Garbled Circuits

Garbled Circuits in Python.

You can find our code at: https://github.com/viewv/garbled-circuits

Run

The main program of this project is the `circuit.py`, in the function `main` in this code file, you can config the a and b.

```python
def main():
    a = '01'
    b = '10'
    ans = ge(a, b)
    if ans == 0:
        print(a, b, 'a < b')
    elif ans == 1:
        print(a, b, 'a ⩾ b')
    else:
        print('error!')
```

By default, the a is `01` and the b is `10`. And we can run the `circuit.py`, and we get the output.

```
~/Develop/garbled-circuits main
base ❱ python -u "/Users/home/Develop/garbled-circuits/circuit.py"
01 10 a < b
```

If we change the a into `10` and b into `01`. And we have:

```
base ❱ python -u "/Users/home/Develop/garbled-circuits/circuit.py"
10 01 a >= b
```

Our garbled circuit works.

Structure & Introduce

In this project, we have three python code files.

- `des.py` : We use the DES as PRF.
- `cmp.py` : For each gate in the circuit, `cmp.py` is used to calculate the truth table of each gate.
- `circuit.py` : Implementation of the circuit.

For ENC function, the encrypt function, we have the code below:

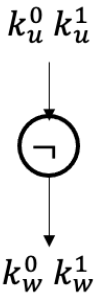
```python
def ENC(k, x):
    r = gen()
    return [r, xor(PRF_F(r, k), x + ''.zfill(64))]

def ENC_R(k, r, x):
    return xor(PRF_F(r, k), x + ''.zfill(64))
```

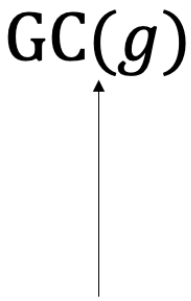
The PRF_F is the PRF using DES, and we have two types of the encrypt function for different gates.

For the NOT gate, we only need to perform for once.

NOT gate g



Choose labels



u	w
0	1
1	0

Truth table

$\mathbf{Enc}(k_u^0, k_w^1)$
$\mathbf{Enc}(k_u^1, k_w^0)$

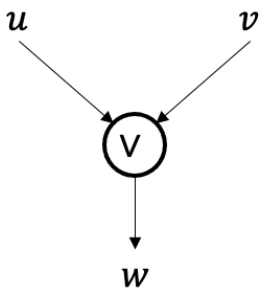
Encrypted gate

$\mathbf{Enc}(k_u^1, k_w^0)$
$\mathbf{Enc}(k_u^0, k_w^1)$

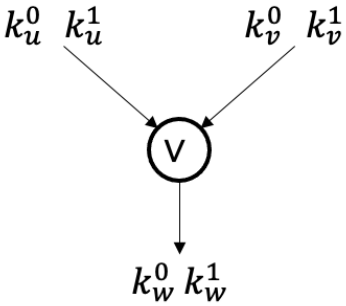
Permuted Encrypted gate

So we need to use the ENC function to generate one random r to perform the encryption.

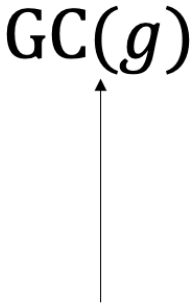
For the AND or OR gate, we need to perform double encryption.



OR gate g



Choose labels



u	v	w
0	0	0
0	1	1
1	0	1
1	1	1

Truth table

$\mathbf{Enc}(k_u^0, \mathbf{Enc}(k_v^0, k_w^0))$
$\mathbf{Enc}(k_u^0, \mathbf{Enc}(k_v^1, k_w^1))$
$\mathbf{Enc}(k_u^1, \mathbf{Enc}(k_v^0, k_w^1))$
$\mathbf{Enc}(k_u^1, \mathbf{Enc}(k_v^1, k_w^1))$

Encrypted gate

$\mathbf{Enc}(k_u^1, \mathbf{Enc}(k_v^0, k_w^1))$
$\mathbf{Enc}(k_u^0, \mathbf{Enc}(k_v^0, k_w^0))$
$\mathbf{Enc}(k_u^1, \mathbf{Enc}(k_v^1, k_w^1))$
$\mathbf{Enc}(k_u^0, \mathbf{Enc}(k_v^1, k_w^1))$

Permuted Encrypted gate

The challenge is that if we still use the normal ENC function, the output of the inner encryption is at length $3n$ since the encryption is $\mathrm{Enc}(k, x) = (r, F_k(r) \oplus (x \| 0^n))$. And we can not use the output from the inner encryption again, so we transform the double encryption into one encryption. We use the ENC_R function, and let the k as k_u, the r as k_v and the x as the k_w. The keys are all random generated. We take the AND gate for one example:

```
def AND(ku0, ku1, kv0, kv1, kw0, kw1):
    a = ENC_R(ku0, kv0, kw0)
    b = ENC_R(ku0, kv1, kw0)
    c = ENC_R(ku1, kv0, kw0)
    d = ENC_R(ku1, kv1, kw1)
    ans = [a, b, c, d]
    random.shuffle(ans)
    return ans
```

We perform encryption for all, and finall shuffle the table. Finally, we use these gates to group the circuit in `circuit.py`. The `ge(a,b)` function simulate the process to use the garbled circuits:

```python
def ge(a, b):
    output = build_circuit()
    k160, k161 = output
    a1, a0 = int(a[0]), int(a[1])
    b1, b0 = int(b[0]), int(b[1])
    alice_set_key(a1, a0)
    result = evaluate(b1, b0)
    if result == k160:
        return 0
    elif result == k161:
        return 1
    else:
        return -1
```

First the Alice uses `output = build_circuit()` to build the circuit, and get the possible output (in code is that she generate all) `k160, k161 = output`. And then Alice sets the value of her using `alice_set_key(a1, a0)`. Then the Bob uses his value to evaluate the circuit using `result = evaluate(b1, b0)`. And finally, the Alice can get the result from Bob's result.