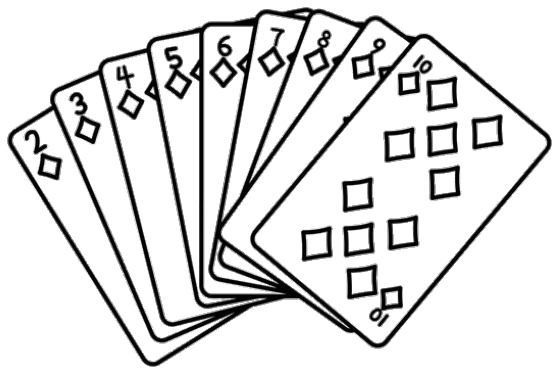
Lightweight Sorting in Approximate Homomorphic Encryption

Lorenzo Rovida, Alberto Leporati and Simone Basile

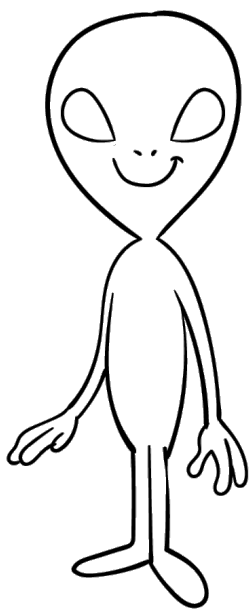
University of Milano-Bicocca, Milan, Italy



ESAT, KU Leuven – COSIC Seminar – June 17th, 2025

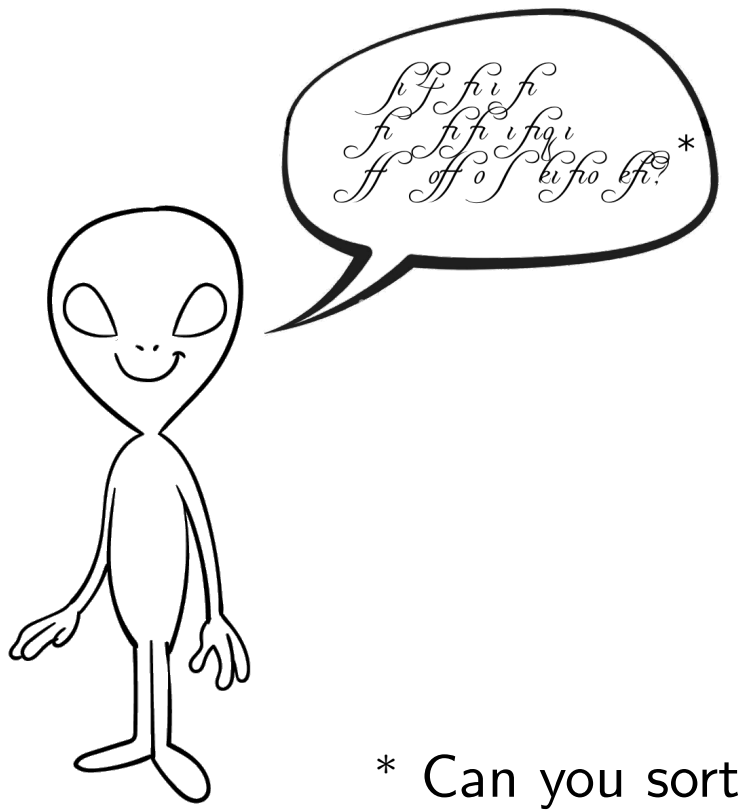
A story - the alien friend

Imagine meeting your alien friend...



A story - the alien friend

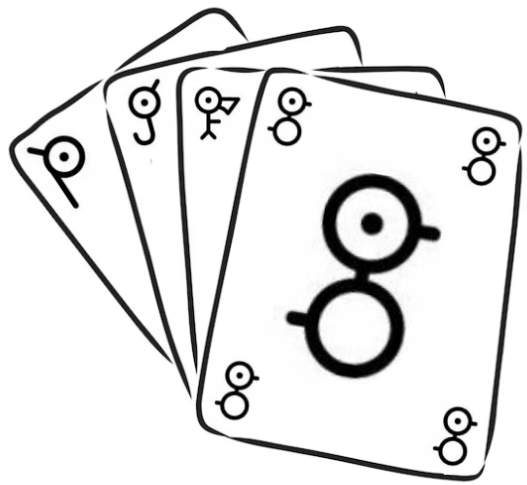
... and they're like:



* Can you sort my deck of cards?

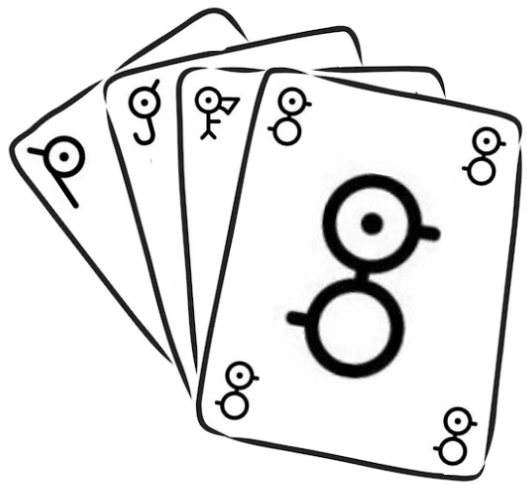
A story - the alien friend

You always help a friend! Therefore, you grab their cards, but...



A story - the alien friend

... they are not readable! Can we still sort them?



Motivations

- Sorting values is a crucial building block for many algorithms and applications.
- Most of sorting procedures are *data-dependent*.
- These fail to work over encrypted inputs!
- Suppose we want to implement a sorting algorithm based on FHE primitives, this begs the question:

How can we efficiently sort encrypted values, with a oblivious algorithm?

The goal - formally

Currently, it seems that the best oblivious algorithms that we have are the following:

- Network-based: based on a fixed sequece of *compare-and-swap* operations.
- Permutation-based: deviates from the *compare-and-swap* approach by finding a permutation matrix P that correctly sorts the input vector.

Some related works

For both approaches, we can find some reference works:

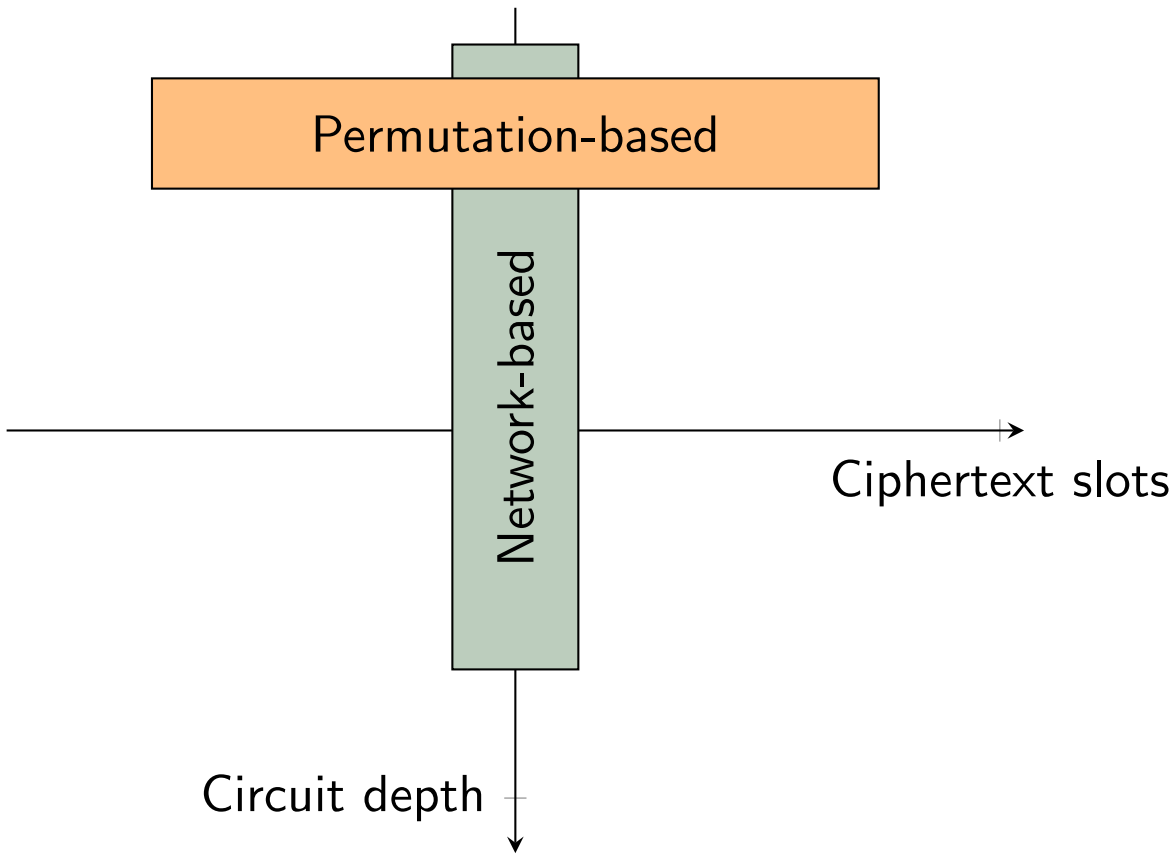
- Network-based: [CGKP25, HKC+21]: the former implements a top-k selection in TFHE, the latter a bitonic sorting based on CKKS, requiring $\Theta(n)$ slots.
- Permutation-based: [MEHP25] proposed very efficient sorting requiring $\Theta(n^2)$ ciphertext slots in CKKS.

[1] Kelong Cong, Robin Geelen, Jiayi Kang, and Jeongeun Park. "Revisiting Oblivious Top-k Selection with Applications to Secure k-NN Classification". In: *SAC 2024*

[2] Seungwan Hong et al. "Efficient Sorting of Homomorphic Encrypted Data With k-Way Sorting Network". In: *IEEE Transactions on Information Forensics and Security* 16 (2021)

[3] Federico Mazzone, Maarten Everts, Florian Hahn, and Andreas Peter. "Efficient Ranking, Order Statistics, and Sorting under CKKS". In: *USENIX Security '25*

Permutation-based vs Network-based



The CKKS scheme

- We will implement the algorithms using RNS-CKKS [KPP22], which is an approximate homomorphic encryption scheme working with fixed-point representation of complex numbers.
- Ciphertexts $\boldsymbol{c} \in (\mathbb{Z}_q[X]/(X^N+1))^2$ represent encryptions of vectors $\boldsymbol{v} \in \mathbb{C}^{N/2}$, and computations are performed slot-wise in parallel with SIMD.
- The basic available operations are additions, multiplications and rotations.

[1] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. "Approximate Homomorphic Encryption with Reduced Approximation Error". In: *CT-RSA 2022*, pp. 120–144

Permutation-based sorting [MEHP25]

This approach, introduced in [MEHP25], is based on two phases:

1. Indexing: given $\boldsymbol{v}$, we want to build its indexing vector $\boldsymbol{s}$, namely the position of each element v_i in sort($\boldsymbol{v}$), e.g.:

   Given $\boldsymbol{v} = (5, 0, 9, 7)$, we get $\boldsymbol{s} = (2, 1, 4, 3)$

Permutation-based sorting [MEHP25]

This approach, introduced in [MEHP25], is based on two phases:

1. Indexing: given $\boldsymbol{v}$, we want to build its indexing vector $\boldsymbol{s}$, namely the position of each element v_i in sort($\boldsymbol{v}$), e.g.:

   Given $\boldsymbol{v} = (5, 0, 9, 7)$, we get $\boldsymbol{s} = (2, 1, 4, 3)$

2. Permutation: given the indexing vector, we want to construct the permutation matrix P, e.g.:

   Given $\boldsymbol{s} = (2, 1, 4, 3)$, we get $P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

Phase 1: Indexing

Given $\boldsymbol{v} \in \mathbb{R}^n$, we can recover the indexing $\boldsymbol{s} \in \mathbb{Z}^n$ by *counting* for each element v_i, how many elements v_j are smaller than v_i:

$$s_i = 0.5 + \sum_{i=1}^{n} \mathsf{sgn}(v_i - v_j)$$

where

$$\mathsf{sgn}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0.5 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

Phase 1: Indexing

For instance, following the previous example with $\boldsymbol{v} = (5, 0, 9, 7)$, we obtain $\boldsymbol{s} = (s_1, s_2, s_3, s_4)$ such that:

$$s_1 = 0.5 + \mathsf{sgn}(5-5) + \mathsf{sgn}(5-0) + \mathsf{sgn}(5-9) + \mathsf{sgn}(5-7) = 2$$
$$s_2 = 0.5 + \mathsf{sgn}(0-5) + \mathsf{sgn}(0-0) + \mathsf{sgn}(0-9) + \mathsf{sgn}(0-7) = 1$$
$$s_3 = 0.5 + \mathsf{sgn}(9-5) + \mathsf{sgn}(9-0) + \mathsf{sgn}(9-9) + \mathsf{sgn}(9-7) = 3$$
$$s_4 = 0.5 + \mathsf{sgn}(7-5) + \mathsf{sgn}(7-0) + \mathsf{sgn}(7-9) + \mathsf{sgn}(7-7) = 4$$

We can parallelize this computations with SIMD by using $\Theta(n^2)$ slots!

Phase 2: Permutation matrix

Now compare each s_i with values from 1 up to n! Given:

$$\text{equal}(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

In order to obtain the i-th column $\boldsymbol{p}_i$ of P, we simply compute[1]:

$$\boldsymbol{p}_i = \text{equal}\left((s_1, s_2, s_3, s_4) - (i, i, \ldots, i)\right)$$

The first column $i = 1$ in our example will be:

$$\begin{aligned} \boldsymbol{p}_1 &= \text{equal}\left((2, 1, 4, 3) - (1, 1, 1, 1)\right) = \text{equal}\left((1, 0, 3, 2)\right) \\ &= (0, 1, 0, 0) \end{aligned}$$

[1] By abuse of notation, we allow the function to be applied on vectors by applying equal(x) independently to each entry.

Handling equal values

- One may notice that this approach does not handle very well repeated elements.
- For example, the indexing $\boldsymbol{s}$ given by the input vector $\boldsymbol{v} = (7, 8, 8, 1)$ will be equal to $\boldsymbol{s} = (2, 3.5, 3.5, 1)$

Handling equal values

- One may notice that this approach does not handle very well repeated elements.
- For example, the indexing $\boldsymbol{s}$ given by the input vector $\boldsymbol{v} = (7, 8, 8, 1)$ will be equal to $\boldsymbol{s} = (2, 3.5, 3.5, 1)$
- Good news: we can easily fix this!
- Idea: compute a *tie-correction offset* $\boldsymbol{o}$. Following the example with $\boldsymbol{s}$, we would need $\boldsymbol{o} = (0, -0.5, 0.5, 0)$.

Let's make it concrete

- The presented framework abstracts from any particular computational method.
- But it well fits SIMD FHE schemes, because of its "horizontal" nature.
- Most of computations can be parallelized by repeating the values in $\boldsymbol{v}$, with the main obstacles being are the two nonlinear functions sgn(x) and equal(x).

Chebyshev polynomials

We introduce a very useful tool that allows for efficient approximations of nonlinear functions.

- By performing a polynomial regression of degree d over the $d+1$ points $(x_i, f(x_i))$, where

$$x_i = \cos\left(\frac{(i+\frac{1}{2})\pi}{d}\right) \text{ with } 0 \leq i < d,$$

  is the i-th Chebyshev root, we obtain an approximation for a nonlinear function that logarithmically far from the optimal.
- We use a modified version of the Paterson-Stockmeyer algorithm to efficiently evaluate them proposed in [CCS19].

[1] Hao Chen, Ilaria Chillotti, and Yongsoo Song. "Improved Bootstrapping for Approximate Homomorphic Encryption". In: *EUROCRYPT '19*

Approximating $\mathrm{sgn}(x)$

However, we should be cautious... $\mathrm{sgn}(x)$ is far from being an easy function!

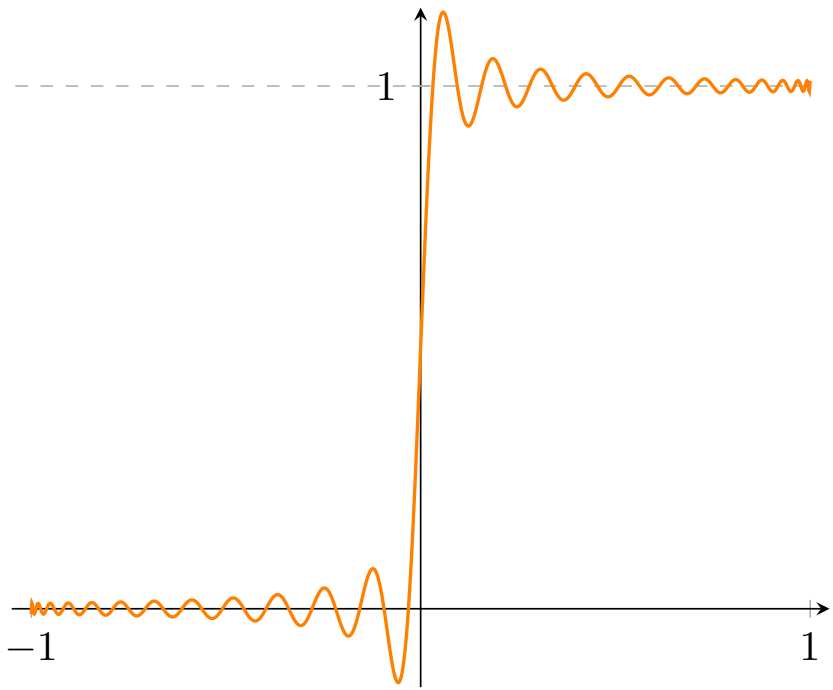


Figure: Chebyshev approximation with degree $d = 50$

Approximating $\operatorname{sgn}(x)$

In order to understand why this happens, look at the positions of the regression points!

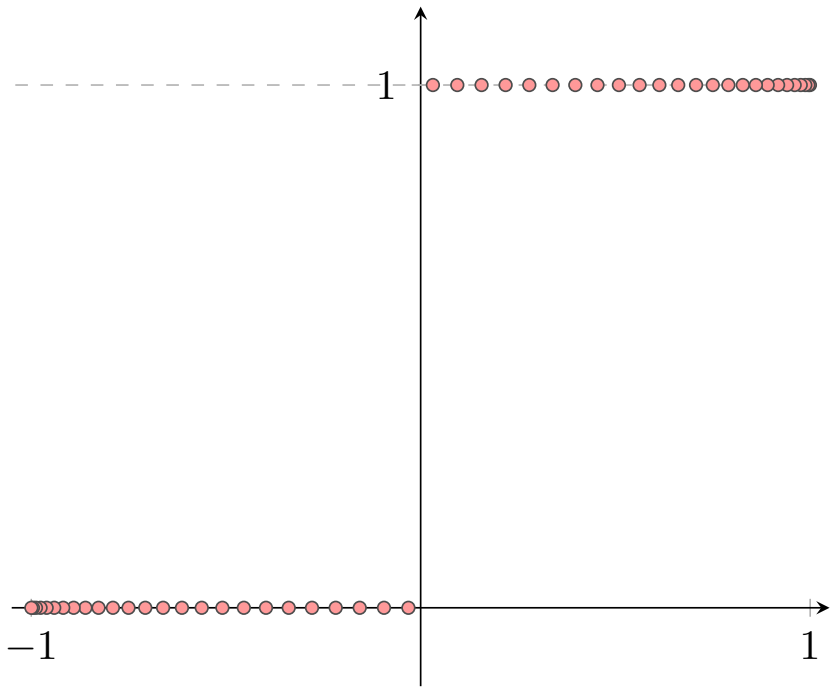


Figure: Regression points when $d = 50$

Approximating sgn(x)

Idea: let us try to smooth the function, use instead $1/(1+e^{-kx})$, for a sufficiently large $k > 0$ (i.e., the sigmoid function):

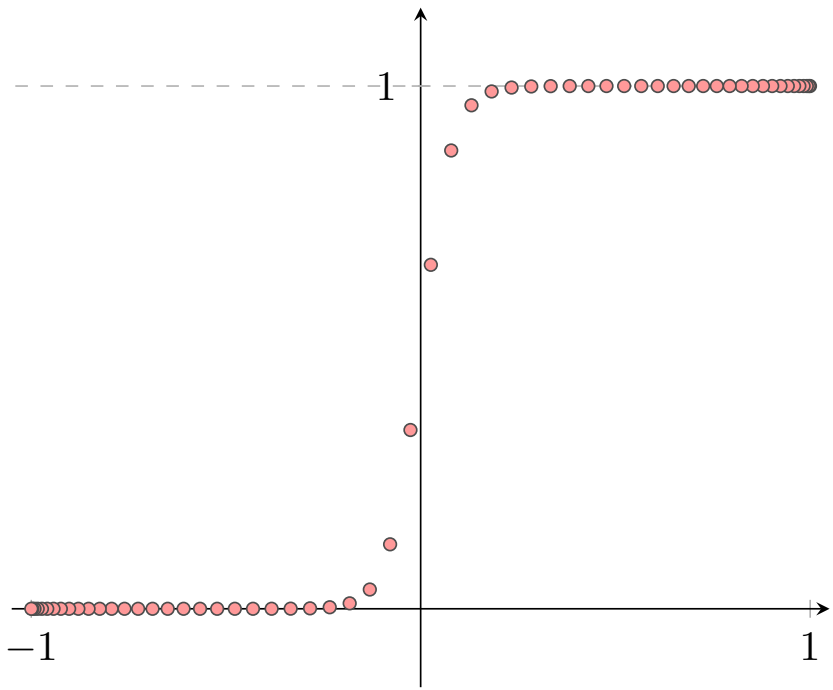


Figure: Points of sigmoid approximation, with $k = 25$ and degree $d = 50$

Approximating $\mathbf{sgn}(x)$

This leads to an approximation that has no oscillations!

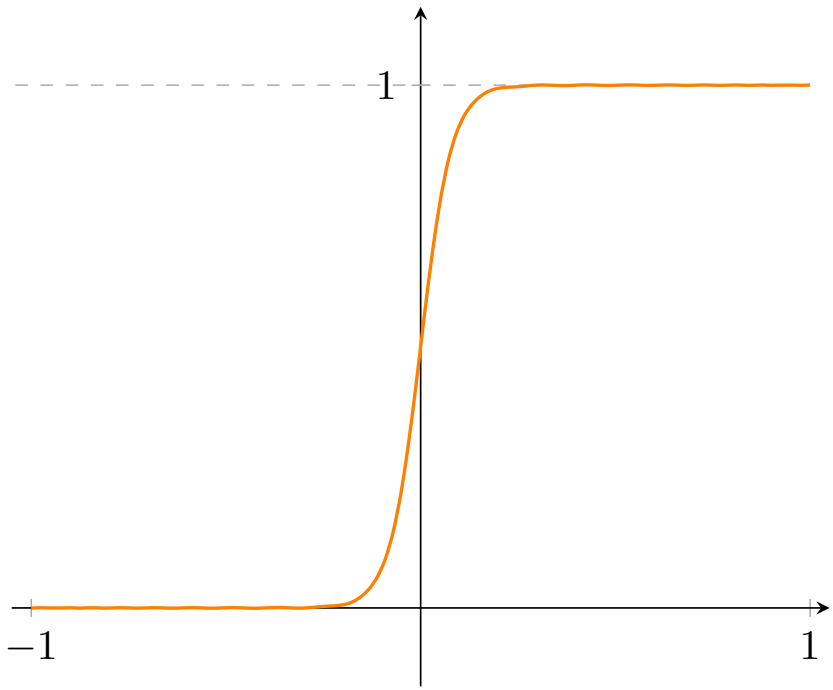


Figure: Sigmoid approximation, with $k = 25$ and degree $d = 50$

Approximating sgn(x)

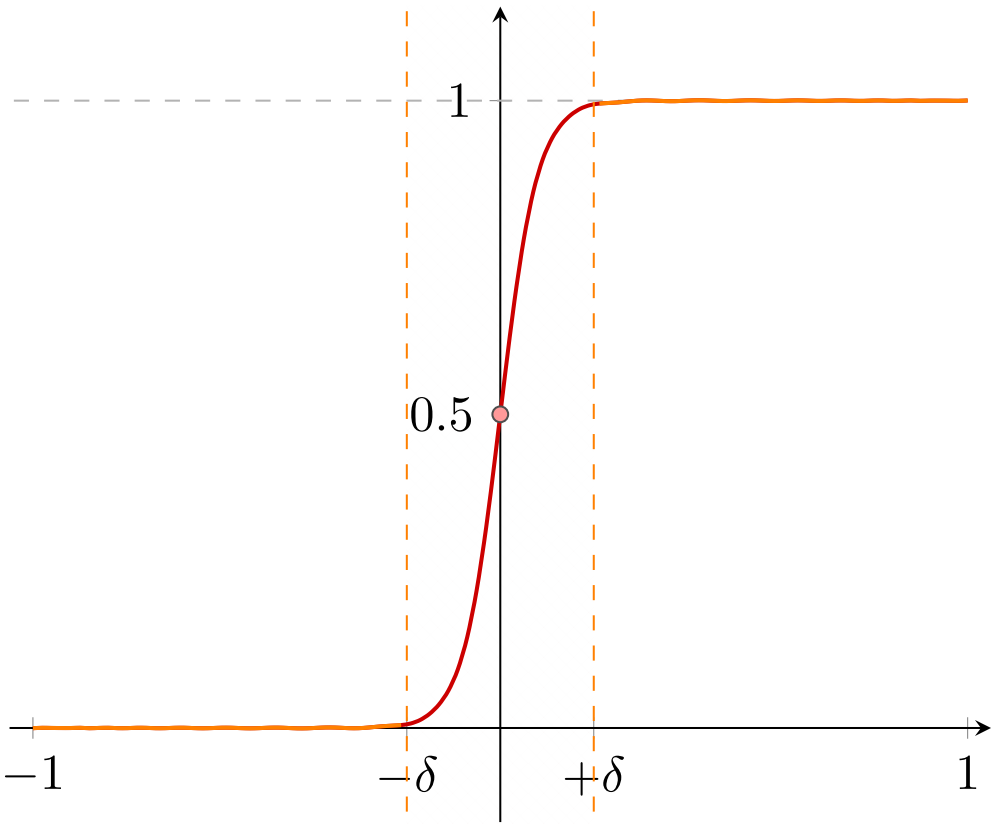
- However, this approximation does not work well when approaching the origin $x = 0$.
- We assume that every pair of element to be sorted is at distance at most δ, e.g., $\delta = 0.01$ for inputs in $[0, 1]$.
- We can fix some appropriate pair (d, k) that well-approximates sgn(x) in the following interval[2]:

$$[-1, -\delta] \cup [0] \cup [\delta, 1]$$

[2] We can prove that $x = 0$ is still approximated correctly as 0.5 for any choice of $\delta > 0, k > 1$ and $d > 1$

Approximating $\mathrm{sgn}(x)$

Visually, if our assumptions on inputs hold, we obtain the following:



Approximating equal(x)

The next function is equal(x), used to check equalities in the indexing vector, which is even worse!

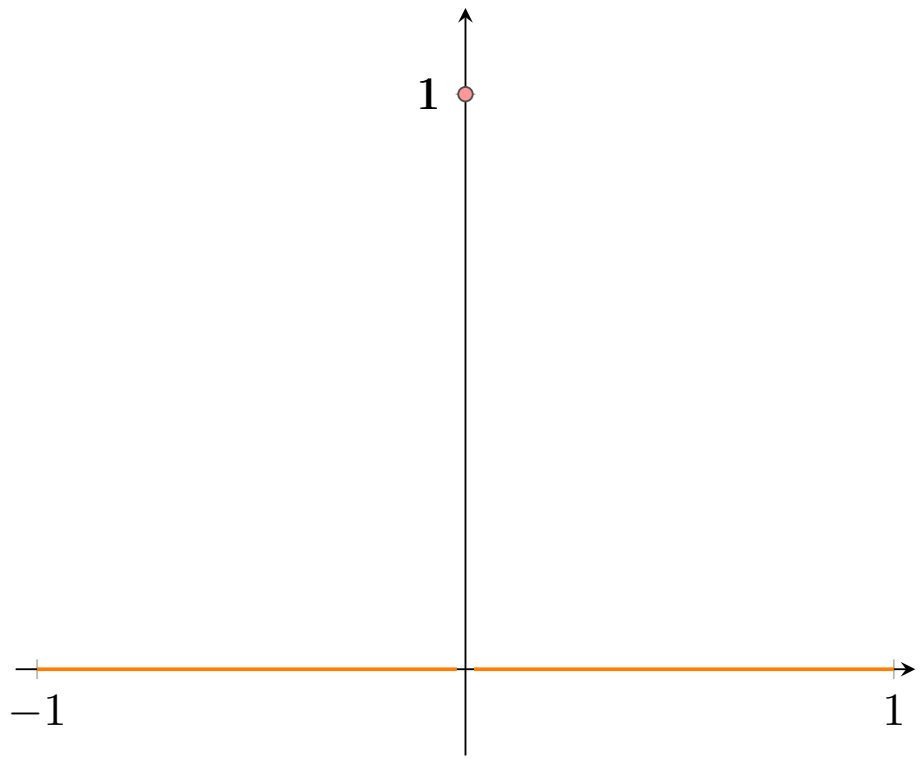


Figure: Plot of the equal(x) function

Approximating equal(x)

In [MEHP25], they propose to use the following, which even allows to handle errors up to ± 0.5:

$$\mathsf{sgn}(x + 0.5/a) \cdot (1 - \mathsf{sgn}(x - 0.5/a))$$

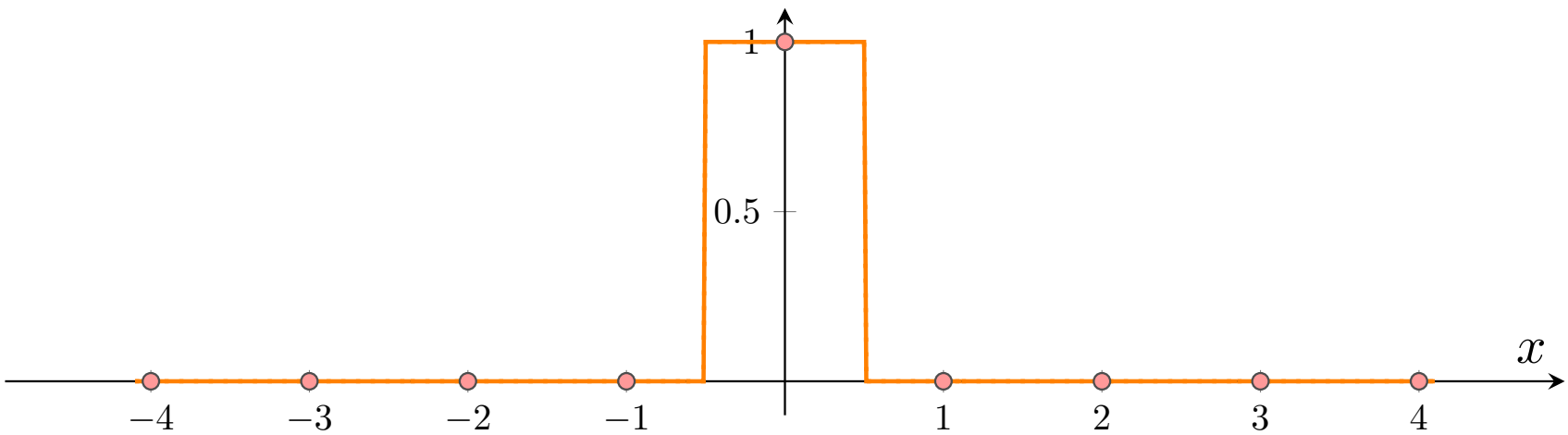


Figure: Plot of approximation of equal(x) function used in [MEHP25]

Approximating equal(x)

Idea: let us "relax" this function by observing that we only require this two constraints:

1. $f(0) = 1$ and
2. $f(x) = 0$ for all $x \in \mathbb{Z} \setminus 0$.

Approximating equal(x)

A nice function that respects these constraints is the sinc(x) function:

$$\text{sinc}(x) = \frac{\sin(x\pi)}{x\pi}$$

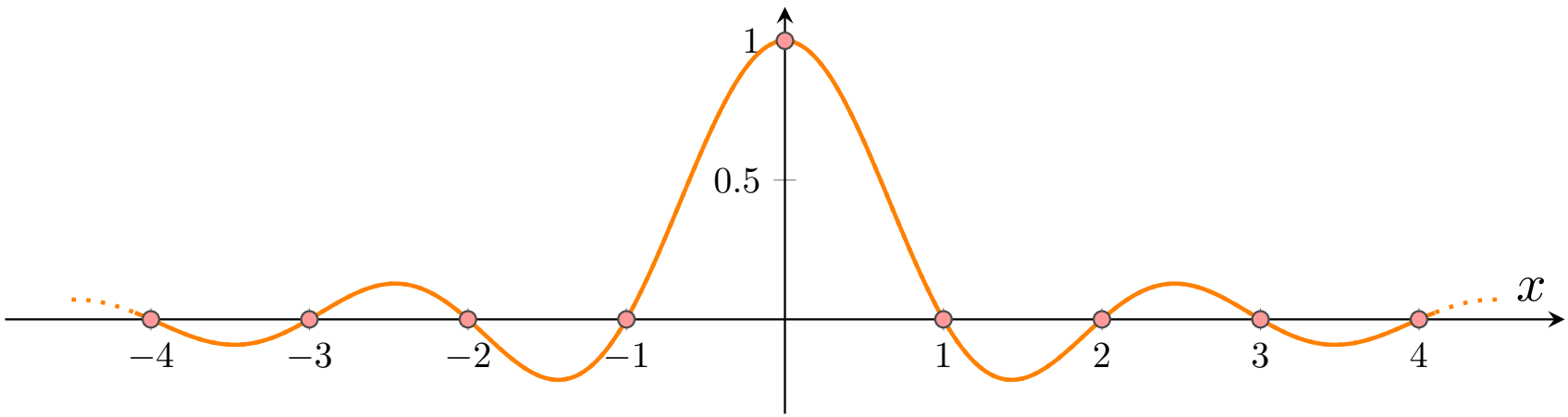


Figure: Plot of the sinc(x) function

Approximating equal(x)

- This function can be approximated without relying on sgn(x)
- Even with small degrees, Chebyshev polynomials allow for very accurate approximations
- The error is quadratic in $x = 0$ and linear in $\mathbb{Z} \setminus 0$, we require the inputs to be as close as possible to $\mathbb{Z}$.

An SIMD example

- Given the input $\boldsymbol{v} = (5, 0, 9, 7)$, we define two different encodings $\boldsymbol{v}_{\text{repeated}}, \boldsymbol{v}_{\text{expaded}}$ and compute their difference:

$\boldsymbol{v}_{\text{repeated}}$:	5	0	9	7	5	0	9	7	5	0	9	7	5	0	9	7	−
$\boldsymbol{v}_{\text{expanded}}$:	5	5	5	5	0	0	0	0	9	9	9	9	7	7	7	7	=
$\boldsymbol{v}_{\Delta}$:	0	−5	4	2	5	0	9	7	−4	−9	0	−2	−2	−7	2	0	

An SIMD example

- Now we apply (an approximation of) the $\mathrm{sgn}(x)$ function over $\boldsymbol{v}_\Delta$.

$\mathrm{sgn}(\boldsymbol{c}_\Delta)$:

0.5	0	1	1	1	0.5	1	1	0	0	0.5	0	0	0	1	0.5

An SIMD example

- Now we apply (an approximation of) the $\mathrm{sgn}(x)$ function on $\boldsymbol{v}_\Delta$.

$\mathrm{sgn}(\boldsymbol{c}_\Delta)$:

0.5	0	1	1	1	0.5	1	1	0	0	0.5	0	0	0	1	0.5

An SIMD example

- Then we count how many elements in $\boldsymbol{v}$ are smaller than each v_i, using a rotate-and-sum procedure at distance n:

$\mathrm{sgn}(\boldsymbol{c}_\Delta)$:	0.5	0	1	1	1	0.5	1	1	0	0	0.5	0	0	0	1	0.5
$0.5 + \sum$:	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3

An SIMD example

- Then we count how many elements in $\boldsymbol{v}$ are smaller than each v_i, using a rotate-and-sum procedure at distance n:

$\mathrm{sgn}(\boldsymbol{c}_\Delta):$	0.5	0	1	1	1	0.5	1	1	0	0	0.5	0	0	0	1	0.5
$0.5 + \sum:$	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3

An SIMD example

- Then we count how many elements in $\boldsymbol{v}$ are smaller than each v_i, using a rotate-and-sum procedure at distance n:

$\mathrm{sgn}(\boldsymbol{c}_\Delta)$:	0.5	0	1	1	1	0.5	1	1	0	0	0.5	0	0	0	1	0.5
$0.5 + \sum$:	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3

An SIMD example

- Then we count how many elements in $\boldsymbol{v}$ are smaller than each v_i, using a rotate-and-sum procedure at distance n:

$\mathrm{sgn}(\boldsymbol{c}_\Delta)$:	0.5	0	1	1	1	0.5	1	1	0	0	0.5	0	0	0	1	0.5
$0.5 + \sum$:	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3

An SIMD example

- To conclude, we simply apply the sinc(x) function!

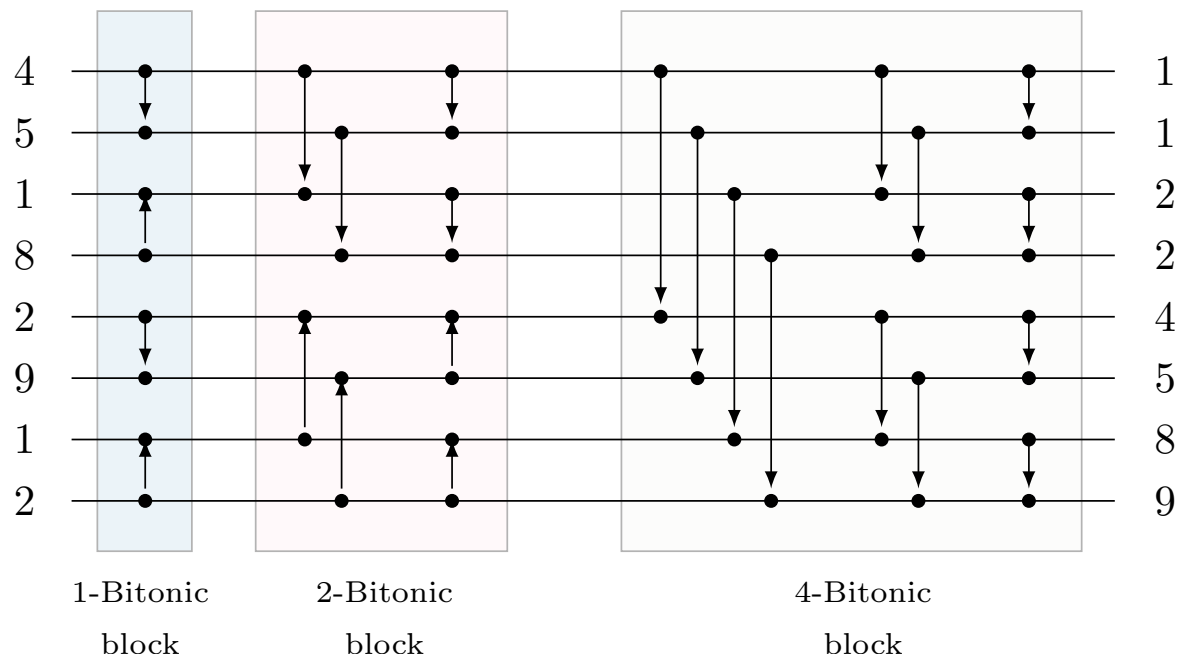
$\boldsymbol{s}$:	2	1	4	3	2	1	4	3	2	1	4	3	2	1	4	3
1 to n:	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
sinc(Δ) :	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1	0

To wrap-up

- The depth of this approach is roughly given by the evaluation of sgn(x) and equal(x) (plus one mult to evaluate the last $P \cdot \boldsymbol{v}$ product).
- Very efficient in terms of depth, might not require bootstrapping at all.
- Very inefficient in packing terms, requires n^2 slots.

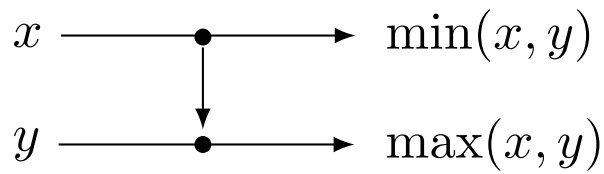
Network-based sorting

- Sorting-networks are oblivious sorting algorithms based on *compare-and-swap* operations.
- *Bitonic networks* in particular allow to sort a power-of-two number of elements in $\log_2(n)(\log_2(n)+1)/2$ layers.



Network-based sorting

- A *comparator* in a sorting network is built as:



- How can we compute $\max(x, y)$ and $\min(x, y)$?

Network-based sorting

- In FHE we can only evaluate additions and multiplications.
- Conditional statements like:

  if $x < y$ **then**
  return x
  else
  return y
  end if

  can not be implemented trivially (we can not change the flux of execution of the program w.r.t. the inputs)

Network-based sorting

- [HKC+21] propose to evaluate $x < y$ as a mask:

$$c = \frac{\text{sgn}(x - y) + 1}{2} = \begin{cases} 1 & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

  where now sgn(x) returns -1 if $x \leq y$, 1 otherwise. Then, this result is used to mask x and y as:

$$\min(x, y) = c \cdot y + (1 - c) \cdot x$$

- Yet another sgn(x) function...

[3] Seungwan Hong et al. "Efficient Sorting of Homomorphic Encrypted Data With k-Way Sorting Network". In: *IEEE Transactions on Information Forensics and Security* 16 (2021)

Network-based sorting

- Idea: we do not really need to compute $\mathsf{sgn}(x)$!
- Observe that:

$$\mathsf{min}(x, y) = x - \mathsf{max}(0, x - y)$$
$$\mathsf{max}(x, y) = x + y + \mathsf{min}(x, y)$$

- $\mathsf{max}(0, x)$ is the ReLU function, which can be accurately approximated with Chebyshev polynomials as it does not have weird steps!

Approximating $\mathrm{sgn}(x)$

Even with a small degree, the ReLU function can be well-approximated with Chebyshev polynomials.

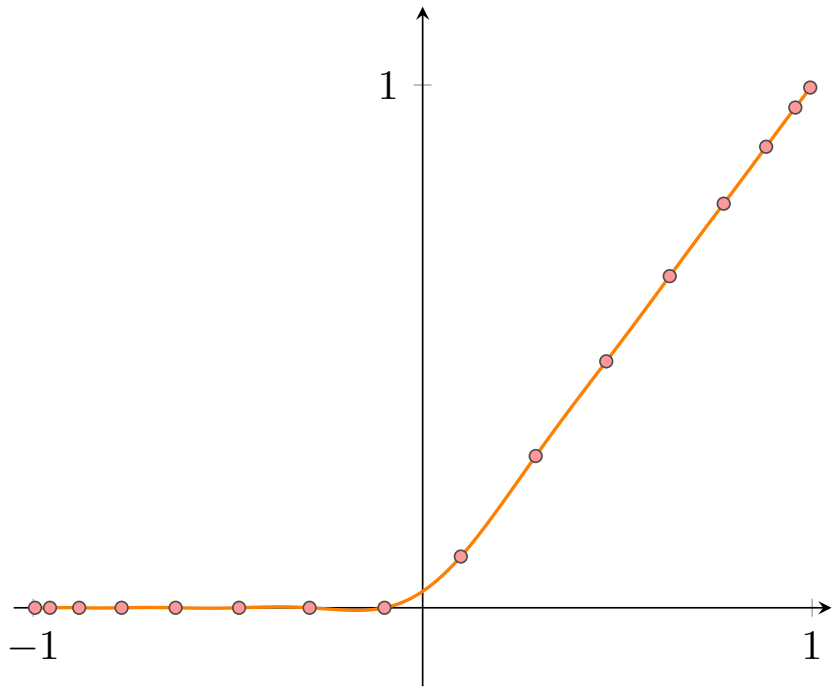
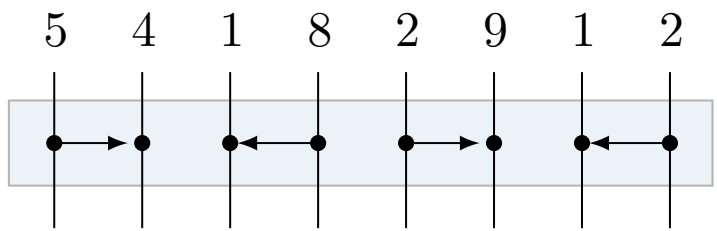


Figure: ReLU approximation, with degree $d = 15$

Putting everything together

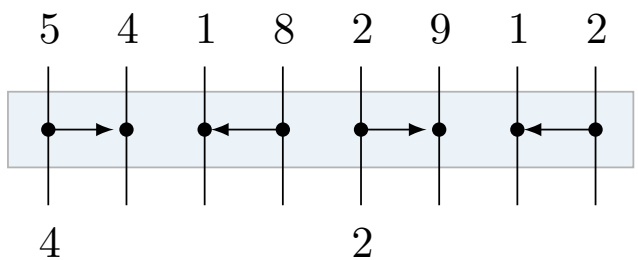
- We are now ready to show how to compute a layer of a sorting network efficiently, using SIMD computations.
- We use as an example the vector $\boldsymbol{v} = (5, 4, 1, 8, 2, 9, 1, 2)$.
- We show how to run a single network layer, the same procedure can be applied to any layer, consider the following:



Evaluating a sorting layer, SIMD version

1. Given the input vector $\boldsymbol{v} \in \mathbb{R}^n$, we first obtain the minimum between each pair connected by arrows:

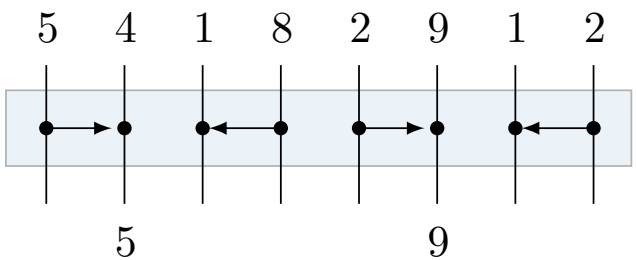
$$\boldsymbol{v}_1 = \underbrace{\boldsymbol{v}}_{x} - \mathsf{ReLU}\Big(\underbrace{\boldsymbol{v}}_{x} - \underbrace{\mathsf{RotLeft}(\boldsymbol{v}, 1)}_{y}\Big)$$



Evaluating a sorting layer, SIMD version

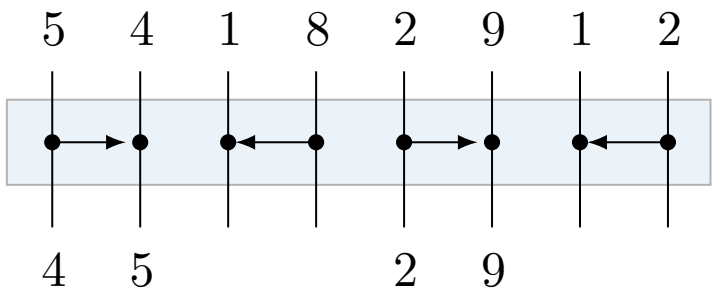
2. Then, find the maximum values between each pair, recycling material from the previous computaiton:

$$\boldsymbol{v}_2 = \big(\underbrace{\boldsymbol{v}}_{x} + \underbrace{\mathsf{RotRight}(\boldsymbol{v}, 1)}_{y} \big) - \underbrace{\mathsf{RotRight}(\boldsymbol{v}_1, 1)}_{\min(x,y)}$$



Evaluating a sorting layer, SIMD version

- In $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ we can find the results of the comparisons between pairs of elements with right arrows, namely:



Evaluating a sorting layer, SIMD version

3. Is is not necessary to evaluate the minimum function again, since the minimum values of the remaining wires have already been computed:

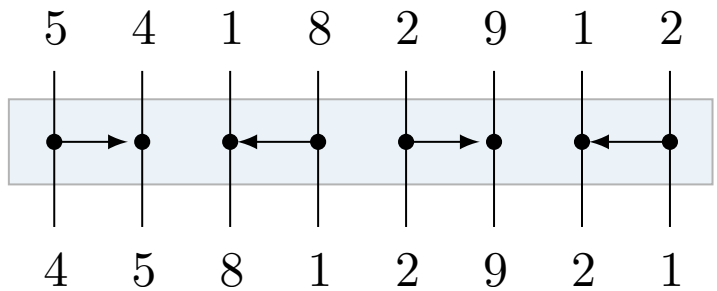
$$\boldsymbol{v}_3 = \mathsf{RotRight}(\boldsymbol{v}_1, 1)$$

4. As before, the maximum values of the remaining wires are computed using the previously computed minimum:

$$\boldsymbol{v}_4 = (\boldsymbol{v} + \mathsf{RotLeft}(\boldsymbol{v}, 1)) - \boldsymbol{v}_1$$

Evaluating a sorting layer, SIMD version

- Now, if we consider $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$ and $\boldsymbol{v}_4$ we have all the required values!



- Before summing them, we have to mask the correct values, as some slots may contain partial computations that we are not interested in (e.g., the minimum between the second and the third value).

Evaluating a sorting layer, SIMD version

- This approach allows to evaluate a sorting network in CKKS requiring $\log(n)(\log(n)+1)/2$ bootstrapping operations.
- The slots required for the SIMD algorithms are just $\Theta(n)$.
- We are not required to approximate $\mathrm{sgn}(x)$!

Experiments

- All experiments are performed on uniformly random elements in from $[0, 1]$, with steps of δ.
- All experiments have been conducted using the OpenFHE implementation of RNS-CKKS on a Macbook M1 Pro laptop with 16 GB of RAM (not the best machine :-)) and are easily reproducible with our open-source code on GitHub.

Experiment 1

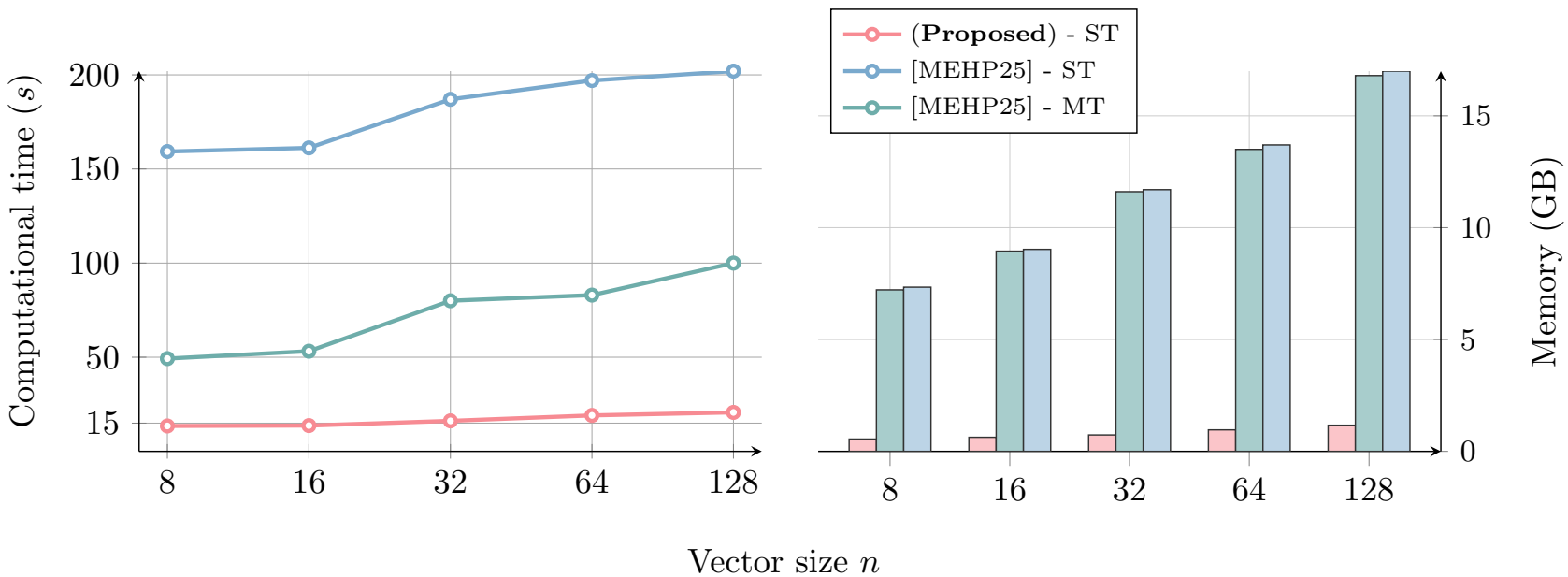
- In the first permutation-based experiment, we define sets of parameters that allow to correctly[3] sort with $\delta = 0.01$. In all experiments, the CKKS ring dimension is set to $N = 2^{16}$.

	CKKS parameters			Data parameters			
n	Δ	Circuit depth	$\log(QP)$	sigmoid$_k(x)$ scale k	sigmoid$_k(x)$ degree d_1	sinc(x) degree d_2	sinc(x) degree d_{tie}
8	2^{45}	17	1233	650	1006	59	495
16	2^{45}	17	1233	650	1006	59	495
32	2^{45}	18	1338	650	1006	119	495
64	2^{45}	19	1383	650	1006	247	495
128	2^{45}	20	1488	650	1006	495	495

[3] Namely, the error in each element of the sorted ciphertext is smaller than δ.

Experiment 1

- We assess the performance of the proposed approach with respect to the state-of-the-art work [MEHP25].



Experiment 1

- We are able to reduce the depth of the circuit and this allows to use smaller rings (in [MEHP25] they use $N = 2^{17}$, while we can safely use $N = 2^{16}$)
- We approximate sgn(x) only once in the indexing phase, while in [MEHP25] it is approximated also in the equal(x) evaluation.
- The complexity of this approach mostly depends on δ.

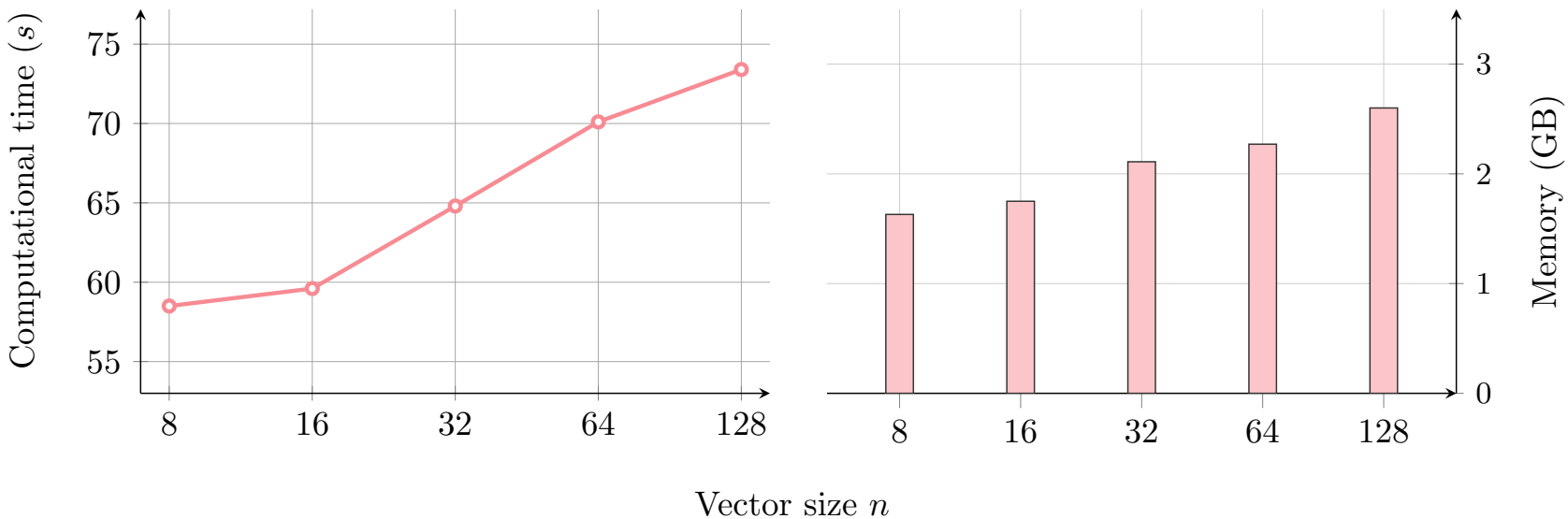
Experiment 2

- As the second experiment, we want to discretize more, so we set $\delta = 0.001$.

	CKKS parameters			Data parameters			
n	Δ	Circuit depth	$\log(QP)$	sigmoid$_k(x)$ scale k	sigmoid$_k(x)$ degree d_1	sinc(x) degree d_2	sinc(x) degree d_{tie}
8	2^{45}	21	1533	9170	16000	59	495
16	2^{45}	21	1533	9170	16000	59	495
32	2^{45}	22	1638	9170	16000	119	495
64	2^{45}	23	1683	9170	16000	247	495
128	2^{45}	24	1728	9170	16000	495	495

Experiment 2

- Results show that it is possible to sort 128 elements discretized with $\delta = 0.001$ in roughly 73 seconds, requiring less than 3 GB of memory.



Experiments - network-based

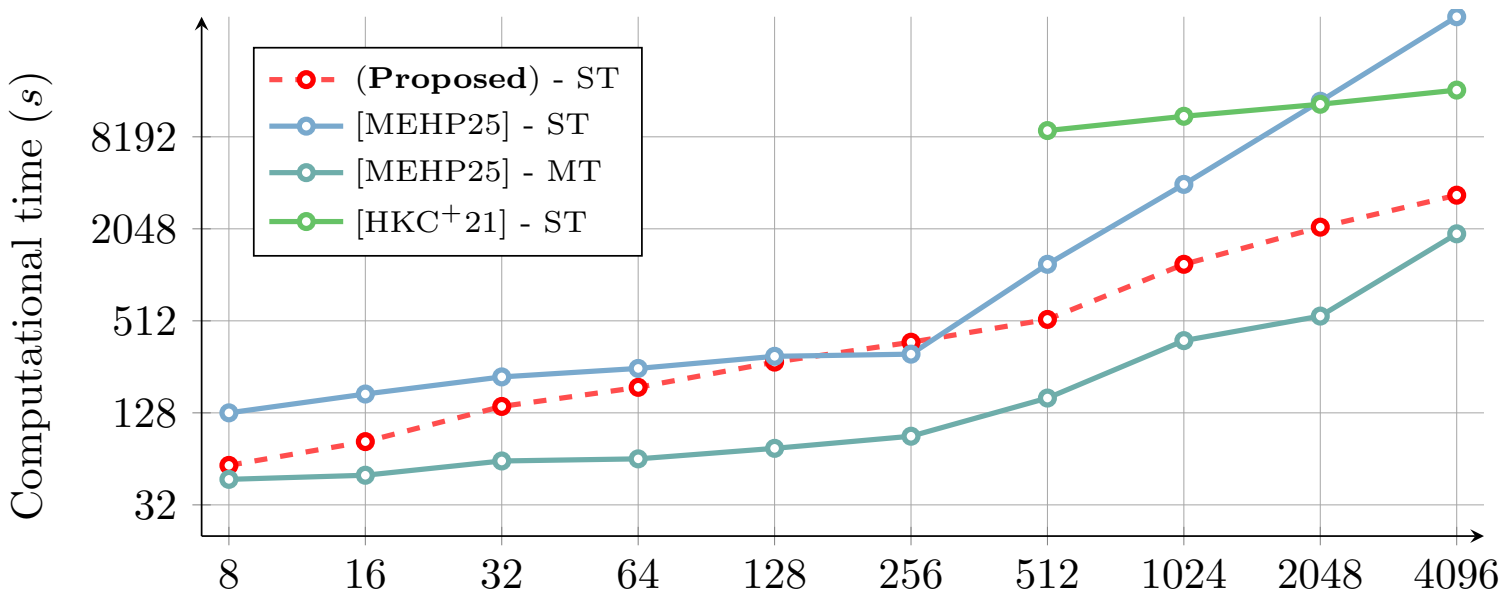
- Now we test our bitonic sorting network, with $\delta = 0.01$ and the following parameters.

	CKKS parameters				Data parameters
n	Δ	Circuit depth	$\log(QP)$	Bootstrapping depth	ReLU(x) degree d
from 8 to 4096	2^{55}	25	1765	16	495

- The complexity solely depends on the approximation of ReLU(x), and in turn on δ!

Experiment 3

- Our network is faster than the state-of-the-art in terms of CKKS sorting networks ([HKC+21]), but not quicker than permutation-based approaches.



Conclusions and open questions

- We proposed two approaches to make sorting of encrypted values faster and lighter in terms of memory and computations.
- Permutation-based:
  - We are still using an approximation of $\mathrm{sgn}(x)$... can we eventually replace it with something easier?
  - How does this compare to a TFHE based version? Could PBS allow for more accurate and fast nonlinear operations?
  - Can similar arguments be ported to a MPC setting?
- Network-based:
  - Can this approach be more efficient, considering the bootstrapping operation at each layer?

Conclusions and open questions

Thank you! Questions?

References I

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. "Improved Bootstrapping for Approximate Homomorphic Encryption". In: *EUROCRYPT '19*.

[CGKP25] Kelong Cong, Robin Geelen, Jiayi Kang, and Jeongeun Park. "Revisiting Oblivious Top-k Selection with Applications to Secure k-NN Classification". In: *SAC 2024*.

[HSJY21] Seungwan Hong, Seunghong Kim, Jiheon Choi, Younho Lee, and Jung Hee Cheon. "Efficient Sorting of Homomorphic Encrypted Data With k-Way Sorting Network". In: *IEEE Transactions on Information Forensics and Security* 16 (2021).

References II

[KPP22] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. "Approximate Homomorphic Encryption with Reduced Approximation Error". In: *CT-RSA 2022*, pp. 120–144.

[MEHP25] Federico Mazzone, Maarten Everts, Florian Hahn, and Andreas Peter. "Efficient Ranking, Order Statistics, and Sorting under CKKS". In: *USENIX Security '25*.