Computation with a Dishonest Majority

Marcel Keller

CSIRO's Data61

7 April 2022

From Honest to Dishonest Majority

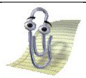
Honest-Majority Secret Sharing

Parties know enough to compute multiplication within secret sharing (e.g., Shamir)

Dishonest-Majority Secret Sharing

Parties know very little, so need more techniques

How to Share a Secret (Additively)

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

How to Share a Secret (Additively)

	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	$x = \sum_i x_i$	$y = \sum_i y_i$	$x + y = \sum_i (x_i + y_i)$	$c \cdot x = \sum_i (c \cdot x_i)$
	$\boxed{x}$	$\boxed{y}$	$\boxed{x + y}$	$\boxed{c \cdot x}$

Towards Multiplication

Have

- Input
- Linear operations
- Output

Want

Multiplication

Assume (for now)

Special randomness

Multiplication with Random Triple (Beaver Randomization)

Have: $\boxed{x}$, $\boxed{y}$, addition in black box

Want: $\boxed{x \cdot y}$

Multiplication with Random Triple (Beaver Randomization)

Have: $\boxed{x}$, $\boxed{y}$, addition in black box

Want: $\boxed{x \cdot y}$

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

Multiplication with Random Triple (Beaver Randomization)

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

Want: $\boxed{x \cdot y}$

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

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

Masked and revealed
(one-time pad)

Random secret triple
(preprocessed)

Multiplication with Random Triple (Beaver Randomization)

Pre: $[x], [y]$, $([a], [b], [ab])$ for uniformly random a, b

Post: $[xy]$

---

1. Parties open $[x + a]$ and $[y + b]$ to σ and ρ
2. Parties output $\sigma \cdot \rho - \rho \cdot [a] - \sigma[b] + [ab]$

Checking Correctness

Problem with additive secret sharing

Every share counts, so changing a share changes the secret value.

Solution: Redundancy

Use second secret sharing to check the first.

How to Share a Secret

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

How to Share a Secret (with Authentication)

	Shares	Tag shares	Tag key
	x_1	$\gamma(x)_1$	α_1
	x_2	$\gamma(x)_2$	α_2
	x_3	$\gamma(x)_3$	α_3
Secret	x $= \sum_i x_i$	$\alpha \cdot x$ $= \sum_i \gamma(x)_i$	α $= \sum_i \alpha_i$
	$= \boxed{x}$		

How to Share a Secret (with Authentication)

	Shares	Tag shares	Tag key
	$x_1 + y_1$	$\gamma(x)_1 + \gamma(y)_1$	α_1
	$x_2 + y_2$	$\gamma(x)_2 + \gamma(y)_2$	α_2
	$x_3 + y_3$	$\gamma(x)_3 + \gamma(y)_3$	α_3
Secret	$x + y$ $= \sum_i x_i + y_i$	$\alpha \cdot (x + y)$ $= \sum_i \gamma(x)_i + \gamma(y)_i$	α $= \sum_i \alpha_i$
	$= x + y$		

Authentication Security

Definition

Corrupt parties cannot create correct shares to "wrong" value.

Proof

Assume correct share $[x], [\gamma(x)]$ and adversary creating a correct share $[x + e], [\gamma(x) + f]$ for $e \neq 0$. Recall $\gamma(x) = \alpha \cdot x$. Then,

$$\begin{aligned} f \cdot e^{-1} &= (\gamma(x + e) - \gamma(x)) \cdot e^{-1} \\ &= (\alpha \cdot (x + e) - \alpha \cdot x) \cdot e^{-1} = \alpha \end{aligned}$$

Requirements

α is secret and every non-zero value is invertible (e.g., compute modulo a prime).

How to Reveal a Secret (with Authentication)

Protocol

$\boxed{x}$: Party i holds additive shares $(x_i, \gamma(x)_i, \alpha_i)$

Reveal Parties broadcast x_i, compute $x = \sum x_i$

Correctness not guaranteed: could send anything

How to Reveal a Secret (with Authentication)

Protocol

$\boxed{x}$: Party i holds additive shares $(x_i, \gamma(x)_i, \alpha_i)$

Reveal Parties broadcast x_i, compute $x = \sum x_i$

Correctness not guaranteed: could send anything

Check
- Parties broadcast $(\gamma(x)_i - x \cdot \alpha_i)$
- Parties check $\sum_i (\gamma(x)_i - x \cdot \alpha_i) \stackrel{?}{=} x \cdot \alpha - x \cdot \alpha = 0$

How to Reveal a Secret (with Authentication)

Protocol

$\boxed{x}$: Party i holds additive shares $(x_i, \gamma(x)_i, \alpha_i)$

Reveal Parties broadcast x_i, compute $x = \sum x_i$

Correctness not guaranteed: could send anything

Check

- Parties broadcast $(\gamma(x)_i - x \cdot \alpha_i)$ by committing first (rushing adversary)
- Parties check $\sum_i (\gamma(x)_i - x \cdot \alpha_i) \stackrel{?}{=} x \cdot \alpha - x \cdot \alpha = 0$

Commitment

- Send "encrypted" information first, open later
- In above context: cannot depend on others' parties messages

Multiplication with Random Triple (Beaver Randomization)

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

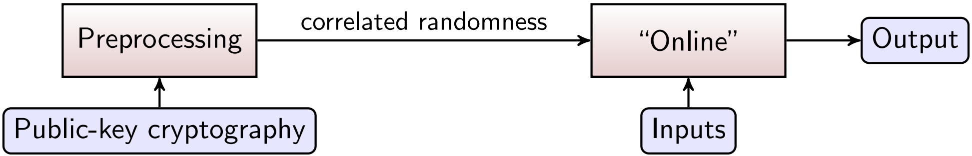
Want: $\boxed{x \cdot y}$

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

Masked and revealed (one-time pad)

Random secret triple (preprocessed)

Preprocessing MPC Protocols



Advantages

- No secret inputs on the line when using crypto
  $\Rightarrow$ No one gets hurt if protocol aborts!
- Online computation might have many rounds, but preprocessing is constant-round.

Preprocessing MPC Protocols

Public-key cryptography options

- Homomorphic encryption: allows operations on encrypted values
- Oblivious transfer: simplest building block for MPC

Section 1

Homomorphic Encryption

Semi-Homomorphic Encryption

Encryption

Encryption Enc_{pk} and decryption Dec_{sk} such that

$$\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(a)) = a$$

but $\mathsf{Enc}_{pk}(a)$ looks "random" to anyone without the secret key sk.

Operations

- $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(a) \boxplus \mathsf{Enc}_{pk}(b)) = a + b$
- $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(a) \boxdot b) = a \cdot b$

Two-Party Multiplication Protocol

Pre: P_A knows a and (pk, sk), P_B knows b and pk

Post: P_A knows c_A, P_B knows c_B such that $c_A + c_B = a \cdot b$

- P_A sends $\mathsf{Enc}_{pk}(a)$ to P_B
- P_B computes $E := b \boxdot \mathsf{Enc}_{pk}(a) \boxminus \mathsf{Enc}_{pk}(c_B)$ for random c_B
- P_B sends E to P_A
- P_A decrypts E to c_A

Complete Multiplication with Two-Party Protocol

Pre: Party P_i knows shares a_i, b_i for $[a], [b]$ where $a = \sum a_i,\ b = \sum b_i$

Post: Party P_i knows share c_i of $[c] = [ab]$

- For every pair $i \neq j$, P_i and P_j run two-party protocol on (a_i, b_j) to obtain shares c_{ij}^A and c_{ij}^B such that $c_{ij}^A + c_{ij}^B = a_i \cdot b_j$
- Every party P_i outputs $c_i = a_i \cdot b_i + \sum_{i \neq j}(c_{ij}^A + c_{ji}^B)$

$$\begin{aligned}\sum_i c_i &= \sum_i a_i \cdot b_i + \sum_{i \neq j}(c_{ij}^A + c_{ji}^B) \\ &= \sum_i a_i \cdot b_i + \sum_{i \neq j}(c_{ij}^A + c_{ij}^B) = \sum_i a_i \cdot b_i + \sum_{i \neq j} a_i \cdot b_j = a \cdot b\end{aligned}$$

Why Not Use Homomorphic Encryption Directly?

- HE is most efficient when working on many values in parallel
  $\Rightarrow$ Perfect for triple generation
- Not using sensitive data simplifies checking for malicious behavior

Somewhat Homomorphic Encryption

Semi-homomorphic

- $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(a)) = \mathsf{a}$
- $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(a) \boxplus \mathsf{Enc}_{pk}(b)) = a + b$
- $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(a) \boxdot b) = a \cdot b$

Multiply ciphertexts

$\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(a) \boxdot \mathsf{Enc}_{pk}(b)) = a \cdot b$

Distributed Homomorphic Encryption

Assume

Can share secret key sk such that the the shares $sk_0, \ldots, sk_{n-1}$ together allow decryption in a protocol that keeps sk secret.

Encryption to secret sharing

1. Party P_i broadcast $\mathsf{Enc}_{pk}(f_i)$ for random f_i
2. Parties decrypt $\mathsf{Enc}_{pk}(a) \boxplus \sum_i \mathsf{Enc}_{pk}(f_i)$ to $(a + \sum_i f_i)$
3. Party P_0 outputs $a_i = a + \sum_i f_i - f_0$, all other parties P_i output $-f_i$

$$\sum_i a_i = a + \sum_i f_i - f_0 + \sum_{i \neq 0} -f_i = a$$

Secure Multiplication Using Somewhat Homomorphic Encryption

Pre: Party P_i knows shares a_i, b_i for $[a], [b]$ where $a = \sum a_i,\ b = \sum b_i$

Post: Party P_i knows share c_i of $[c] = [ab]$

- Party P_i broadcasts $\mathsf{Enc}_{pk}(a_i)$ and $\mathsf{Enc}_{pk}(b_i)$
- Parties convert $(\sum_i \mathsf{Enc}_{pk}(a_i)) \boxdot (\sum_i \mathsf{Enc}_{pk}(b_i))$ to secret sharing

Towards Malicious Security

Adding Authentication Tags

Run multiplicatoin protocol between $[\alpha]$ and $([a], [b], [c])$ to get authenticated secret sharing.

Cheating Potential

What if corrupted parties use different shares for $a \cdot b$ and $(a \cdot b \cdot \alpha)$?

Solution

Generate two independent triples and check one using the other.

Triple Sacrifice

Pre: Independent authenticated triples $([a], [b], [c])$ and $([g], [f], [h])$

Post: Triple $([a], [b], [c])$ with $c = ab$ guaranteed

---

1. Generate fresh random value t
2. Open $t \cdot [a] - [f]$ as ρ and $[b] - [g]$ as σ
3. Compute and open $t \cdot [c] - [h] - \sigma \cdot [f] - \rho \cdot [g] - \sigma \cdot \rho$
4. Abort if the result is not zero or the opening is incorrect

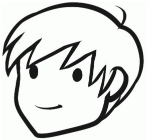
Correctness Straight-forward*

Security Adversary has to commit to error before t is fixed. If the domain is large enough, the check is unlikely to pass.*

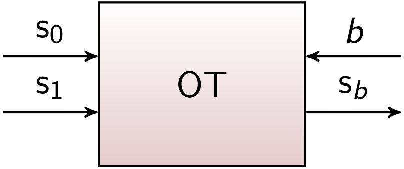
Section 2

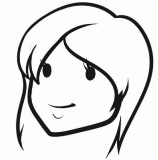
Oblivious Transfer

1-out-of-2 Oblivious Transfer



Sender

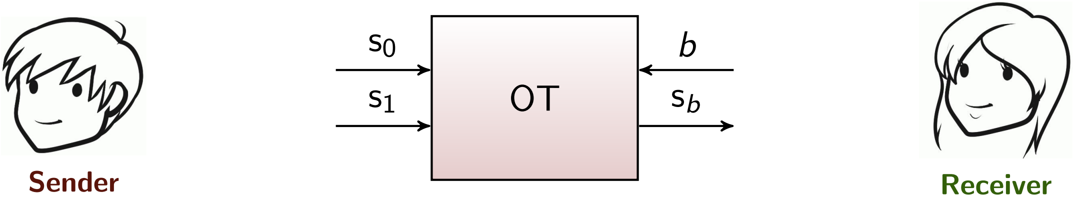




Receiver

- The **Sender** inputs two strings s_0 and s_1 and learns nothing.
- The **Receiver** inputs a bit b and learns only s_b.

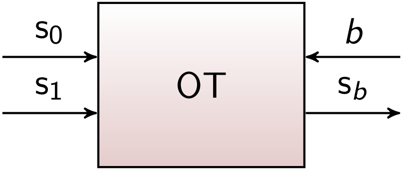
1-out-of-2 Oblivious Transfer



Why is it so special?

- Only slightly more than one input, one output
- Sending any of the inputs directly would break security

Partial Secure Multiplication from Oblivious Transfer*



Pre:
- P_A knows element $a \in \mathbb{Z}_M$
- P_B knows bit b

Post: P_A knows c_A, P_B knows c_B such that $c_A + c_B = a \cdot b$

- P_A samples random c_A
- P_A and P_B use OT with $s_0 := c_A$, $s_1 := c_A - a$
- P_B learns s_b and outputs $c_B := -s_b$.

Complete Secure Multiplication from Oblivious Transfer

From element-bit to element-element

Break down $\mathbb{Z}_M \times \mathbb{Z}_M$ multiplication to $\log M$ multiplications of bit and element in $\mathbb{Z}_M$:

$$x = \sum_{i=0}^{\log M} 2^i \cdot x_i \quad \Rightarrow \quad x \cdot y = \sum_{0}^{\log M} 2^i \cdot (x_i \cdot y)$$

From known values to secret sharing

Run pair-wise multiplication on shares as before

Constructing OT Like Diffie-Hellman*

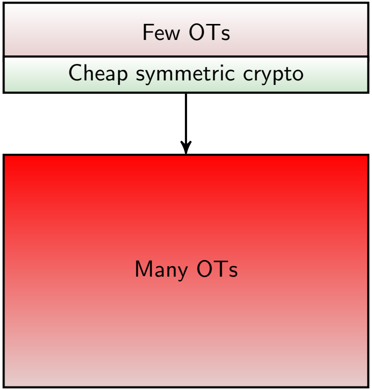
Ingredients

- Discrete logarithm
- Hash function
- Symmetric encryption

Cost

Discrete exponentation is expensive and limits throughput to 10,000 OT per second.
How to avoid?

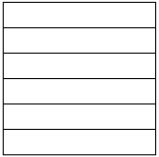
OT Extension — Basic Idea



Speedup

From 10,000 OT per second to 7 million

OT Extension with Passive Security



1.	Base OTs	k random OTs / k bits
2.	Extend length with PRG	k random OTs / n bits
3.	Introduce correlation	k correlated OTs / n bits
4.	Transpose	n correlated OTs / k bits
5.	Hash to break correlation	n random OTs / k bits

Computational security parameter $k = 128$

Number of OTs produced $n \geq 128$

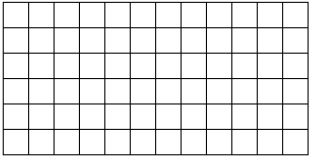
OT Extension with Passive Security

1.	Base OTs	k random OTs / k bits
2.	Extend length with PRG	k random OTs / n bits
3.	Introduce correlation	k correlated OTs / n bits
4.	Transpose	n correlated OTs / k bits
5.	Hash to break correlation	n random OTs / k bits

Computational security parameter $k = 128$

Number of OTs produced $n \geq 128$

OT Extension with Passive Security

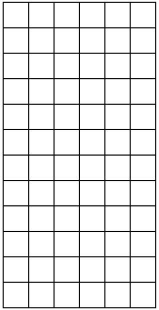


1. Base OTs — k random OTs / k bits
2. Extend length with PRG — k random OTs / n bits
3. Introduce correlation — k correlated OTs / n bits
4. Transpose — n correlated OTs / k bits
5. Hash to break correlation — n random OTs / k bits

Computational security parameter $k = 128$

Number of OTs produced $n \geq 128$

OT Extension with Passive Security

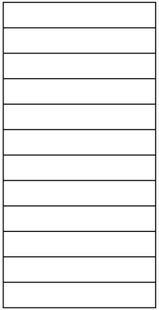


1. Base OTs — k random OTs / k bits
2. Extend length with PRG — k random OTs / n bits
3. Introduce correlation — k correlated OTs / n bits
4. Transpose — n correlated OTs / k bits
5. Hash to break correlation — n random OTs / k bits

Computational security parameter $k = 128$

Number of OTs produced $n \geq 128$

OT Extension with Passive Security

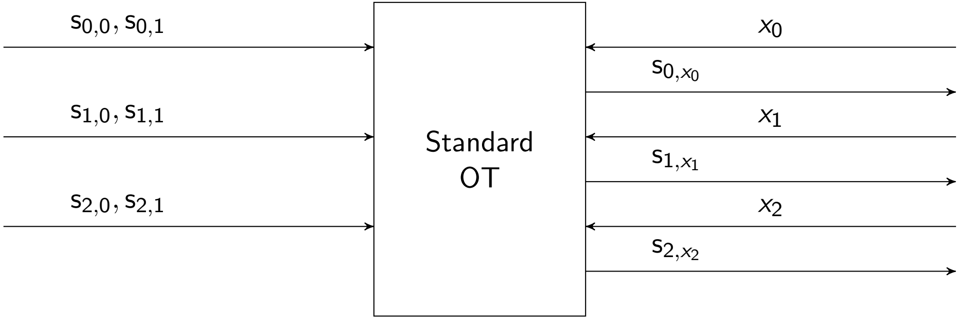


1.	Base OTs	k random OTs / k bits
2.	Extend length with PRG	k random OTs / n bits
3.	Introduce correlation	k correlated OTs / n bits
4.	Transpose	n correlated OTs / k bits
5.	Hash to break correlation	n random OTs / k bits

Computational security parameter $k = 128$

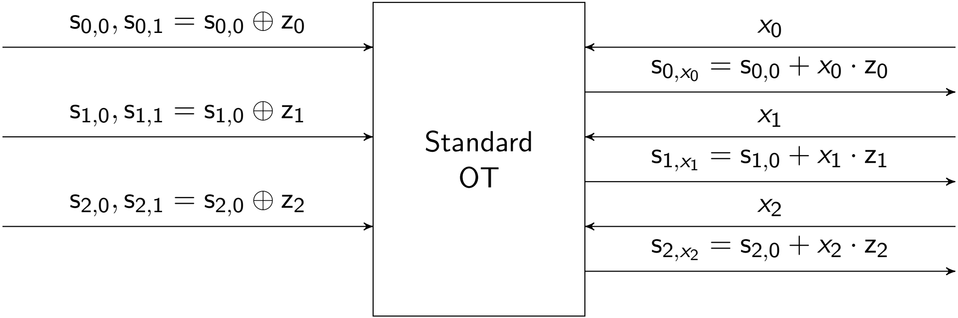
Number of OTs produced $n \geq 128$

Another Look at OT



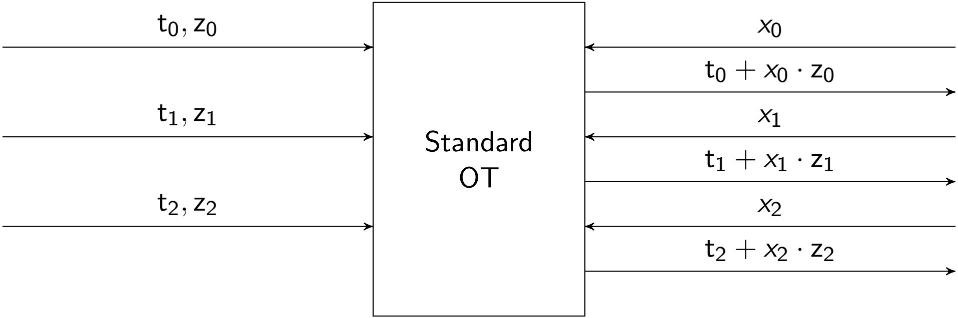
x_i: selection bit
$s_{i,0}, s_{i,1}, t_i, z_i, y$: strings

Another Look at OT



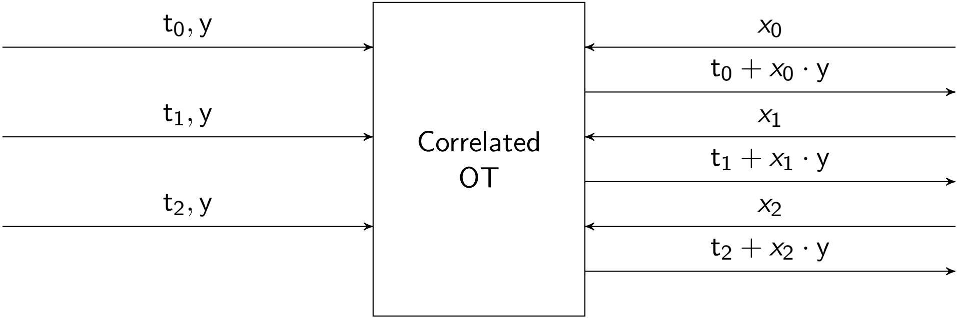
x_i: selection bit
$s_{i,0}, s_{i,1}, t_i, z_i, y$: strings

Another Look at OT



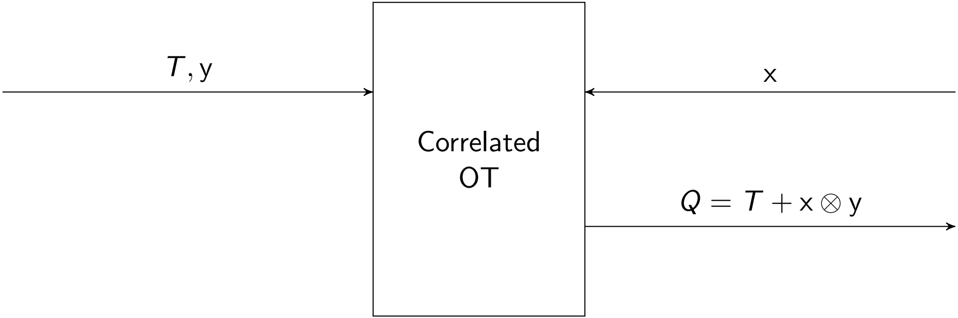
x_i: selection bit
$\mathsf{s}_{i,0}, \mathsf{s}_{i,1}, \mathsf{t}_i, \mathsf{z}_i, \mathsf{y}$: strings

Another Look at OT



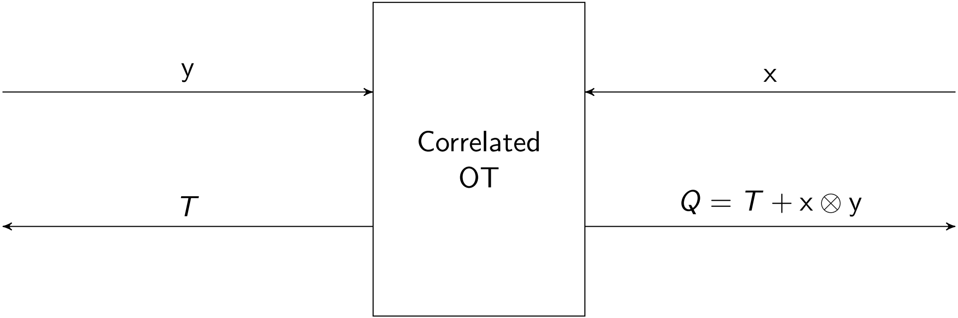
x_i: selection bit
$s_{i,0}, s_{i,1}, t_i, z_i, y$: strings

Another Look at OT



x, y: strings / vectors in $(\mathbb{F}_2)^k$ and $(\mathbb{F}_2)^n$, respectively
Q, T, Z: matrices in $(\mathbb{F}_2)^{k \times n}$
$x \otimes y$: tensor product, matrix of all possible products

Another Look at OT



x, y:	strings / vectors in $(\mathbb{F}_2)^k$ and $(\mathbb{F}_2)^n$, respectively
Q, T, Z:	matrices in $(\mathbb{F}_2)^{k \times n}$
$\mathsf{x} \otimes \mathsf{y}$:	tensor product, matrix of all possible products

Summary: Dishonest-Majority Computation

Multiplication using preprocessed triples

- Making use of vectorized homomorphic encryption
- Simplify checking on malicious behavior

Security against malicious behavior

- Use double sharing to check on openings
- Sacrifice triples to guarantee correct triples
- Zero-knowledge proofs to check on encryption
- More also required for OT-based generation