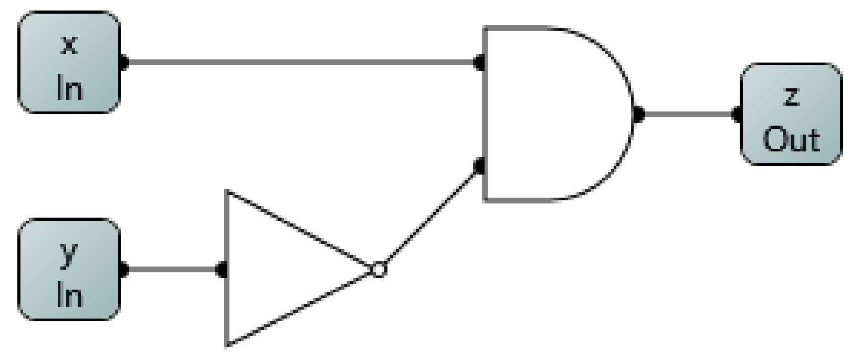
The Millionaire Problem: Yao's Garbled Circuits

Two millionaires Alice and Bob want to figure out who has more money. How can they figure this out without revealing their bank statements?

In 1982, Andrew C. Yao came up with a brilliant solution that not only solves this problem but any problem Alice and Bob want to figure out. He called his protocol garbled circuits. To show how Alice and Bob can use garbled circuits to solve the millionaire problem, we first need to understand logic circuits.

What are logic circuits?

Logic circuits are a way of describing functions using logic gates, such as AND gates or OR gates. They take a certain number of binary inputs (each input is a 0 or a 1), use logic gates to do computation on these inputs, and then output another set of binary values.



Logic gates are simple binary functions with 1 or 2 inputs and with 1 output. An easy way to represent a gate is with its truth table, which shows which inputs will have which outputs. Here are some example tables:

AND

Input: x	Input: y	Output: z
0	0	0
0	1	0
1	0	0
1	1	1

OR

Input: x	Input: y	Output: z
0	0	0
0	1	1
1	0	1
1	1	1

How can we make them secure?

If Alice crafts a logic circuit and then hands it to Bob, Bob can easily determine what's happening inside the circuit. This offers Alice no secrecy in terms of what Bob can compute. How can we secure logic circuits to prevent this? Hint: **garbled circuits.**

Let's say Alice wants to "encrypt" a logic circuit which simply consists of a single AND gate. This means the entire circuit can be represented using the logic table from before. However, what if Alice changes the input bits from simple 1's and 0's to keys? To do this, she needs to generate four keys: a 0 and a 1 key for the x input, and a 0 and a 1 key for the y input. She would then have:

AND

Input: x	Input: y	Output: z
$k_{x,0}$	$k_{y,0}$	
$k_{x,0}$	$k_{y,1}$	
$k_{x,1}$	$k_{y,0}$	
$k_{x,1}$	$k_{y,1}$	

How should she make the output for the new truth table?

Encryption

Let's say Alice has a secure symmetric encryption function $\boldsymbol{E(k,p)}$ which takes as its input a key $\boldsymbol{k}$ and a plaintext $\boldsymbol{p}$ as well as a matching decrypt function $\boldsymbol{D(k,c)}$ which decrypts a ciphertext $\boldsymbol{c}$. Then for each row, she can determine the output by first encrypting with the key from the y column and then encrypting that with the key from the x column. This leaves her the following truth table:

AND

Input: x	Input: y	Output: z
$k_{x,0}$	$k_{y,0}$	$E\left(k_{x,0}, E\left(k_{y,0}, 0\right)\right)$
$k_{x,0}$	$k_{y,1}$	$E\left(k_{x,0}, E\left(k_{y,1}, 0\right)\right)$
$k_{x,1}$	$k_{y,0}$	$E\left(k_{x,1}, E\left(k_{y,0}, 0\right)\right)$
$k_{x,1}$	$k_{y,1}$	$E\left(k_{x,1}, E\left(k_{y,1}, 1\right)\right)$

Decryption

The beauty of this table is that given one x key and one y key, Bob can only decrypt one of the output values. Let's say Bob has $k_{x,1}$ and $k_{y,0}$ and all the possible outputs. Let's see what happens when he tries to decrypt all the outputs:

$$D(k_{x,1}, E\left(k_{x,0}, E\left(k_{y,0}, 0\right)\right)) = FAIL$$

$$D(k_{x,1}, E\left(k_{x,0}, E\left(k_{y,1}, 0\right)\right)) = FAIL$$

$$D(k_{x,1}, E\left(k_{x,1}, E\left(k_{y,0}, 0\right)\right)) = E\left(k_{y,0}, 0\right) => D\left(k_{y,0}, E\left(k_{y,0}, 0\right)\right) = 0$$

$$D(k_{x,1}, E\left(k_{x,1}, E\left(k_{y,1}, 0\right)\right)) = E\left(k_{y,1}, 0\right) => D\left(k_{y,0}, E\left(k_{y,1}, 0\right)\right) = FAIL$$

After the first round of decryption, Bob will fail on two of the possible outputs because the x keys differ. He will also fail on one of the decryptions in the second round, because the y key will fail. This leaves him with only one output, 0, which is the one that Alice wanted him to have.

Without the other keys, Bob has no way of knowing which truth table is used and gets no information about which values the input keys represent.

Applying this method to logic circuits is simple: simply encrypt each gate, making the inputs to the gate the possible outputs of the previous gate. Alice now has a way to send a circuit over to Bob and have him evaluate it without fear that he can determine her inputs or what the function even is.

One final problem: how can Bob get his inputs?

We now have an almost complete solution to the Millionaire problem:

1. Alice crafts a logic circuit which computes who has more money.
2. Alice encrypts the circuit using Yao's protocol.
3. Alice sends both the garbled circuit and her own input to Bob.
4. ?????
5. Bob runs the circuit (same as decrypting) using Alice's and his own input and tells Alice the answer.

Obviously, we have a problem. How can Bob get the inputs to the circuit from Alice? If he tells Alice his inputs and then Alice sends him the corresponding keys, this would defeat the whole point as Alice would know how much money Bob has. Alternatively, if Alice sent Bob all the possible inputs and then selected his own, this would allow Bob to repeatedly run the circuit using different inputs to determine how much money Alice has, yet again defeating the purpose. What can they do?

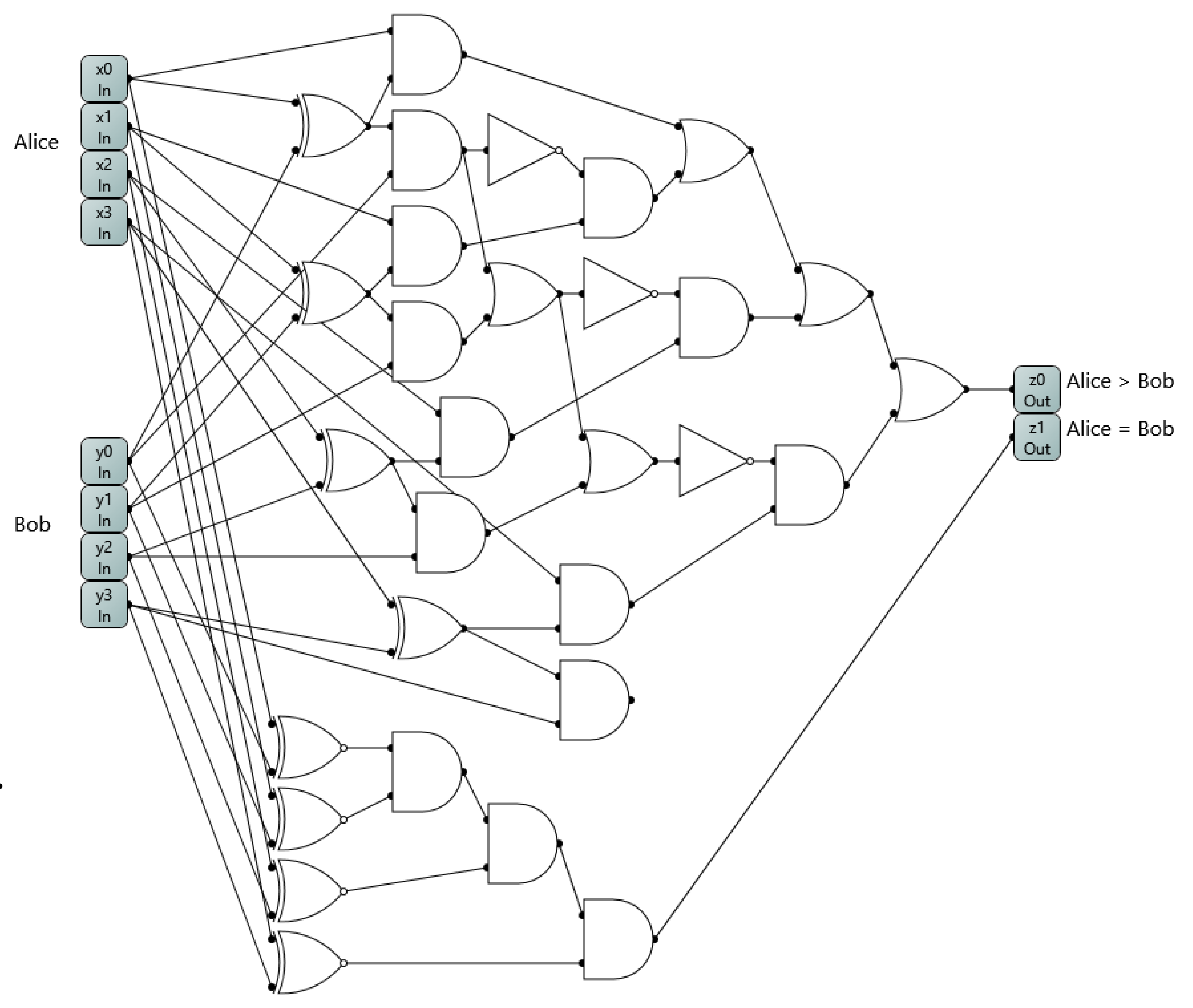
The solution: oblivious transfer

Oblivious transfer allows Alice to send Bob the input he wants while ensuring that she can not figure out Bob's input and that Bob does not get any other input other than what he has chosen.

Ya-Fen Chang authored an oblivious transfer protocol based on RSA that allows Bob to grab t out of n of Alice's inputs. This is perfect for the millionaire problem, as Bob will need to grab half of the possible inputs from Alice.

The millionaire problem circuit

Of course, to actually run the millionaire problem, Alice will need to make a circuit which compares her and Bob's wealth input and outputs whose is greater. A circuit which works with 4-bit can be seen below:


Alice
x0
In
x1
In
x2
In
x3
In
Bob
y0
In
y1
In
y2
In
y3
In
z0
Out
Alice > Bob
z1
Out
Alice = Bob