

Hiding Access Patterns

Obliviousness and Differential Privacy

Giuseppe Persiano

Università di Salerno

February 12, 2019

Describing joint work with:
Sarvar Patel, Mariana Raykova and Kevin Yeo (Google LLC)

- 1 Privacy in Cloud Storage
- 2 Oblivious Algorithms
- 3 An inefficient ORAM
- 4 An insecure ORAM
- 5 A first secure ORAM
- 6 Shuffling without Sorting
- 7 A second construction
- 8 A Recursive Construction
- 9 Lower Bounds
- 10 Differential Privacy
- 11 Where are we?

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?

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.

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.

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.

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

C;100

D;150

A;200

B;300

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.

- 1 download 1 and 3. decrypt, swap if out of order, re-encrypt, upload.
- 2 download 2 and 4. decrypt, swap if out of order, re-encrypt, upload.

C;100

D;150

A;200

B;300

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.

- 1 download 1 and 3. decrypt, swap if out of order, re-encrypt, upload.
- 2 download 2 and 4. decrypt, swap if out of order, re-encrypt, upload.
- 3 download 1 and 2. decrypt, swap if out of order, re-encrypt, upload.

C;100

D;150

A;200

B;300

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.

- 1 download 1 and 3. decrypt, swap if out of order, re-encrypt, upload.
- 2 download 2 and 4. decrypt, swap if out of order, re-encrypt, upload.
- 3 download 1 and 2. decrypt, swap if out of order, re-encrypt, upload.
- 4 download 3 and 4. decrypt, swap if out of order, re-encrypt, upload.

C;100

D;150

A;200

B;300

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.

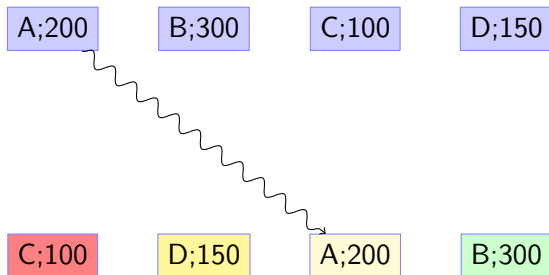
Example

4 customers must be sorted according to revenue.

- 1 download 1 and 3. decrypt, swap if out of order, re-encrypt, upload.
- 2 download 2 and 4. decrypt, swap if out of order, re-encrypt, upload.
- 3 download 1 and 2. decrypt, swap if out of order, re-encrypt, upload.
- 4 download 3 and 4. decrypt, swap if out of order, re-encrypt, upload.
- 5 download 2 and 3. decrypt, swap if out of order, re-encrypt, upload.

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*

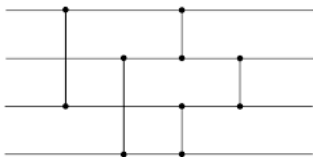
- Download, Decrypt, **Swap or Not**, Re-encrypt, Upload

Chosen-Ciphertext Security: Standard notion of security for encryption guarantee that \mathcal{M} is unable to deduce if a swap has happened.

Enter Obliviousness

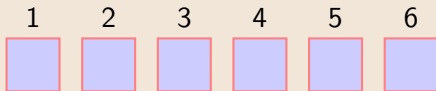
Definition (Weak Obliviousness)

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

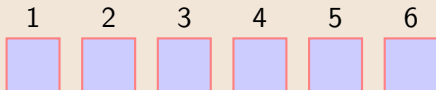
The adversarial setting



\mathcal{M}
 \mathcal{O}

A horizontal dashed green line extends from the right of the \mathcal{O} symbol.

The adversarial setting

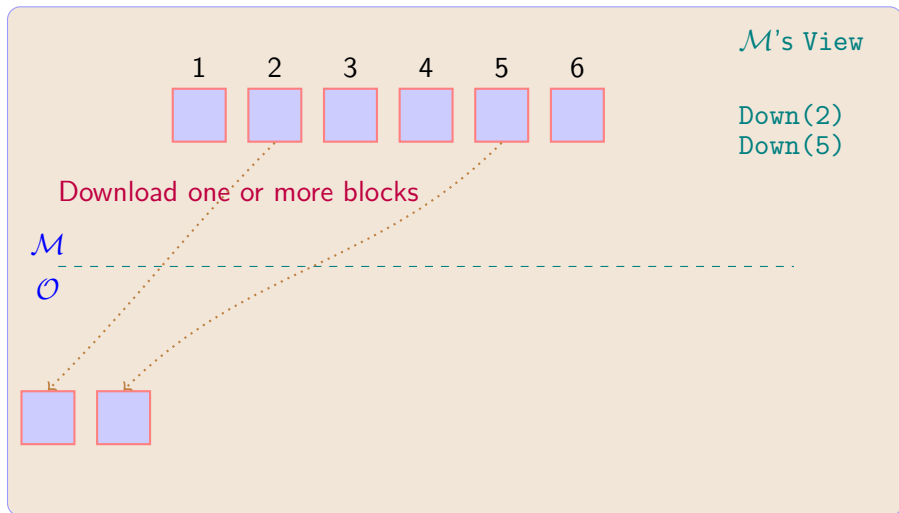


Download one or more blocks

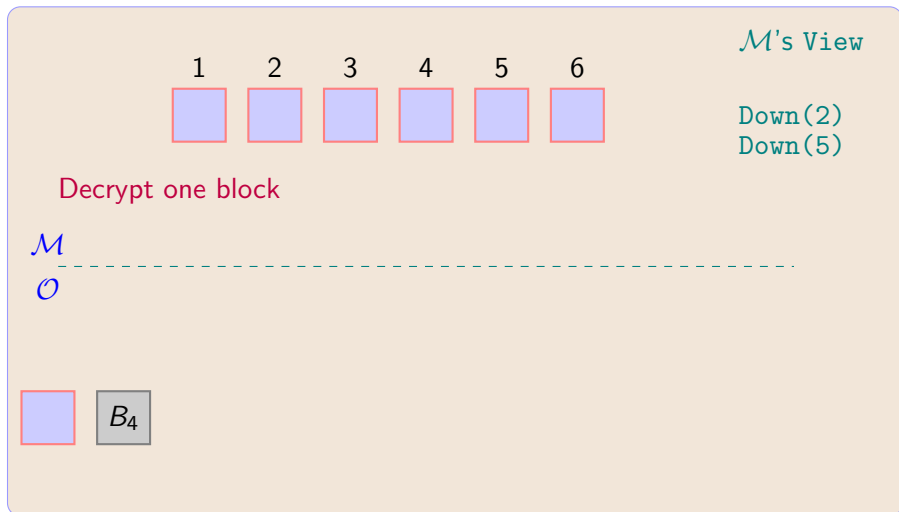
\mathcal{M}

\mathcal{O}

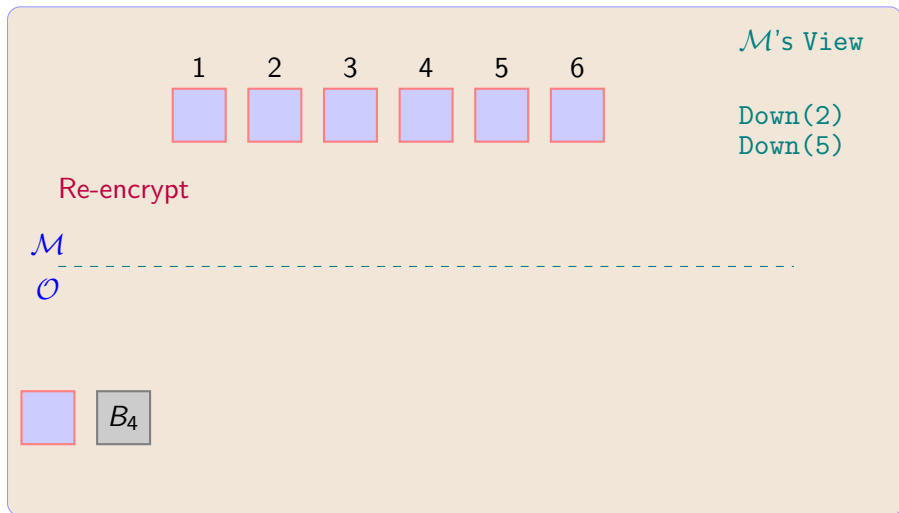
The adversarial setting



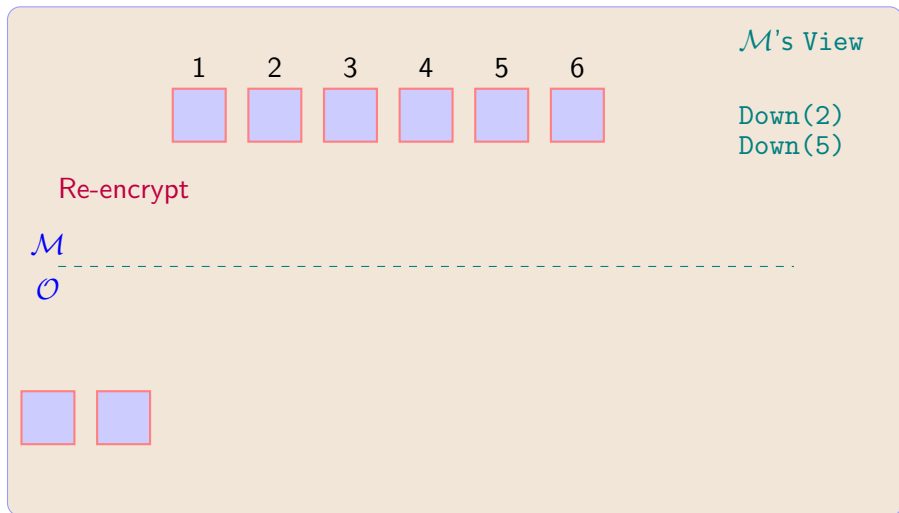
The adversarial setting



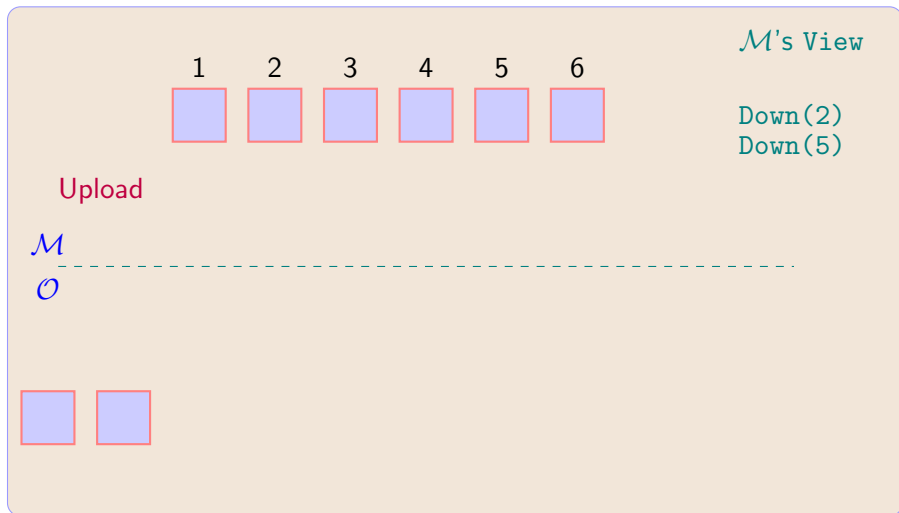
The adversarial setting



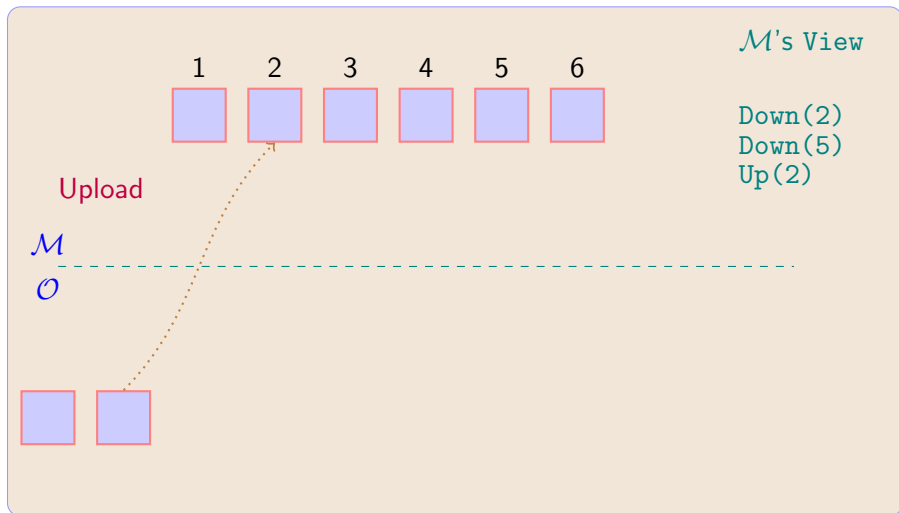
The adversarial setting



The adversarial setting



The adversarial setting



A new industry

Job Opportunities for Algorithmists

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

A new industry

Job Opportunities for Algorithmists

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

Abbiamo abolito la povertà

A new industry

Job Opportunities for Algorithmists

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

Abbiamo abolito la povertà ... per gli algoritmist

A new industry

Job Opportunities for Algorithmists

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

What? Just move on to the next slide and stop talking politics

Hiding the Algorithm

A new threat

- which algorithm is being run **should** also be private information

A;200

B;300

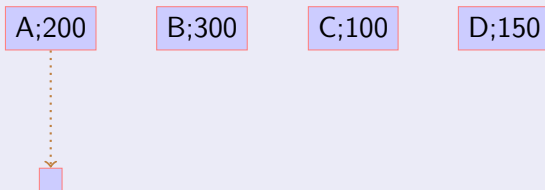
C;100

D;150

Hiding the Algorithm

A new threat

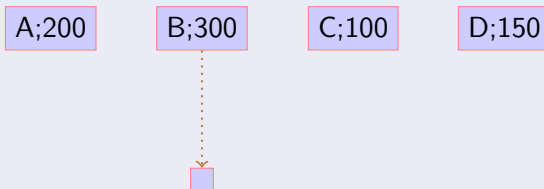
- which algorithm is being run **should** also be private information



Hiding the Algorithm

A new threat

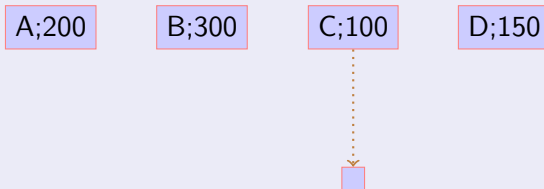
- which algorithm is being run **should** also be private information



Hiding the Algorithm

A new threat

- which algorithm is being run **should** also be private information



Hiding the Algorithm

A new threat

- which algorithm is being run **should** also be private information

A;200

B;300

C;100

D;150



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 $\mathbb{B} = B_1, \dots, B_n$ and $\mathbb{C} = C_1, \dots, C_n$
 - ▶ For any two access sequences $I = (i_1, \dots, i_l)$ and $J = (j_1, \dots, j_l)$
 - ★ performing accesses i_1, \dots, i_l on $\mathbb{B} = B_1, \dots, B_n$;
 - ★ performing access j_1, \dots, j_l on $\mathbb{C} = C_1, \dots, C_n$
- generate the same distribution of accesses to the data stored by \mathcal{M}

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 $\mathbb{B} = B_1, \dots, B_n$ and $\mathbb{C} = C_1, \dots, C_n$
 - ▶ For any two access sequences $I = (i_1, \dots, i_l)$ and $J = (j_1, \dots, j_l)$
 - ★ performing accesses i_1, \dots, i_l on $\mathbb{B} = B_1, \dots, B_n$;
 - ★ performing access j_1, \dots, j_l on $\mathbb{C} = C_1, \dots, C_n$
- generate the same distribution of accesses to the data stored by \mathcal{M}

For every predicate A

$$\begin{aligned} & \text{Prob}[\text{view} \leftarrow \text{View}(I, \mathbb{B}) : A(\text{view}) = 1] \\ & \leq e^0 \cdot \text{Prob}[\text{view} \leftarrow \text{View}(J, \mathbb{C}) : A(\text{view}) = 1] + \text{negl}(n) \end{aligned}$$

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.

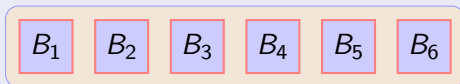
Is ORAM possible at all?

Yes! This is possible!

A Trivial ORAM

► Jump ahead

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



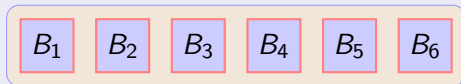
- Every time \mathcal{O} needs to access block B_i , all the blocks are downloaded and all except for B_i are discarded.

Yes! This is possible!

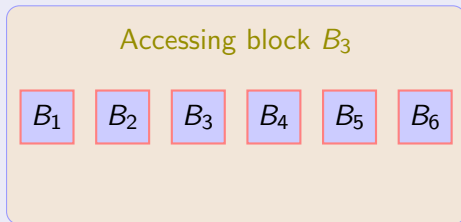
A Trivial ORAM

► Jump ahead

- 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.

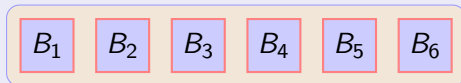


Yes! This is possible!

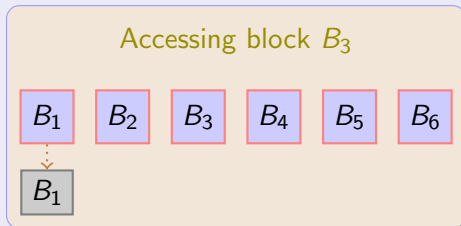
A Trivial ORAM

► Jump ahead

- 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.

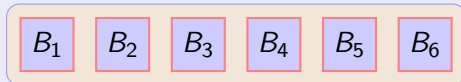


Yes! This is possible!

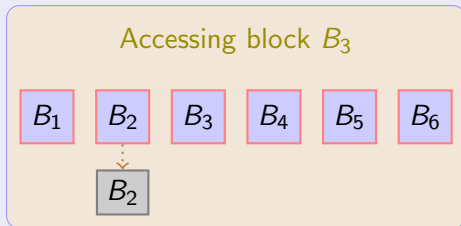
A Trivial ORAM

► Jump ahead

- 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.

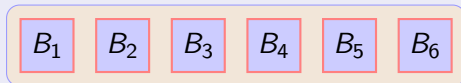


Yes! This is possible!

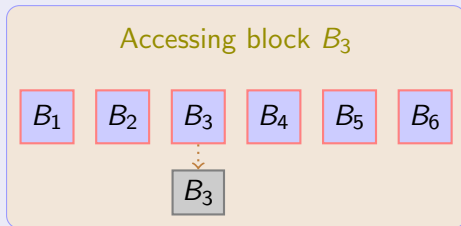
A Trivial ORAM

► Jump ahead

- 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.

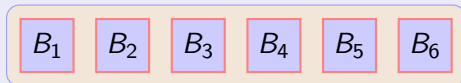


Yes! This is possible!

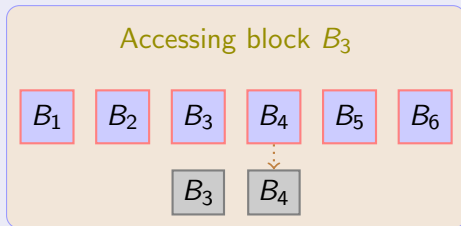
A Trivial ORAM

► Jump ahead

- 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.

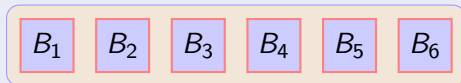


Yes! This is possible!

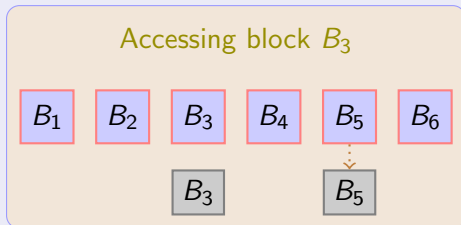
A Trivial ORAM

► Jump ahead

- 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.

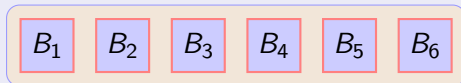


Yes! This is possible!

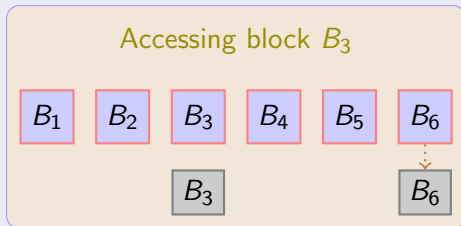
A Trivial ORAM

► Jump ahead

- 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.

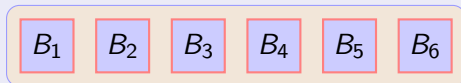


Yes! This is possible!

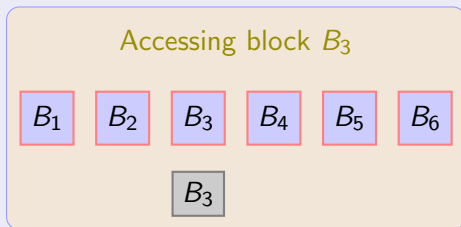
A Trivial ORAM

► Jump ahead

- 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.



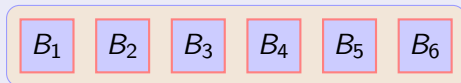
**Access pattern independent from the block accessed
but...**

Yes! This is possible!

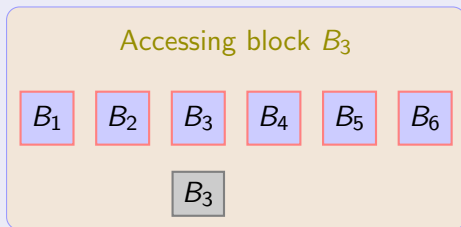
A Trivial ORAM

▶ Jump ahead

- 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.



**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 $\pi(i)$;



- \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 $\pi(i)$;



- \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

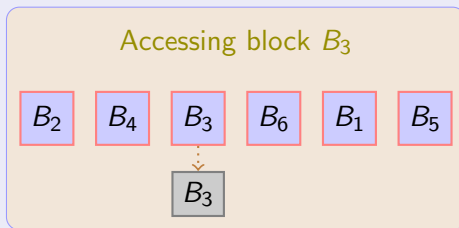


First try

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

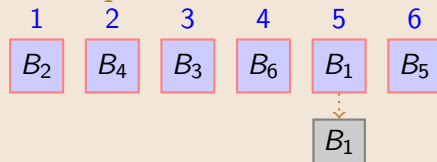


Access pattern seen by \mathcal{M} :

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_1

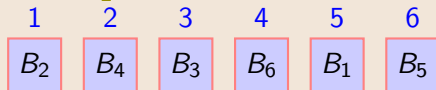


Access pattern seen by \mathcal{M} : 5

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_2

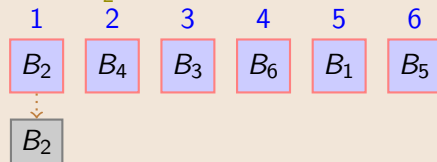


Access pattern seen by \mathcal{M} : 5

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_2

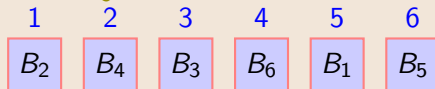


Access pattern seen by \mathcal{M} : 5, 1

First try: Security

Access sequence: B_1, B_2, B_3

Accessing block B_3

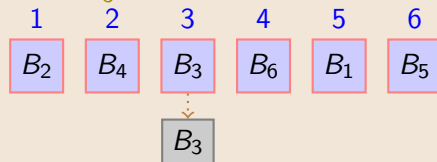


Access pattern seen by \mathcal{M} : 5, 1

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

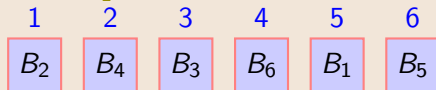
1	2	3	4	5	6
B_2	B_4	B_3	B_6	B_1	B_5

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

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_1

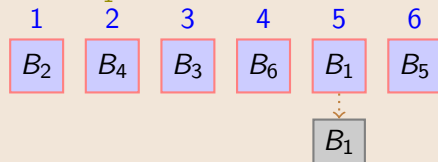


Access pattern seen by \mathcal{M} :

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

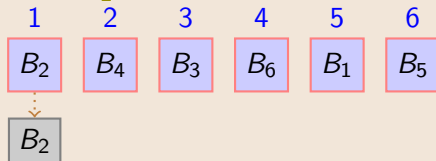


Access pattern seen by \mathcal{M} : 5

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_2

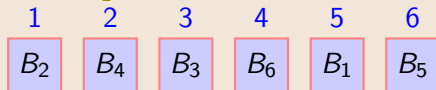


Access pattern seen by \mathcal{M} : 5, 1

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block B_1

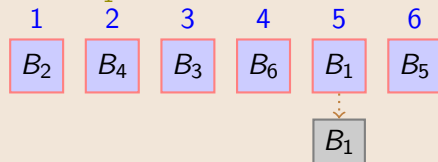


Access pattern seen by \mathcal{M} : 5, 1

First try: Security

Access sequence: B_1, B_2, B_1

Accessing block 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

Oblivious RAM

Obliviousness

For any two *access sequences* $O_1 = (i_1^1, \dots, i_l^1)$ $O_2 = (i_1^2, \dots, i_l^2)$ of the same length, the distribution of the positions requested by \mathcal{O} to \mathcal{M} is the same.

Oblivious RAM

Obliviousness

For any two *access sequences* $O_1 = (i_1^1, \dots, i_l^1)$ $O_2 = (i_1^2, \dots, i_l^2)$ of the same length, the distribution of the positions requested by \mathcal{O} to \mathcal{M} is the same.

Oblivious for Non-repeating sequences

- $k_1 \neq k_2$ implies $i_{k_1}^1 \neq i_{k_2}^1$ and $i_{k_1}^2 \neq i_{k_2}^2$;
- \mathcal{M} sees requests for l different randomly chosen blocks both for O_1 and for O_2 .

Repetition Pattern is leaked

Repetition Pattern

If the same block is requested twice by \mathcal{O} then \mathcal{M} sees the same position accessed twice.

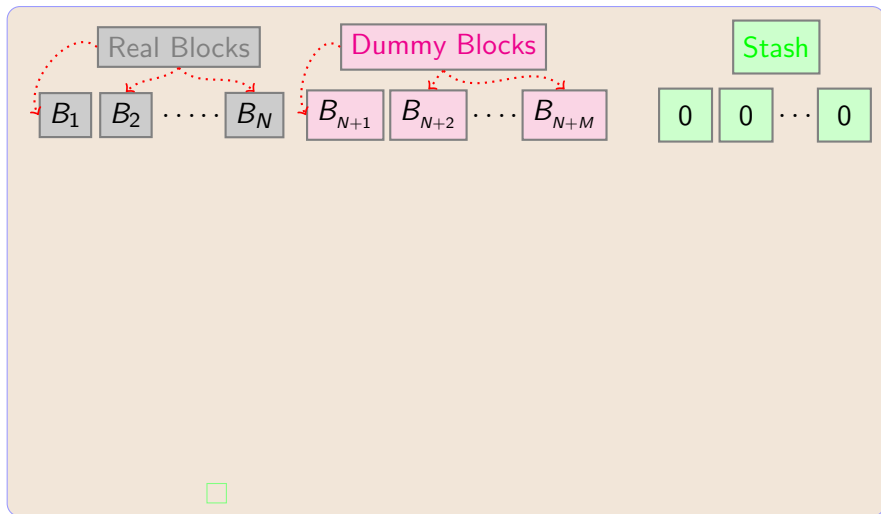
Block	3	4	7	8	4	2	4	10	12	8	6
Position	12	2	9	3	2	6	2	10	1	3	5

Hiding the Repetition Pattern

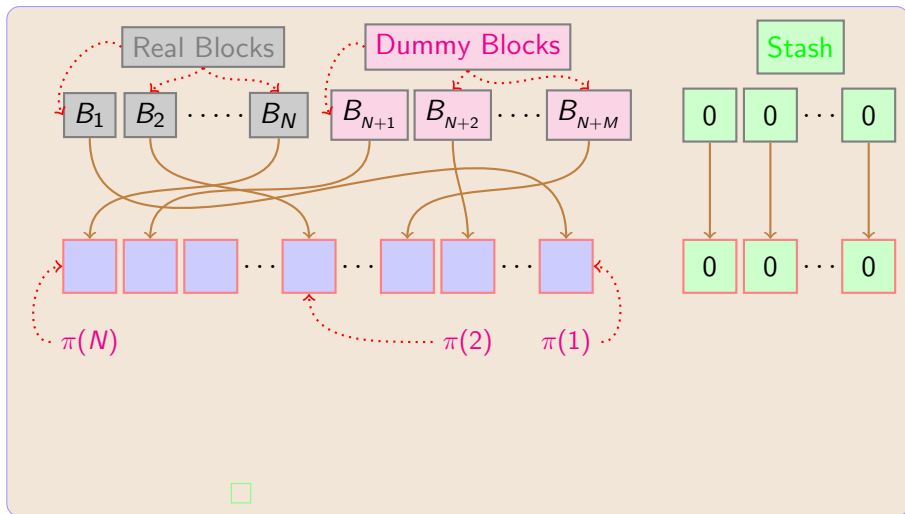
Initialization for N blocks

- 1 N *real* blocks B_1, \dots, B_N ;
- 2 create M *dummy* blocks B_{N+1}, \dots, B_{N+M} ;
- 3 create M *stash* blocks S_1, \dots, 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 $\text{nxt} = 1, \text{cnt} = 1$;
- 8 π is kept private;

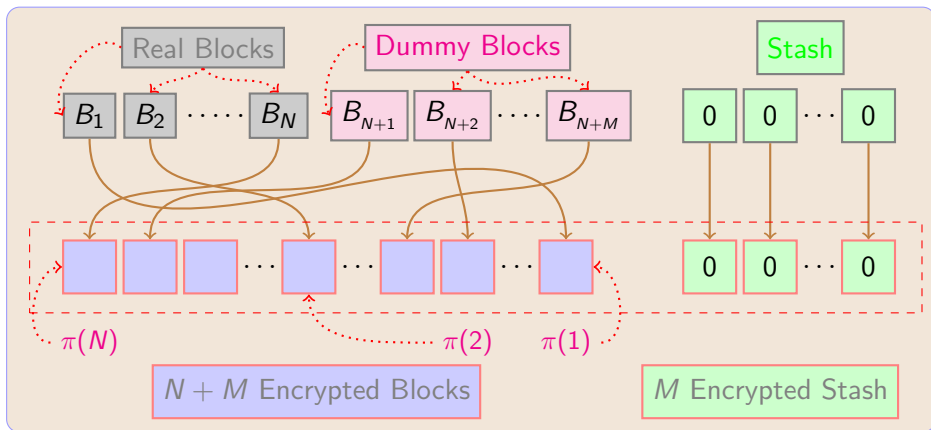
Initial Configuration



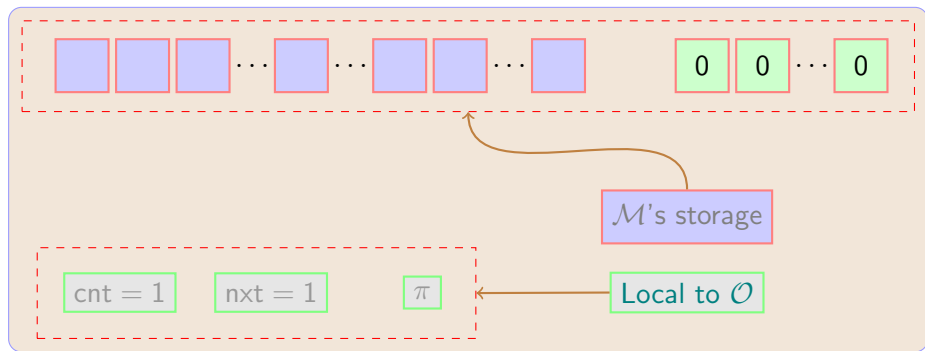
Initial Configuration



Initial Configuration



Initial Configuration

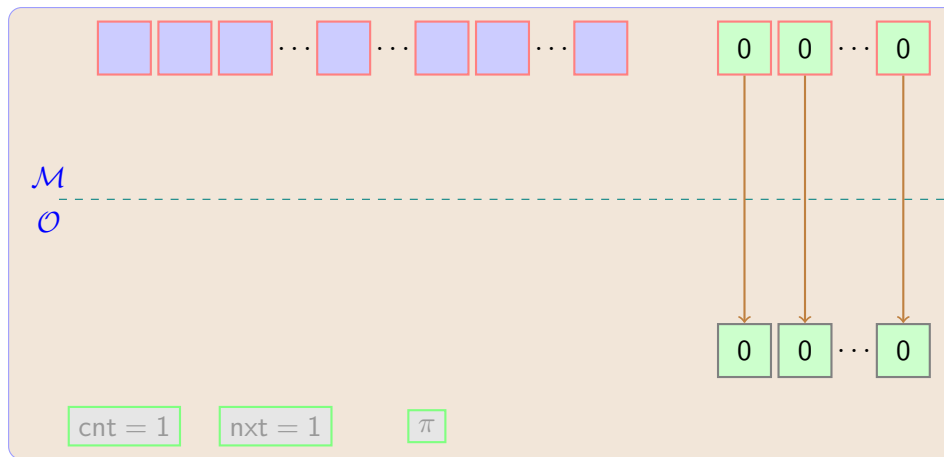


Reading Block B_i

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

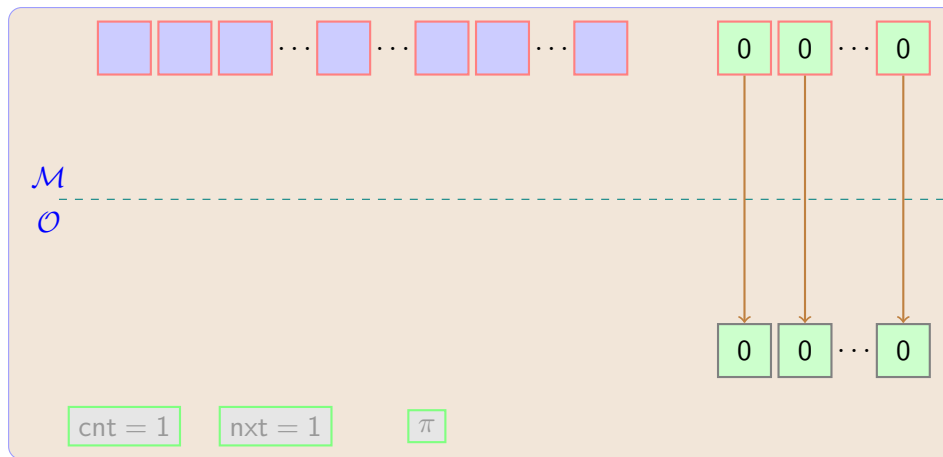
Reading Block B_1

Download and decrypt all blocks from Stash



Reading Block B_1

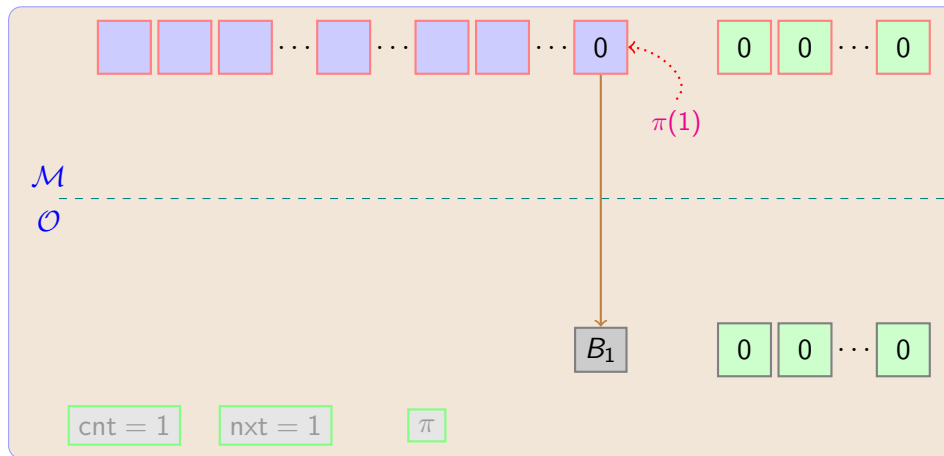
Download and decrypt all blocks from Stash



B_1 is not found in the stash

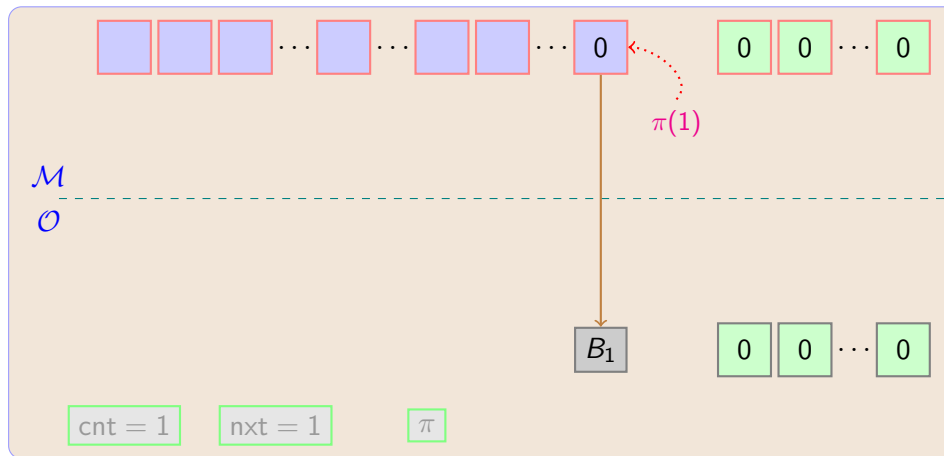
Reading Block B_1

Download block in position $\pi(1)$



Reading Block B_1

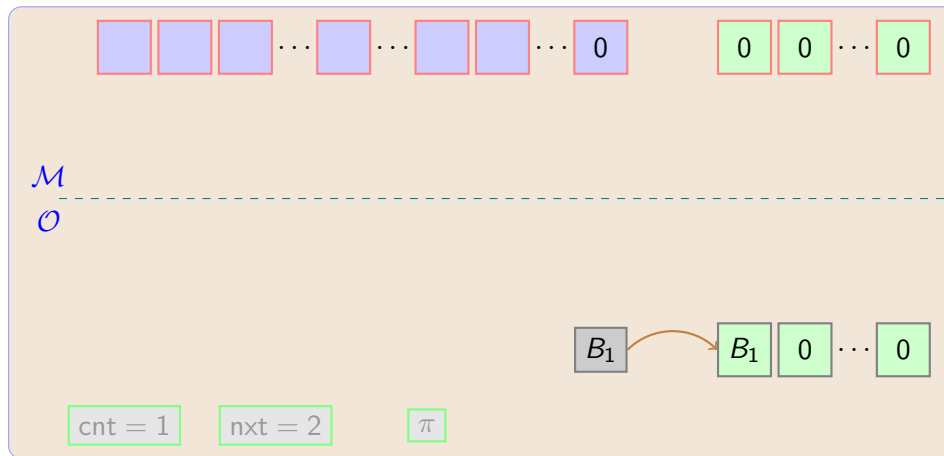
Download block in position $\pi(1)$



Decrypt and obtain B_1

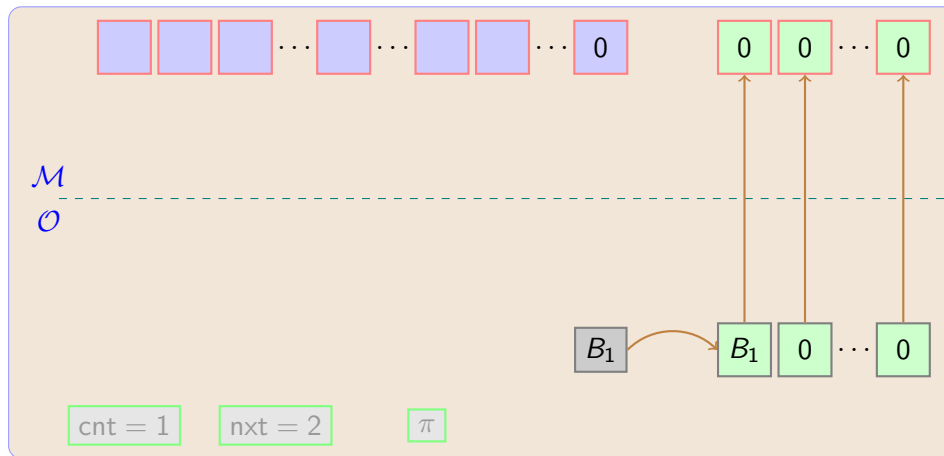
Reading Block B_1

Copy B_1 in the Stash at position next



Reading Block B_1

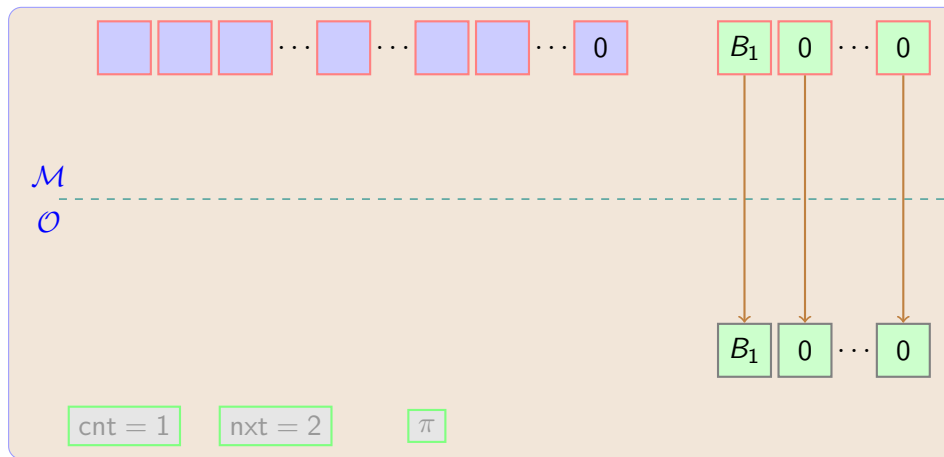
Copy B_1 in the Stash at position next



Encrypt and Upload the Stash

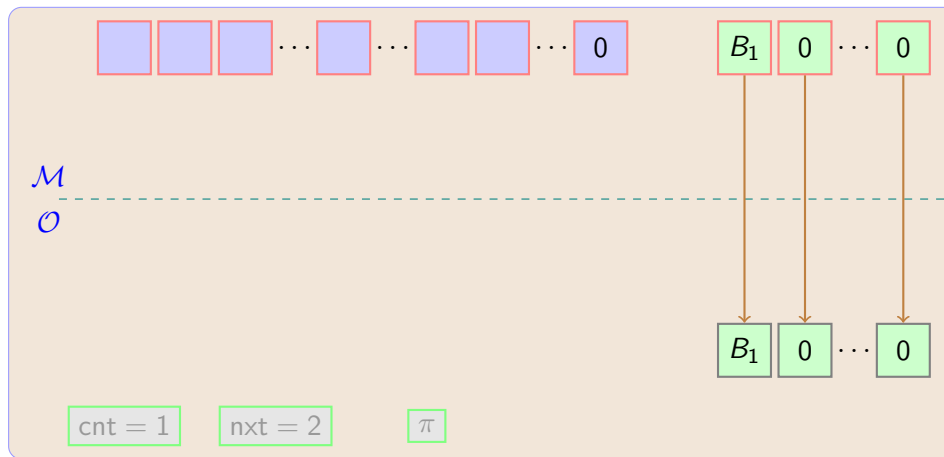
Reading Block B_2

Download and decrypt all blocks from Stash



Reading Block B_2

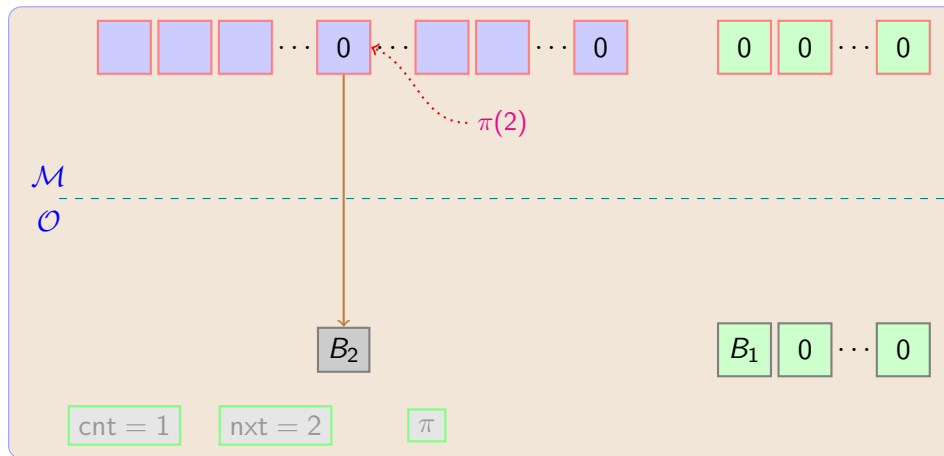
Download and decrypt all blocks from Stash



B_2 is not found in the Stash

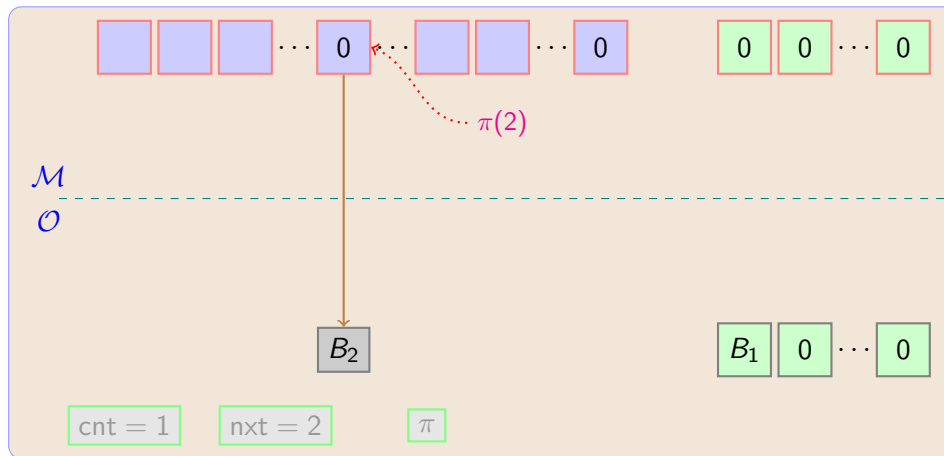
Reading Block B_2

Download block in position $\pi(2)$



Reading Block B_2

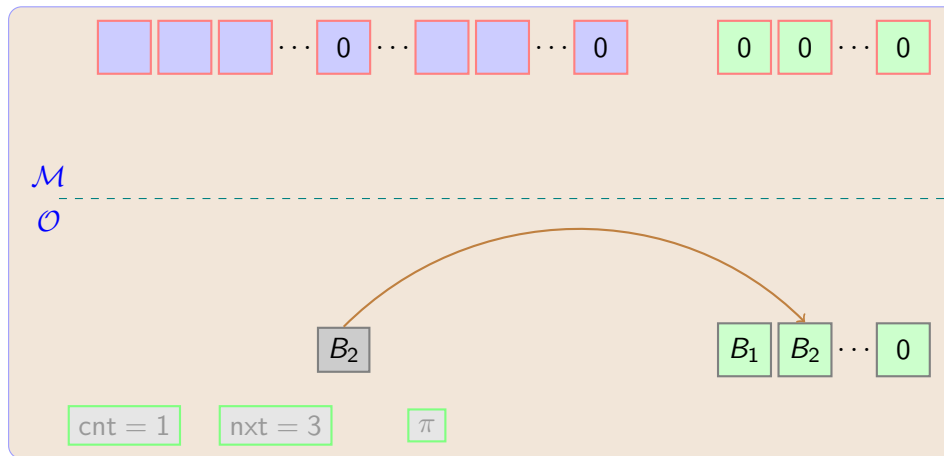
Download block in position $\pi(2)$



Decrypt and obtain B_2

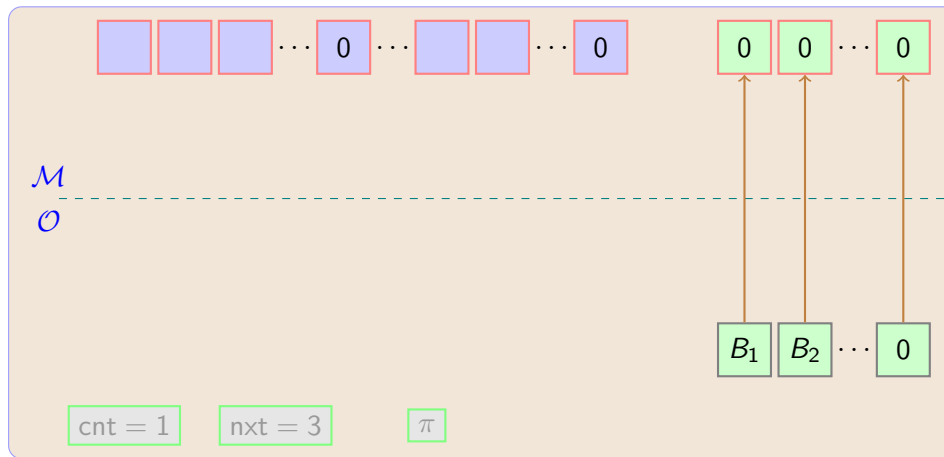
Reading Block B_2

Copy B_2 in the Stash at position next



Reading Block B_2

Copy B_2 in the Stash at position next



Encrypt and Upload the Stash

Status after reading B_1 and B_2



\mathcal{M}

 \mathcal{O}

cnt = 1

nxt = 2

π

Status after reading B_1 and B_2

Now read B_1 again



\mathcal{M}
—
 \mathcal{O}

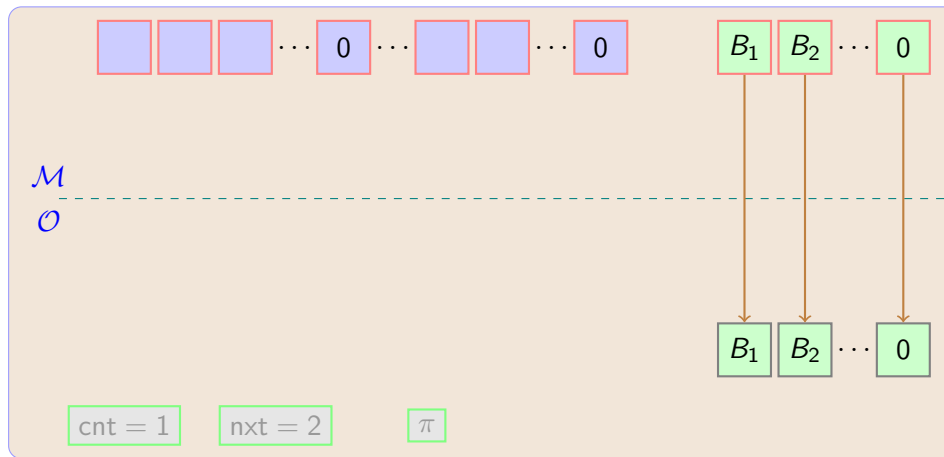
cnt = 1

nxt = 2

π

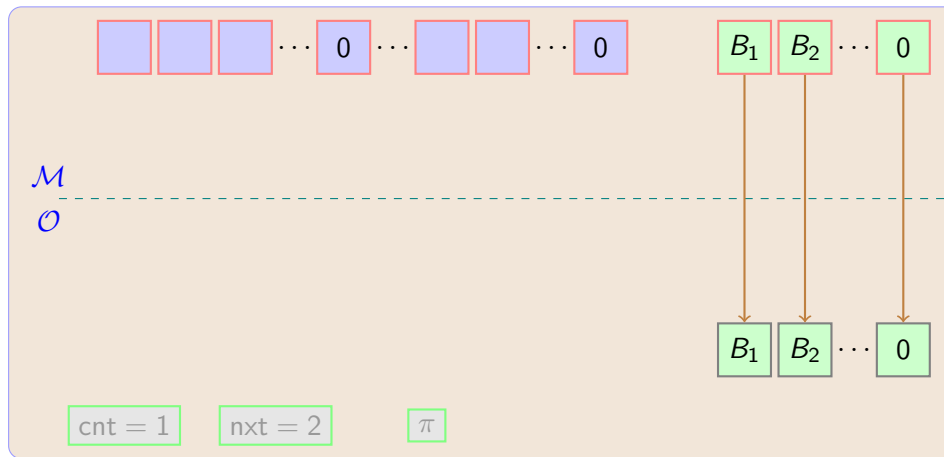
Status after reading B_1 and B_2

Download and decrypt all blocks from Stash



Status after reading B_1 and B_2

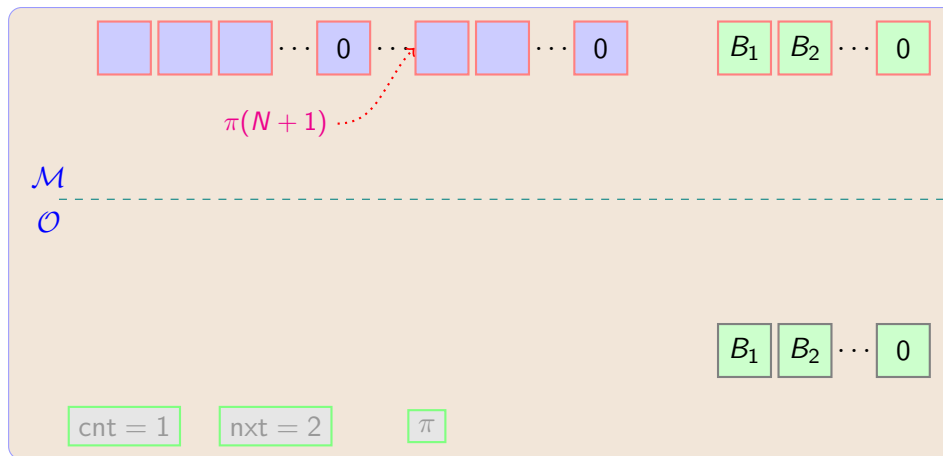
Download and decrypt all blocks from Stash



B_1 is found in the Stash

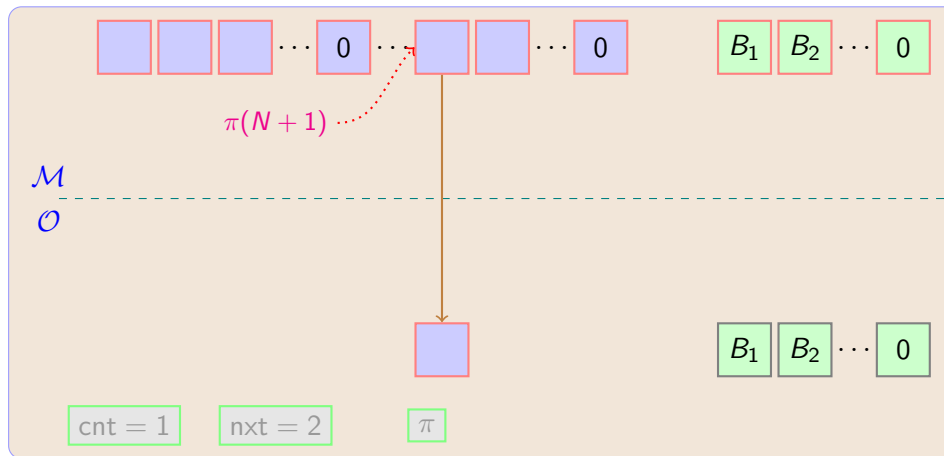
Reading Block B_1 (again)

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



Reading Block B_1 (again)

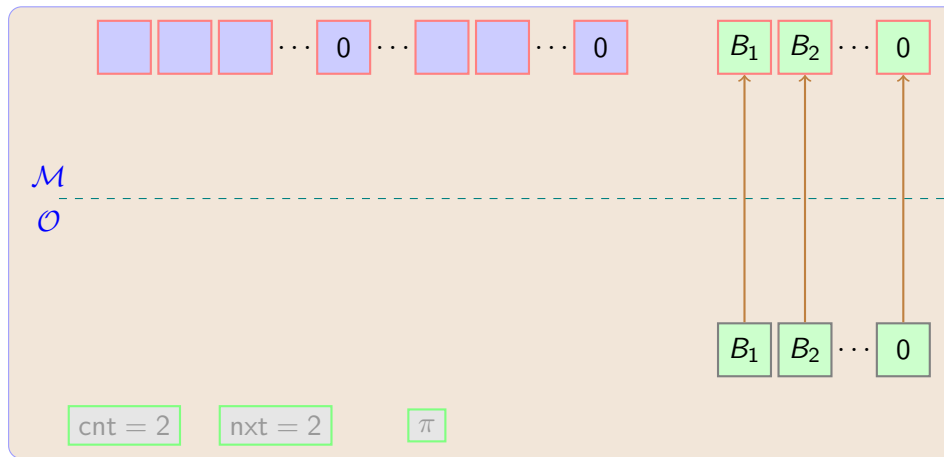
Download block in position $\pi(N + \text{cnt})$



No need to decrypt

Reading Block B_1 (again)

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



Encrypt and Upload Stash

Insert slide in which we argue obliviousness

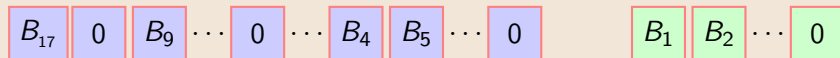
Two issues to be dealt with

- 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.

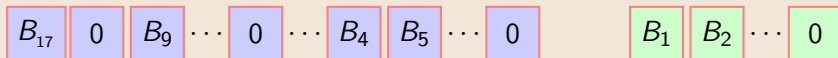
Overflowing the Stash



\mathcal{M}
—
 \mathcal{O}



Overflowing the Stash

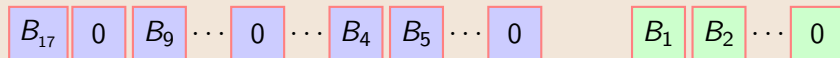


\mathcal{M}
 \mathcal{O}

Randomly select a new permutation σ



Overflowing the Stash

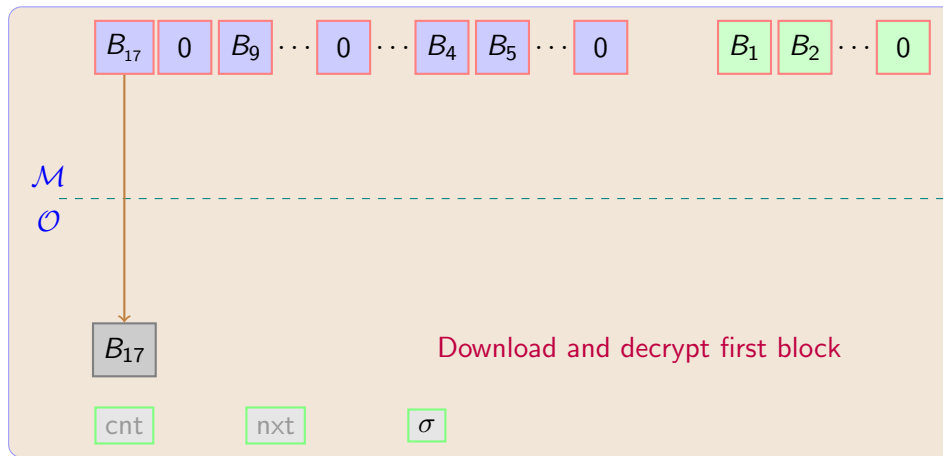


\mathcal{M}

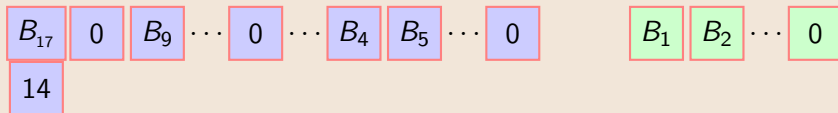
 \mathcal{O}



Overflowing the Stash



Overflowing the Stash

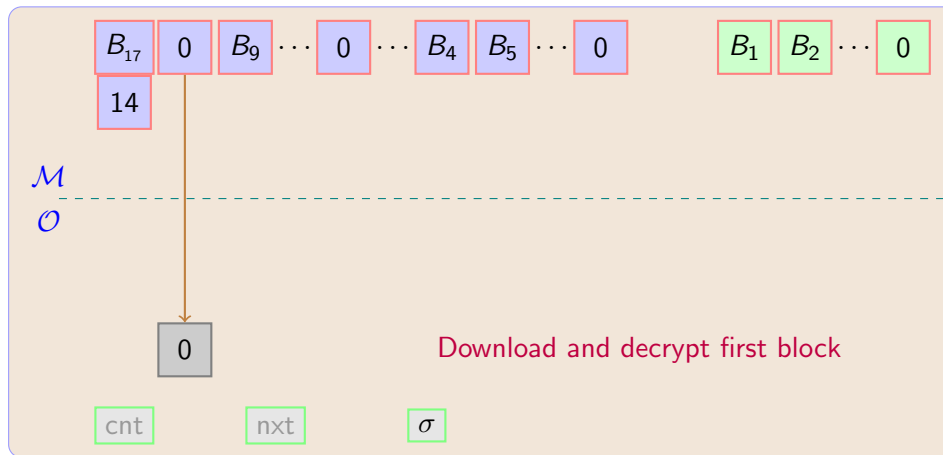


\mathcal{M}
 \mathcal{O}

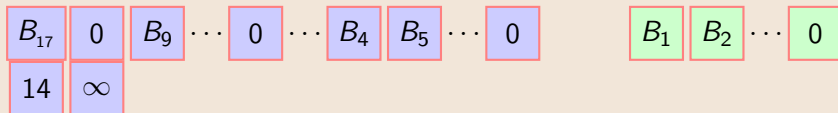
Tag it with $\sigma(17)$ encrypt and upload



Overflowing the Stash



Overflowing the Stash

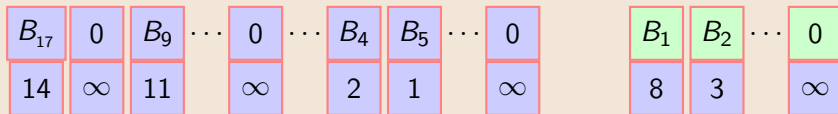


\mathcal{M}
 \mathcal{O}

Tag it with ∞ encrypt and upload



Overflowing the Stash

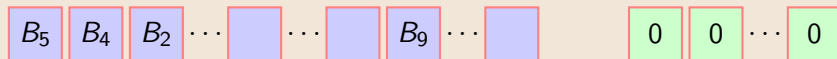


\mathcal{M}
—
 \mathcal{O}

Obliviously sort according to tags

cnt nxt σ

Overflowing the Stash



\mathcal{M}
—
 \mathcal{O}

cnt = 1

nxt = 1

σ

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.

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

Huge constant

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 Batchers'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

Shuffling without Sorting

- **Input:** N blocks stored in $S[1, \dots, N]$ according to π

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

Blocks in S according to π

Shuffling without Sorting

- **Input:** N blocks stored in $S[1, \dots, N]$ according to π
 - ▶ Block B_I is found in $[\pi(I)]$

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

Blocks in S according to π

Shuffling without Sorting

- **Input:** N blocks stored in $S[1, \dots, N]$ according to π
 - ▶ Block B_i is found in $[\pi(i)]$
- **Output:** N blocks stored in $D[1, \dots, N]$ according to σ

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

Blocks in S according to π

Shuffling without Sorting

- **Input:** N blocks stored in $S[1, \dots, N]$ according to π
 - ▶ Block B_I is found in $[\pi(I)]$
- **Output:** N blocks stored in $D[1, \dots, N]$ according to σ
 - ▶ Block B_I will be in $[\sigma(I)]$

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

Blocks in S according to π

16	15	5	1
14	2	4	9
10	7	11	6
13	12	8	3

Blocks in D according to σ

Shuffling $N = 16$

An easy case:

Partition the N blocks in \sqrt{N} groups of \sqrt{N}

Blocks in S according to π

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

\mathcal{M}

\mathcal{O}

Shuffling $N = 16$

An easy case:

Partition the N destinations in \sqrt{N} groups of \sqrt{N}

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download first source group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download first source group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

\mathcal{O}

3	4	2	10
---	---	---	----

Shuffling $N = 16$

An easy case:

One block to each destination group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
0	0	0	0
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download second source group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
0	0	0	0
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download second source group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
0	0	0	0
10	2	4	3

\mathcal{O}

9	8	15	16
---	---	----	----

Shuffling $N = 16$

An easy case:

One block to each destination group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
16	15	8	9
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download second source group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
16	15	8	9
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download second source group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
0	0	0	0
16	15	8	9
10	2	4	3

\mathcal{O}

1	5	7	14
---	---	---	----

Shuffling $N = 16$

An easy case:

One block to each destination group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
14	7	5	1
16	15	8	9
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download second source group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
14	7	5	1
16	15	8	9
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download second source group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

0	0	0	0
14	7	5	1
16	15	8	9
10	2	4	3

\mathcal{O}

6	11	12	13
---	----	----	----

Shuffling $N = 16$

An easy case:

One block to each destination group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

13	12	11	6
14	7	5	1
16	15	8	9
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Each block in the right destination group

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

13	12	11	6
14	7	5	1
16	15	8	9
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

13	12	11	6
14	7	5	1
16	15	8	9
10	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

13	12	11	6
14	7	5	1
16	15	8	9
10	2	4	3

\mathcal{O}

10	16	14	13
----	----	----	----

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

16	12	11	6
14	7	5	1
10	15	8	9
13	2	4	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

16	12	11	6
14	7	5	1
10	15	8	9
13	2	4	3

\mathcal{O}

2	15	7	12
---	----	---	----

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

16	15	11	6
14	2	5	1
10	7	8	9
13	12	4	3

\mathcal{O}

4	8	5	11
---	---	---	----

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

16	15	5	6
14	2	4	1
10	7	11	9
13	12	8	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

16	15	5	6
14	2	4	1
10	7	11	9
13	12	8	3

\mathcal{O}

3	9	1	6
---	---	---	---

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

\mathcal{M}

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

16	15	5	1
14	2	4	9
10	7	11	6
13	12	8	3

\mathcal{O}

Shuffling $N = 16$

An easy case:

Download each group and upload in correct position

Blocks in S according to π

10	16	14	13
2	15	7	12
4	8	5	11
3	9	1	6

Blocks in D according to σ

16	15	5	1
14	2	4	9
10	7	11	6
13	12	8	3

\mathcal{M}

\mathcal{O}

Analysis of Shuffling Algorithm

- **Obliviousness:** access pattern independent of π, σ

Analysis of Shuffling Algorithm

- **Obliviousness:** access pattern independent of π, σ
 - ▶ download each source group

Analysis of Shuffling Algorithm

- **Obliviousness:** access pattern independent of π, σ
 - ▶ download each source group
 - ▶ upload one block to each destination group in the next available empty location

Analysis of Shuffling Algorithm

- **Obliviousness:** access pattern independent of π, σ
 - ▶ download each source group
 - ▶ upload one block to each destination group in the next available empty location
 - ▶ download each destination group

Analysis of Shuffling Algorithm

- **Obliviousness:** access pattern independent of π, σ
 - ▶ download each source group
 - ▶ upload one block to each destination group in the next available empty location
 - ▶ download each destination group
 - ▶ upload each destination group

Analysis of Shuffling Algorithm

- **Obliviousness**: access pattern independent of π, σ
 - ▶ download each source group
 - ▶ upload one block to each destination group in the next available empty location
 - ▶ download each destination group
 - ▶ upload each destination group
- **bandwidth**: $4N$
each block

Analysis of Shuffling Algorithm

- **Obliviousness**: access pattern independent of π, σ
 - ▶ download each source group
 - ▶ upload one block to each destination group in the next available empty location
 - ▶ download each destination group
 - ▶ upload each destination group
- **bandwidth**: $4N$
each block
 - ▶ downloaded exactly twice

Analysis of Shuffling Algorithm

- **Obliviousness**: access pattern independent of π, σ
 - ▶ download each source group
 - ▶ upload one block to each destination group in the next available empty location
 - ▶ download each destination group
 - ▶ upload each destination group
- **bandwidth**: $4N$
each block
 - ▶ downloaded exactly twice
 - ▶ uploaded exactly twice

Analysis of Shuffling Algorithm

- **Obliviousness**: access pattern independent of π, σ
 - ▶ download each source group
 - ▶ upload one block to each destination group in the next available empty location
 - ▶ download each destination group
 - ▶ upload each destination group
- **bandwidth**: $4N$
each block
 - ▶ downloaded exactly twice
 - ▶ uploaded exactly twice
- **Luck**: so much!!!

Analysis of Shuffling Algorithm

- **Obliviousness**: access pattern independent of π, σ
 - ▶ download each source group
 - ▶ upload one block to each destination group in the next available empty location
 - ▶ download each destination group
 - ▶ upload each destination group
- **bandwidth**: $4N$
each block
 - ▶ downloaded exactly twice
 - ▶ uploaded exactly twice
- **Luck**: so much!!!
 - ▶ each source group contains exactly one block for each destination group under σ

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase
 - ▶ Download each source group and spray exactly one block to each destination group

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase
 - ▶ Download each source group and spray exactly one block to each destination group
 - ▶ if none available, spray a dummy block

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase
 - ▶ Download each source group and spray exactly one block to each destination group
 - ▶ if none available, spray a dummy block
 - ▶ if more than one available, spray exactly one and store the extra blocks in local cache

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase
 - ▶ Download each source group and spray exactly one block to each destination group
 - ▶ if none available, spray a dummy block
 - ▶ if more than one available, spray exactly one and store the extra blocks in local cache
- Recalibrate phase

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase
 - ▶ Download each source group and spray exactly one block to each destination group
 - ▶ if none available, spray a dummy block
 - ▶ if more than one available, spray exactly one and store the extra blocks in local cache
- Recalibrate phase
 - ▶ download each destination group

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase
 - ▶ Download each source group and spray exactly one block to each destination group
 - ▶ if none available, spray a dummy block
 - ▶ if more than one available, spray exactly one and store the extra blocks in local cache
- Recalibrate phase
 - ▶ download each destination group
 - ▶ remove dummies

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase
 - ▶ Download each source group and spray exactly one block to each destination group
 - ▶ if none available, spray a dummy block
 - ▶ if more than one available, spray exactly one and store the extra blocks in local cache
- Recalibrate phase
 - ▶ download each destination group
 - ▶ remove dummies
 - ▶ add blocks found in local cache

Efficient Shuffling: CacheShuffleRoot

when you know you are not going to be lucky, just randomize

- Randomly partition destination array in $(1 + 2\epsilon)\sqrt{N}$ groups
 - ▶ Each group has size at most $(1 - \epsilon)\sqrt{N}$, except with negligible probability
- Spray phase
 - ▶ Download each source group and spray exactly one block to each destination group
 - ▶ if none available, spray a dummy block
 - ▶ if more than one available, spray exactly one and store the extra blocks in local cache
- Recalibrate phase
 - ▶ download each destination group
 - ▶ remove dummies
 - ▶ add blocks found in local cache
 - ▶ upload in correct order

Analysis of CacheShuffleRoot

- bandwidth: $(4 + \epsilon)N$ blocks

Analysis of CacheShuffleRoot

- **bandwidth**: $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once
 - ▶ number of **dummy** blocks is at most $\epsilon \cdot N$ except with negligible probability

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once
 - ▶ number of **dummy** blocks is at most $\epsilon \cdot N$ except with negligible probability
- **Obliviousness:** access pattern is fixed for all executions

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once
 - ▶ number of **dummy** blocks is at most $\epsilon \cdot N$ except with negligible probability
- **Obliviousness:** access pattern is fixed for all executions
 - ▶ download each source group

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once
 - ▶ number of **dummy** blocks is at most $\epsilon \cdot N$ except with negligible probability
- **Obliviousness:** access pattern is fixed for all executions
 - ▶ download each source group
 - ▶ upload one block to each destination group

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once
 - ▶ number of **dummy** blocks is at most $\epsilon \cdot N$ except with negligible probability
- **Obliviousness:** access pattern is fixed for all executions
 - ▶ download each source group
 - ▶ upload one block to each destination group
 - ▶ when done, download and upload each destination group

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once
 - ▶ number of **dummy** blocks is at most $\epsilon \cdot N$ except with negligible probability
- **Obliviousness:** access pattern is fixed for all executions
 - ▶ download each source group
 - ▶ upload one block to each destination group
 - ▶ when done, download and upload each destination group
- **\mathcal{M} 's storage:** $\Theta(N)$

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once
 - ▶ number of **dummy** blocks is at most $\epsilon \cdot N$ except with negligible probability
- **Obliviousness:** access pattern is fixed for all executions
 - ▶ download each source group
 - ▶ upload one block to each destination group
 - ▶ when done, download and upload each destination group
- **\mathcal{M} 's storage:** $\Theta(N)$
- **\mathcal{O} 's storage**

Analysis of CacheShuffleRoot

- **bandwidth:** $(4 + \epsilon)N$ blocks
 - ▶ each **real** block is downloaded exactly twice and uploaded exactly twice
 - ▶ each **dummy** block is uploaded exactly once and downloaded exactly once
 - ▶ number of **dummy** blocks is at most $\epsilon \cdot N$ except with negligible probability
- **Obliviousness:** access pattern is fixed for all executions
 - ▶ download each source group
 - ▶ upload one block to each destination group
 - ▶ when done, download and upload each destination group
- **\mathcal{M} 's storage:** $\Theta(N)$
- **\mathcal{O} 's storage**
 - ▶ next slide

Analysis of CacheShuffleRoot: \mathcal{O} 's storage

Intuition

- \mathcal{O} 's keeps a cache for each of the $(1 + 2\epsilon)\sqrt{N}$ destination groups

Analysis of CacheShuffleRoot: \mathcal{O} 's storage

Intuition

- \mathcal{O} 's keeps a cache for each of the $(1 + 2\epsilon)\sqrt{N}$ destination groups
- each cache has arrival rate $\approx (1 - 2\epsilon)$

Analysis of CacheShuffleRoot: \mathcal{O} 's storage

Intuition

- \mathcal{O} 's keeps a cache for each of the $(1 + 2\epsilon)\sqrt{N}$ destination groups
- each cache has arrival rate $\approx (1 - 2\epsilon)$
- departure rate 1

Analysis of CacheShuffleRoot: \mathcal{O} 's storage

Intuition

- \mathcal{O} 's keeps a cache for each of the $(1 + 2\epsilon)\sqrt{N}$ destination groups
- each cache has arrival rate $\approx (1 - 2\epsilon)$
- departure rate 1
- each cache is expected to hold a constant number of blocks

Analysis of CacheShuffleRoot: \mathcal{O} 's storage

Intuition

- \mathcal{O} 's keeps a cache for each of the $(1 + 2\epsilon)\sqrt{N}$ destination groups
- each cache has arrival rate $\approx (1 - 2\epsilon)$
- departure rate 1
- each cache is expected to hold a constant number of blocks
- total \mathcal{O} 's storage is $\Theta(\sqrt{N})$ except with negligible probability

Analysis of CacheShuffleRoot: \mathcal{O} 's storage

Intuition

- \mathcal{O} 's keeps a cache for each of the $(1 + 2\epsilon)\sqrt{N}$ destination groups
- each cache has arrival rate $\approx (1 - 2\epsilon)$
- departure rate 1
- each cache is expected to hold a constant number of blocks
- total \mathcal{O} 's storage is $\Theta(\sqrt{N})$ except with negligible probability

Analysis of CacheShuffleRoot: \mathcal{O} 's storage

Intuition

- \mathcal{O} 's keeps a cache for each of the $(1 + 2\epsilon)\sqrt{N}$ destination groups
- each cache has arrival rate $\approx (1 - 2\epsilon)$
- departure rate 1
- each cache is expected to hold a constant number of blocks
- total \mathcal{O} 's storage is $\Theta(\sqrt{N})$ except with negligible probability

Formal argument

- cache sizes are not independent so cannot use Chernoff bound

Analysis of CacheShuffleRoot: \mathcal{O} 's storage

Intuition

- \mathcal{O} 's keeps a cache for each of the $(1 + 2\epsilon)\sqrt{N}$ destination groups
- each cache has arrival rate $\approx (1 - 2\epsilon)$
- departure rate 1
- each cache is expected to hold a constant number of blocks
- total \mathcal{O} 's storage is $\Theta(\sqrt{N})$ except with negligible probability

Formal argument

- cache sizes are not independent so cannot use Chernoff bound
- prove negative association and then use Chernoff bound

Keep the stash in \mathcal{O} 's memory and use CacheShuffleRoot

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 and use CacheShuffleRoot

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 and use CacheShuffleRoot

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 and use CacheShuffleRoot

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:

$$4 \cdot 10^3 \approx 4000$$

Keep the stash in \mathcal{O} 's memory and use CacheShuffleRoot

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:

$$4 \cdot 10^3 \approx 4000$$

- Online cost: 2 blocks per read

Keep the stash in \mathcal{O} 's memory and use CacheShuffleRoot

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:

$$4 \cdot 10^3 \approx 4000$$

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

Keep the stash in \mathcal{O} 's memory and use CacheShuffleRoot

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:

$$4 \cdot 10^3 \approx 4000$$

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

Keep the stash in \mathcal{O} 's memory and use CacheShuffleRoot

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:

$$4 \cdot 10^3 \approx 4000$$

- 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 and use CacheShuffleRoot

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:

$$4 \cdot 10^3 \approx 4000$$

- 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

Where are we now?

► Construction -1

Keep It

- \mathcal{M} 's storage: 0
- \mathcal{O} 's storage: N
- bandwidth 0

► Construction 0

Download It

- \mathcal{M} 's storage: N
- \mathcal{O} 's storage: 1
- bandwidth N

► Construction 1

Download Stash

- \mathcal{M} 's storage: $N + \sqrt{N}$
- \mathcal{O} 's storage: 1
- Online Comm. $O(\sqrt{N})$
- Am. Comm.
 $O(\sqrt{N} \cdot \log N)$

► Construction 2

Keep Stash

- \mathcal{M} 's storage: $N + \sqrt{N}$
- \mathcal{O} 's storage: \sqrt{N}
- Online Comm. 1
- Am. Comm.
 $O(\sqrt{N} \cdot \log N)$

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 **⌚** hide access to the **Stash**?

Download and Keep Stash

- Use a cache (the **Stash**) to hide the repetition pattern
- How does **0** hide access to the **Stash**?
 - ▶ Use an ORAM!

Download and Keep Stash

- Use a cache (the **Stash**) to hide the repetition pattern
- How does **0** 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 **🕒** 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 **🕒** 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 **🕒** 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

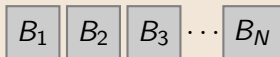
- Use a cache (the **Stash**) to hide the repetition pattern
- How does **🕒** hide access to the **Stash**?
 - ▶ Use an ORAM!
 - ▶ We only have two possible ORAMs:
 - ▶ **Download Stash** uses the **Download It** ORAM to manage the **Stash**
 - ▶ **Keep Stash** uses the **Keep It** ORAM to manage the **Stash**

But now we have more ORAMs!!!

\mathcal{M}

\mathcal{O}

N blocks of data





$$N + \rho N$$

\mathcal{M}

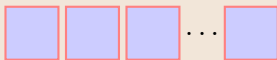
\mathcal{O}

ρN blocks of stash





$$N + \rho N$$



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

\mathcal{M}

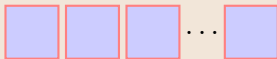
\mathcal{O}

$\rho^2 N$ blocks of stash





$$N + \rho N$$



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



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

\mathcal{M}

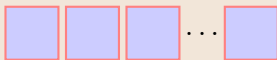
\mathcal{O}

$\rho^3 N$ blocks of stash





$$N + \rho N$$



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



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

\mathcal{M}

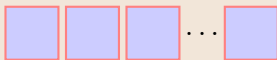
\mathcal{O}

$\rho^3 N$ blocks of stash





$$N + \rho N$$



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



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

\mathcal{M}

\mathcal{O}

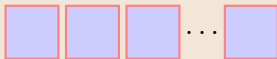
\sqrt{N} blocks of stash

$$\rho = N^{-1/6}$$





$$N + N^{5/6}$$



$$N^{5/6} + N^{2/3}$$



$$N^{2/3} + N^{1/2}$$

\mathcal{M}

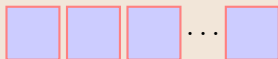
\mathcal{O}

\sqrt{N} blocks of stash





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



Level 0

Position Map

$(\text{lev}_i, \text{pos}_i) \ i = 1, \dots, N$

3-level ORAM: Querying

Querying B_q

- retrieve $(\text{lev}_q, \text{pos}_q)$ from local memory;

3-level ORAM: Querying

Querying B_q

- retrieve $(\text{lev}_q, \text{pos}_q)$ from local memory;
- if $\text{lev}_q \neq 0$

3-level ORAM: Querying

Querying B_q

- retrieve $(\text{lev}_q, \text{pos}_q)$ from local memory;
- if $\text{lev}_q \neq 0$
 - ▶ for all $l \neq \text{lev}_q$, \mathcal{O} asks for the next dummy in level l ;

3-level ORAM: Querying

Querying B_q

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

3-level ORAM: Querying

Querying B_q

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

3-level ORAM: Querying

Querying B_q

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

3-level ORAM: Querying

Querying B_q

- retrieve $(\text{lev}_q, \text{pos}_q)$ from local memory;
- if $\text{lev}_q \neq 0$
 - ▶ for all $l \neq \text{lev}_q$, \mathcal{O} asks for the next **dummy** in level l ;
 - ▶ asks for block in pos_q from level lev_q ;
 - ▶ B_q is then stored in level 0 and $(\text{lev}_q, \text{pos}_q)$ 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 $(\text{lev}_q, \text{pos}_q)$ from local memory;
- if $\text{lev}_q \neq 0$
 - ▶ for all $l \neq \text{lev}_q$, \mathcal{O} asks for the next **dummy** in level l ;
 - ▶ asks for block in pos_q from level lev_q ;
 - ▶ B_q is then stored in level 0 and $(\text{lev}_q, \text{pos}_q)$ 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 ≈ 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
 - ▶ ...

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
 - ▶ ...
 - ▶ $3\sqrt{N}$ 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
 - ▶ ...
 - ▶ $3\sqrt{N}$ 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
 - ▶ ...
 - ▶ $3\sqrt{N}$ 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
 - ▶ ...
 - ▶ $3\sqrt{N}$ blocks
- Amortized bandwidth $0.55 \cdot \log_2 N$
 - ▶ For $N = 10^6$, 21 Blocks
- OnLine bandwidth: 1 Block

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
 - ▶ ...
 - ▶ $3\sqrt{N}$ blocks
- Amortized bandwidth $0.55 \cdot \log_2 N$
 - ▶ For $N = 10^6$, 21 Blocks
- OnLine bandwidth: 1 Block

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
 - ▶ ...
 - ▶ $3\sqrt{N}$ blocks
- Amortized bandwidth $0.55 \cdot \log_2 N$
 - ▶ For $N = 10^6$, 21 Blocks
- OnLine bandwidth: 1 Block

Techniques to reduce bandwidth

- XOR Technique
- Homomorphic Selection
- Compression via Polynomial Interpolation

The XOR technique to reduce bandwidth

ℓ -level ORAM

- when asking for B_q , \mathcal{O} asks one block for each level

The XOR technique to reduce bandwidth

ℓ -level ORAM

- when asking for B_q , \mathcal{O} asks one block for each level
- at most one block is real

The XOR technique to reduce bandwidth

l -level ORAM

- when asking for B_q , \mathcal{O} asks one block for each level
- at most one block is real
- suppose \mathcal{O} can compute locally the $l - 1$ dummy blocks requested

The XOR technique to reduce bandwidth

l -level ORAM

- when asking for B_q , \mathcal{O} asks one block for each level
- at most one block is real
- suppose \mathcal{O} can compute locally the $l - 1$ dummy blocks requested
 - ▶ \mathcal{M} instead of sending all l blocks individually, xors them together and sends the result to \mathcal{O}

The XOR technique to reduce bandwidth

l -level ORAM

- when asking for B_q , \mathcal{O} asks one block for each level
- at most one block is real
- suppose \mathcal{O} can compute locally the $l - 1$ dummy blocks requested
 - ▶ \mathcal{M} instead of sending all l blocks individually, xors them together and sends the result to \mathcal{O}
 - ▶ \mathcal{O} computes the $l - 1$ dummy blocks and xors them with the block received from \mathcal{M}

Assumption:

suppose \mathcal{O} can compute any *dummy* block without interacting with \mathcal{M}

- each block uniquely identified by (l, pos)
- a *dummy* block is an AES-ECB encryption of 0^{len}
- using key $\mathcal{F}(K, (l, pos))$
 - ▶ \mathcal{F} is a pseudorandom function
 - ▶ K is a randomly chosen seed private to \mathcal{O}

Some Theory

A Taxonomy

• OnLine vs OffLine ORAM

- ▶ In an **OnLine** ORAM, all requests come one at the time and must be satisfied before the next one
- ▶ in an **OffLine** ORAM, all requests come together

• BallsAndBins

- ▶ Blocks are atomic and opaque blobs of data

• Passive vs Active \mathcal{M}

- ▶ A **Passive** \mathcal{M} only moves data
- ▶ An **Active** \mathcal{M} can perform computation on data
 - ★ The **XOR technique** requires an **Active** \mathcal{M}

What we know in the asymptotic world

- All the following require $\Omega(\log N)$

What we know in the asymptotic world

- All the following require $\Omega(\log N)$
 - ▶ **BallsAndBins** and **OffLine** with **Passive** \mathcal{M}

What we know in the asymptotic world

- All the following require $\Omega(\log N)$
 - ▶ **BallsAndBins** and **OffLine** with **Passive** \mathcal{M}
 - ▶ **NonBallsAndBins** and **OnLine** with **Passive** \mathcal{M}

What we know in the asymptotic world

- All the following require $\Omega(\log N)$
 - ▶ **BallsAndBins** and **OffLine** with **Passive** \mathcal{M}
 - ▶ **NonBallsAndBins** and **OnLine** with **Passive** \mathcal{M}
- There are $\tilde{O}(\log N)$ **OnLine** ORAM in the Balls and Bins model with **Passive** \mathcal{M}

What we know in the asymptotic world

- All the following require $\Omega(\log N)$
 - ▶ **BallsAndBins** and **OffLine** with **Passive** \mathcal{M}
 - ▶ **NonBallsAndBins** and **OnLine** with **Passive** \mathcal{M}
- There are $\tilde{O}(\log N)$ **OnLine** ORAM in the Balls and Bins model with **Passive** \mathcal{M}
- There are $o(\log N)$ **OnLine** ORAM with **Active** \mathcal{M}

What we know in the asymptotic world

- All the following require $\Omega(\log N)$
 - ▶ **BallsAndBins** and **OffLine** with **Passive** \mathcal{M}
 - ▶ **NonBallsAndBins** and **OnLine** with **Passive** \mathcal{M}
- There are $\tilde{O}(\log N)$ **OnLine** ORAM in the Balls and Bins model with **Passive** \mathcal{M}
- There are $o(\log N)$ **OnLine** ORAM with **Active** \mathcal{M}
- Proving lower bound for **Non-BallsAndBins** and **OffLine** with **Passive** \mathcal{M} would give a superlinear lower bound for sorting circuits.

(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 $\mathbb{B} = B_1, \dots, B_n$ and $\mathbb{C} = C_1, \dots, C_n$
 - ▶ For any two access sequences i_1, \dots, i_l and j_1, \dots, j_l **that differ in one position**
 - ★ performing access i_1, \dots, i_l on $\mathbb{B} = B_1, \dots, B_n$;
 - ★ performing access j_1, \dots, j_l on $\mathbb{C} = C_1, \dots, C_n$

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

(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 $\mathbb{B} = B_1, \dots, B_n$ and $\mathbb{C} = C_1, \dots, C_n$
 - ▶ For any two access sequences i_1, \dots, i_l and j_1, \dots, j_l that differ in one position
 - ★ performing access i_1, \dots, i_l on $\mathbb{B} = B_1, \dots, B_n$;
 - ★ performing access j_1, \dots, j_l on $\mathbb{C} = C_1, \dots, C_n$

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

For every predicate A

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

Differential Privacy

- it protects only **individual** accesses

Differential Privacy

- it protects only **individual** accesses
- if the probability of an access is larger than $1/e^\epsilon$ no guarantee is given

Differential Privacy

- it protects only **individual** accesses
- if the probability of an access is larger than $1/e^\epsilon$ no guarantee is given
- it protects **unusual accesses**

Differential Privacy

- it protects only **individual** accesses
- if the probability of an access is larger than $1/e^\epsilon$ no guarantee is given
- it protects **unusual accesses**
- how unusual?

Differential Privacy

- it protects only **individual** accesses
- if the probability of an access is larger than $1/e^\epsilon$ no guarantee is given
- it protects **unusual accesses**
- how unusual?
 - ▶ it depends on ϵ

Differential Privacy

- it protects only **individual** accesses
- if the probability of an access is larger than $1/e^\epsilon$ no guarantee is given
- it protects **unusual accesses**
- how unusual?
 - ▶ it depends on ϵ
- no protection is offered for **high-probability** accesses

Differential Privacy

- it protects only **individual** accesses
- if the probability of an access is larger than $1/e^\epsilon$ no guarantee is given
- it protects **unusual accesses**
- how unusual?
 - ▶ it depends on ϵ
- no protection is offered for **high-probability** accesses
 - ▶ **nothing is lost: my daily routine is public**

Differential Privacy

- it protects only **individual** accesses
- if the probability of an access is larger than $1/e^\epsilon$ no guarantee is given
- it protects **unusual accesses**
- how unusual?
 - ▶ it depends on ϵ
- no protection is offered for **high-probability** accesses
 - ▶ **nothing is lost: my daily routine is public**
- I really want to protect **unusual accesses** to documents that might reveal something

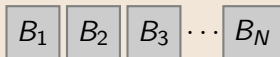
Differential Privacy

- it protects only **individual** accesses
- if the probability of an access is larger than $1/e^\epsilon$ no guarantee is given
- it protects **unusual accesses**
- how unusual?
 - ▶ it depends on ϵ
- no protection is offered for **high-probability** accesses
 - ▶ **nothing is lost: my daily routine is public**
- I really want to protect **unusual accesses** to documents that might reveal something
 - ▶ I am checking my medical records from some time ago...

\mathcal{M}

\mathcal{O}

N blocks of data



B_1 B_2 B_3 \dots B_N

N encrypted blocks

probability p

\mathcal{M}

 \mathcal{O}

B_1 B_2 B_3 \dots B_N

N encrypted blocks

probability p

\mathcal{M}

\mathcal{O}

Stash

B_7 B_9 \dots B_{N-2}

pN expected blocks

B_1 B_2 B_3 \dots B_N

N encrypted blocks

probability p

\mathcal{M}

\mathcal{O}

Stash

B_7 B_9 \dots B_{N-2}

pN expected blocks

B_1 B_2 B_3 \dots B_N

N encrypted blocks

probability p

\mathcal{M}

\mathcal{O}

Stash

B_7 B_9 \dots B_{N-2}

pN expected blocks

B_1 B_2 B_3 \dots B_N

N encrypted blocks

probability p

\mathcal{M}

\mathcal{O}

Stash

B_7 B_9 \dots B_{N-2}

pN expected blocks

Querying for B_i

Download Phase

- if B_i is found in the stash:

Overwrite Phase

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it

Overwrite Phase

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash

Overwrite Phase

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it

Overwrite Phase

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:

Overwrite Phase

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:
 - ▶ ask \mathcal{M} for B_i

Overwrite Phase

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:
 - ▶ ask \mathcal{M} for B_i

Overwrite Phase

- Toss a coin with probability $(p, 1 - p)$

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:
 - ▶ ask \mathcal{M} for B_i

Overwrite Phase

- Toss a coin with probability $(p, 1 - p)$
- if head

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:
 - ▶ ask \mathcal{M} for B_i

Overwrite Phase

- Toss a coin with probability $(p, 1 - p)$
- if head
 - ▶ add B_i to stash

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:
 - ▶ ask \mathcal{M} for B_i

Overwrite Phase

- Toss a coin with probability $(p, 1 - p)$
- if head
 - ▶ add B_i to stash
 - ▶ ask \mathcal{M} for a randomly selected block, re-encrypt it and upload to the same location

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:
 - ▶ ask \mathcal{M} for B_i

Overwrite Phase

- Toss a coin with probability $(p, 1 - p)$
- if head
 - ▶ add B_i to stash
 - ▶ ask \mathcal{M} for a randomly selected block, re-encrypt it and upload to the same location
- if tail

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:
 - ▶ ask \mathcal{M} for B_i

Overwrite Phase

- Toss a coin with probability $(p, 1 - p)$
- if head
 - ▶ add B_i to stash
 - ▶ ask \mathcal{M} for a randomly selected block, re-encrypt it and upload to the same location
- if tail
 - ▶ download B_i from \mathcal{M}

Querying for B_i

Download Phase

- if B_i is found in the stash:
 - ▶ return it
 - ▶ remove it from the stash
 - ▶ ask \mathcal{M} for a random block and then discard it
- if B_i is not found in the stash:
 - ▶ ask \mathcal{M} for B_i

Overwrite Phase

- Toss a coin with probability $(p, 1 - p)$
- if head
 - ▶ add B_i to stash
 - ▶ ask \mathcal{M} for a randomly selected block, re-encrypt it and upload to the same location
- if tail
 - ▶ download B_i from \mathcal{M}
 - ▶ decrypt and re-encrypt and upload it to the same location

Analysis

- Two blocks of communication

Analysis

- Two blocks of communication
- $p = \omega(\log n/n)$ with $\omega(\log n)$ client memory

Analysis

- Two blocks of communication
- $p = \omega(\log n/n)$ with $\omega(\log n)$ client memory
- $\epsilon = \Theta(\log n)$

Analysis

- Two blocks of communication
- $p = \omega(\log n/n)$ with $\omega(\log n)$ client memory
- $\epsilon = \Theta(\log n)$

Analysis

- Two blocks of communication
- $p = \omega(\log n/n)$ with $\omega(\log n)$ client memory
- $\epsilon = \Theta(\log n)$

Theorem

For any $\epsilon \geq 0$, any DP-RAM with error probability $\alpha \geq 0$ in the BallsAndBins model and a client that stores at most c blocks must operate on

$$\Omega \left(\log_c \left(\frac{(1 - \alpha) \cdot n}{e^\epsilon} \right) \right)$$

records.

In the non-BallsAndBins the bound is

$$\Omega \left(\frac{b}{w} \log \frac{nb}{c} \right)$$

for any constant ϵ , $\delta \leq 1/3$ and error probability $1/3$.

Conclusions

- **Leaking the access pattern can be dangerous**

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data
- **Differential Privacy**: a weaker security notion

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data
- **Differential Privacy**: a weaker security notion
 - ▶ very efficient

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data
- **Differential Privacy**: a weaker security notion
 - ▶ very efficient
 - ▶ suitable only for specific applications

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data
- **Differential Privacy**: a weaker security notion
 - ▶ very efficient
 - ▶ suitable only for specific applications
- Other security notions?

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data
- **Differential Privacy**: a weaker security notion
 - ▶ very efficient
 - ▶ suitable only for specific applications
- Other security notions?
- Better analysis under reasonable assumptions for access distributions?
Zipf?

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data
- **Differential Privacy**: a weaker security notion
 - ▶ very efficient
 - ▶ suitable only for specific applications
- Other security notions?
- Better analysis under reasonable assumptions for access distributions?
Zipf?

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data
- **Differential Privacy**: a weaker security notion
 - ▶ very efficient
 - ▶ suitable only for specific applications
- Other security notions?
- Better analysis under reasonable assumptions for access distributions?
Zipf?

Conclusions

- **Leaking the access pattern can be dangerous**
- **Obliviousness**: a strong security notion
 - ▶ requires $> 20\times$ overhead for 4 Giga of data
- **Differential Privacy**: a weaker security notion
 - ▶ very efficient
 - ▶ suitable only for specific applications
- Other security notions?
- Better analysis under reasonable assumptions for access distributions?
Zipf?

<https://github.com/giuper/talks/nonTechnical/oramTutorial.pdf>

The original papers



O. Goldreich.

Towards a Theory of Software Protection and Simulation by Oblivious RAMs.

In *STOC*, 1987.



R. Ostrovsky.

Efficient computation on oblivious RAMs.

In *STOC*, pages 514–523, 1990.



O. Goldreich and R. Ostrovsky.

Software Protection and Simulation on Oblivious RAMs.

J. ACM, 43(3), 1996.

Asymptotics



G. Asharov, I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, and E. Shi.

OptORAMa: Optimal oblivious RAM.

Cryptology ePrint Archive, Report 2018/892.



S. Patel, G. Persiano, M. Raykova, and K. Yeo.

PanORAMa: Oblivious RAM with logarithmic overhead.

Cryptology ePrint Archive, Report 2018/373, 2018.

<https://eprint.iacr.org/2018/373>.

FOCS '18.

Constant client memory

Efficient constructions for large blocks



E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas.

Path ORAM: An Extremely Simple Oblivious RAM Protocol.
In *CCS '13*, pages 299–310, 2013.



S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs.

Onion ORAM: A constant bandwidth blowup oblivious RAM.
In *TCC*, pages 145–174, 2015.



T. Moataz, T. Mayberry, and E.-O. Blass.

Constant communication ORAM with small blocksize.
In *CCS*, pages 862–873, 2015.

polylog(n)-bit blocks

Efficient constructions for large client memory



L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas.

Constants count: Practical improvements to oblivious RAM.
In *USENIX Security*, pages 415–430, 2015.



E. Stefanov, E. Shi, and D. X. Song.

Towards practical oblivious RAM.
In *NDSS 2012*, 2012.





S. Patel, G. Persiano, and K. Yeo.


Recursive ORAMs with practical constructions.
Cryptology ePrint Archive, Report 2017/964, 2017.


$O(\sqrt{n})$ **client memory**

Lower bounds

 E. Boyle and M. Naor.
Is There an Oblivious RAM Lower Bound?
In *ITCS*, pages 357–368, 2016.

 K. G. Larsen and J. B. Nielsen.
Yes, there is an oblivious RAM lower bound!
Cryptology ePrint Archive, Report 2018/423, 2018.
CRYPTO '18

 M. Weiss and D. Wichs.
Is there an Oblivious RAM lower bound for online reads?
Cryptology ePrint Archive, Report 2018/619, 2018.
TCC '18

 G. Persiano and K. Yeo.
Lower bounds for differentially private RAMs.
Cryptology ePrint Archive, Report 2018/1051, 2018.
Eurocrypt '19

Oblivious Sorting and Shuffling



M. T. Goodrich.

Zig-Zag Sort: A Simple Deterministic Data-oblivious Sorting Algorithm Running in $O(N \log N)$ Time.

In *STOC*, pages 684–693, 2014.



O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal.

The melbourne shuffle: Improving oblivious storage in the cloud.

In *ICALP '14*, pages 556–567. Springer, 2014.



S. Patel, G. Persiano, and K. Yeo.

CacheShuffle: A Family of Oblivious Shuffles.

In *ICALP*, pages 161:1–161:13, 2018.