An Introduction to Practical Multiparty Computation

Jack Doerner  [Northeastern U]
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

<table>
<thead>
<tr>
<th>Year</th>
<th>Scheme(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1982</td>
<td>Yao's Garbled Circuits</td>
</tr>
<tr>
<td>2004</td>
<td>Fairplay</td>
</tr>
<tr>
<td>2016</td>
<td>FairplayMP, Obliv-C, ObliVM, FastGC, TASTY, SPDZ, EMP, TinyOT, ShareMind, PCF, Sharemonad, TinyOT, Fresco, Wysteria, …</td>
</tr>
</tbody>
</table>

Plus, many schemes that have never been implemented!
MPC Frameworks

Obliv-C   ObliVM
SPDZ      Sharemind
The $n$ Millionaires Problem

```javascript
1  function nMillionaires(array balances[n], var n):
2      var winning_millionaire = -1
3      var winning_balance = -1
4      for ii from 1 to n:
5          if balances[ii] > winning_balance:
6              winning_balance = balances[ii]
7              winning_millionaire = ii
8      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]
```c
#include <obliv.h>

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;
}
```
Language features not seen

- obliv functions
- ~obliv
- intelligent typecasting
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.

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.

<table>
<thead>
<tr>
<th>Algorithm Phase</th>
<th>Time (hours)</th>
<th>Billions of Non-Free Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing</td>
<td>1.07</td>
<td>18.14</td>
</tr>
<tr>
<td>Setup</td>
<td>1.60</td>
<td>29.65</td>
</tr>
<tr>
<td>Permutation</td>
<td>0.56</td>
<td>6.56</td>
</tr>
<tr>
<td>Proposal/Rejection</td>
<td>15.01</td>
<td>172.52</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>18.22</td>
<td>226.87</td>
</tr>
</tbody>
</table>
Scalability Example: Linear System Solving

![Graph showing time (seconds) vs. input dimension (d) for different methods]

<table>
<thead>
<tr>
<th>d</th>
<th>OT</th>
<th>Cholesky</th>
<th>CGD 5</th>
<th>CGD 10</th>
<th>CGD 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.052</td>
<td>2.751</td>
<td>7.585</td>
<td>15.099</td>
<td>22.608</td>
</tr>
<tr>
<td>20</td>
<td>0.146</td>
<td>12.507</td>
<td>23.492</td>
<td>46.798</td>
<td>70.089</td>
</tr>
<tr>
<td>50</td>
<td>0.698</td>
<td>124.918</td>
<td>120.002</td>
<td>239.209</td>
<td>358.467</td>
</tr>
<tr>
<td>100</td>
<td>2.509</td>
<td>841.372</td>
<td>446.744</td>
<td>890.811</td>
<td>1334.814</td>
</tr>
<tr>
<td>200</td>
<td>9.608</td>
<td>6144.301</td>
<td>1713.717</td>
<td>3417.499</td>
<td>5121.407</td>
</tr>
<tr>
<td>500</td>
<td>57.791</td>
<td>89193.308</td>
<td>10474.579</td>
<td>20888.350</td>
<td>31300.942</td>
</tr>
</tbody>
</table>

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.
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]
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]
```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)

        if current_millionaire_balance > winning_balance:
            winning_millionaire = sint(ii)
            winning_balance = current_millionaire_balance
            end_if()

    return winning_millionaire.reveal()
```
```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
• 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]
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);
}
Scalability Example: Tax Fraud Detection

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

<table>
<thead>
<tr>
<th>Regions</th>
<th>Client instance</th>
<th>Computing instances</th>
<th>Latency (round-trip)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>us-east – c3.8xlarge</td>
<td>us-east – 12x c3.8xlarge</td>
<td>&lt; 0.1ms between all nodes</td>
</tr>
<tr>
<td>2</td>
<td>eu-west – c3.8xlarge</td>
<td>eu-west – 8x c3.8xlarge, eu-central – 4x c3.8xlarge</td>
<td>&lt; 0.1ms between eu-west nodes, 19ms – eu-west, eu-central</td>
</tr>
<tr>
<td>3</td>
<td>us-east – c3.8xlarge</td>
<td>us-east – 4x c3.8xlarge, us-west – 4x c3.8xlarge, eu-west – 4x c3.8xlarge</td>
<td>77ms – us-east, us-west, 133ms – us-west, eu-west, 76ms – us-east, eu-west</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>No. of companies</th>
<th>No. of transaction partner pairs</th>
<th>Total no. of transactions</th>
<th>Total raw XML data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 000</td>
<td>200 000</td>
<td>25 000 000</td>
<td>8.61GB</td>
</tr>
<tr>
<td>40 000</td>
<td>400 000</td>
<td>50 000 000</td>
<td>17.26GB</td>
</tr>
<tr>
<td>80 000</td>
<td>800 000</td>
<td>100 000 000</td>
<td>34.51GB</td>
</tr>
</tbody>
</table>

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

[BJSV15]
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.

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

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.
# MPC Frameworks

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

**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

```
1  function nmillionaires(array balances[n], var n, var x):
2       balances = sort(balances)
3       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.

Heapsort: $O(\log n)$

Quicksort: $O(n)$
Batcher’s Mergesort

1. function batcher_sort(array input[n], var n):
2.   lower_half_sorted = batcher_sort(input[0:n/2])
3.   upper_half_sorted = batcher_sort(input[n/2:n])
4.   result = batcher_merge(lower_half_sorted, upper_half_sorted)
5.   return input
Batcher’s Mergesort

A sorting algorithm with no data-dependent branches

```python
1 function batcher_sort(array input[n], var n):
2     lower_half_sorted = batcher_sort(input[0:n/2])
3     upper_half_sorted = batcher_sort(input[n/2:n])
4     result = batcher_merge(lower_half_sorted, upper_half_sorted)
5     return input

6 function batcher_merge(array lower_half[n], array upper_half[n]):
7     lower_evens = even_elements(lower_half)
8     upper_evens = even_elements(upper_half)
9     lower_odds = odd_elements(lower_half)
10    upper_odds = odd_elements(upper_half)
11    merged_evens = batcher_merge(lower_evens, upper_evens)
12    merged_odds = batcher_merge(lower_odds, upper_odds)
13    merged_all = interleave(merged_evens, merged_odds)
14    result = compare_and_swap_neighbors(merged_all)
15    return result
```
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

<table>
<thead>
<tr>
<th>Structure</th>
<th>Complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batcher Merge</td>
<td>$O(n \log n)$</td>
<td>[B68]</td>
</tr>
<tr>
<td>Batcher Odd-Even Mergesort</td>
<td>$O(n \log^2 n)$</td>
<td>[B68]</td>
</tr>
<tr>
<td>AKS Sorting Network</td>
<td>$O(n \log n)$</td>
<td>[AKS83]</td>
</tr>
<tr>
<td>Waksman Permutation Network</td>
<td>$O(n \log n)$</td>
<td>[W68]</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Complexity</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack, Queue</td>
<td>$O(\log n)$</td>
<td>[ZE13]</td>
</tr>
<tr>
<td>Square-root ORAM</td>
<td>$O(\sqrt{n \log^3 n})$</td>
<td>[ZWRGDEK15]</td>
</tr>
<tr>
<td>Tree ORAM (Circuit, Path)</td>
<td>$O(\log^3 n)$</td>
<td>[SDSFRYD13]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[WCS15]</td>
</tr>
<tr>
<td>Algorithm-Specific</td>
<td>$O(?)$</td>
<td>[BSA13]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[DEs16]</td>
</tr>
<tr>
<td></td>
<td>Time Complexity</td>
<td>References</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Stack, Queue</td>
<td>$O(\log n)$</td>
<td>[ZE13]</td>
</tr>
<tr>
<td>Square-root ORAM</td>
<td>$O(\sqrt{n \log^3 n})$</td>
<td>[ZWRGDEK15]</td>
</tr>
<tr>
<td>Tree ORAM (Circuit, Path)</td>
<td>$O(\log^3 n)$</td>
<td>[SDSFRYD13], [WCS15]</td>
</tr>
<tr>
<td>Algorithm-Specific</td>
<td>$O(?)$</td>
<td>[BSA13], [DEs16]</td>
</tr>
</tbody>
</table>
Custom Protocols
MPC Frameworks

Obliv-C  oblivc.org

ObliVM  oblivvm.com

SPDZ  www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ

Sharemind  sharemind.cyber.ee
An Introduction to Practical Multiparty Computation

Jack Doerner
Northeastern U
jackdoerner.net