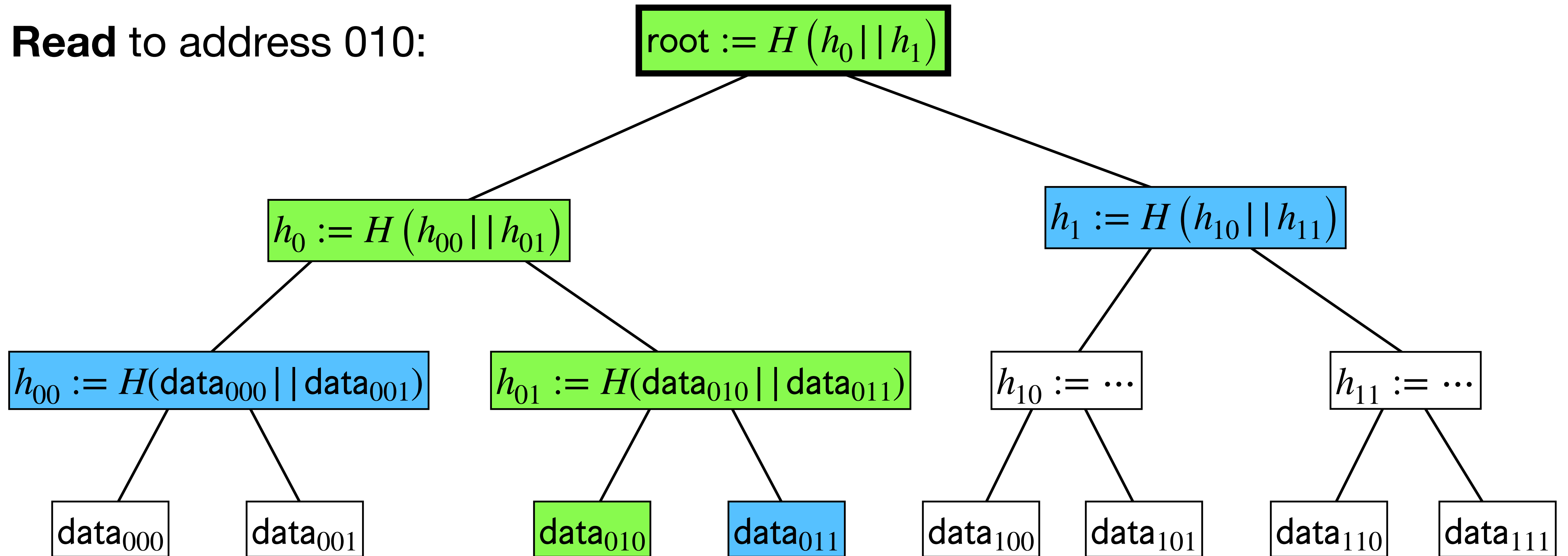
Merkle Trees

Read to address 010:



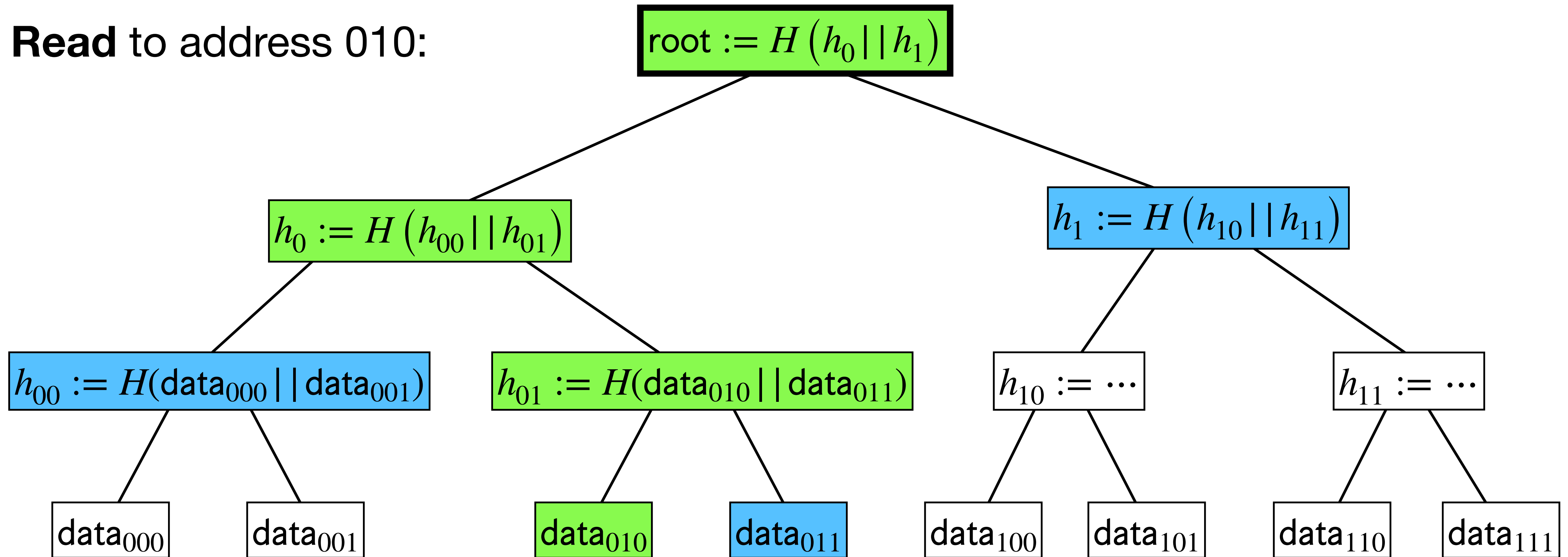
Merkle Trees

Read to address 010:



Merkle Trees

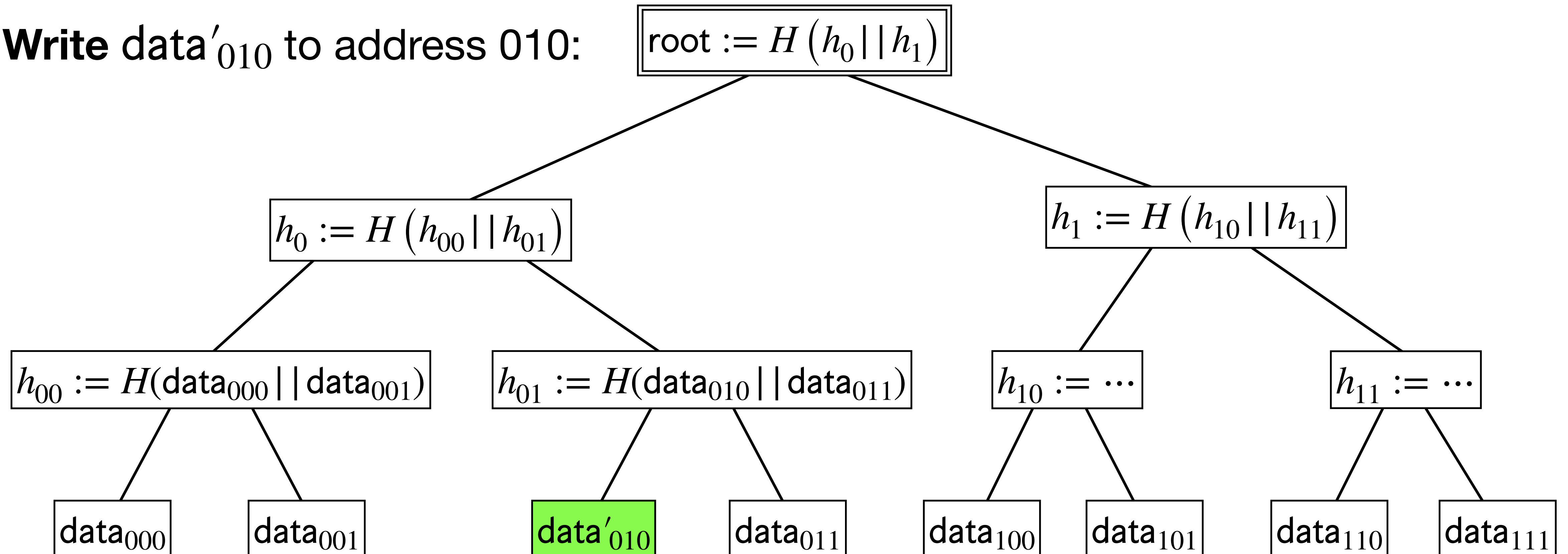
Read to address 010:



If all hashes to root are consistent, **return** data_{010} . Otherwise, **abort**.

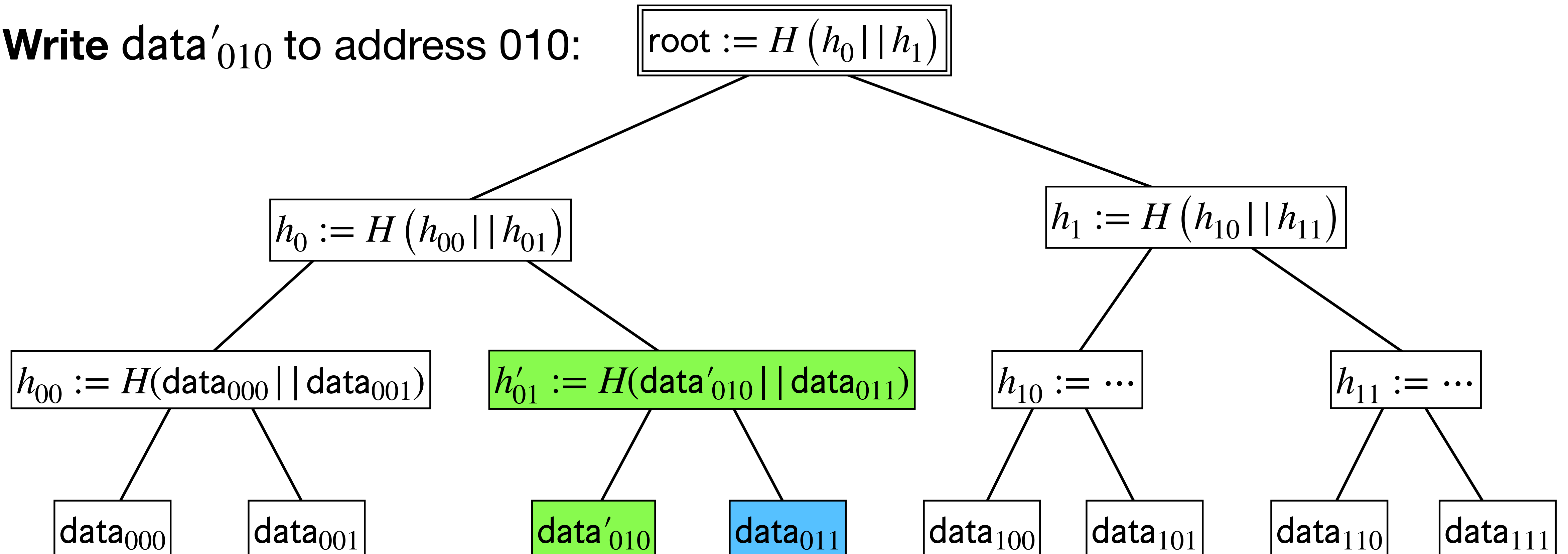
Merkle Trees

Write data'_{010} to address 010:



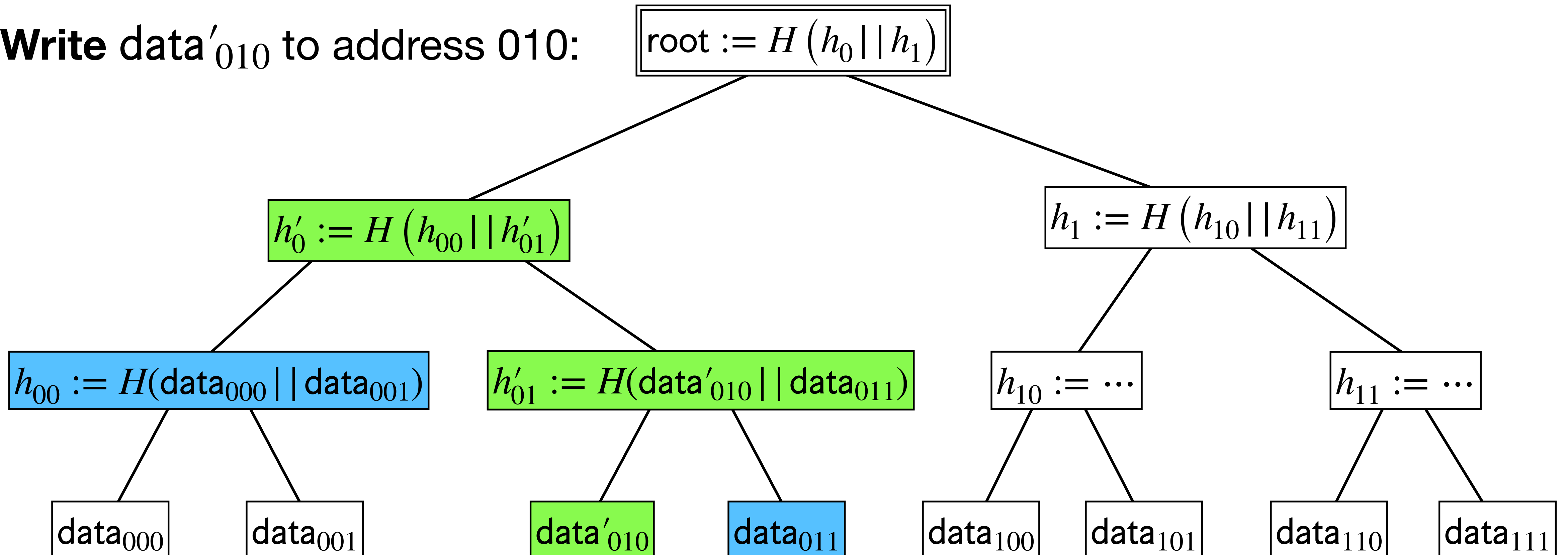
Merkle Trees

Write data'_{010} to address 010:



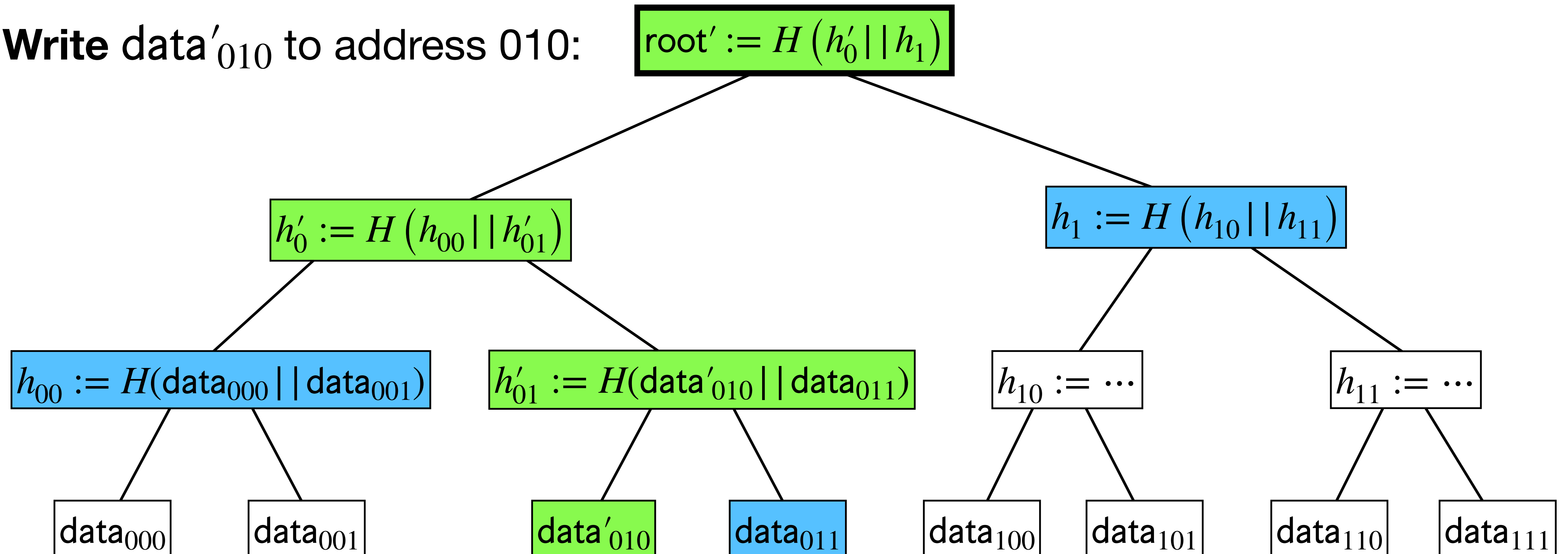
Merkle Trees

Write data'_{010} to address 010:



Merkle Trees

Write data'_{010} to address 010:



Merkle Trees

- **Efficiency analysis:**

Merkle Trees

- **Efficiency analysis:**
 - Query all **nodes** on path from leaf to root: $\approx \log N$.

Merkle Trees

- **Efficiency analysis:**
 - Query all **nodes** on path from leaf to root: $\approx \log N$.
 - Query all **neighbors** along the path: $\approx \log N$.

Merkle Trees

- **Efficiency analysis:**
 - Query all **nodes** on path from leaf to root: $\approx \log N$.
 - Query all **neighbors** along the path: $\approx \log N$.
 - **Total Overhead:** $\approx 2 \log N$.

Merkle Trees

- **Efficiency analysis:**
 - Query all **nodes** on path from leaf to root: $\approx \log N$.
 - Query all **neighbors** along the path: $\approx \log N$.
 - **Total Overhead:** $\approx 2 \log N$.
 - **Local Space:** Hash root and key (can both be made small).

Merkle Trees

- **Security:**

Merkle Trees

- **Security:**
 - Suppose adversary cheats (undetectably forces wrong output on some read).

Merkle Trees

- **Security:**
 - Suppose adversary cheats (undetectably forces wrong output on some read).
 - Consider first, top-most entry that adversary gives **wrong** hash value.

Merkle Trees

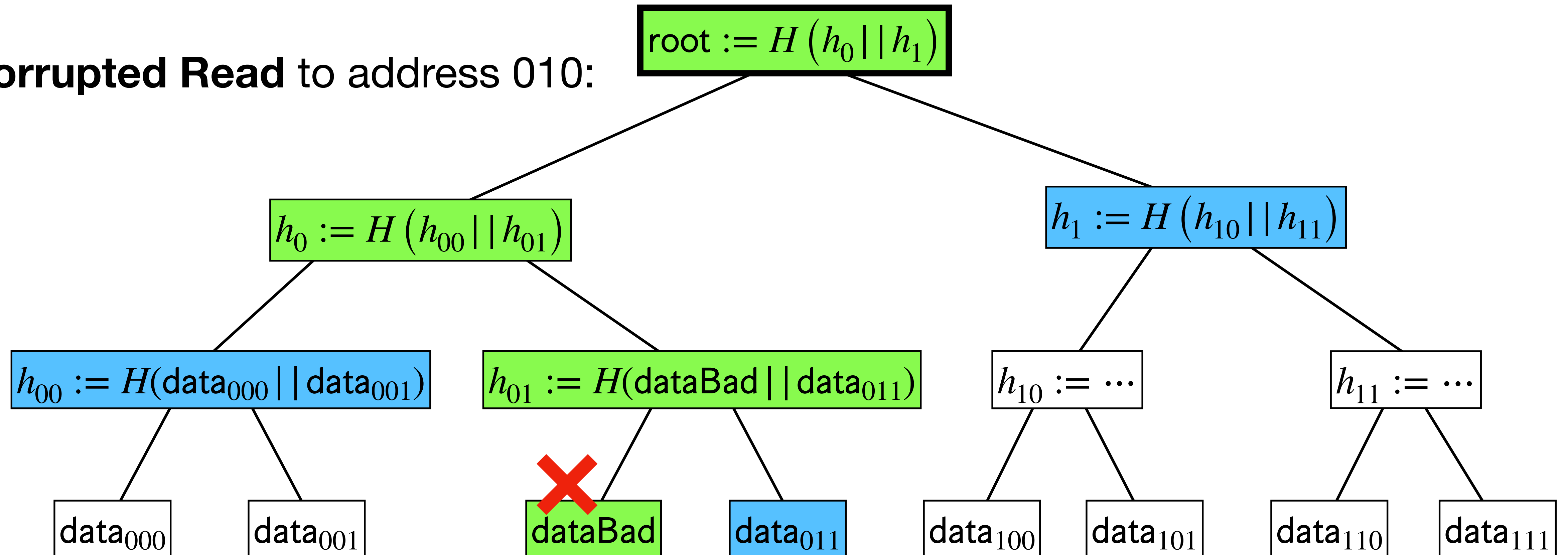
- **Security:**
 - Suppose adversary cheats (undetectably forces wrong output on some read).
 - Consider first, top-most entry that adversary gives **wrong** hash value.
 - Can't be the root, because we store the root locally.

Merkle Trees

- **Security:**
 - Suppose adversary cheats (undetectably forces wrong output on some read).
 - Consider first, top-most entry that adversary gives **wrong** hash value.
 - Can't be the root, because we store the root locally.
 - This will be a hash collision!

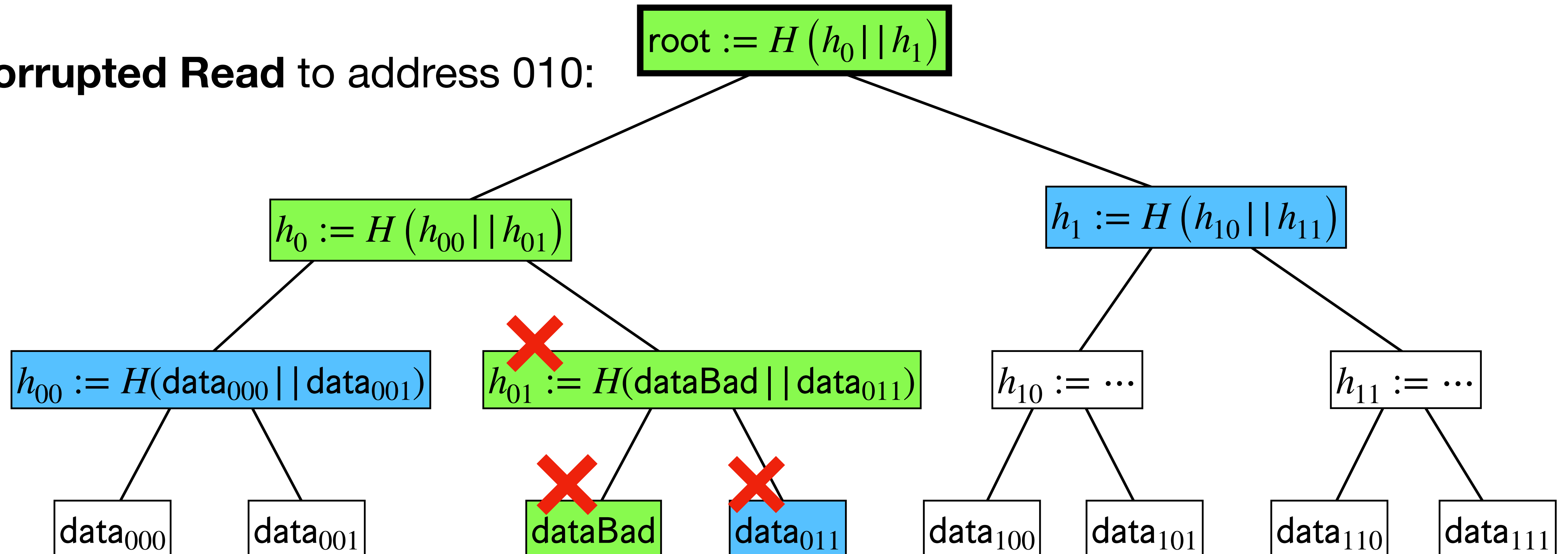
Merkle Trees

Corrupted Read to address 010:



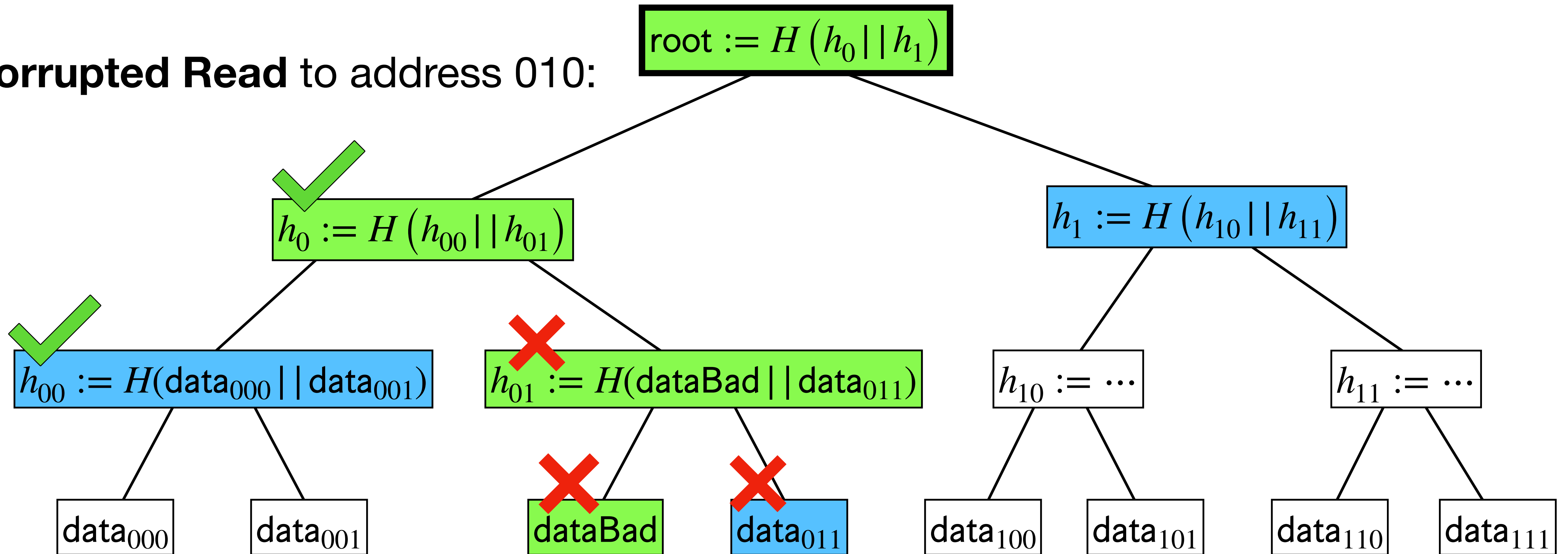
Merkle Trees

Corrupted Read to address 010:



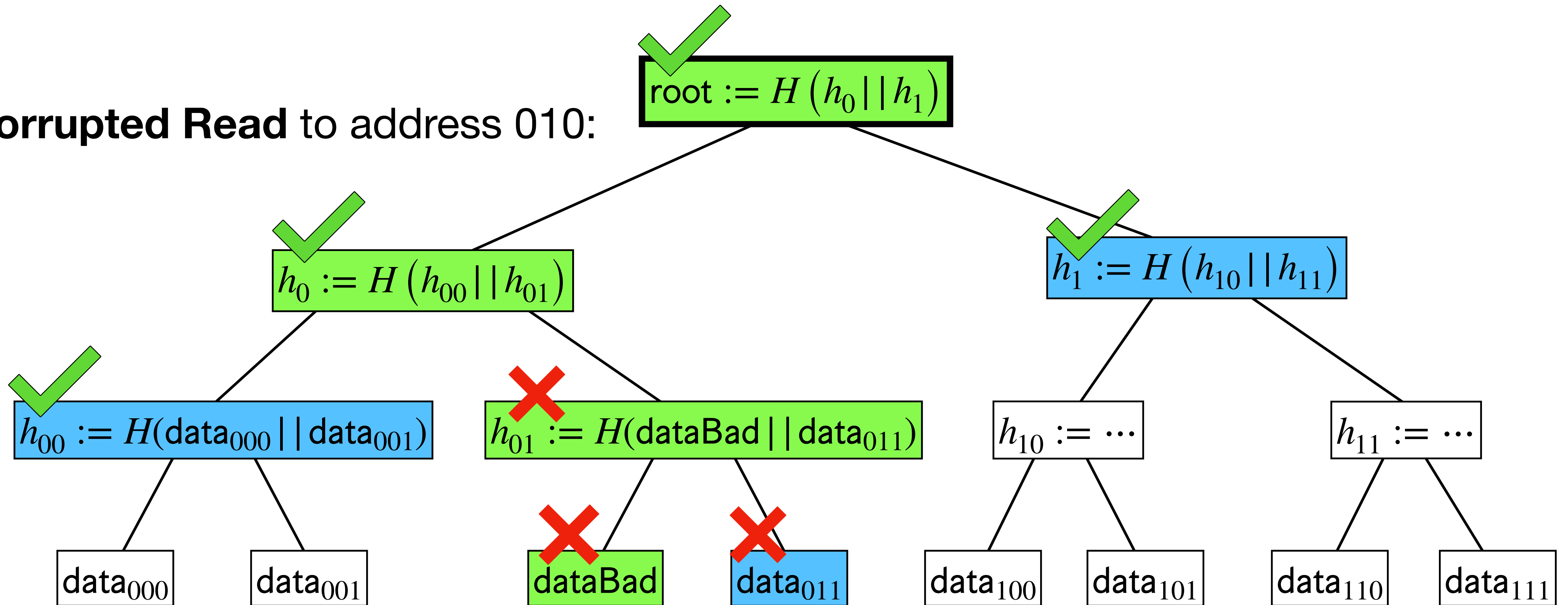
Merkle Trees

Corrupted Read to address 010:



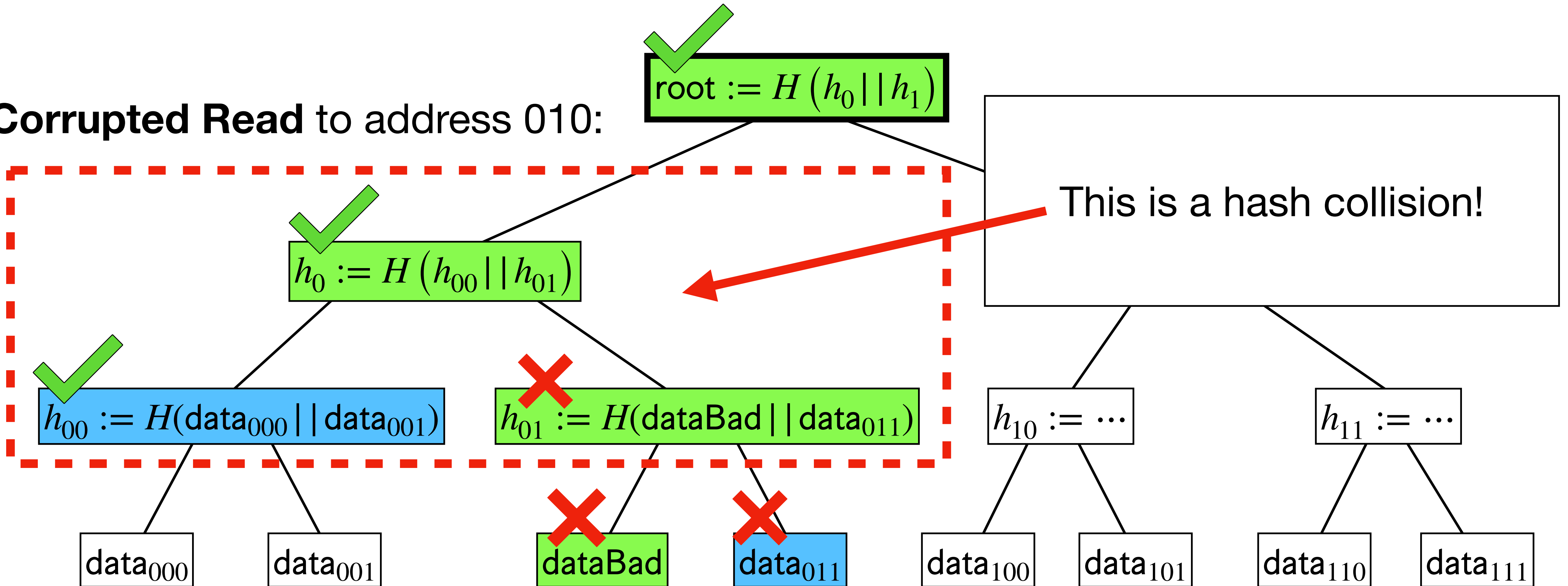
Merkle Trees

Corrupted Read to address 010:



Merkle Trees

Corrupted Read to address 010:



Merkle Trees

Merkle Trees

This is just one use case of Merkle trees. There's more!

Merkle Trees

This is just one use case of Merkle trees. There's more!

1. Succinct Argument System for NP (Merkle trees + PCP theorem).

Merkle Trees

This is just one use case of Merkle trees. There's more!

1. Succinct Argument System for NP (Merkle trees + PCP theorem).
2. Trusted Hardware (e.g., Apple's Secure Enclave).

Merkle Trees

This is just one use case of Merkle trees. There's more!

1. Succinct Argument System for NP (Merkle trees + PCP theorem).
2. Trusted Hardware (e.g., Apple's Secure Enclave).
3. Blockchains (e.g., bitcoin)!



Merkle Trees in Bitcoin

Merkle Trees in Bitcoin

Merkle trees also allow for **inclusion proofs**.

Merkle Trees in Bitcoin

Merkle trees also allow for **inclusion proofs**.

- Relative to a fixed Merkle root, allows prover to prove existence of a leaf.

Merkle Trees in Bitcoin

Merkle trees also allow for **inclusion proofs**.

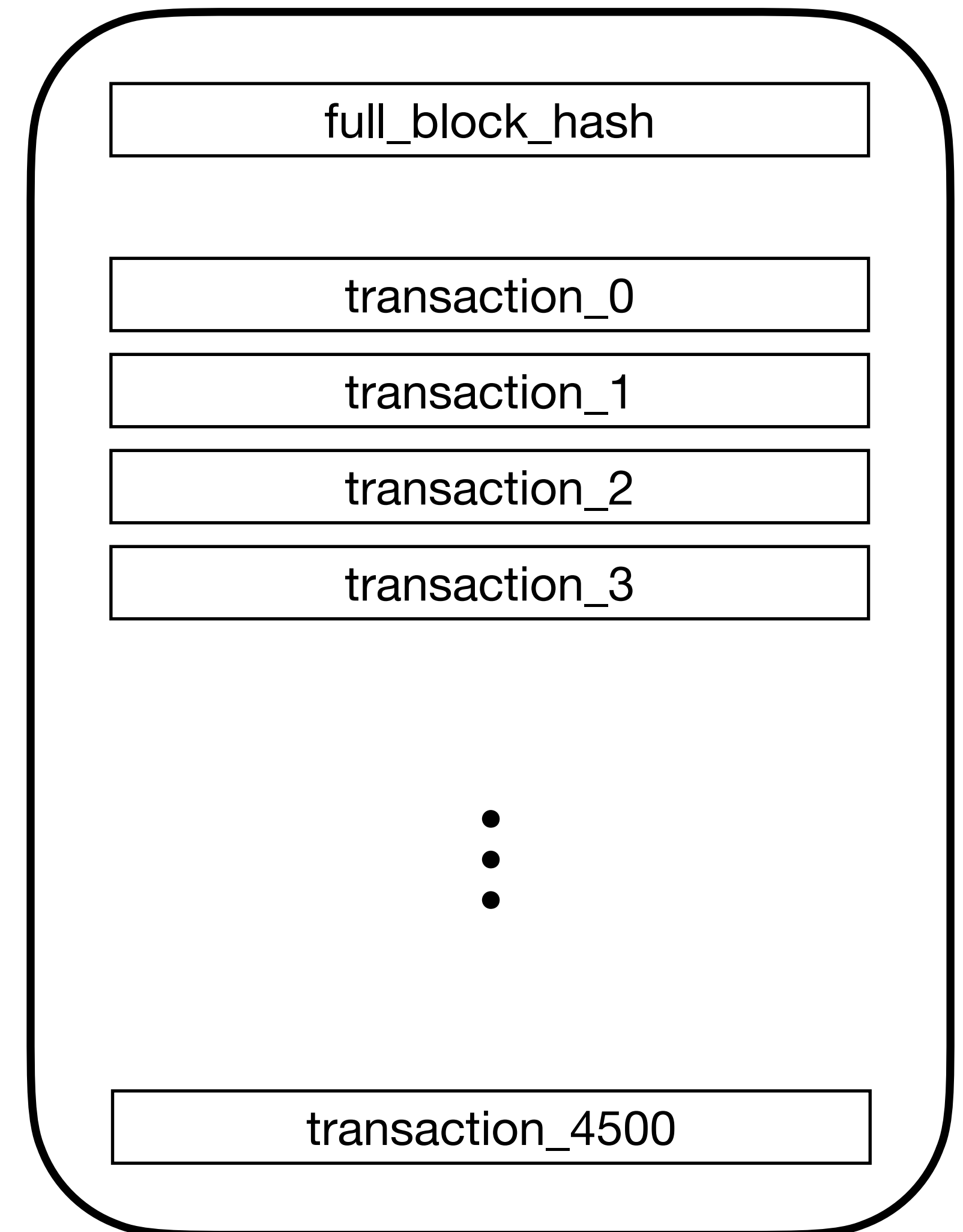
- Relative to a fixed Merkle root, allows prover to prove existence of a leaf.

In Bitcoin:

...



Transaction Block



Merkle Trees in Bitcoin

Merkle trees also allow for **inclusion proofs**.

- Relative to a fixed Merkle root, allows prover to prove existence of a leaf.

In Bitcoin:

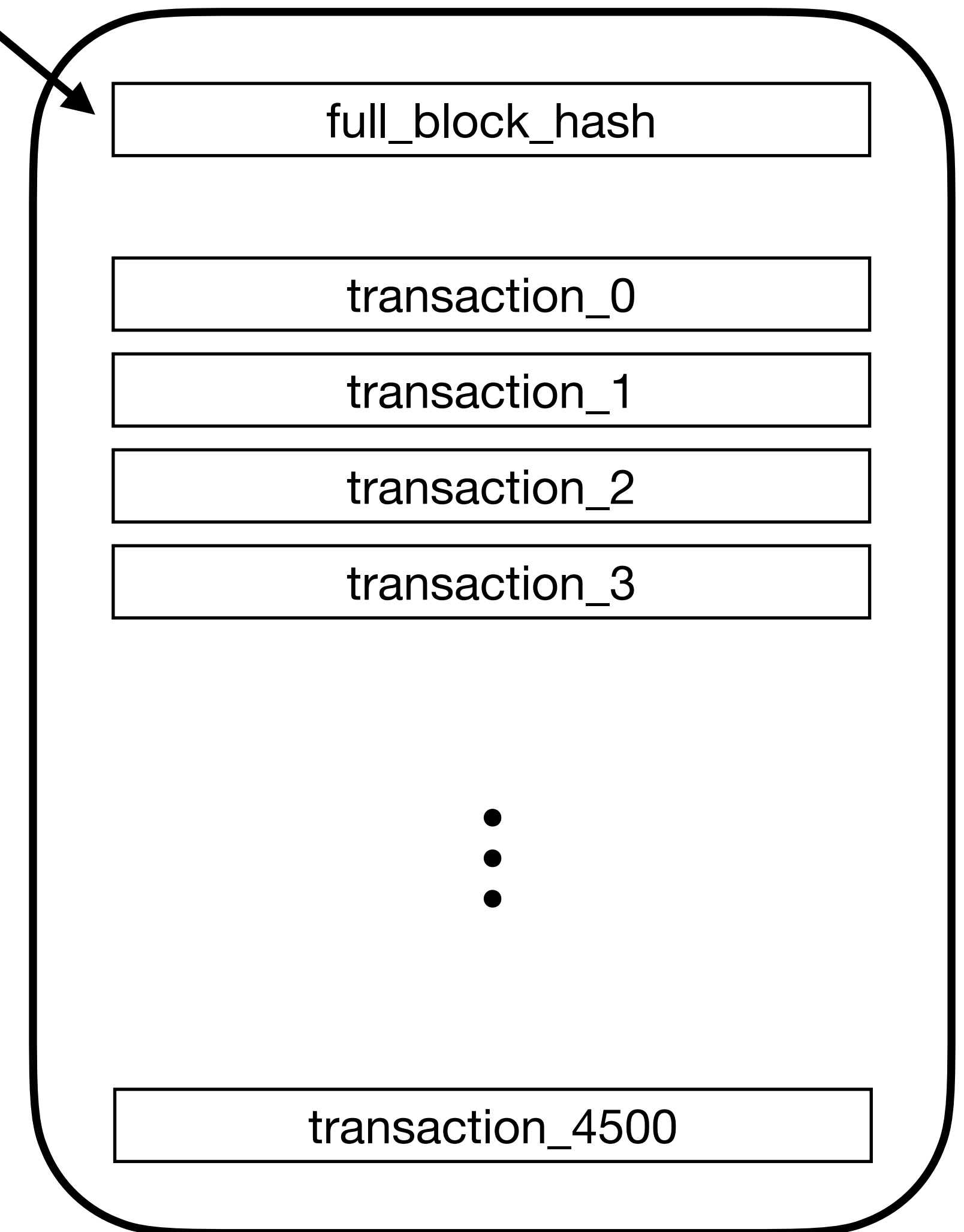
- Merkle root hash authenticity guaranteed by the blockchain.

...



Merkle root

Transaction Block



Merkle Trees in Bitcoin

Merkle trees also allow for **inclusion proofs**.

- Relative to a fixed Merkle root, allows prover to prove existence of a leaf.

In Bitcoin:

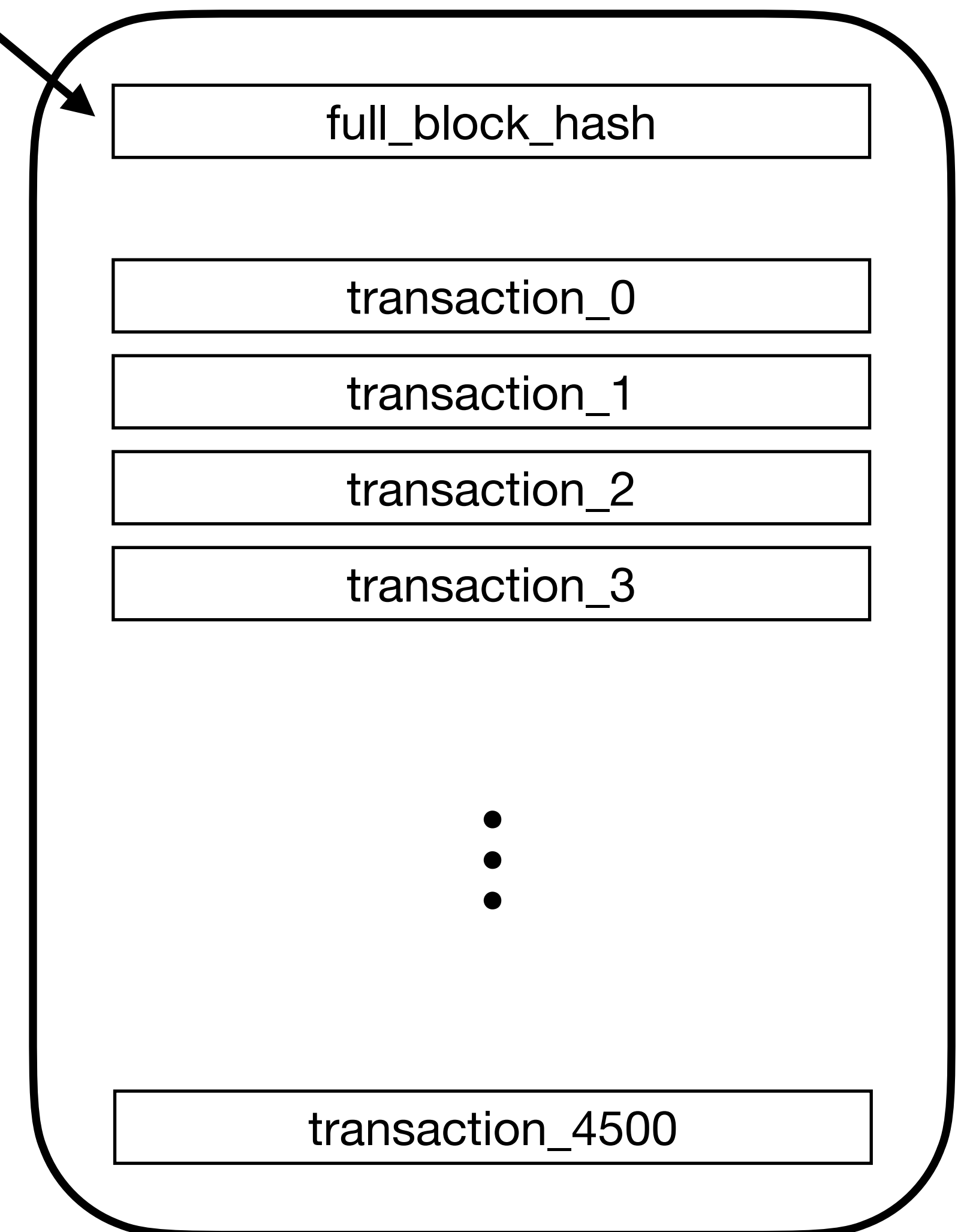
- Merkle root hash authenticity guaranteed by the blockchain.
- Leaves are transactions.

...



Merkle root

Transaction Block



Merkle Trees in Bitcoin

Merkle trees also allow for **inclusion proofs**.

- Relative to a fixed Merkle root, allows prover to prove existence of a leaf.

In Bitcoin:

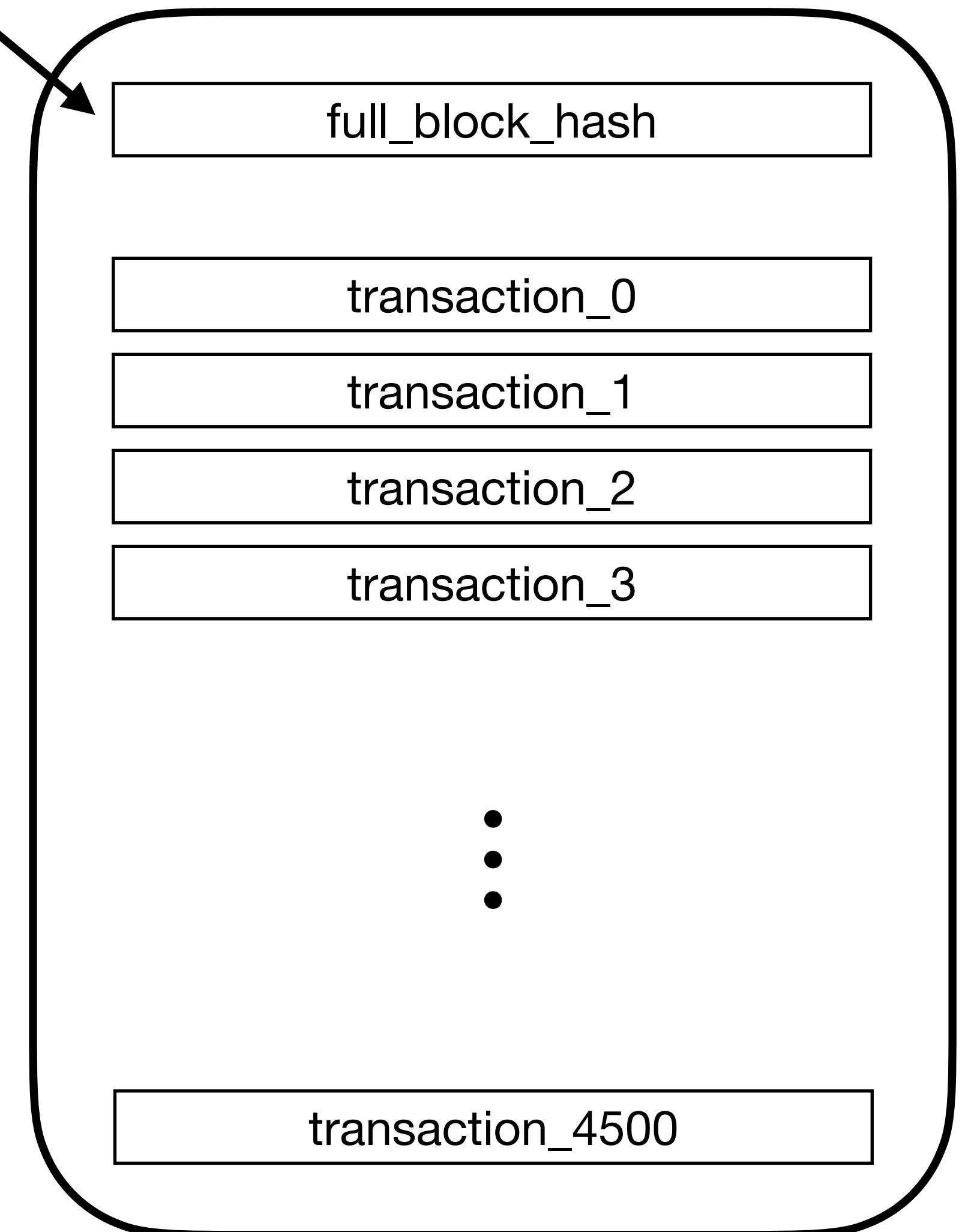
- Merkle root hash authenticity guaranteed by the blockchain.
- Leaves are transactions.
- Proof contains hashes of path and siblings.

...



Merkle root

Transaction Block



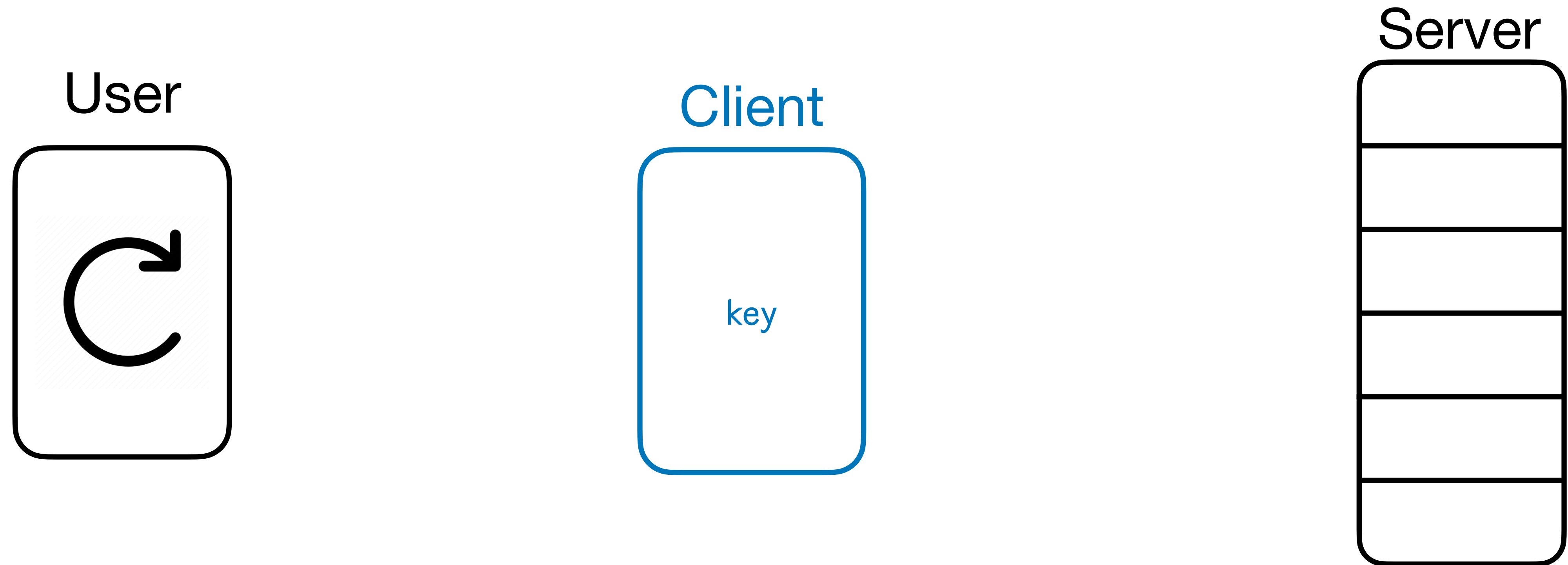
Solving Privacy: Oblivious RAM

Oblivious RAM (Solving Privacy Issue)

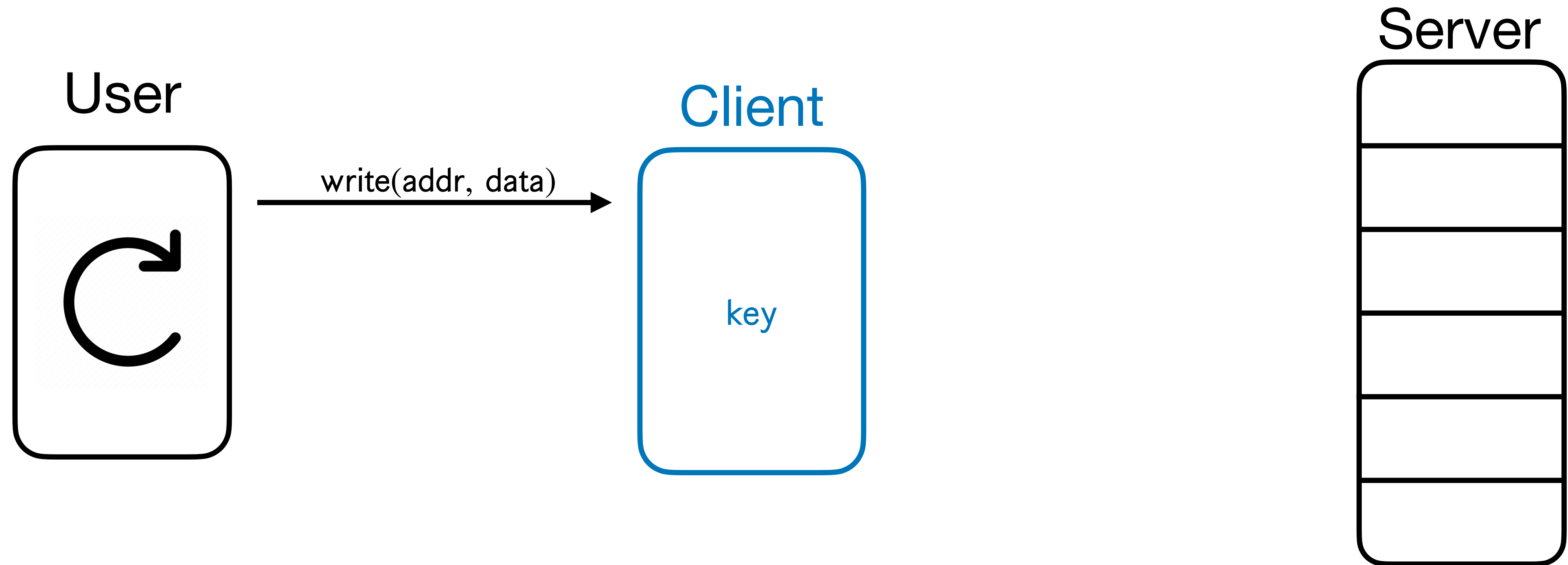
Oblivious RAM (Solving Privacy Issue)

Wait, does encryption solve the privacy issue?

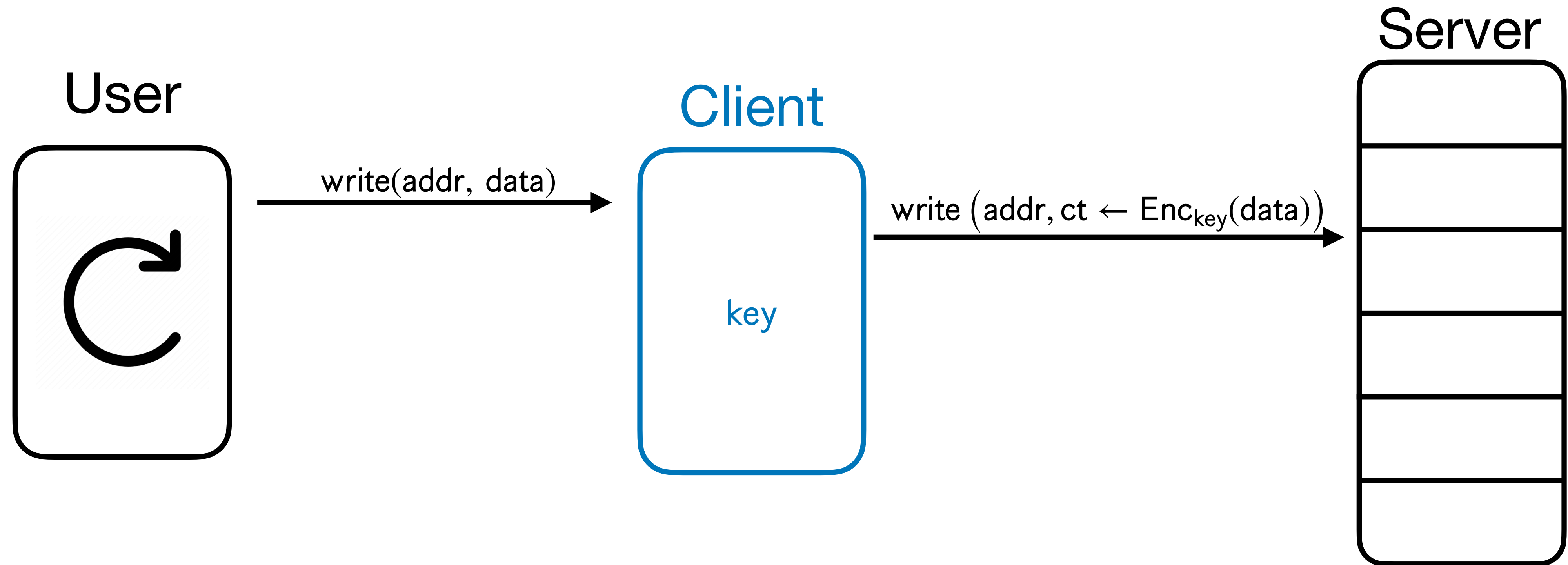
Encryption as ORAM?



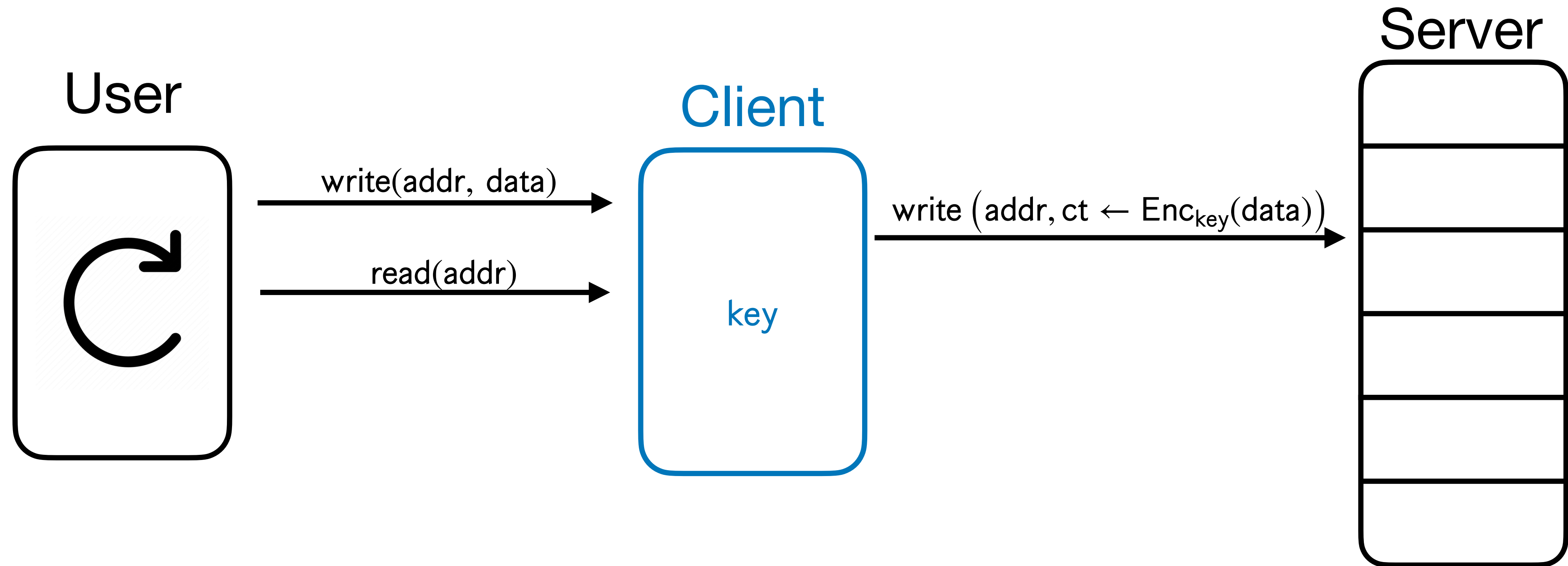
Encryption as ORAM?



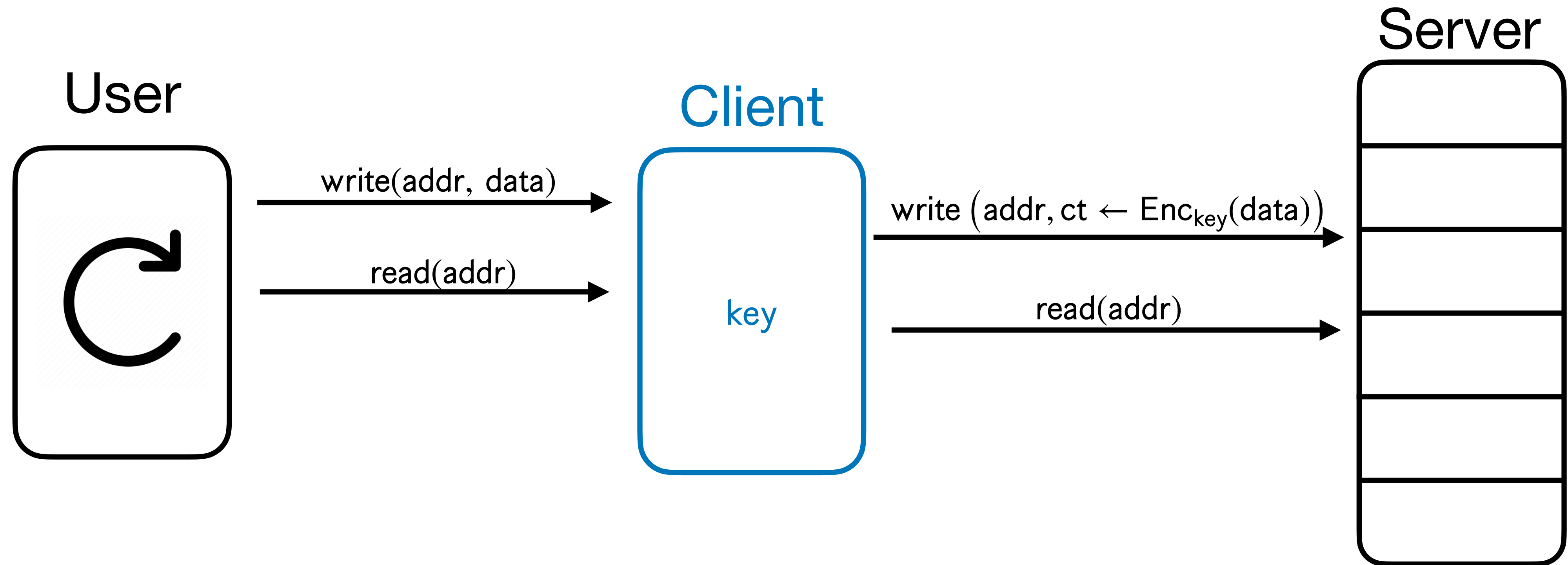
Encryption as ORAM?



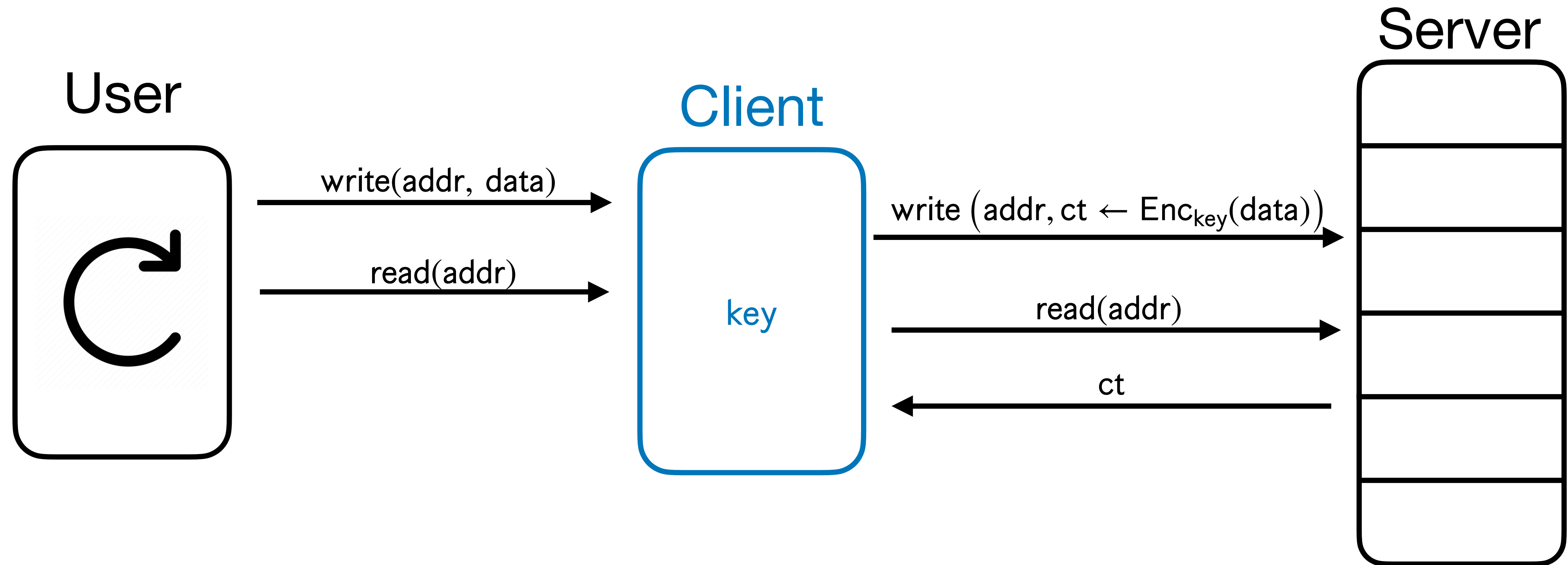
Encryption as ORAM?



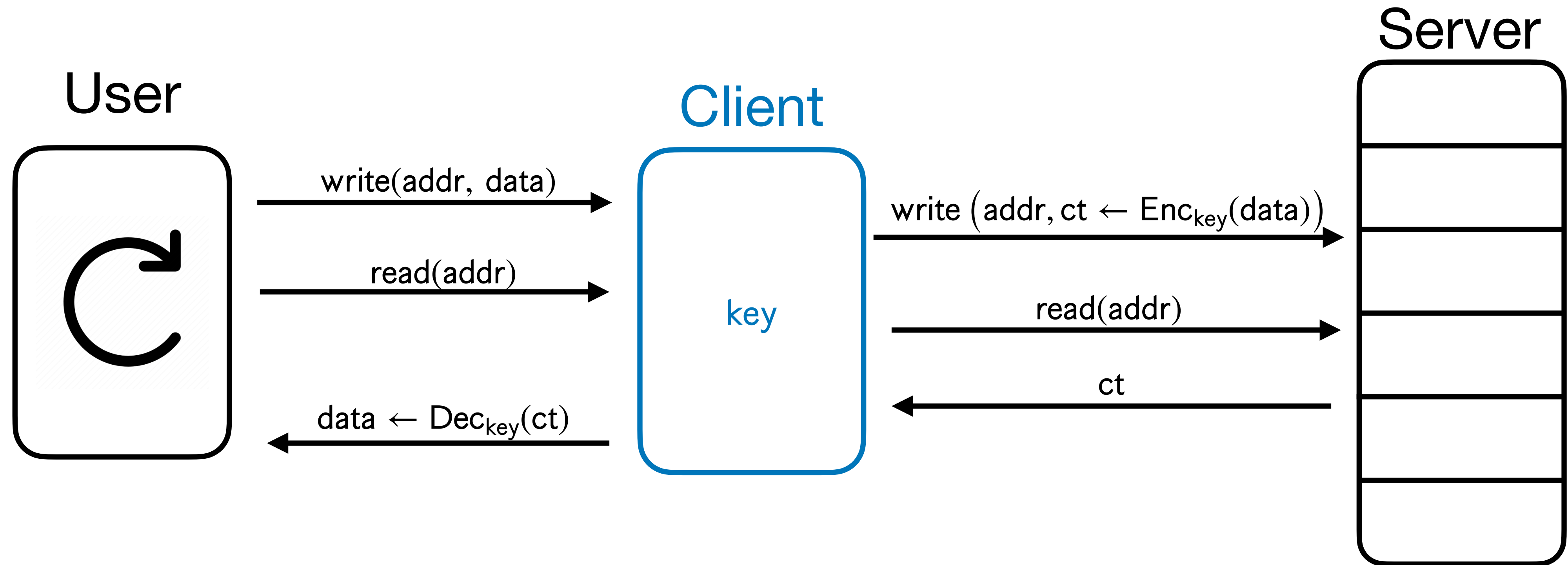
Encryption as ORAM?



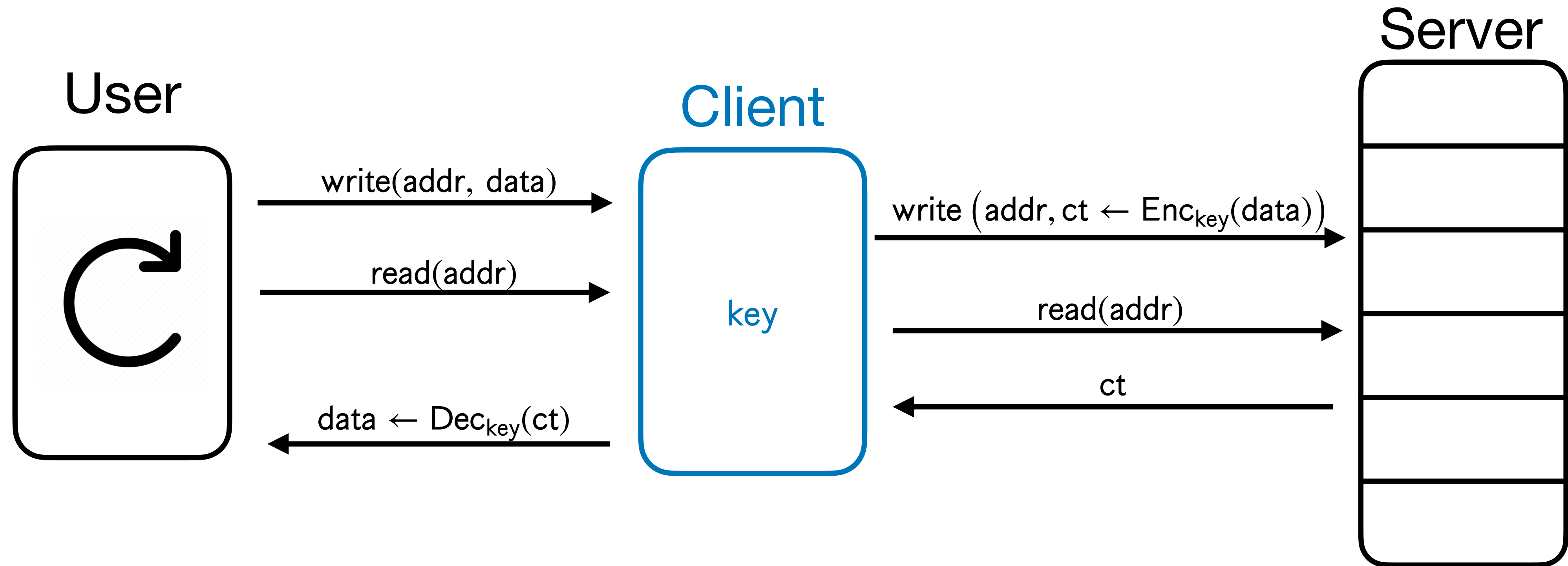
Encryption as ORAM?



Encryption as ORAM?



Encryption as ORAM?



Does this work?

No!

No!

- This solution still reveals the **access pattern** of the user.

No!

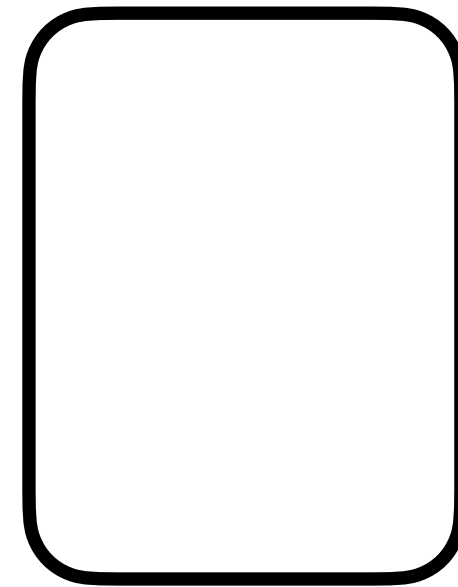
- This solution still reveals the **access pattern** of the user.
- Server knows **where** the user is querying.

No!

- This solution still reveals the **access pattern** of the user.
 - Server knows **where** the user is querying.
- This matters!

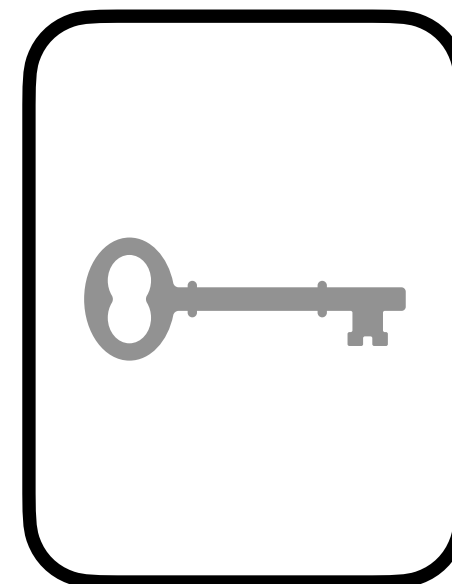
No!

- This solution still reveals the **access pattern** of the user. Server
- Server knows **where** the user is querying.
- This matters!

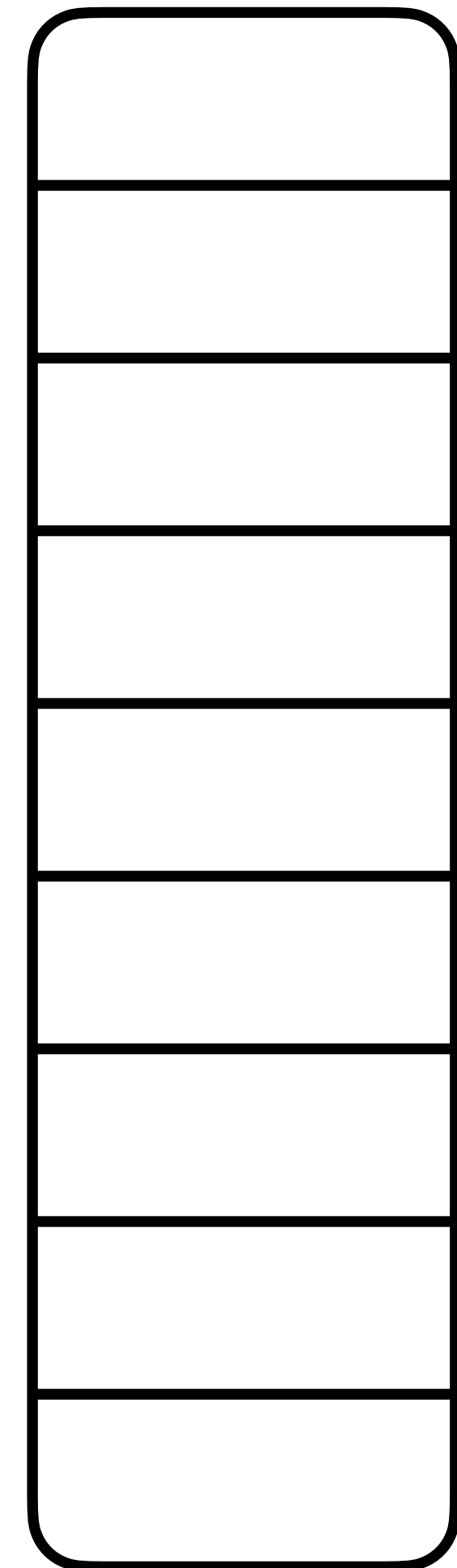


No!

- This solution still reveals the **access pattern** of the user.
- Server knows **where** the user is querying.
- This matters!



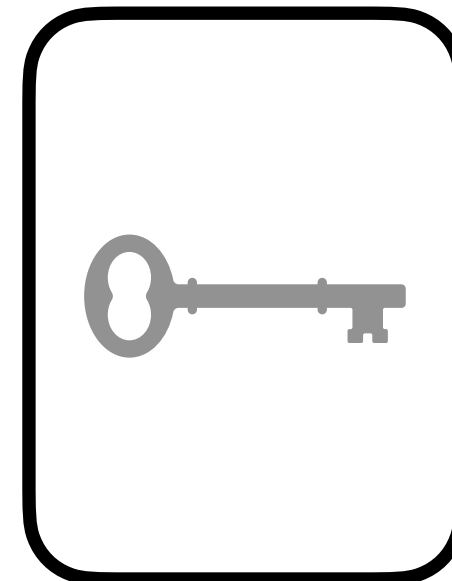
Server



No!

- This solution still reveals the **access pattern** of the user.
- Server knows **where** the user is querying.
- This matters!

Scientist



Server



Brain
Data

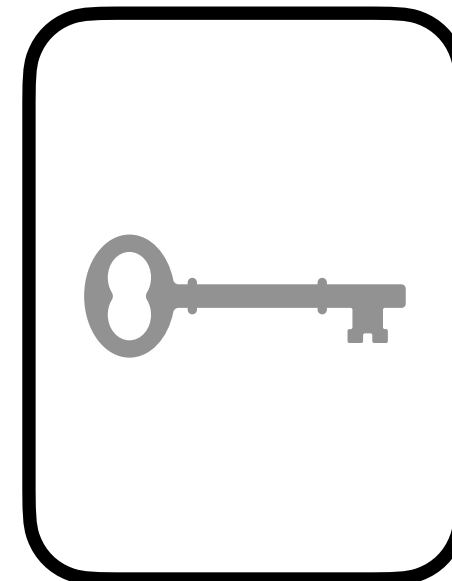
Kidney
Data

Heart
Data

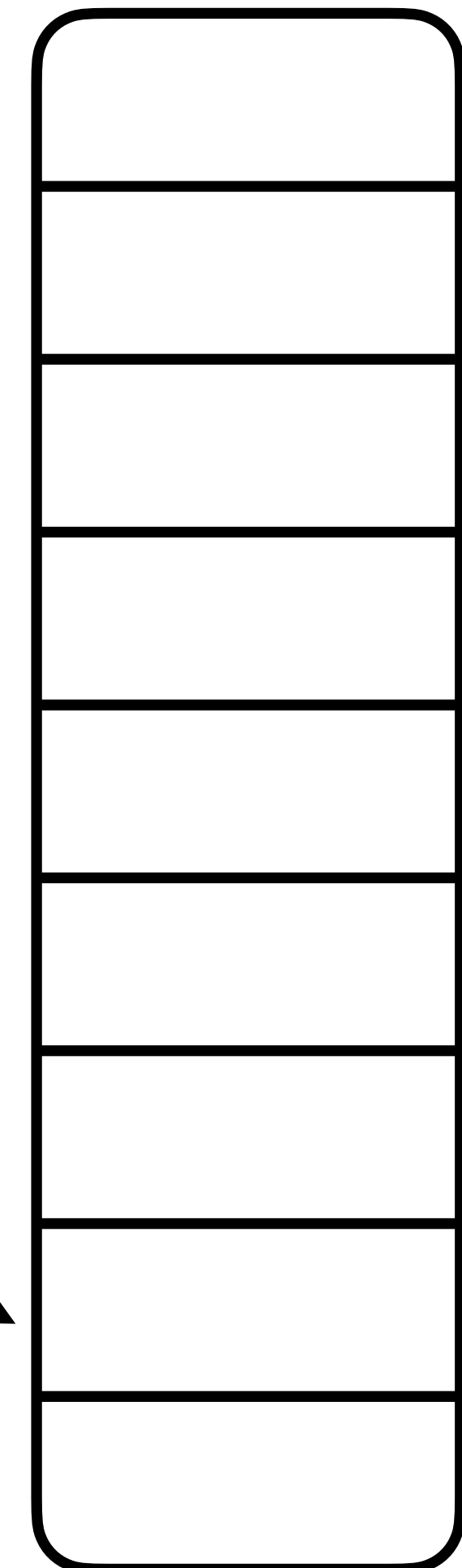
No!

- This solution still reveals the **access pattern** of the user.
- Server knows **where** the user is querying.
- This matters!

Scientist



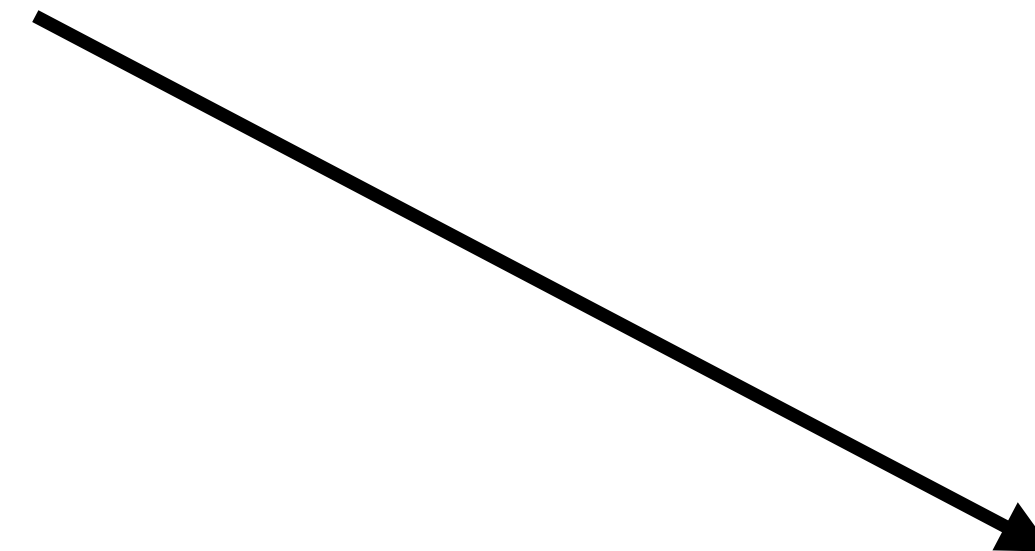
Server



Brain
Data

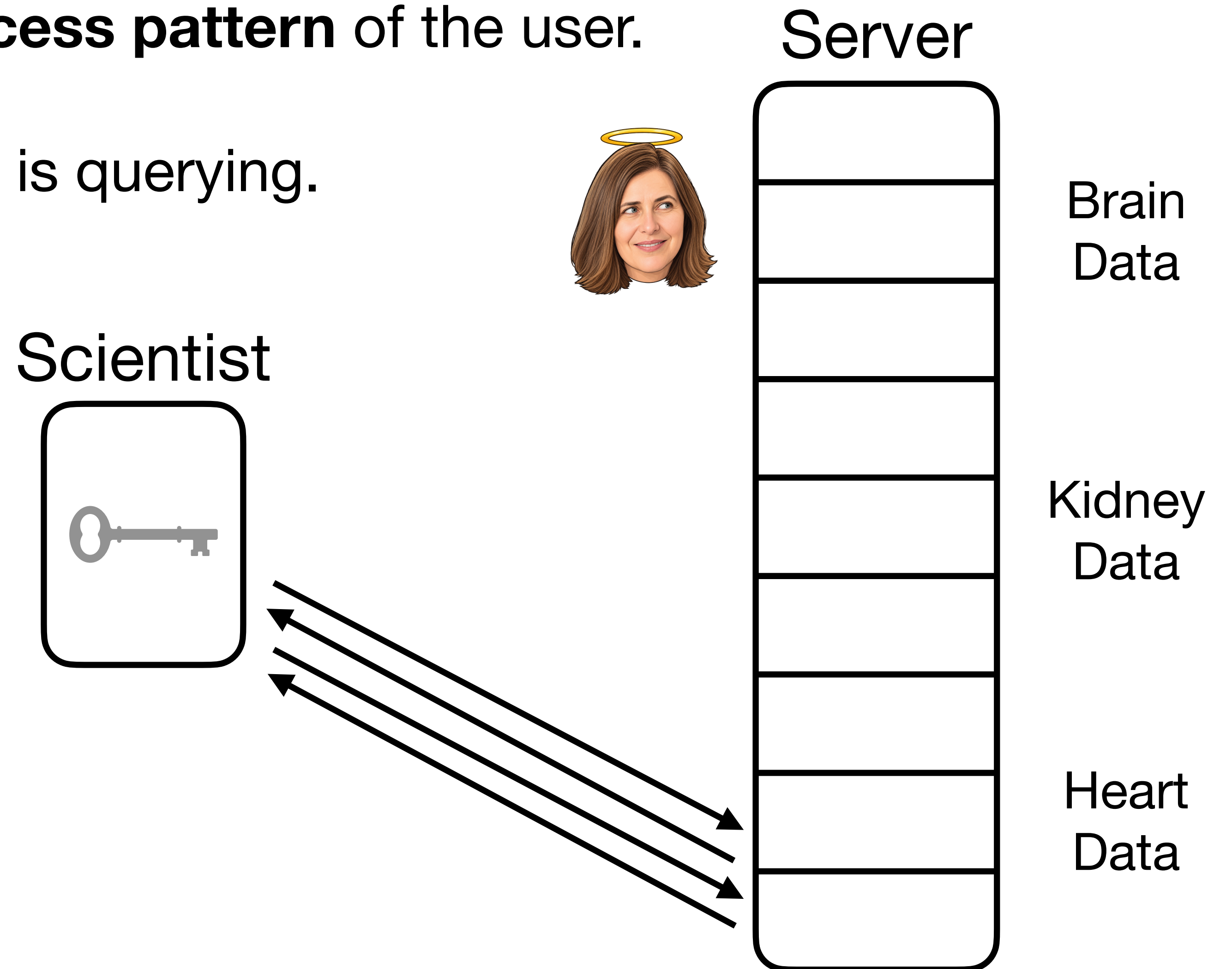
Kidney
Data

Heart
Data



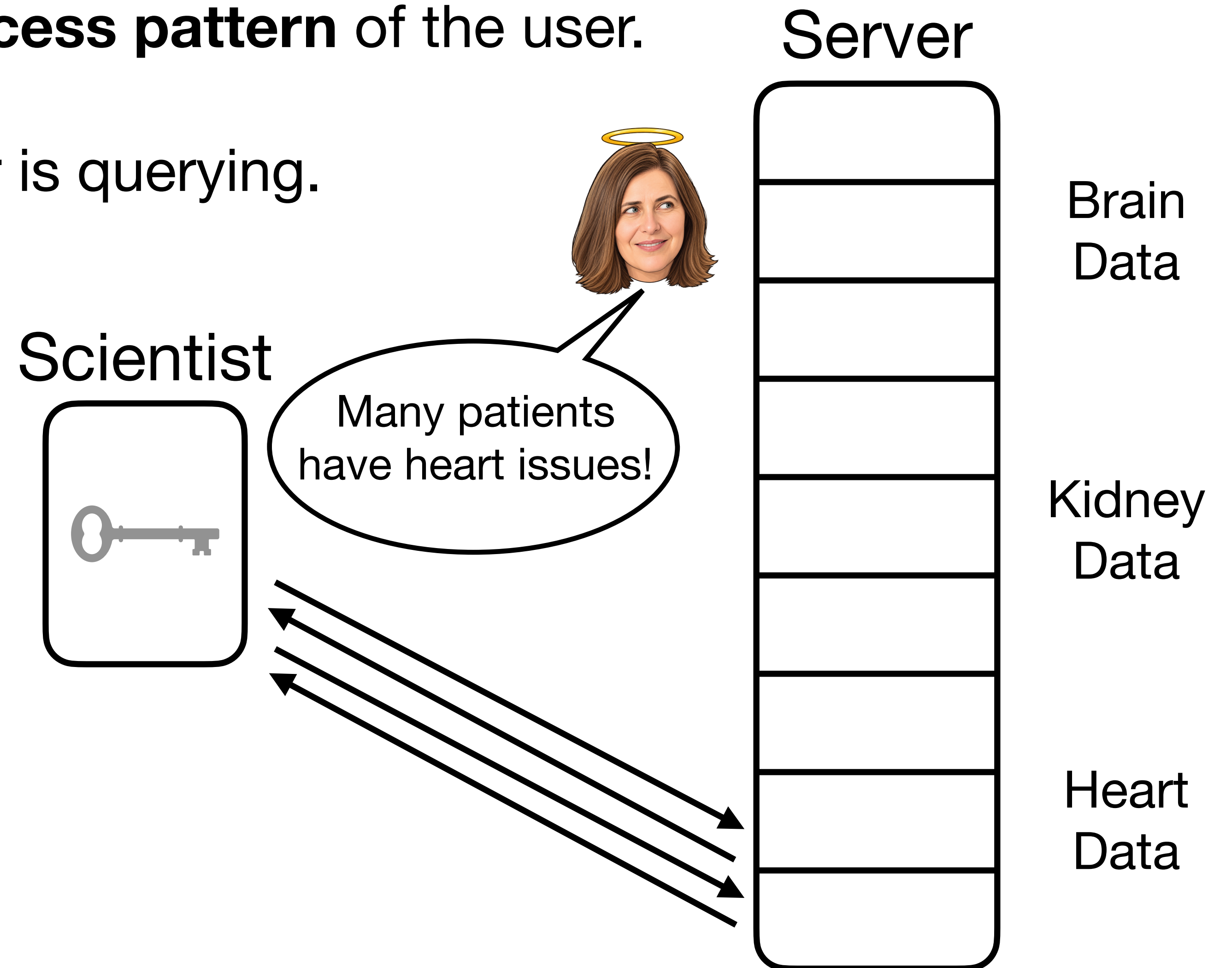
No!

- This solution still reveals the **access pattern** of the user.
- Server knows **where** the user is querying.
- This matters!



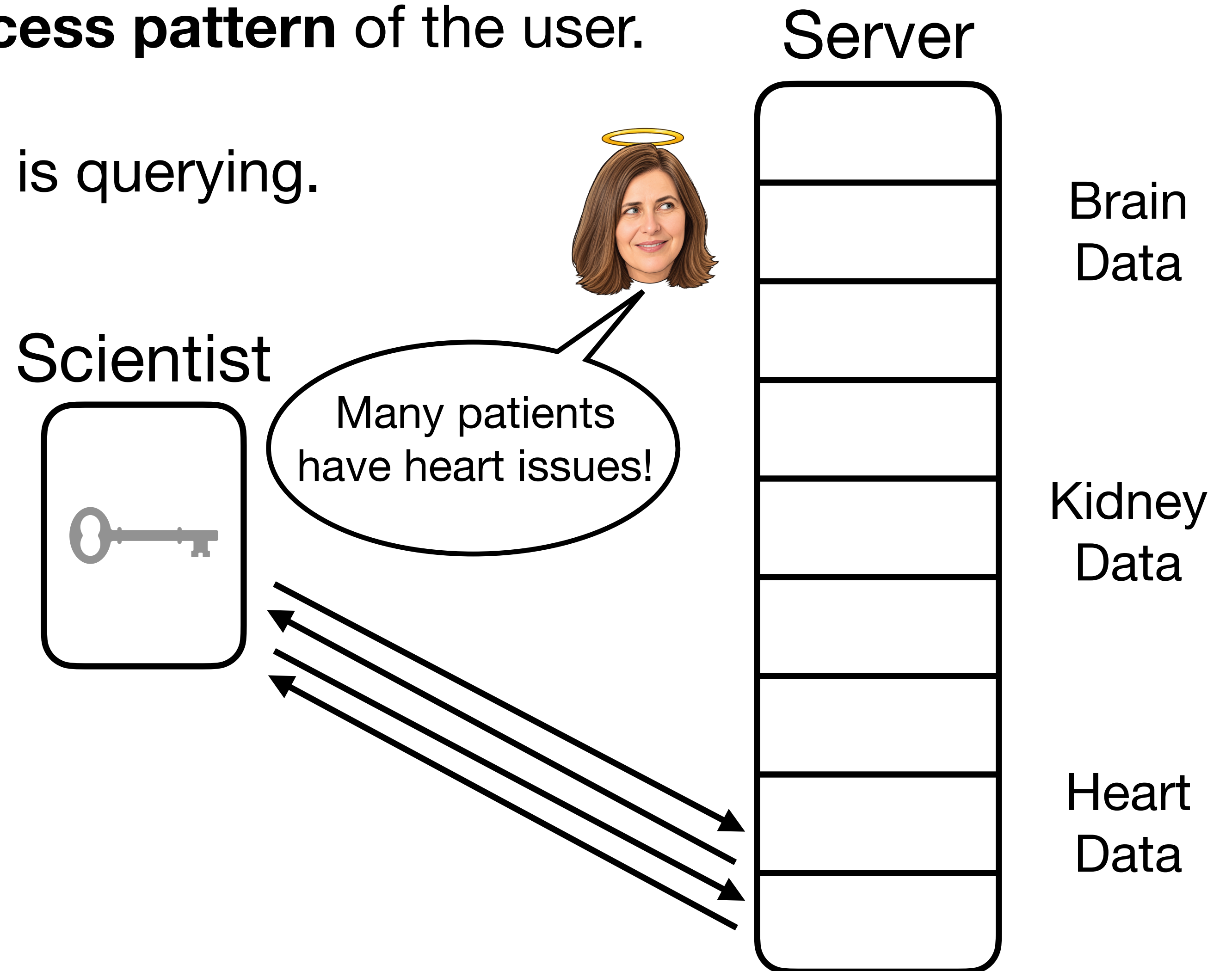
No!

- This solution still reveals the **access pattern** of the user.
- Server knows **where** the user is querying.
- This matters!



No!

- This solution still reveals the **access pattern** of the user.
- Server knows **where** the user is querying.
- This matters!
- **Real world example:**
Adversary looking at accesses to encrypted email repository can recover as much as 80% of search queries [IKK '12].



What about permuting addresses?

What about permuting addresses?

- Fine, but why not randomly shuffle the address space?

What about permuting addresses?

- Fine, but why not randomly shuffle the address space?
- Specifically, apply a (pseudorandom) permutation to address space and encrypt?

What about permuting addresses?

- Fine, but why not randomly shuffle the address space?
- Specifically, apply a (pseudorandom) permutation to address space and encrypt?
- What goes wrong?

What about permuting addresses?

- Fine, but why not randomly shuffle the address space?
- Specifically, apply a (pseudorandom) permutation to address space and encrypt?
- What goes wrong?
- **Reveals repeated queries!**

What about permuting addresses?

- Fine, but why not randomly shuffle the address space?
- Specifically, apply a (pseudorandom) permutation to address space and encrypt?
- What goes wrong?
- **Reveals repeated queries!**
- Idea: “freshly” randomize address space each time.

Path ORAM

Path ORAM

- Once again, we'll use a binary tree.

Path ORAM

- Once again, we'll use a binary tree.
- (Throughout, we'll encrypt everything using secret-key encryption.)

Path ORAM

- Once again, we'll use a binary tree.
- (Throughout, we'll encrypt everything using secret-key encryption.)
- Each vertex of a binary tree will store a bucket of $O(1)$ data “blocks”.

Path ORAM

- Once again, we'll use a binary tree.
- (Throughout, we'll encrypt everything using secret-key encryption.)
- Each vertex of a binary tree will store a bucket of $O(1)$ data “blocks”.
- Let $\text{pos}[\text{addr}]$ be a locally stored array containing addr 's “assigned” leaf.

Path ORAM

- Once again, we'll use a binary tree.
- (Throughout, we'll encrypt everything using secret-key encryption.)
- Each vertex of a binary tree will store a bucket of $O(1)$ data “blocks”.
- Let $\text{pos}[\text{addr}]$ be a locally stored array containing addr 's “assigned” leaf.
 - This is $\Omega(N)$ local storage! Let's not worry about it for now.

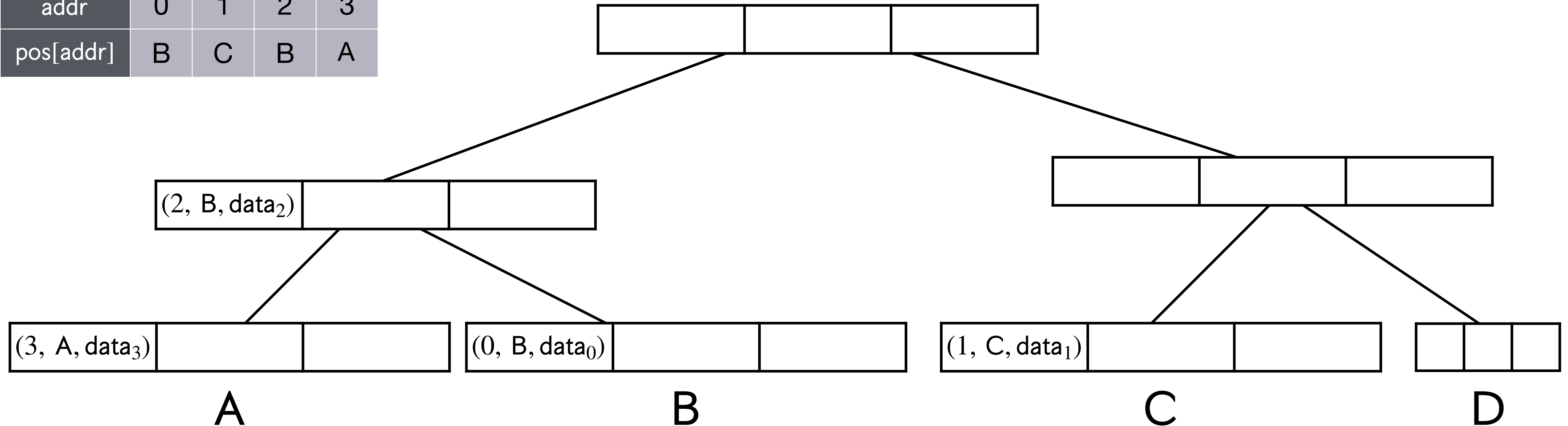
Path ORAM

- Once again, we'll use a binary tree.
- (Throughout, we'll encrypt everything using secret-key encryption.)
- Each vertex of a binary tree will store a bucket of $O(1)$ data “blocks”.
- Let $\text{pos}[\text{addr}]$ be a locally stored array containing addr 's “assigned” leaf.
 - This is $\Omega(N)$ local storage! Let's not worry about it for now.
- Each data block consists of $(\text{addr}, \text{pos}[\text{addr}], \text{data})$.

Path ORAM

(Here, $N = 4$.)

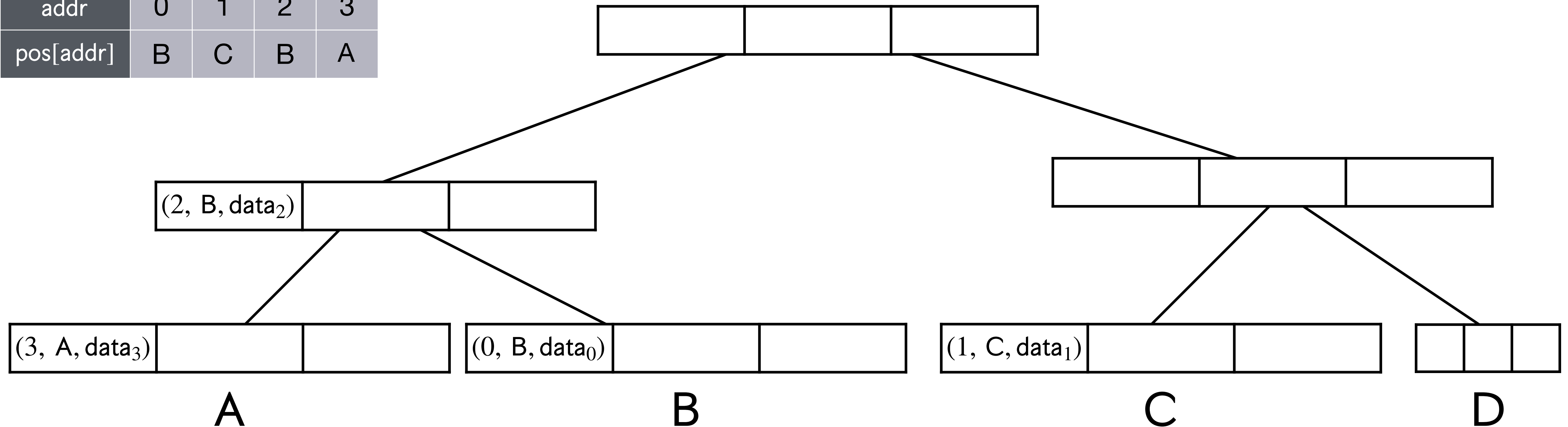
addr	0	1	2	3
pos[addr]	B	C	B	A



Path ORAM

(Here, $N = 4$.)

addr	0	1	2	3
pos[addr]	B	C	B	A

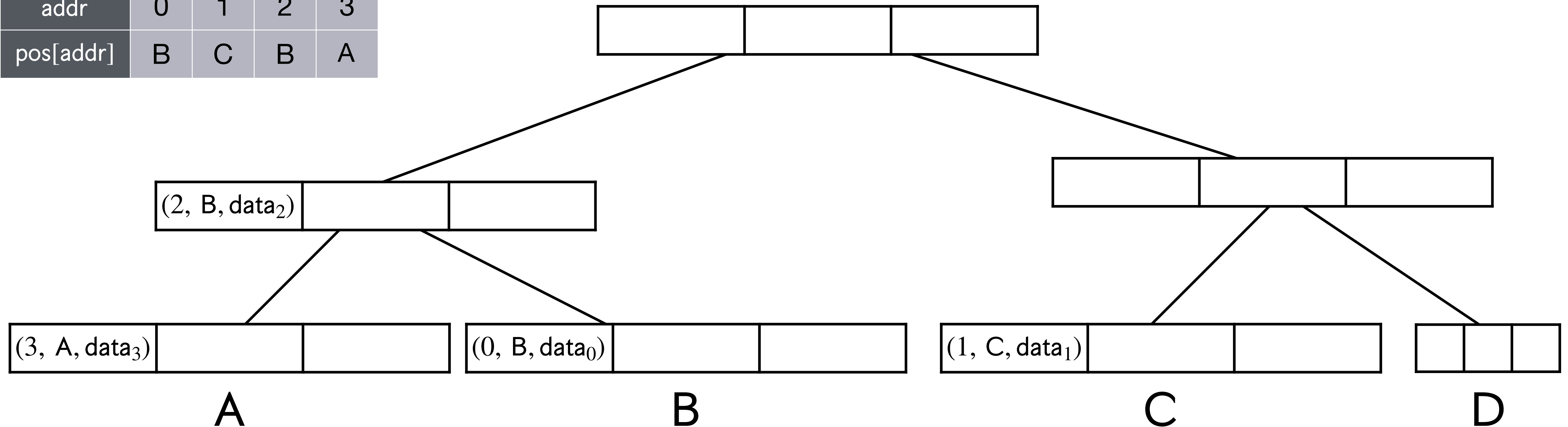


On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

Path ORAM

(Here, $N = 4$.)

addr	0	1	2	3
pos[addr]	B	C	B	A



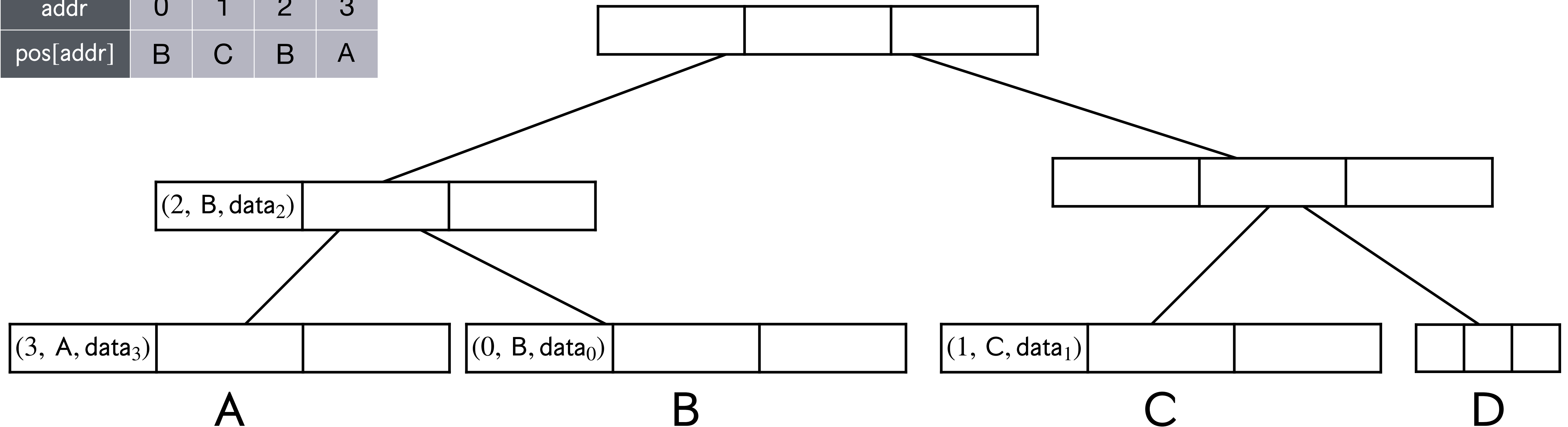
On each query to $addr \in \{0, 1, 2, 3\}$:

1. **Look up** $pos[addr] \in \{A, B, C, D\}$ locally.

Path ORAM

(Here, $N = 4$.)

addr	0	1	2	3
pos[addr]	B	C	B	A



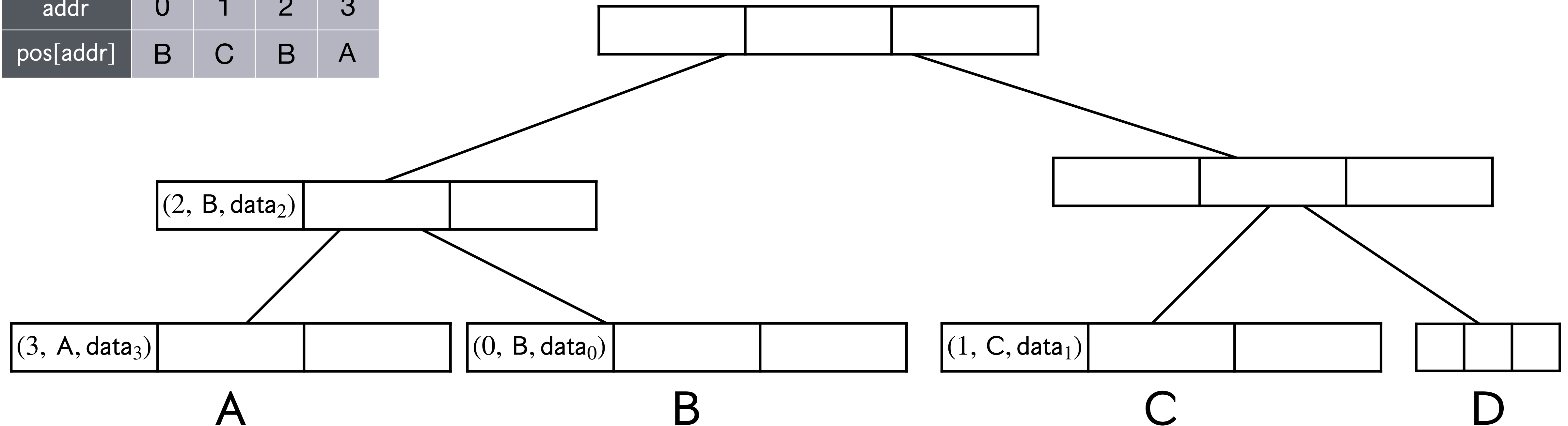
On each query to $addr \in \{0, 1, 2, 3\}$:

1. **Look up** $pos[addr] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $pos[addr]$.

Path ORAM

(Here, $N = 4$.)

addr	0	1	2	3
pos[addr]	B	C	B	A



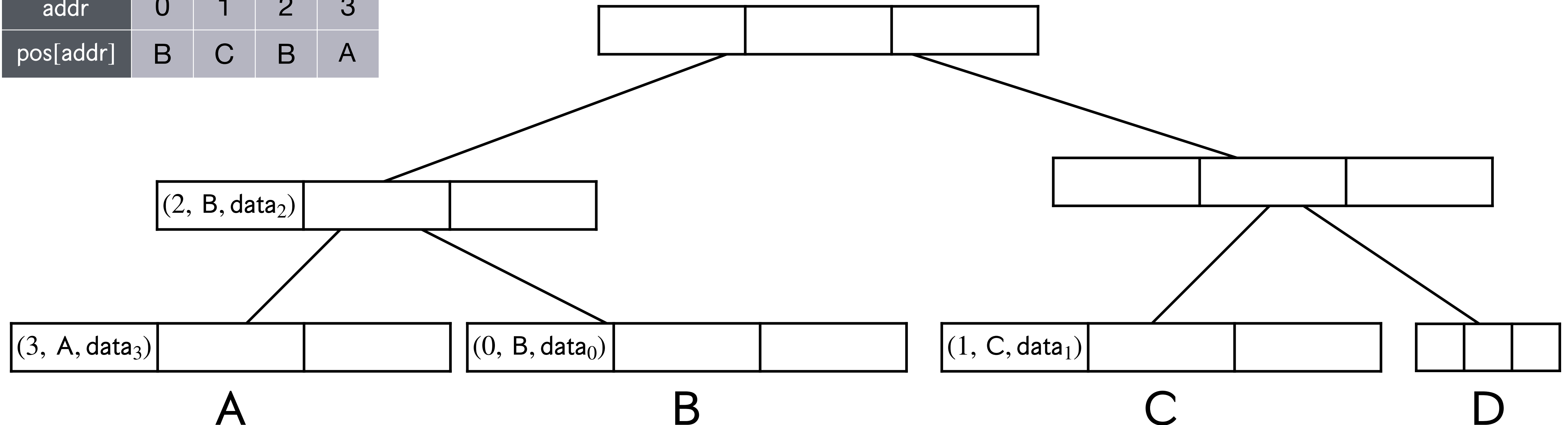
On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.

Path ORAM

(Here, $N = 4$.)

addr	0	1	2	3
pos[addr]	B	C	B	A



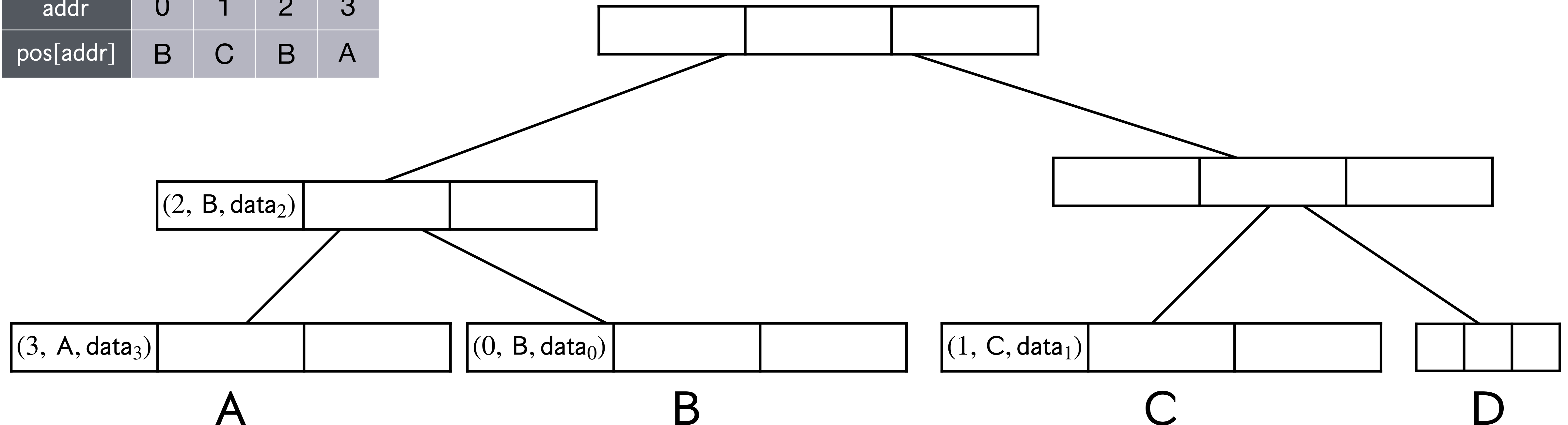
On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Path ORAM

(Here, $N = 4$.)

addr	0	1	2	3
pos[addr]	B	C	B	A



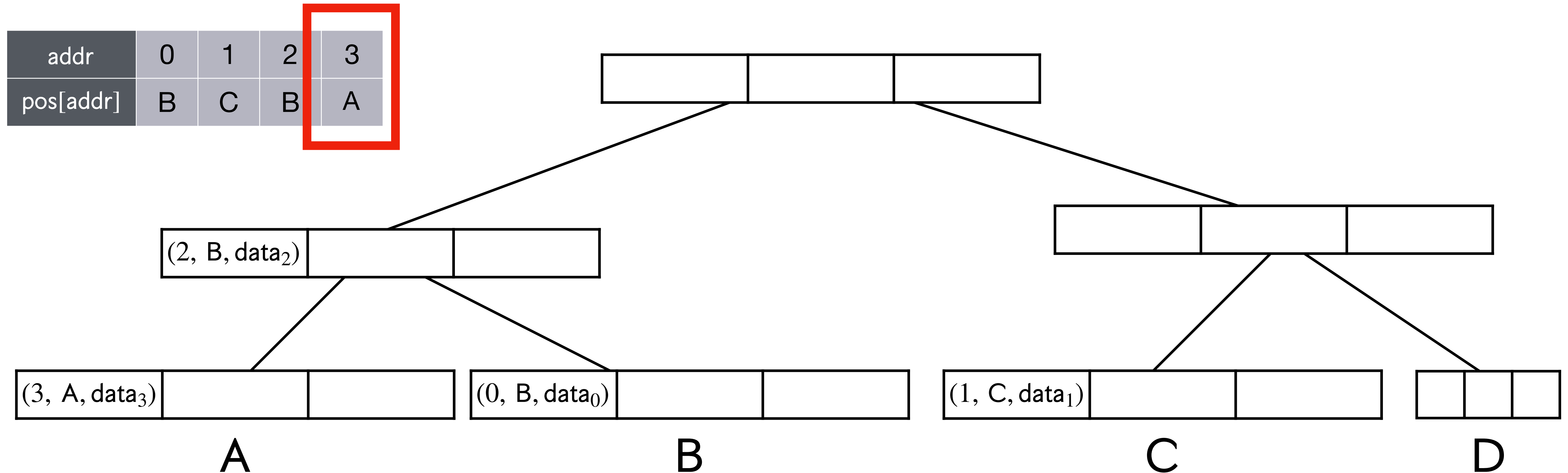
On each query to $addr \in \{0, 1, 2, 3\}$:

1. **Look up** $pos[addr] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $pos[addr]$.
3. Randomly **sample** new value $pos[addr] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Example: Read $addr = 3$

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

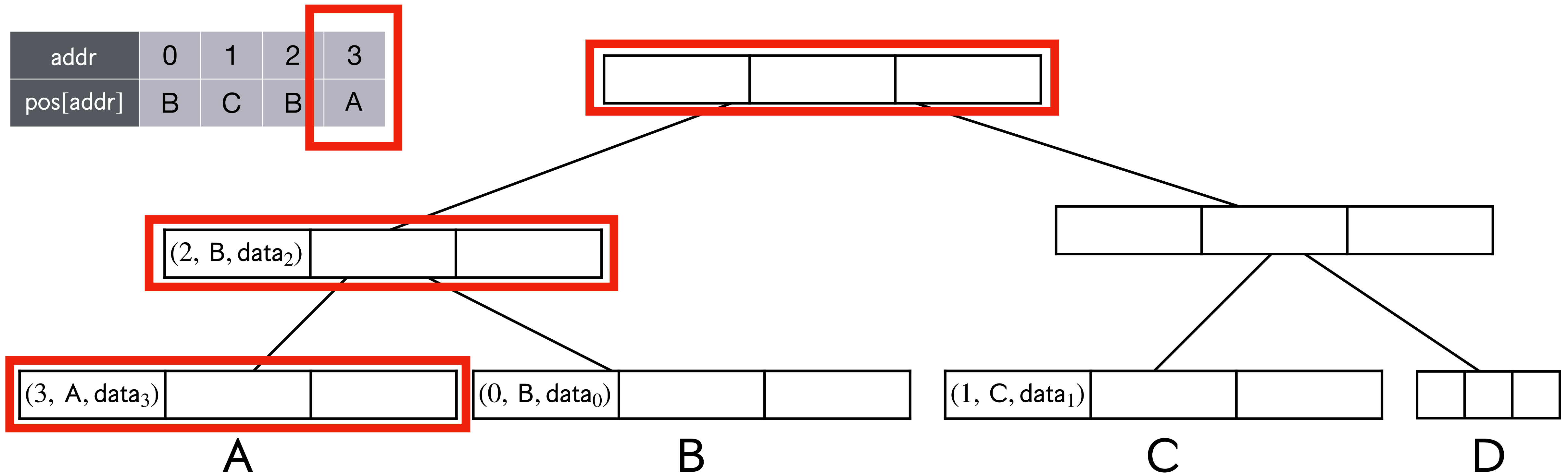
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Example: Read $\text{addr} = 3$

1. **Look up** $\text{pos}[3] = A$ locally.

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

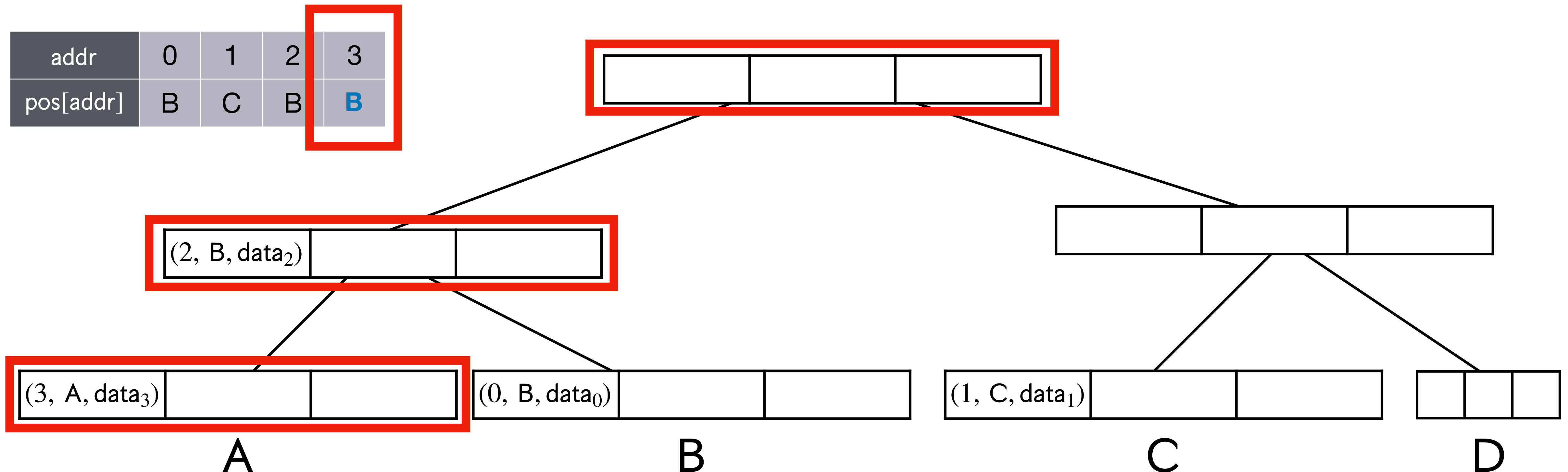
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Example: Read $\text{addr} = 3$

1. **Look up** $\text{pos}[3] = A$ locally.
2. **Read** full path for leaf A.

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

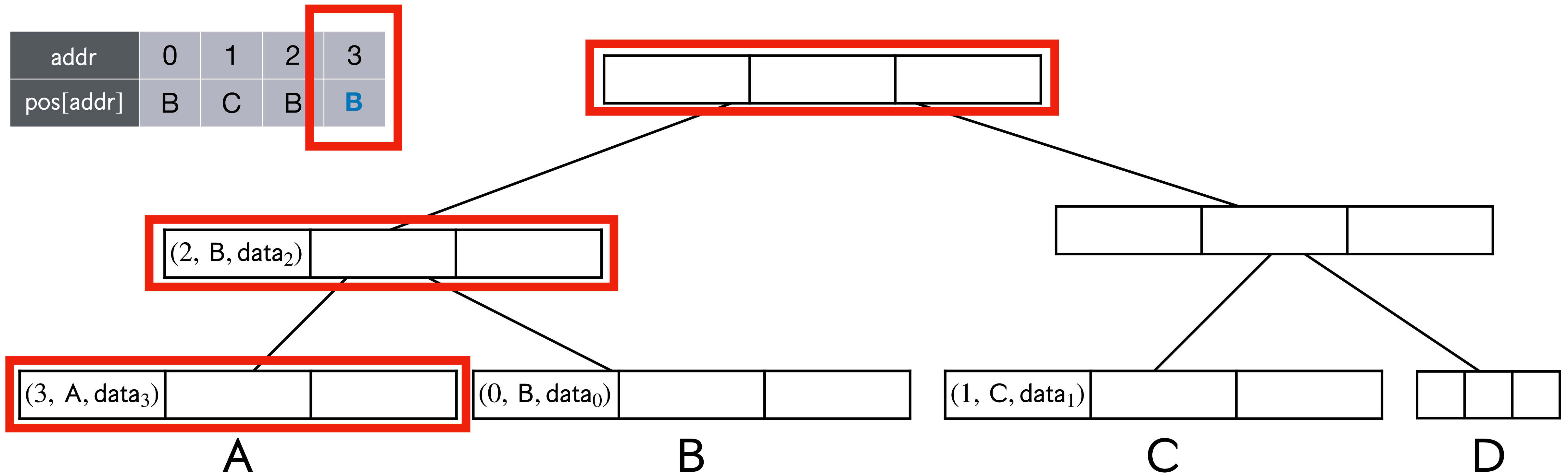
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Example: Read $\text{addr} = 3$

1. **Look up** $\text{pos}[3] = A$ locally.
2. **Read** full path for leaf A.
3. Set $\text{pos}[3] \leftarrow \{A, B, C, D\}$ randomly, say B.

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

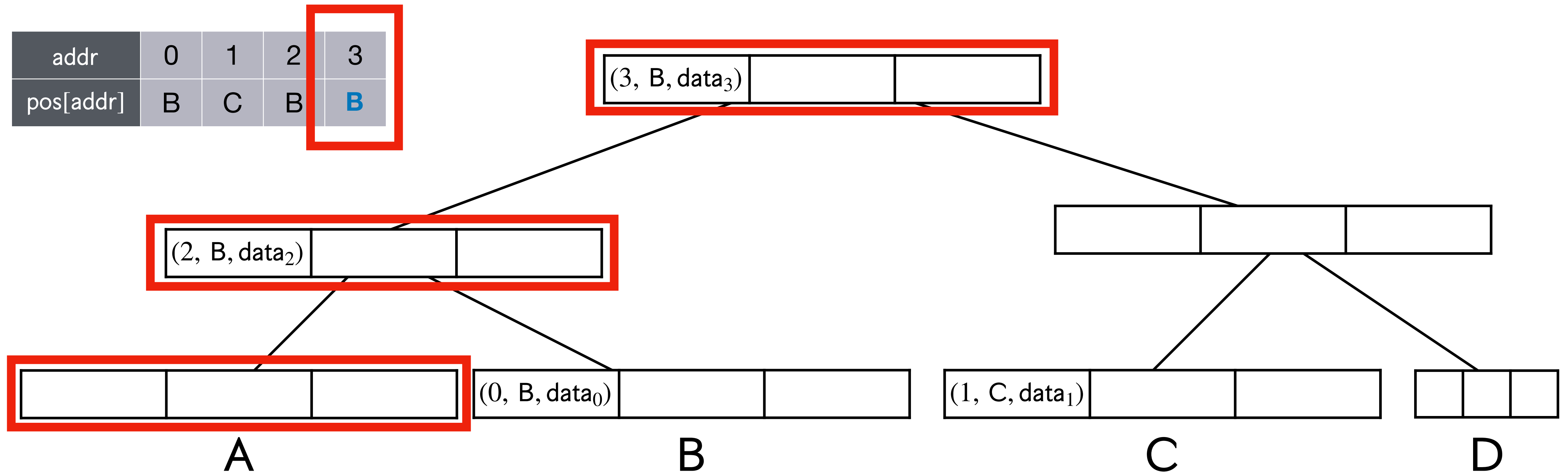
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Example: Read $\text{addr} = 3$

1. **Look up** $\text{pos}[3] = A$ locally.
2. **Read** full path for leaf A.
3. Set $\text{pos}[3] \leftarrow \{A, B, C, D\}$ randomly, say B.
4. **Push new** (and other) blocks down A path.

Path ORAM

(Here, $N = 4$.)



On each query to $addr \in \{0, 1, 2, 3\}$:

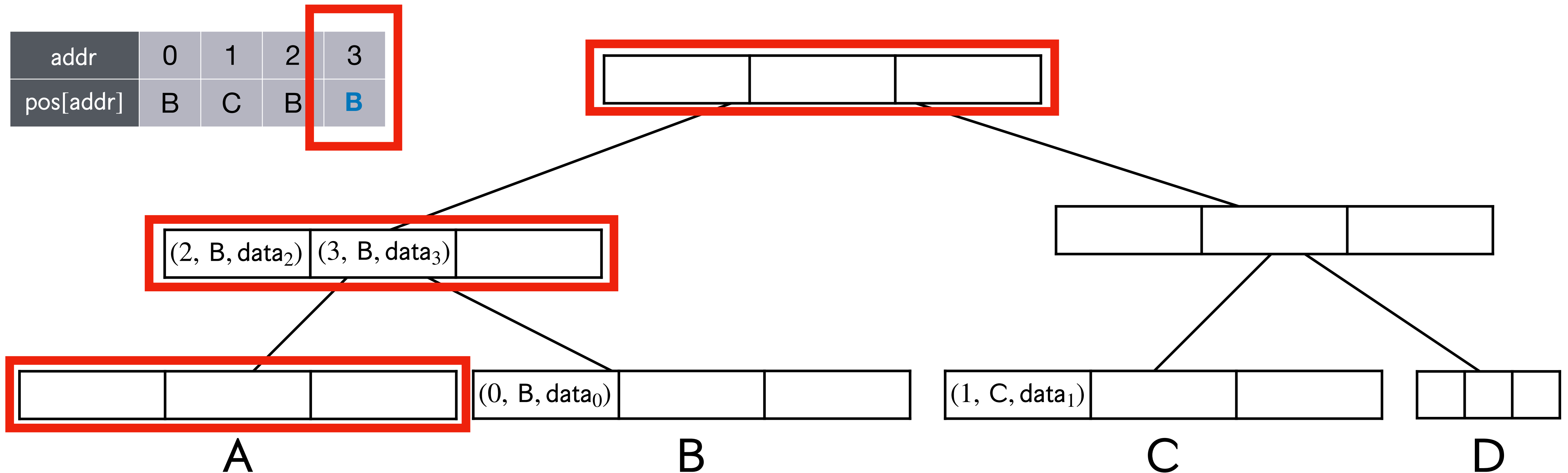
1. **Look up** $pos[addr] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $pos[addr]$.
3. Randomly **sample** new value $pos[addr] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Example: Read $addr = 3$

1. **Look up** $pos[3] = A$ locally.
2. **Read** full path for leaf A.
3. Set $pos[3] \leftarrow \{A, B, C, D\}$ randomly, say B.
4. **Push new** (and other) blocks down A path.

Path ORAM

(Here, $N = 4$.)



On each query to $addr \in \{0, 1, 2, 3\}$:

1. **Look up** $pos[addr] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $pos[addr]$.
3. Randomly **sample** new value $pos[addr] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

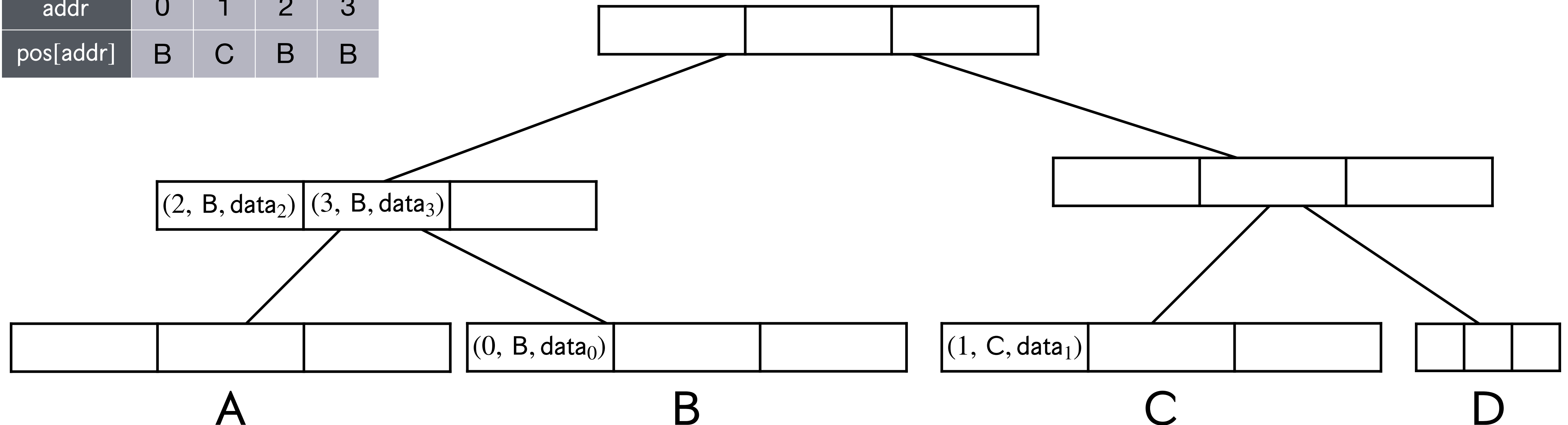
Example: Read $addr = 3$

1. **Look up** $pos[3] = A$ locally.
2. **Read** full path for leaf A.
3. Set $pos[3] \leftarrow \{A, B, C, D\}$ randomly, say B.
4. **Push new** (and other) blocks down A path.

Path ORAM

(Here, $N = 4$.)

addr	0	1	2	3
pos[addr]	B	C	B	B



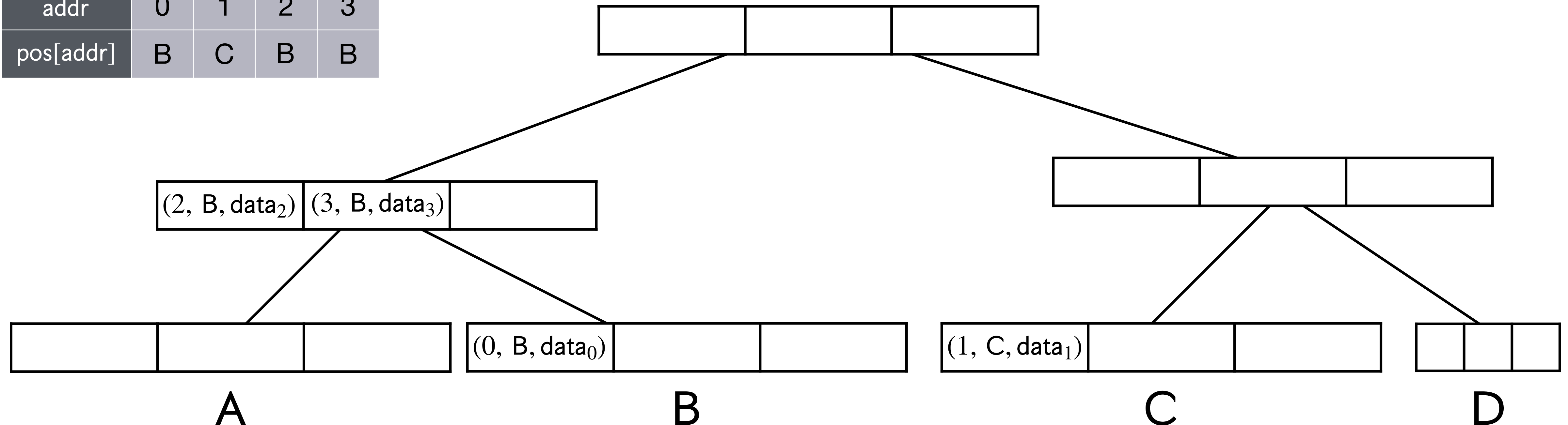
On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Path ORAM

(Here, $N = 4$.)

addr	0	1	2	3
pos[addr]	B	C	B	B



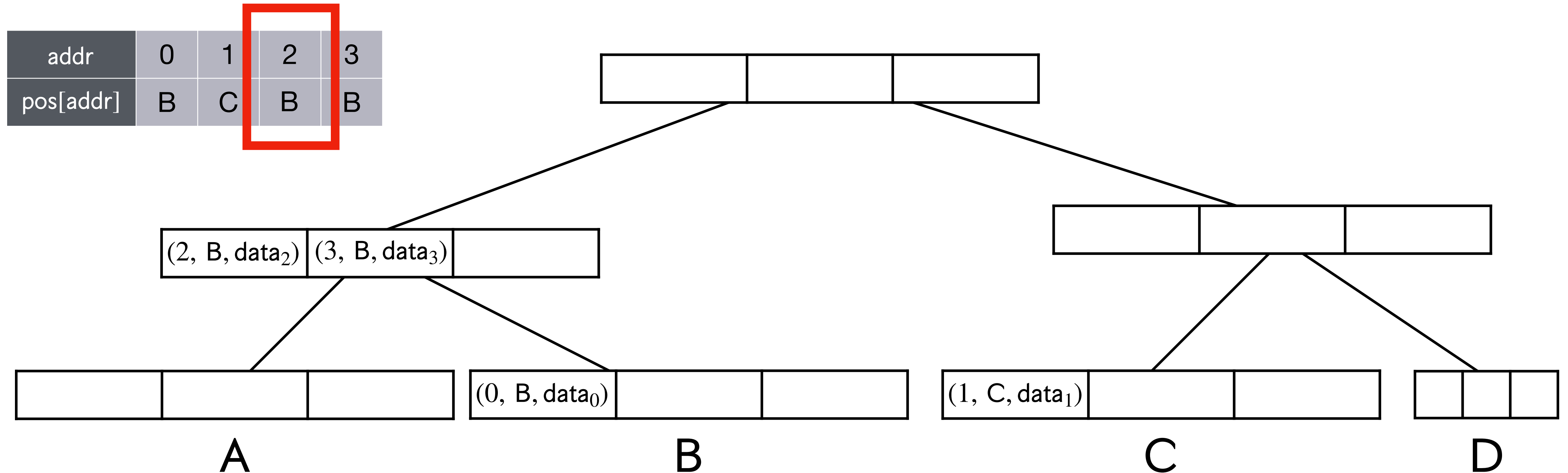
On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Next Example: Write $\text{addr} = 2$ with data'_2 .

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

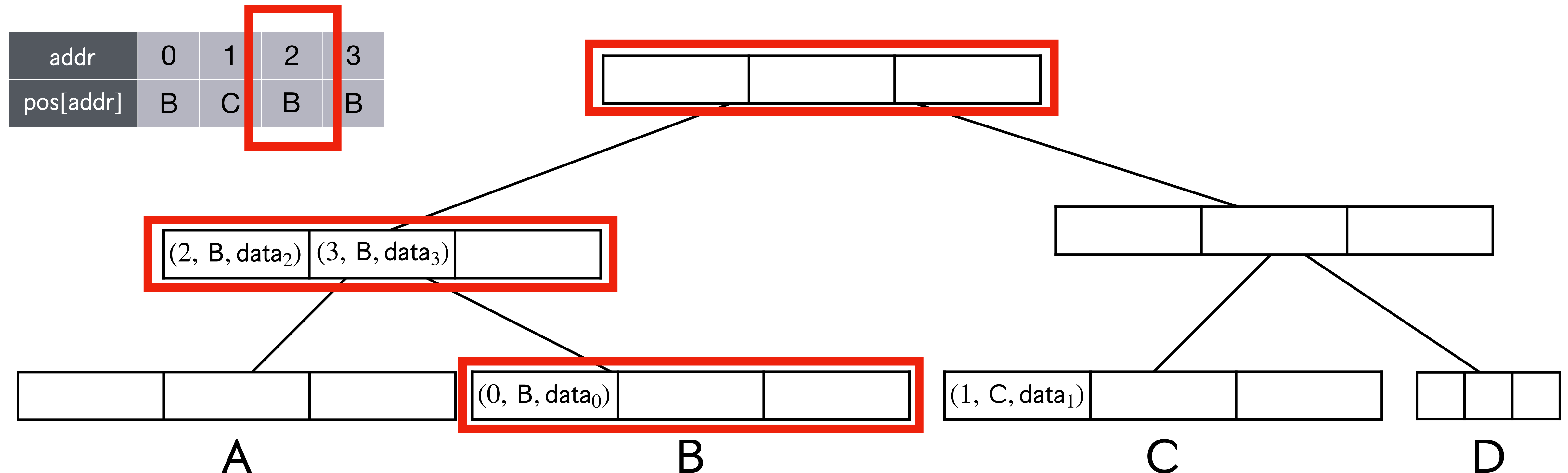
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Next Example: Write $\text{addr} = 2$ with data'_2 .

1. **Look up** $\text{pos}[2] = B$ locally.

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

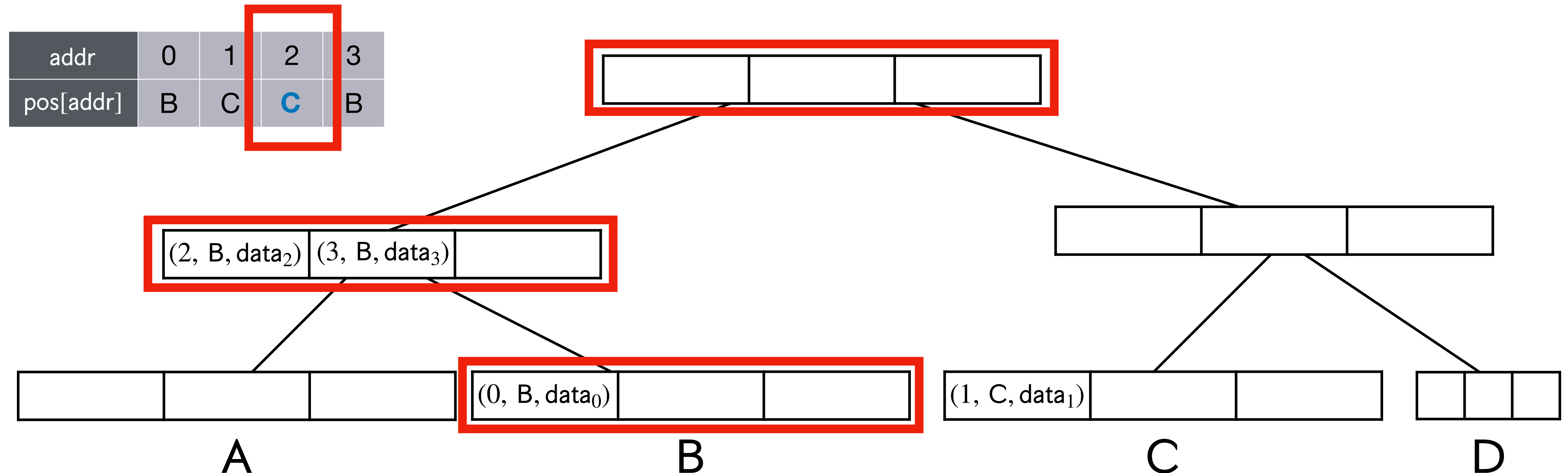
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Next Example: Write $\text{addr} = 2$ with data'_2 .

1. **Look up** $\text{pos}[2] = B$ locally.
2. **Read** full path for leaf B.

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

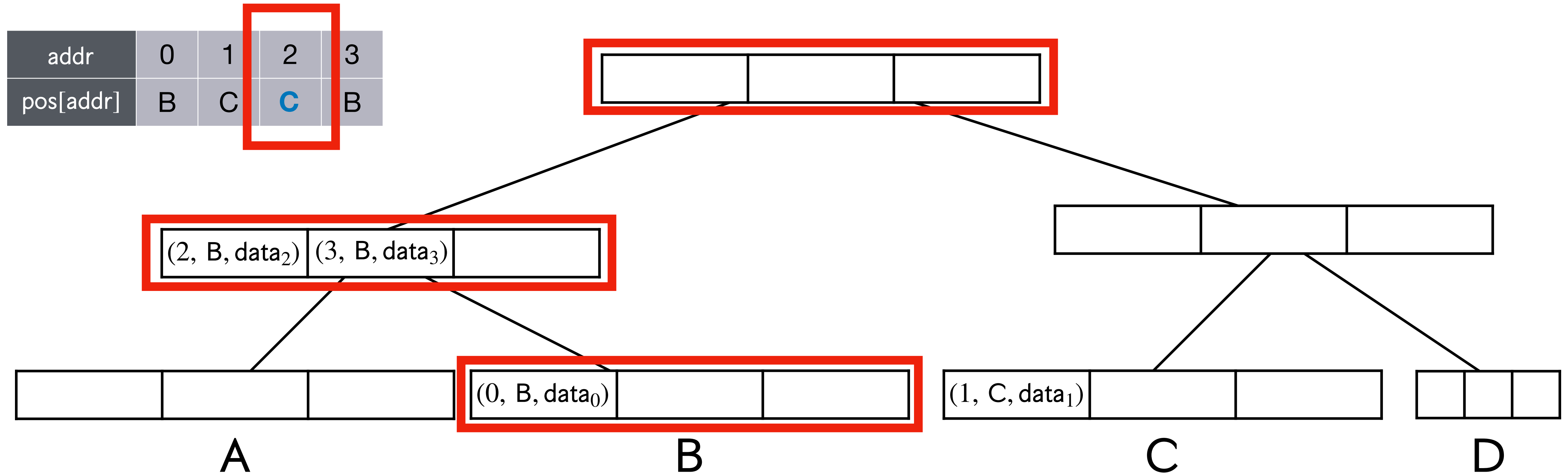
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Next Example: Write $\text{addr} = 2$ with data'_2 .

1. **Look up** $\text{pos}[2] = B$ locally.
2. **Read** full path for leaf B.
3. Set $\text{pos}[2] \leftarrow \{A, B, C, D\}$ randomly, say C.

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

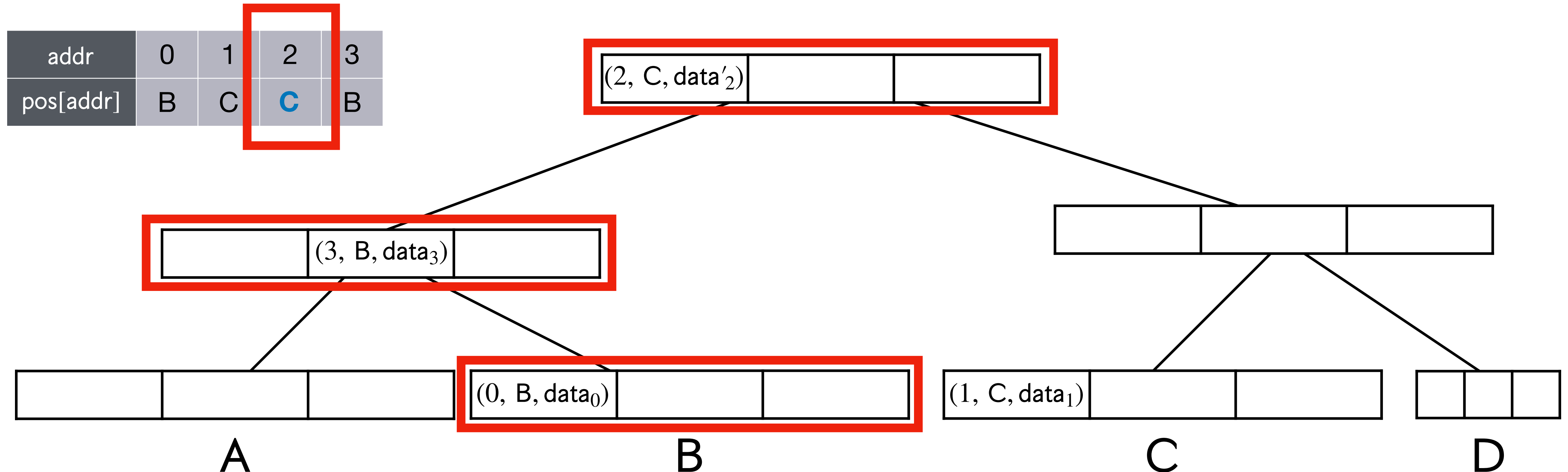
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Next Example: Write $\text{addr} = 2$ with data'_2 .

1. **Look up** $\text{pos}[2] = B$ locally.
2. **Read** full path for leaf B.
3. Set $\text{pos}[2] \leftarrow \{A, B, C, D\}$ randomly, say C.
4. **Push new** (and other) blocks down B path.

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

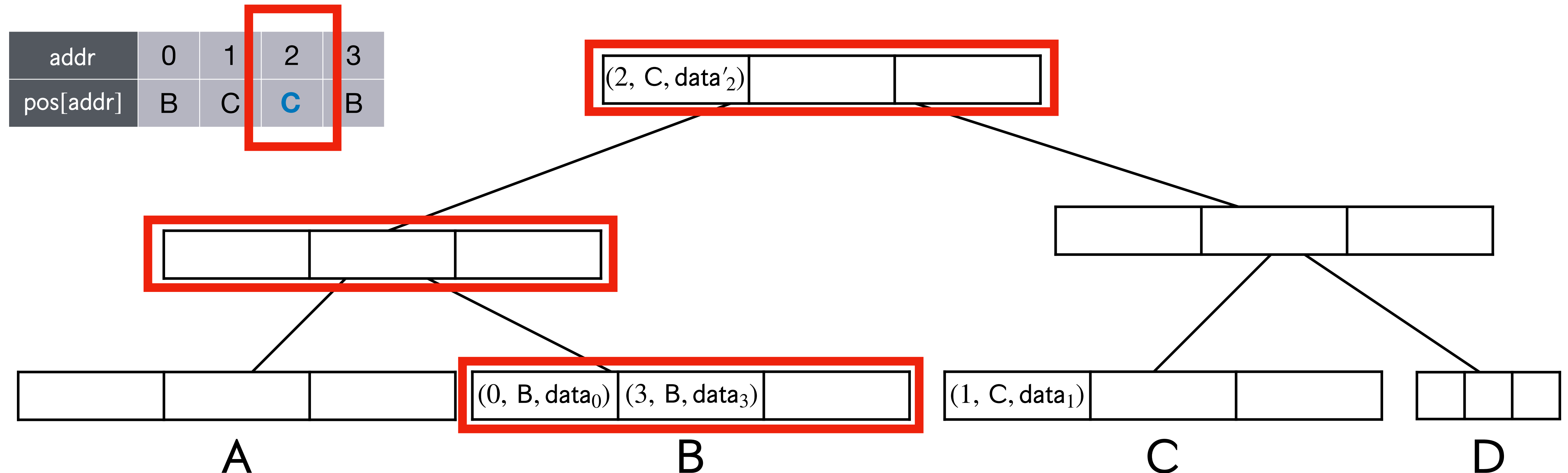
1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Next Example: Write $\text{addr} = 2$ with data'_2 .

1. **Look up** $\text{pos}[2] = B$ locally.
2. **Read** full path for leaf B.
3. Set $\text{pos}[2] \leftarrow \{A, B, C, D\}$ randomly, say C.
4. **Push new** (and other) blocks down B path.

Path ORAM

(Here, $N = 4$.)



On each query to $\text{addr} \in \{0, 1, 2, 3\}$:

1. **Look up** $\text{pos}[\text{addr}] \in \{A, B, C, D\}$ locally.
2. **Read** full path for leaf $\text{pos}[\text{addr}]$.
3. Randomly **sample** new value $\text{pos}[\text{addr}] \leftarrow \{A, B, C, D\}$.
4. **Push new** (and other) data blocks down old path.

Next Example: Write $\text{addr} = 2$ with data'_2 .

1. **Look up** $\text{pos}[2] = B$ locally.
2. **Read** full path for leaf B.
3. Set $\text{pos}[2] \leftarrow \{A, B, C, D\}$ randomly, say C.
4. **Push new** (and other) blocks down B path.

Analysis

Analysis

- **Overhead:** $\approx \log N!$

Analysis

- **Overhead:** $\approx \log N!$
- Local space:

Analysis

- **Overhead:** $\approx \log N!$
- Local space:
 - Position map: $\approx N \log N$ bits...

Analysis

- **Overhead:** $\approx \log N!$
- Local space:
 - **Position map:** $\approx N \log N$ bits...
- If each data was $\approx 2 \log N$ bits long, then we are compressing database by factor of 2!

Analysis

- **Overhead:** $\approx \log N!$
- Local space:
 - **Position map:** $\approx N \log N$ bits...
 - If each data was $\approx 2 \log N$ bits long, then we are compressing database by factor of 2!
 - So: recurse! Will be become $\log N$ levels, giving overhead $\log^2(N)$.

Analysis

- **Overhead:** $\approx \log N!$
- Local space:
 - **Position map:** $\approx N \log N$ bits...
 - If each data was $\approx 2 \log N$ bits long, then we are compressing database by factor of 2!
 - So: recurse! Will be become $\log N$ levels, giving overhead $\log^2(N)$.
 - (Technicality: also need to store $\omega(\log N)$ -sized stash to prevent bucket overflow.)

Security

Security

- Why is this oblivious?

Security

- Why is this oblivious?
 - Every query, the lookup is to an independent, uniformly random leaf!

Security

- Why is this oblivious?
 - Every query, the lookup is to an independent, uniformly random leaf!
 - Everything else is hidden by encryption.

Path ORAM is Used in Practice!

Path ORAM is Used in Practice!

- Signal previously used linear scans (trivial overhead N ORAM) for private contact discovery.



**Solving Privacy and Integrity
Simultaneously:
Maliciously Secure ORAM**

Exercise

Show that Path ORAM is *not* maliciously secure, in the sense that a tampering adversary can **distinguish** between different user queries.

Solving Issue 3: Maliciously Secure ORAM

Solving Issue 3: Maliciously Secure ORAM

- Intuitively, memory checking seems to solve the issue of a tampering adversary in ORAM. Combine them!

Solving Issue 3: Maliciously Secure ORAM

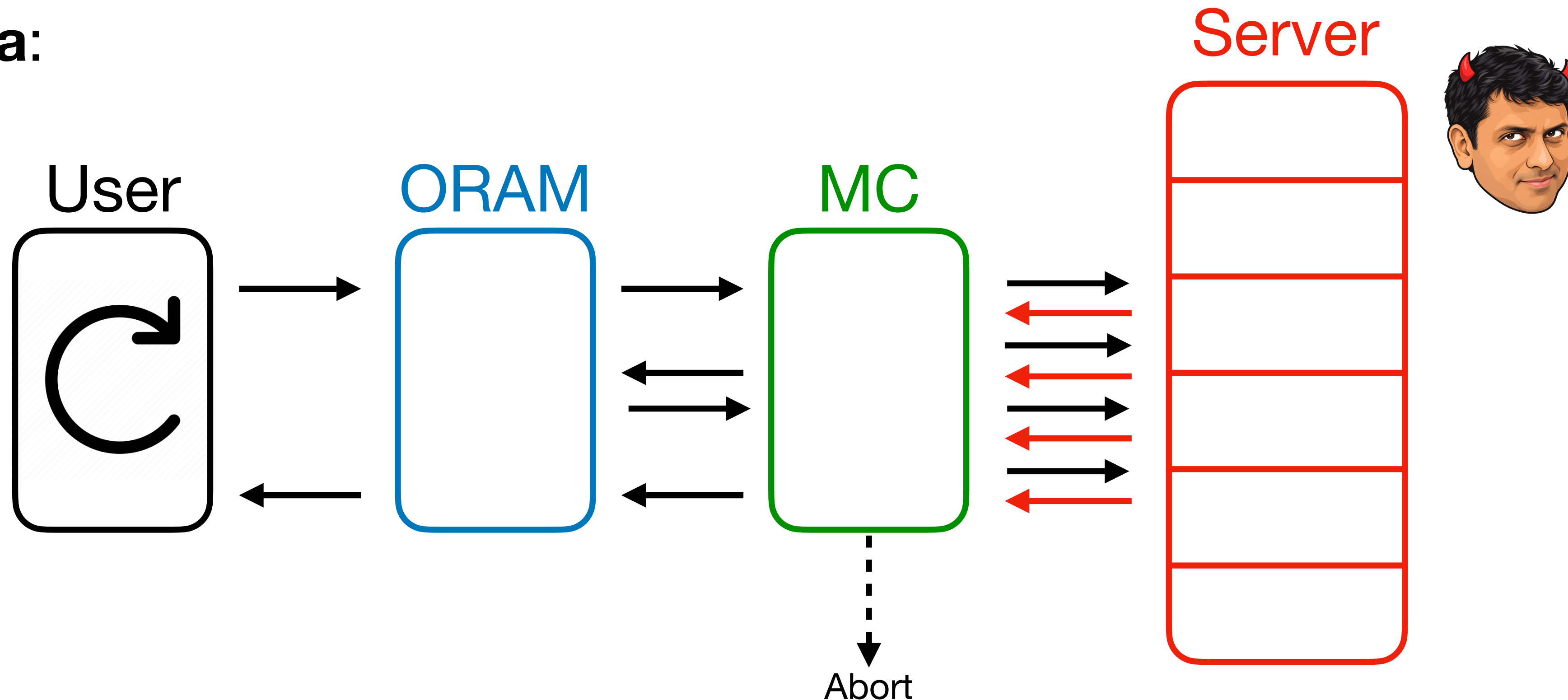
- Intuitively, memory checking seems to solve the issue of a tampering adversary in ORAM. Combine them!
- **Theorem:** Honest-but-curious ORAM + MC = maliciously secure ORAM.

Solving Issue 3: Maliciously Secure ORAM

- Intuitively, memory checking seems to solve the issue of a tampering adversary in ORAM. Combine them!
- **Theorem:** Honest-but-curious ORAM + MC = maliciously secure ORAM.
- **Idea:**

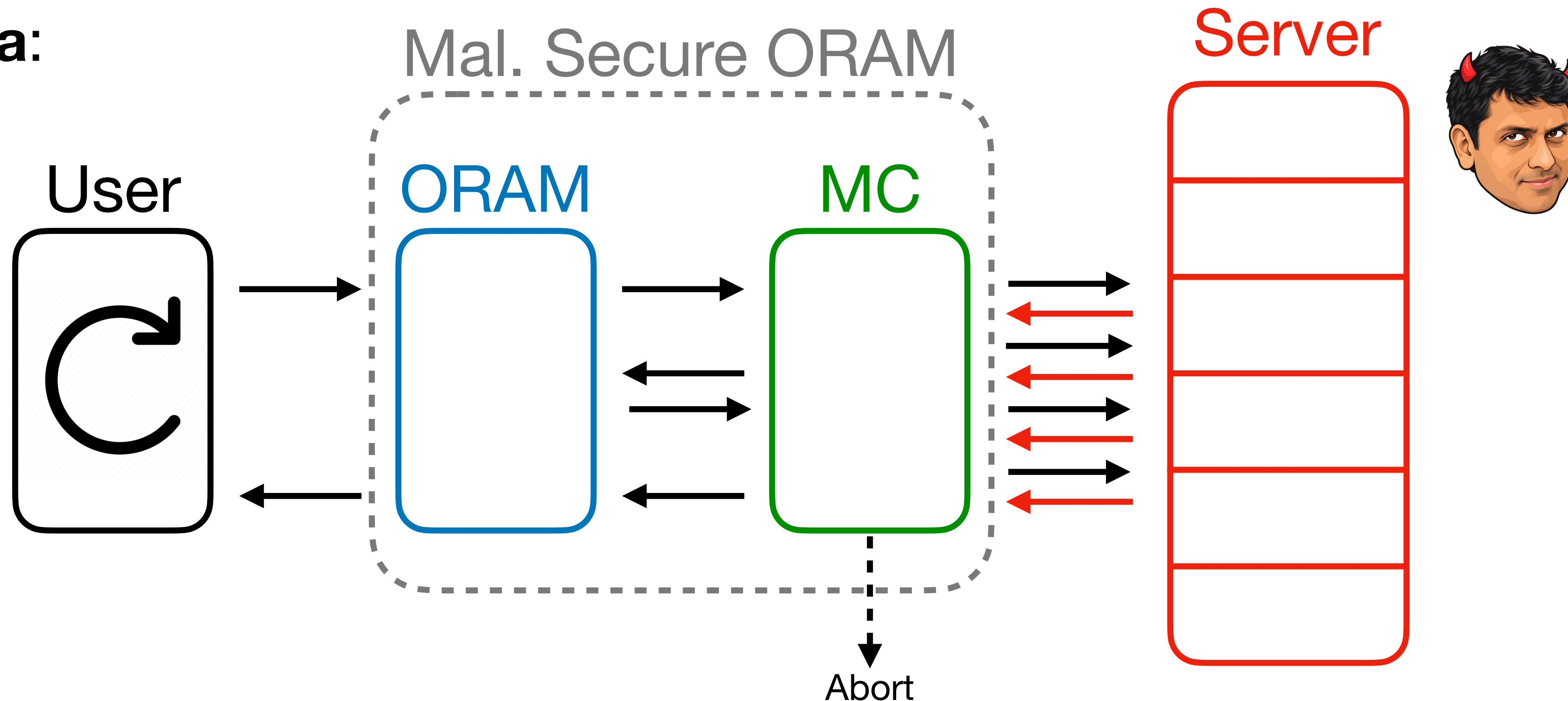
Solving Issue 3: Maliciously Secure ORAM

- Intuitively, memory checking seems to solve the issue of a tampering adversary in ORAM. Combine them!
- **Theorem:** Honest-but-curious ORAM + MC = maliciously secure ORAM.
- **Idea:**



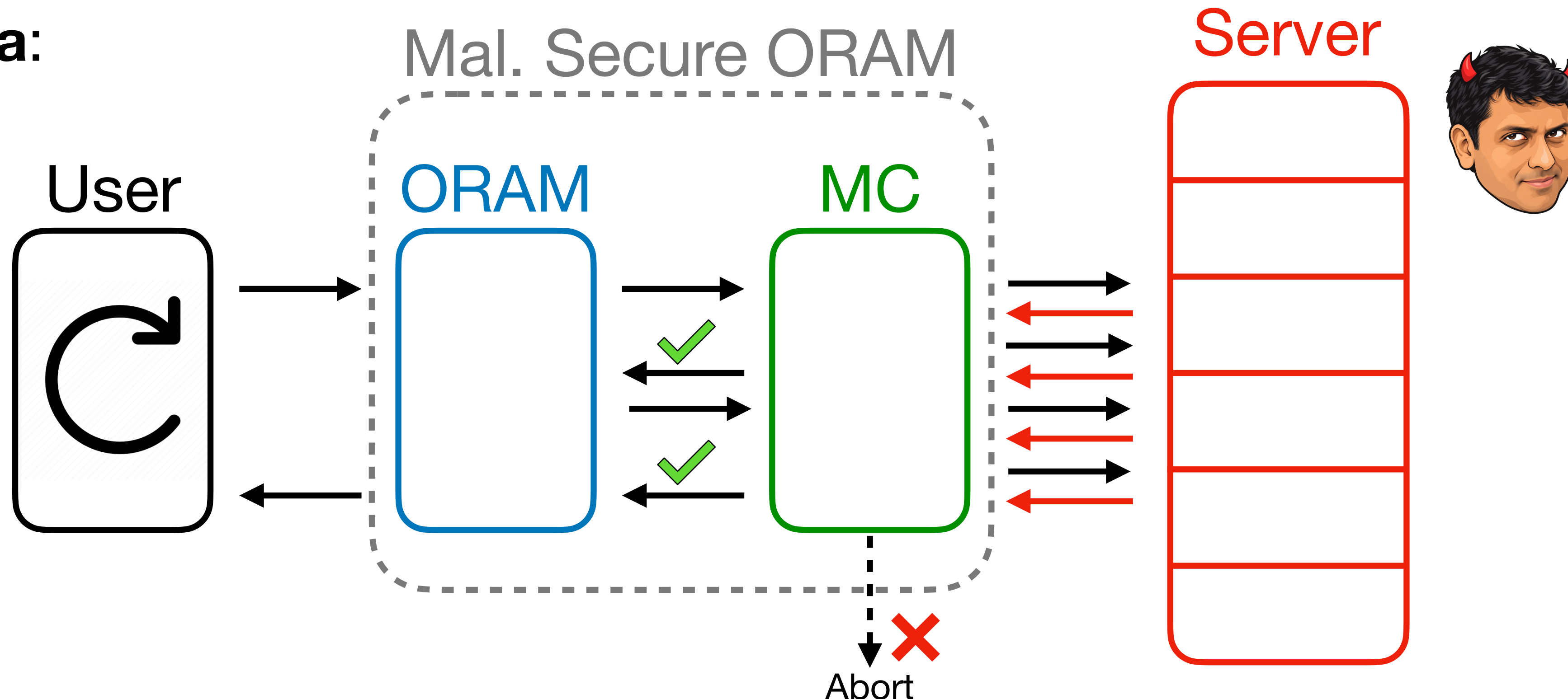
Solving Issue 3: Maliciously Secure ORAM

- Intuitively, memory checking seems to solve the issue of a tampering adversary in ORAM. Combine them!
- **Theorem:** Honest-but-curious ORAM + MC = maliciously secure ORAM.
- **Idea:**



Solving Issue 3: Maliciously Secure ORAM

- Intuitively, memory checking seems to solve the issue of a tampering adversary in ORAM. Combine them!
- **Theorem:** Honest-but-curious ORAM + MC = maliciously secure ORAM.
- **Idea:**



Solving Issue 3: Maliciously Secure ORAM

- Great! But this isn't efficient enough.

Solving Issue 3: Maliciously Secure ORAM

- Great! But this isn't efficient enough.

$$\text{Overhead}(\text{ORAM}_{\text{Mal}}) = \text{Overhead}(\text{ORAM}_{\text{HBC}}) \cdot \text{Overhead}(\text{MC})$$

Solving Issue 3: Maliciously Secure ORAM

- Great! But this isn't efficient enough.

$$\text{Overhead}(\text{ORAM}_{\text{Mal}}) = \text{Overhead}(\text{ORAM}_{\text{HBC}}) \cdot \text{Overhead}(\text{MC})$$

$$\uparrow$$
$$\log^2(N)$$

Solving Issue 3: Maliciously Secure ORAM

- Great! But this isn't efficient enough.

$$\text{Overhead}(\text{ORAM}_{\text{Mal}}) = \text{Overhead}(\text{ORAM}_{\text{HBC}}) \cdot \text{Overhead}(\text{MC})$$

$$\uparrow$$
$$\log^2(N)$$

$$\uparrow$$
$$\log N$$

Solving Issue 3: Maliciously Secure ORAM

- Great! But this isn't efficient enough.

$$\text{Overhead}(\text{ORAM}_{\text{Mal}}) = \text{Overhead}(\text{ORAM}_{\text{HBC}}) \cdot \text{Overhead}(\text{MC})$$


$$\log^3(N)$$


$$\log^2(N)$$


$$\log N$$

Solving Issue 3: Maliciously Secure ORAM

- Great! But this isn't efficient enough.

$$\text{Overhead}(\text{ORAM}_{\text{Mal}}) = \text{Overhead}(\text{ORAM}_{\text{HBC}}) \cdot \text{Overhead}(\text{MC})$$


$$\log^3(N)$$


$$\log^2(N)$$


$$\log N$$

- Can we non-trivially combine the two constructions we saw?

Yes!

Yes!

- They're both trees! Do them both at the same time!

Yes!

- They're both trees! Do them both at the same time!
- Specifically, for both constructions, each user query results in a lookup of the path from the root to the tree.

Yes!

- They're both trees! Do them both at the same time!
- Specifically, for both constructions, each user query results in a lookup of the path from the root to the tree.
- Run Path ORAM, and store and compute hashes along the way.

Yes!

- They're both trees! Do them both at the same time!
- Specifically, for both constructions, each user query results in a lookup of the path from the root to the tree.
- Run Path ORAM, and store and compute hashes along the way.
- **Result:** Maliciously secure ORAM with $O(\log^2 N)$ overhead!

Questions?