(De)Motivating Application: “Genetic Dating”

Alice

Bob

WARNING! Don’t Reproduce

WARNING! Don’t Reproduce

Genome Compatibility Protocol
Cost to sequence human genome

Moore’s Law prediction (halve every 18 months)

genome.gov/sequencingcosts
Cost to sequence human genome

Moore’s Law prediction (halve every 18 months)

Ion torrent Personal Genome Machine

gene.gov/sequencingcosts
<table>
<thead>
<tr>
<th>Year</th>
<th>reference</th>
<th>Technology</th>
<th>Sample</th>
<th>Average Reported Coverage depth (fold)</th>
<th>Reported sequencing consumables cost</th>
<th>Estimated cost per 40-fold coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>Sanger (ABI)</td>
<td>JCV</td>
<td>7</td>
<td>$10,000,000</td>
<td>$57,000,000</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>Roche(454)</td>
<td>JDW</td>
<td>7</td>
<td>$1,000,000</td>
<td>$5,700,000</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>Illumina</td>
<td>NA18507</td>
<td>30</td>
<td>$250,000</td>
<td>$330,000</td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>Helicos</td>
<td>SRQ</td>
<td>28</td>
<td>$48,000</td>
<td>$69,000</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>this work</td>
<td>NA07022</td>
<td>87</td>
<td>$8,005</td>
<td>$3,700</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>this work</td>
<td>NA19240</td>
<td>63</td>
<td>$3,451</td>
<td>$2,200</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>this work</td>
<td>NA20431</td>
<td>45</td>
<td>$1,726</td>
<td>$1,500</td>
<td></td>
</tr>
</tbody>
</table>

Dystopia

Personalized Medicine
Secure Two-Party Computation

Bob’s Genome: ACTG...
Markers (~1000): [0,1, ..., 0]

Alice’s Genome: ACTG...
Markers (~1000): [0, 0, ..., 1]

\[ x = f(g_A, g_B) \]

Can Alice and Bob compute a function of their private data, without exposing anything about their data besides the result?
Secure Function Evaluation

Alice (circuit generator)
- Picks $a \in \{0, 1\}^s$

Bob (circuit evaluator)
- Picks $b \in \{0, 1\}^t$

Agree on
- $f(a, b) \rightarrow x$

Garbled Circuit Protocol

Outputs $x = f(a, b)$ without revealing $a$ to Bob or $b$ to Alice.

Andrew Yao, 1982/1986
## Regular Logic

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Diagram:**

```
 AND
```

**Inputs:**

- $a$:
  - 0
  - 1

- $b$:
  - 0
  - 1

**Output:**

- $x$:
  - 0
  - 1
Computing with Meaningless Values?

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
</tbody>
</table>

$a_i$, $b_i$, $x_i$ are random values, chosen by the circuit generator but meaningless to the circuit evaluator.
Computing with Garbled Tables

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
</tbody>
</table>

Bob can only decrypt one of these!

AND

Garbled And Gate

- $\text{Enc}_{a_0,b_1}(x_0)$
- $\text{Enc}_{a_1,b_1}(x_1)$
- $\text{Enc}_{a_1,b_0}(x_0)$
- $\text{Enc}_{a_0,b_0}(x_0)$

Random Permutation
Garbled Circuit Protocol

Alice (circuit generator)
Creates random keys: $a_0, a_1, b_0, b_1, x_0, x_1$

Bob (circuit evaluator)

How does the Bob learn his own input wires?

Sends $a_i$ to Bob based on her input value

Garbled Gate

<table>
<thead>
<tr>
<th>Gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Enc_{a_0,b_1}(x_0)$</td>
</tr>
<tr>
<td>$Enc_{a_1,b_1}(x_1)$</td>
</tr>
<tr>
<td>$Enc_{a_1,b_0}(x_0)$</td>
</tr>
<tr>
<td>$Enc_{a_0,b_0}(x_0)$</td>
</tr>
</tbody>
</table>
Primitive: Oblivious Transfer

Oblivious Transfer Protocol

Alice

Knows $b_0, b_1$

Bob

Picks $i \in \{0, 1\}$

Learns nothing

Learns $b_i$ (only)

Oblivious: Alice doesn’t learn which secret Bob obtains

Transfer: Bob learns one of Alice’s secrets

Rabin, 1981; Even, Goldreich, and Lempel, 1985; many subsequent papers
Chaining Garbled Circuits

We can do any computation privately this way!
Building Computing Systems

<table>
<thead>
<tr>
<th>Digital Electronic Circuits</th>
<th>Garbled Circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operate on <strong>known data</strong></td>
<td>Operate on <strong>encrypted wire labels</strong></td>
</tr>
<tr>
<td>One-bit logical operation requires moving a few electrons a few nanometers (hundreds of Billions per second)</td>
<td>One-bit logical operation requires performing (up to) 4 encryption operations: <strong>very slow execution</strong></td>
</tr>
<tr>
<td>Reuse is great!</td>
<td>Reuse is not allowed for privacy: <strong>huge circuits needed</strong></td>
</tr>
</tbody>
</table>

![Digital Electronic Circuits Diagram]

\[
\begin{align*}
Enc_{x_0, x_1}(x_{2_1}) \\
Enc_{x_0, x_1}(x_{2_1}) \\
Enc_{x_0, x_1}(x_{2_1}) \\
Enc_{x_0, x_1}(x_{2_0})
\end{align*}
\]
program Millionaires {
    type int = Int<4>; // 4-bit integer
    type AliceInput = int;
    type BobInput = int;
    type AliceOutput = Boolean;
    type BobOutput = Boolean;
    type Output = struct {
        AliceOutput alice, BobOutput bob;
    }
    type Input = struct {
        AliceInput alice, BobInput bob;
    }
    function Output out(Input inp) {
        out.alice = inp.alice > inp.bob;
        out.bob = inp.bob > inp.alice;
    }
}
Faster Circuit Execution

Pipelined Execution

Optimized Circuit Library

Partial Evaluation

Pipelined Execution

Circuit-Structure

Circuit-Level Application

GC Framework (Generator)

GC Framework (Evaluator)

Saves memory: never need to keep whole circuit in memory
Pipelining

- Circuit Generation
- Circuit Transmission
- Evaluation

Saves **time**: reduces latency and improves throughput
Results

Scalability (billions of gates)

Performance (10,000x non-free gates per second)
Semi-Honest is Half-Way There

**Privacy**
Nothing is revealed other than the output

**Correctness**
The output of the protocol is indeed $f(x,y)$

How can we get both correctness, and maintain privacy while giving both parties result?

As long as evaluator doesn’t send result back, and a malicious-resistant OT is used, **privacy** for evaluator is guaranteed.
Dual Execution Protocols

**Dual Execution Protocol**

Alice

- **generator**
- **evaluator**
  - $z' = f(x, y)$
  - $z'$, learned output wire labels

**first round execution (semi-honest)**

Bob

- **evaluator**
  - $z = f(x, y)$
- **generator**
  - $z$, learned output wire labels

**second round execution (semi-honest)**

**fully-secure, authenticated equality test**

Pass if $z = z'$ and correct wire labels

[Mohassel and Franklin, PKC’06]
Security Properties

Correctness: guaranteed by authenticated, secure equality test

Privacy:Leaks one (extra) bit on average adversary circuit generator provides a circuit that fails on ½ of inputs

Malicious generator can decrease likelihood of being caught, and increase information leaked when caught (but decreases average information leaked): at extreme, circuit fails on just one input
1-bit Leak

Victim's input space

Cheating detected
Proving Security: Malicious

Ideal World

Show equivalence

Real World

Adversary receives: $f(x', y')$

Corrupted party behaves arbitrarily

Secure Computation Protocol

Standard Malicious Model: can’t prove this for Dual Execution
Proof of Security: One-Bit Leakage

Ideal World

Controlled by malicious $\mathcal{A}$

Trusted Party in Ideal World

A $\xrightarrow{x'}$ Trusted Party in Ideal World

B $\xleftarrow{y'}$

Adversary receives: $f(x', y')$ and $g(x', y')$

$g \in R \rightarrow \{0, 1\}$

$g$ is an arbitrary Boolean function selected by adversary

Can prove equivalence to this for Dual Execution protocols
Implementation

\[ z' = f(x, y) \]

Pass if \( z = z' \) and correct wire labels

Recall: work to generate is \( \sim 3x \) work to evaluate!

\[ z = f(x, y) \]

Fully-secure, authenticated equality test

generator

evaluator

First round execution (semi-honest)

Second round execution (semi-honest)

Alice

Bob

Generator

Evaluator

\( z' \), learned output wire labels

\( z \), learned output wire labels
Circuits of arbitrary size can be done this way
Applications

Privacy-Preserving Biometric Matching

Private AES Encryption

Private Personal Genomics

Private Set Intersection
<table>
<thead>
<tr>
<th>Problem</th>
<th>Best Previous Result</th>
<th>Our Result</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Private Set Intersection</strong> (contact matching, common disease carrier)</td>
<td>Competitive with best custom protocols, scales to millions of 32-bit elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Hamming Distance</strong> (Face Recognition)</td>
<td>213s [SCiFI, 2010]</td>
<td>0.051s</td>
<td>4176</td>
</tr>
<tr>
<td><strong>Levenshtein Distance</strong> (genome, text comparison) – two 200-character inputs</td>
<td>534s [Jha+, 2008]</td>
<td>18.4s</td>
<td>29</td>
</tr>
<tr>
<td><strong>Smith-Waterman</strong> (genome alignment) – two 60-nucleotide sequences</td>
<td>[Not Implementable]</td>
<td>447s</td>
<td>-</td>
</tr>
<tr>
<td><strong>AES Encryption</strong></td>
<td>3.3s [Henecka, 2010]</td>
<td>0.2s</td>
<td>16.5</td>
</tr>
<tr>
<td><strong>Fingerprint Matching</strong> (1024-entry database, 640x8bit vectors)</td>
<td>~83s [Barni, 2010]</td>
<td>18s</td>
<td>4.6</td>
</tr>
</tbody>
</table>
Crazy Things in Typical Code

\[ a[i] = x \]
Circuit for Array Update

\[ a[i] = x \]
Easy (and Common) Case

```c
for (i = 0; i < n; i++)
a[i] += 1
```

```
a[0]  a[1]  a[2]  ...  a[n-1]
```

+1  +1  +1  +1  +1
Circuit Structures

Design circuits to support typical data structures efficiently

Non-trivial access patterns, but patterns nonetheless

Main opportunities:

Locality and Batching

if (x != 0)
  a[i] += 1
if (a[i] > 10)
  i += 1
a[i] = 5

t := a.top() + 1
a.cond_update(x != 0, t)
a.cond_push(x != 0 && t > 10, *)
a.cond_update(x != 0, 5)
Naïve Conditional Push
Naïve Conditional Push
More Efficient Stack

Level 0: [2, 9, 3]
\[ t = 3 \]
Level 1: [4, 7, 5, 4]
\[ t = 2 \]
Level 2: [8, 8, 2, 3, 8]
\[ t = 3 \]
...

Block size = \(2^{level}\)
Each level has 5 blocks, at least 2 full and 2 empty
Level 0

<table>
<thead>
<tr>
<th>2</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
</table>
\[ t = 3 \]

Level 1

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
\[ t = 2 \]

Conditional push (True, 7)

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
</table>
\[ t = 4 \]

Level 2

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
\[ t = 3 \]

<table>
<thead>
<tr>
<th>4</th>
<th>7</th>
<th>5</th>
<th>4</th>
</tr>
</thead>
</table>

Conditional push (True, 8)

<table>
<thead>
<tr>
<th>8</th>
<th>7</th>
<th>2</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
</table>
\[ t = 5 \]

Level 1

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
\[ t = 2 \]

<table>
<thead>
<tr>
<th>9</th>
<th>3</th>
<th>4</th>
<th>7</th>
<th>5</th>
<th>4</th>
</tr>
</thead>
</table>
\[ t = 3 \]

Shift

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>7</th>
</tr>
</thead>
</table>
\[ t = 3 \]
Amortized $\Theta(\log n)$ gates per operation
Arbitrary Array Accesses

(Associative Maps)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>m[0]</td>
<td>'A'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m[2]</td>
<td></td>
<td>'U'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m[9]</td>
<td></td>
<td></td>
<td>'M'</td>
<td></td>
</tr>
<tr>
<td>m[7]</td>
<td>'R'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m[0]</td>
<td>'D'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m[9]</td>
<td></td>
<td></td>
<td>'Y'</td>
<td></td>
</tr>
<tr>
<td>m[9]</td>
<td></td>
<td></td>
<td>'K'</td>
<td></td>
</tr>
<tr>
<td>m[7]</td>
<td>'C'</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution trace: **indexes and values are private values**
Batching Updates

\[
\begin{align*}
\text{m}[0] &= 'A' \\
\text{m}[2] &= 'U' \\
\text{m}[9] &= 'M' \\
\text{m}[7] &= 'R' \\
\text{m}[0] &= 'D' \\
\text{m}[9] &= 'Y' \\
\text{m}[9] &= 'K' \\
\text{m}[7] &= 'C'
\end{align*}
\]

Sort by Key

\[
\begin{align*}
\text{m}[0] &= 'A' \\
\text{m}[2] &= 'D' \\
\text{m}[0] &= 'D' \\
\text{m}[2] &= 'U' \\
\text{m}[7] &= 'R' \\
\text{m}[7] &= 'C' \\
\text{m}[9] &= 'M' \\
\text{m}[9] &= 'Y' \\
\text{m}[9] &= 'K'
\end{align*}
\]

stable sort!
Batching Updates

\[ m[0] = 'A' \]
\[ m[2] = 'U' \]
\[ m[9] = 'M' \]
\[ m[7] = 'R' \]
\[ m[0] = 'D' \]
\[ m[9] = 'Y' \]
\[ m[9] = 'K' \]
\[ m[7] = 'C' \]

Sort by Key

stable sort!

\[ m[0] = 'A' \]
\[ m[0] = 'D' \]
\[ m[2] = 'U' \]
\[ m[2] = 'U' \]
\[ m[7] = 'R' \]
\[ m[7] = 'C' \]
\[ m[9] = 'M' \]
\[ m[9] = 'Y' \]
\[ m[9] = 'K' \]
Batching Updates

Sort by Key

m[0] = 'A'
m[0] = 'D'
m[2] = 'U'
m[7] = 'R'
m[7] = 'C'
m[9] = 'M'
m[9] = 'Y'
m[9] = 'K'

stable sort!

Compare Adjacent

m[0] = 'A'
m[0] = 'D'
m[2] = 'U'
m[7] = 'R'
m[7] = 'C'
m[9] = 'M'
m[9] = 'Y'
m[9] = 'K'
Batching Updates

\[
\begin{align*}
&A' & m[0] &= 'A' \\
&U' & m[0] &= 'D' \\
&M' & m[2] &= 'U' \\
&R' & m[7] &= 'R' \\
&D' & m[7] &= 'C' \\
&Y' & m[9] &= 'M' \\
&K' & m[9] &= 'Y' \\
&C' & m[9] &= 'K'
\end{align*}
\]
Batching Updates

Sort by Key

stable sort!

m[0] = 'A'
m[0] = 'D'
m[2] = 'U'
m[7] = 'R'
m[7] = 'C'
m[9] = 'M'
m[9] = 'Y'
m[9] = 'K'

Compare Adjacent

m[0] = 'A'
m[0] = 'D'
m[2] = 'U'
m[7] = 'R'
m[7] = 'C'
m[9] = 'M'
m[9] = 'Y'
m[9] = 'K'
Sort by Key

stable sort!

Compare Adjacent

output wires

Discarded

Sort by Liveness

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>U</td>
<td>C</td>
<td>M</td>
</tr>
<tr>
<td>D</td>
<td>U</td>
<td>R</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>K</td>
</tr>
</tbody>
</table>

m[0] = 'A'
m[0] = 'D'
m[2] = 'U'
m[7] = 'C'
m[7] = 'R'
m[9] = 'M'
m[9] = 'Y'
m[9] = 'K'
Associative Map Cost

Oblivious Stable Sort

Comparisons and Liveness Marking

Oblivious Sort
Liveness/Key

Circuit Size:
\( \Theta(n \log n) \times \text{comparison cost} \)
\( \Theta(n \log^2 n) \)
Example Application: **DBScan**

Density-based clustering:
- depth-first search to find dense clusters

Alice’s Data  Bob’s Data  Joint Clusters

Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu. *KDD* 1996
\[ n \leftarrow |P| \]
\[ c \leftarrow 0 \]
\[ s \leftarrow \text{emptyStack} \]
\[ \text{cluster} \leftarrow [0, 0, \ldots] \]

for \( i \leftarrow [1, n] \) do
  if \( \text{cluster}[i] \neq 0 \) then
    continue
  \( V \leftarrow \text{getNeighbors}(i, P, \text{minpts}, \text{radius}) \)
  if \( \text{count}(V) < \text{minpts} \) then
    continue
  \( c \leftarrow c + 1 \) \hfill \text{Start a new cluster}

for \( j \leftarrow [1, n] \) do
  if \( V[j] = \text{true} \wedge \text{cluster}[j] \neq 0 \) then
    \( \text{cluster}[j] \leftarrow c \)
    \( s.\text{push}(j) \)

while \( s \neq \emptyset \) do
  \( k \leftarrow s.\text{pop()} \)
  \( V \leftarrow \text{getNeighbors}(k, P, \text{minpts}, \text{radius}) \)
  if \( \text{count}(V) < \text{minpts} \) then
    continue
  for \( j \leftarrow [1, n] \) do
    if \( V[j] = \text{true} \wedge \text{cluster}[j] \neq 0 \) then
      \( \text{cluster}[j] \leftarrow c \)
      \( s.\text{push}(j) \)

---

**Private Input:** \( P \) – array of points (combines private points from both parties)

**Public inputs:** \( \text{minpts}, \text{radius} \)

**Output:** cluster number for each point

---

**Conditional Push!**

**Array update!**
<table>
<thead>
<tr>
<th>Data Size</th>
<th>Normal Data Structures</th>
<th>Optimized Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>120</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>240</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>480</td>
<td>9.7 hours</td>
<td>55 minutes</td>
</tr>
</tbody>
</table>

**Execution Time (seconds)**

- **Normal Data Structures**
- **Optimized Structures**

**Graph Notes:**
- The graph compares execution time in seconds for different data sizes.
- Normal data structures show high execution times, climbing dramatically as data size increases.
- Optimized structures maintain a low execution time, even as data size increases significantly.

**Key Observations:**
- At a data size of 480, normal data structures take approximately 9.7 hours to execute, while optimized structures take only 55 minutes.
Cost to sequence human genome

Moore’s Law prediction (halve every 18 months)
Questions?