Jack Doerner [Northeastern U]

An Introduction to Practical Multiparty Computation

This Talk

MPC Frameworks - General Computation

Circuit Structures - Solving Specific Problems

The Memory Problem - A Perpetual Bugbear

Custom Protocols - Beyond Circuits

But not:
Theory, Protocols, Security Models

MPC History

1982	Yao's Garbled Circuits
2004	Fairplay
2016	FairplayMP, Obliv-C, ObliVM, FastGC, TASTY, SPDZ, EMP, TinyOT, ShareMind, PCF, Sharemonad, TinyOT, Fresco, Wysteria, …

Plus, many schemes that have never been implemented!

MPC Frameworks

Obliv-C

ObliVM

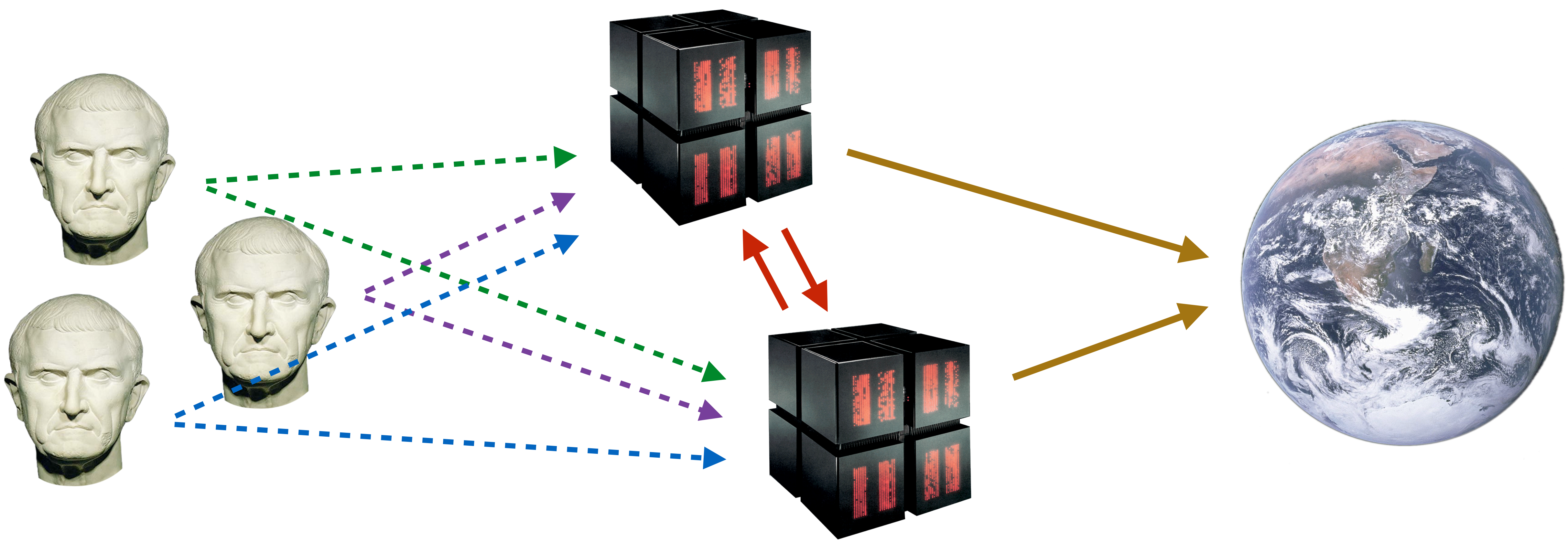
SPDZ

Sharemind

The *n* Millionaires Problem

```
function nmillionaires(array balances[n], var n):
    var winning_millionaire = -1
    var winning_balance = -1
    for ii from 1 to n:
        if balances[ii] > winning_balance:
            winning_balance = balances[ii]
            winning_millionaire = ii
    return winning_millionaire
```

The *n* Millionaires Problem



1. Millionaires additively share their inputs
2. Computation authorities engage in MPC
3. Result is revealed

MPC Frameworks

Obliv-C

ObliVM

SPDZ

Sharemind

Obliv-C

- Protocol: Yao's Garbled Circuits (others possible)
- Language type: C-compatible DSL
- Philosophy: Minimalism and expressiveness
  Only one additional keyword over C
- Raw speed: 3M+ AND gates per second reported
- Unique feature: Compiled; C-compatible

[ZE15]

Obliv-C

```c
#include <obliv.oh>

int nmillionaires(int * inputs, int number_of_millionaires) {

    obliv int winning_millionaire = -1;
    obliv int winning_balance = -1;

    for (int ii = 0; ii < number_of_millionaires; ii++) {

        obliv int current_millionaire_balance = feedOblivInt(inputs[ii], 1);
        current_millionaire_balance -= feedOblivInt(inputs[ii], 2);

        obliv if (current_millionaire_balance > winning_balance) {
            winning_millionaire = ii;
            winning_balance = current_millionaire_balance;
        }
    }

    int result;
    revealOblivInt(&result, winning_millionaire, 0);
    return result;
}
```

Obliv-C

Language features not seen

- obliv functions
- ~obliv
- intelligent typecasting

Scalability Example: Secure Stable Matching

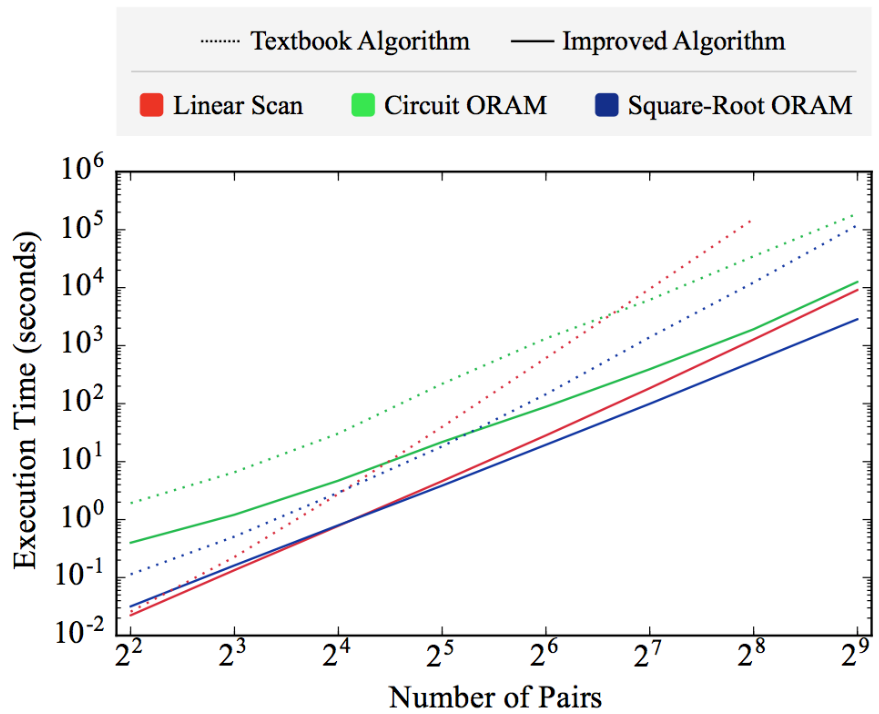


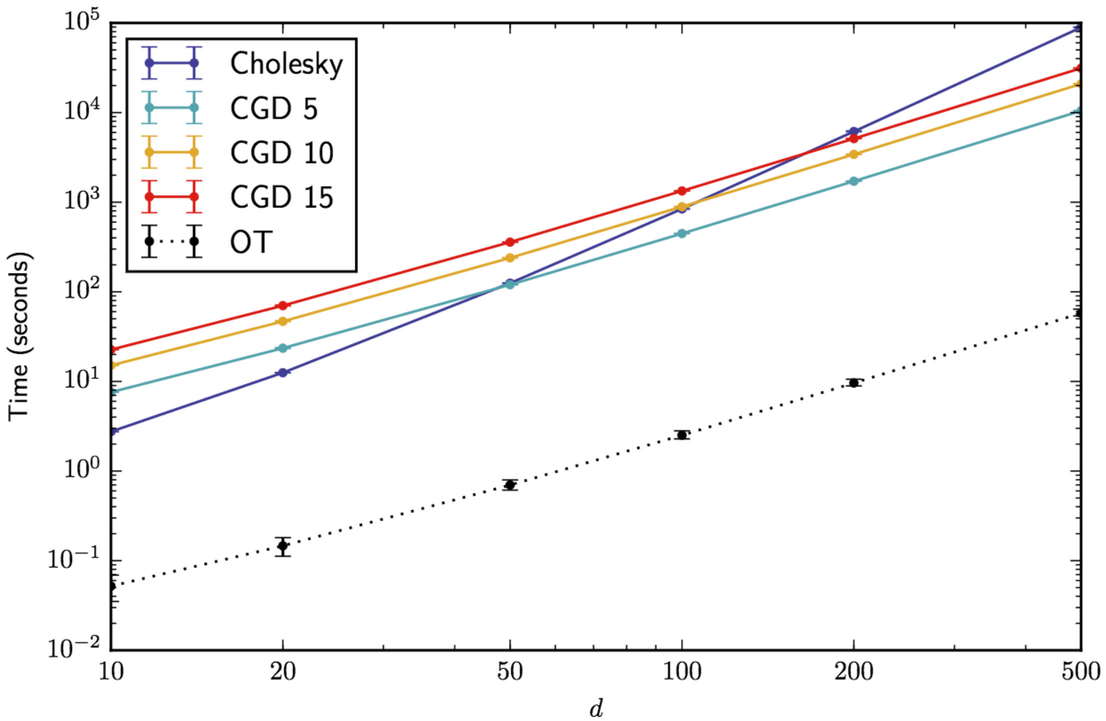
Figure 8: **Secure Gale-Shapley Execution Time vs Pair Count.** Values are mean wall-clock times in seconds for full protocol execution including initialization. For benchmarks of 4–64 pairs, we collected 30 samples; for 128–256 pairs we collected three samples; and for 512 pairs we collected one sample.

Algorithm Phase	Time (hours)	Billions of Non-Free Gates
Sharing	1.07	18.14
Setup	1.60	29.65
Permutation	0.56	6.56
Proposal/Rejection	15.01	172.52
Total	18.22	226.87

Table 2: **Secure Roth-Peranson NRMP Benchmark Results.** For this benchmark we set $n = 35476$, $m = 4836$, $q = 15$, $r = 120$, and $s = 12$. These parameters are intended to be representative of the match performed by the National Residency Matching Program.

[DEs16]

Scalability Example: Linear System Solving



d	OT	Cholesky	CGD 5	CGD 10	CGD 15
10	0.052	2.751	7.585	15.099	22.608
20	0.146	12.507	23.492	46.798	70.089
50	0.698	124.918	120.002	239.209	358.467
100	2.509	841.372	446.744	890.811	1334.814
200	9.608	6144.301	1713.717	3417.499	5121.407
500	57.791	89 193.308	10 474.579	20 888.350	31 300.942

Figure 6: Comparison between different methods for solving linear systems: Running time (seconds) of our Cholesky and CGD (with 5, 10, and 15 iterations) implementations as a function of input dimension. While Cholesky is faster than CGD for lower values of d, it is quickly overtaken by the latter as d increases. This shows that for high-dimensional data, iterative methods are preferable in terms of computation time. The time spent running oblivious transfers is also shown, and corresponds to a small fraction of the running time.

[GSBRDZE16]

MPC Frameworks

Obliv-C

ObliVM

SPDZ

Sharemind

ObliVM

- Protocol: Yao's Garbled Circuits
- Language type: Java/C++ style DSL
- Philosophy: Common operations are first-class language constructs. Includes everything and the kitchen sink.
- Raw speed: 700K AND gates per second reported or 1.8M with preprocessing

[LWNHS15]

ObliVM

```
secure int nmillionaires@n(secure int[n] inputs) {
    public int number_of_millionaires = n;

    secure int winning_millionaire = -1;
    secure int winning_balance = -1;

    for (int ii = 0; ii < number_of_millionaires; ii++) {

        secure int current_millionaire_balance = inputs[ii];

        if (current_millionaire_balance > winning_balance) {
            winning_millionaire = ii;
            winning_balance = current_millionaire_balance;
        }
    }

    return winning_millionaire;
}
```

ObliVM

Language features not seen

- phantom functions
- shared random types
- bounded loops
- hinted loop-coalescing
- automatic ORAM
- built-in map + reduce
- C-style structs

MPC Frameworks

Obliv-C

ObliVM

SPDZ

Sharemind

SPDZ

- Protocol: *n*-party Linear Secret Sharing + SHE
- No Language: programmed via python library calls
- Raw Speed (2PC Online): 358K multiplications/second
  (2PC Offline): 4800 multiplications/second
- Unique feature: Covert or Malicious security against dishonest majority

[DPSZ11] [DKLPSS12] [KOS16]

SPDZ

```
from Compiler.types import *
from Compiler.util import *

def nmillionaires(inputs, number_of_millionaires):

    winning_millionaire = sint(-1)
    winning_balance = sint(-1)

    for ii in range(number_of_millionaires):

        current_millionaire_balance = sint.get_raw_input_from(0)
        current_millionaire_balance -= sint.get_raw_input_from(1)

        if_then(current_millionaire_balance > winning_balance)
        winning_millionaire = sint(ii)
        winning_balance = current_millionaire_balance
        end_if()

    return winning_millionaire.reveal()
```

SPDZ

```python
from Compiler.types import *
from Compiler.util import *

def nmillionaires(inputs, number_of_millionaires):

    winning_millionaire = sint(-1)
    winning_balance = sint(-1)

    for ii in range(number_of_millionaires):

        current_millionaire_balance = sint.get_raw_input_from(0)
        current_millionaire_balance -= sint.get_raw_input_from(1)

        overwrite = current_millionaire_balance > winning_balance
        winning_millionaire = overwrite.if_else(winning_millionaire, sint(ii))
        winning_balance = overwrite.if_else(winning_balance, current_millionaire_balance)

    return winning_millionaire.reveal()
```

SPDZ

Language features not seen

- Native GF(2^n) types
- Many bits of syntax

MPC Frameworks

Obliv-C

ObliVM

SPDZ

Sharemind

sharemind

- A Commercial “Application Server Platform” (free for researchers). Similar to Java or .NET
- Originally used a 3-party semi-honest protocol; now includes SPDZ, YGC, three-party malicious
- Programming environments:
  - C/C++ library calls
  - SecreC, a C-like DSL
  - Rmind, an R-inspired statistical analysis language
- Unique feature: vector optimized

[sharemind.cyber.ee] [BLW08] [J10] [BKLS14]

sharemind

```
public int nmillionaires(private int inputs[], public int number_of_millionaires) {

    private int winning_millionaire = -1;
    private int winning_balance = -1;

    for (public int ii = 0; ii < number_of_millionaires; ii++) {

        private int current_millionaire_balance = inputs[ii];

        private bool overwrite = current_millionaire_balance > winning_balance;
        winning_millionaire = overwrite*ii + (1-overwrite)*winning_millionaire;
        winning_balance = overwrite*current_millionaire_balance + (1-overwrite)*winning_balance;
    }

    return declassify(winning_millionaire);
}
```

sharemind

Scalability Example: Tax Fraud Detection

Table 2: The three regional instance deployments used, modelling one or many cloud providers

Regions	Client instance	Computing instances	Latency (round-trip)
1	us-east – c3.8xlarge	us-east – 12x c3.8xlarge	< 0.1ms between all nodes
2	eu-west – c3.8xlarge	eu-west – 8x c3.8xlarge eu-central – 4x c3.8xlarge	< 0.1ms between eu-west nodes 19ms – eu-west, eu-central
3	us-east – c3.8xlarge	us-east – 4x c3.8xlarge us-west – 4x c3.8xlarge eu-west – 4x c3.8xlarge	77ms – us-east, us-west 133ms – us-west, eu-west 76ms – us-east, eu-west

Table 3: Descriptions of the three data sets used in the experiments

No. of companies	No. of transaction partner pairs	Total no. of transactions	Total raw XML data size
20 000	200 000	25 000 000	8.61GB
40 000	400 000	50 000 000	17.26GB
80 000	800 000	100 000 000	34.51GB

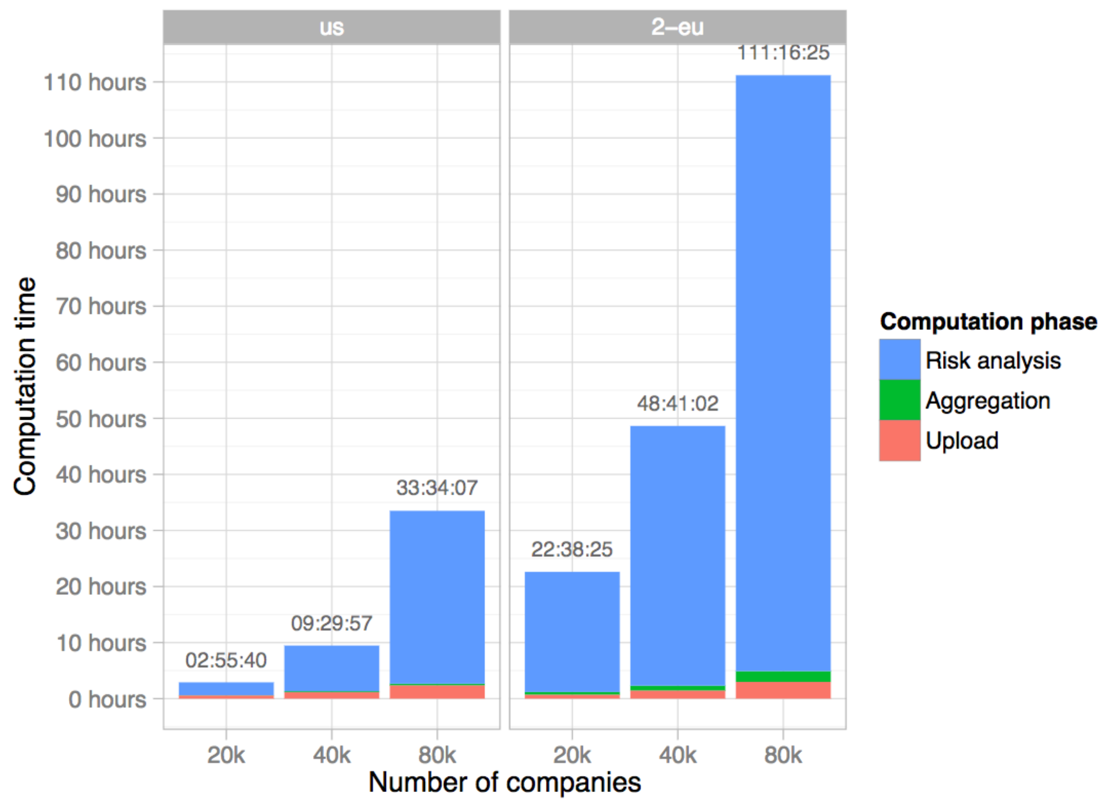
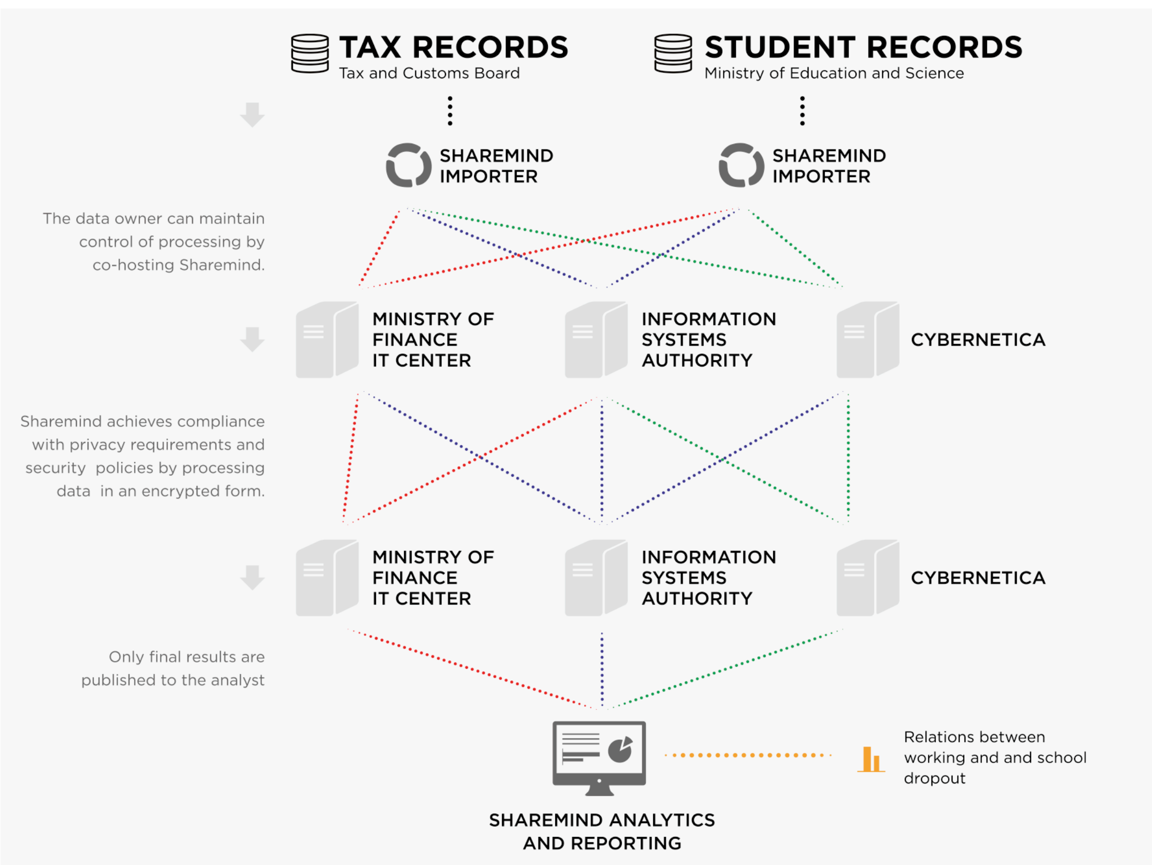


Figure 6: Running times of the computation using slower risk analysis algorithm that does not rely on admissible leakage

[BJSV15]

sharemind

Scalability Example: Population-scale Statistical Studies



We generated two sets of test databases: a smaller set for correctness testing that contained 354 education records and 8,201 tax records (test set A); and a larger set that was comparable in size to the expected real dataset (test set B) with 831,424 education records and 16,205,641 tax records. We used the larger dataset to test performance on a SHAREMIND installation in a local area network. The final real-world data imported by the data owners contained 623,361 education records and 10,495,760 tax records.

ETL script	**Test set B (Laboratory)**	**Real data (Production)**
(1) Aggregation of education data	25 min	2 h
(2) Aggregation of tax data (monthly income)	18 h 10 min	221 h 55 min
(3) Aggregation of tax data (average yearly income)	1 h 55 min	15 h 14 min
(4) Joining the two datasets	32 min	4 h 15 min
(5) Compiling the analysis table (shifting)	39 h 3 min	141 h 11 min
Total ETL time	60 h 5 min	384 h 35 min

Table 1. Running times of the privacy-preserving ETL scripts on test set B in a laboratory environment and the final imported data in the production environment.

[sharemind.cyber.ee] [BKKRST16]

MPC Frameworks

	Obliv-C	ObliVM	SPDZ	Sharemind
Protocol	Yao's GC (others possible)	Yao's GC	*n*-party LSS + SHE	Multiple
Programming Paradigm	C-compatible DSL	Java-like DSL	Python Library	"Application Server Platform"
Philosophy	Minimalism, Be like C	Do the sensible thing	No front-end Language	Commercial, Ever-growing
Advantages	Is like C, Compiled, fast	Many language features	Malicious or Covert Security	Diverse Toolset, Vector-optimized
Disadvantages	Is like C, No Floating Point	Complicated Syntax	Precomputation, Leaky Abstraction	Commercial

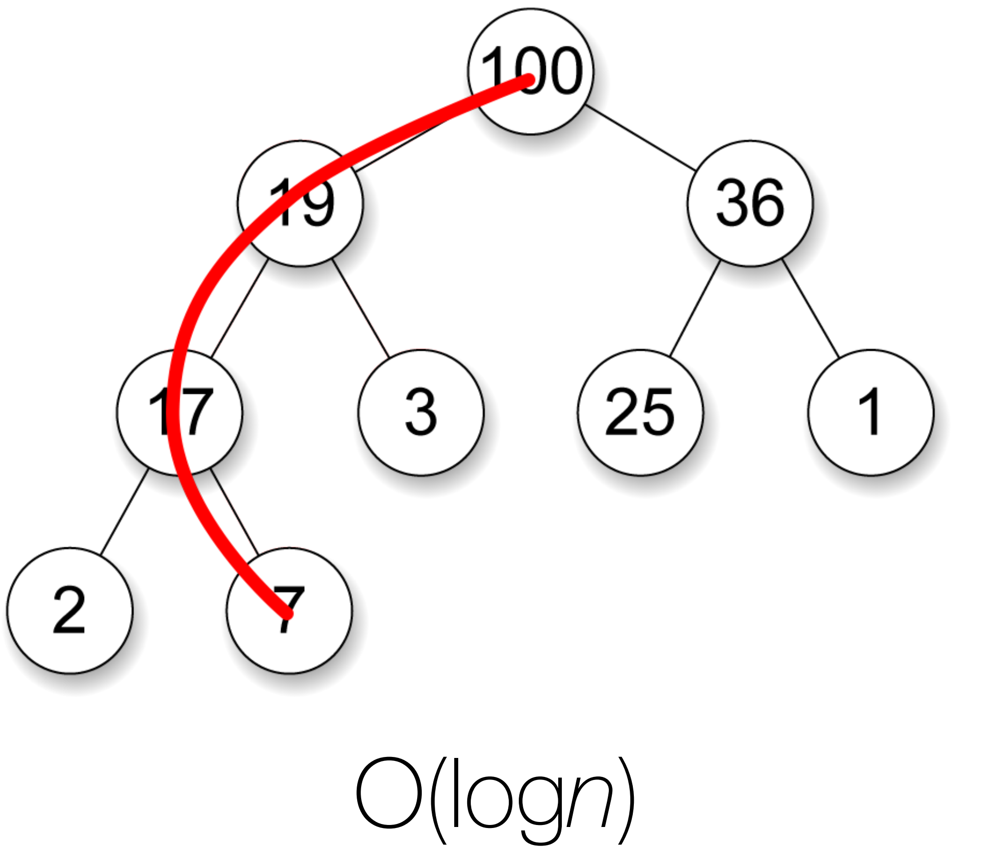
Circuit Structures

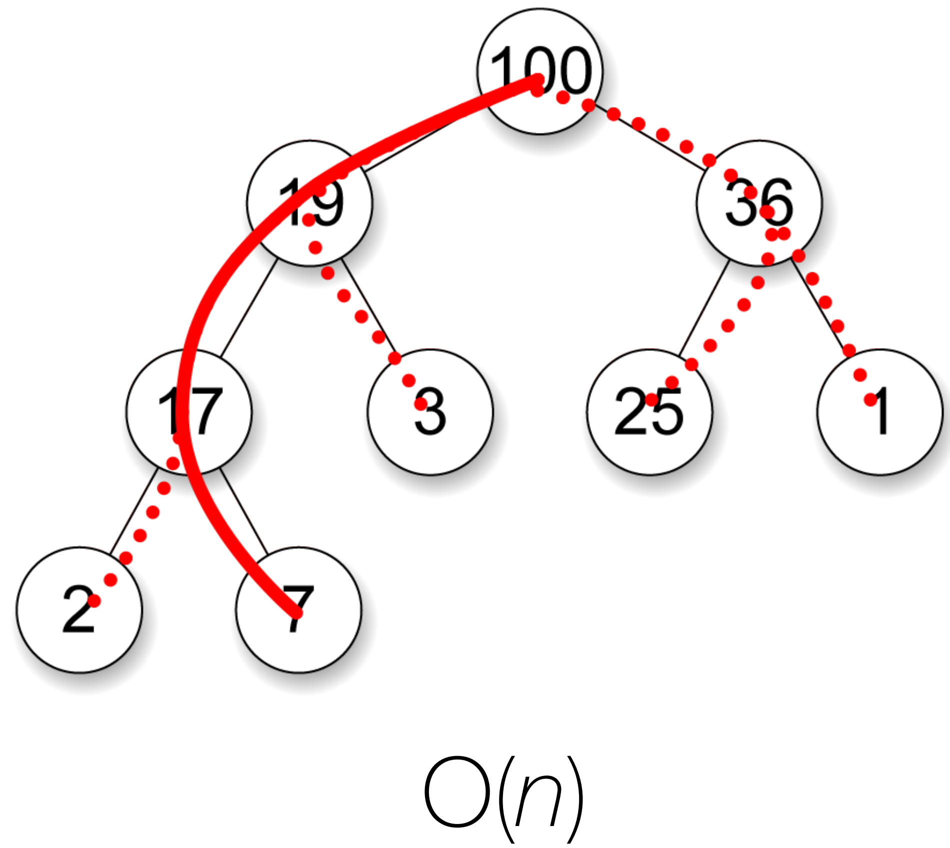
Circuit Structures

```
function nmillionaires(array balances[n], var n, var x):
    balances = sort(balances)
    return balances[:x]
```

Seems simple enough, right?
But how do we sort?

“Standard” Sorts





Heapsort’s data-dependent branches make it inefficient
Quicksort is totally unsuitable

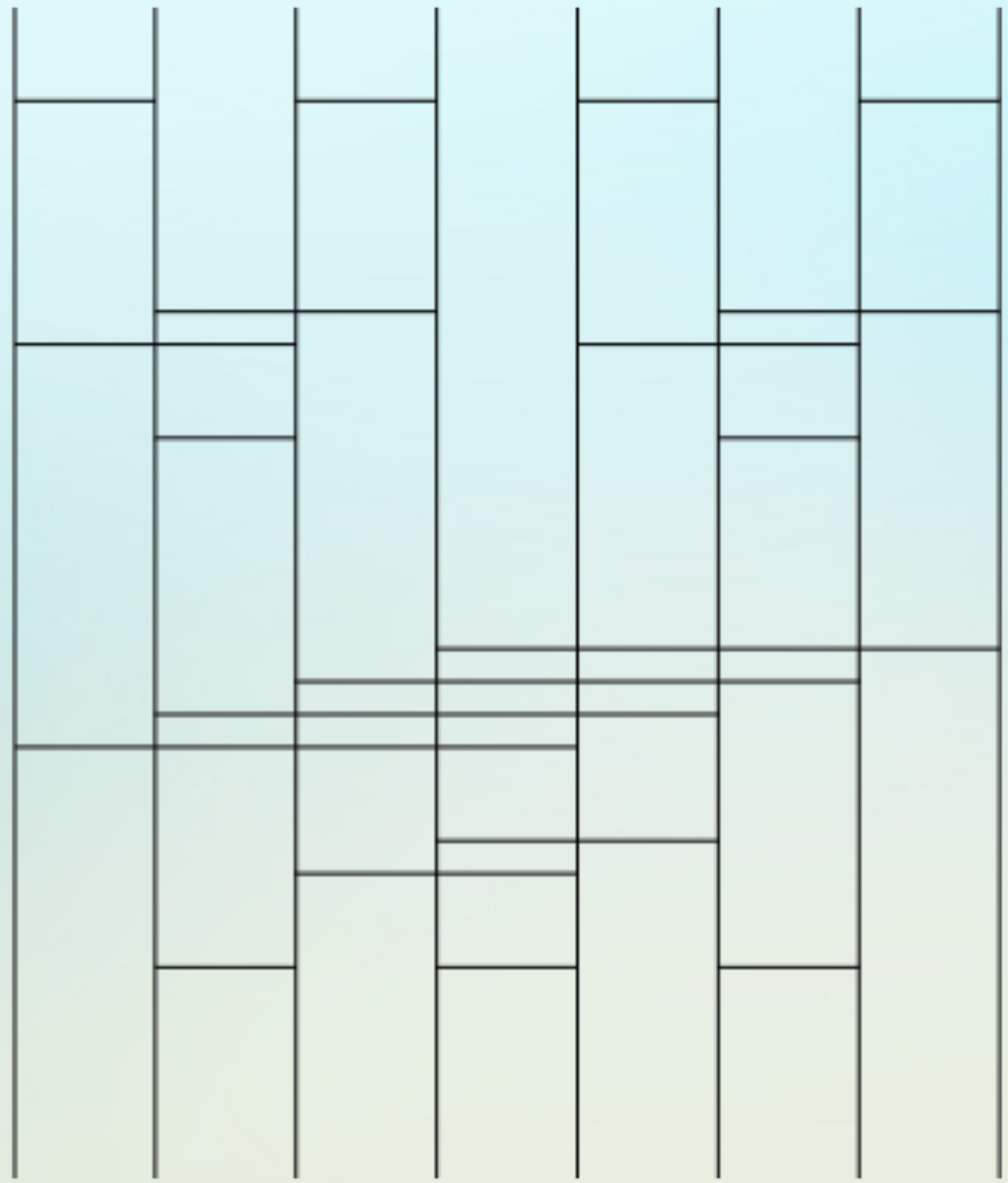
Batcher’s Mergesort

```
function batcher_sort(array input[n], var n):
    lower_half_sorted = batcher_sort(input[0:n/2])
    upper_half_sortd = batcher_sort(input[n/2:n])
    result = batcher_merge(lower_half_sorted, upper_half_sortd)
    return input
```

Batcher’s Mergesort

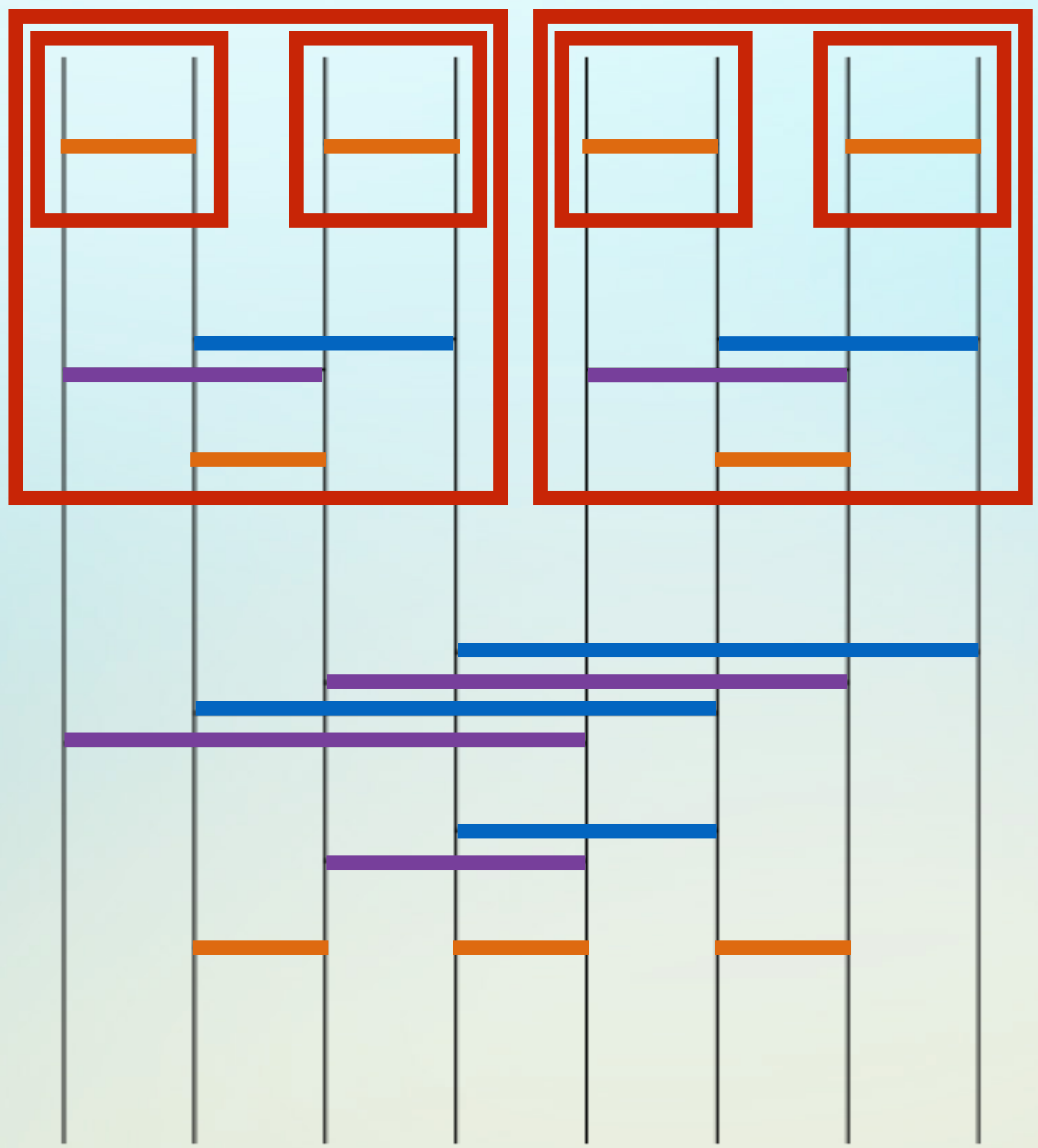
```
function batcher_sort(array input[n], var n):
    lower_half_sorted = batcher_sort(input[0:n/2])
    upper_half_sortd = batcher_sort(input[n/2:n])
    result = batcher_merge(lower_half_sorted, upper_half_sortd)
    return input

function batcher_merge(array lower_half[n], array upper_half[n]):
    lower_evens = even_elements(lower_half)
    upper_evens = even_elements(upper_half)
    lower_odds = odd_elements(lower_half)
    upper_odds = odd_elements(upper_half)
    merged_evens = batcher_merge(lower_evens, upper_evens)
    merged_odds = batcher_merge(lower_odds, upper_odds)
    merged_all = interleave(merged_evens, merged_odds)
    result = compare_and_swap_neighbors(merged_all)
    return result
```

A sorting algorithm with
no data-dependent branches



Recursively
Sort Lower Half

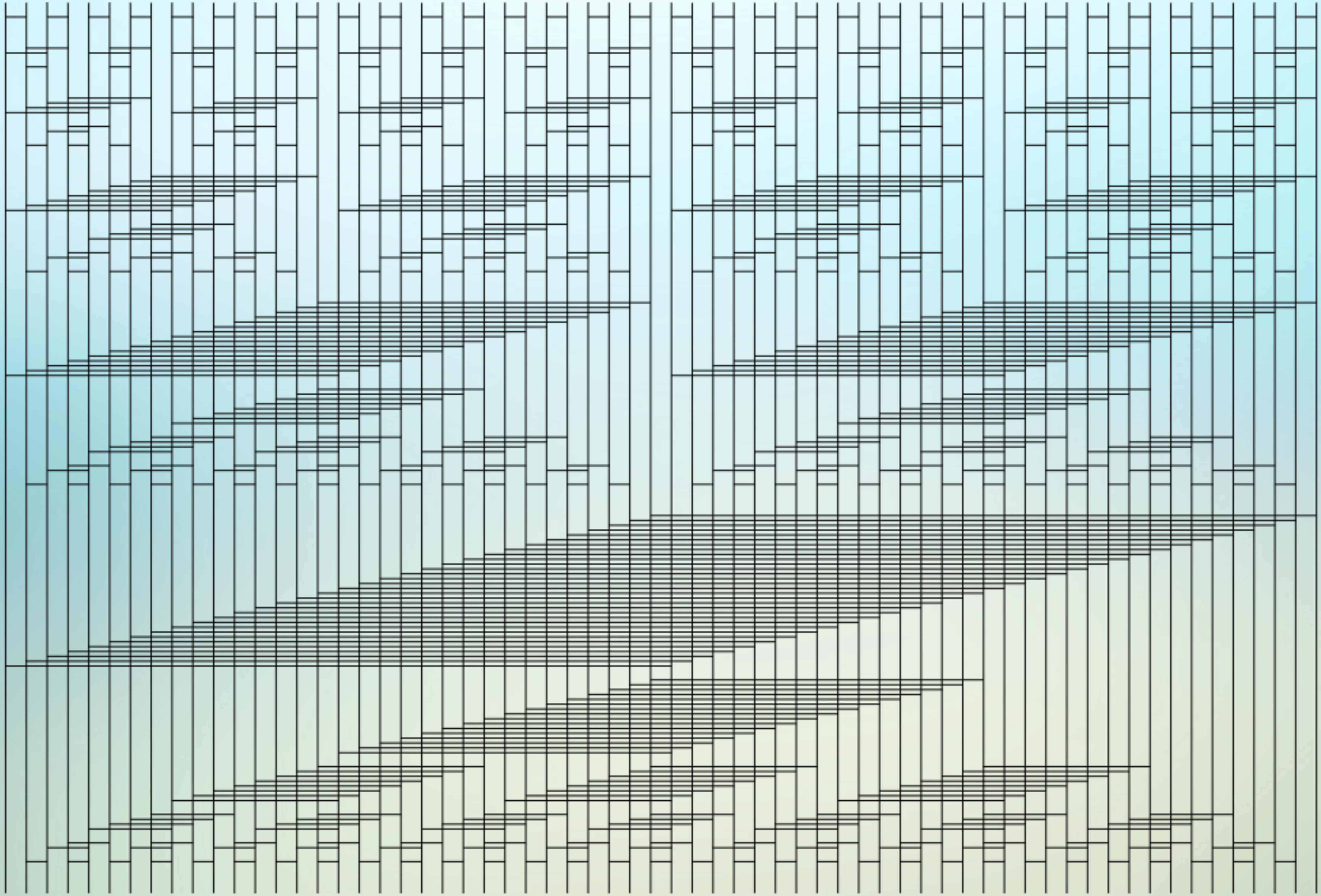
Recursively
Sort Upper Half



Merge Even
Rows

Merge Odd
Rows

Compare Neighbor
Elements



Circuit Structures

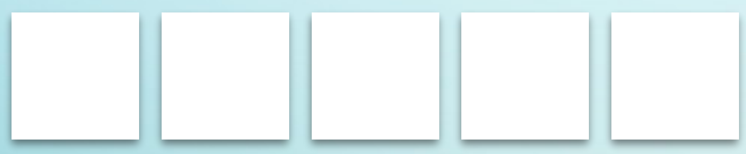
Batcher Merge	$O(n\log n)$	[B68]
Batcher Odd-Even Mergesort	$O(n\log^2 n)$	[B68]
AKS Sorting Network	$O(n\log n)$	[AKS83]
Waksman Permutation Network	$O(n\log n)$	[W68]

Circuit Structures

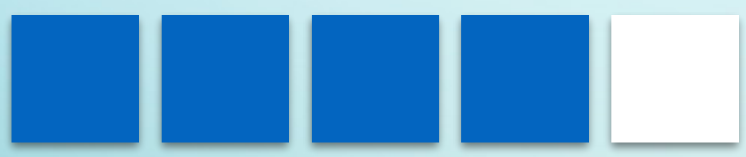
Batcher Merge	$O(n\log n)$	[B68]
Batcher Odd-Even Mergesort	$O(n\log^2 n)$	[B68]
AKS Sorting Network	$O(n\log n)$	[AKS83]
Waksman Permutation Network	$O(n\log n)$	[W68]

The Memory Problem

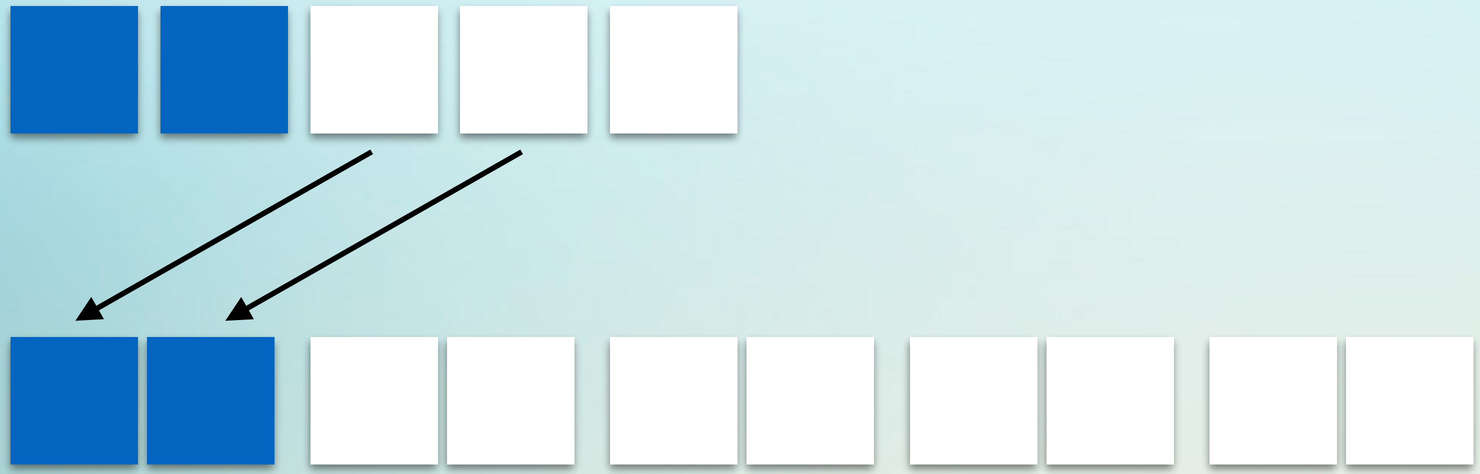
Oblivious Stack



Oblivious Stack

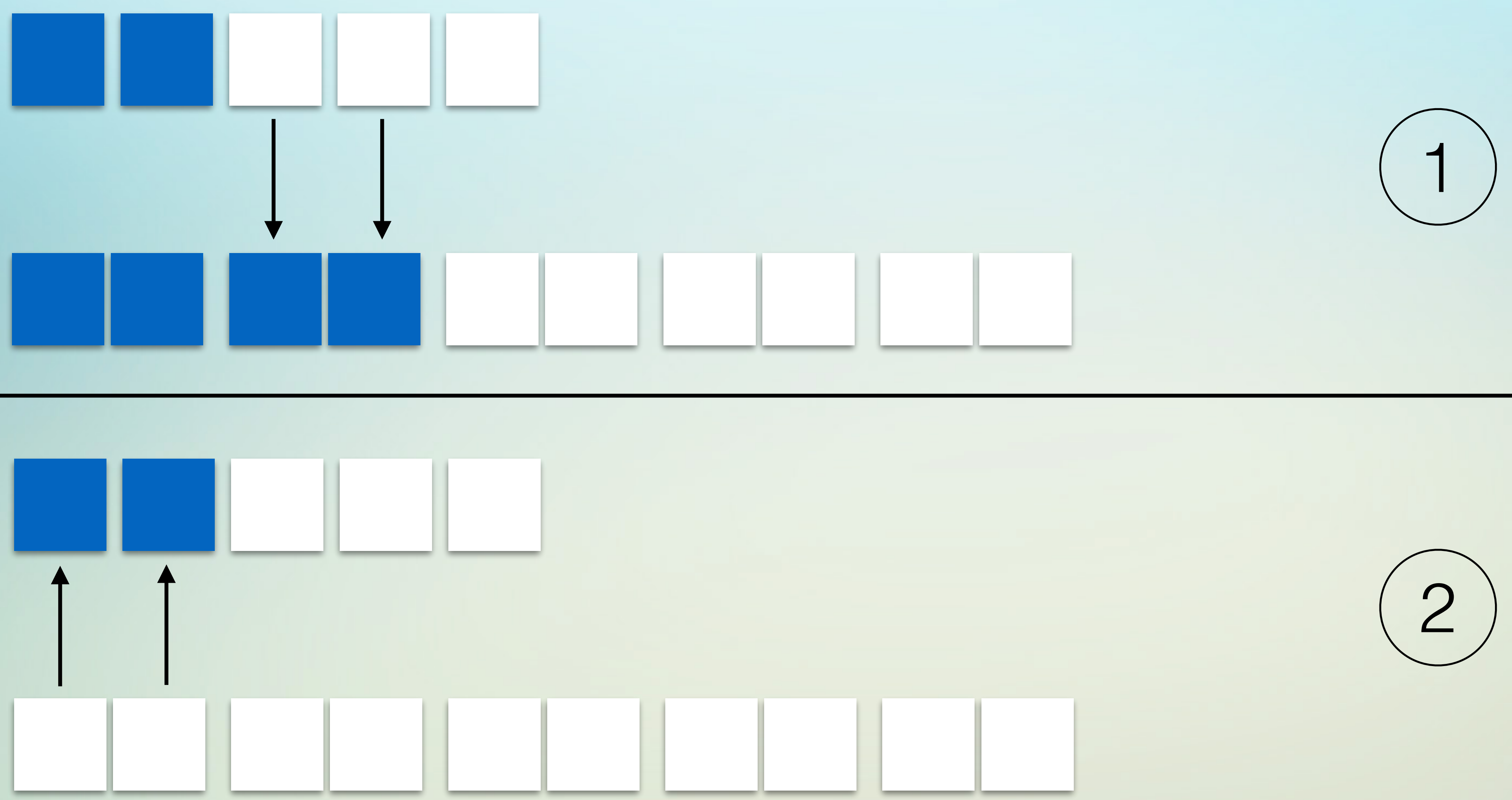


Oblivious Stack



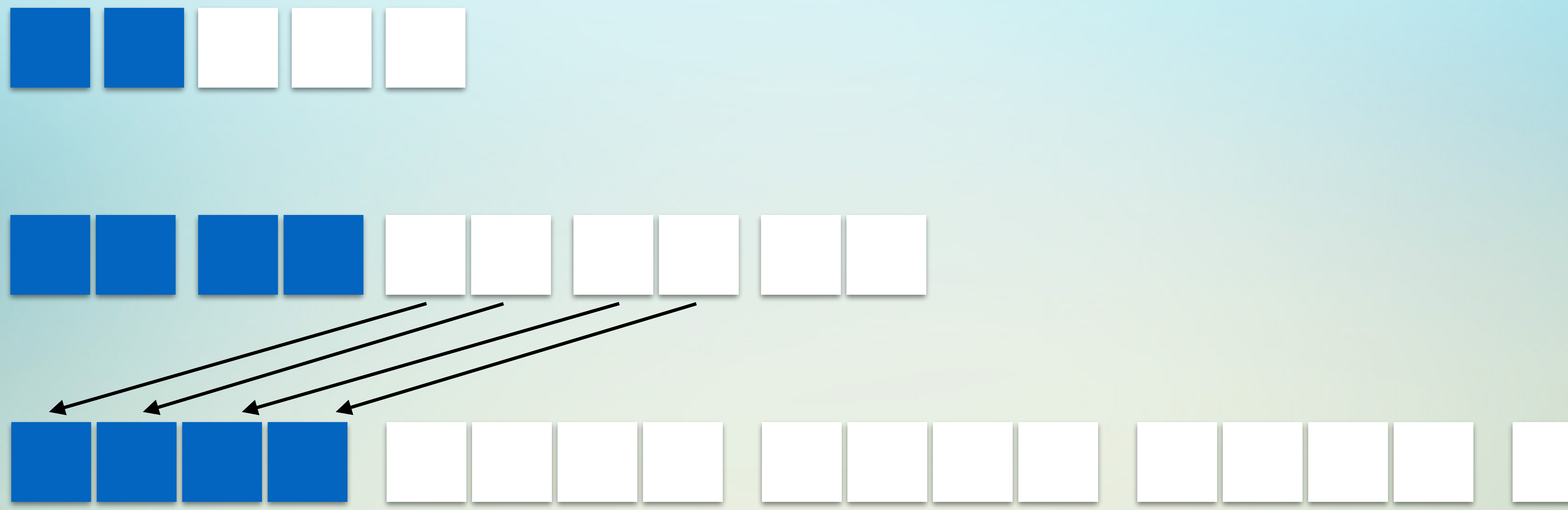
Oblivious Stack

Oblivious Stack

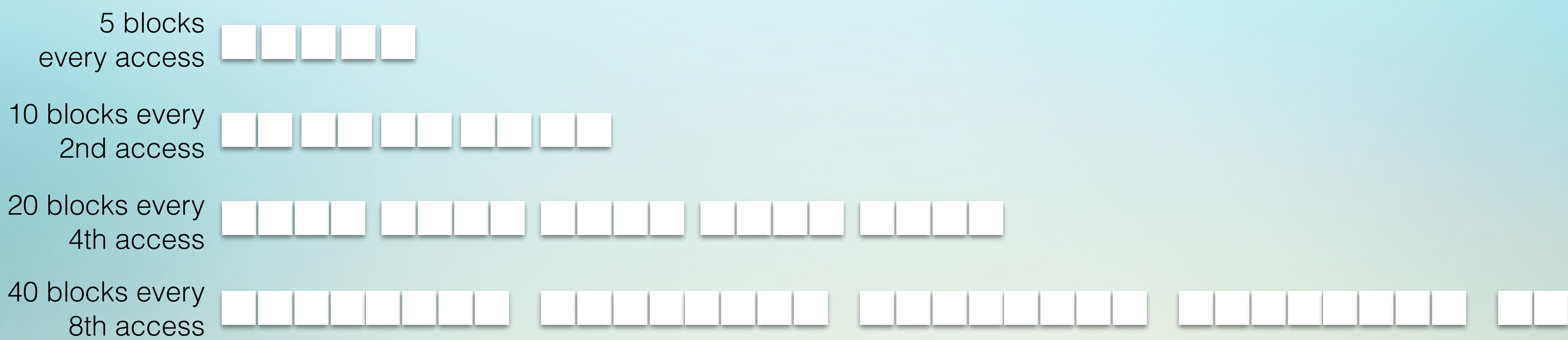


Oblivious Stack

Oblivious Stack



Oblivious Stack



Amortized cost: 5 blocks per layer per access

Layers: $O(\log n)$

Sublinear-time Memories

Stack, Queue	$O(\log n)$	[ZE13]
Square-root ORAM	$O(\mathrm{sqrt}(n\log^3 n))$	[ZWRGDEK15]
Tree ORAM (Circuit, Path)	$O(\log^3 n)$	[SDSFRYD13] [WCS15]
Algorithm-Specific	O(?)	[BSA13] [DEs16]

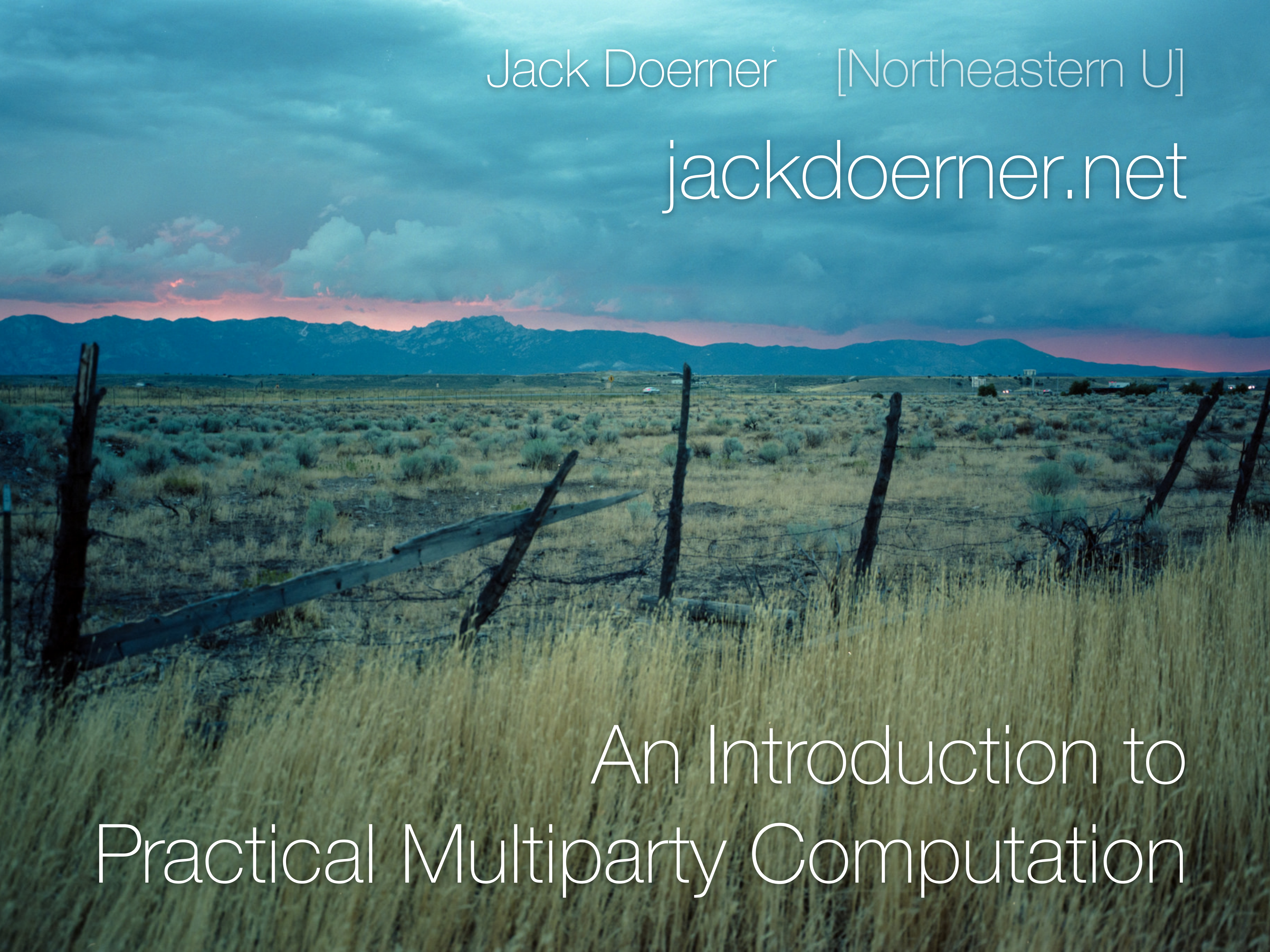
Sublinear-time Memories

Stack, Queue	O(log$n$)	[ZE13]
Square-root ORAM	O(sqrt($n$log^3n))	[ZWRGDEK15]
Tree ORAM (Circuit, Path)	O(log^3n)	[SDSFRYD13] [WCS15]
Algorithm-Specific	O(?)	[BSA13] [DEs16]

Custom Protocols

MPC Frameworks

Obliv-C	oblivc.org
ObliVM	oblivm.com
SPDZ	www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ
Sharemind	sharemind.cyber.ee



Jack Doerner [Northeastern U]

jackdoerner.net

An Introduction to Practical Multiparty Computation