The Complexity of Secure RAMs

Giuseppe Persiano

Università di Salerno and Google LLC

CIAC in Cyprus – June 15, 2023

Cloud Storage (simplified)

The perfect marriage of two parties

- The Data Owner $\mathcal{O}$:
  owns large amount of data and not enough local storage
- The Storage Manager $\mathcal{M}$:
  owns large amount of storage and not enough data

Cloud Storage (simplified)

The perfect marriage of two parties

- The Data Owner $\mathcal{O}$:
  owns large amount of data and not enough local storage
- The Storage Manager $\mathcal{M}$:
  owns large amount of storage and not enough data

If $\mathcal{O}$ and $\mathcal{M}$ trust each other

Cloud Storage (simplified)

The perfect marriage of two parties

- The Data Owner $\mathcal{O}$:
  owns large amount of data and not enough local storage
- The Storage Manager $\mathcal{M}$:
  owns large amount of storage and not enough data

If $\mathcal{O}$ and $\mathcal{M}$ trust each other no problem. we can go home now.

Cloud Storage (simplified)

The perfect marriage of two parties

- The Data Owner $\mathcal{O}$:
  owns large amount of data and not enough local storage
- The Storage Manager $\mathcal{M}$:
  owns large amount of storage and not enough data

If $\mathcal{O}$ and $\mathcal{M}$ trust each other no problem. we can go home now.

Lack of trust is much more interesting.

Enter Encryption

$\mathcal{O}$ does not trust $\mathcal{M}$ because $\mathcal{O}$'s data contain personal data.

Enter Encryption

$\mathcal{O}$ should not trust $\mathcal{M}$ because $\mathcal{O}$'s data contain personal data.

Enter Encryption

$\mathcal{O}$ should not trust $\mathcal{M}$ because $\mathcal{O}$'s data contain personal data.

Use Encryption

- Private Key: if $\mathcal{O}$ is the source of data
- Public Key: if data come from various sources

Enter Encryption

$\mathcal{O}$ should not trust $\mathcal{M}$ because $\mathcal{O}$'s data contain personal data.

Use Encryption

- Private Key: if $\mathcal{O}$ is the source of data
- Public Key: if data come from various sources

Data is

- encrypted before being uploaded to $\mathcal{M}$

Enter Encryption

$\mathcal{O}$ should not trust $\mathcal{M}$ because $\mathcal{O}$'s data contain personal data.

Use Encryption

- Private Key: if $\mathcal{O}$ is the source of data
- Public Key: if data come from various sources

Data is

- encrypted before being uploaded to $\mathcal{M}$
- decrypted when downloaded from $\mathcal{M}$

Are we done?

What if $\mathcal{O}$ wants to run an algorithm on the encrypted data?

Running an algorithm might reveal information on the data.

Suppose $\mathcal{O}$ wants to sort the data.

Are we done?

What if $\mathcal{O}$ wants to run an algorithm on the encrypted data?

Running an algorithm might reveal information on the data.

Suppose $\mathcal{O}$ wants to sort the data.

Example

4 customers must be sorted according to revenue.

A;200	B;300	C;100	D;150

download 1 and 3. decrypt, swap if out of order, re-encrypt, upload.

Are we done?

What if $\mathcal{O}$ wants to run an algorithm on the encrypted data?

Running an algorithm might reveal information on the data.

Suppose $\mathcal{O}$ wants to sort the data.

Example

4 customers must be sorted according to revenue.

C;100	B;300	A;200	D;150

download 2 and 4. decrypt, swap if out of order, re-encrypt, upload.

Are we done?

What if $\mathcal{O}$ wants to run an algorithm on the encrypted data?

Running an algorithm might reveal information on the data.

Suppose $\mathcal{O}$ wants to sort the data.

Example

4 customers must be sorted according to revenue.

C;100	D;150	A;200	B;300

download 1 and 2. decrypt, swap if out of order, re-encrypt, upload.

Are we done?

What if $\mathcal{O}$ wants to run an algorithm on the encrypted data?

Running an algorithm might reveal information on the data.

Suppose $\mathcal{O}$ wants to sort the data.

Example

4 customers must be sorted according to revenue.

C;100	D;150	A;200	B;300

download 3 and 4. decrypt, swap if out of order, re-encrypt, upload.

Are we done?

What if $\mathcal{O}$ wants to run an algorithm on the encrypted data?

Running an algorithm might reveal information on the data.

Suppose $\mathcal{O}$ wants to sort the data.

Example

4 customers must be sorted according to revenue.

C;100	D;150	A;200	B;300

download 2 and 3. decrypt, swap if out of order, re-encrypt, upload.

Are we done?

What if $\mathcal{O}$ wants to run an algorithm on the encrypted data?

Running an algorithm might reveal information on the data.

Suppose $\mathcal{O}$ wants to sort the data.

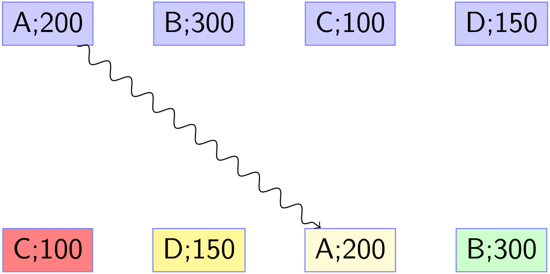
Example

4 customers must be sorted according to revenue.

C;100 D;150 A;200 B;300

Security

Can $\mathcal{M}$ link the first record in the starting configuration to its position in the last configuration?



Two Concepts

Indistinguishability of *Swap or Not*

- Download, Decrypt, Swap or Not, Re-encrypt, Upload

Two Concepts

Indistinguishability of *Swap or Not*

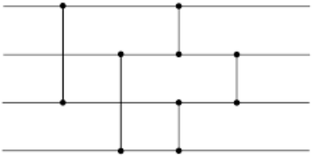
- Download, Decrypt, Swap or Not, Re-encrypt, Upload

Chosen-Plaintext Security: Standard notion of security for encryption guarantee that $\mathcal{M}$ is unable to infere whether a swap has taken place.

Enter Obliviousness

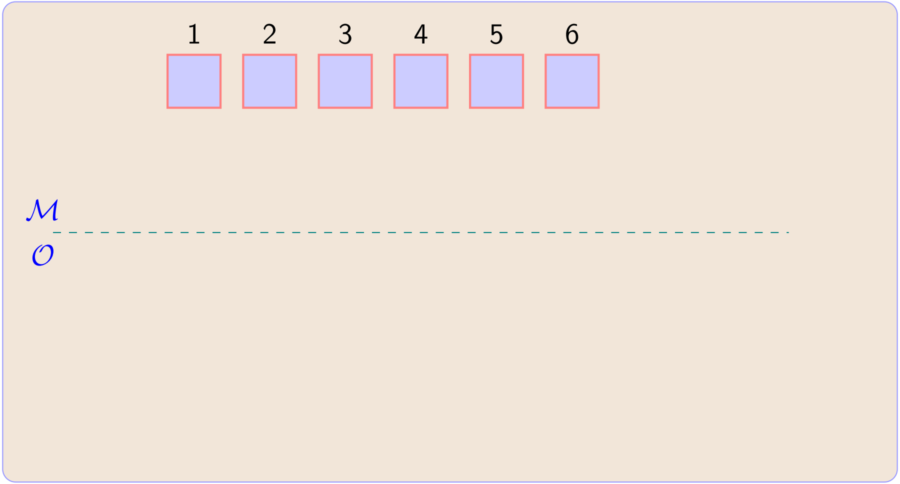
Definition (Weak Obliviousness)

An algorithm is *weakly oblivious* if the *access pattern* to data is the same for all possible inputs of the same length.

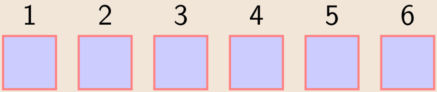


Thanks to Wikipedia for the image

The adversarial setting



The adversarial setting

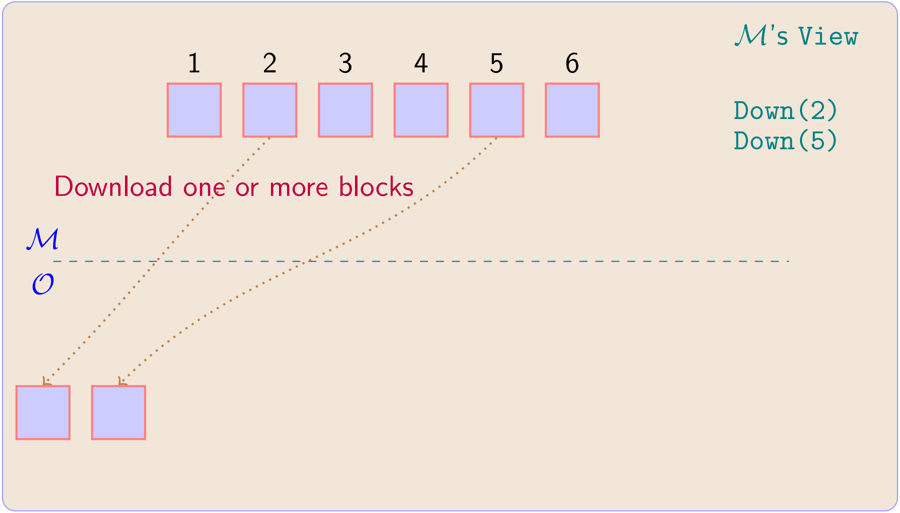


Download one or more blocks

$\mathcal{M}$

$\mathcal{O}$

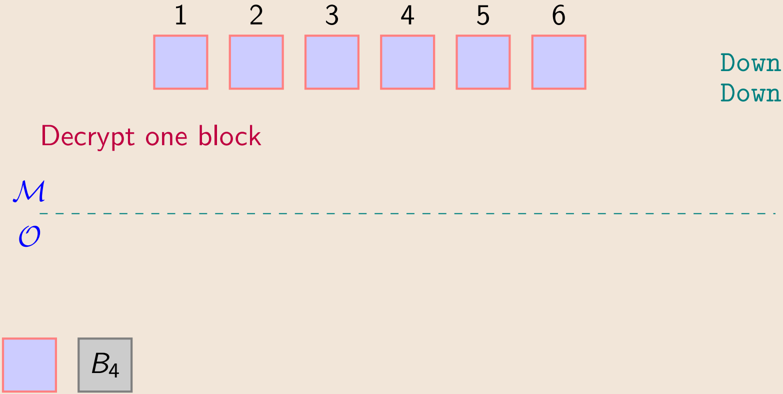
The adversarial setting



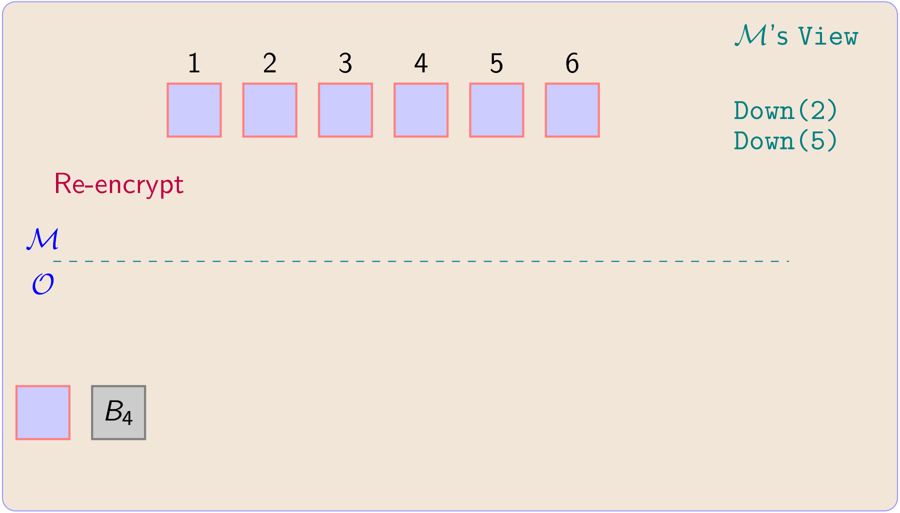
The adversarial setting

$\mathcal{M}$'s `View`

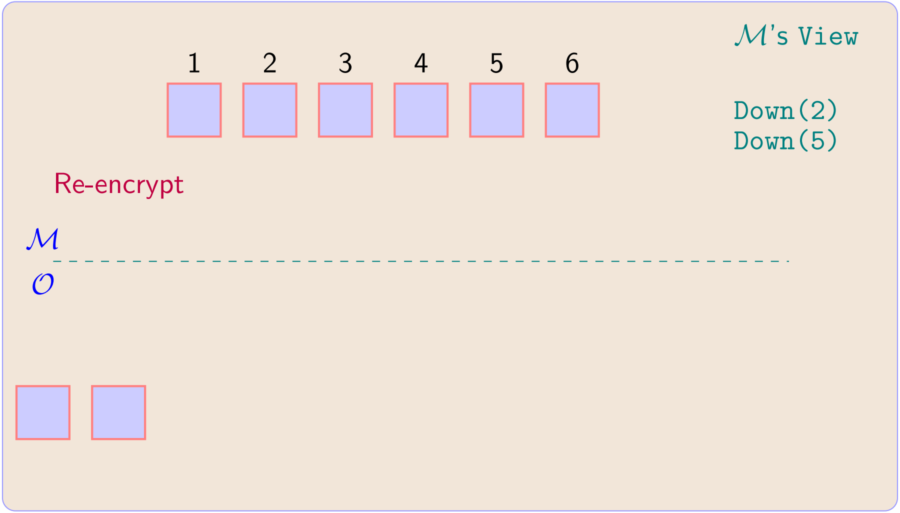
`Down(2)`
`Down(5)`



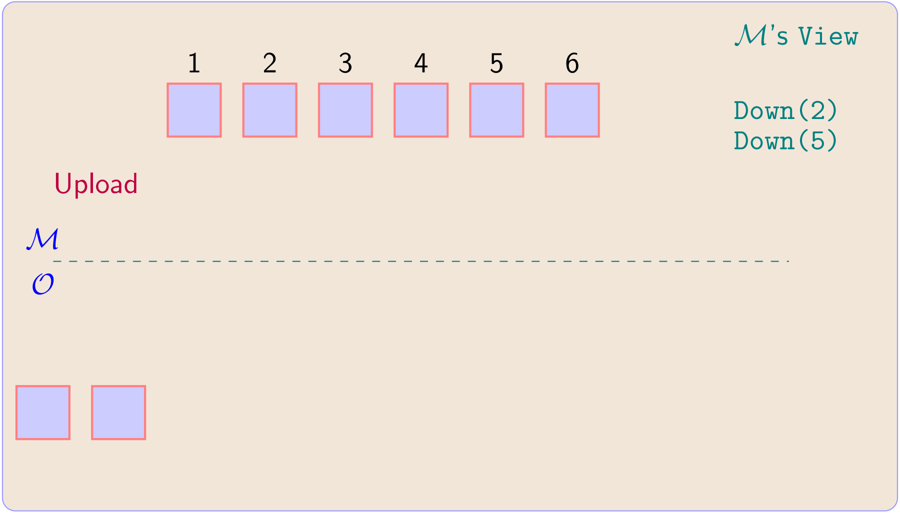
The adversarial setting



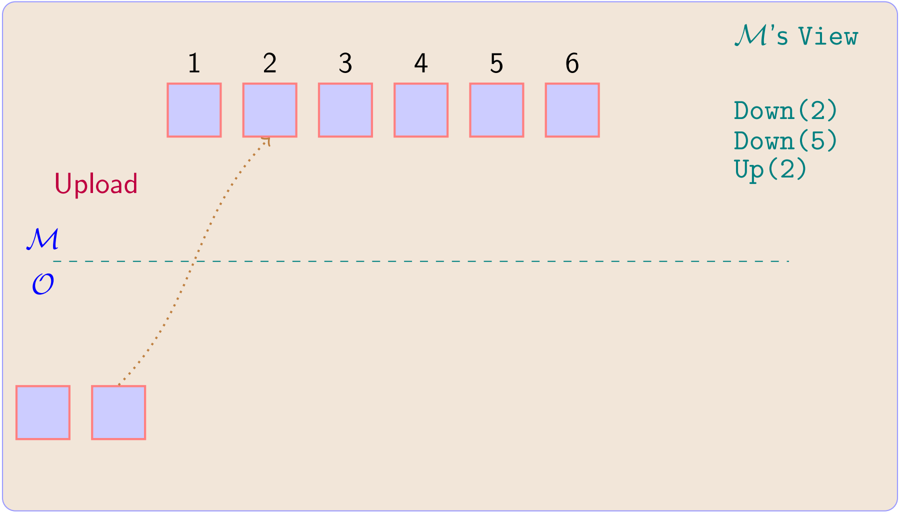
The adversarial setting



The adversarial setting



The adversarial setting



A new industry

Job Opportunities for Algorithmists

- Re-design all algorithms to be oblivious!
- Remove all *if*s, and *while*s
- Insertion Sort is not oblivious:
  - when the last element of the array is inserted, $\mathcal{M}$ sees where it lands

Hiding the Algorithm

A new threat

- which algorithm is being run should also be private information

A;200	B;300	C;100	D;150

Hiding the Algorithm

A new threat

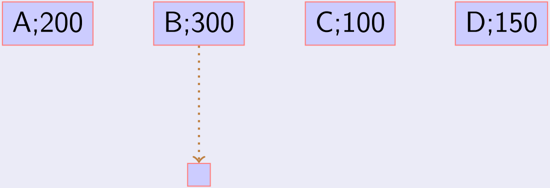
- which algorithm is being run should also be private information

A;200 B;300 C;100 D;150

Hiding the Algorithm

A new threat

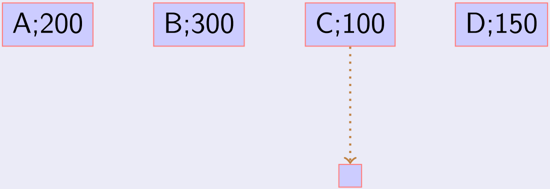
- which algorithm is being run should also be private information



Hiding the Algorithm

A new threat

- which algorithm is being run should also be private information



Hiding the Algorithm

A new threat

- which algorithm is being run should also be private information

A;200 B;300 C;100 D;150

Enter Oblivious RAM

Differential Privacy

ORAM [Goldreich-Ostrovsky]

- $\mathcal{M}$ stores n blocks of memory.
- Every time $\mathcal{O}$ wants a block, he asks $\mathcal{M}$ one or more blocks.
- Security notion:
  - For any two block sequences $\mathbb{B} = B_1, \ldots, B_n$ and $\mathbb{C} = C_1, \ldots, C_n$
  - For any two access sequences $I = (i_1, \ldots, i_l)$ and $J = (j_1, \ldots, j_l)$
    - performing accesses $i_1, \ldots, i_l$ on $\mathbb{B} = B_1, \ldots, B_n$;
    - performing access $j_1, \ldots, j_l$ on $\mathbb{C} = C_1, \ldots, C_n$

  generate the same distribution of accesses to the data stored by $\mathcal{M}$

Enter Oblivious RAM

Differential Privacy

ORAM [Goldreich-Ostrovsky]

- $\mathcal{M}$ stores n blocks of memory.
- Every time $\mathcal{O}$ wants a block, he asks $\mathcal{M}$ one or more blocks.
- Security notion:
  - For any two block sequences $\mathbb{B} = B_1, \ldots, B_n$ and $\mathbb{C} = C_1, \ldots, C_n$
  - For any two access sequences $I = (i_1, \ldots, i_l)$ and $J = (j_1, \ldots, j_l)$
    - performing accesses $i_1, \ldots, i_l$ on $\mathbb{B} = B_1, \ldots, B_n$;
    - performing access $j_1, \ldots, j_l$ on $\mathbb{C} = C_1, \ldots, C_n$

  generate the same distribution of accesses to the data stored by $\mathcal{M}$

For every predicate A

$$\mathrm{Prob}[\mathtt{view} \leftarrow \mathtt{View}(I, \mathbb{B}) : A(\mathtt{view}) = 1]$$
$$\leq e^0 \cdot \mathrm{Prob}[\mathtt{view} \leftarrow \mathtt{View}(J, \mathbb{C}) : A(\mathtt{view}) = 1] + \mathsf{negl}(n)$$

ORAM makes all Algorithms Oblivious

Composing ORAM and Non-Oblivious Algorithms

- $\mathcal{O}$ runs the algorithm
- when a block of memory is requested, $\mathcal{O}$ retrieves it from $\mathcal{M}$ using ORAM.

ORAM makes all Algorithms Oblivious

Composing ORAM and Non-Oblivious Algorithms

- $\mathcal{O}$ runs the algorithm
- when a block of memory is requested, $\mathcal{O}$ retrieves it from $\mathcal{M}$ using ORAM.

Is ORAM possible at all?

Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.

$\boxed{B_1}\ \boxed{B_2}\ \boxed{B_3}\ \boxed{B_4}\ \boxed{B_5}\ \boxed{B_6}$

- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.

Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.

B_1	B_2	B_3	B_4	B_5	B_6

- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.

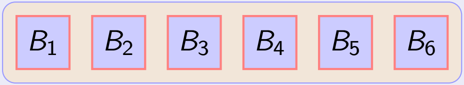
Accessing block B_3

B_1	B_2	B_3	B_4	B_5	B_6

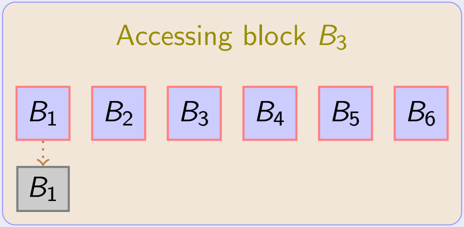
Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.



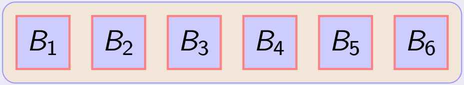
- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.



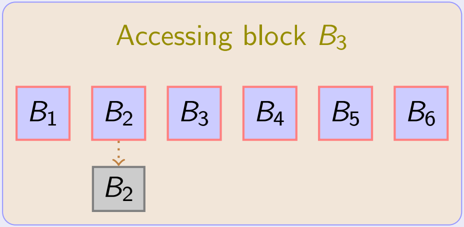
Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.



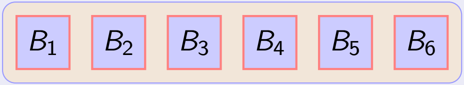
- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.



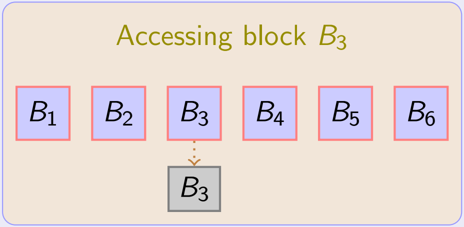
Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.



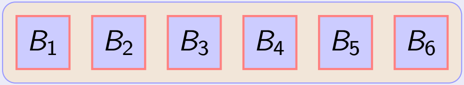
- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.



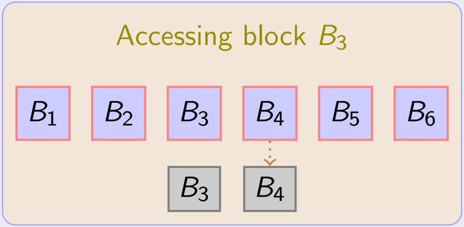
Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.



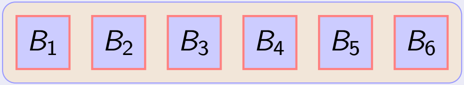
- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.



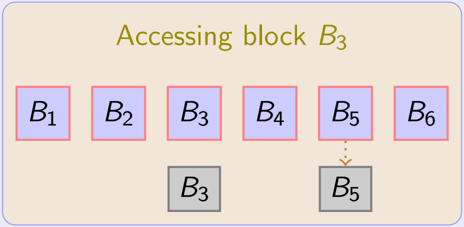
Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.



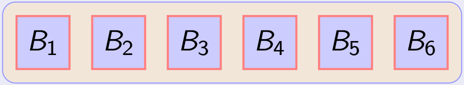
- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.



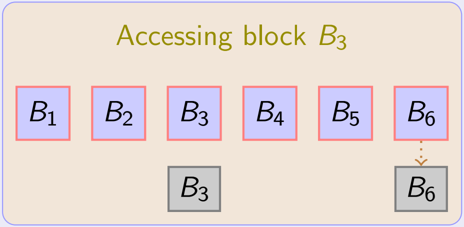
Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.



- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.



Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.

B_1 B_2 B_3 B_4 B_5 B_6

- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.

Accessing block B_3

B_1 B_2 B_3 B_4 B_5 B_6

B_3

Access pattern independent from the block accessed but...

Yes! This is possible!

A Trivial ORAM

- All blocks are uploaded to $\mathcal{M}$ in encrypted form.

$\boxed{B_1}$ $\boxed{B_2}$ $\boxed{B_3}$ $\boxed{B_4}$ $\boxed{B_5}$ $\boxed{B_6}$

- Every time $\mathcal{O}$ needs to access block B_i, all the blocks are downloaded and all except for B_i are discarded.

Accessing block B_3

$\boxed{B_1}$ $\boxed{B_2}$ $\boxed{B_3}$ $\boxed{B_4}$ $\boxed{B_5}$ $\boxed{B_6}$

$\boxed{B_3}$

Access pattern independent from the block accessed

but... **linear slowdown!!**

First try

Can this be made efficient?

First try

Can this be made efficient?

First try: Initialization

- permute blocks according to permutation π
  - an encryption of B_i is uploaded in position $\pi(i)$;

$$\boxed{B_1} \quad \boxed{B_2} \quad \boxed{B_3} \quad \boxed{B_4} \quad \boxed{B_5} \quad \boxed{B_6}$$

- $\mathcal{O}$ keeps π private;

First try

Can this be made efficient?

First try: Initialization

- permute blocks according to permutation π
  - an encryption of B_i is uploaded in position $\pi(i)$;

$$\boxed{B_2}\ \boxed{B_4}\ \boxed{B_3}\ \boxed{B_6}\ \boxed{B_1}\ \boxed{B_5}$$

- $\mathcal{O}$ keeps π private;

First try

Can this be made efficient?

First try: Reading block i

- ask $\mathcal{M}$ for block in position $\pi(i)$;
- decrypt to obtain B_i;
- re-encrypt and upload in position $\pi(i)$;

Accessing block B_3

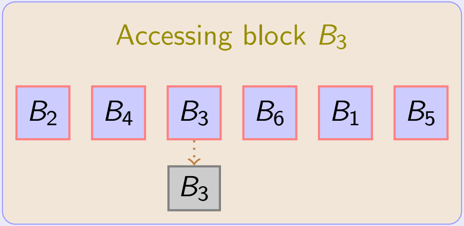
B_2 B_4 B_3 B_6 B_1 B_5

First try

Can this be made efficient?

First try: Reading block i

- ask $\mathcal{M}$ for block in position $\pi(i)$;
- decrypt to obtain B_i;
- re-encrypt and upload in position $\pi(i)$;



First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_1

1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

Access pattern seen by $\mathcal{M}$:

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_1

1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5
				B_1	

Access pattern seen by $\mathcal{M}$: 5

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_2

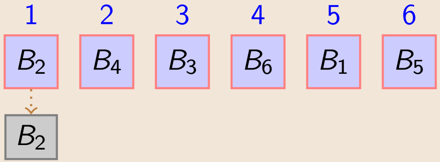
1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

Access pattern seen by $\mathcal{M}$: 5

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_2



Access pattern seen by $\mathcal{M}$: 5, 1

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_3

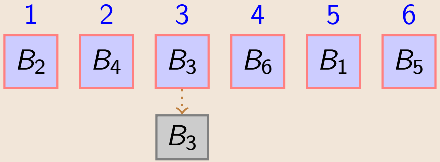
1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

Access pattern seen by $\mathcal{M}$: $5, 1$

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_3



Access pattern seen by $\mathcal{M}$: $5, 1, 3$

First try: Security

Access sequence: B_1, B_2, B_3

1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

Access pattern seen by $\mathcal{M}$: x, y, z

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_1

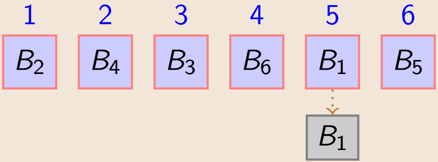
1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

Access pattern seen by $\mathcal{M}$:

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_1



Access pattern seen by $\mathcal{M}$: 5

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_2

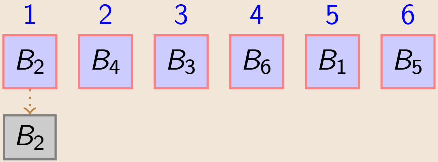
1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

Access pattern seen by $\mathcal{M}$: 5

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_2



Access pattern seen by $\mathcal{M}$: 5, 1

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_1

1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

Access pattern seen by $\mathcal{M}$: $5, 1$

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_1

1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5
				B_1	

Access pattern seen by $\mathcal{M}$: $5, 1, 5$

First try: Security

Access sequence: B_1, B_2, B_1

1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

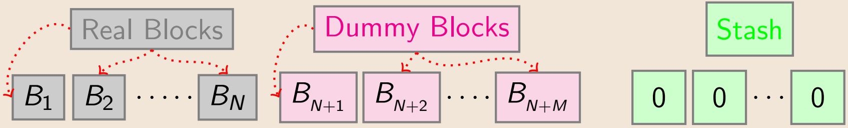
Access pattern seen by $\mathcal{M}$: x, y, x

Hiding the Repetition Pattern

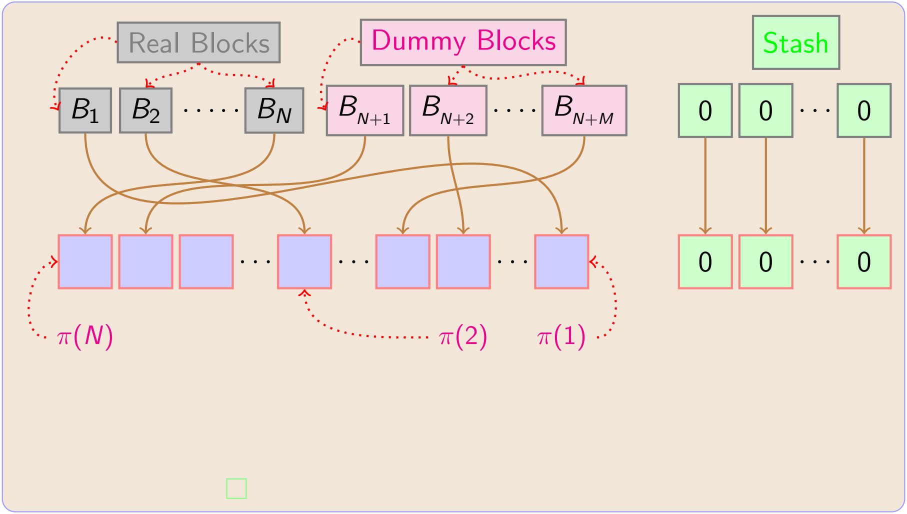
Initialization for N blocks

1. N *real* blocks $B_1, \ldots, B_N$;
2. create M *dummy* blocks $B_{N+1}, \ldots, B_{N+M}$;
3. create M *stash* blocks $S_1, \ldots, S_M$ initialized to 0;
4. pick a random permutation π over $[N + M]$;
5. permute *real* and *dummy* blocks according to permutation π
   - an encryption of B_i is uploaded in position $\pi(i)$;
6. upload all stash blocks in encrypted form;
7. initialize $\mathtt{nxt} = 1, \mathtt{cnt} = 1$;
8. π is kept private;

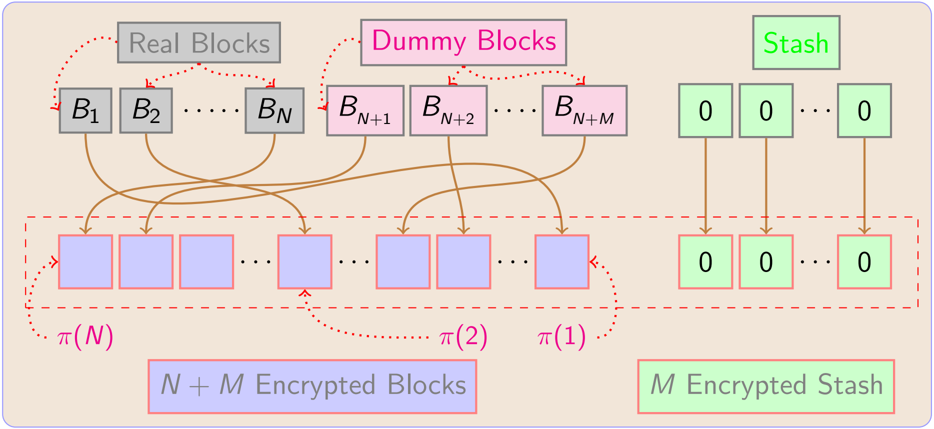
Initial Configuration



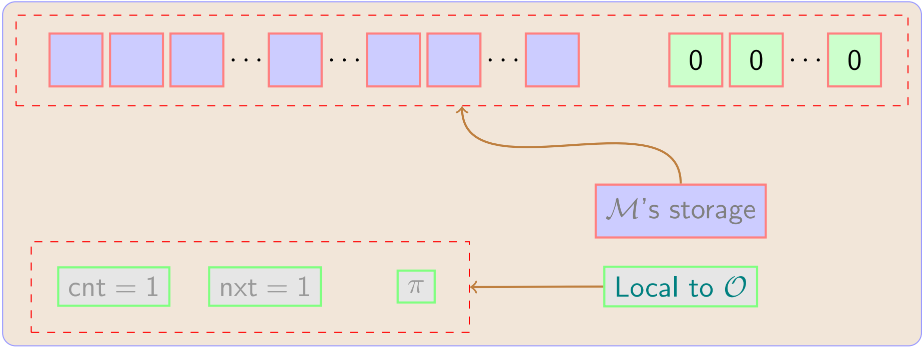
Initial Configuration



Initial Configuration



Initial Configuration



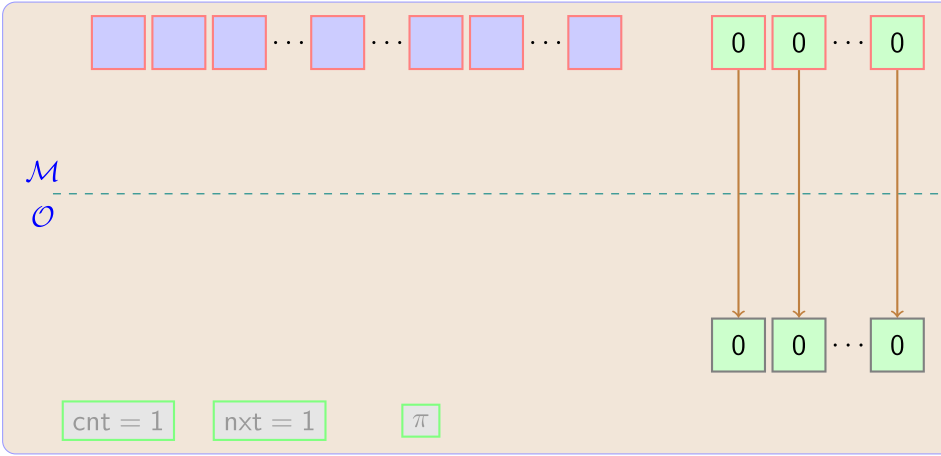
Reading Block B_i

1. download and decrypt all M blocks in the Stash;
2. if B_i is found in the Stash then
   - download dummy block $\pi(N + \texttt{cnt})$;
   - set $\texttt{cnt} = \texttt{cnt} + 1$;

   else
   - download encrypted real block in position $\pi(i)$;
   - decrypt and obtain real block B_i;
   - set next available Stash block $S_{\text{nxt}} = B_i$;
   - set $\text{nxt} = \text{nxt} + 1$;
3. re-encrypt and upload all blocks in the Stash;

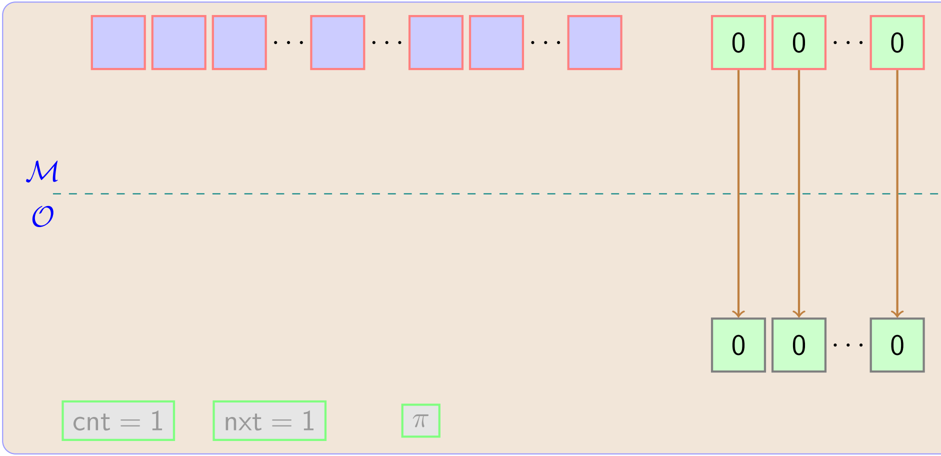
Reading Block B_1

Download and decrypt all blocks from Stash



Reading Block B_1

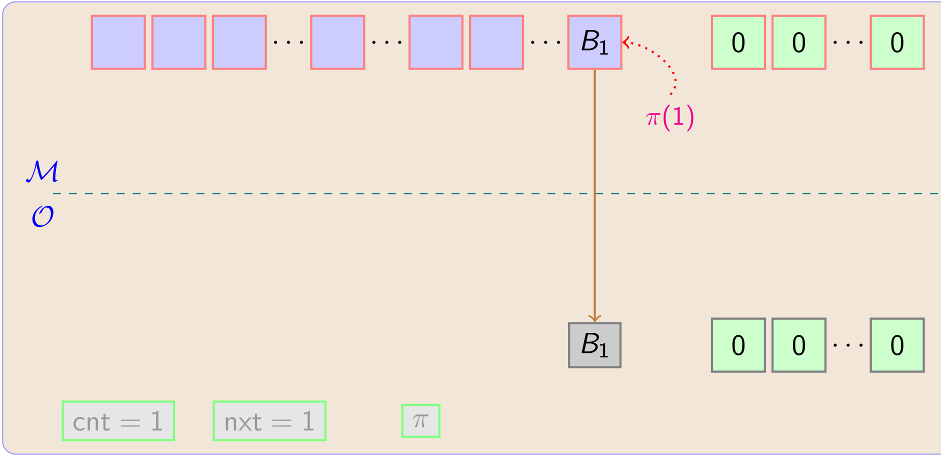
Download and decrypt all blocks from Stash



B_1 is not found in the stash

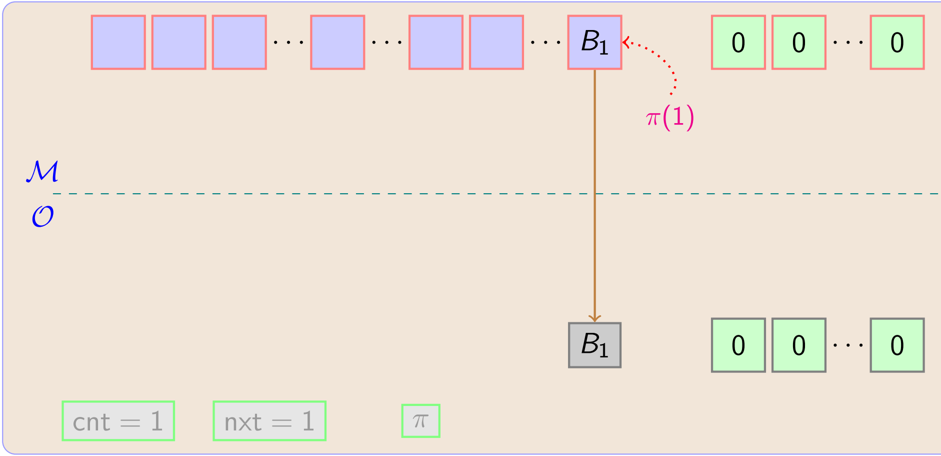
Reading Block B_1

Download block in position $\pi(1)$



Reading Block B_1

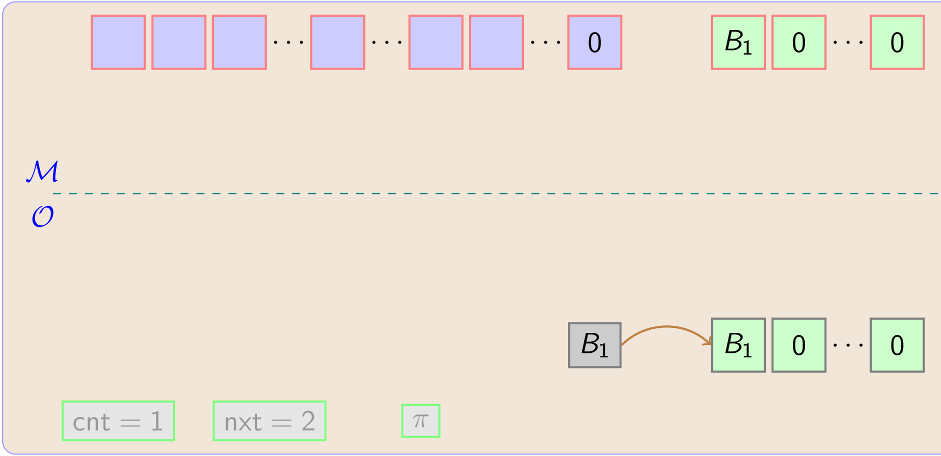
Download block in position $\pi(1)$



Decrypt and obtain B_1

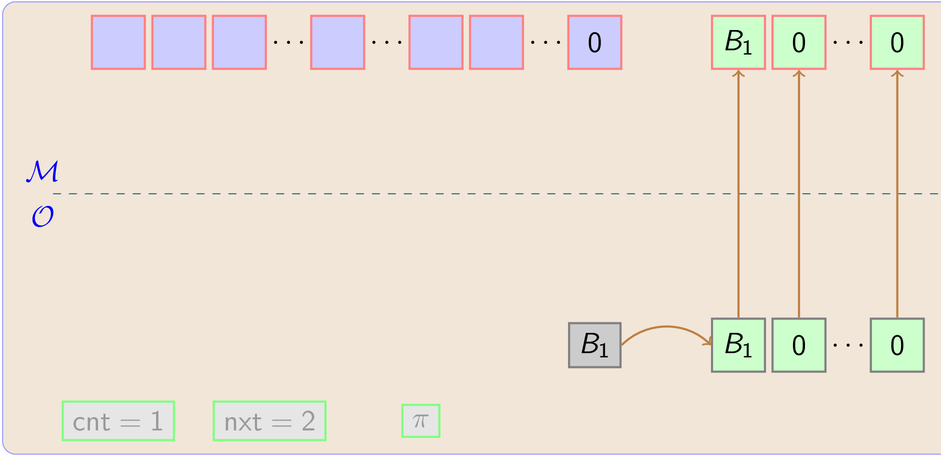
Reading Block B_1

Copy B_1 in the Stash at position nxt



Reading Block B_1

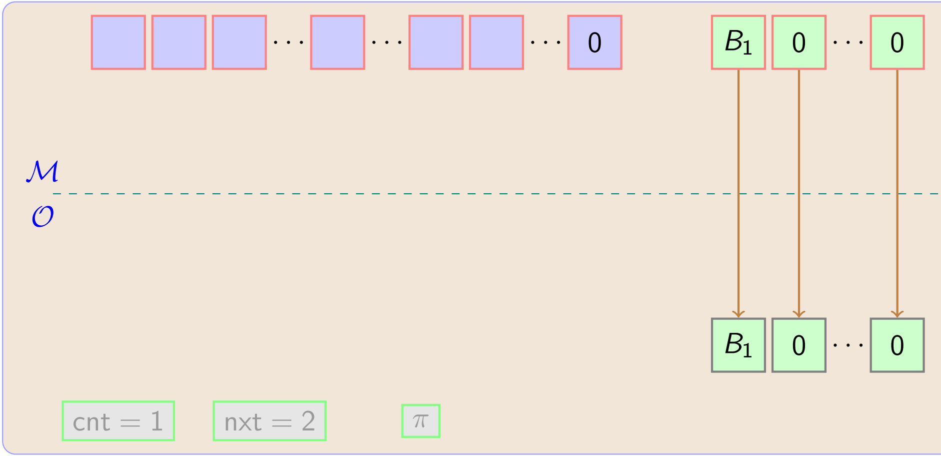
Copy B_1 in the Stash at position nxt



Encrypt and Upload the Stash

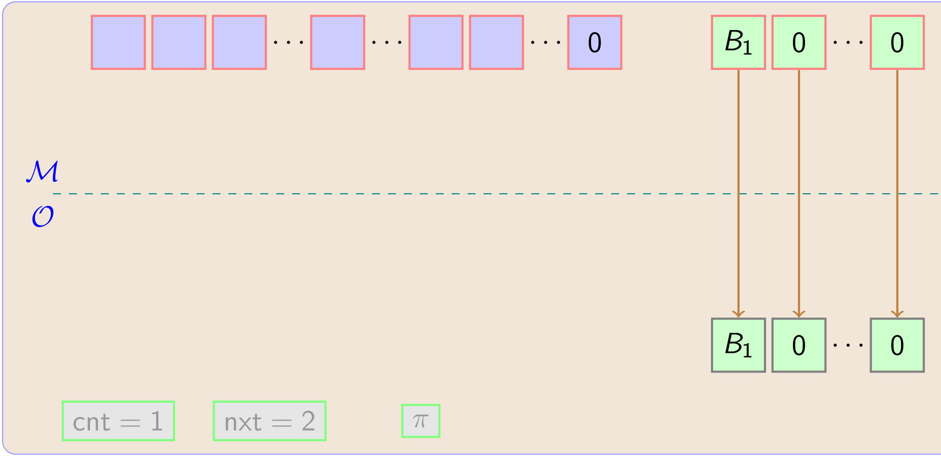
Reading Block B_2

Download and decrypt all blocks from Stash



Reading Block B_2

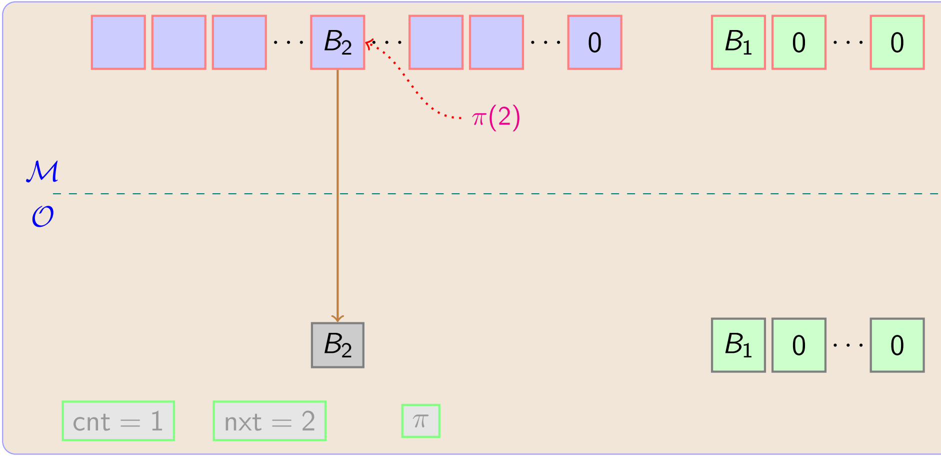
Download and decrypt all blocks from Stash



B_2 is not found in the Stash

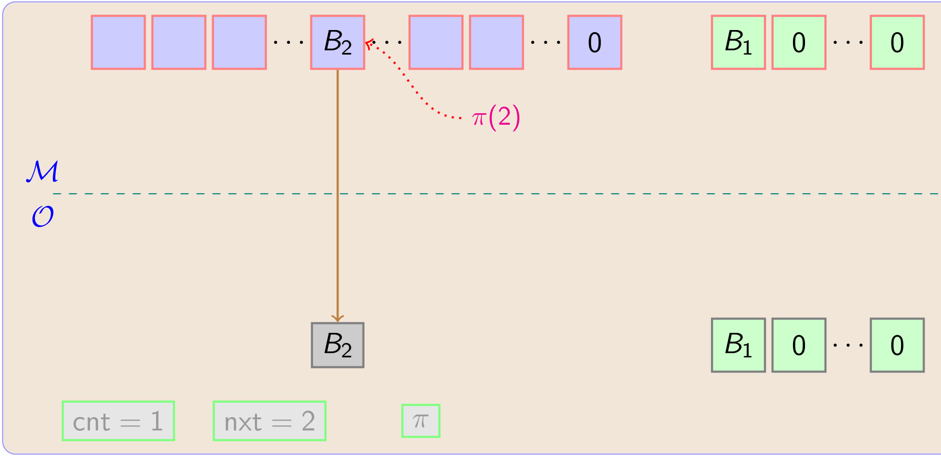
Reading Block B_2

Download block in position $\pi(2)$



Reading Block B_2

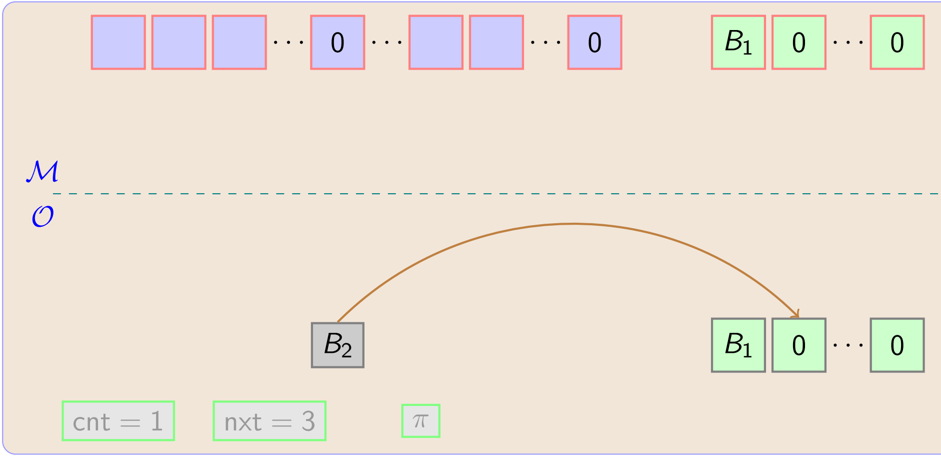
Download block in position $\pi(2)$



Decrypt and obtain B_2

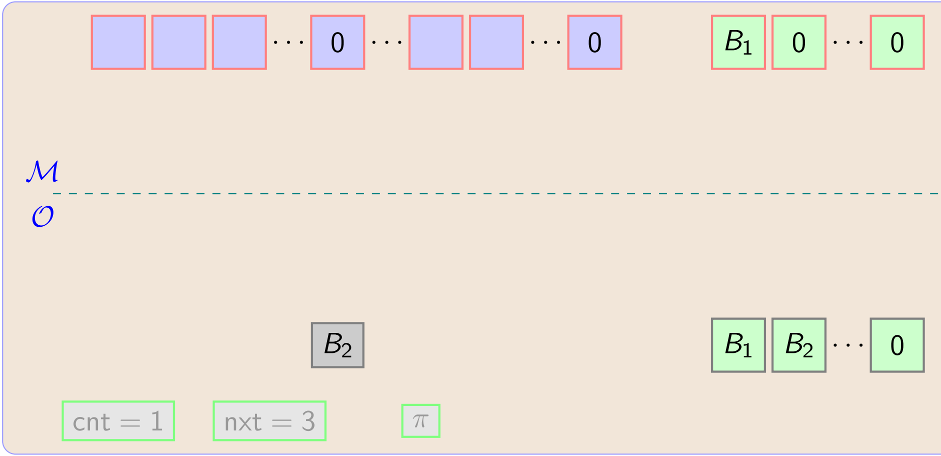
Reading Block B_2

Copy B_2 in the Stash at position nxt



Reading Block B_2

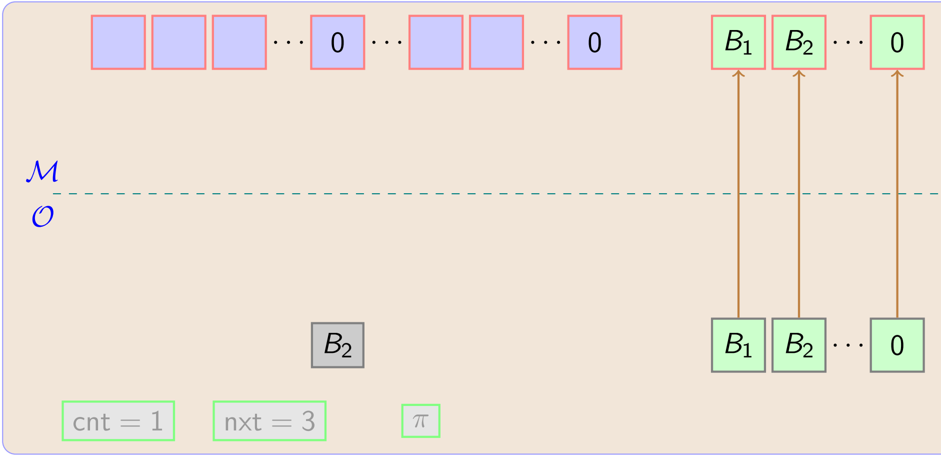
Copy B_2 in the Stash at position nxt



Encrypt and Upload the Stash

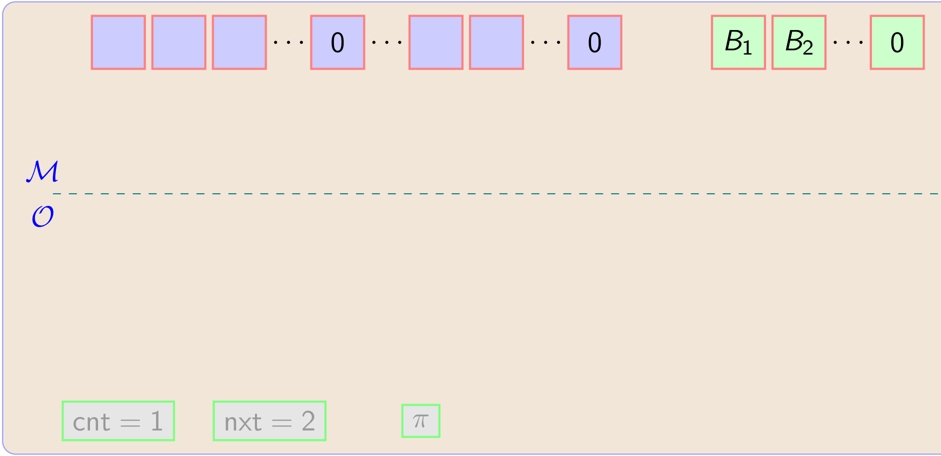
Reading Block B_2

Copy B_2 in the Stash at position nxt



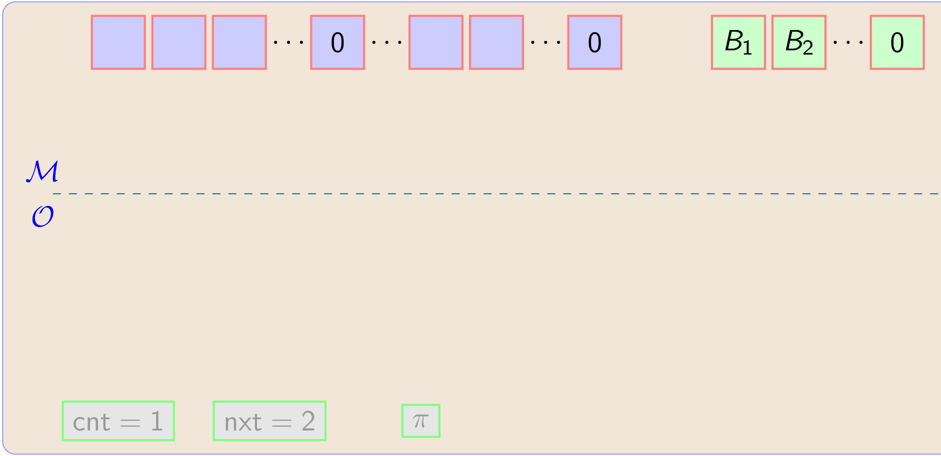
Encrypt and Upload the Stash

Status after reading B_1 and B_2



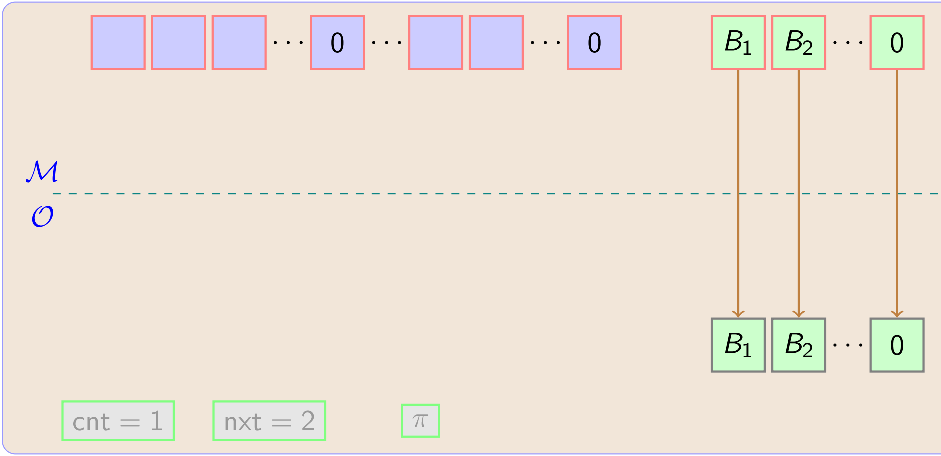
Status after reading B_1 and B_2

Now read B_1 again



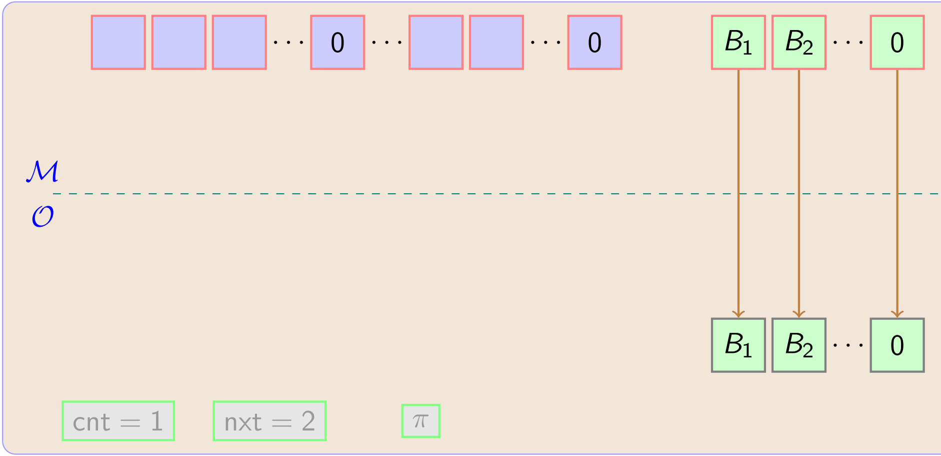
Status after reading B_1 and B_2

Download and decrypt all blocks from Stash



Status after reading B_1 and B_2

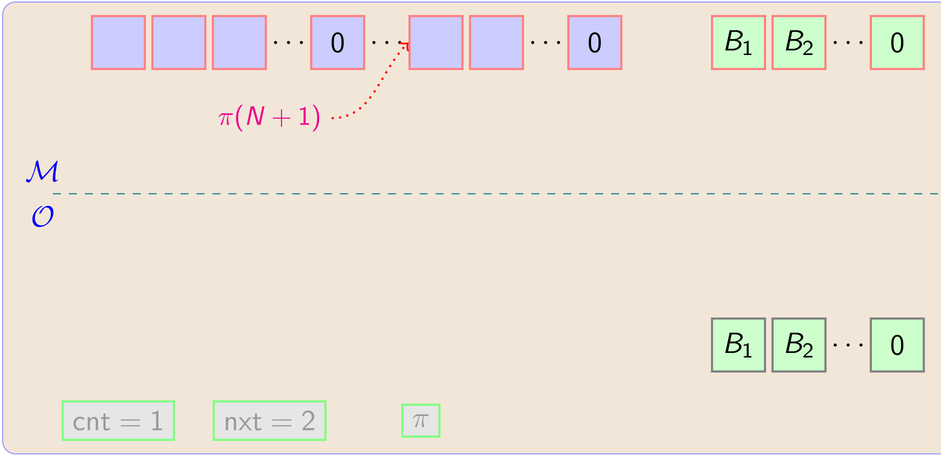
Download and decrypt all blocks from Stash



B_1 is found in the Stash

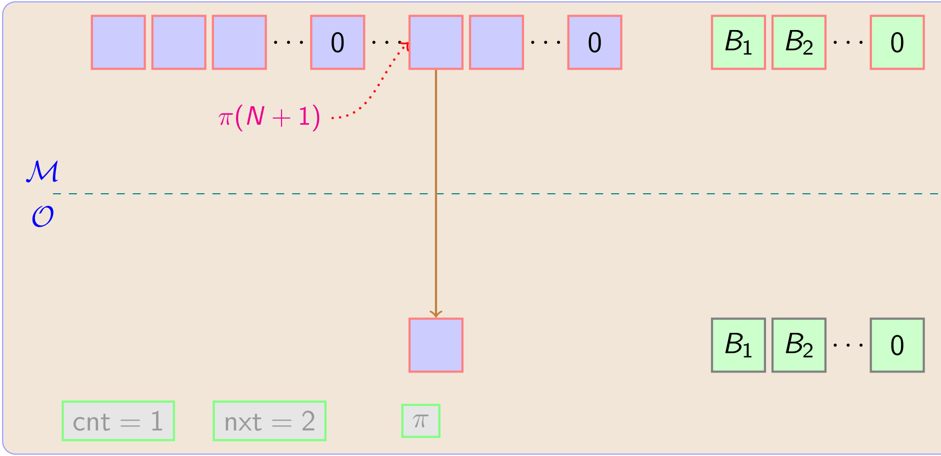
Reading Block B_1 (again)

Download block in position $\pi(N + \mathtt{cnt})$



Reading Block B_1 (again)

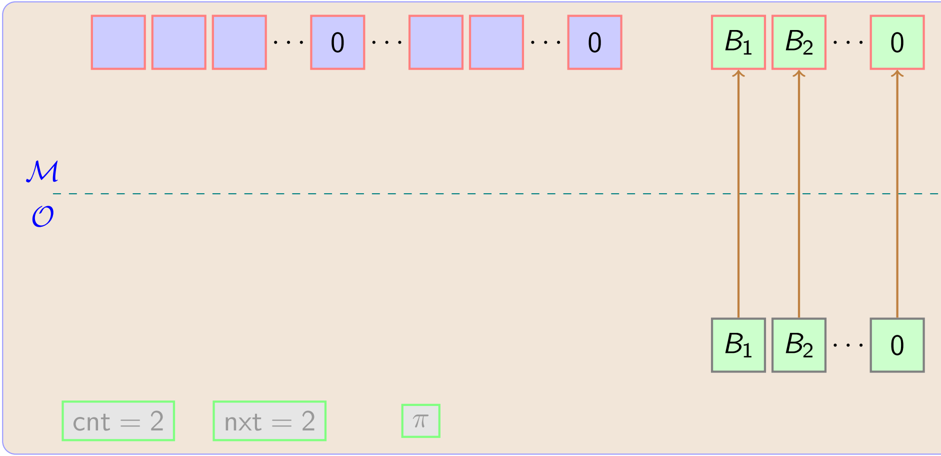
Download block in position $\pi(N + \mathtt{cnt})$



No need to decrypt

Reading Block B_1 (again)

Download block in position $\pi(N + \mathtt{cnt})$



Encrypt and Upload Stash

Why is this oblivious?

Independently from the operation, we have the following

- Download stash
- Download a random location that has not been downloaded yet
- Upload re-encrypted stash

Two issues to be dealt with

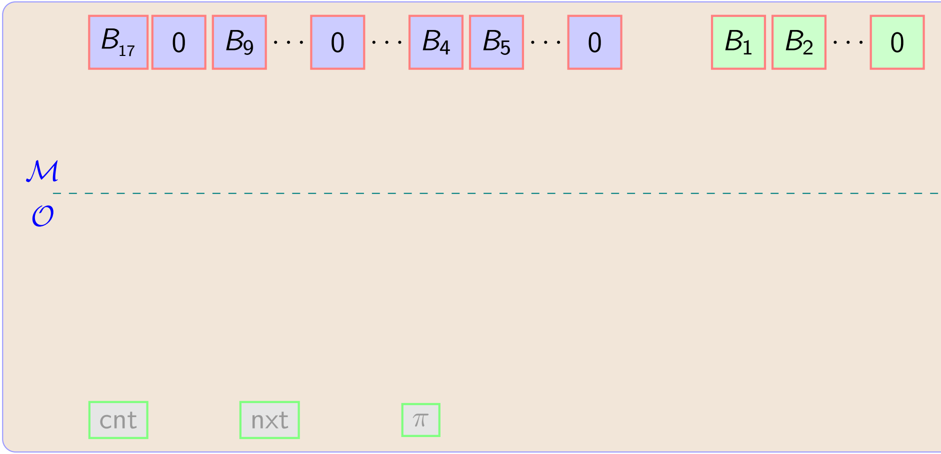
- What happens when the Stash is full?

Two issues to be dealt with

- What happens when the Stash is full?

- How much memory does $\mathcal{O}$ need?
  - needs to store `cnt` and nxt: $\Theta(1)$ memory;
  - π needs $O(N)$ memory.

Overflowing the Stash



Overflowing the Stash

B_{17} | 0 | B_9 $\cdots$ 0 $\cdots$ B_4 | B_5 $\cdots$ 0 B_1 | B_2 $\cdots$ 0

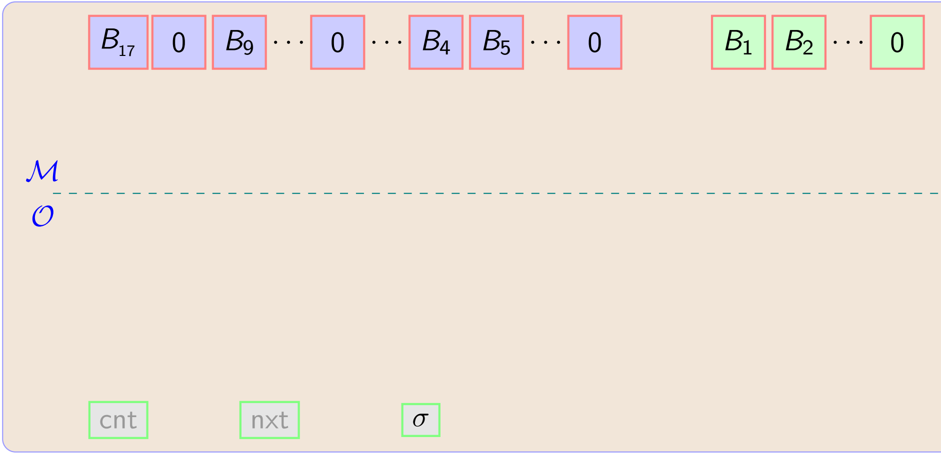
$\mathcal{M}$

$\mathcal{O}$

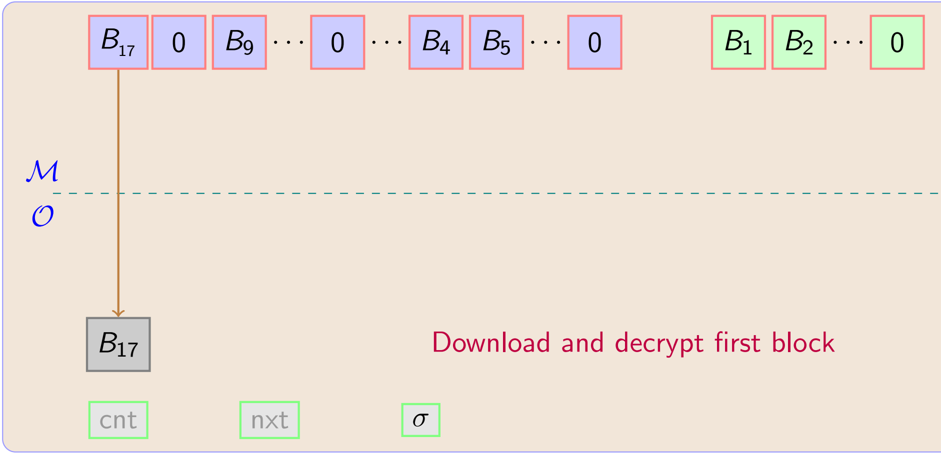
Randomly select a new permutation σ

cnt   nxt π

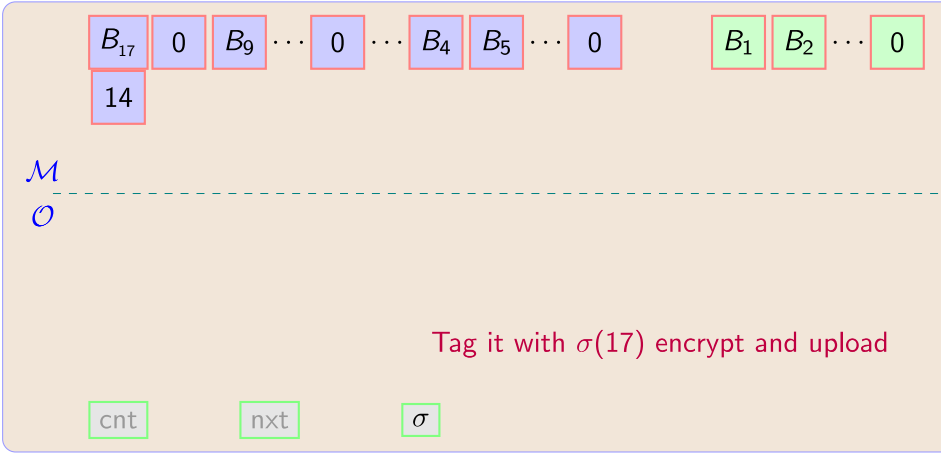
Overflowing the Stash



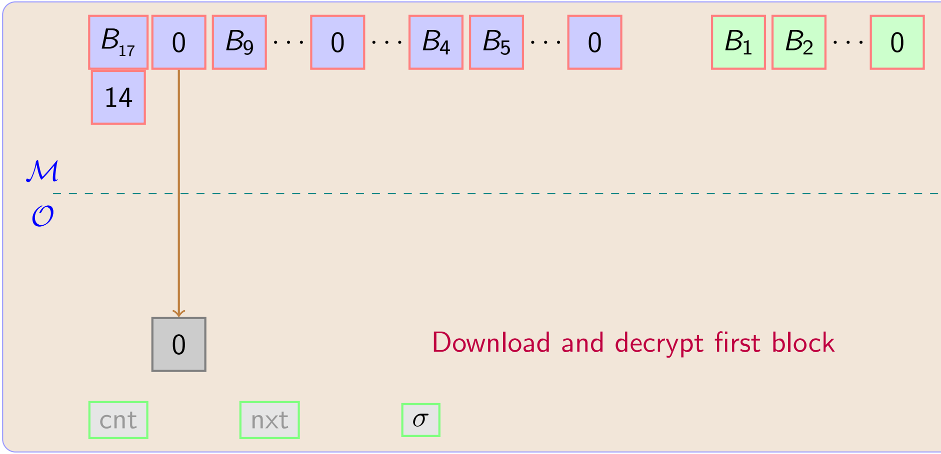
Overflowing the Stash



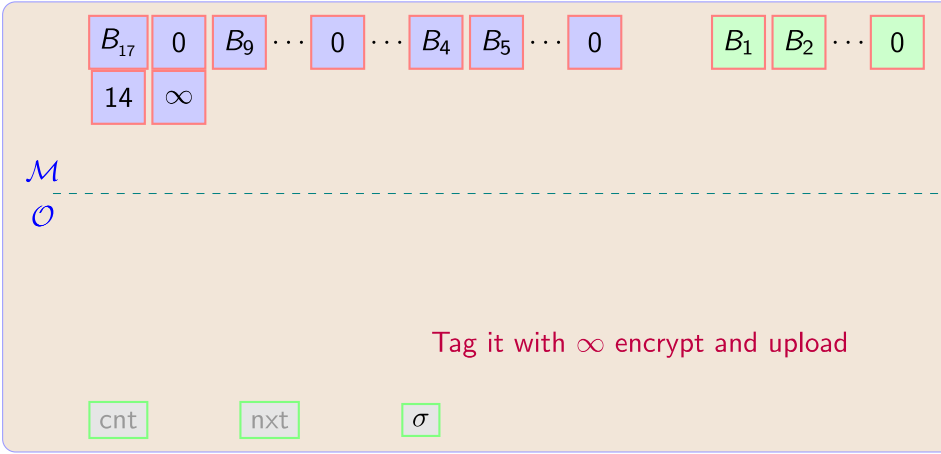
Overflowing the Stash



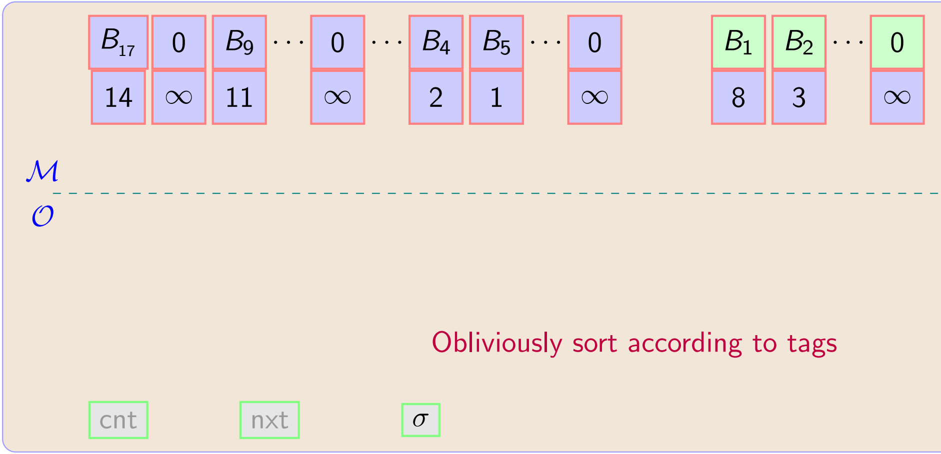
Overflowing the Stash



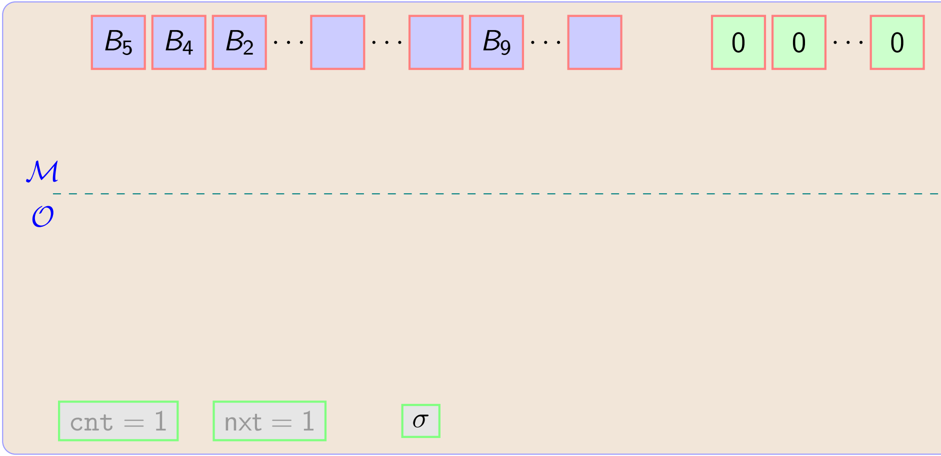
Overflowing the Stash



Overflowing the Stash



Overflowing the Stash



Amortized cost per read operation

Let us count:

Amortized cost per read operation

Let us count:

- each read costs

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle
  - $\Theta(M+N)=\Theta(N)$ blocks of bandwidth for tagging;

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle
  - $\Theta(M+N) = \Theta(N)$ blocks of bandwidth for tagging;
  - $\Theta((M+N)\log(M+N)) = \Theta(N \log N)$ blocks of bandwidth for sorting;

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle
  - $\Theta(M+N)=\Theta(N)$ blocks of bandwidth for tagging;
  - $\Theta((M+N)\log(M+N))=\Theta(N\log N)$ blocks of bandwidth for sorting;

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle
  - $\Theta(M+N)=\Theta(N)$ blocks of bandwidth for tagging;
  - $\Theta((M+N)\log(M+N))=\Theta(N\log N)$ blocks of bandwidth for sorting;

for an amortized cost of

$$\Theta\left(M+\frac{N\log N}{M}\right)$$

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle
  - $\Theta(M+N) = \Theta(N)$ blocks of bandwidth for tagging;
  - $\Theta((M+N)\log(M+N)) = \Theta(N \log N)$ blocks of bandwidth for sorting;

for an amortized cost of

$$\Theta\left(M + \frac{N \log N}{M}\right)$$

for $M = \sqrt{N}$ we have

$$\sqrt{N} \cdot \log N.$$

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle
  - $\Theta(M+N) = \Theta(N)$ blocks of bandwidth for tagging;
  - $\Theta((M+N)\log(M+N)) = \Theta(N \log N)$ blocks of bandwidth for sorting;

for an amortized cost of

$$\Theta\left(M + \frac{N \log N}{M}\right)$$

for $M = \sqrt{N}$ we have

$$\sqrt{N} \cdot \log N.$$

Using AKS to sort.

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle
  - $\Theta(M + N) = \Theta(N)$ blocks of bandwidth for tagging;
  - $\Theta((M + N)\log(M + N)) = \Theta(N \log N)$ blocks of bandwidth for sorting;

for an amortized cost of

$$\Theta\left(M + \frac{N \log N}{M}\right)$$

for $M = \sqrt{N}$ we have

$$\sqrt{N} \cdot \log N.$$

Using AKS to sort. Huge constant

Amortized cost per read operation

Let us count:

- each read costs
  - $\Theta(M)$ blocks of bandwidth for the stash;
  - $\Theta(1)$ blocks of bandwidth for real/dummy blocks;
- after M reads, we shuffle
  - $\Theta(M+N) = \Theta(N)$ blocks of bandwidth for tagging;
  - $\Theta((M+N)\log(M+N)) = \Theta(N\log N)$ blocks of bandwidth for sorting;

for an amortized cost of

$$\Theta\left(M + \frac{N\log N}{M}\right)$$

for $M = \sqrt{N}$ we have

$$\sqrt{N}\cdot\log N.$$

Using AKS to sort.

Huge constant

In practice $\sqrt{N}\cdot\log^2 N$.

In practice...

Jump ahead

One possible setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

In practice...

Jump ahead

One possible setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

In practice...

Jump ahead

One possible setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks.

In practice...

Jump ahead

One possible setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks.
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

using Batcher's sort

In practice...

Jump ahead

One possible setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks.
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

  using Batcher's sort
- Online cost

$$2 \cdot 10^3 \approx 2000$$

In practice...

Jump ahead

One possible setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks.
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

  using Batcher's sort
- Online cost

$$2 \cdot 10^3 \approx 2000$$

- $\mathcal{O}$'s storage

In practice...

Jump ahead

One possible setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks.
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

  using Batcher's sort
- Online cost

$$2 \cdot 10^3 \approx 2000$$

- $\mathcal{O}$'s storage
  - `cnt` and nxt use constant storage

In practice...

Jump ahead

One possible setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks.
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

  using Batcher's sort
- Online cost

$$2 \cdot 10^3 \approx 2000$$

- $\mathcal{O}$'s storage
  - `cnt` and nxt use constant storage
  - π requires storing 10^6 4 bytes integers=4 Megabytes

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

- Online cost: 2 blocks per read

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

- Online cost: 2 blocks per read
- $\mathcal{O}$'s storage

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

- Online cost: 2 blocks per read
- $\mathcal{O}$'s storage
  - `cnt` and `nxt` use constant storage

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

- Online cost: 2 blocks per read
- $\mathcal{O}$'s storage
  - `cnt` and `nxt` use constant storage
  - π requires storing 10^6 4-byte integers=4 Megabytes

Keep the stash in $\mathcal{O}$'s memory

Jump ahead

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash

Resources needed:

- $\mathcal{M}$'s storage: $N + M = 10^6 + 10^3$ blocks
- Cost of shuffling amortized per read operation:

$$1/2 \cdot 6^2 \cdot 10^3 \approx 18000$$

- Online cost: 2 blocks per read
- $\mathcal{O}$'s storage
  - `cnt` and `nxt` use constant storage
  - π requires storing 10^6 4-byte integers=4 Megabytes
  - 1000 blocks of stash for a total of 4 Megabytes

Download and Keep Stash

- Use a cache (the Stash) to hide the repetition pattern

Download and Keep Stash

- Use a cache (the Stash) to hide the repetition pattern
- How does $\mathcal{O}$ hide access to the Stash?

Download and Keep Stash

- Use a cache (the Stash) to hide the repetition pattern
- How does $\mathcal{O}$ hide access to the Stash?
  - Use an ORAM!

Download and Keep Stash

- Use a cache (the Stash) to hide the repetition pattern
- How does $\mathcal{O}$ hide access to the Stash?
  - Use an ORAM!
  - We only have two possible ORAMs:

Download and Keep Stash

- Use a cache (the Stash) to hide the repetition pattern
- How does $\mathcal{O}$ hide access to the Stash?
  - Use an ORAM!
  - We only have two possible ORAMs:
  - Download Stash uses the Download It ORAM to manage the Stash

Download and Keep Stash

- Use a cache (the Stash) to hide the repetition pattern
- How does $\mathcal{O}$ hide access to the Stash?
  - Use an ORAM!
  - We only have two possible ORAMs:
  - Download Stash uses the Download It ORAM to manage the Stash
  - Keep Stash uses the Keep It ORAM to manage the Stash

Download and Keep Stash

- Use a cache (the Stash) to hide the repetition pattern
- How does $\mathcal{O}$ hide access to the Stash?
  - Use an ORAM!
  - We only have two possible ORAMs:
  - Download Stash uses the Download It ORAM to manage the Stash
  - Keep Stash uses the Keep It ORAM to manage the Stash

Download and Keep Stash

- Use a cache (the Stash) to hide the repetition pattern
- How does $\mathcal{O}$ hide access to the Stash?
  - Use an ORAM!
  - We only have two possible ORAMs:
  - Download Stash uses the Download It ORAM to manage the Stash
  - Keep Stash uses the Keep It ORAM to manage the Stash

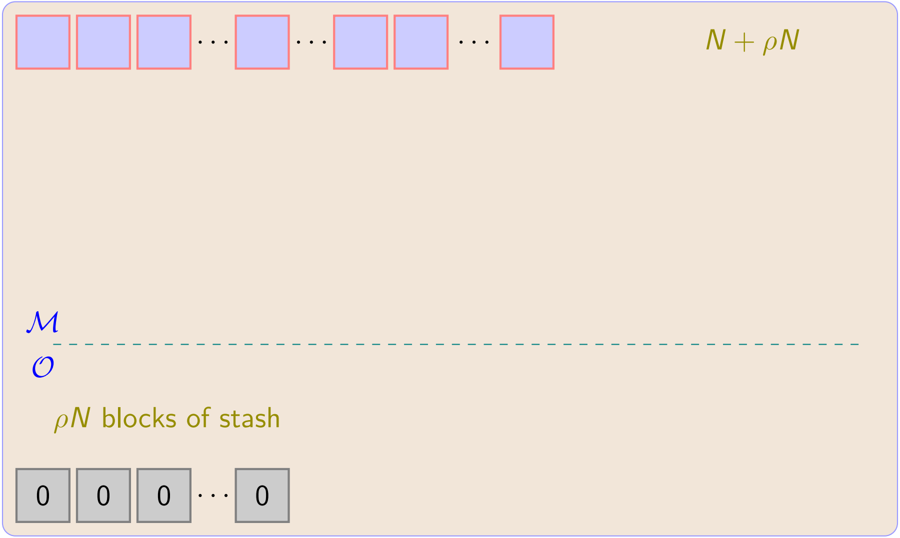
But now we have more ORAMs!!!

$\mathcal{M}$

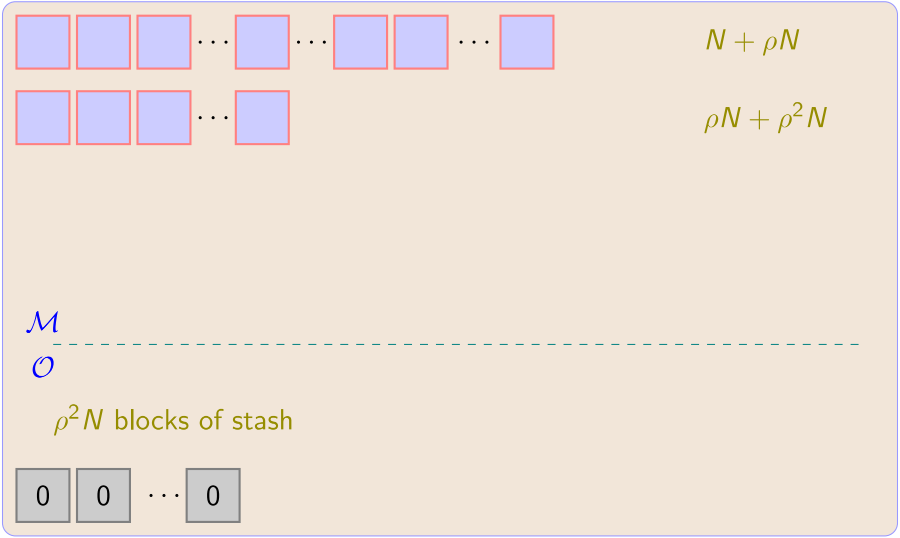
$\mathcal{O}$

N blocks of data

B_1 B_2 B_3 $\cdots$ B_N

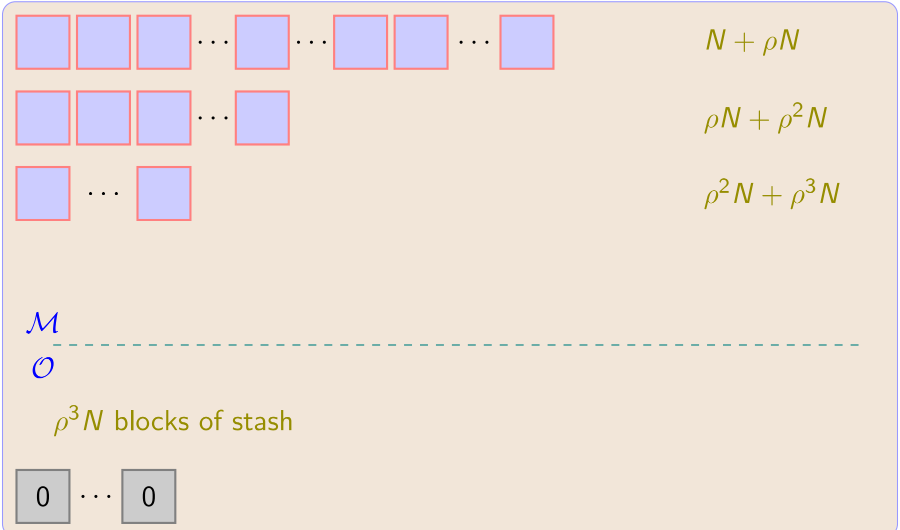


$N + \rho N$
$\mathcal{M}$
$\mathcal{O}$
ρN blocks of stash
0 0 0 ⋯ 0

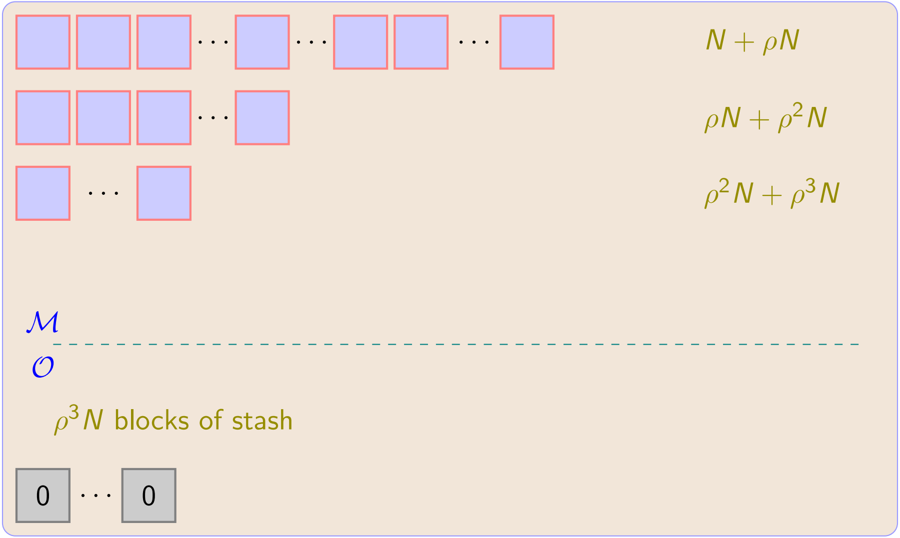


$N + \rho N$
$\rho N + \rho^2 N$
$\mathcal{M}$
$\mathcal{O}$
$\rho^2 N$ blocks of stash
0
0
0

Analysis



$N + \rho N$
$\rho N + \rho^2 N$
$\rho^2 N + \rho^3 N$
$\mathcal{M}$
$\mathcal{O}$
$\rho^3 N$ blocks of stash
0
0



$N + \rho N$
$\rho N + \rho^2 N$
$\rho^2 N + \rho^3 N$
$\mathcal{M}$
$\mathcal{O}$
$\rho^3 N$ blocks of stash
0 ··· 0

Analysis

$N + \rho N$

$\rho N + \rho^2 N$

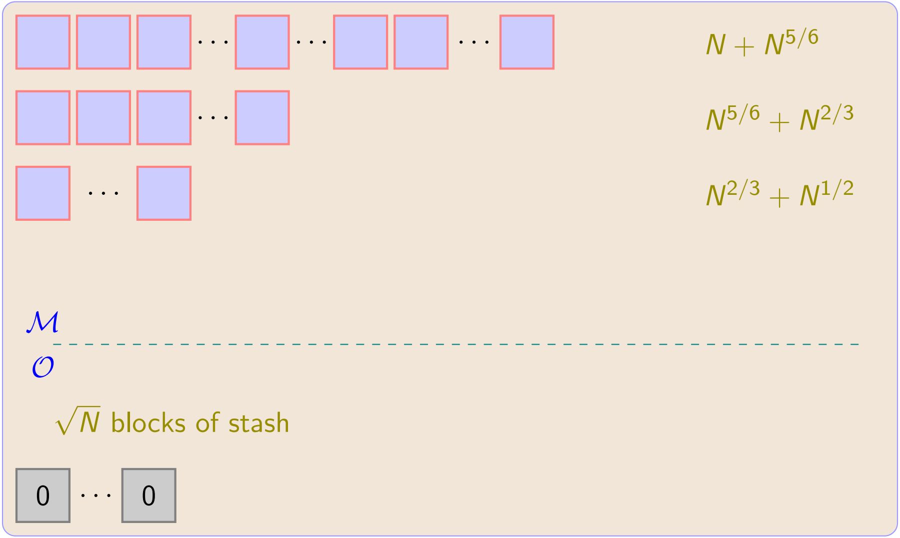
$\rho^2 N + \rho^3 N$

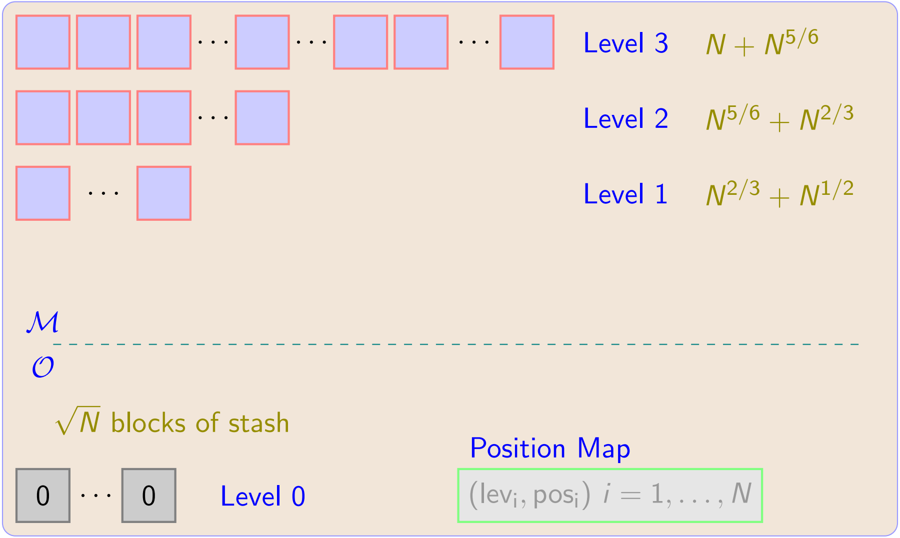
$\mathcal{M}$

$\mathcal{O}$

$\sqrt{N}$ blocks of stash $\rho = N^{-1/6}$

0 $\cdots$ 0


$N + N^{5/6}$
$N^{5/6} + N^{2/3}$
$N^{2/3} + N^{1/2}$
$\mathcal{M}$
$\mathcal{O}$
$\sqrt{N}$ blocks of stash
0
0



Level 3 $N + N^{5/6}$
Level 2 $N^{5/6} + N^{2/3}$
Level 1 $N^{2/3} + N^{1/2}$
$\mathcal{M}$
$\mathcal{O}$
$\sqrt{N}$ blocks of stash
Position Map
0 $\cdots$ 0
Level 0
$(\mathrm{lev}_i, \mathrm{pos}_i)\ i = 1, \ldots, N$

3-level ORAM: Querying

Querying B_q

- retrieve $(\mathsf{lev_q}, \mathsf{pos_q})$ from local memory;

3-level ORAM: Querying

Querying B_q

- retrieve $(\mathsf{lev_q}, \mathsf{pos_q})$ from local memory;
- if $\mathsf{lev_q} \neq 0$

3-level ORAM: Querying

Querying B_q

- retrieve $(\mathsf{lev_q}, \mathsf{pos_q})$ from local memory;
- if $\mathsf{lev_q} \neq 0$
  - for all $l \neq \mathsf{lev_q}$, $\mathcal{O}$ asks for the next dummy in level l;

3-level ORAM: Querying

Querying B_q

- retrieve $(\mathsf{lev_q}, \mathsf{pos_q})$ from local memory;
- if $\mathsf{lev_q} \neq 0$
  - for all $l \neq \mathsf{lev_q}$, $\mathcal{O}$ asks for the next dummy in level l;
  - asks for block in $\mathsf{pos_q}$ from level $\mathsf{lev_q}$;

3-level ORAM: Querying

Querying B_q

- retrieve $(\mathsf{lev_q}, \mathsf{pos_q})$ from local memory;
- if $\mathsf{lev_q} \neq 0$
  - for all $l \neq \mathsf{lev_q}$, $\mathcal{O}$ asks for the next dummy in level l;
  - asks for block in $\mathsf{pos_q}$ from level $\mathsf{lev_q}$;
  - B_q is then stored in level 0 and $(\mathsf{lev_q}, \mathsf{pos_q})$ is updated;

3-level ORAM: Querying

Querying B_q

- retrieve $(\mathsf{lev_q}, \mathsf{pos_q})$ from local memory;
- if $\mathsf{lev_q} \neq 0$
  - for all $l \neq \mathsf{lev_q}$, $\mathcal{O}$ asks for the next dummy in level l;
  - asks for block in $\mathsf{pos_q}$ from level $\mathsf{lev_q}$;
  - B_q is then stored in level 0 and $(\mathsf{lev_q}, \mathsf{pos_q})$ is updated;
- else

3-level ORAM: Querying

Querying B_q

- retrieve $(\mathsf{lev_q}, \mathsf{pos_q})$ from local memory;
- if $\mathsf{lev_q} \neq 0$
  - for all $l \neq \mathsf{lev_q}$, $\mathcal{O}$ asks for the next dummy in level l;
  - asks for block in $\mathsf{pos_q}$ from level $\mathsf{lev_q}$;
  - B_q is then stored in level 0 and $(\mathsf{lev_q}, \mathsf{pos_q})$ is updated;
- else
  - for $l = 1, 2, 3$, $\mathcal{O}$ asks for the next dummy in level l;

3-level ORAM: Querying

Querying B_q

- retrieve $(\mathsf{lev_q}, \mathsf{pos_q})$ from local memory;
- if $\mathsf{lev_q} \neq 0$
  - for all $l \neq \mathsf{lev_q}$, $\mathcal{O}$ asks for the next dummy in level l;
  - asks for block in $\mathsf{pos_q}$ from level $\mathsf{lev_q}$;
  - B_q is then stored in level 0 and $(\mathsf{lev_q}, \mathsf{pos_q})$ is updated;
- else
  - for $l = 1, 2, 3$, $\mathcal{O}$ asks for the next dummy in level l;
  - block B_q is retrieved from local stash (level 0);

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times
  - total cost: $4 \cdot N^{7/6}$

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 2 is full after $N^{5/6}$ queries

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 2 is full after $N^{5/6}$ queries
  - it is shuffled with the level 3 of size N;

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 2 is full after $N^{5/6}$ queries
  - it is shuffled with the level 3 of size N;
  - each shuffle costs $4N$

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 2 is full after $N^{5/6}$ queries
  - it is shuffled with the level 3 of size N;
  - each shuffle costs $4N$
  - over N queries, it happens $N^{1/6}$ times

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 2 is full after $N^{5/6}$ queries
  - it is shuffled with the level 3 of size N;
  - each shuffle costs $4N$
  - over N queries, it happens $N^{1/6}$ times
  - total cost: $4 \cdot N^{7/6}$

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 2 is full after $N^{5/6}$ queries
  - it is shuffled with the level 3 of size N;
  - each shuffle costs $4N$
  - over N queries, it happens $N^{1/6}$ times
  - total cost: $4 \cdot N^{7/6}$
- Over N queries, the cost is $12 \cdot N^{7/6}$

3-level ORAM: Analysis

- at the start only level 3 contains real blocks;
- the local stash is full after $N^{1/2}$ queries
  - it is shuffled with the level 1 of size $N^{2/3}$;
  - each shuffle costs $4N^{2/3}$
  - over N queries, it happens $N^{1/2}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 1 is full after $N^{2/3}$ queries
  - it is shuffled with the level 2 of size $N^{5/6}$;
  - each shuffle costs $4N^{5/6}$
  - over N queries, it happens $N^{1/3}$ times
  - total cost: $4 \cdot N^{7/6}$
- level 2 is full after $N^{5/6}$ queries
  - it is shuffled with the level 3 of size N;
  - each shuffle costs $4N$
  - over N queries, it happens $N^{1/6}$ times
  - total cost: $4 \cdot N^{7/6}$
- Over N queries, the cost is $12 \cdot N^{7/6}$
  - each query has an amortized cost of $12N^{1/6}$ blocks;

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

Resources needed:

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

Resources needed:

- $\mathcal{M}$'s storage: $10^6 + 2 \cdot 10^5 + 2 \cdot 10^4 + 10^3 = 1,221,000$ blocks

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

Resources needed:

- $\mathcal{M}$'s storage: $10^6 + 2 \cdot 10^5 + 2 \cdot 10^4 + 10^3 = 1,221,000$ blocks
- Cost of shuffling amortized per read operation:

$$4 \cdot 10^3 \approx 120$$

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

Resources needed:

- $\mathcal{M}$'s storage: $10^6 + 2 \cdot 10^5 + 2 \cdot 10^4 + 10^3 = 1,221,000$ blocks
- Cost of shuffling amortized per read operation:

$$4 \cdot 10^3 \approx 120$$

- Online cost: 3 blocks downloaded

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

Resources needed:

- $\mathcal{M}$'s storage: $10^6 + 2 \cdot 10^5 + 2 \cdot 10^4 + 10^3 = 1,221,000$ blocks
- Cost of shuffling amortized per read operation:

$$4 \cdot 10^3 \approx 120$$

- Online cost: 3 blocks downloaded
- $\mathcal{O}$'s storage

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

Resources needed:

- $\mathcal{M}$'s storage: $10^6 + 2 \cdot 10^5 + 2 \cdot 10^4 + 10^3 = 1,221,000$ blocks
- Cost of shuffling amortized per read operation:

$$4 \cdot 10^3 \approx 120$$

- Online cost: 3 blocks downloaded
- $\mathcal{O}$'s storage
  - `cnt` and `nxt` use constant storage

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

Resources needed:

- $\mathcal{M}$'s storage: $10^6 + 2 \cdot 10^5 + 2 \cdot 10^4 + 10^3 = 1,221,000$ blocks
- Cost of shuffling amortized per read operation:

$$4 \cdot 10^3 \approx 120$$

- Online cost: 3 blocks downloaded
- $\mathcal{O}$'s storage
  - `cnt` and nxt use constant storage
  - position map requires storing 10^6 6-byte integers $\approx$ 4 Megabytes

3-level ORAM: in practice

Same setting:

- $N = 10^6$ blocks of 4K each for a total of 4 Gigabytes
- $M = 10^3$ blocks of stash for a total of 4 Megabytes

Resources needed:

- $\mathcal{M}$'s storage: $10^6 + 2 \cdot 10^5 + 2 \cdot 10^4 + 10^3 = 1,221,000$ blocks
- Cost of shuffling amortized per read operation:

$$4 \cdot 10^3 \approx 120$$

- Online cost: 3 blocks downloaded
- $\mathcal{O}$'s storage
  - `cnt` and `nxt` use constant storage
  - position map requires storing 10^6 6-byte integers $\approx$ 4 Megabytes
  - 1000 blocks of stash for a total of 4 Megabytes

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$
- $1/2 \log_2 N$ levels with

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$
- $1/2 \log_2 N$ levels with
  - $N + N/2$ blocks

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$
- $1/2 \log_2 N$ levels with
  - $N + N/2$ blocks
  - $N/2 + N/4$ blocks

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$
- $1/2 \log_2 N$ levels with
  - $N + N/2$ blocks
  - $N/2 + N/4$ blocks
  - $\ldots$

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$
- $1/2 \log_2 N$ levels with
  - $N + N/2$ blocks
  - $N/2 + N/4$ blocks
  - $\ldots$
  - $3\sqrt{N}$ blocks

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$
- $1/2 \log_2 N$ levels with
  - $N + N/2$ blocks
  - $N/2 + N/4$ blocks
  - $\ldots$
  - $3\sqrt{N}$ blocks
- Amortized bandwidth $0.55 \cdot \log_2 N$

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$
- $1/2 \log_2 N$ levels with
  - $N + N/2$ blocks
  - $N/2 + N/4$ blocks
  - $\ldots$
  - $3\sqrt{N}$ blocks
- Amortized bandwidth $0.55 \cdot \log_2 N$
  - For $N = 10^6$, 21 Blocks

Why stop at 3 levels?

$\mathcal{O}$'s memory $\approx \sqrt{N}$

- set $\rho = 1/2$
- $1/2 \log_2 N$ levels with
  - $N + N/2$ blocks
  - $N/2 + N/4$ blocks
  - $\ldots$
  - $3\sqrt{N}$ blocks
- Amortized bandwidth $0.55 \cdot \log_2 N$
  - For $N = 10^6$, 21 Blocks
- OnLine bandwidth: 1 Block

Asymptotics

Hierarchical Approach with **constant** client memory

- $O(\log^3 N)$ – Goldreich-Ostrovsky 1987-1990
- $O((\log^2 N)/\log\log N)$ – Kushilevitz-Lu-Ostrovsky 2012
- $O(\log N \cdot \log\log N)$ – Patel-P-Raykova-Yeo 2018
- $O(\log N)$ – Asharov-Komargodski-Lin-Nayak-Peserico-Shi 2020

Asymptotics

Hierarchical Approach with **constant** client memory

- $O(\log^3 N)$ – Goldreich-Ostrovsky 1987-1990
- $O((\log^2 N)/\log\log N)$ – Kushilevitz-Lu-Ostrovsky 2012
- $O(\log N \cdot \log\log N)$ – Patel-P-Raykova-Yeo 2018
- $O(\log N)$ – Asharov-Komargodski-Lin-Nayak-Peserico-Shi 2020

$\Omega(\log(N/C))$ – Larsen-Nielsen 2018

Differential Privacy

ORAM

(ϵ, δ)-Differential Privacy

- $\mathcal{M}$ stores n blocks of memory.
- Every time $\mathcal{O}$ wants a block, he asks $\mathcal{M}$ one or more blocks.
- Security notion:
  - For any two block sequences $\mathbb{B} = B_1, \ldots, B_n$ and $\mathbb{C} = C_1, \ldots, C_n$
  - For any two access sequences $i_1, \ldots, i_l$ and $j_1, \ldots, j_l$ **that differ in one position**
    - performing access $i_1, \ldots, i_l$ on $\mathbb{B} = B_1, \ldots, B_n$;
    - performing access $j_1, \ldots, j_l$ on $\mathbb{C} = C_1, \ldots, C_n$

generate the same distribution of accesses to the data stored by $\mathcal{M}$

Differential Privacy

▸ ORAM

(ϵ, δ)-Differential Privacy

- $\mathcal{M}$ stores n blocks of memory.
- Every time $\mathcal{O}$ wants a block, he asks $\mathcal{M}$ one or more blocks.
- Security notion:
  - For any two block sequences $\mathbb{B} = B_1, \ldots, B_n$ and $\mathbb{C} = C_1, \ldots, C_n$
  - For any two access sequences $i_1, \ldots, i_l$ and $j_1, \ldots, j_l$ **that differ in one position**
    - performing access $i_1, \ldots, i_l$ on $\mathbb{B} = B_1, \ldots, B_n$;
    - performing access $j_1, \ldots, j_l$ on $\mathbb{C} = C_1, \ldots, C_n$

  generate the same distribution of accesses to the data stored by $\mathcal{M}$

For every predicate A

$$\begin{aligned}\mathrm{Prob}[\mathtt{view} \leftarrow \mathtt{View}(I, \mathbb{B}) : A(\mathtt{view}) = 1] \\ \leq e^{\epsilon} \cdot \mathrm{Prob}[\mathtt{view} \leftarrow \mathtt{View}(J, \mathbb{C}) : A(\mathtt{view}) = 1] + \delta\end{aligned}$$

Differential Privacy

ORAM

(ϵ, δ)-Differential Privacy

- $\mathcal{M}$ stores n blocks of memory.
- Every time $\mathcal{O}$ wants a block, he asks $\mathcal{M}$ one or more blocks.
- Security notion:
  - For any two block sequences $\mathbb{B} = B_1, \ldots, B_n$ and $\mathbb{C} = C_1, \ldots, C_n$
  - For any two access sequences $i_1, \ldots, i_l$ and $j_1, \ldots, j_l$ **that differ in one position**
    - performing access $i_1, \ldots, i_l$ on $\mathbb{B} = B_1, \ldots, B_n$;
    - performing access $j_1, \ldots, j_l$ on $\mathbb{C} = C_1, \ldots, C_n$

  generate the same distribution of accesses to the data stored by $\mathcal{M}$

For every predicate A

$$\begin{aligned} &\mathrm{Prob}[\mathtt{view} \leftarrow \mathtt{View}(I, \mathbb{B}) : A(\mathtt{view}) = 1] \\ &\quad \leq e^{\epsilon} \cdot \mathrm{Prob}[\mathtt{view} \leftarrow \mathtt{View}(J, \mathbb{C}) : A(\mathtt{view}) = 1] + \delta \end{aligned}$$

$\Omega(\log(N/C))$ – P-Yeo 2019

The snapshot adversary

the Server is the adversary

Snapshot Adversary

Du, Genkin, Grubbs, 2022

- The adversary gets control of the Server for L consecutive operations
  - Slowdown $O(\log L)$

The snapshot adversary

the Server is the adversary

Snapshot Adversary

Du, Genkin, Grubbs, 2022

- The adversary gets control of the Server for L consecutive operations
  - Slowdown $O(\log L)$

What if the adversary is active for more than one *window*?

The snapshot adversary

Snapshot window (t, ℓ)

- A *snapshot window* of length ℓ starting at time t.
- The adversary receives
  - snapshot of server memory content before operation t has been executed
  - transcript of server's operations for the following ℓ operations that take place at times $t, t+1, \ldots, t+\ell-1$.
- For $\ell = 0$, only memory content before operation t.

A (S, L)-snapshot adversary

Specifies a sequence of *snaspshot windows* $\mathcal{S} = ((t_1, \ell_1), \ldots, (t_s, \ell_s))$ such that

- $s \leq S$, at most S windows,
  - at most S snapshots
- $\sum \ell_i \leq L$, for a total duration of at most L operations
  - at most L transcripts

The Lower Bound

Theorem (P-Yeo 23)

For any $0 \leq \epsilon \leq 1/16$, *let* DS *be a* $(3, 1, \epsilon)$*-snapshot private RAM data structure for* n *entries each of* b *bits implemented over* $w = \Omega(\log n)$ *bits using client storage of* c *bits in the cell probe model. If* DS *has amortized write time* t_w *and expected amortized read time* t_r *with failure probability at most* $1/3$, *then*

$$t_r + t_w = \Omega\left(b/w \cdot \log(nb/c)\right).$$

The Lower Bound

Theorem (P-Yeo 23)

For any $0 \leq \epsilon \leq 1/16$, *let* DS *be a* $(3, 1, \epsilon)$*-snapshot private RAM data structure for* n *entries each of* b *bits implemented over* $w = \Omega(\log n)$ *bits using client storage of* c *bits in the cell probe model. If* DS *has amortized write time* t_w *and expected amortized read time* t_r *with failure probability at most* $1/3$, *then*

$$t_r + t_w = \Omega\left(b/w \cdot \log(nb/c)\right).$$

n *logical* blocks of b bits

The Lower Bound

Theorem (P-Yeo 23)

For any $0 \le \epsilon \le 1/16$*, let* DS *be a* $(3, 1, \epsilon)$*-snapshot private RAM data structure for* n *entries each of* b *bits implemented over* $w = \Omega(\log n)$ *bits using client storage of* c *bits in the cell probe model. If* DS *has amortized write time* t_w *and expected amortized read time* t_r *with failure probability at most* $1/3$*, then*

$$t_r + t_w = \Omega\left(b/w \cdot \log(nb/c)\right).$$

$w < b$ is size *physical* words

The Lower Bound

Theorem (P-Yeo 23)

For any $0 \leq \epsilon \leq 1/16$, *let* DS *be a* $(3, 1, \epsilon)$*-snapshot private RAM data structure for* n *entries each of* b *bits implemented over* $w = \Omega(\log n)$ *bits using client storage of* c *bits in the cell probe model. If* DS *has amortized write time* t_w *and expected amortized read time* t_r *with failure probability at most* $1/3$, *then*

$$t_r + t_w = \Omega\left(b/w \cdot \log(nb/c)\right).$$

Client has c bits of *local memory*

The Lower Bound

Theorem (P-Yeo 23)

For any $0 \leq \epsilon \leq 1/16$, *let* DS *be a* $(3, 1, \epsilon)$*-snapshot private RAM data structure for* n *entries each of* b *bits implemented over* $w = \Omega(\log n)$ *bits using client storage of* c *bits in the cell probe model. If* DS *has amortized write time* t_w *and expected amortized read time* t_r *with failure probability at most* $1/3$, *then*

$$t_r + t_w = \Omega\left(b/w \cdot \log(nb/c)\right).$$

Adversary receives at most 3 memory *snapshots* and 1 operation *transcript*

The Lower Bound

Theorem (P-Yeo 23)

For any $0 \leq \epsilon \leq 1/16$, *let* DS *be a* $(3, 1, \epsilon)$*-snapshot private RAM data structure for* n *entries each of* b *bits implemented over* $w = \Omega(\log n)$ *bits using client storage of* c *bits in the cell probe model. If* DS *has amortized write time* t_w *and expected amortized read time* t_r *with failure probability at most* $1/3$, *then*

$$t_r + t_w = \Omega\left(b/w \cdot \log(nb/c)\right).$$

ϵ is the adversary's advantage in the security game

The security game

$\mathrm{Expt}^{n,\beta}_{\mathrm{DS},\mathcal{A}}$

- Receive $(O_0, O_1, ((t_1, \ell_1), \ldots, (t_s, \ell_s)))$ from $\mathcal{A}_0(1^n)$.
- Set $\mathcal{L} \leftarrow \emptyset$, $\mathrm{DS} \leftarrow (R_1, \ldots, R_n)$, $i \leftarrow 1$.
- While $i \leq |O_\beta|$:
  - If $i = t_j$ for some $1 \leq j \leq s$:
    - Set $\mathcal{L} \leftarrow \mathcal{L} \,||\, (\mathrm{memory}, M)$.
    - For $k = 1, \ldots, \ell_j$:
      Execute $\mathrm{DS}^{\mathrm{LRead},\mathrm{LWrite}}(O_b[i])$ and set $i \leftarrow i + 1$.
  - Else:
    Execute $\mathrm{DS}^{\mathrm{Read},\mathrm{Write}}(O_b[i])$ and set $i \leftarrow i + 1$.
- Return $\mathcal{A}_1(\mathcal{L})$.

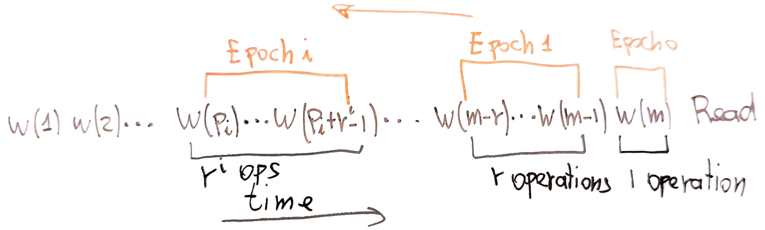
$$\left|\Pr[\mathrm{Expt}^{n,0}_{\mathrm{DS},\mathcal{A}} = 1] - \Pr[\mathrm{Expt}^{n,1}_{\mathrm{DS},\mathcal{A}} = 1]\right| \leq \epsilon,$$

for all PPT $\mathcal{A}$ that are (S, L)-snapshot adversaries.

The Epoch structure

The sequence and the epochs

- n *logical indices*
- $m \leftarrow \{n/2+1, \ldots, n\}$
- m writes of **random** b-bit blocks at indices $1, 2, \ldots, m$
- followed by one read.



A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1

A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices

A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1

A $(3,1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch

A $(3,1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch
- **correctness of O_1**
  touch about b/w cells updated in epoch i

A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch
- **correctness of** O_1
  touch about b/w cells updated in epoch i
  - epochs preceding epoch i are **independent**

A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch
- **correctness of** O_1
  touch about b/w cells updated in epoch i
  - epochs preceding epoch i are **independent**
  - epochs following epoch i are **not large enough**

A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch
- **correctness of** O_1
  touch about b/w cells updated in epoch i
  - epochs preceding epoch i are **independent**
  - epochs following epoch i are **not large enough**
  - pick i so that client memory is **too small**

A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch
- **correctness of O_1**
  touch about b/w cells updated in epoch i
  - epochs preceding epoch i are **independent**
  - epochs following epoch i are **not large enough**
  - pick i so that client memory is **too small**
- **correctness of O_0**
  read of O_0 does not depend on epoch i

A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch
- **correctness of** O_1
  touch about b/w cells updated in epoch i
  - epochs preceding epoch i are **independent**
  - epochs following epoch i are **not large enough**
  - pick i so that client memory is **too small**
- **correctness of** O_0
  read of O_0 does not depend on epoch i
- **security**
  but if it does not, then security fails

A $(3,1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch
- **correctness of** O_1
  touch about b/w cells updated in epoch i
  - epochs preceding epoch i are **independent**
  - epochs following epoch i are **not large enough**
  - pick i so that client memory is **too small**
- **correctness of** O_0
  read of O_0 does not depend on epoch i
- **security**
  but if it does not, then security fails
- **final step**
  this holds for *all* epochs except for those that have fewer than c/b writes.

A $(3, 1)$-snapshot adversary – Intuition

Jump

- Two sequences of operations O_0, O_1
  - Both write **random** blocks to the first m indices
  - O_0 reads index 1
  - O_1 reads a randomly selected index j written in the i-th epoch
- **correctness of** O_1
  touch about b/w cells updated in epoch i
  - epochs preceding epoch i are **independent**
  - epochs following epoch i are **not large enough**
  - pick i so that client memory is **too small**
- **correctness of** O_0
  read of O_0 does not depend on epoch i
- **security**
  but if it does not, then security fails
- **final step**
  this holds for *all* epochs except for those that have fewer than c/b writes.
- **we have a lower bound** $\Omega(b/w \cdot \log(nb/c))$

A $(3, 1)$-snapshot adversary – Part 0

$\mathcal{A}_0^i(1^n)$

- Randomly select integer m from $[n/2, n]$.
- Randomly and ind. select $\mathsf{B}_1, \ldots, \mathsf{B}_m \leftarrow \{0, 1\}^b$.
- Set $O_0 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(m))$.
- Randomly select $j \in [p_i, p_i + r^i - 1]$,
- Set $O_1 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(j))$.
- Set $\mathcal{S} = ((p_i, 0), (p_i + r^i, 0), (m + 1, 1))$.
- Return (O_0, O_1, S).

A $(3,1)$-snapshot adversary – Part 0

$\mathcal{A}_0^i(1^n)$

- Randomly select integer m from $[n/2, n]$.
- Randomly and ind. select $\mathsf{B}_1, \ldots, \mathsf{B}_m \leftarrow \{0,1\}^b$.
- Set $O_0 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(m))$.
- Randomly select $j \in [p_i, p_i + r^i - 1]$,
- Set $O_1 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(j))$.
- Set $\mathcal{S} = ((p_i, 0), (p_i + r^i, 0), (m+1, 1))$.
- Return (O_0, O_1, S).

- $(p_i, 0)$: snapshot of server memory before epoch i

A $(3,1)$-snapshot adversary – Part 0

$\mathcal{A}_0^i(1^n)$

- Randomly select integer m from $[n/2, n]$.
- Randomly and ind. select $\mathsf{B}_1, \ldots, \mathsf{B}_m \leftarrow \{0,1\}^b$.
- Set $O_0 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(m))$.
- Randomly select $j \in [p_i, p_i + r^i - 1]$,
- Set $O_1 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(j))$.
- Set $\mathcal{S} = ((p_i, 0), (p_i + r^i, 0), (m+1, 1))$.
- Return (O_0, O_1, S).

- $(p_i, 0)$: snapshot of server memory before epoch i
- $(p_i + r^i, 0)$: snapshot of server memory after epoch i

A $(3,1)$-snapshot adversary – Part 0

$\mathcal{A}_0^i(1^n)$

- Randomly select integer m from $[n/2, n]$.
- Randomly and ind. select $\mathsf{B}_1, \ldots, \mathsf{B}_m \leftarrow \{0,1\}^b$.
- Set $O_0 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(m))$.
- Randomly select $j \in [p_i, p_i + r^i - 1]$,
- Set $O_1 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(j))$.
- Set $\mathcal{S} = ((p_i, 0), (p_i + r^i, 0), (m+1, 1))$.
- Return (O_0, O_1, S).

- $(p_i, 0)$: snapshot of server memory before epoch i
- $(p_i + r^i, 0)$: snapshot of server memory after epoch i
- $(m+1, 1)$: snapshot before read and transcript of read operation

A $(3,1)$-snapshot adversary – Part 0

$\mathcal{A}_0^i(1^n)$

- Randomly select integer m from $[n/2, n]$.
- Randomly and ind. select $\mathsf{B}_1, \ldots, \mathsf{B}_m \leftarrow \{0,1\}^b$. **Important**
- Set $O_0 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(m))$.
- Randomly select $j \in [p_i, p_i + r^i - 1]$,
- Set $O_1 = (\mathsf{write}(1, \mathsf{B}_1), \ldots, \mathsf{write}(m, \mathsf{B}_m), \mathsf{read}(j))$.
- Set $\mathcal{S} = ((p_i, 0), (p_i + r^i, 0), (m+1, 1))$.
- Return (O_0, O_1, S).

- $(p_i, 0)$: snapshot of server memory before epoch i
- $(p_i + r^i, 0)$: snapshot of server memory after epoch i
- $(m+1, 1)$: snapshot before read and transcript of read operation

A $(3, 1)$-snapshot adversary – Part 1

- U_i memory locations overwritten during epoch i
  - by comparing the **initial** and **final** snapshot of epoch i
- V_i memory locations overwritten since epoch i
  - by comparing the **final** snapshot of epoch i with snapshot **before the read**
- W_i memory location overwritten during epoch i that have not been modified when the read starts
  - $W_i = U_i \setminus V_i$
- Q_j cells from W_i read during read(j),
- $|Q_j| \approx b/w$
  - $\mathcal{A}^1$ returns 0 iff $|Q_j| \leq \rho \cdot b/w$

The coding argument

Suppose

$$t_w = o(b/w \log(nb/c))$$

then there exists $\rho > 0$ such that, for most epochs i,

$$|Q_j| \geq \rho \cdot b/w$$

with probability $\geq 1/8$ for j in epoch i.

The coding argument

Suppose

$$t_w = o(b/w \log(nb/c))$$

then there exists $\rho > 0$ such that, for most epochs i,

$$|Q_j| \geq \rho \cdot b/w$$

with probability $\geq 1/8$ for j in epoch i.

Suppose not.

The coding argument

Suppose

$$t_w = o(b/w \log(nb/c))$$

then there exists $\rho > 0$ such that, for most epochs i,

$$|Q_j| \geq \rho \cdot b/w$$

with probability $\geq 1/8$ for j in epoch i.

Suppose not.
Then we can encode the $r^i \cdot b$ bits of epoch i using fewer bits.

The coding argument - I

A coding game

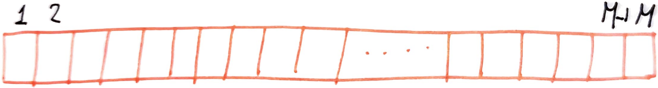
- S wants to send B^i to R
  - the r^i blocks from epoch i
- S and R share
  - B^{-i} (all except epoch i)
  - randomness $\mathcal{R}$ to execute DS.

$$\mathcal{H}(B^i | \mathcal{R}, B^{-i}) = r^i \cdot b.$$

The coding argument - II

- S and R execute all epochs $> i$

$$\mathrm{write}(1, \mathrm{B}_1), \ldots, \mathrm{write}(p_i - 1, \mathrm{B}_{p_i - 1})$$

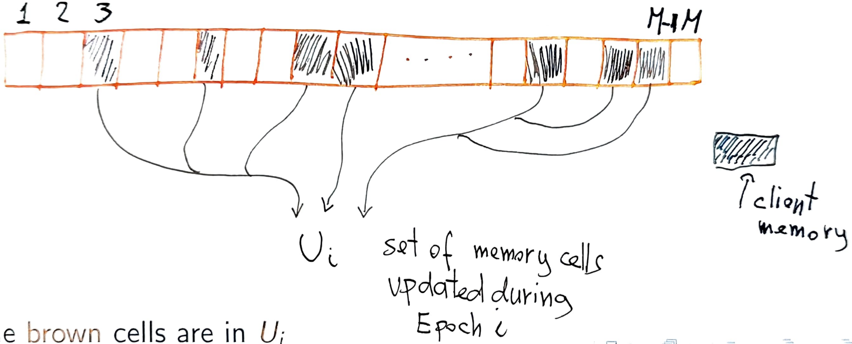


The coding argument

- S executes epoch i

$$\text{write}(p_i, \mathrm{B}_{p_i}), \ldots, \text{write}(p_i + r^i - 1, \mathrm{B}_{p_i+r^i-1})$$

- Note: R cannot execute epoch i

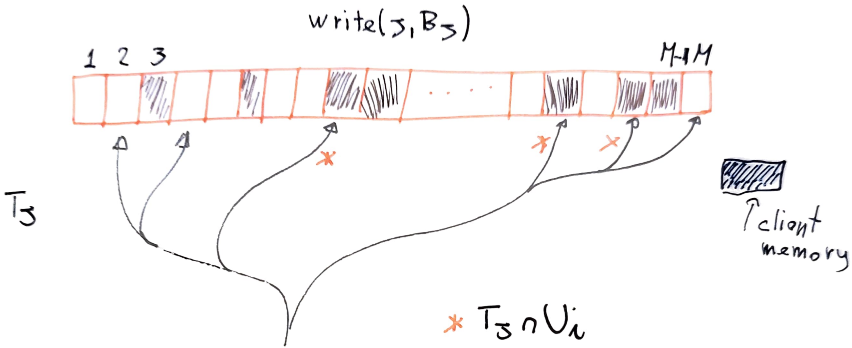


The brown cells are in U_i

The coding argument

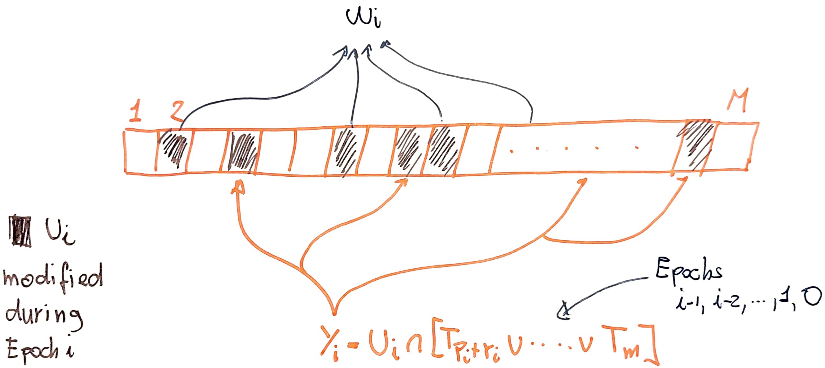
- S and R execute epochs $< i$
  - R needs some help
    - client memory: c bits.

- For $j = p_{i-1}, \ldots, m$
  - execute write(j, B_j) touching T_j
  - R needs $U_i \cap T_j$ (cell location and content)



The coding argument

- c bits + set $Y_i := U_i \cap (T_{p_i+r_i} \cup \cdots \cup T_m)$



The coding argument

$\mathbb{S}$ memory state after $\mathsf{write}(m, \mathsf{B}_m)$

- For $j = p_i, \ldots, p_i + r^i - 1$
  - S and R execute $\mathsf{read}(j)$ starting from $\mathbb{S}$
  - R needs $Q_j := W_i \cap T_j^m$
  - if read errs or $Q_j > \rho b/w$
    - B_j is added to encoding
  - else
    - Q_j is added to encoding

Length of encoding

Length depends on

- Set Y_i
  - for most epochs i, $\mathbb{E}[|Y_i|] \leq r^{i-1} b/w$
- Set Q_j
  - By assumption $|Q_j| < \rho \cdot b/w$ with prob $\geq 7/8$

Length of encoding

Length depends on

- Set Y_i
  - for most epochs i, $\mathbb{E}[|Y_i|] \leq r^{i-1} b/w$
- Set Q_j
  - By assumption $|Q_j| < \rho \cdot b/w$ with prob $\geq 7/8$

Encoding is too small

Getting there...

$$t_w = o(b/w \log(nb/c))$$

implies that, for most epochs i,

$$|Q_j| \geq \rho \cdot b/w$$

with probability $\geq 1/8$ for j in epoch i. from epoch i.

Getting there...

$$t_w = o(b/w \log(nb/c))$$

implies that, for most epochs i,

$\mathcal{A}$ outputs 1 with probability $\geq 1/8$ when reading j from epoch i.

Getting there...

$$t_w = o(b/w \log(nb/c))$$

implies that, for most epochs i,

$\mathcal{A}$ outputs 1 with probability $\geq 1/8$ when reading j from epoch i.

If $\epsilon = 1/16$ then $\mathcal{A}$ outputs 1 with probability $\geq 1/16$ when reading m

Getting there...

$$t_w = o(b/w \log(nb/c))$$

implies that, for most epochs i,

$\mathcal{A}$ outputs 1 with probability $\geq 1/8$ when reading j from epoch i.

If $\epsilon = 1/16$ then $\mathcal{A}$ outputs 1 with probability $\geq 1/16$ when reading m

$\mathrm{read}(m)$ must touch $\geq \rho \cdot b/w$ cells from epoch i

Getting there...

$$t_w = o(b/w \log(nb/c))$$

implies that, for most epochs i,

$\mathcal{A}$ outputs 1 with probability $\geq 1/8$ when reading j from epoch i.

If $\epsilon = 1/16$ then $\mathcal{A}$ outputs 1 with probability $\geq 1/16$ when reading m

$\mathrm{read}(m)$ must touch $\geq \rho \cdot b/w$ cells from epoch i

$$\Omega(b/w \cdot \log nb/c)$$

Wrapping up

Now...

If writes are fast

$$t_w = o(b/w \log(nb/c))$$

then $\mathsf{read}(j)$ in epoch i has $Q_j = \Omega(b/w)$ with prob at least $1/8$.

Wrapping up

Now...

If writes are fast

$$t_w = o(b/w \log(nb/c))$$

then $\mathsf{read}(j)$ in epoch i has $Q_j = \Omega(b/w)$ with prob at least $1/8$.

Reading 1

Must touch from each large epoch $O(b/w)$ cells otherwise we lose security.

$$\Omega(b/w \cdot \log(nb/c))$$

$(\infty, 0)$-snapshot secure stacks

Adversary gets snapshots of memory after all operations.

Snapshot Secure Stacks

- Init()
  - randomly choose encryption key K
  - set $\mathtt{cnt} = 0$ and $\mathtt{top} = -1$.
- Push(v)
  - upload $\mathrm{Enc}(K, (v, \mathtt{top}))$ to location $\mathtt{cnt}$
  - set $\mathtt{top} \leftarrow \mathtt{cnt}$
  - set $\mathtt{cnt} \leftarrow \mathtt{cnt} + 1$
- Pop()
  - download pair (v, t) from location $\mathtt{top}$
  - upload a dummy encryption to location $\mathtt{cnt}$
  - set $\mathtt{top} \leftarrow t$
  - set $\mathtt{cnt} \leftarrow \mathtt{cnt} + 1$
  - return v

$(\infty, 1)$-snapshot secure stacks

Adversary gets snapshots of memory after all operations and one transcript.

$(\infty, 1)$-snapshot secure stacks

Adversary gets snapshots of memory after all operations and one transcript.

Snapshot Secure Stacks

- Init()
  - randomly choose seed S
  - randomly choose encryption key K
  - set $\texttt{cnt} = 0$ and $\texttt{top} = -1$.
- Push(v)
  - download from location $F(S, \texttt{top})$ and discard
  - upload $\text{Enc}(K, (v, \texttt{top}))$ to location $F(S, \texttt{cnt})$
  - $\texttt{top} \leftarrow \texttt{cnt}$
  - $\texttt{cnt} \leftarrow \texttt{cnt} + 1$
- Pop()
  - download pair (v, t) from location $F(S, \texttt{top})$
  - upload dummy encryption at location $F(S, \texttt{cnt})$
  - set $\texttt{top} \leftarrow t$
  - set $\texttt{cnt} \leftarrow \texttt{cnt} + 1$
  - return v

Conclusions

- $\Theta(\log(N/C))$ for ORAM
  - Oblivious
  - DP
  - Leakage
  - Snapshot Adversary

Take home items

- Access pattern leakage is a privacy threat
  - Metadata
- It is possible to hide access pattern
  - at the cost of a logarithmic slowdown
- Theoretical questions:
  - is there a meaningfull security notion that requires constant slowdown?
  - construct oblivious algorithms for specific problems
- Theoretical questions:
  - can we get a *practical* Secure RAM for reasonable parameters?
    - server memory of about 100 GigaBytes
    - client memory of about 100 Megabyte
    - single digit slowdown