# Secure Computation in Online Social Networks

Presenter: Yi LIU

# Secure Multiparty Computation

- Lover matching

# Secure Multiparty Computation

*Protocols for Secure Computations.* *FOCS’82*

Andrew Yao 姚期智
Turing Award (2000)

# Secure Multiparty Computation

*Protocols for Secure Computations.* *FOCS’82*

Yao's Millionaires' Problem

Andrew Yao 姚期智
Turing Award (2000)

# Secure Multiparty Computation

*Protocols for Secure Computations.* *FOCS'82*

## Yao's Millionaires' Problem

For definiteness, suppose Alice has $i$ millions and Bob has $j$ millions, where $1 < i, j < 10$. We need a protocol for them to decide whether $i < j$, such that this is also the only thing they know in the end (aside from their own values). Let $M$ be the set of all $N$-bit nonnegative integers, and $Q_N$ be the set of all 1-1 onto functions from $M$ to $M$. Let $E_a$ be the public key of Alice, generated by choosing a random element from $Q_N$.

1. Bob picks a random $N$-bit integer, and computes privately the value of $E_a(x)$; call the result $k$.
2. Bob sends Alice the number $k - j + 1$;
3. Alice computes privately the values of $y_u = D_a(k - j + u)$ for $u = 1, 2, \ldots, 10$.
4. Alice generates a random prime $p$ of $N/2$ bits, and computes the values $z_u = y_u \pmod{p}$ for all $u$; if all $z_u$ differ by at least $2$ in the $\bmod p$ sense, stop; otherwise generates another random prime and repeat the process until all $z_u$ differ by at least $2$; let $p$, $z_u$ denote this final set of numbers;
5. Alice sends the prime $p$ and the following $10$ numbers to $B$: $z_1, z_2, \ldots, z_i$ followed by $z_i + 1, z_{i+1} + 1, \ldots, z_{10} + 1$; the above numbers should be interpreted in the $\bmod p$ sense.
6. Bob looks at the $j$-th number (not counting $p$) sent from Alice, and decides that $i \geq j$ if it is equal to $x \bmod p$, and $i < j$ otherwise.
7. Bob tells Alice what the conclusion is.

Andrew Yao 姚期智
Turing Award (2000)

# Secure Multiparty Computation

## Why (mod p)?

1. Bob picks a random $N$-bit integer, and computes privately the value of $E_a(x)$; call the result $k$.
2. Bob sends Alice the number $k - j + 1$;
3. Alice computes privately the values of $y_u = D_a(k - j + u)$ for $u = 1, 2, \ldots, 10$.
4. Alice generates a random prime $p$ of $N/2$ bits, and computes the values $z_u = y_u \pmod p$ for all $u$; if all $z_u$ differ by at least 2 in the $\bmod\ p$ sense, stop; otherwise generates another random prime and repeat the process until all $z_u$ differ by at least 2; let $p$, $z_u$ denote this final set of numbers;
5. Alice sends the prime $p$ and the following 10 numbers to $B$: $z_1, z_2, \ldots, z_i$ followed by $z_i + 1, z_{i+1} + 1, \ldots, z_{10} + 1$; the above numbers should be interpreted in the $\bmod\ p$ sense.
6. Bob looks at the $j$-th number (not counting $p$) sent from Alice, and decides that $i \geq j$ if it is equal to $x \bmod p$, and $i < j$ otherwise.
7. Bob tells Alice what the conclusion is.

Andrew Yao 姚期智
Turing Award (2000)

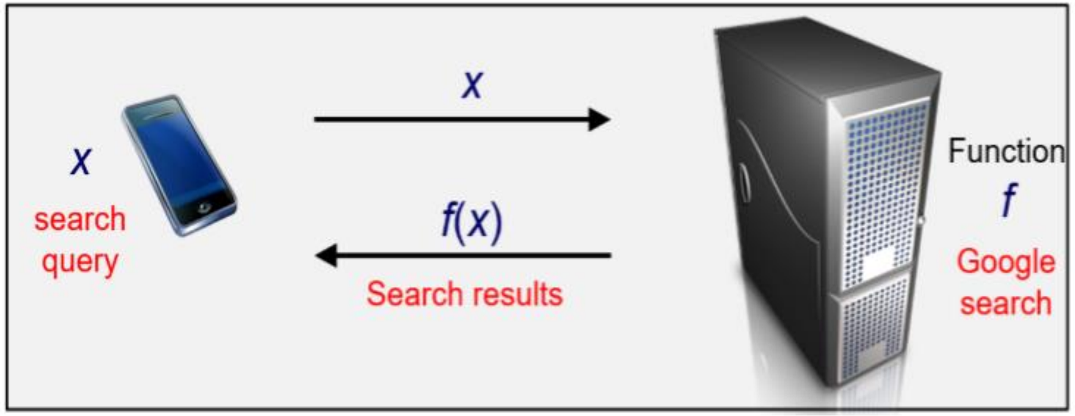
# Homomorphic Encryption

Two groups $G$ and $G'$ are *homomorphic* if there exists a function (*homomorphism*) $f : G \to G'$ such that for all $x, y \in G$, $f(x +_G y) = f(x) +_{G'} f(y)$.

# Homomorphic Encryption

Two groups $G$ and $G'$ are *homomorphic* if there exists a function (*homomorphism*) $f : G \to G'$ such that for all $x, y \in G$, $f(x +_G y) = f(x) +_{G'} f(y)$.

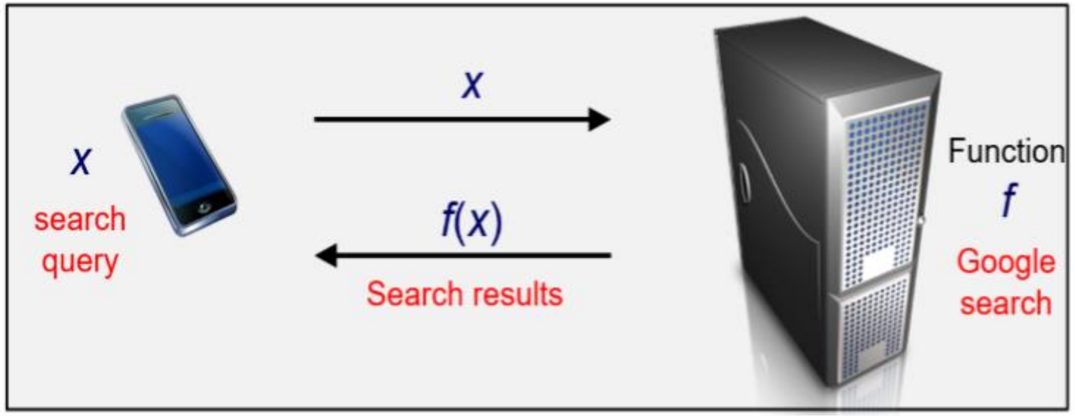
Why do we need *homomorphic encryption*?

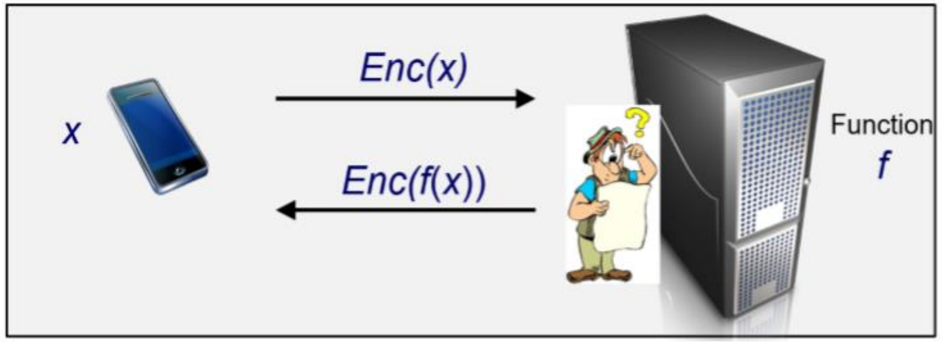


# Homomorphic Encryption

Two groups $G$ and $G'$ are *homomorphic* if there exists a function (*homomorphism*) $f : G \to G'$ such that for all $x, y \in G$, $f(x +_G y) = f(x) +_{G'} f(y)$.

Why do we need *homomorphic encryption*?



# Homomorphic Encryption

Recall RSA encryption

$E(m_1) = m_1^e \bmod n$, $E(m_2) = m_2^e \bmod n$

# Homomorphic Encryption

Recall RSA encryption

$E(m_1) = m_1^e \bmod n$, $E(m_2) = m_2^e \bmod n$

$E(m_1) \cdot E(m_2) = m_1^e \cdot m_2^e = (m_1 \cdot m_2)^e = E(m_1 \cdot m_2)$

# Homomorphic Encryption

Recall RSA encryption

$E(m_1) = m_1^e \bmod n$, $E(m_2) = m_2^e \bmod n$

$E(m_1) \cdot E(m_2) = m_1^e \cdot m_2^e = (m_1 \cdot m_2)^e = E(m_1 \cdot m_2)$

RSA is multiplicatively homomorphic, but not additively homomorphic.

# Homomorphic Encryption

Recall RSA encryption

$E(m_1) = m_1^e \bmod n$, $E(m_2) = m_2^e \bmod n$

$E(m_1) \cdot E(m_2) = m_1^e \cdot m_2^e = (m_1 \cdot m_2)^e = E(m_1 \cdot m_2)$

RSA is multiplicatively homomorphic, but not additively homomorphic.

Paillier cryptosystem: additively homomorphic

# Homomorphic Encryption

Recall RSA encryption

$$E(m_1) = m_1^e \bmod n,\ E(m_2) = m_2^e \bmod n$$

$$E(m_1) \cdot E(m_2) = m_1^e \cdot m_2^e = (m_1 \cdot m_2)^e = E(m_1 \cdot m_2)$$

RSA is multiplicatively homomorphic, but not additively homomorphic.

Paillier cryptosystem (*EUROCRYPT'99)*: additively homomorphic

The original system: semantic security against chosen-plaintext attacks (IND-CPA)

The improved system: IND-CCA2 secure in the random oracle model

# Homomorphic Encryption

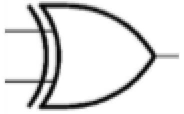
We need **both**!

What people really wanted was the ability to do arbitrary computing on encrypted data, and this requires the abibility to compute both sums and products.

# Homomorphic Encryption

Why SUMs and PRODUCTs?

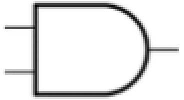
SUM



**XOR**

$x + y \bmod 2$

PRODUCT



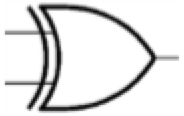
**AND**

$x \cdot y \bmod 2$

# Homomorphic Encryption

Why SUMs and PRODUCTs?

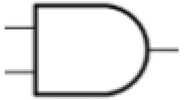
SUM



**XOR**

$x + y \bmod 2$

PRODUCT



**AND**

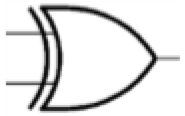
$x \cdot y \bmod 2$

{XOR, AND} is complete, i.e.,
any function is a combination of XOR and AND. (e.g., OR)

# Homomorphic Encryption

Why SUMs and PRODUCTs?

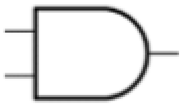
SUM



**XOR**

$x + y \bmod 2$

PRODUCT



**AND**

$x \cdot y \bmod 2$

{XOR, AND} is complete, i.e.,
any function is a combination of XOR and AND. (e.g., OR)

**Example**

$x\ OR\ y = x + y + x \cdot y \bmod 2.$

# Homomorphic Encryption

Because {XOR, AND} is *complete*, if we can compute SUMs and PRODUCTs on encrypted bits, we can compute any function on encrypted inputs.

# Homomorphic Encryption

Because {XOR, AND} is *complete*, if we can compute SUMs and PRODUCTs on encrypted bits, we can compute any function on encrypted inputs.

*Fully-homomorphic encryption*!

We can delegate arbitrary processing of data without giving away access to it.

# Homomorphic Encryption

Because {XOR, AND} is *complete*, if we can compute SUMs and PRODUCTs on encrypted bits, we can compute any function on encrypted inputs.

*Fully-homomorphic encryption*!

We can delegate arbitrary processing of data without giving away access to it.

**Applications**: *private cloud computing, private information retrieval, multi-party secure computation, encrypted search, ...*

# Homomorphic Encryption

STOC'09

**Fully Homomorphic Encryption Using Ideal Lattices**

Craig Gentry
Stanford University and IBM Watson
cgentry@cs.stanford.edu

**ABSTRACT**

We propose a fully homomorphic encryption scheme – i.e., a scheme that allows one to evaluate circuits over encrypted data without being able to decrypt. Our solution comes in three steps. First, we provide a general result – that, to construct an encryption scheme that permits evaluation of *arbitrary circuits*, it suffices to construct an encryption

duced by Rivest, Adleman and Dertouzos [54] shortly after the invention of RSA by Rivest, Adleman and Shamir [55]. Basic RSA is a multiplicatively homomorphic encryption scheme – i.e., given RSA public key pk $= (N, e)$ and ciphertexts $\{\psi_i \leftarrow \pi_i^e \bmod N\}$, one can efficiently compute $\prod_i \psi_i = (\prod_i \pi_i)^e \bmod N$, a ciphertext that encrypts the product of the original plaintexts. Rivest et al. [54] asked

**Fully Homomorphic Encryption over the Integers**

Marten van Dijk[1], Craig Gentry[2], Shai Halevi[2], and Vinod Vaikuntanathan[2]

[1] MIT CSAIL
[2] IBM Research

EUROCRYPT'10

**Abstract.** We construct a simple fully homomorphic encryption scheme, using only elementary modular arithmetic. We use Gentry's technique to construct a fully homomorphic scheme from a "bootstrappable" somewhat homomorphic scheme. However, instead of using ideal lattices over a

Craig Gentry

# Server-Aided Secure Computation with Off-line Parties

- ESORICS’17

# Server-Aided Secure Computation with Off-line Parties

- ESORICS’17
- Extend version: *Secure Computation in Online Social Networks*

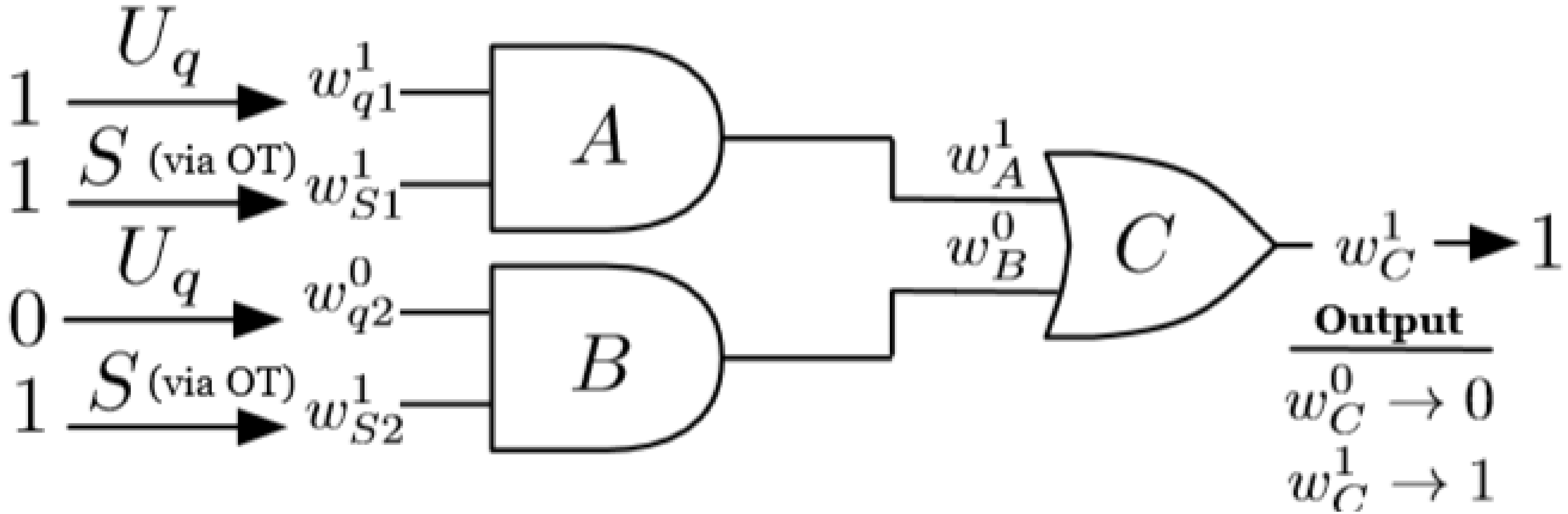
# Server-Aided Secure Computation with Off-line Parties

- Foteini Baldimtsi1, Dimitrios Papadopoulos, Stavros Papadopoulos, Alessandra Scafuro, and Nikos Triandopoulos
- ESORICS’17
- Extend version: *Secure Computation in Online Social Networks*
- Contribution
  - First MPC model that is specifically tailored for secure computation in the OSN setting (efficiency, friend non-participation and data re-usability)
  - Two very well-studied techniques from secure two-party computation (garbled circuits and mixed protocols) can be adapted for use in this setting
  - Implementation and experimental evaluation

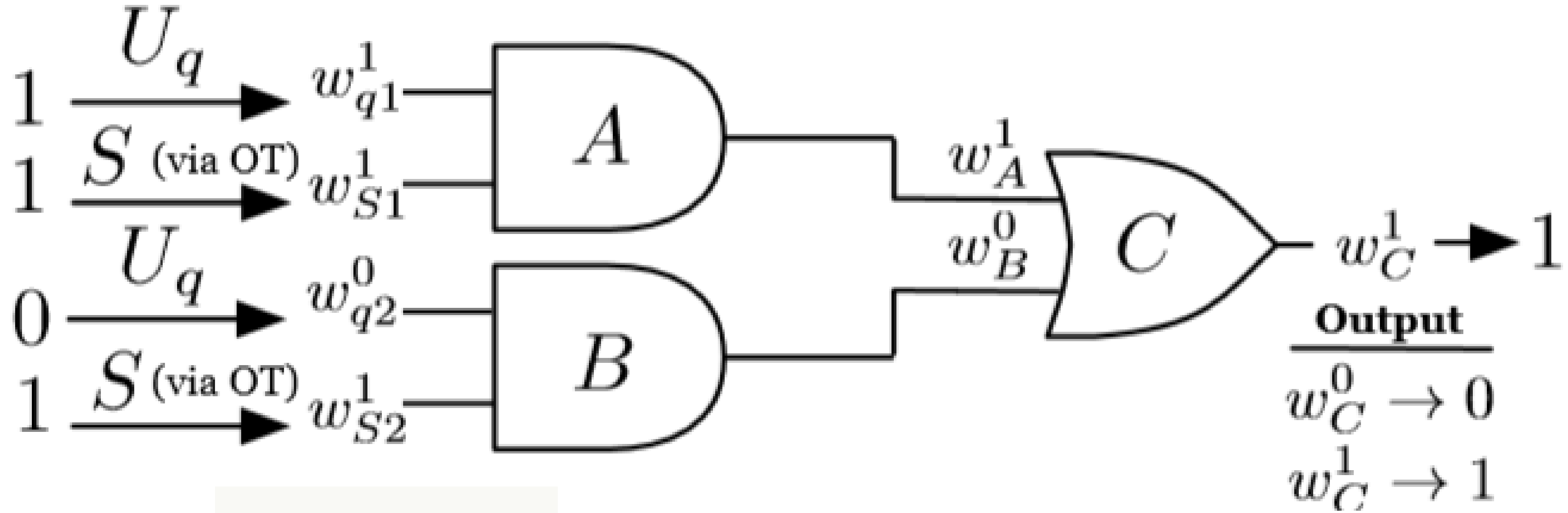
# Assumption

- No collusion between server and users!

# Garbled circuit



# Garbled circuit




How to obtain $w^{*}_{S*}$

# Protocol

1. $\mathsf{Join}\langle U_i(1^\lambda), S(\mathcal{G})\rangle$: On input $1^\lambda$, $U_i$ randomly chooses a PRF key $K_i \in \{0,1\}^\lambda$, and sends her public-key $pk_i$ to $S$. $S$ adds $v_i$ initialized with value $pk_i$ into $\mathcal{V}$ of $\mathcal{G}$.
2. $\mathsf{Connect}\langle U_i(K_i), U_j(K_j)\rangle$: $U_i$ receives the public key $pk_j$ of $U_j$ from $S$. Sets $k_{i\to j}$ to $E'(pk_j, K_i)$ and sends it to $S$. $U_j$ computes and sends $k_{j\to i}$ to $S$ who then creates edge $e_{ij}$ storing $k_{i\to j}$, $k_{j\to i}$, and adds it to $\mathcal{E}$ of $\mathcal{G}$.
3. $\mathsf{Upload}\langle U_i(K_i, x_i), S(\mathcal{G})\rangle$: $U_i$ chooses nonce $r_i$, computes value $X_{il}^{x_i[l]}$ as $F_{K_i}(x_i[l], l, r_i)$ $\forall\ l \in [\ell]$, and sends them to $S$ who stores the value $c_i = ((X_{i1}^{x_i[1]}, \ldots, X_{i\ell}^{x_i[\ell]}), r_i)$ in $v_i$.
4. $\mathsf{Query}\langle U_q(K_q, \alpha), S(\mathcal{G})\rangle(f)$: $U_q$ does the following:
   (a) **Key and nonce retrieval.** For each $U_j \in \mathcal{G}_q$, retrieve key $k_{j\to q}$ and (latest) nonce $r_j$ from $S$, and decrypt $k_{j\to q}$ to get $K_j$.
   (b) **Garbled circuit computation.** $U_q$ transforms $f$ into a circuit, and garbles it as $GC$.
   (c) **Selection table generation.** For each user $U_j$ in $\mathcal{G}_q$ and index $l \in [\ell]$:
   *Compute selection keys:* Generate $s_{jl}^0 = F_{K_j}(0, l, r_j)$, $s_{jl}^1 = F_{K_j}(1, l, r_j)$.
   *Compute garbled inputs:* Produce encryptions $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ with the selection keys.
   *Set selection table entry:* Store $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ into $T_q[j, l]$ in a random order.
   (d) **Circuit transmission.** Send $GC$, $T_q$ to $S$.
   $S$ then decrypts the garbled values of each $U_j \in \mathcal{G}_q$ from $T_q$, with the encoding $X_{jl}^{x_j[l]}$ for each $l \in [\ell]$. He evaluates $GC$ and sends output to $U_q$ who Obtains the result $y$ by decoding the circuit output.

# Protocol

1. $\mathsf{Join}\langle U_i(1^\lambda), S(\mathcal{G})\rangle$: On input $1^\lambda$, $U_i$ randomly chooses a PRF key $K_i \in \{0,1\}^\lambda$, and sends her public-key $pk_i$ to $S$. $S$ adds $v_i$ initialized with value $pk_i$ into $\mathcal{V}$ of $\mathcal{G}$.

# Protocol

2. $\mathsf{Connect}\langle U_i(K_i), U_j(K_j)\rangle$: $U_i$ receives the public key $pk_j$ of $U_j$ from $S$. Sets $k_{i\to j}$ to $E'(pk_j, K_i)$ and sends it to $S$. $U_j$ computes and sends $k_{j\to i}$ to $S$ who then creates edge $e_{ij}$ storing $k_{i\to j}$, $k_{j\to i}$, and adds it to $\mathcal{E}$ of $\mathcal{G}$.

# Protocol

3. Upload$\langle U_i(K_i, x_i), S(\mathcal{G})\rangle$: $U_i$ chooses nonce $r_i$, computes value $X_{il}^{x_i[l]}$ as $F_{K_i}(x_i[l], l, r_i)$ $\forall$ $l \in [\ell]$, and sends them to $S$ who stores the value $c_i = ((X_{i1}^{x_i[1]}, \ldots, X_{i\ell}^{x_i[\ell]}), r_i)$ in $v_i$.

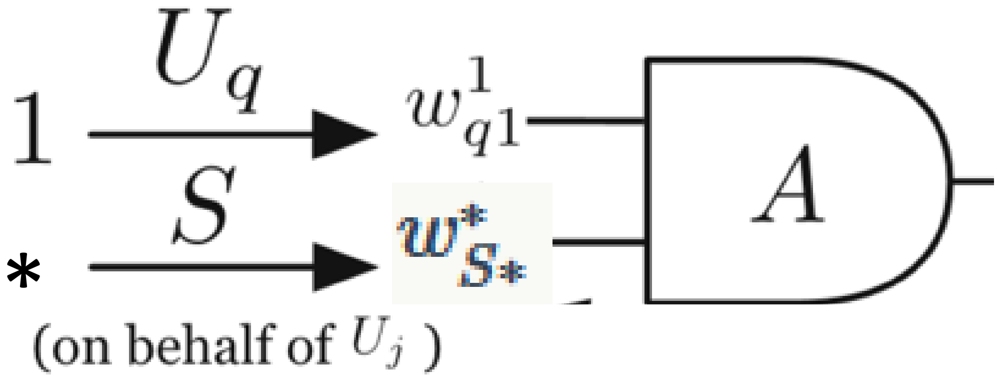
# Protocol

3. Upload$\langle U_i(K_i, x_i), S(\mathcal{G})\rangle$: $U_i$ chooses nonce $r_i$, computes value $X_{il}^{x_i[l]}$ as $F_{K_i}(x_i[l], l, r_i)$ $\forall$ $l \in [\ell]$, and sends them to $S$ who stores the value $c_i = ((X_{i1}^{x_i[1]}, \ldots, X_{i\ell}^{x_i[\ell]}), r_i)$ in $v_i$.

$$F_{K_a}(x_1, 1, r)$$

# Protocol

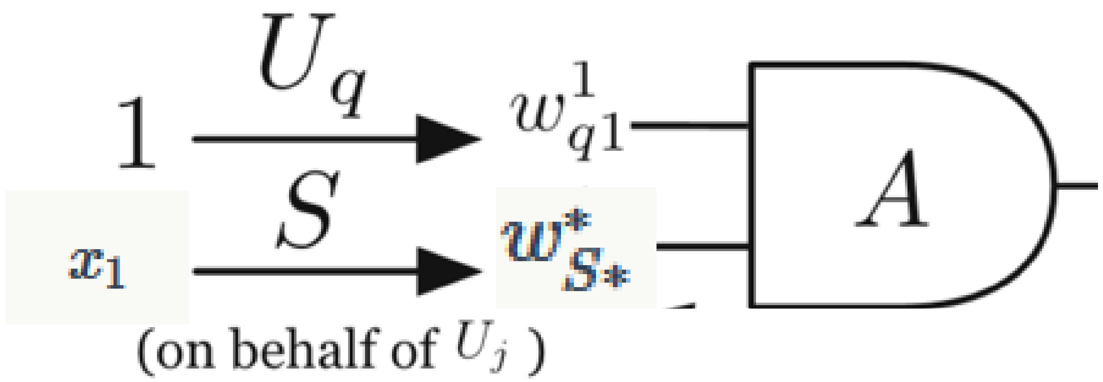
$$F_{K_a}(x_1, 1, r)$$



# Protocol

How to obtain $w^*_{S*}$ without learning $x_1$

$$F_{K_a}(x_1, 1, r)$$



# Protocol

4. Query$\langle U_q(K_q, \alpha), S(\mathcal{G})\rangle(f)$: $U_q$ does the following:
   (a) **Key and nonce retrieval.** For each $U_j \in \mathcal{G}_q$, retrieve key $k_{j\to q}$ and (latest) nonce $r_j$ from $S$, and decrypt $k_{j\to q}$ to get $K_j$.
   (b) **Garbled circuit computation.** $U_q$ transforms $f$ into a circuit, and garbles it as $GC$.
   (c) **Selection table generation.** For each user $U_j$ in $\mathcal{G}_q$ and index $l \in [\ell]$:
   *Compute selection keys:* Generate $s^0_{jl} = F_{K_j}(0, l, r_j)$, $s^1_{jl} = F_{K_j}(1, l, r_j)$.
   *Compute garbled inputs:* Produce encryptions $E_{s^0_{jl}}(w^0_{jl})$ and $E_{s^1_{jl}}(w^1_{jl})$ with the selection keys.
   *Set selection table entry:* Store $E_{s^0_{jl}}(w^0_{jl})$ and $E_{s^1_{jl}}(w^1_{jl})$ into $T_q[j, l]$ in a random order.
   (d) **Circuit transmission.** Send $GC, T_q$ to $S$.
   $S$ then decrypts the garbled values of each $U_j \in \mathcal{G}_q$ from $T_q$, with the encoding $X^{x_j[l]}_{jl}$ for each $l \in [\ell]$. He evaluates $GC$ and sends output to $U_q$ who Obtains the result $y$ by decoding the circuit output.

# Protocol

4. $\mathsf{Query}\langle U_q(K_q, \tilde{\alpha}), \tilde{S}(\mathcal{G})\rangle(f)$: $U_q$ does the following:
   (a) **Key and nonce retrieval.** For each $U_j \in \mathcal{G}_q$, retrieve key $k_{j\to q}$ and (latest) nonce $r_j$ from $S$, and decrypt $k_{j\to q}$ to get $K_j$.

# Protocol

4. Query$\langle U_q(K_q, \alpha), S(\mathcal{G})\rangle(f)$: $U_q$ does the following:
   (a) **Key and nonce retrieval.** For each $U_j \in \mathcal{G}_q$, retrieve key $k_{j\to q}$ and (latest) nonce $r_j$ from $S$, and decrypt $k_{j\to q}$ to get $K_j$.

Get $K_a$ r

# Protocol

(b) **Garbled circuit computation.** $U_q$ transforms $f$ into a circuit, and garbles it as $GC$.

(c) **Selection table generation.** For each user $U_j$ in $\mathcal{G}_q$ and index $l \in [\ell]$:
*Compute selection keys:* Generate $s_{jl}^0 = F_{K_j}(0, l, r_j)$, $s_{jl}^1 = F_{K_j}(1, l, r_j)$.
*Compute garbled inputs:* Produce encryptions $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ with the selection keys.
*Set selection table entry:* Store $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ into $T_q[j, l]$ in a random order.

# Protocol

(b) **Garbled circuit computation.** $U_q$ transforms $f$ into a circuit, and garbles it as $GC$.

(c) **Selection table generation.** For each user $U_j$ in $\mathcal{G}_q$ and index $l \in [\ell]$:
*Compute selection keys:* Generate $s_{jl}^0 = F_{K_j}(0, l, r_j)$, $s_{jl}^1 = F_{K_j}(1, l, r_j)$.
*Compute garbled inputs:* Produce encryptions $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ with the selection keys.
*Set selection table entry:* Store $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ into $T_q[j, l]$ in a random order.

$F_{K_a}(x_1, 1, r)$

$K_a$

r

# Protocol

(b) **Garbled circuit computation.** $U_q$ transforms $f$ into a circuit, and garbles it as $GC$.

(c) **Selection table generation.** For each user $U_j$ in $\mathcal{G}_q$ and index $l \in [\ell]$:
*Compute selection keys:* Generate $s^0_{jl} = F_{K_j}(0, l, r_j)$, $s^1_{jl} = F_{K_j}(1, l, r_j)$.
*Compute garbled inputs:* Produce encryptions $E_{s^0_{jl}}(w^0_{jl})$ and $E_{s^1_{jl}}(w^1_{jl})$ with the selection keys.
*Set selection table entry:* Store $E_{s^0_{jl}}(w^0_{jl})$ and $E_{s^1_{jl}}(w^1_{jl})$ into $T_q[j, l]$ in a random order.

$$F_{K_a}(x_1, 1, r)$$

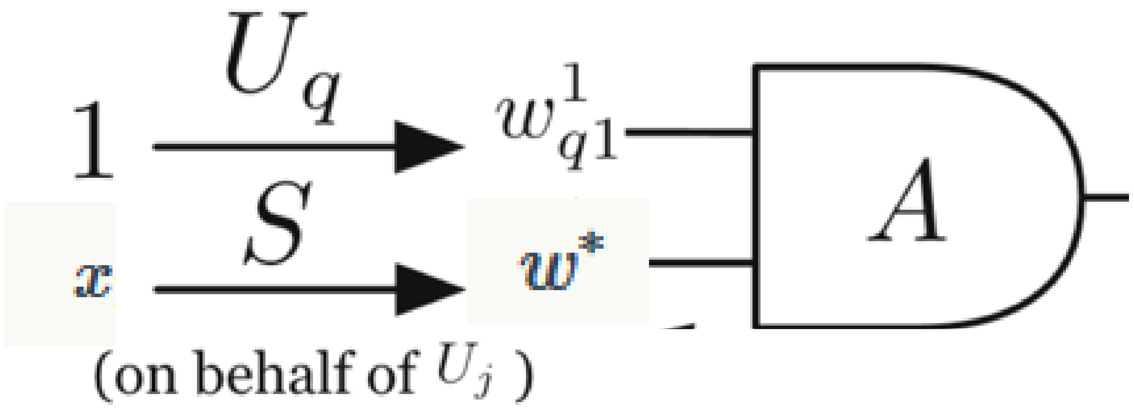
$$K_a$$

$$s^0 = F_{K_a}(0, 1, r)$$

$$s^1 = F_{K_a}(1, 1, r)$$

r

# Protocol




$F_{K_a}(x_1, 1, r)$

$K_a$

$s^0 = F_{K_a}(0, 1, r)$

$s^1 = F_{K_a}(1, 1, r)$

r

# Protocol

(b) **Garbled circuit computation.** $U_q$ transforms $f$ into a circuit, and garbles it as $GC$.

(c) **Selection table generation.** For each user $U_j$ in $\mathcal{G}_q$ and index $l \in [\ell]$:
*Compute selection keys:* Generate $s_{jl}^0 = F_{K_j}(0, l, r_j)$, $s_{jl}^1 = F_{K_j}(1, l, r_j)$.
*Compute garbled inputs:* Produce encryptions $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ with the selection keys.
*Set selection table entry:* Store $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ into $T_q[j, l]$ in a random order.

$$F_{K_a}(x_1, 1, r)$$

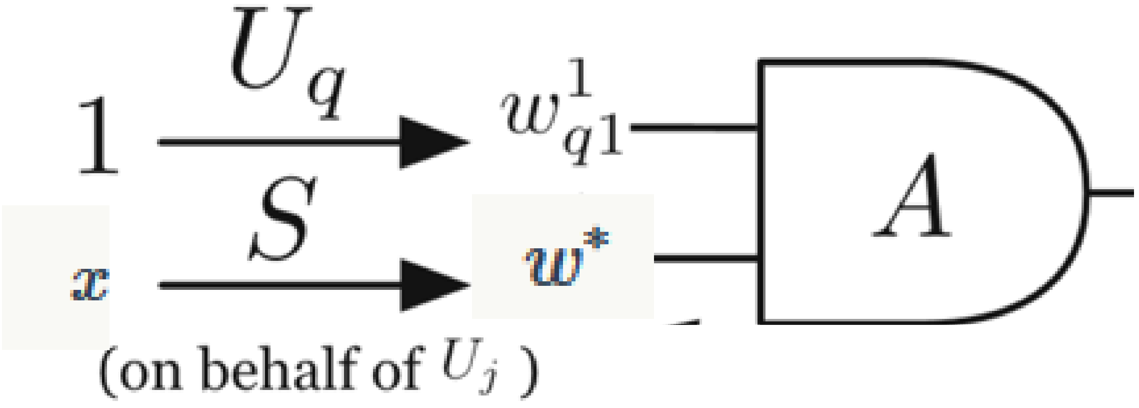
$$K_a$$

$$s^0 = F_{K_a}(0, 1, r)$$

$$s^1 = F_{K_a}(1, 1, r)$$

r

# Protocol




$$w^0, w^1$$

$$F_{K_a}(x_1, 1, r)$$

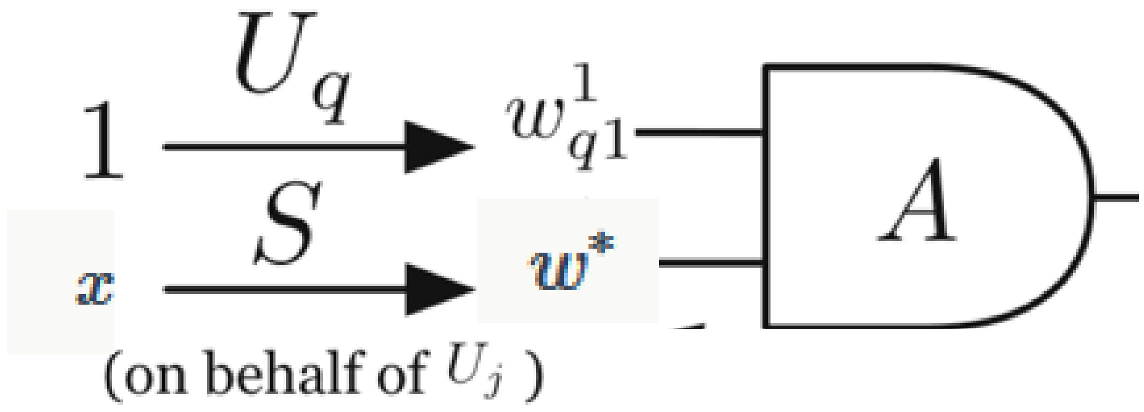
$$K_a$$

$$s^0 = F_{K_a}(0, 1, r)$$

$$s^1 = F_{K_a}(1, 1, r)$$

r

# Protocol




$$w^0, w^1$$

$$E_{s^0}(w^0), E_{s^1}(w^1)$$

$$F_{K_a}(x_1, 1, r)$$

$$K_a$$

$$s^0 = F_{K_a}(0, 1, r)$$

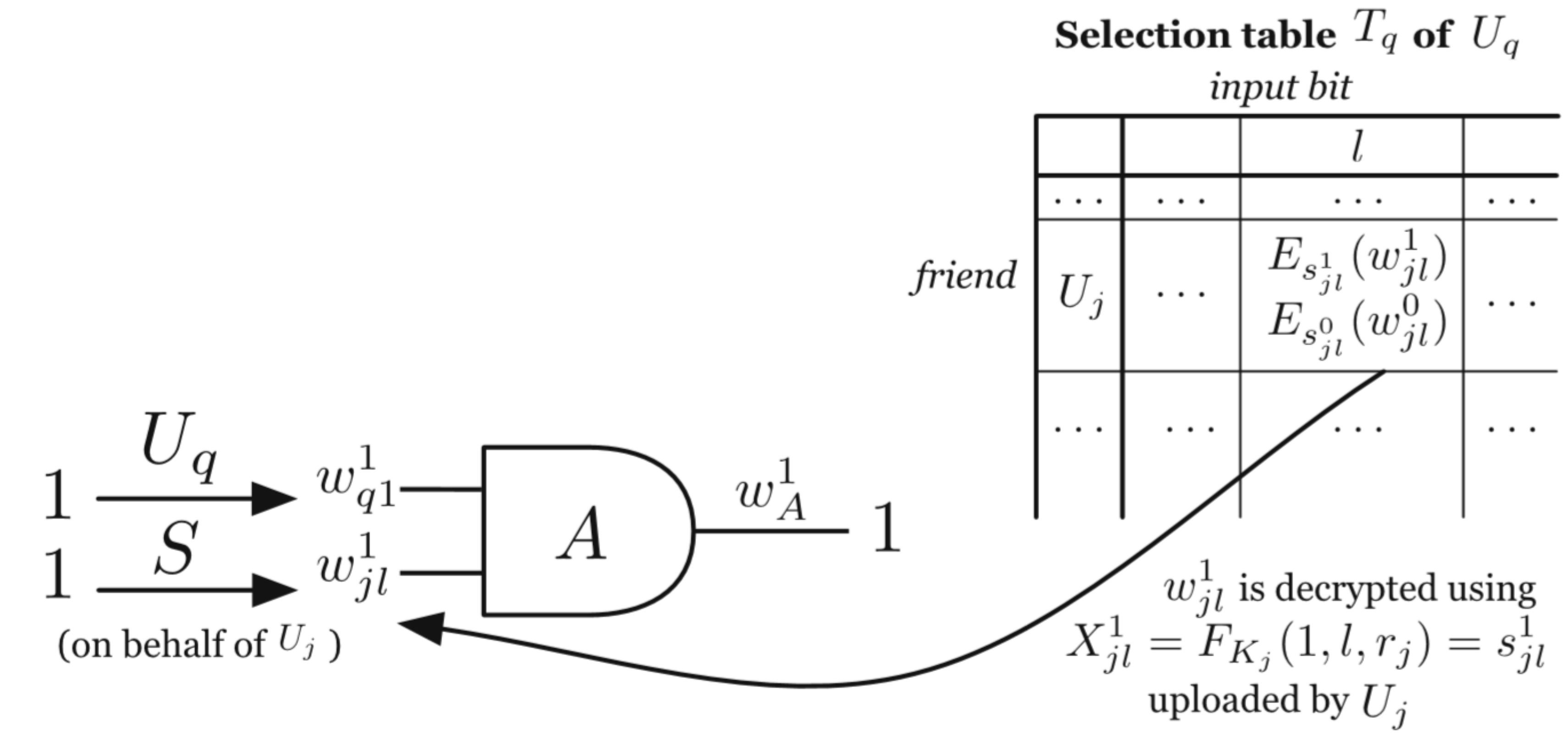
$$s^1 = F_{K_a}(1, 1, r)$$

r

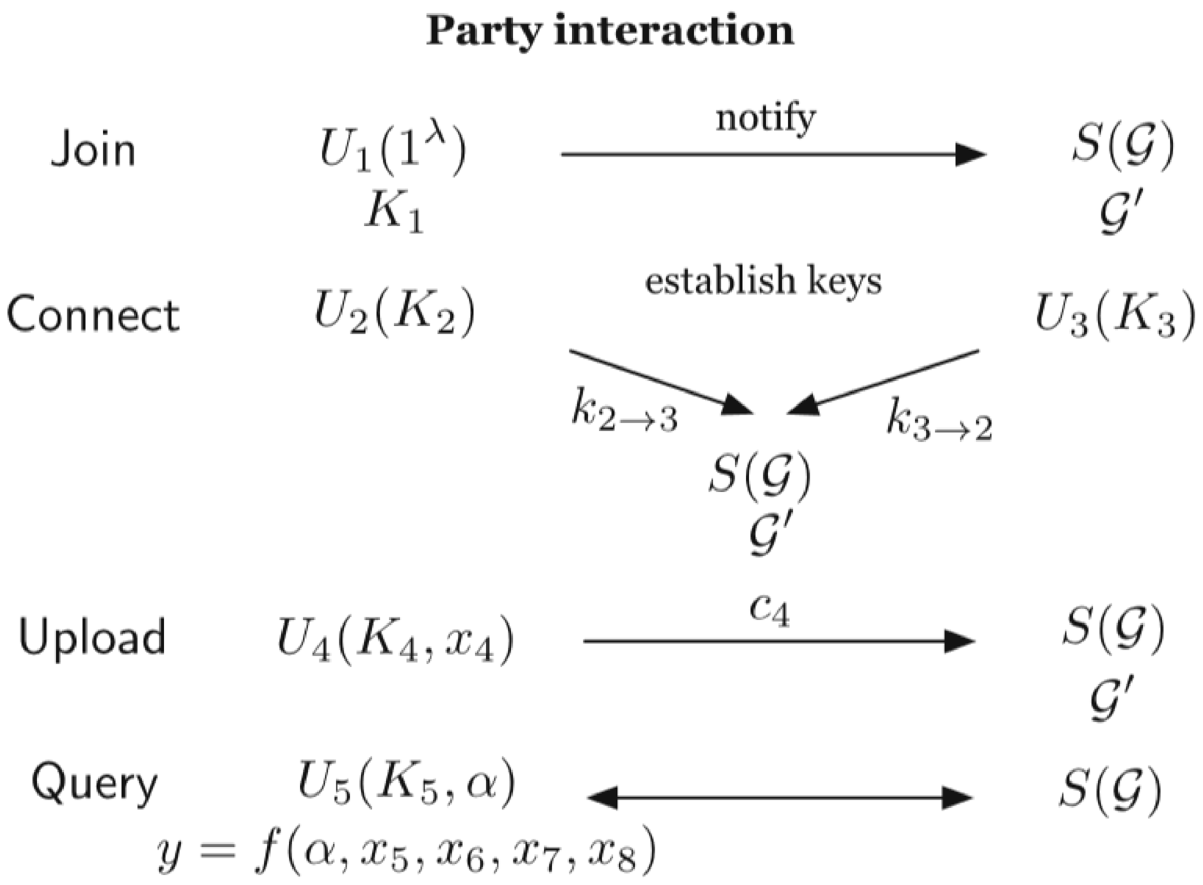
# Protocol

4. Query$\langle U_q(K_q, \alpha), S(\mathcal{G})\rangle(f)$: $U_q$ does the following:
   (a) **Key and nonce retrieval.** For each $U_j \in \mathcal{G}_q$, retrieve key $k_{j\to q}$ and (latest) nonce $r_j$ from $S$, and decrypt $k_{j\to q}$ to get $K_j$.
   (b) **Garbled circuit computation.** $U_q$ transforms $f$ into a circuit, and garbles it as $GC$.
   (c) **Selection table generation.** For each user $U_j$ in $\mathcal{G}_q$ and index $l \in [\ell]$:
   *Compute selection keys:* Generate $s_{jl}^0 = F_{K_j}(0, l, r_j)$, $s_{jl}^1 = F_{K_j}(1, l, r_j)$.
   *Compute garbled inputs:* Produce encryptions $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ with the selection keys.
   *Set selection table entry:* Store $E_{s_{jl}^0}(w_{jl}^0)$ and $E_{s_{jl}^1}(w_{jl}^1)$ into $T_q[j, l]$ in a random order.
   (d) **Circuit transmission.** Send $GC, T_q$ to $S$.
   $S$ then decrypts the garbled values of each $U_j \in \mathcal{G}_q$ from $T_q$, with the encoding $X_{jl}^{x_j[l]}$ for each $l \in [\ell]$. He evaluates $GC$ and sends output to $U_q$ who Obtains the result $y$ by decoding the circuit output.

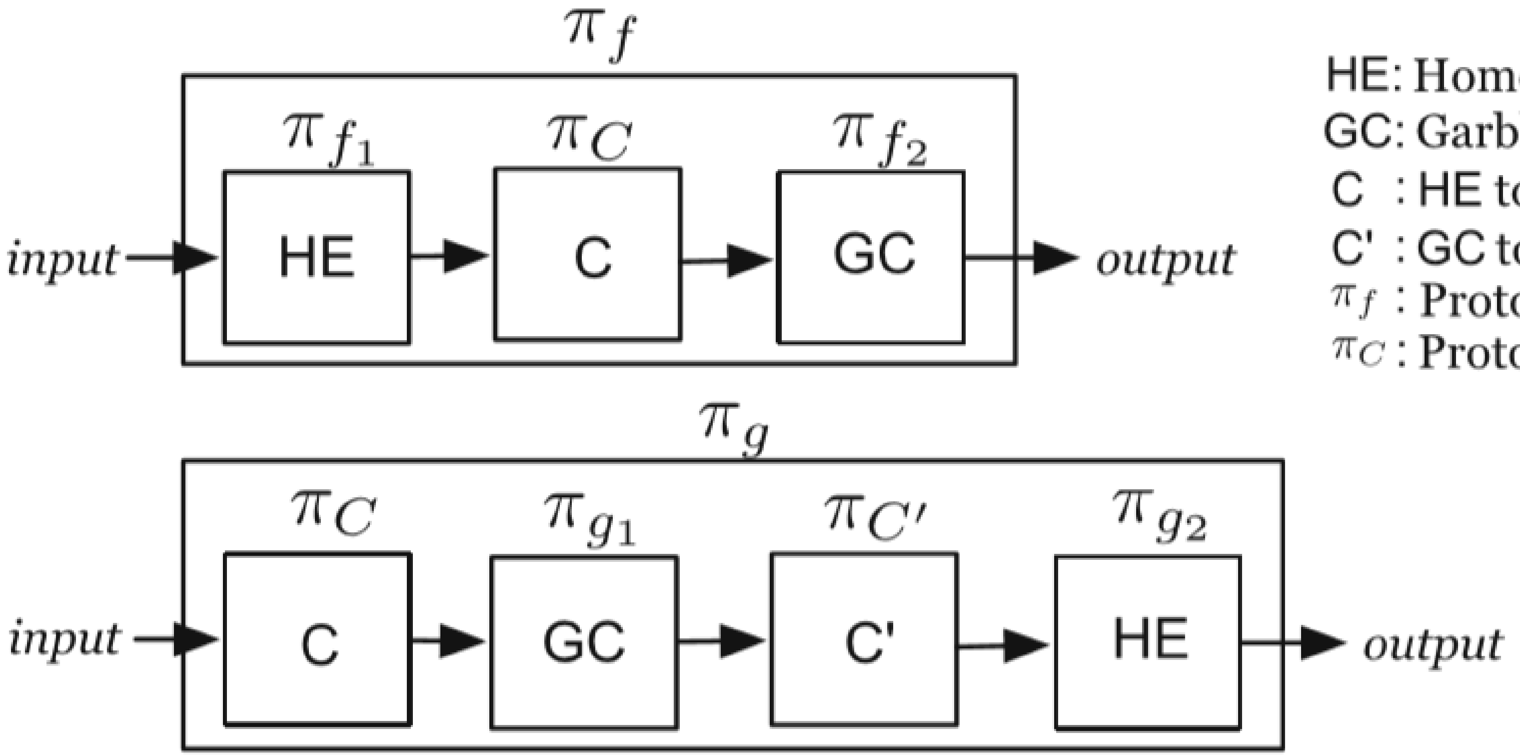
# Protocol


Selection table $T_q$ of $U_q$
*input bit*
$l$
*friend*
$U_j$
$E_{s^1_{jl}}(w^1_{jl})$
$E_{s^0_{jl}}(w^0_{jl})$
$U_q$
$S$
1
1
$w^1_{q1}$
$w^1_{jl}$
$A$
$w^1_A$
1
(on behalf of $U_j$ )
$w^1_{jl}$ is decrypted using
$X^1_{jl} = F_{K_j}(1, l, r_j) = s^1_{jl}$
uploaded by $U_j$

# Protocol


Party interaction
Join
$U_1(1^\lambda)$
$K_1$
notify
$S(\mathcal{G})$
$\mathcal{G}'$
Connect
$U_2(K_2)$
establish keys
$U_3(K_3)$
$k_{2\to 3}$
$k_{3\to 2}$
$S(\mathcal{G})$
$\mathcal{G}'$
Upload
$U_4(K_4, x_4)$
$c_4$
$S(\mathcal{G})$
$\mathcal{G}'$
Query
$U_5(K_5, \alpha)$
$S(\mathcal{G})$
$y = f(\alpha, x_5, x_6, x_7, x_8)$

# Mixed Protocol




HE: Homomorphic Encryption
GC: Garbled Circuit
C : HE to GC conversion
C' : GC to HE conversion
$\pi_f$ : Protocol securely evaluating $f$
$\pi_C$ : Protocol for conversion C

# Mixed Protocol

- How do server obtain $[[x_a]]$ while preserving the original value?

# Mixed Protocol

- How do server obtain $[[x_a]]$ while preserving the original value?

We need Re-Encryption Protocol!

# Re-Encryption Protocol

1. $\mathsf{Join}\langle U_i(1^\lambda), S(\mathcal{G})\rangle$: On input the security parameter $\lambda$, $U_i$ generates a PRF key $K_i$, and notifies $S$ that she joins the system by sending $pk_i$. $S$ adds node $v_i$ (initialized with $pk_i$) to graph $\mathcal{G}$.
2. $\mathsf{Connect}\langle U_i(K_i), U_j(K_j), S(\mathcal{G})\rangle$: Users $U_i$ and $U_j$, having each other public keys, compute $k_{j \to i} = [\![K_j]\!]_{pk_i}$, $k_{i \to j} = [\![K_i]\!]_{pk_j}$ respectively, and send them to $S$. Then, $S$ creates an edge $e_{ij}$ in $\mathcal{G}$ storing the two values.

# Re-Encryption Protocol

3. $\mathsf{Upload}\langle U_i(K_i, x_i), S(\mathcal{G})\rangle$: User $U_i$ picks random nonce $r_i$, computes $\rho_i = F_{K_i}(r_i)$, and sends $c_i = (x_i + \rho_i, r_i)$ to $S$, who stores it into $v_i \in \mathcal{G}$.
4. $\mathsf{Query}\langle U_q(K_q, \alpha), S(\mathcal{G})\rangle(f)$: User $U_q$ and $S$ run $\pi_{\mathsf{RE}}$, where $U_q$ has as input $K_q$ and $S$ has $\mathcal{G}$. Recall that $\mathcal{G}$ contains $c_j$ and $k_{j\rightarrow q}$ for every friend $U_j$ of $U_q$. The server receives as output $[\![x_j]\!]_{pk_q}$, where $x_j$ is the private input of a friend $U_j$. Subsequently, $S$ and $U_q$ execute $\pi_f$, where $S$ uses as input the ciphertexts $[\![x_j]\!]_{pk_q}$, along with $[\![\alpha]\!]_{pk_q}$ which is provided by the querier. At the end of this protocol, $U_q$ learns $y = f(\alpha, x_q, \{x_j \mid \forall j : U_j \in \mathcal{G}_q\})$.

# Re-Encryption Protocol

3. $\mathsf{Upload}\langle U_i(K_i, x_i), S(\mathcal{G})\rangle$: User $U_i$ picks random nonce $r_i$, computes $\rho_i = F_{K_i}(r_i)$, and sends $c_i = (x_i + \rho_i, r_i)$ to $S$, who stores it into $v_i \in \mathcal{G}$.

# Re-Encryption Protocol

$$\rho_i = F_{K_i}(r_i)$$

$$c_i = (x_i + \rho_i, r_i)$$

| $U_q(sk_q)$ | | $S(c_j, k_{j\to q})$ |
|---|---|---|
| | | 1. parse $c_j$ as $(x_j + \rho_j, r_j)$ |
| | | 2. pick random $\rho^*$ |
| | | 3. compute $(x_j + \rho_j) + \rho^* = c_j^*$ |
| | 4. send $c_j^*, r_j$ and $k_{j\to q} = [\![K_j]\!]$ ← | |
| 5. decrypt $[\![K_j]\!]$ | | |
| 6. compute $[\![c_j^* - F_{K_j}(r_j)]\!] = [\![x_j + \rho^*]\!]$ | | |
| | 7. send $[\![x_j + \rho^*]\!]$ → | |
| | | 8. compute $[\![x_j + \rho^*]\!] \cdot [\![\rho^*]\!]^{-1} = [\![x_j]\!]$ |

# Acknowledgement

- Some materials are extracted from the slides created by Prof. Qi WANG in the course Cryptography and Network Security