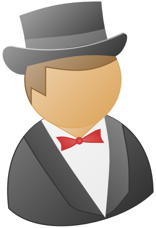
An Introduction to Multi-Party Computation

Marcel Keller

CSIRO's Data61

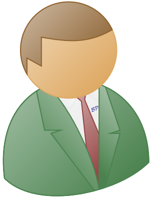
28 January 2025

Millionaire's Problem



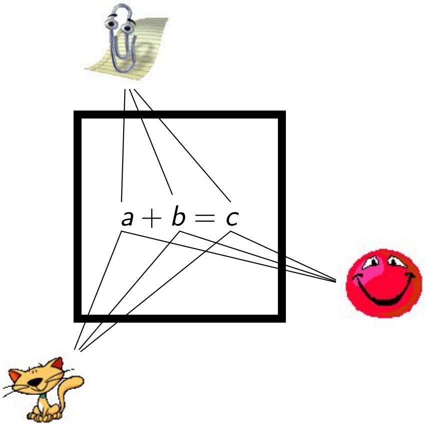
$\$x$

$x < y?$



$\$y$

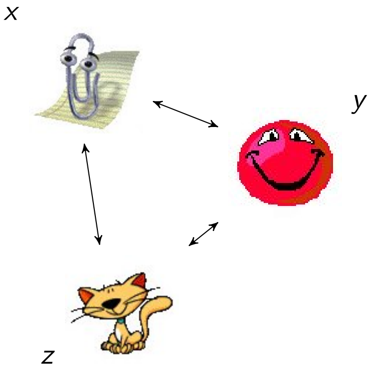
Imagine a Magic Black Box Between a Set of Parties



Parties

- Have handles to values
- Don't know the values
- Can input values
- Can agree on computations creating new values
- Can agree on outputting values

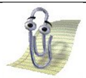
Secure Multiparty Computation: Black Box as Protocol



Wanted: $f(x, y, z)$

- Computation on secret inputs
- Replace black box
- Central questions in MPC
  - How many honest parties?
  - Dishonest parties still follow the protocol?
- MP-SPDZ supports $>$ 40 protocol variants across all properties

Core Technology: Secret Sharing

	Random Shares
	x_1
	x_2
	x_3
Secret	$x = \sum_i x_i$

Core Technology: Secret Sharing

	Random Shares			
	x_1	y_1	$x_1 + y_1$	$c \cdot x_1$
	x_2	y_2	$x_2 + y_2$	$c \cdot x_2$
	x_3	y_3	$x_3 + y_3$	$c \cdot x_3$
Secret	$x = \sum_i x_i$	$y = \sum_i y_i$	$x + y = \sum_i (x_i + y_i)$	$c \cdot x = \sum_i (c \cdot x_i)$

Security of Secret Sharing

Example

- Secret 7, shares 3 and 4
- If you only know 3 but not 4, you can only guess

Mathematics

- Need a finite domain for uniform randomness
- Example: computation modulo 2^{64} like 64-bit computers

What About Liars? (Malicious Security)

Problem

If the secret sharing is a simple sum, any party can alter the result by using the wrong number.

Possible solution: Replication

If every party holds enough shares, they can check on each other.

Example Sharing $x = a + b + c$

Party 1 holds (a, b), Party 2 holds (b, c), Party 3 holds (a, c)

Change in security model

It doesn't take all parties to learn the secret.

Multiplication with Random Triple (Beaver Randomization)

Have: x, y, addition in black box

Want: $x \cdot y$

Multiplication with Random Triple (Beaver Randomization)

Have: x, y, addition in black box

Want: $x \cdot y$

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

Multiplication with Random Triple (Beaver Randomization)

Have: x, y, addition in black box, (a, b, $a \cdot b$ for random a, b)

Want: $x \cdot y$

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

Masked and revealed (one-time pad)

Random secret triple (preprocessed)

Multiplication Protocol

- Fetch $\boxed{a}$, $\boxed{b}$, $\boxed{a \cdot b}$
- Reveal $(a + x, b + y)$
- Compute $(x + a) \cdot (y + b) - (y + b) \cdot \boxed{a} - (x + a) \cdot \boxed{b} + \boxed{a \cdot b}$

I/O Parallelization

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

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

I/O Parallelization

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

1. Compute z
2. Compute u

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

1. Compute z and u

Unified C++ Interface

```
for (int i = 0; i < n; i++)
  sum[i] = a[i] + b[i];

protocol.init_mul();
for (int i = 0; i < n; i++)
  protocol.prepare_mul(a[i], b[i]);
protocol.exchange();
for (int i = 0; i < n; i++)
  product[i] = protocol.finalize_mul();
```

- Addition is straightforward
- Similar for multiplication would lead to sequential execution
- Prepare/exchange/finalize minimal interface for parallel execution

Goal: Automatize I/O Parallelization

Manual parallelization is tedious:

$x_{10} = x_2 \cdot x_3$	$x_{20} = x_4 + x_6$	$x_{30} = x_{19} \cdot x_1$	$x_{40} = x_{12} \cdot x_{39}$	$x_{50} = x_{28} + x_{16}$
$x_{11} = x_8 + x_4$	$x_{21} = x_{16} + x_2$	$x_{31} = x_{16} + x_{26}$	$x_{41} = x_{34} + x_7$	$x_{51} = x_{15} + x_{38}$
$x_{12} = x_{10} \cdot x_1$	$x_{22} = x_0 + x_{12}$	$x_{32} = x_0 \cdot x_{10}$	$x_{42} = x_{32} + x_5$	$x_{52} = x_{50} \cdot x_{46}$
$x_{13} = x_7 + x_9$	$x_{23} = x_{22} + x_{14}$	$x_{33} = x_{26} + x_{32}$	$x_{43} = x_{12} + x_{26}$	$x_{53} = x_{19} + x_2$
$x_{14} = x_7 \cdot x_1$	$x_{24} = x_{11} + x_{19}$	$x_{34} = x_7 + x_3$	$x_{44} = x_{43} \cdot x_{38}$	$x_{54} = x_{20} \cdot x_{13}$
$x_{15} = x_9 + x_{12}$	$x_{25} = x_4 \cdot x_{19}$	$x_{35} = x_9 \cdot x_{29}$	$x_{45} = x_{38} + x_{14}$	$x_{55} = x_{21} + x_{22}$
$x_{16} = x_{13} \cdot x_{14}$	$x_{26} = x_{23} \cdot x_9$	$x_{36} = x_{33} + x_{22}$	$x_{46} = x_{44} \cdot x_{27}$	$x_{56} = x_{19} \cdot x_6$
$x_{17} = x_0 + x_{11}$	$x_{27} = x_7 \cdot x_5$	$x_{37} = x_{29} \cdot x_{24}$	$x_{47} = x_{22} + x_{24}$	$x_{57} = x_{46} + x_1$
$x_{18} = x_{11} \cdot x_{15}$	$x_{28} = x_{13} + x_{21}$	$x_{38} = x_{16} + x_{23}$	$x_{48} = x_{39} \cdot x_{38}$	$x_{58} = x_{38} \cdot x_{55}$
$x_{19} = x_{13} \cdot x_7$	$x_{29} = x_{14} + x_{27}$	$x_{39} = x_{15} + x_{37}$	$x_{49} = x_{21} \cdot x_3$	$x_{59} = x_{47} + x_{29}$

Compilation

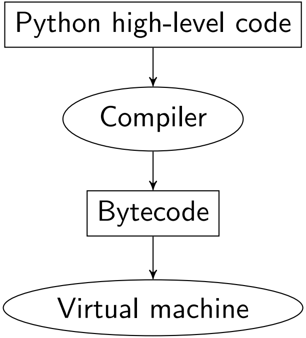
Have

```
x[0] = x[1] * x[2]
x[3] = x[0] * (x[4] * x[5] + x[6])
```

Want

```
protocol.init_mul();
protocol.prepare_mul(x[1], x[2]);
protocol.prepare_mul(x[4], x[5]);
protocol.exchange();
x[0] = protocol.finalize_mul();
tmp = protocol.finalize_mul() + x[6];
protocol.init_mul();
protocol.prepare_mul(tmp, x[0]);
x[3] = protocol.finalize_mul();
```

Toolchain Overview



Compiler

- Implemented in Python
- Optimization (reduce network rounds)
- Library for various arithmetic: integer, fractional, mathematical
- Machine learning functionality

Virtual machine

- One per protocol
- Implemented in C++ for speed

Links

https://eprint.iacr.org/2020/300

https://github.com/data61/MP-SPDZ

https://mp-spdz.readthedocs.io