How to Implement (ORAM in) MPC

Marcel Keller Peter Scholl Nigel Smart

University of Bristol

21 February 2014

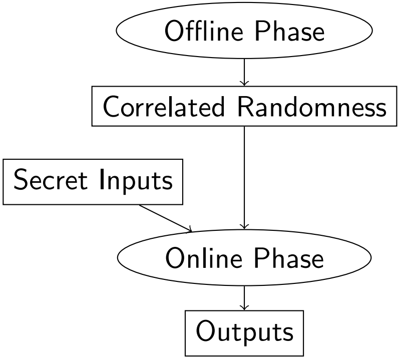
Overview

1. How to Implement MPC
2. ORAM in MPC

Part I

How to Implement MPC

SPDZ: MPC with Preprocessing



- Offline Phase
  - Independent of secret inputs
  - Homomorphic encryption with distributed decryption
  - Highly parallelizable
- Online Phase
  - No encryption
  - Information-theoretic security in random oracle model

How to Share a Secret with Authentication

	Shares	MAC shares	MAC key
	a_1	$\gamma(a)_1$	α_1
	a_2	$\gamma(a)_2$	α_2
	a_3	$\gamma(a)_3$	α_3
	a $= \sum_i a_i$	$\alpha \cdot a$ $= \sum_i \gamma(a)_i$	α $= \sum_i \alpha_i$
	$= \boxed{a}$		

How to Share a Secret with Authentication

	Shares	MAC shares	MAC key
	$a_1 + b_1$	$\gamma(a)_1 + \gamma(b)_1$	α_1
	$a_2 + b_2$	$\gamma(a)_2 + \gamma(b)_2$	α_2
	$a_3 + b_3$	$\gamma(a)_3 + \gamma(b)_3$	α_3
	$a + b$ $= \sum_i a_i + b_i$	$\alpha \cdot (a + b)$ $= \sum_i \gamma(a)_i + \gamma(b)_i$	α $= \sum_i \alpha_i$
	$= a + b$		

Multiplication with Random Triple (Beaver Randomization)

$$\begin{aligned} x \cdot y &= (x + a - a) \cdot (y + b - b) \\ &= (x + a) \cdot (y + b) - (y + b) \cdot a - (x + a) \cdot b + a \cdot b \end{aligned}$$

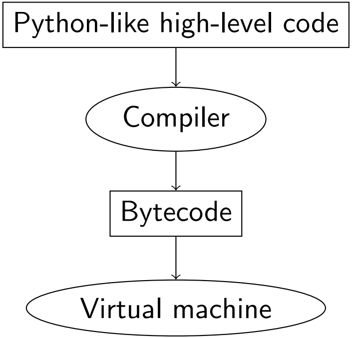
Multiplication with Random Triple (Beaver Randomization)

$$\boxed{x \cdot y} = (x + a - a) \cdot (y + b - b)$$
$$= \boxed{(x + a)} \cdot \boxed{(y + b)} - \boxed{(y + b)} \cdot \boxed{a} - \boxed{(x + a)} \cdot \boxed{b} + \boxed{a \cdot b}$$

Masked and opened

Random secret triple

Toolchain Overview



- Compiler
  - Python
  - Easier development
  - Circuit optimization
  - Speed not an issue
  - Memory overhead
- Virtual machine
  - Online phase
  - C++
  - Fast
  - $\sim$ 150 instructions

Core Technique: I/O Parallelization

$$z = x \cdot y$$
$$u = z \cdot w$$

$$z = x \cdot y$$
$$u = v \cdot w$$

Core Technique: I/O Parallelization

$$z = x \cdot y$$
$$u = z \cdot w$$

1. Mask and open x and y
2. Compute z
3. Mask and open z and w
4. Compute u

$$z = x \cdot y$$
$$u = v \cdot w$$

1. Mask and open x, y, v, and w
2. Compute z and u

Goal: Automatize I/O Parallelization

- Manual parallelization is tedious:

$$\begin{aligned} x_{10} &= x_2 \cdot x_3 \\ x_{11} &= x_8 + x_4 \\ x_{12} &= x_{10} \cdot x_1 \\ x_{13} &= x_7 + x_9 \\ x_{14} &= x_7 \cdot x_1 \\ x_{15} &= x_9 + x_{12} \\ x_{16} &= x_{13} \cdot x_{14} \\ x_{17} &= x_0 + x_{11} \\ x_{18} &= x_{11} \cdot x_{15} \\ x_{19} &= x_{13} \cdot x_7 \end{aligned}$$

$$\begin{aligned} x_{20} &= x_4 + x_6 \\ x_{21} &= x_{16} + x_2 \\ x_{22} &= x_0 + x_{12} \\ x_{23} &= x_{22} + x_{14} \\ x_{24} &= x_{11} + x_{19} \\ x_{25} &= x_4 \cdot x_{19} \\ x_{26} &= x_{23} \cdot x_9 \\ x_{27} &= x_7 \cdot x_5 \\ x_{28} &= x_{13} + x_{21} \\ x_{29} &= x_{14} + x_{27} \end{aligned}$$

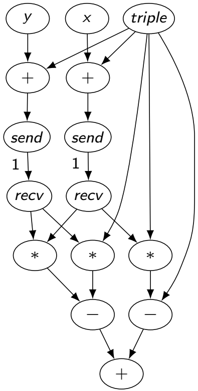
$$\begin{aligned} x_{30} &= x_{19} \cdot x_1 \\ x_{31} &= x_{16} + x_{26} \\ x_{32} &= x_0 \cdot x_{10} \\ x_{33} &= x_{26} + x_{32} \\ x_{34} &= x_7 + x_3 \\ x_{35} &= x_9 \cdot x_{29} \\ x_{36} &= x_{33} + x_{22} \\ x_{37} &= x_{29} \cdot x_{24} \\ x_{38} &= x_{16} + x_{23} \\ x_{39} &= x_{15} + x_{37} \end{aligned}$$

$$\begin{aligned} x_{40} &= x_{12} \cdot x_{39} \\ x_{41} &= x_{34} + x_7 \\ x_{42} &= x_{32} + x_5 \\ x_{43} &= x_{12} + x_{26} \\ x_{44} &= x_{43} \cdot x_{38} \\ x_{45} &= x_{38} + x_{14} \\ x_{46} &= x_{44} \cdot x_{27} \\ x_{47} &= x_{22} + x_{24} \\ x_{48} &= x_{39} \cdot x_{38} \\ x_{49} &= x_{21} \cdot x_3 \end{aligned}$$

$$\begin{aligned} x_{50} &= x_{28} + x_{16} \\ x_{51} &= x_{15} + x_{38} \\ x_{52} &= x_{50} \cdot x_{46} \\ x_{53} &= x_{19} + x_2 \\ x_{54} &= x_{20} \cdot x_{13} \\ x_{55} &= x_{21} + x_{22} \\ x_{56} &= x_{19} \cdot x_6 \\ x_{57} &= x_{46} + x_1 \\ x_{58} &= x_{38} \cdot x_{55} \\ x_{59} &= x_{47} + x_{29} \end{aligned}$$

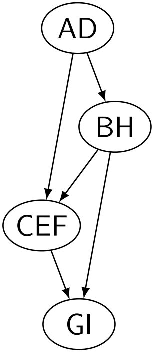
- SIMD not suitable for every application

Circuit as Directed Acyclic Graph



- Nodes: Instructions
- Edges: Output of instruction is input to another
- Two instructions for open: sending and receiving
- Edge weight:
  - One between sending and receiving
  - Zero otherwise
- Longest path with respect to weights determines communication round
- Merge all communication per round
  $\Rightarrow$ Optimal number of rounds

Merge by Communication Round / Longest Path



- Need to re-compute topological order (no edges pointing backwards)
  $\Rightarrow$ Standard algorithm linear in number of edges
- Heuristic to shorten lifetime of variables
  $\Rightarrow$ Reduced memory usage

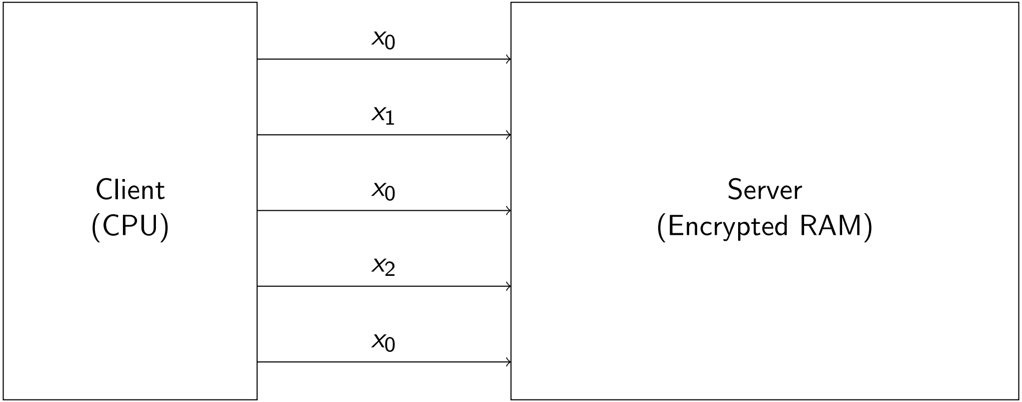
Part II

Oblivious RAM in MPC

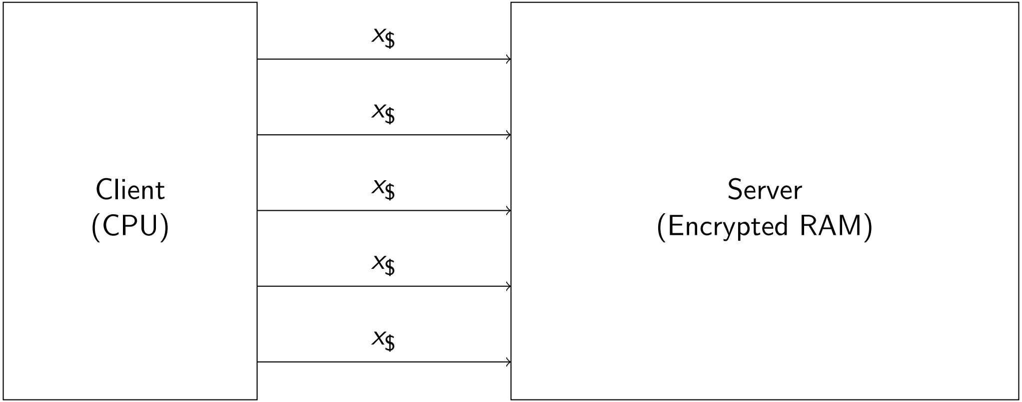
Goal: Oblivious Data Structures

- Generally
  - Secret pointers
  - Secret type of access if needed
- Oblivious array / dictionary
  - Secret index / key
  - Secret whether reading or writing
- Oblivious priority queue
  - Secret priority and value
  - Secret whether decreasing priority or inserting

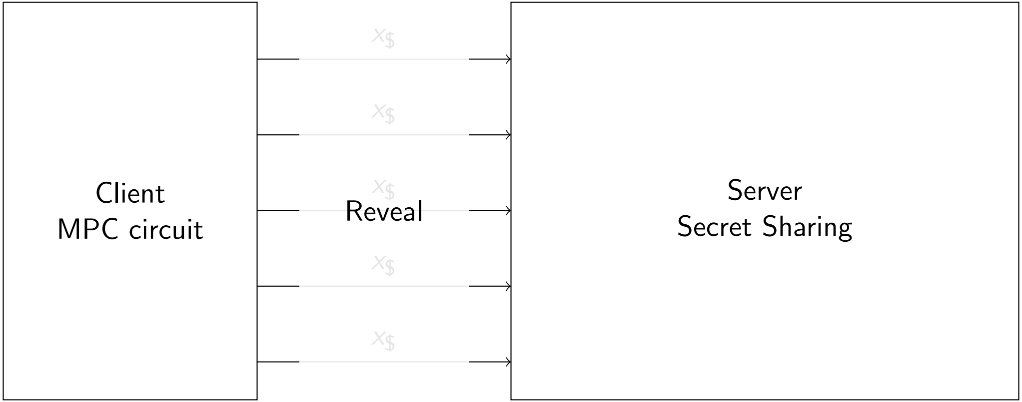
Oblivious RAM



Oblivious RAM



Oblivious RAM in MPC



Simple Oblivious Array (Trivial ORAM)

Inner product with index vector

$$\begin{pmatrix} [0] \\ \vdots \\ [0] \\ [1] \\ [0] \\ \vdots \\ [0] \end{pmatrix} \cdot \begin{pmatrix} [a_0] \\ \vdots \\ [a_{i-1}] \\ [a_i] \\ [a_{i+1}] \\ \vdots \\ [a_{n-1}] \end{pmatrix} = [a_i]$$

Index vector computation

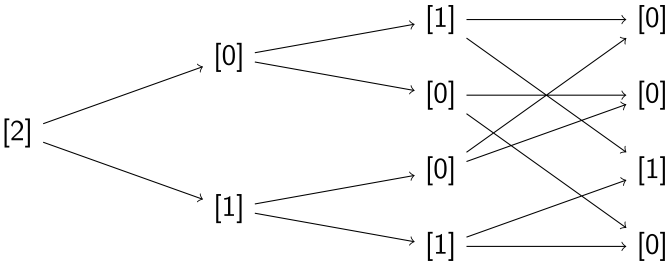
$$[i] \mapsto \begin{pmatrix} [i] \stackrel{?}{=} 0 \\ \vdots \\ [i] \stackrel{?}{=} i \\ \vdots \\ [i] \stackrel{?}{=} n-1 \end{pmatrix}$$

Simple Oblivious Array

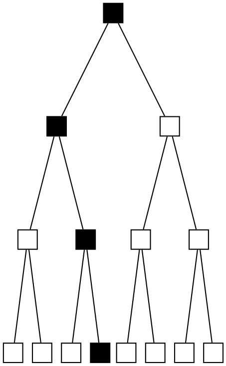
Index vector computation without equality test

- Bit decomposition: $[x] \mapsto ([x_0], \ldots, [x_{n-1}])$ such that $x = \sum_i x_i 2^i$
- Demux: $([x_0], \ldots, [x_{n-1}]) \mapsto ([\delta_0], \ldots, [\delta_{2^n-1}])$ such that $\delta_i = (i \stackrel{?}{=} x)$

Example: $n = 4$, $i = 2$



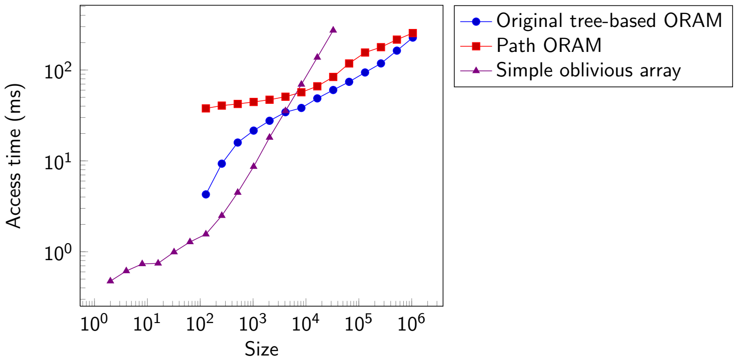
Tree-based ORAM



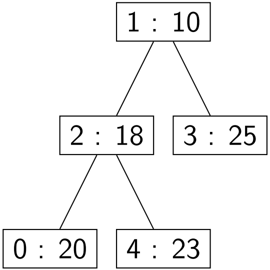
- Entries stored in tree of trivial ORAMs of fixed size (buckets)
- Access only buckets on path to specific leaf per ORAM access
- Store path in smaller ORAM $\Rightarrow$ recursion
- Data-independent eviction to distribute entries over tree
- Original idea by Shi et al. (Asiacrypt 2011)
- Path ORAM: improved eviction by Stefanov et al. (CCS 2013)

Oblivious Array Access Timings

Two Parties, Online Phase



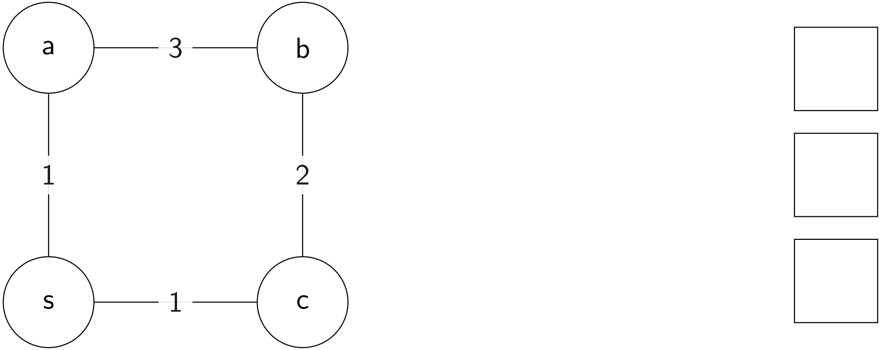
Oblivious Priority Queue



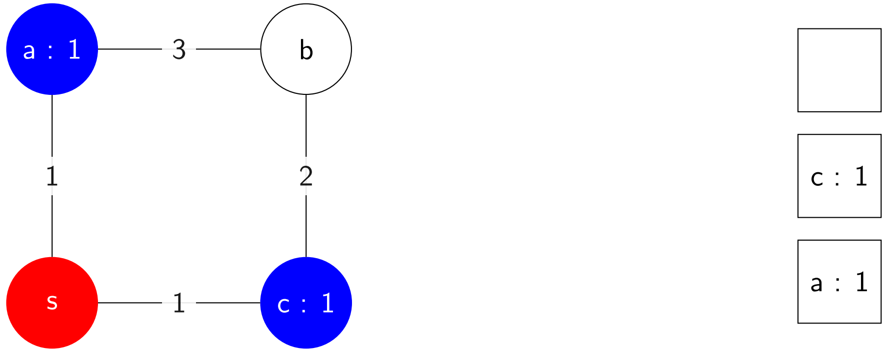
$[00, \lambda, 0, 1, 01]$

- Store value-priority pairs
- Values unique
- Operations
  - Remove value with minimal priority
  - Insert new pair
  - Update value to lower priority
- Two oblivious arrays
  - Binary heap by priority
  - Index to find entries in heap by value

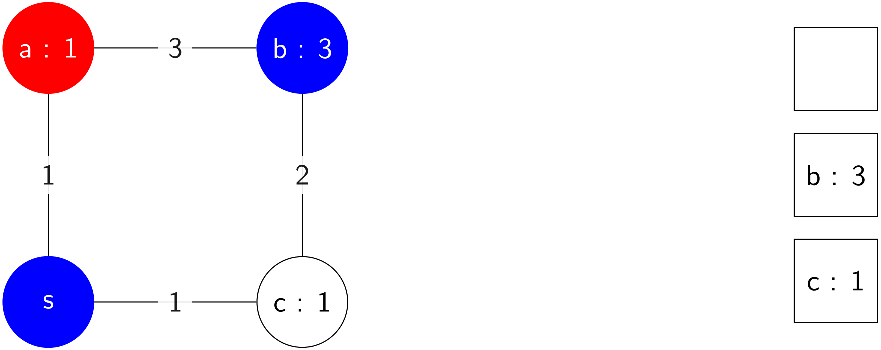
Dijkstra's Shortest Path Algorithm



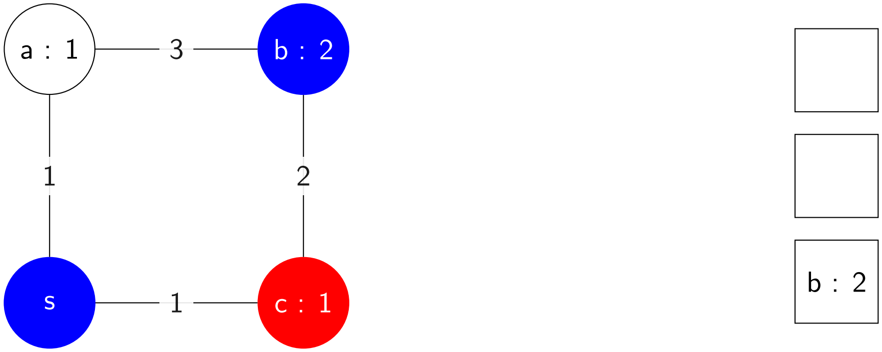
Dijkstra's Shortest Path Algorithm



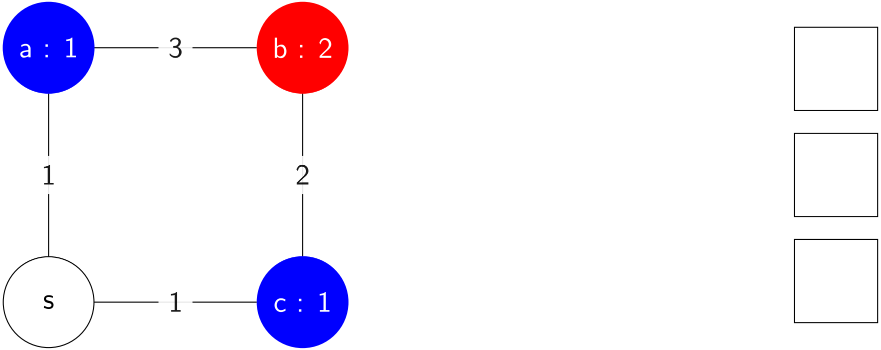
Dijkstra's Shortest Path Algorithm



Dijkstra's Shortest Path Algorithm



Dijkstra's Shortest Path Algorithm



Dijkstra's Algorithm in MPC

for each vertex **do**
  outer loop body
  for each neighbor **do**
    inner loop body

- Number of vertices and edges public
- Graph structure in two oblivious arrays (vertices and edges)
- Use oblivious priority queue
- Dijkstra's algorithms uses two nested loops
  - One vertices, one of neighbors thereof
  - MPC would reveal the number of neighbors for every vertex
  - Replace by loop over all edges in same order
  - Flag set when starting with a new vertex
- Polylog overhead over classical algorithm
- Previous work: polynomial overhead

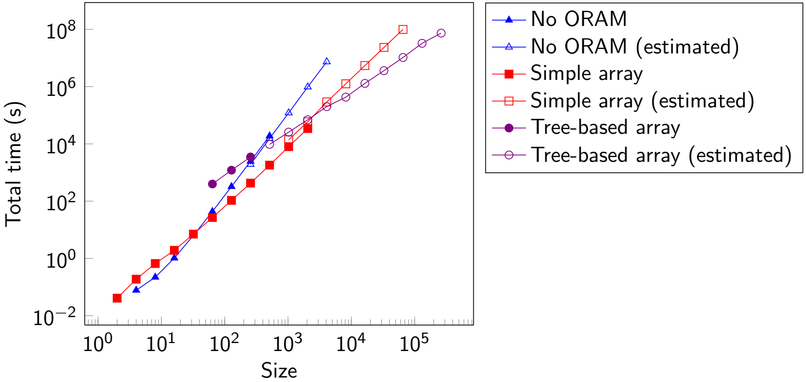
Dijkstra's Algorithm in MPC

for each edge **do**
    outer loop body
    (dummy if same vertex)
    inner loop body

- Number of vertices and edges public
- Graph structure in two oblivious arrays (vertices and edges)
- Use oblivious priority queue
- Dijkstra's algorithms uses two nested loops
  - One vertices, one of neighbors thereof
  - MPC would reveal the number of neighbors for every vertex
  - Replace by loop over all edges in same order
  - Flag set when starting with a new vertex
- Polylog overhead over classical algorithm
- Previous work: polynomial overhead

Dijkstra's Algorithm in MPC

Timings for Cycle Graphs



Conclusion

- Practical MPC requires a dedicated compiler.
  (ACM CCS 2013 / ePrint 2013:143)
- Oblivious data structures in MPC are feasible and useful.
  (ePrint 2014:137)