Revisiting and Improving Algorithms for the 3XOR Problem

Claire Delaplace[1,2]

joint work with Charles Bouillaguet[1] Pierre-Alain Fouque[2]

[1] University of Lille, CRIStAL, France

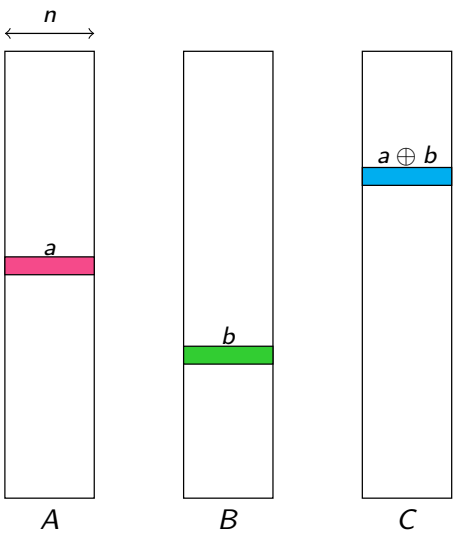
[2] University of Rennes 1, IRISA, France

Caen, May 2018

3XOR Problem

Problem

Given three lists A, B, and C of uniformly random elements of $\{0,1\}^n$, find $(a,b,c) \in A \times B \times C$, such that $a \oplus b \oplus c = 0$.



- Difficult case of Generalised Birthday Problem
- Application in cryptanalysis of some authenticated encryption scheme
- Lists formed by querying oracles $\Rightarrow$ can be as big as we want
- $|A| \cdot |B| \cdot |C| \geq 2^n \Rightarrow$ solution w.h.p.

Our Contributions

- Discuss practical issues arising from the computation of a 3XOR
- Propose a new efficient algorithm
- Present an adaptation of an [BDP05] to the 3XOR problem
- Compute a 96-bit 3XOR of SHA-256

A Naive Quadratic Algorithm

Problem: find $(a, b, c) \in A \times B \times C$ s.t. $a \oplus b \oplus c = 0$

Idea

- Create all $v = a \oplus b$
- Check if v is in C

A Naive Quadratic Algorithm

Problem: find $(a, b, c) \in A \times B \times C$ s.t. $a \oplus b \oplus c = 0$

Idea

- Create all $v = a \oplus b$
- Check if v is in C

- Time complexity: $\mathcal{O}(|A| \cdot |B| + |C|)$
- Space: $\mathcal{O}(|A| + |B| + |C|)$
- $|A| = |B| = |C| = 2^{n/3} \Rightarrow$ Time: $\mathcal{O}\left(2^{2n/3}\right)$, Space: $\mathcal{O}\left(2^{n/3}\right)$
- $|A| = |B| = 2^{n/4}$, $|C| = 2^{n/2} \Rightarrow$ Time: $\mathcal{O}\left(2^{n/2}\right)$, Space: $\mathcal{O}\left(2^{n/2}\right)$

A Naive Quadratic Algorithm

Problem: find $(a, b, c) \in A \times B \times C$ s.t. $a \oplus b \oplus c = 0$

Idea

- Create all $v = a \oplus b$
- Check if v is in C

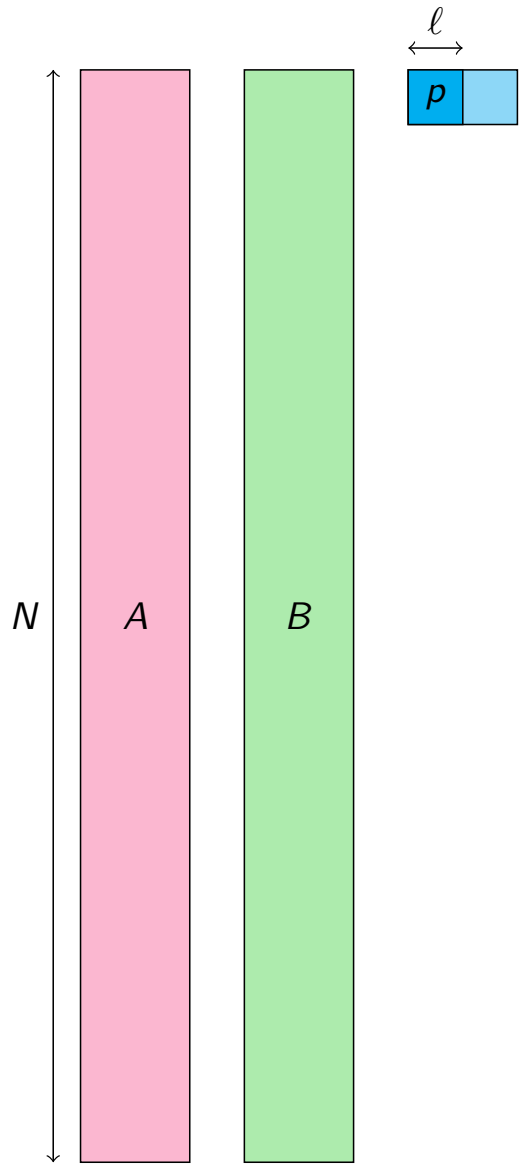
- Time complexity: $\mathcal{O}(|A| \cdot |B| + |C|)$
- Space: $\mathcal{O}(|A| + |B| + |C|)$
- $|A| = |B| = |C| = 2^{n/3} \Rightarrow$ Time: $\mathcal{O}\left(2^{2n/3}\right)$, Space: $\mathcal{O}\left(2^{n/3}\right)$
- $|A| = |B| = 2^{n/4}$, $|C| = 2^{n/2} \Rightarrow$ Time: $\mathcal{O}\left(2^{n/2}\right)$, Space: $\mathcal{O}\left(2^{n/2}\right)$

Tradeoffs are to be made (e.g. [Wagner02], [Bernstein07]).

About Wagner's Algorithm

- Designed for the kXOR problem
- Works best when k is a power of 2
- Not better than quadratic Algorithm in case $k = 3$
- Has known improvement since ([Joux09, NS14])

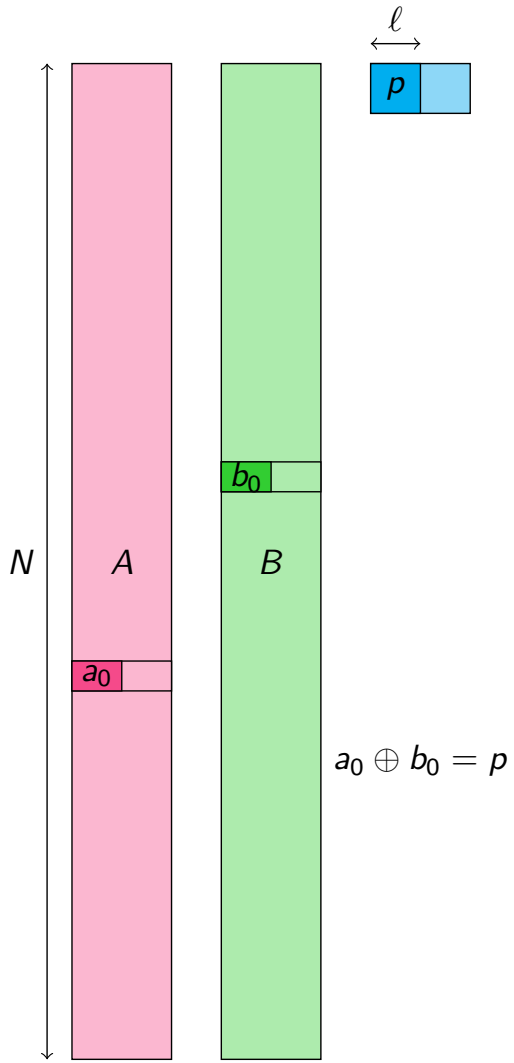
Wagner's idea



Description

- Number of queries: increased up to $N \simeq 2^{n/2}$
- Elements of C start by p

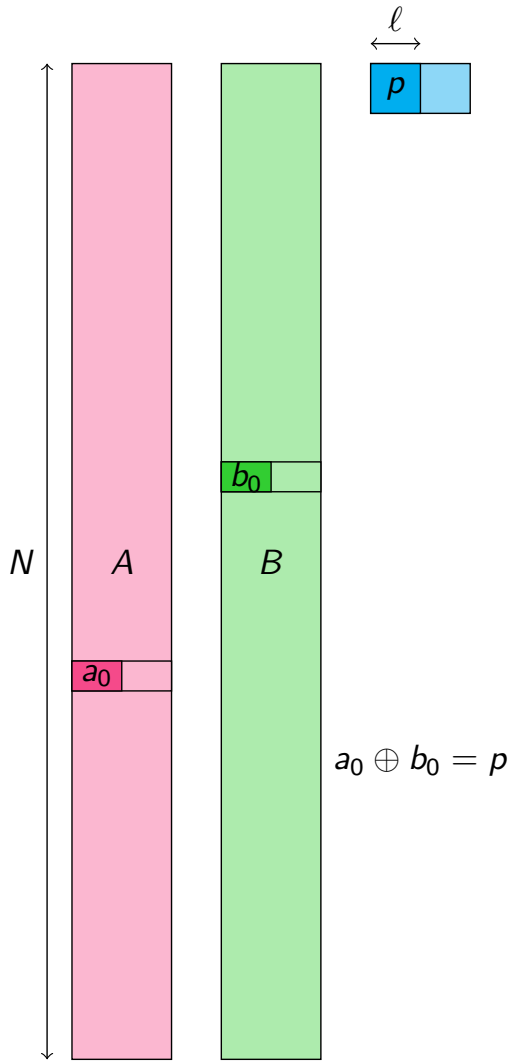
Wagner's idea



Description

- Number of queries: increased up to $N \simeq 2^{n/2}$
- Elements of C start by p
- Sort A and B on the first ℓ bits
- For all a, b s.t. $a \oplus b = (p|*)$
  - search $a \oplus b$ in C

Wagner's idea



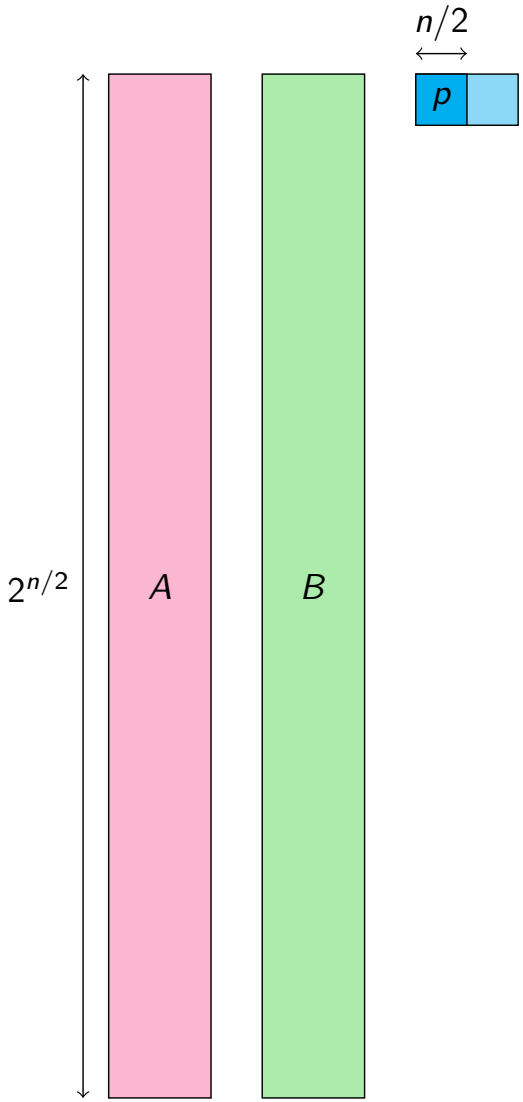
Description

- Number of queries: increased up to $N \simeq 2^{n/2}$
- Elements of C start by p
- Sort A and B on the first ℓ bits
- For all a, b s.t. $a \oplus b = (p|*)$
  - search $a \oplus b$ in C

Complexity

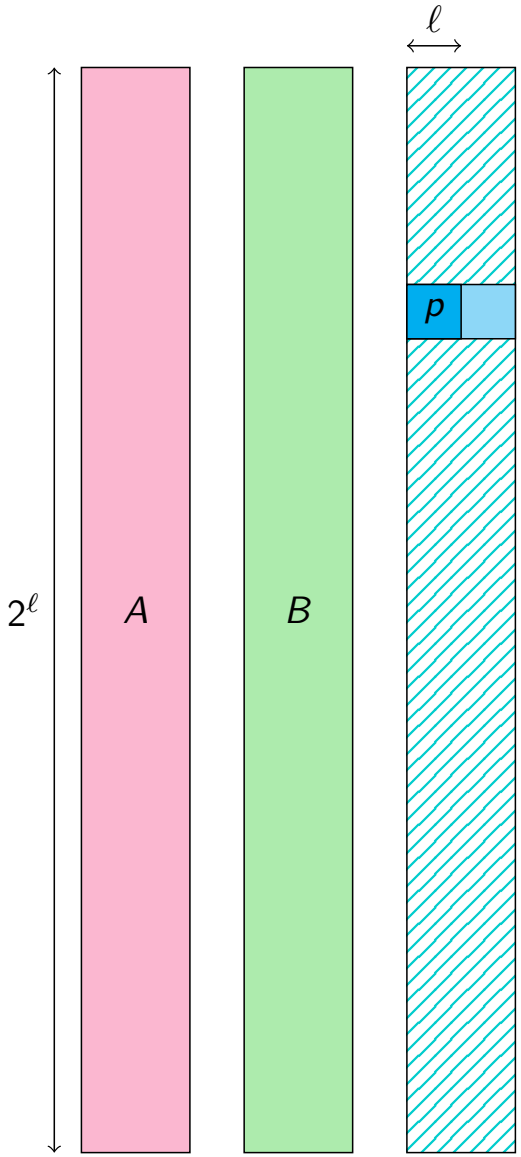
$\ell \simeq \log(N)$
Time and space: $\mathcal{O}(N)$

Wagner [Wagner02]



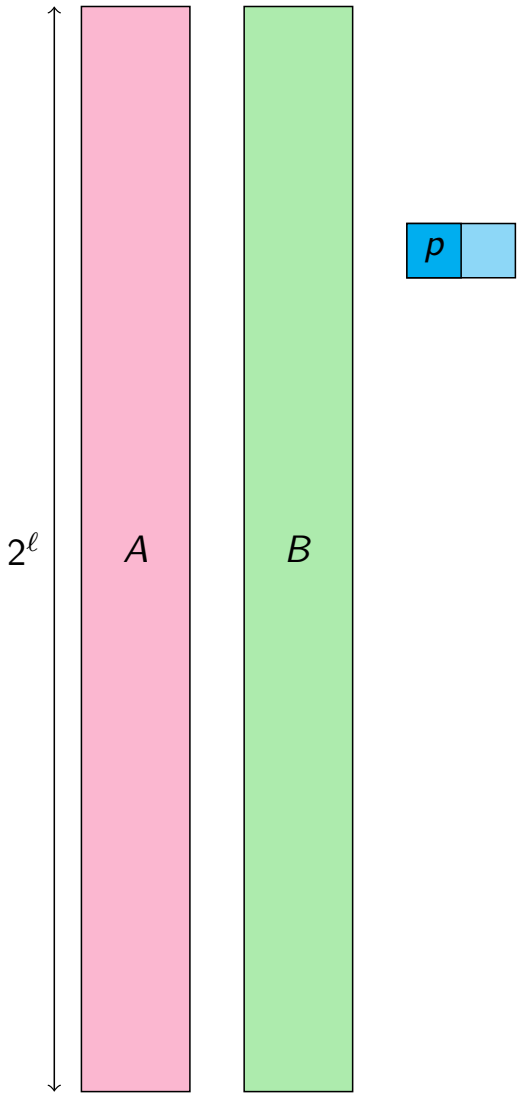
- $N = 2^{n/2}$, $\ell = n/2$
- p: arbitrary chosen
- Expected $|C|$: 1
- Time/Space: $\mathcal{O}\left(2^{n/2}\right)$.
- No improvement compared to Quad

Nicolić and Sasaki [NS14]



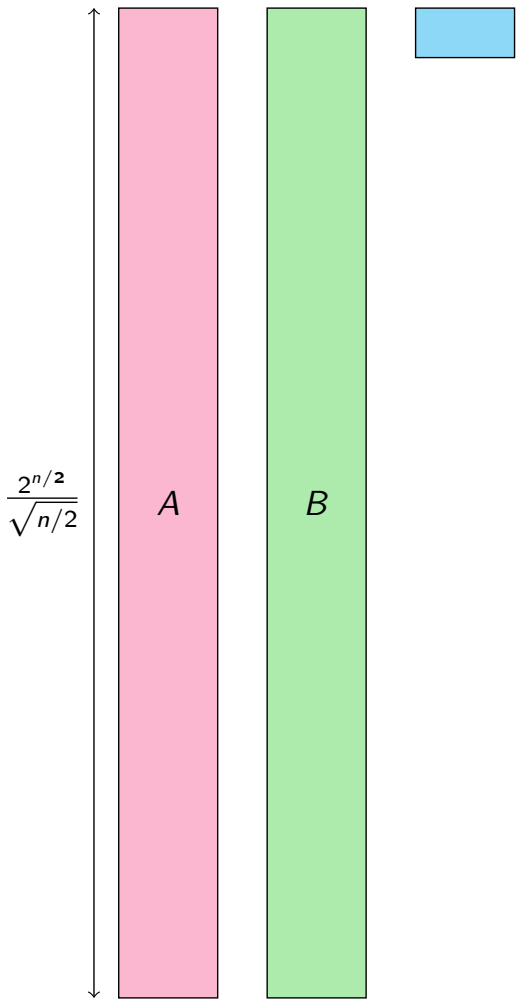
- $N \simeq 2^{n/2}/\sqrt{(n/2)/\ln(n/2)}$, $\ell = \log(N)$
- Begin with $|C| = 2^\ell$
- p: Most frequent prefix in whole C

Nicolić and Sasaki [NS14]



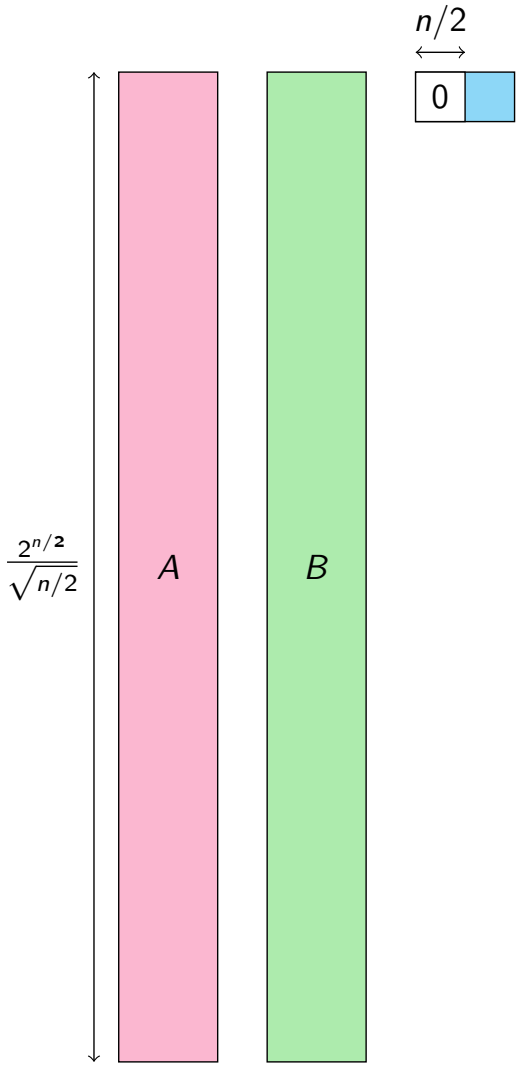
- $N \simeq 2^{n/2}/\sqrt{(n/2)/\ln(n/2)}$, $\ell = \log(N)$
- Begin with $|C| = 2^\ell$
- p: Most frequent prefix in whole C
- Discard other elements of C
- Expected $|C|$: $\ell/\ln(\ell)$
- Time/Space: $\mathcal{O}\left(2^{n/2}/\sqrt{n/\ln(n)}\right)$
- $\simeq \sqrt{\frac{n}{\ln(n)}}$ speedup compared to Wagner

Joux [Joux09]



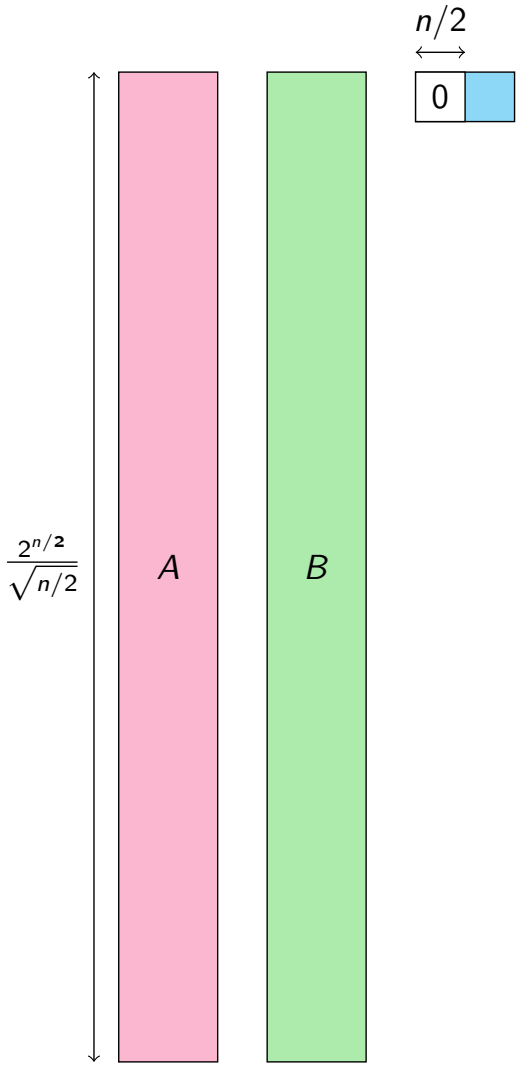
- $N = 2^{n/2}/\sqrt{n/2}$, $\ell = n/2$
- $|C| = n/2$

Joux [Joux09]



- $N = 2^{n/2}/\sqrt{n/2}$, $\ell = n/2$
- $|C| = n/2$
- Perform a basis change that forces p to 0
- Time/Space: $\mathcal{O}\left(2^{n/2}/\sqrt{n/2}\right)$.
- $\simeq \sqrt{n}$ speedup compared to Wagner

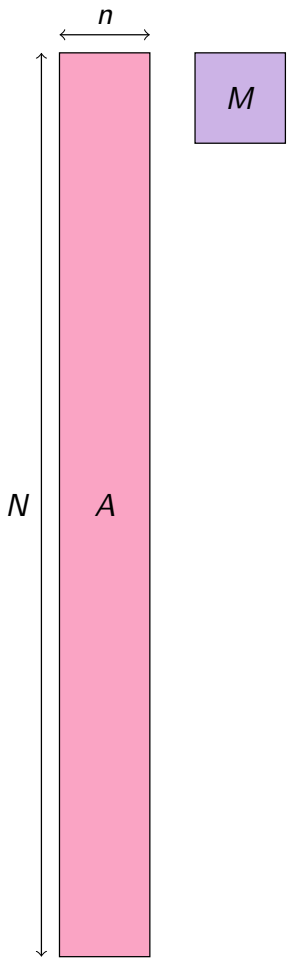
Joux [Joux09]



- $N = 2^{n/2}/\sqrt{n/2}$, $\ell = n/2$
- $|C| = n/2$
- Perform a basis change that forces p to 0
- Time/Space: $\mathcal{O}\left(2^{n/2}/\sqrt{n/2}\right)$.
- $\simeq \sqrt{n}$ speedup compared to Wagner

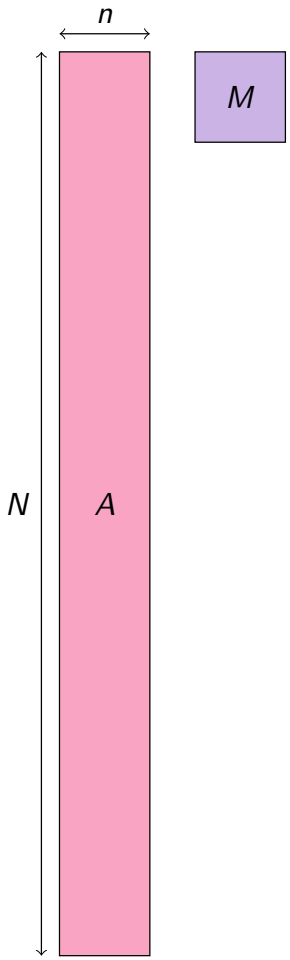
What about the cost of basis change?

Computing the Basis Change



Compute the product of A and M in time $\mathcal{O}(N)$: really?

Computing the Basis Change

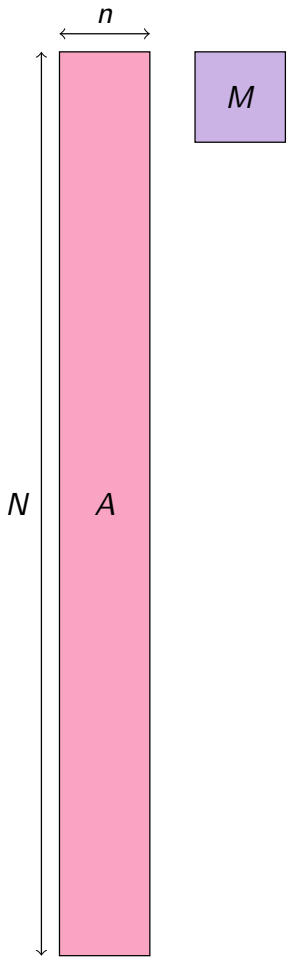


Compute the product of A and M in time $\mathcal{O}(N)$: really?
Yes, under some conditions...

Hypothesis

- A and M are matrices over $\mathbb{F}_2$
- $N \gg n$

Computing the Basis Change



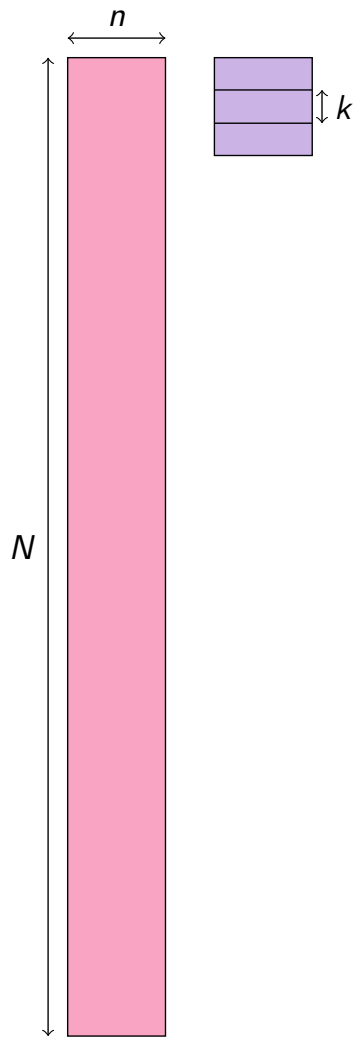
Compute the product of A and M in time $\mathcal{O}(N)$: really?
Yes, under some conditions...

Hypothesis

- A and M are matrices over $\mathbb{F}_2$
- $N \gg n$

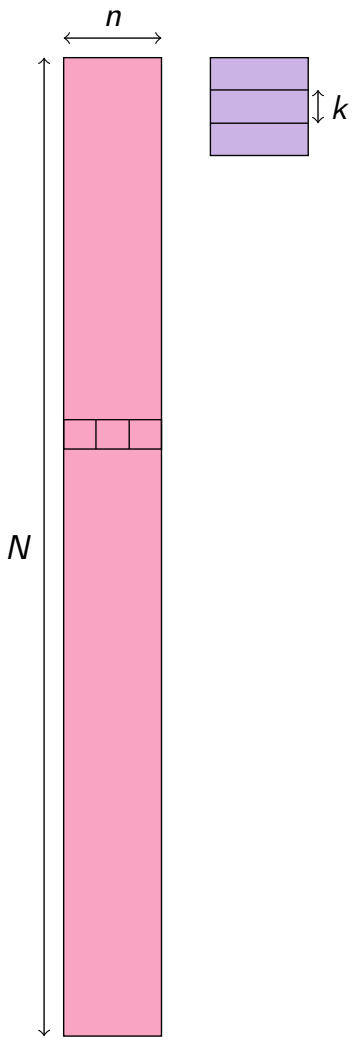
We use a 4-Russian trick:

The Trick



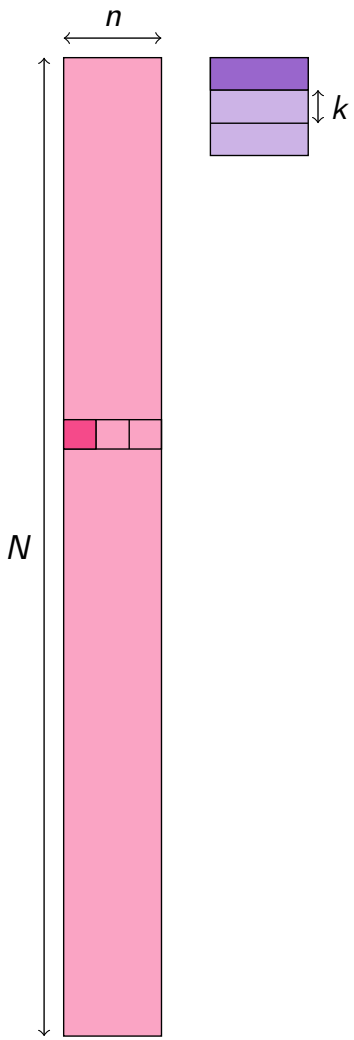
- Divide M into slices of $k = (\log N)/(1+\epsilon)$ rows
- Store all linear combinations of each slice in tables $T_1, \ldots, T_{n/k}$ (required memory: $< N$)

The Trick



- Divide M into slices of $k = (\log N)/(1+\epsilon)$ rows
- Store all linear combinations of each slice in tables $T_1, \ldots, T_{n/k}$ (required memory: $< N$)
- For each rows a of A:
  - Divide a into slices $a_1, \ldots, a_{n/k}$ of k columns

The Trick



- Divide M into slices of $k = (\log N)/(1+\epsilon)$ rows
- Store all linear combinations of each slice in tables $T_1, \ldots, T_{n/k}$ (required memory: $< N$)
- For each rows a of A:
  - Divide a into slices $a_1, \ldots, a_{n/k}$ of k columns
  - Get the corresponding linear combinaisons $T_1[a_1], \ldots, T_{n/k}[a_{n/k}]$
  - Xor them

Discussion

Joux's Algorithm best time complexity but...

Discussion

Joux's Algorithm best time complexity but...

96-bit 3XOR

- Joux Algorithm: about 2^{48} operations
- But about 1 PB of data $\Longrightarrow$ Impractical

Discussion

Joux's Algorithm best time complexity but...

96-bit 3XOR

- Joux Algorithm: about 2^{48} operations
- But about 1 PB of data $\Longrightarrow$ Impractical
- Quad algorithm: with $|A| = |B| = |C| = 2^{n/3}$: about 2^{64} operations
- But only 206 GB of data $\Longrightarrow$ Practical

Discussion

Joux's Algorithm best time complexity but...

96-bit 3XOR

- Joux Algorithm: about 2^{48} operations
- But about 1 PB of data $\Longrightarrow$ Impractical
- Quad algorithm: with $|A| = |B| = |C| = 2^{n/3}$: about 2^{64} operations
- But only 206 GB of data $\Longrightarrow$ Practical

$\Rightarrow$ Keep the lists small!

The Clamping Trick [Berstein07]

- **Idea:** Increase the number of queries to reduce the storage

The Clamping Trick [Berstein07]

- **Idea:** Increase the number of queries to reduce the storage
- 2^k queries, $k \geq n/3$
- ℓ s.t. $(n-\ell)/3 = k - \ell$
- Discard vectors that do not start with ℓ zeroes

The Clamping Trick [Berstein07]

- **Idea:** Increase the number of queries to reduce the storage
- 2^k queries, $k \geq n/3$
- ℓ s.t. $(n-\ell)/3 = k - \ell$
- Discard vectors that do not start with ℓ zeroes
- Let $n' = n - \ell$
- $\Rightarrow$ 3 lists A, B, C of $2^{k-\ell} = 2^{n'/3}$ of n'-bits vectors
- Solve the 3XOR problem over A, B, C with $|A| \cdot |B| \cdot |C| = 2^{n'}$

The Clamping Trick [Berstein07]

- **Idea:** Increase the number of queries to reduce the storage
- 2^k queries, $k \geq n/3$
- ℓ s.t. $(n-\ell)/3 = k - \ell$
- Discard vectors that do not start with ℓ zeroes
- Let $n' = n - \ell$
- $\Rightarrow$ 3 lists A, B, C of $2^{k-\ell} = 2^{n'/3}$ of n'-bits vectors
- Solve the 3XOR problem over A, B, C with $|A| \cdot |B| \cdot |C| = 2^{n'}$

$2^{n/2}$ Queries

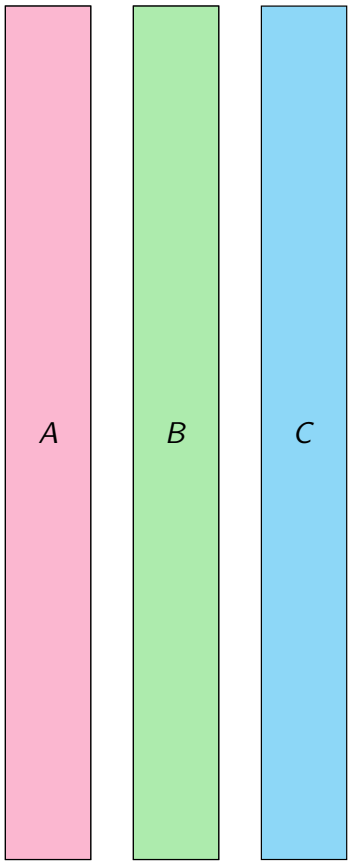
- $\ell = n/4$, $n' = 3n/4$
- **Stored data:** $\mathcal{O}\left(2^{n/4}\right)$ words
- **Time Complexity:** $\mathcal{O}\left(2^{n/2}\right)$ with Quadratic Algorithm

1. Background

2. Our New Algorithm

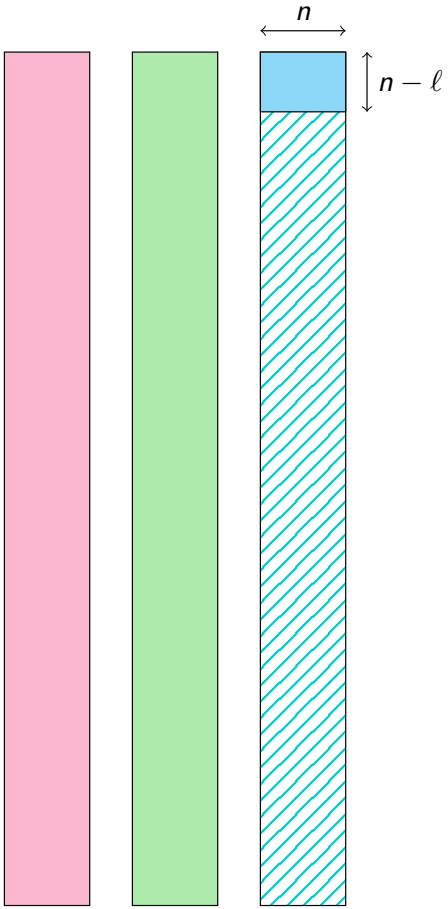
3. Adaptation of BDP Algorithm for the 3SUM problem

Our Work: A Generalization of Joux Algorithm



Generalization to any size of input lists

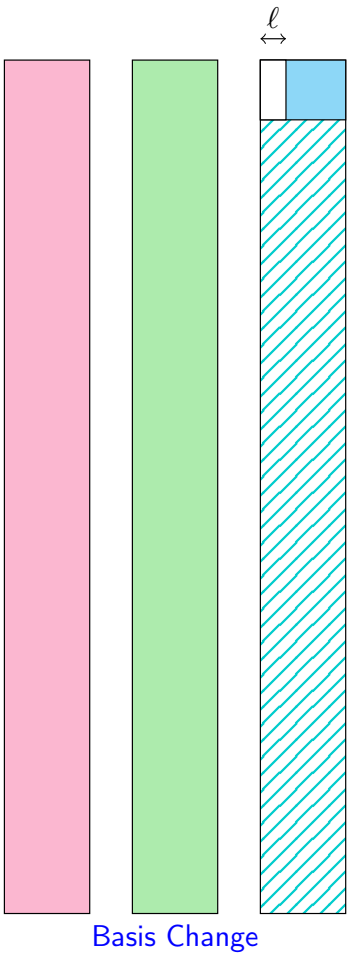
Our Work: A Generalization of Joux Algorithm



Generalization to any size of input lists

- Pick $n - \ell$ arbitrary entries in C (the first ones)

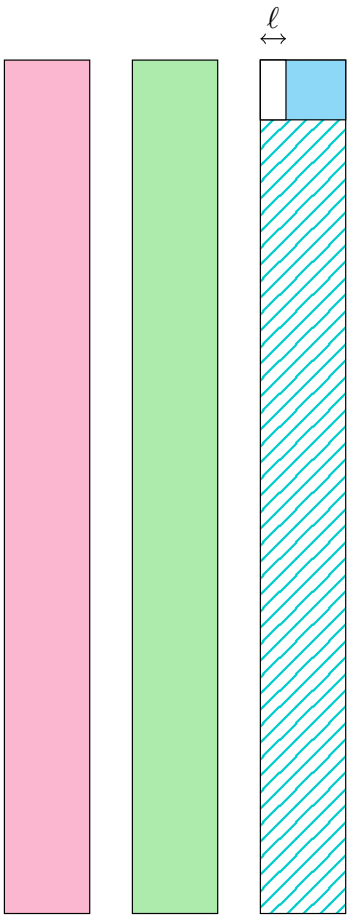
Our Work: A Generalization of Joux Algorithm



Generalization to any size of input lists

- Pick $n - \ell$ arbitrary entries in C (the first ones)
- Apply Joux's Algorithm

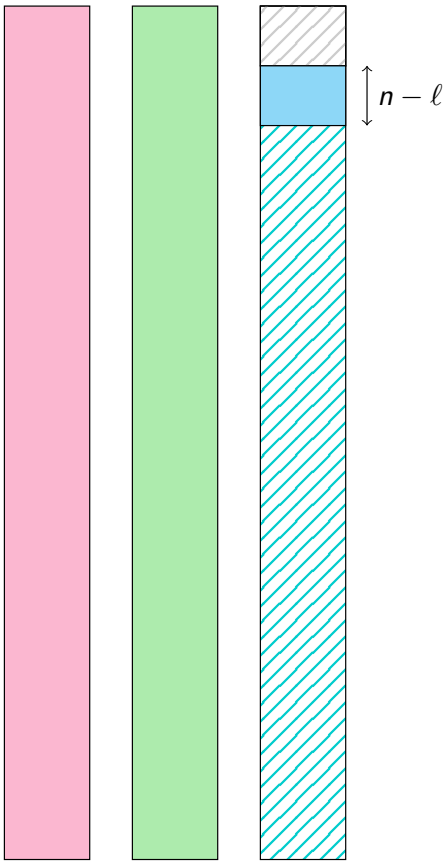
Our Work: A Generalization of Joux Algorithm



Generalization to any size of input lists

- Pick $n - \ell$ arbitrary entries in C (the first ones)
- Apply Joux's Algorithm $(\mathcal{O}(|A| + |B|))$

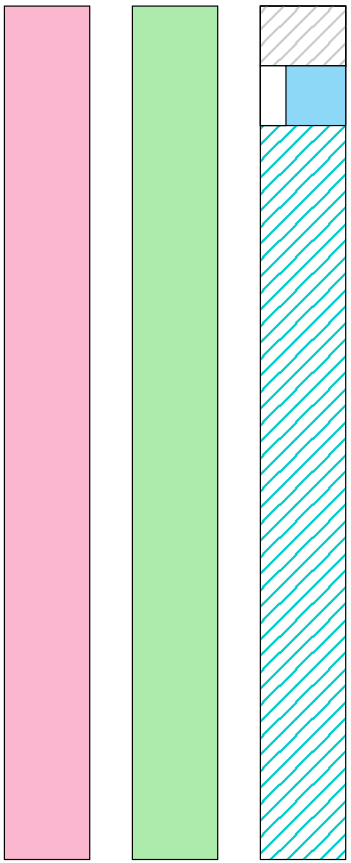
Our Work: A Generalization of Joux Algorithm



Generalization to any size of input lists

- Pick $n - \ell$ arbitrary entries in C (the first ones)
- Apply Joux's Algorithm $(\mathcal{O}(|A| + |B|))$
- Re-iterate with $n - \ell$ other rows...

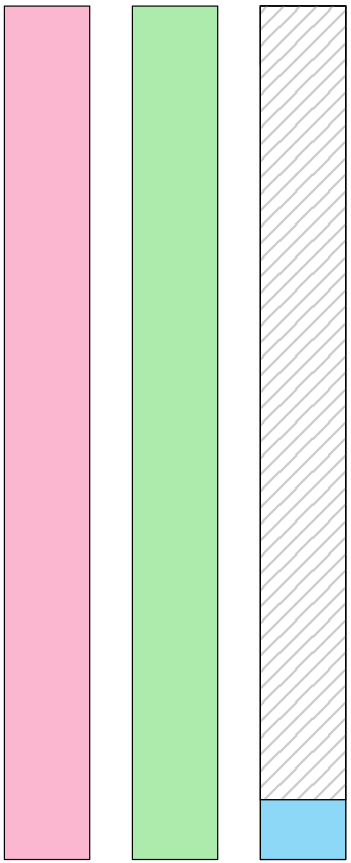
Our Work: A Generalization of Joux Algorithm



Generalization to any size of input lists

- Pick $n - \ell$ arbitrary entries in C (the first ones)
- Apply Joux's Algorithm $(\mathcal{O}(|A| + |B|))$
- Re-iterate with $n - \ell$ other rows...

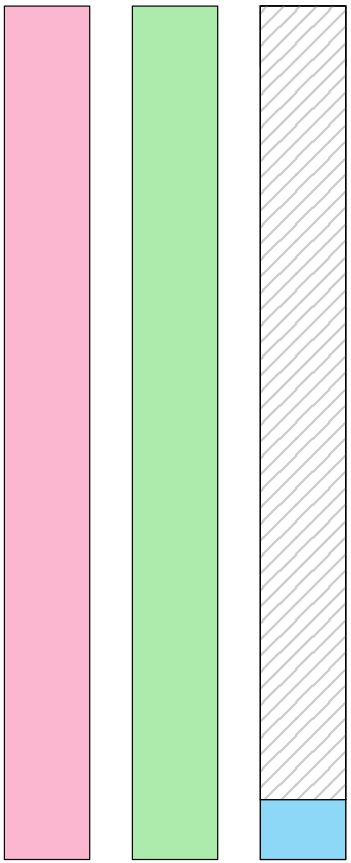
Our Work: A Generalization of Joux Algorithm



Generalization to any size of input lists

- Pick $n - \ell$ arbitrary entries in C (the first ones)
- Apply Joux's Algorithm $(\mathcal{O}(|A| + |B|))$
- Re-iterate with $n - \ell$ other rows...
- ... until all C has been watched $\left(\simeq \frac{|C|}{n-\ell} \text{ iterations}\right)$

Our Work: A Generalization of Joux Algorithm

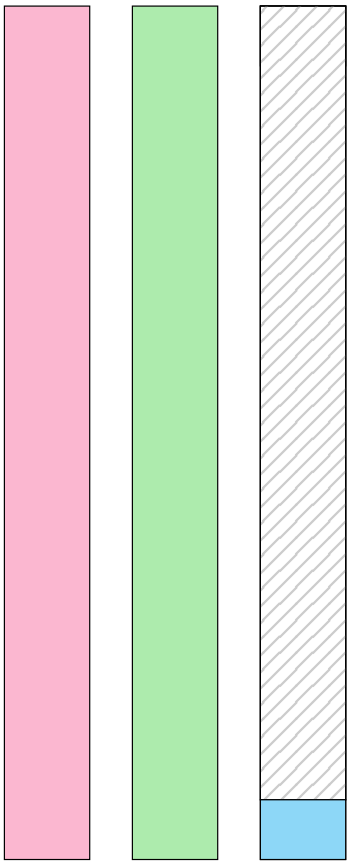


Generalization to any size of input lists

- Pick $n - \ell$ arbitrary entries in C (the first ones)
- Apply Joux's Algorithm $(\mathcal{O}(|A| + |B|))$
- Re-iterate with $n - \ell$ other rows...
- ... until all C has been watched $\left(\simeq \frac{|C|}{n-\ell} \text{ iterations}\right)$

$$\ell = \log(\min(|A|, |B|)), \text{ Time: } \mathcal{O}\left((|A| + |B|) \cdot \frac{|C|}{n}\right)$$

Our Work: A Generalization of Joux Algorithm



Generalization to any size of input lists

- Pick $n - \ell$ arbitrary entries in C (the first ones)
- Apply Joux's Algorithm $(\mathcal{O}(|A| + |B|))$
- Re-iterate with $n - \ell$ other rows...
- ... until all C has been watched $\left(\simeq \frac{|C|}{n-\ell}\right.$ **iterations**$\left.\right)$

$$|A| = |B| = |C| = 2^{n/3}; \ell = n/3, \text{ Time: } \mathcal{O}\left(\frac{2^{2n/3}}{n}\right)$$

Using the Clamping Trick

$2^{n/2-\epsilon}$ **Queries**

- Clamping performed over $(n/2 - 3 \cdot \epsilon)/2$ bits
- Size of lists: $\simeq 2^{n/4+\epsilon/2}$
- Time Complexity: $\mathcal{O}\left(2^{n/2+\epsilon}/n\right)$

Using the Clamping Trick

$2^{n/2-\epsilon}$ Queries

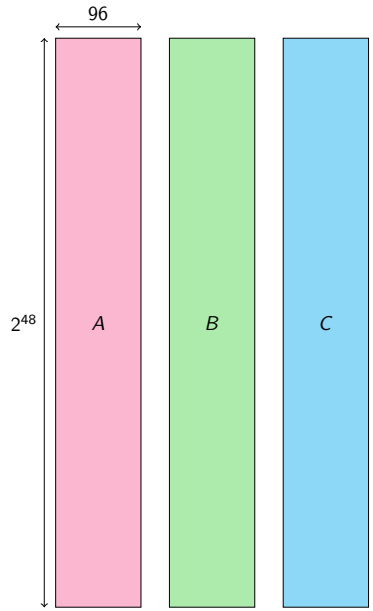
- Clamping performed over $(n/2 - 3 \cdot \epsilon)/2$ bits
- Size of lists: $\simeq 2^{n/4+\epsilon/2}$
- Time Complexity: $\mathcal{O}\left(2^{n/2+\epsilon}/n\right)$

Nikolić and Sasaki's setting

- $\epsilon = 1/2 \cdot \log((n/2)/\ln(n/2))$
- Time: $\mathcal{O}\left(2^{n/2}/\sqrt{n \cdot \ln(n)}\right)$, space $\mathcal{O}\left(2^{n/4+\epsilon/2}\right)$
- $\ln(n)$ speedup compared to [NS14]

Concrete Example

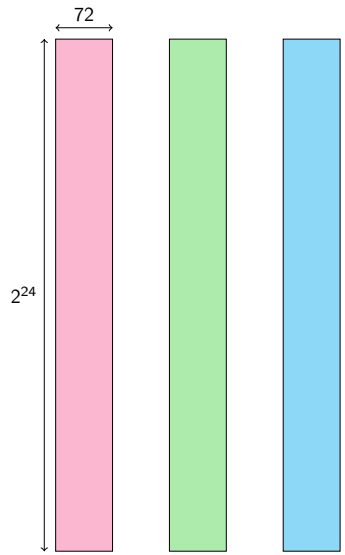
A 96-bit 3XOR



- Require $3 \cdot 2^{48}$ queries

Concrete Example

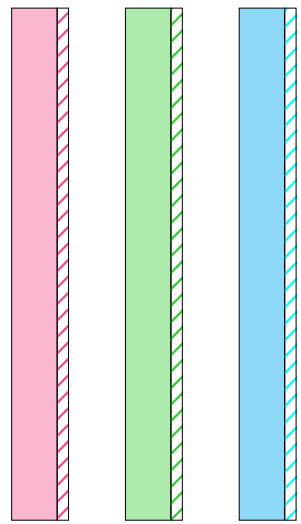
A 96-bit 3XOR



- Require $3 \cdot 2^{48}$ queries
- Perform the clamping on 24 bits

Concrete Example

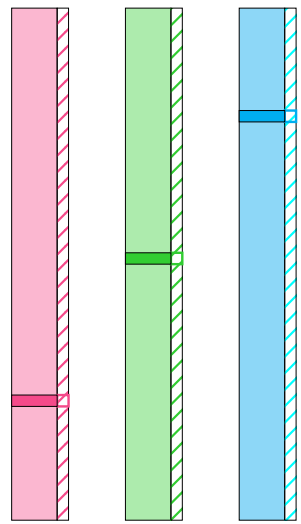
A 96-bit 3XOR



- Require $3 \cdot 2^{48}$ queries
- Perform the clamping on 24 bits
- Process the lists on the first 64 bits of each entries (Find all solutions)

Concrete Example

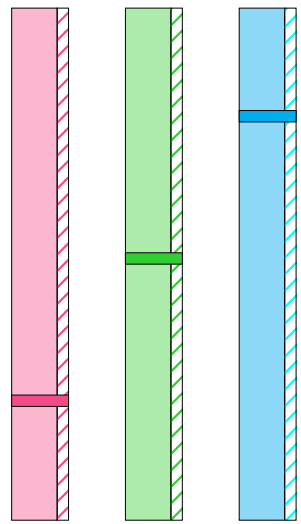
A 96-bit 3XOR



- Require $3 \cdot 2^{48}$ queries
- Perform the clamping on 24 bits
- Process the lists on the first 64 bits of each entries (Find all solutions)
- Test them on the remaining 8 bits (about 256 tests)

Concrete Example

A 96-bit 3XOR



- Require $3 \cdot 2^{48}$ queries
- Perform the clamping on 24 bits
- Process the lists on the first 64 bits of each entries (Find all solutions)
- Test them on the remaining 8 bits (about 256 tests)

Experimentations

- 3XOR of 96 bits of SHA-256
- Tests performed on a Haswell Core i5 CPU

Timing

	Quadratic	Our Algorithm
CPU hours	340	105
Data	576 MB	576 MB

Experimentations

- 3XOR of 96 bits of SHA-256
- Tests performed on a Haswell Core i5 CPU

Timing

	Quadratic	Our Algorithm
CPU hours	340	105
Data	576 MB	576 MB

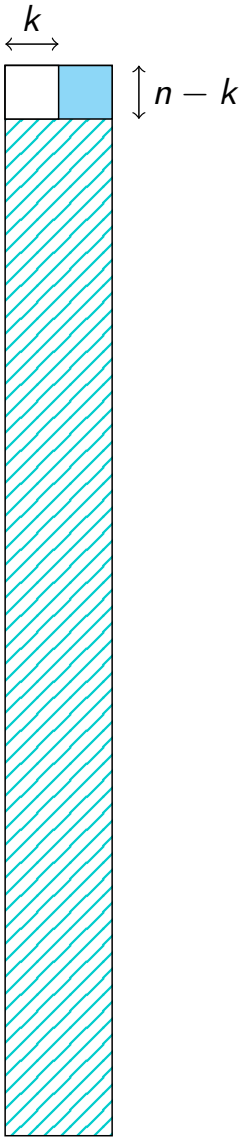
Creation of the lists: $\times 100$ slower than processing them!

In a Nutshell

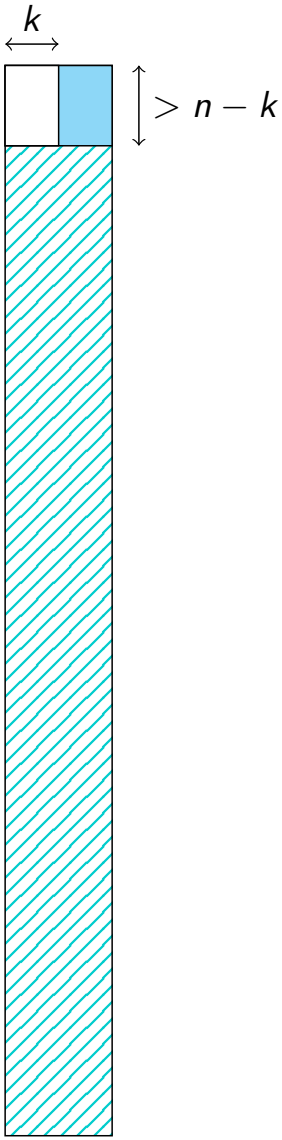
This Algorithm...

- can be applied to any size of input list
- has a $\times n$ speed-up compared to the Quadratic Algorithm
- is about 3 times faster, in practice ($n = 96$)
- is faster than [NS14] with the same amount of data, in theory
- is the same than [Joux09] with the same amount of data

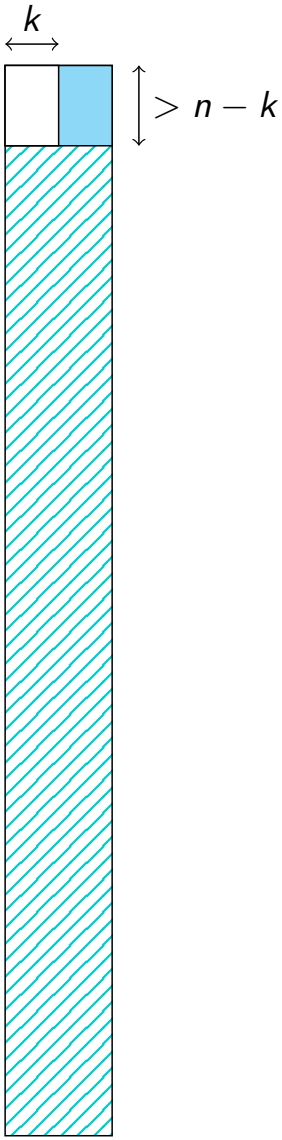
Possible Improvement



Possible Improvement



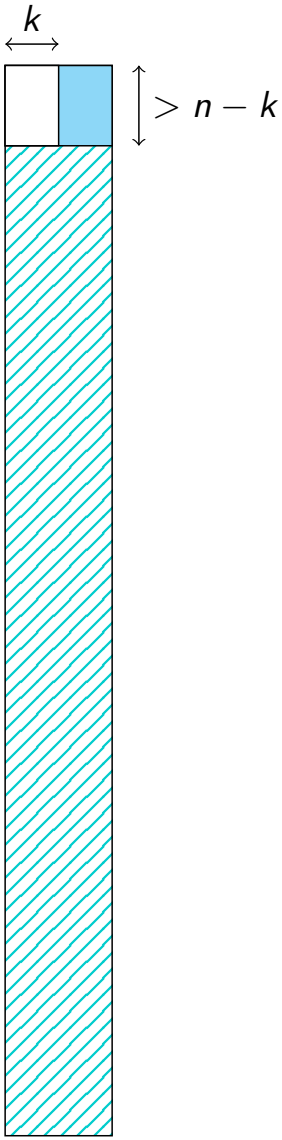
Possible Improvement



Problem

Given a list C of size N of n-bit uniformly random elements, find a k-dimensional vectorial subspace $\mathcal{V}$ of $\mathbb{F}_2^n$, s.t. $\mathcal{V} \cap C$ is maximized.

Possible Improvement

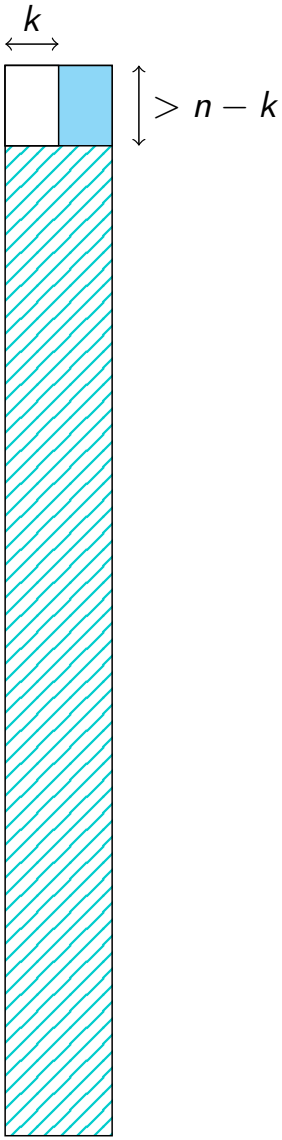


Problem

Given a list C of size N of n-bit uniformly random elements, find a k-dimensional vectorial subspace $\mathcal{V}$ of $\mathbb{F}_2^n$, s.t. $\mathcal{V} \cap C$ is maximized.

- This problem is hard
- Many subspaces are required
- This step must not dominate the whole procedure

Possible Improvement



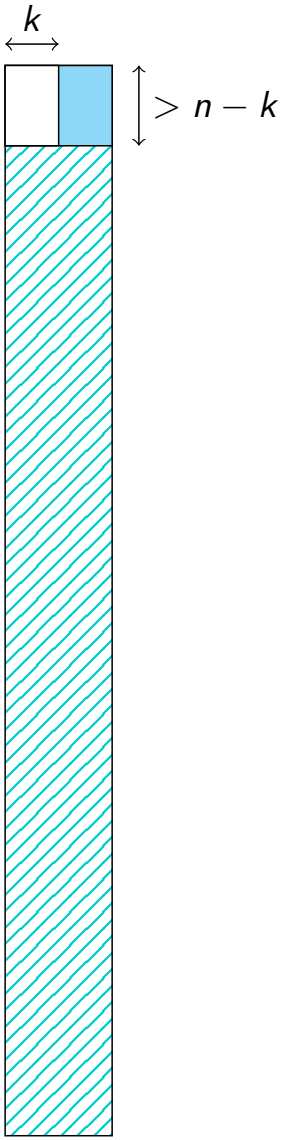
Problem

Given a list C of size N of n-bit uniformly random elements, find a k-dimensional vectorial subspace $\mathcal{V}$ of $\mathbb{F}_2^n$, s.t. $\mathcal{V} \cap C$ is maximized.

- This problem is hard
- Many subspaces are required
- This step must not dominate the whole procedure

$\Rightarrow$ Our goal: Find many $\mathcal{V}_i$ such that $\mathcal{V}_i \cap C$ is large.

Possible Improvement



Problem

Given a list C of size N of n-bit uniformly random elements, find a k-dimensional vectorial subspace $\mathcal{V}$ of $\mathbb{F}_2^n$, s.t. $\mathcal{V} \cap C$ is maximized.

- This problem is hard
- Many subspaces are required
- This step must not dominate the whole procedure

$\Rightarrow$ Our goal: Find many $\mathcal{V}_i$ such that $\mathcal{V}_i \cap C$ is large.

For now: Only constant time improvement

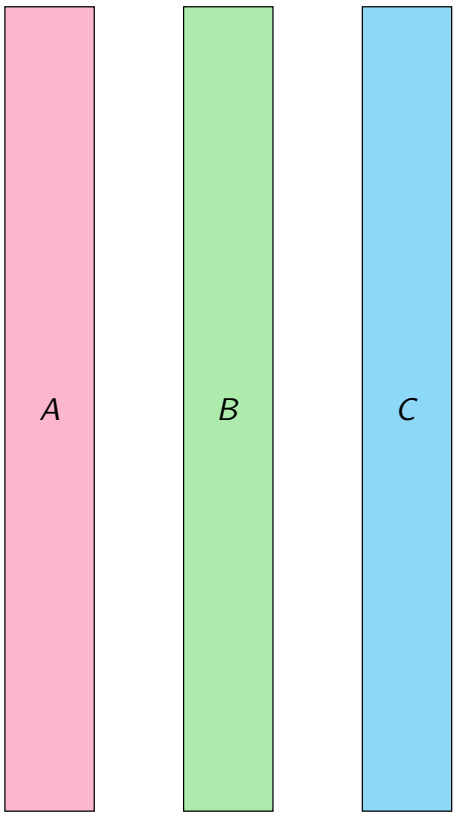
1. Background

2. Our New Algorithm

3. Adaptation of BDP Algorithm for the 3SUM problem

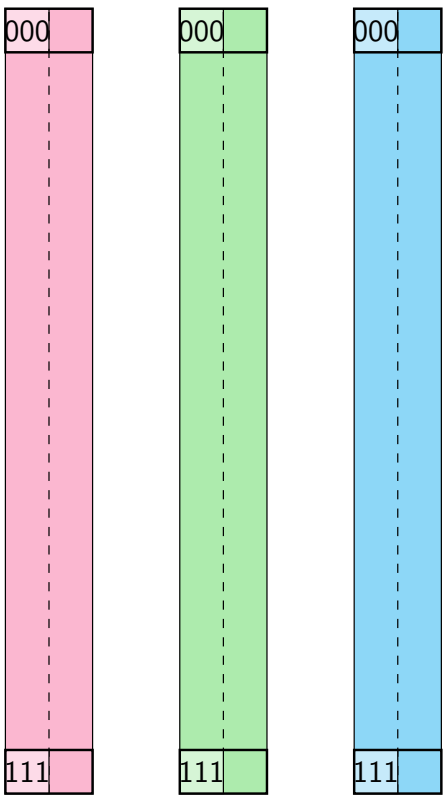
A 3XOR Adaptation of [BDP05]

- Originally designed for the 3SUM Problem over $(\mathbb{Z}, +)$
- We transposed it for the 3XOR Problem



A 3XOR Adaptation of [BDP05]

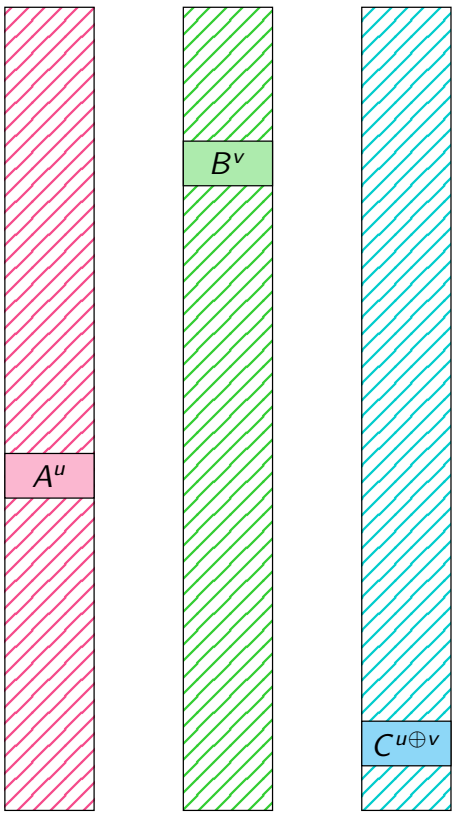
- Originally designed for the 3SUM Problem over $(\mathbb{Z}, +)$
- We transposed it for the 3XOR Problem



- Dispatch entries according to first k bits
- A^u: Bucket of elements of A starting by u

A 3XOR Adaptation of [BDP05]

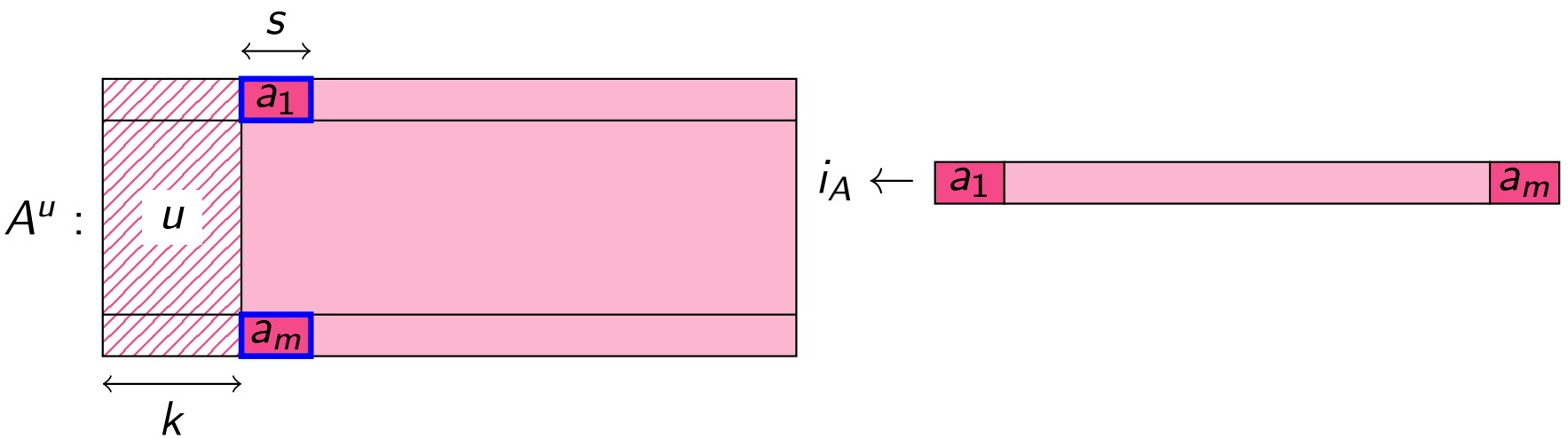
- Originally designed for the 3SUM Problem over $(\mathbb{Z}, +)$
- We transposed it for the 3XOR Problem



- Dispatch entries according to first k bits
- A^u: Bucket of elements of A starting by u
- For each triplet $(A^u, B^v, C^{u\oplus v})$ perform constant time preliminary test
- If the test fail: no solution for sure
- If the test succeed: there may be a solution
  - Solve the small instance

Preliminary Test

Instance $(A^u, B^v, C^{u \oplus v})$

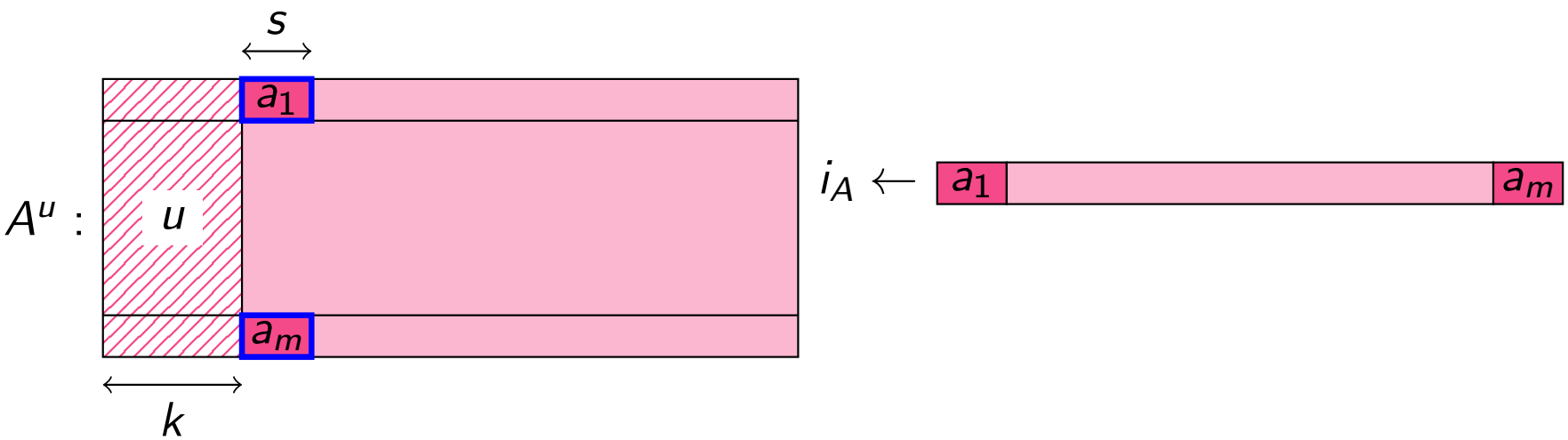


$$T[i] = 1 \iff \exists j, k, \ell \text{ s.t. } a_j \oplus b_k \oplus c_\ell = 0$$

$$T[i] = 0 \Rightarrow \text{No solution in } (A^u, B^v, C^{u \oplus v})$$

Preliminary Test

Instance $(A^u, B^v, C^{u \oplus v})$

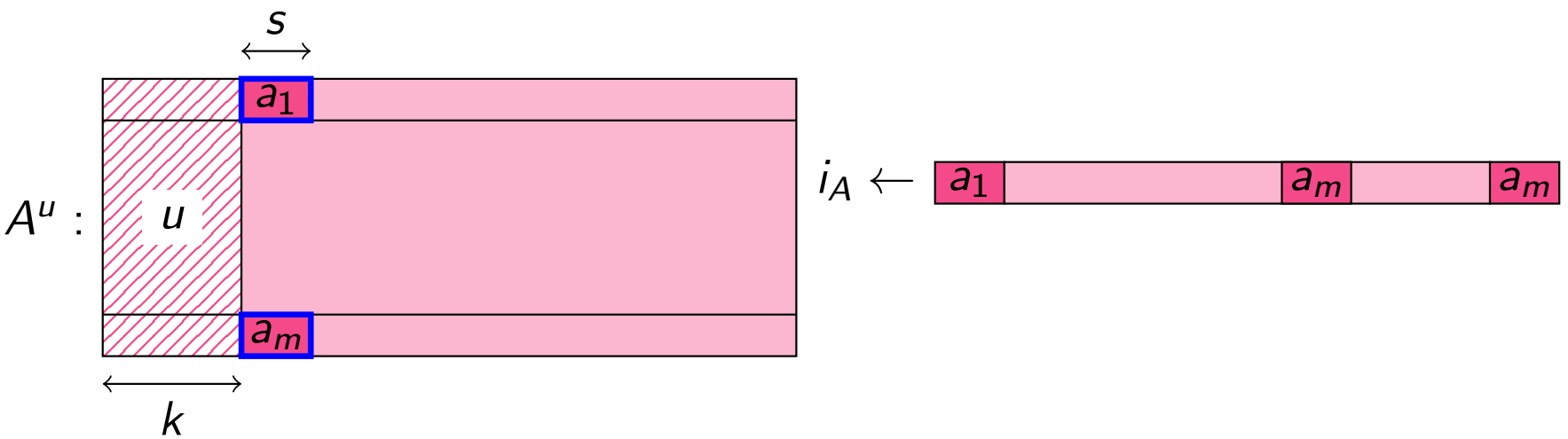


$$T[i] = 1 \iff \exists j, k, \ell \text{ s.t. } a_j \oplus b_k \oplus c_\ell = 0$$

$$T[i] = 0 \Rightarrow \text{No solution in } (A^u, B^v, C^{u \oplus v})$$

Preliminary Test

Instance $(A^u, B^v, C^{u \oplus v})$



$$T[i] = 1 \iff \exists j, k, \ell \text{ s.t. } a_j \oplus b_k \oplus c_\ell = 0$$

$$T[i] = 0 \Rightarrow \text{No solution in } (A^u, B^v, C^{u \oplus v})$$

The Gory Part Starts Now...

WARNING

The following slides

- are very messy
- introduces a lot of notations
- talk about probability

The Gory Part Starts Now...

WARNING

The following slides

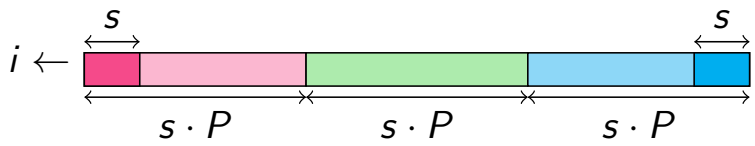
- are very messy
- introduces a lot of notations
- talk about probability

But...

Don't worry: This is nearly over!

BDP in Theory

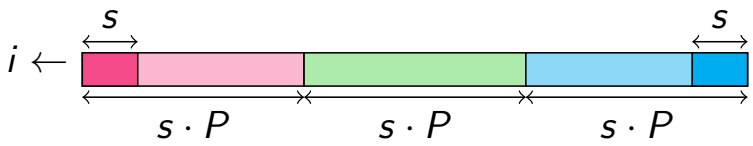
For each $(A^u, B^v, C^{u \oplus v})$:



- $T[i] = 1 \iff$ Partial solution over s bits
- $T[i] = 0 \implies$ No solution

BDP in Theory

For each $(A^u, B^v, C^{u \oplus v})$:

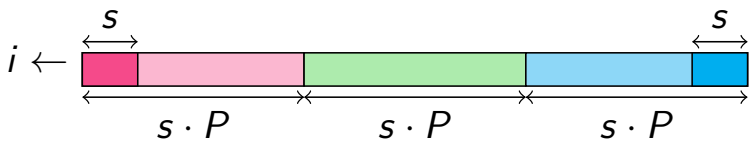


- $T[i] = 1 \iff$ Partial solution over s bits
- $T[i] = 0 \implies$ No solution

- m: expected size of a bucket
- w: size of a machine word
- $P = c1 \cdot m$: upper bound on size of a bucket w.h.p. (Chernoff)
- $s = c2 \cdot \log(w)$: only one triplet passes the test for large n
- $3 \cdot s \cdot P \leq n/3$: No space overhead.

BDP in Theory

For each $(A^u, B^v, C^{u \oplus v})$:



- $T[i] = 1 \iff$ Partial solution over s bits
- $T[i] = 0 \implies$ No solution

- m: expected size of a bucket
- w: size of a machine word
- $P = c1 \cdot m$: upper bound on size of a bucket w.h.p. (Chernoff)
- $s = c2 \cdot \log(w)$: only one triplet passes the test for large n
- $3 \cdot s \cdot P \leq n/3$: No space overhead.

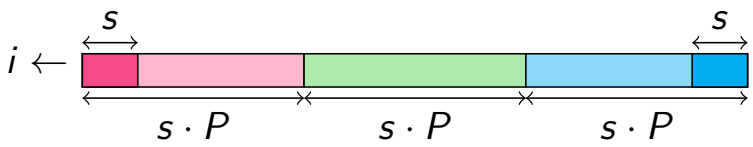
Complexity

When n grows up to infinity:

$$\text{Time: } \mathcal{O}\left(\frac{2^{2n/3}\log^2(n)}{n^2}\right), \text{Space: } \mathcal{O}\left(2^{n/3}\right)$$

BDP In Practice

For each $(A^u, B^v, C^{u \oplus v})$:

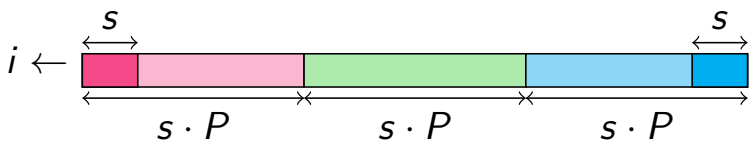


- $T[i] = 1 \iff$ Partial solution over s bits
- $T[i] = 0 \implies$ No solution

- m: expected size of a bucket
- $w = 64$: size of a machine word
- $P = 4.162 \cdot m$: upper bound on size of a bucket w.h.p. (Chernoff)
- $s = 3 \cdot \log(w)$: only one triplet passes the test for large n
- $3 \cdot s \cdot P = n/3$: No space overhead.

BDP In Practice

For each $(A^u, B^v, C^{u \oplus v})$:



- $T[i] = 1 \iff$ Partial solution over s bits
- $T[i] = 0 \implies$ No solution

- m: expected size of a bucket
- $w = 64$: size of a machine word
- $P = 4.162 \cdot m$: upper bound on size of a bucket w.h.p. (Chernoff)
- $s = 3 \cdot \log(w)$: only one triplet passes the test for large n
- $3 \cdot s \cdot P = n/3$: No space overhead.

Concrete example

n = 96:

Expected size of a bucket: $m = 0.14$

$\implies$ Completely impractical

Conclusion

This work

- Discusses issues arising from the 3XOR problem
- Propose a new practical algorithm for the 3XOR problem, that is
  - $n\times$ faster than the quadratic algorithm in theory
  - $3\times$ faster than the quadratic algorithm with our benchmarks
  - Code available:
    - `https://github.com/cbouilla/3XOR`
- Propose an adaptation of [BDP05] algorithm that is
  - asymptotically faster than other algorithms
  - Totally impractical

Conclusion

This work

- Discusses issues arising from the 3XOR problem
- Propose a new practical algorithm for the 3XOR problem, that is
  - $n\times$ faster than the quadratic algorithm in theory
  - $3\times$ faster than the quadratic algorithm with our benchmarks
  - Code available:
    - `https://github.com/cbouilla/3XOR`
- Propose an adaptation of [BDP05] algorithm that is
  - asymptotically faster than other algorithms
  - Totally impractical

What's Next?

- Compute a 128-bit 3XOR on SHA-256
  - Expect to have the lists in about 1.5-2 years (using one Antminer S7)
- Look at different settings (e.g. Sparse entries, 3SUM over $(\mathbb{Z}, +)$)

Thank you for your time!