Secure Multiparty Computation with Free Branching

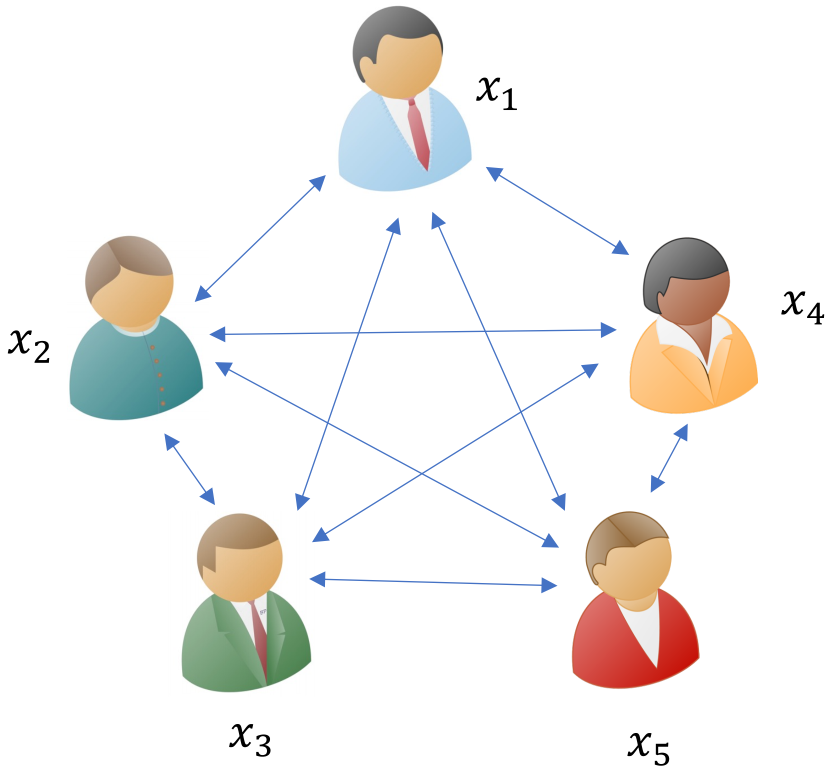
Aarushi Goel Mathias Hall-Andersen Aditya Hegde Abhishek Jain

JOHNS HOPKINS UNIVERSITY

AARHUS UNIVERSITY

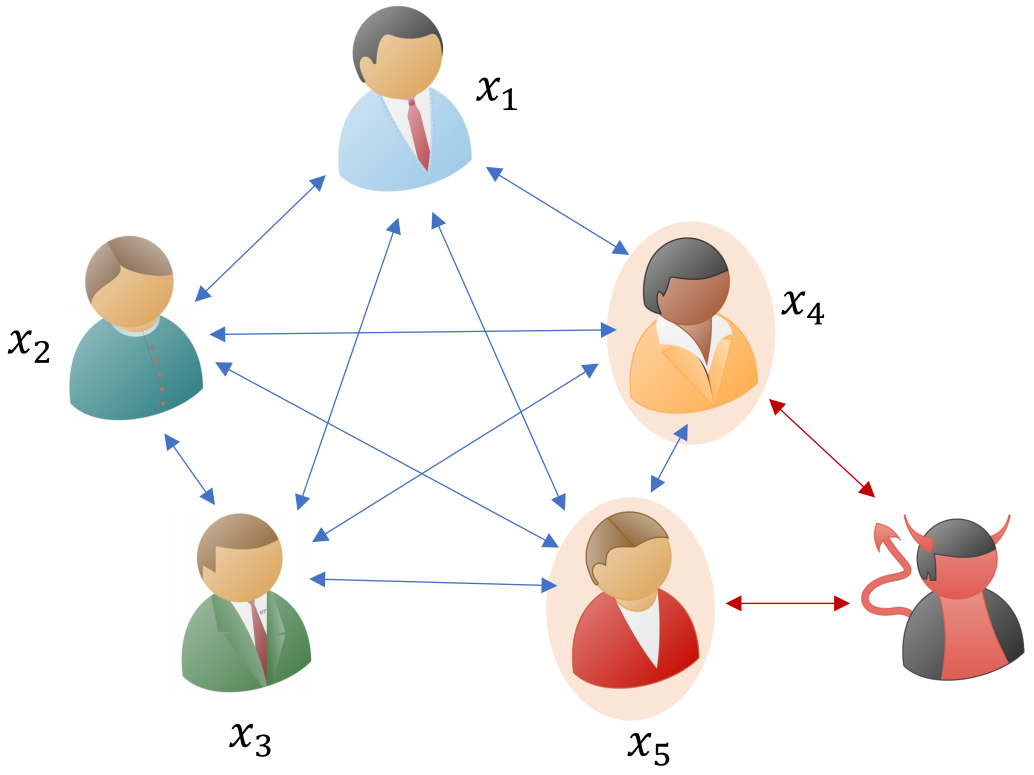
EUROCRYPT 2022

Secure Multiparty Computation (MPC)



MPC protocol for computing $y = f(x_1, x_2, x_3, x_4, x_5)$

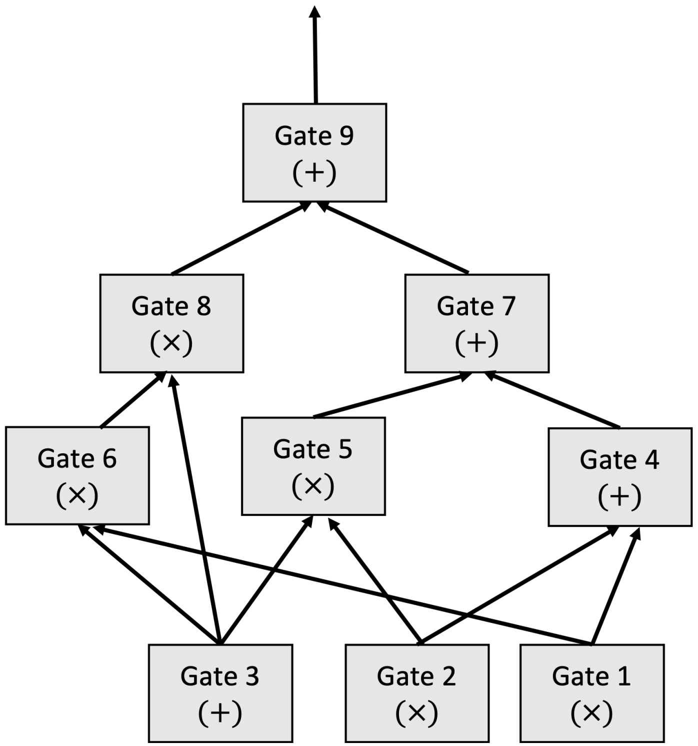
Secure Multiparty Computation (MPC)



Adversary learns nothing beyond the output y

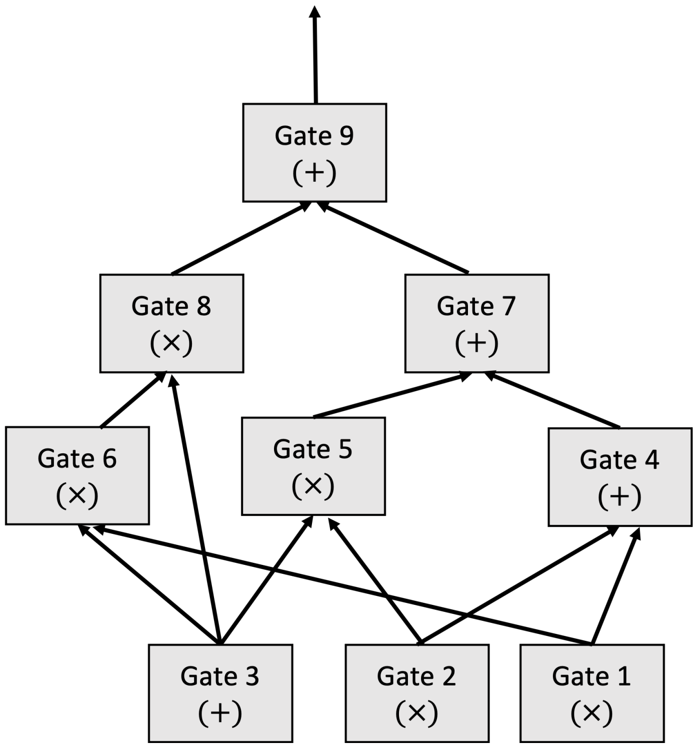
MPC protocol for computing $y = f(x_1, x_2, x_3, x_4, x_5)$

Limitation of Existing Efficient MPC Protocols



Existing protocols rely on circuit representation of functions

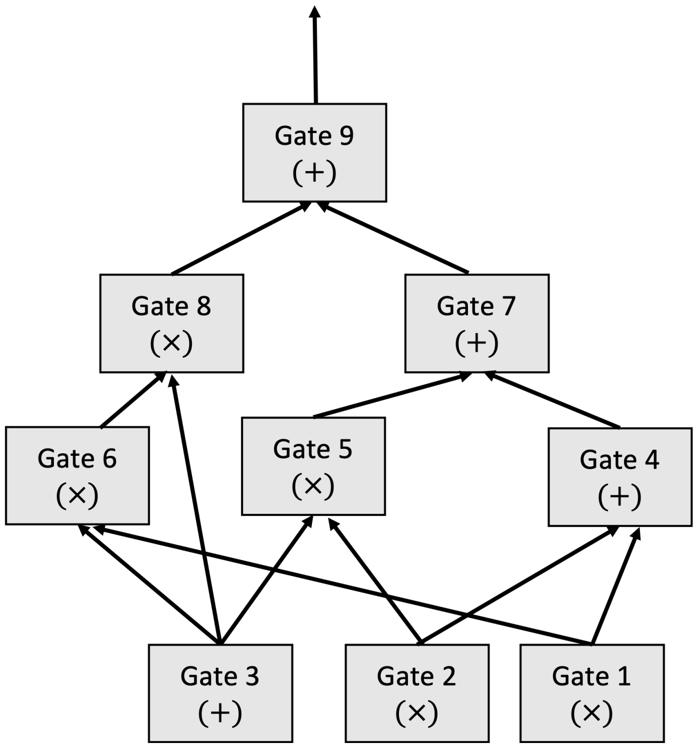
Limitation of Existing Efficient MPC Protocols



Existing protocols rely on circuit representation of functions

Communication complexity is linear in the size of circuit

Limitation of Existing Efficient MPC Protocols

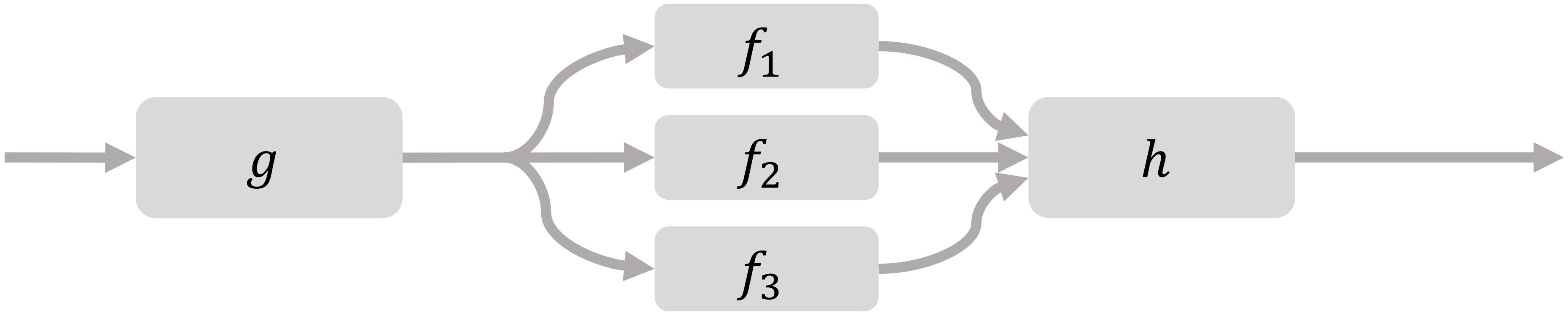


Existing protocols rely on circuit representation of functions

Communication complexity is linear in the size of circuit

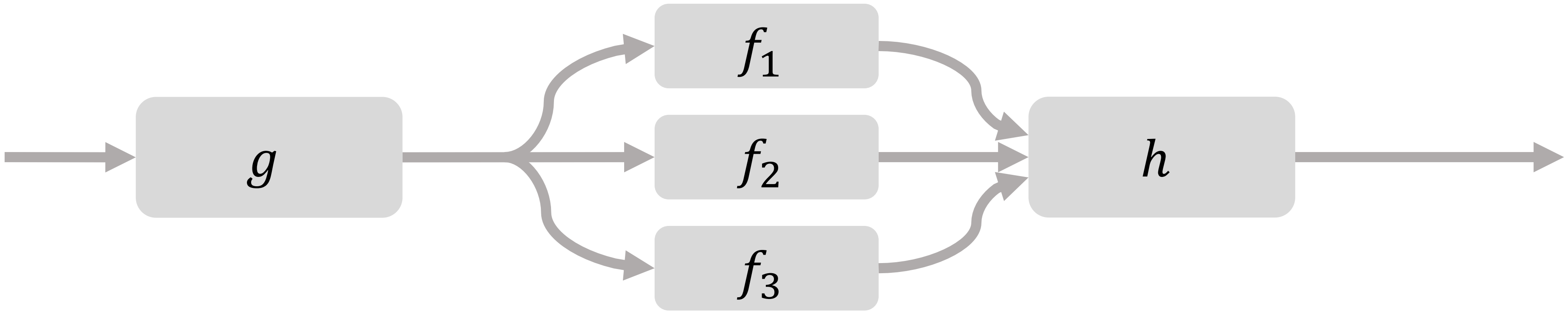
What about functions that don’t have an efficient circuit representation?

Example: Conditional Branches



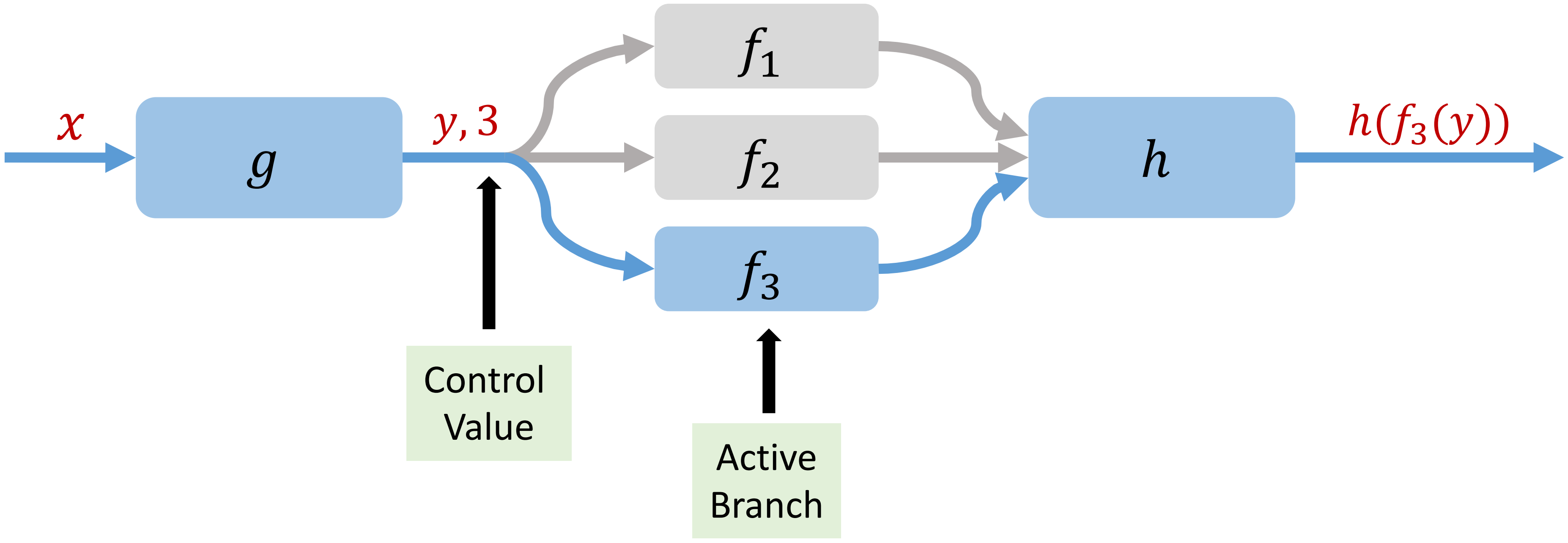
Example: Conditional Branches

Circuit Representation: depends on ALL 3 branches



Example: Conditional Branches

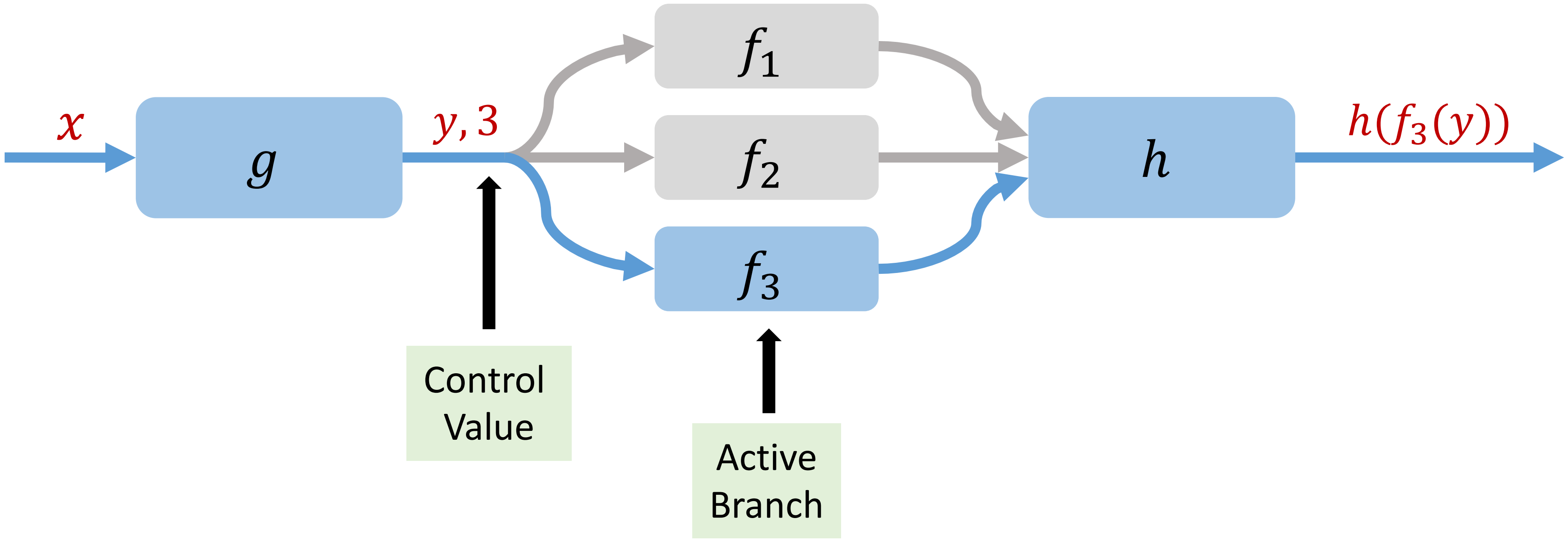
Circuit Representation: depends on ALL 3 branches



Example: Conditional Branches

Circuit Representation: depends on ALL 3 branches

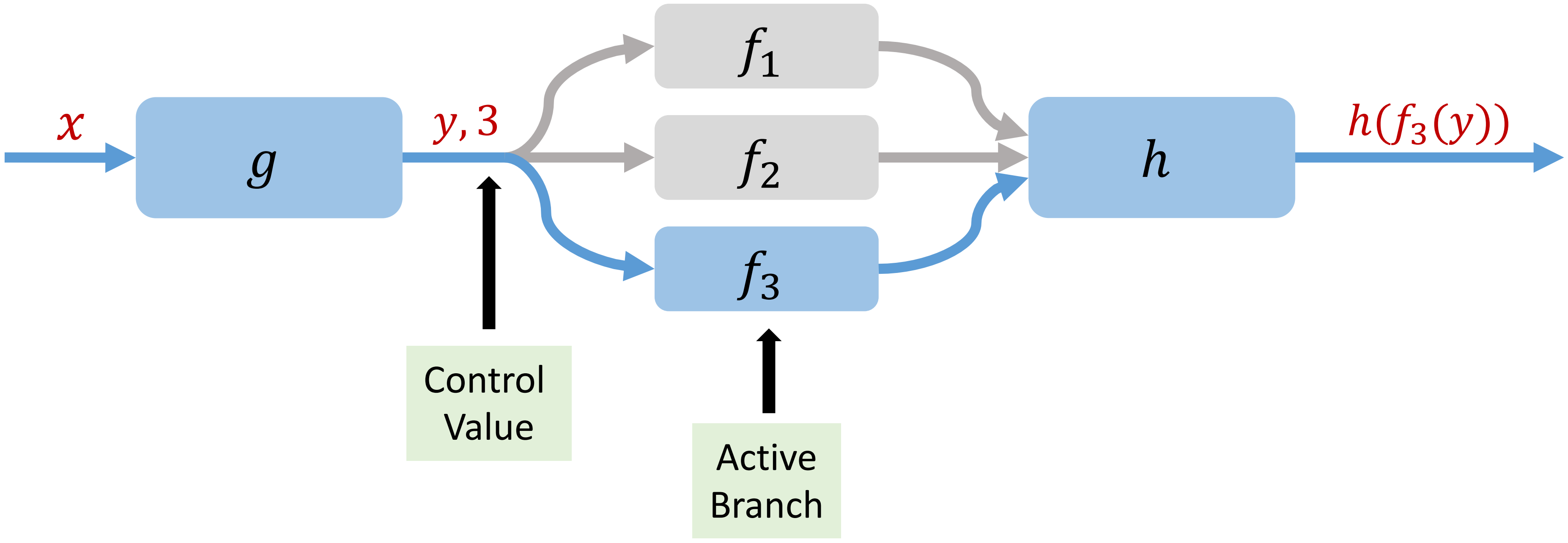
Circuit Evaluation: only depends on ONE branch



Example: Conditional Branches

Circuit Representation: depends on ALL 3 branches

Circuit Evaluation: only depends on ONE branch



Naïve use of existing MPC protocols will result in communication proportional to all 3 branches

Main Question

Circuit Representation: depends on ALL 3 branches

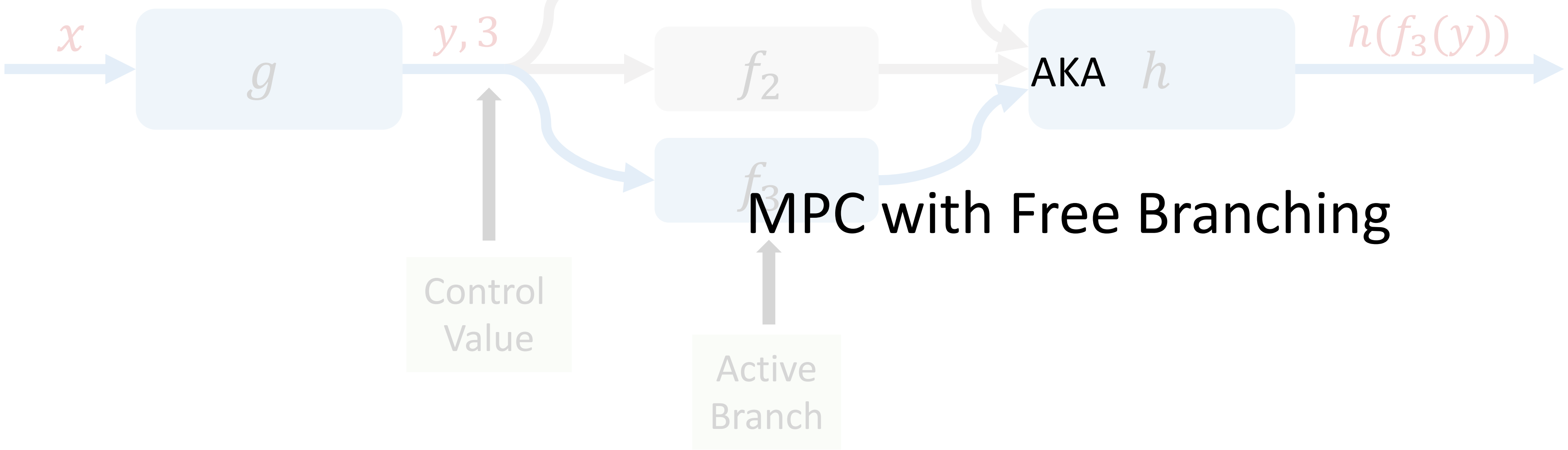
Can we design efficient MPC protocols for computing conditional branches, where communication only depends on the size of a single branch?

Naïve use of existing MPC protocols will result in communication proportional to all 3 branches

Main Question

Circuit Representation: depends on ALL 3 branches

Can we design efficient MPC protocols for computing conditional branches, where communication only depends on the size of a single branch?



AKA MPC with Free Branching

Naïve use of existing MPC protocols will result in communication proportional to all 3 branches

MPC with Free Branching: Applications

Control flow instructions in computer programs

Collection of servers providing services that clients can pay and for and obliviously avail

Prior Work

k = # branches

$|C| =$ size of largest branch

Result	No. of Parties	Communication	Security	Rounds	Type of Circuits
[HK20]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HK21]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HKP20,HKP21]	n	$O(kn^2\|C\|)$	Semi-Honest	Linear in depth	Boolean

Prior Work

k = # branches

$|C| =$ size of largest branch

Result	No. of Parties	Communication	Security	Rounds	Type of Circuits
[HK20]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HK21]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HKP20,HKP21]	n	$O(kn^2\|C\|)$	Semi-Honest	Linear in depth	Boolean

No n-party protocol where communication only depends on the size of one branch

Prior Work

k = # branches

$|C| =$ size of largest branch

Result	No. of Parties	Communication	Security	Rounds	Type of Circuits
[HK20]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HK21]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HKP20,HKP21]	n	$O(kn^2\|C\|)$	Semi-Honest	Linear in depth	Boolean

No maliciously
secure protocol

Prior Work

k = # branches

$|C| =$ size of largest branch

Result	No. of Parties	Communication	Security	Rounds	Type of Circuits
[HK20]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HK21]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HKP20,HKP21]	n	$O(kn^2\|C\|)$	Semi-Honest	Linear in depth	Boolean

No protocol for arithmetic circuits

Our Results

k = # branches

$|C| =$ size of largest branch

Result	No. of Parties	Communication	Security	Rounds	Type of Circuits
[HK20]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HK21]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HKP20,HKP21]	n	$O(kn^2\|C\|)$	Semi-Honest	Linear in depth	Boolean
Our Work	n	$O(n^2\|C\|)$	Semi-Honest	Linear in depth	Arithmetic

Our Results

k = # branches

$|C| =$ size of largest branch

Result	No. of Parties	Communication	Security	Rounds	Type of Circuits
[HK20]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HK21]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HKP20,HKP21]	n	$O(kn^2\|C\|)$	Semi-Honest	Linear in depth	Boolean
Our Work	n	$O(n^2\|C\|)$	Semi-Honest	Linear in depth	Arithmetic
Our Work	n	$O(n^2 s\|C\|)$	Malicious	Linear in depth	Arithmetic

Statistical security parameter

Our Results

k = # branches

$|C| =$ size of largest branch

Result	No. of Parties	Communication	Security	Rounds	Type of Circuits
[HK20]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HK21]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HKP20,HKP21]	n	$O(kn^2\|C\|)$	Semi-Honest	Linear in depth	Boolean
Our Work	n	$O(n^2\|C\|)$	Semi-Honest	Linear in depth	Arithmetic
Our Work	n	$O(n^2 s\|C\|)$	Malicious	Linear in depth	Arithmetic
Our Work	n	$O(n^2\|C\|)$	Semi-Honest	Constant	Boolean

Our Results

k = # branches

$|C| =$ size of largest branch

Result	No. of Parties	Communication	Security	Rounds	Type of Circuits
[HK20]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HK21]	2	$O(\|C\|)$	Semi-Honest	Non-interactive	Boolean
[HKP20,HKP21]	n	$O(kn^2\|C\|)$	Semi-Honest	Linear in depth	Boolean
Our Work	n	$O(n^2\|C\|)$	Semi-Honest	Linear in depth	Arithmetic
Our Work	n	$O(n^2 s\|C\|)$	Malicious	Linear in depth	Arithmetic
Our Work	n	$O(n^2\|C\|)$	Semi-Honest	Constant	Boolean

Main Ideas

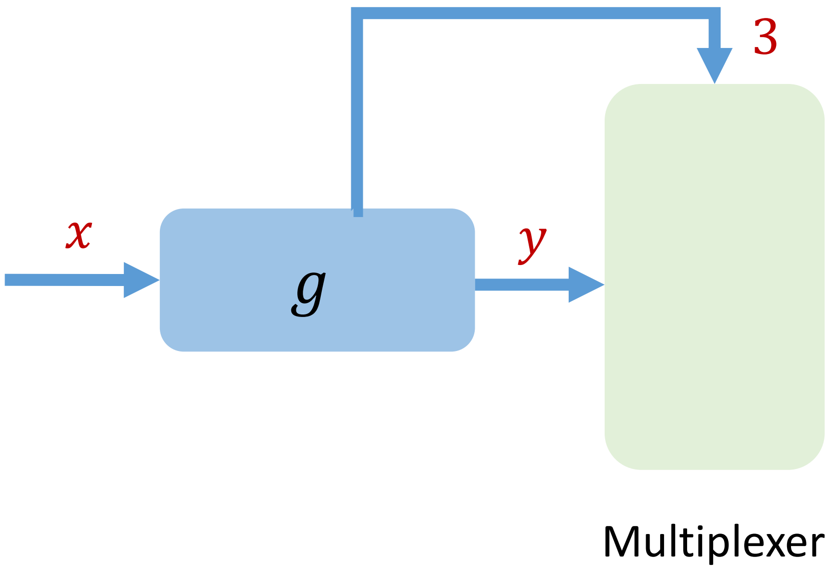
Prior Work: High-Level Approach

Evaluate all branches and then filter the correct output

Prior Work: High-Level Approach

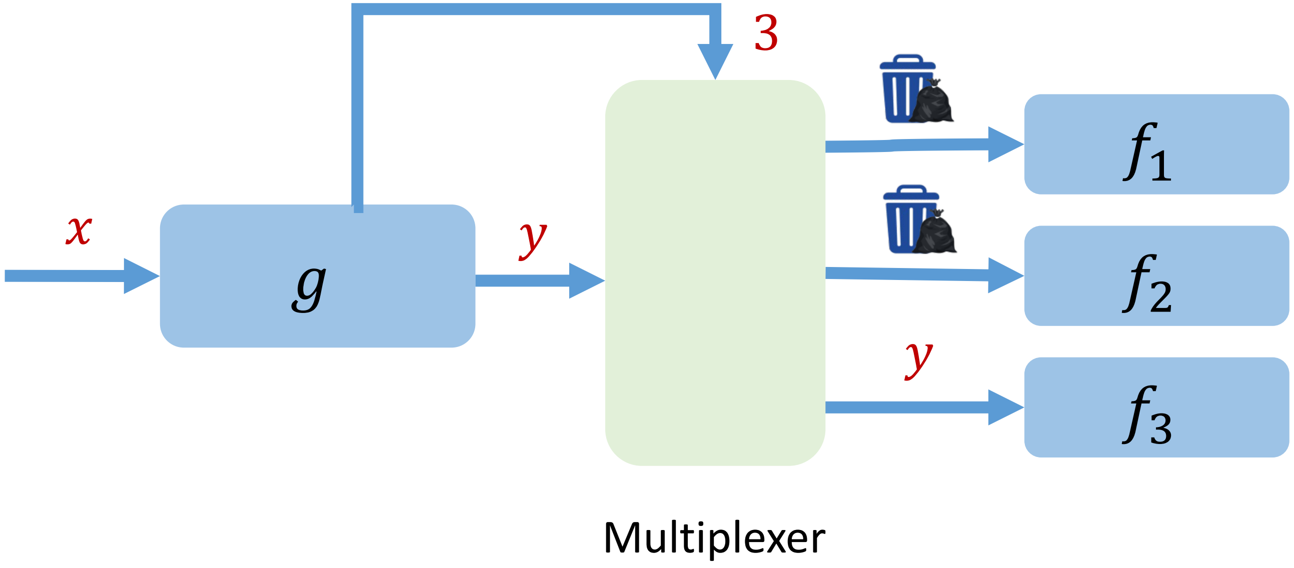
Evaluate all branches and then filter the correct output

Prior Work: High-Level Approach



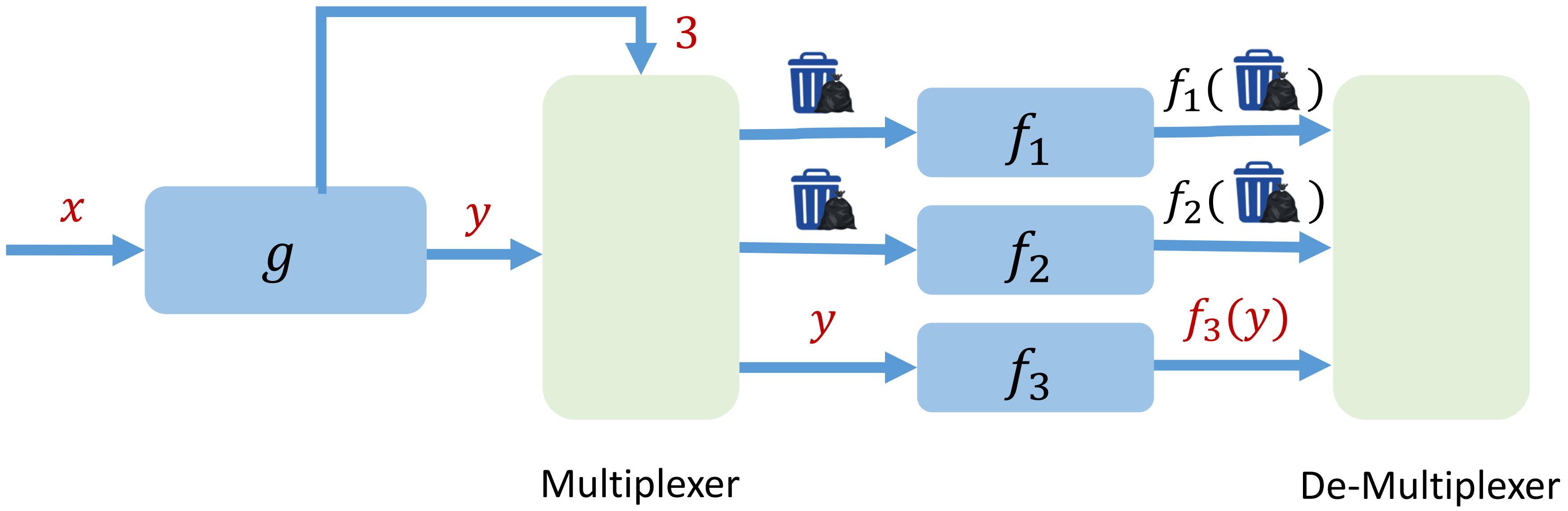
Evaluate all branches and then filter the correct output

Prior Work: High-Level Approach



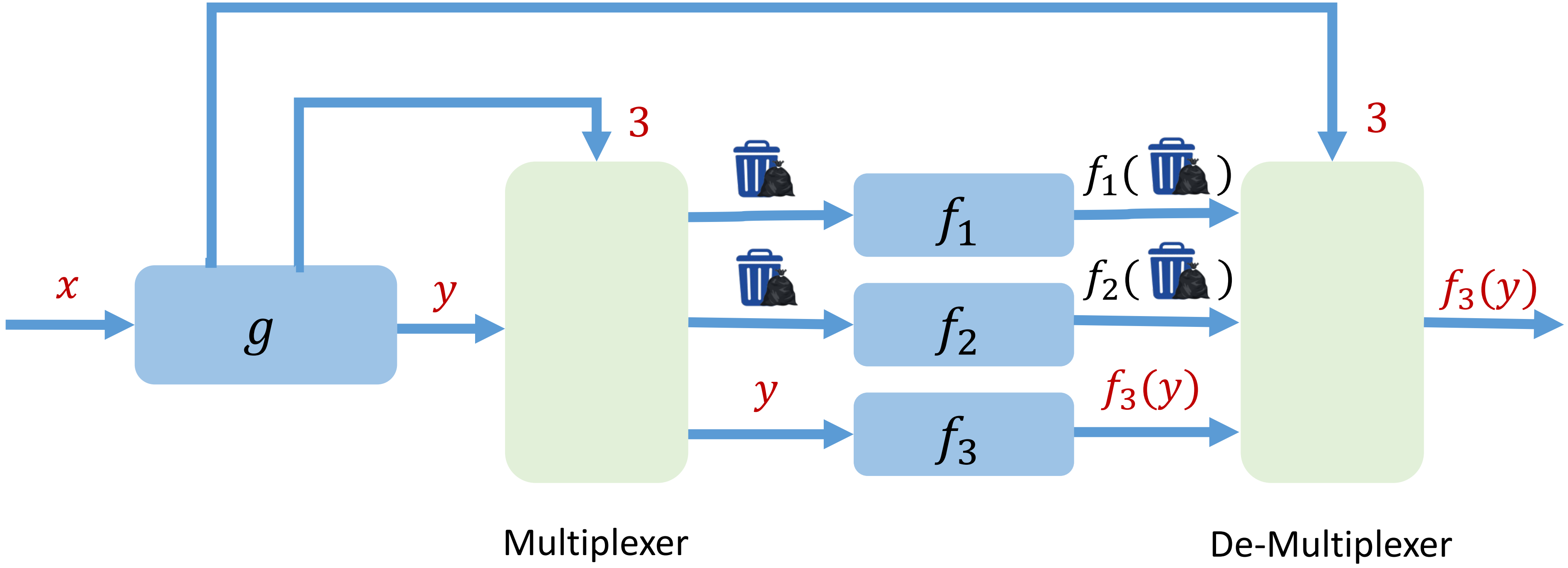
Evaluate all branches and then filter the correct output

Prior Work: High-Level Approach



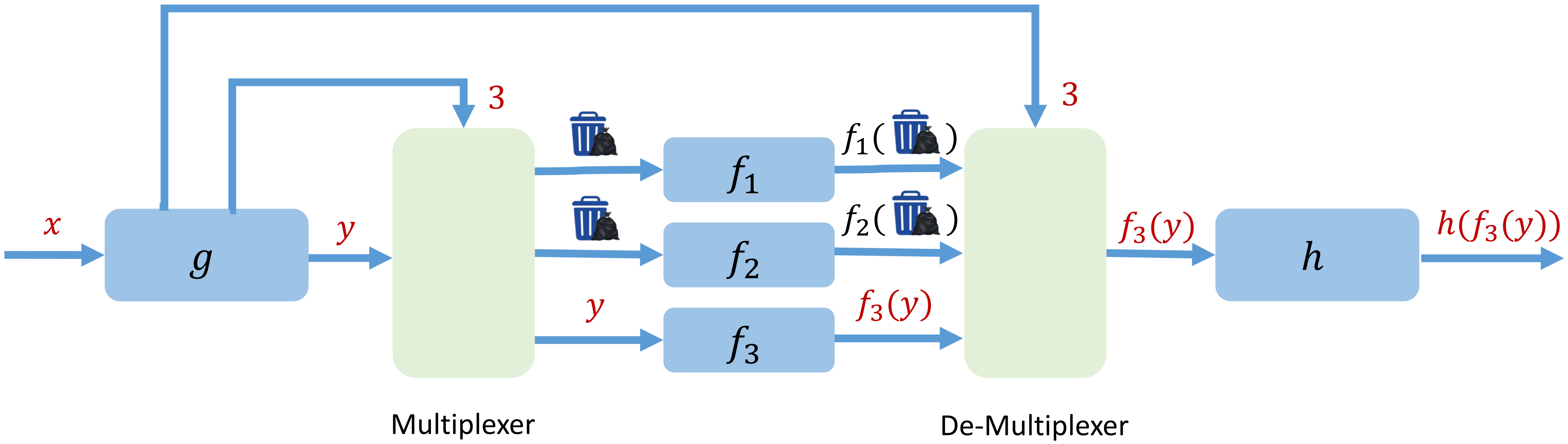
Evaluate all branches and then filter the correct output

Prior Work: High-Level Approach



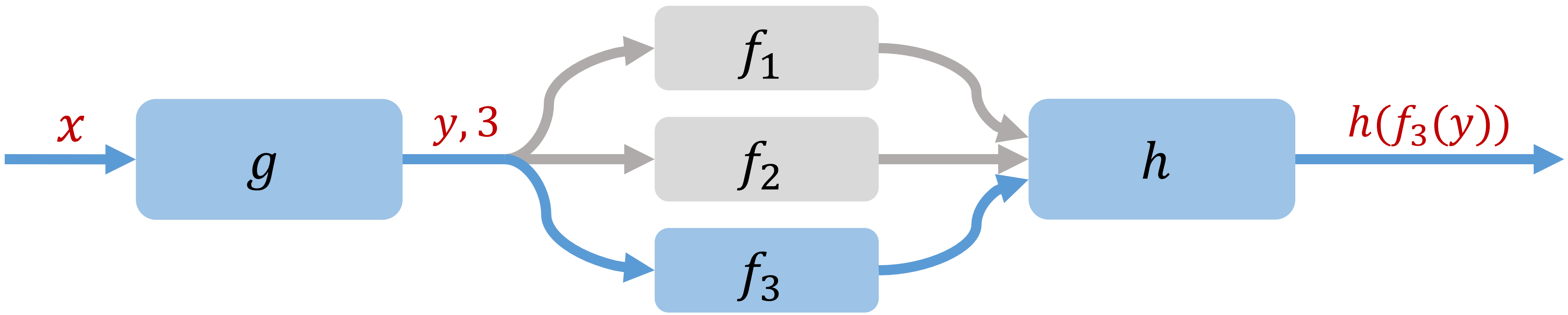
Evaluate all branches and then filter the correct output

Prior Work: High-Level Approach



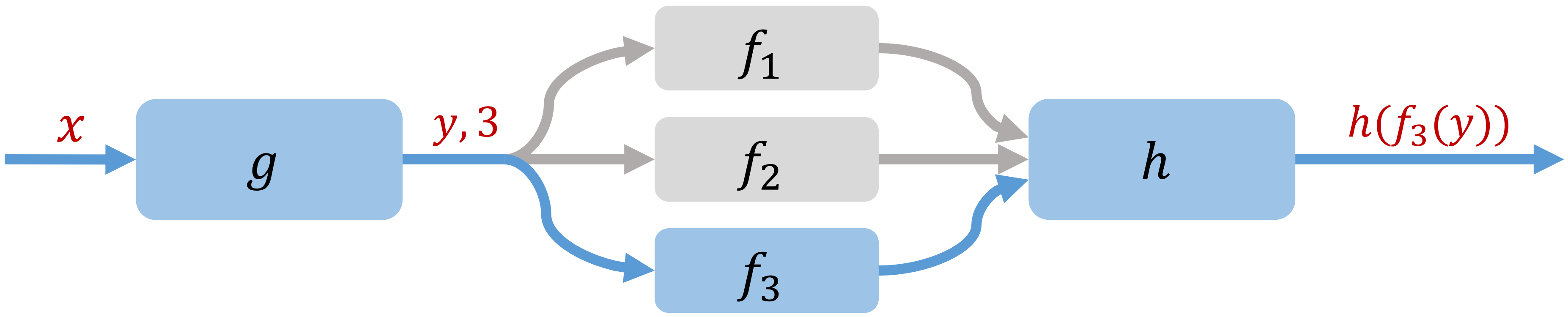
Evaluate all branches and then filter the correct output

Our Work: High-Level Approach



Obliviously select the active branch and only evaluate that on correct inputs

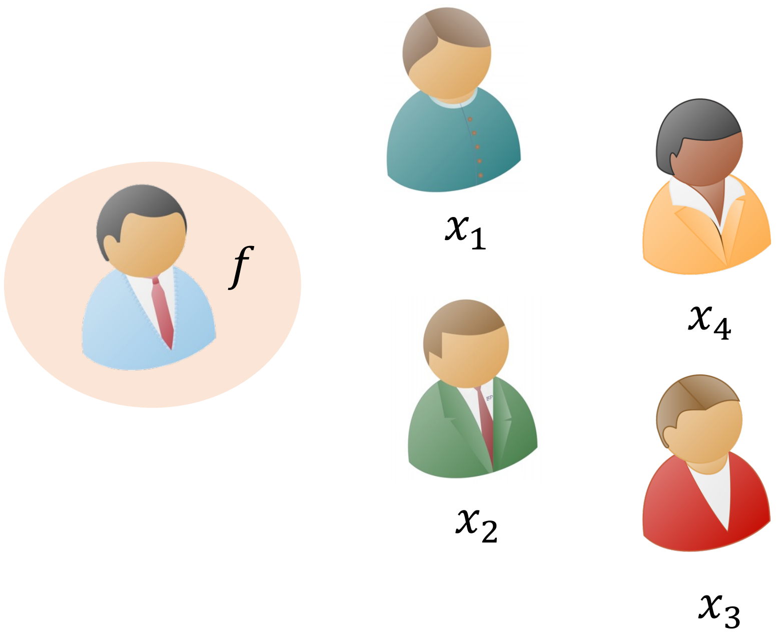
Our Work: High-Level Approach



Obliviously select the active branch and only evaluate that on correct inputs

Since the active branch must remain hidden, how does one compute on a hidden function?

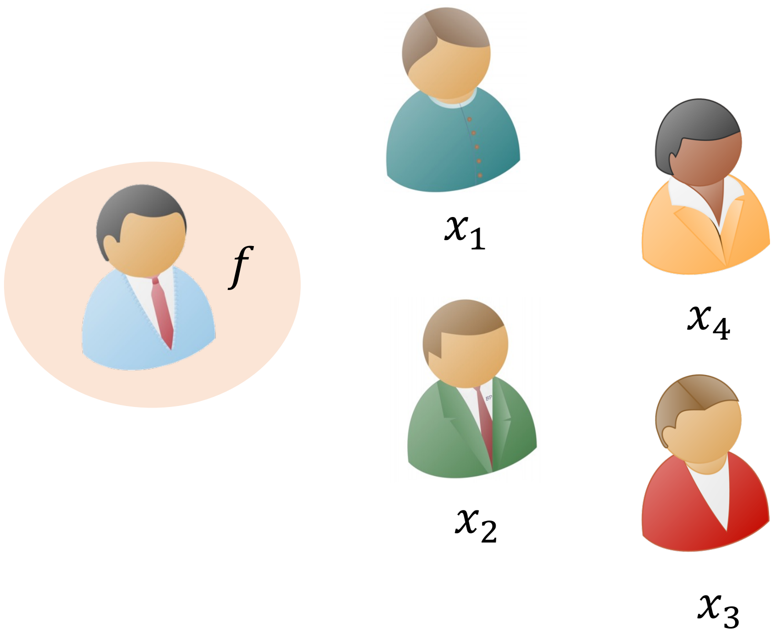
Our Initial Observation: Similarities with PFE



Private Function Evaluation

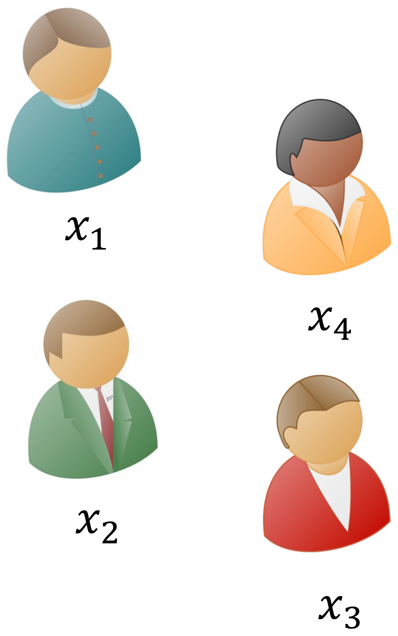
only 1 person knows the function!

Our Initial Observation: Similarities with PFE



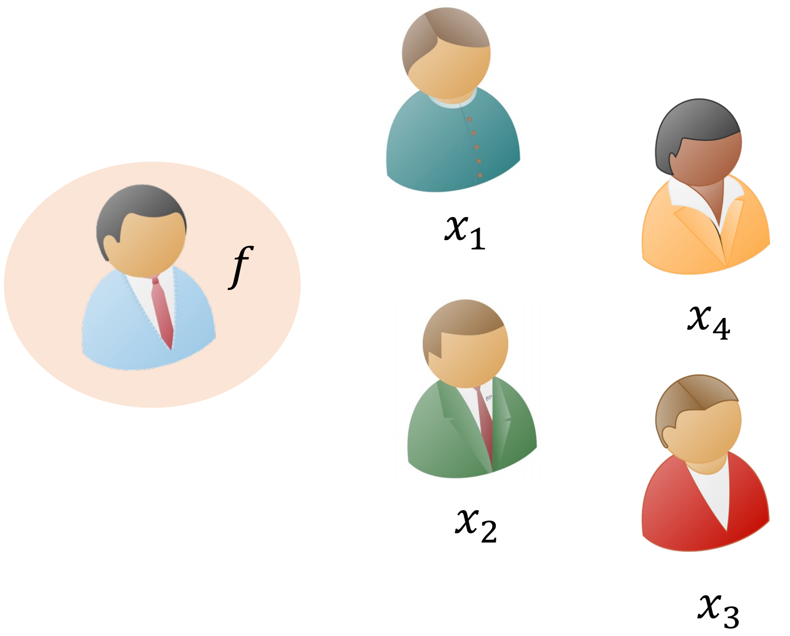
Private Function Evaluation
only 1 person knows the function!

$$f \in \{f_1, f_2, f_3\}$$



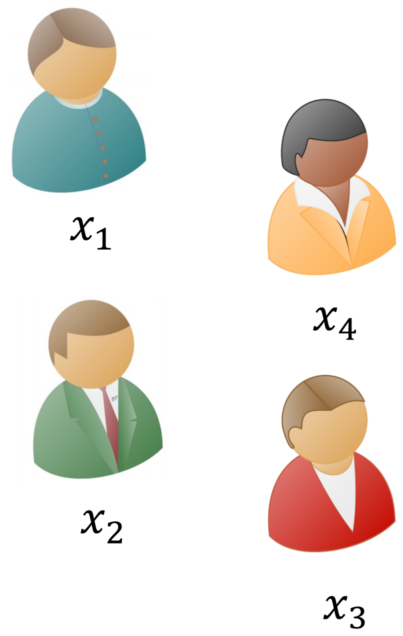
MPC for Conditional Branches
No one knows the function!

Our Initial Observation: Similarities with PFE



Private Function Evaluation
only 1 person knows the function!

$$f \in \{f_1, f_2, f_3\}$$



MPC for Conditional Branches
No one knows the function!

Parties collectively hold information about the active branch

Talk Outline

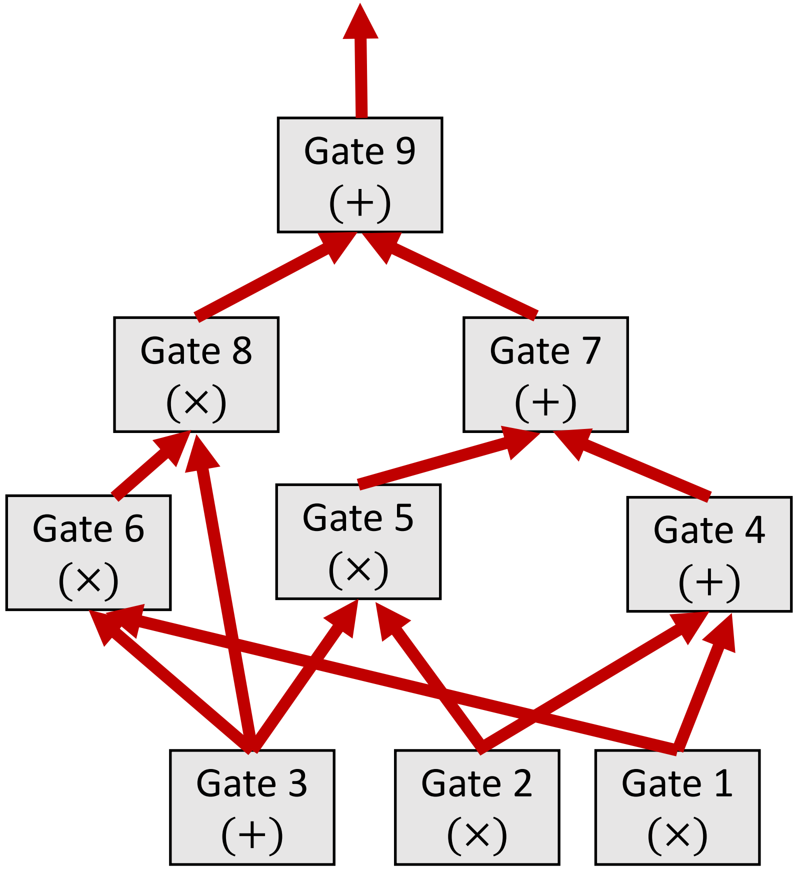
- Overview of the [MS13] PFE protocol
- Our semi-honest non-constant round protocol for conditional branches
- Performance Evaluation
- Remarks on additional results

Talk Outline

- Overview of the [MS13] PFE protocol
- Our semi-honest non-constant round protocol for conditional branches
- Performance Evaluation
- Remarks on additional results

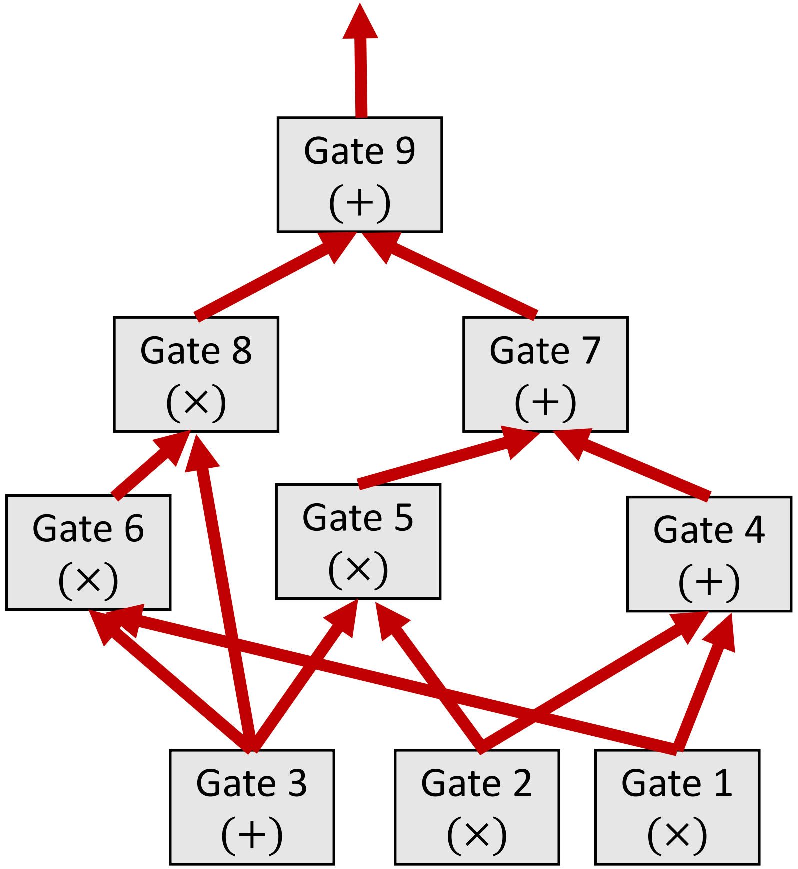
PFE: How to Hide Circuit Topology? [MS13]

PFE: How to Hide Circuit Topology? [MS13]

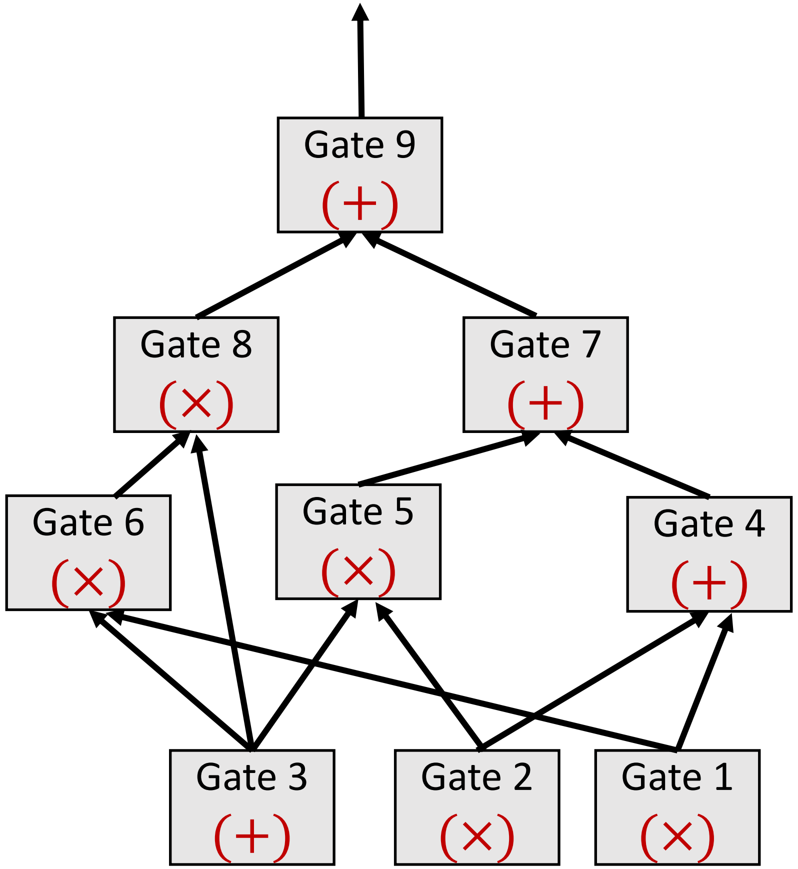


Hide wire configurations

PFE: How to Hide Circuit Topology? [MS13]



Hide wire configurations



Hide gate functions

PFE: Hiding Gate Functions [MS13]

PFE: Hiding Gate Functions [MS13]

f

For every gate g:

x_1 x_2 x_3 x_4

- Let $type_g = 0$: if g is a multiplication gate
- Let $type_g = 1$: if g is an addition gate

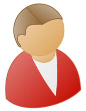
PFE: Hiding Gate Functions [MS13]

f

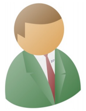
For every gate g:

x_1

x_2



x_3

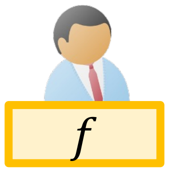


x_4

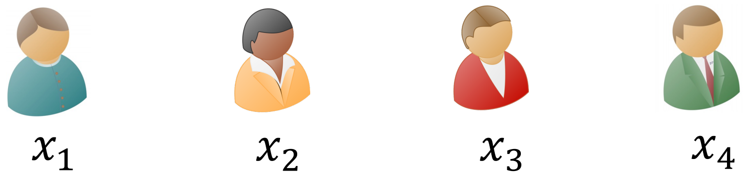
- Let $type_g = 0$: if g is a multiplication gate
- Let $type_g = 1$: if g is an addition gate

Secret share $type_g$

PFE: Hiding Gate Functions [MS13]



For every gate g:

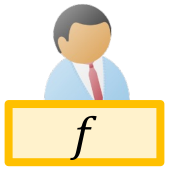


- Let $type_g = 0$: if g is a multiplication gate
- Let $type_g = 1$: if g is an addition gate

Secret share $type_g$

Given shares $[L_g], [R_g]$ of left and right input wires, compute

PFE: Hiding Gate Functions [MS13]



For every gate g:

- Let $type_g = 0$: if g is a multiplication gate
- Let $type_g = 1$: if g is an addition gate

Secret share $type_g$

Compute using an MPC that can operate over shares!

Given shares $[L_g], [R_g]$ of left and right input wires, compute

$$[type_g].([L_g].[R_g]) + (1 - [type_g]).([L_g] + [R_g])$$

PFE: Hiding Gate Functions [MS13]

f

For every gate g:

x_1 x_2 x_3 x_4

- Let $type_g = 0$: if g is a multiplication gate
- Let $type_g = 1$: if g is an addition gate

Secret share $type_g$

How to compute this without knowing wire configurations?

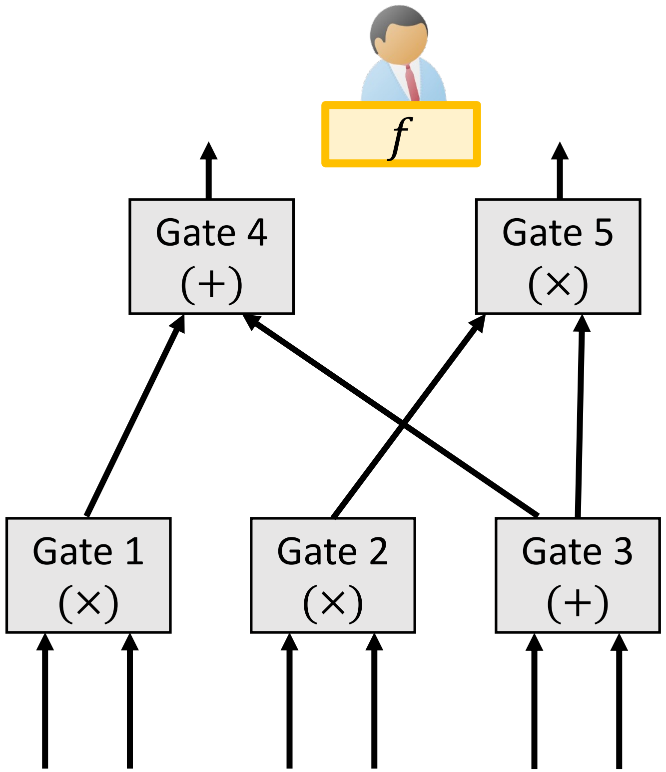
Given shares $[L_g], [R_g]$ of left and right input wires, compute

$$[type_g].([L_g].[R_g]) + (1 - [type_g]).([L_g] + [R_g])$$

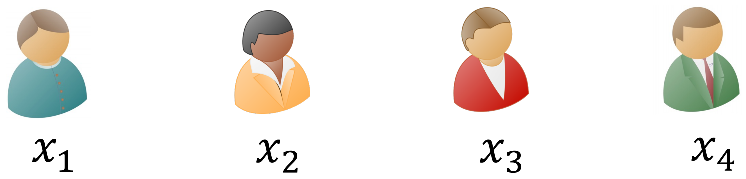
Compute using an MPC that can operate over shares!

PFE: Hiding Wire Configuration [MS13]

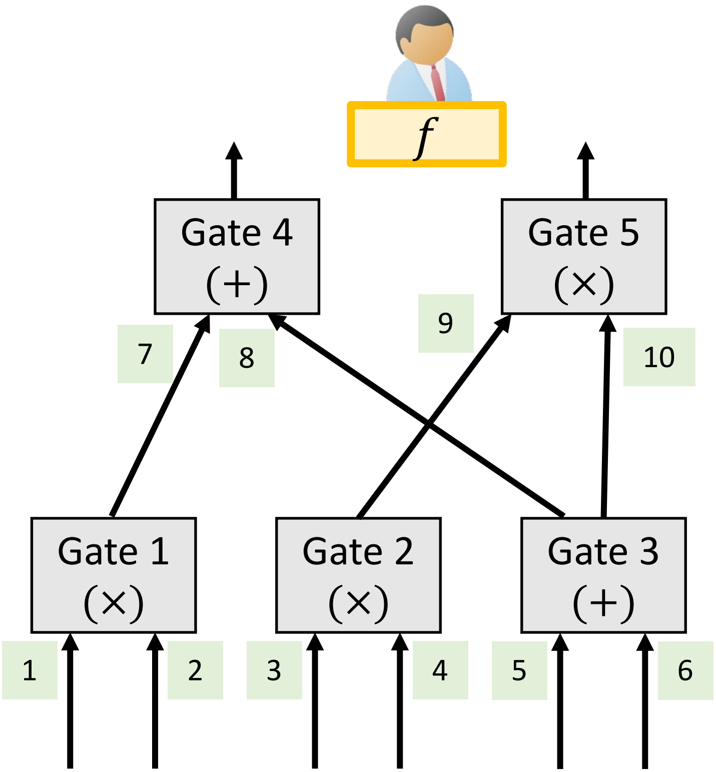
PFE: Hiding Wire Configuration [MS13]



Pre-Processing Phase



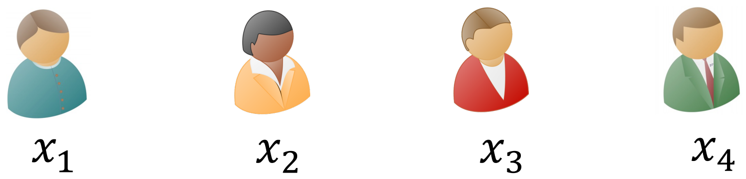
PFE: Hiding Wire Configuration [MS13]



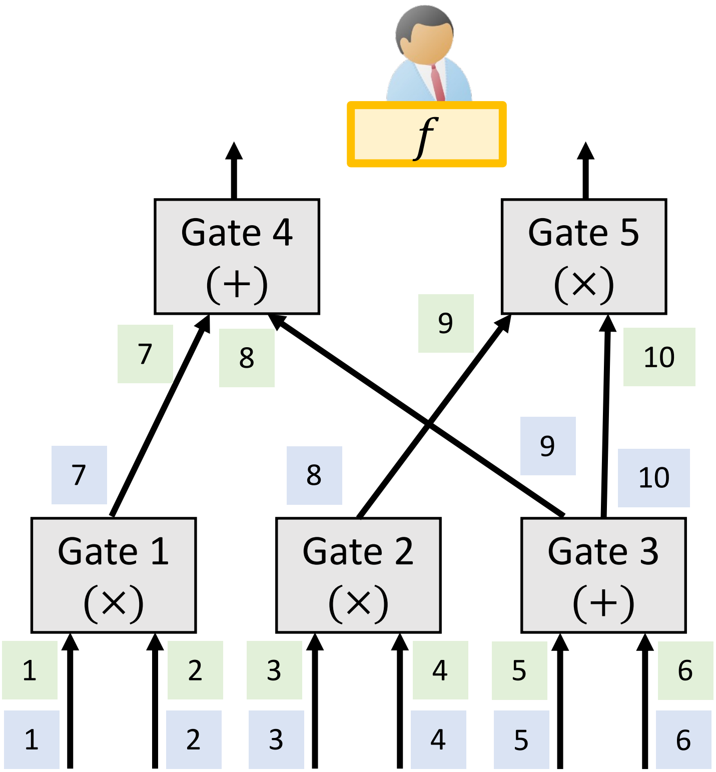
Pre-Processing Phase

For every wire w:

1 Assign an incoming label



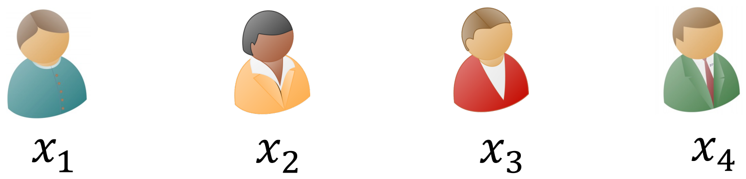
PFE: Hiding Wire Configuration [MS13]



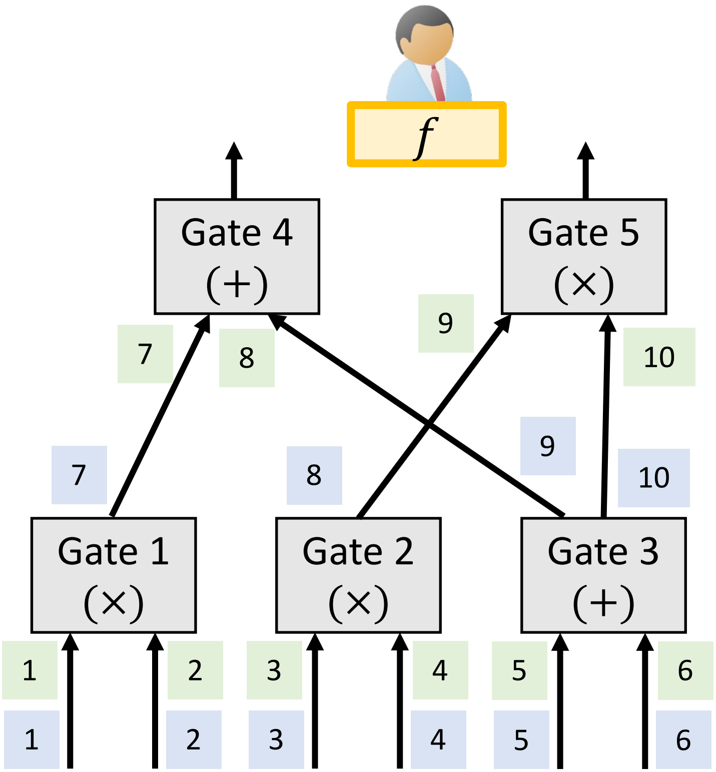
Pre-Processing Phase

For every wire w:

1 Assign an incoming label and an outgoing label



PFE: Hiding Wire Configuration [MS13]



Pre-Processing Phase

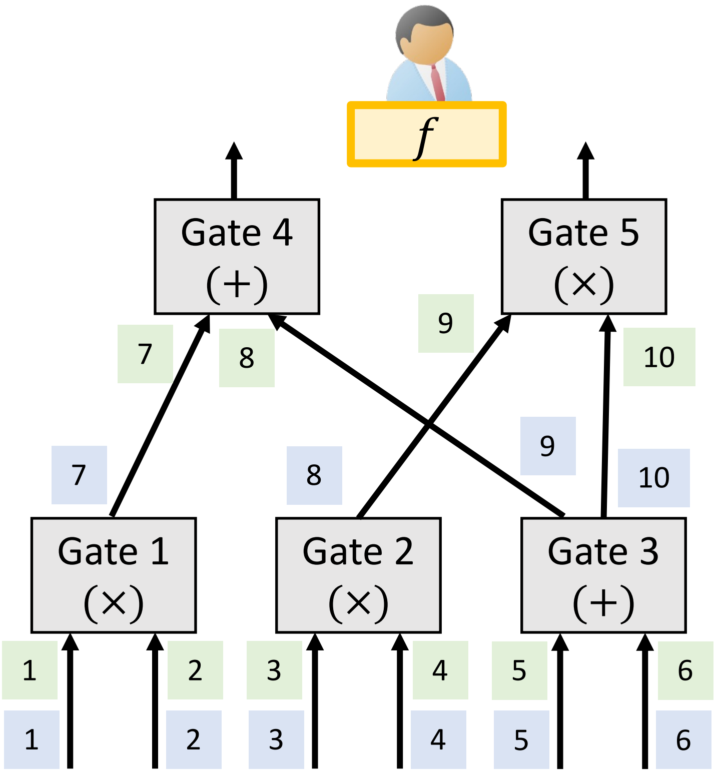
For every wire w:

1. Assign an incoming label and an outgoing label

2. Compute a mapping: $\pi: in \rightarrow out$

x_1 x_2 x_3 x_4

PFE: Hiding Wire Configuration [MS13]

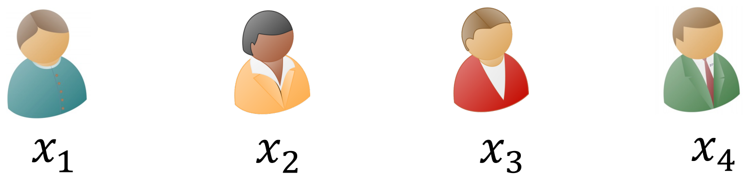


Pre-Processing Phase

For every wire w:

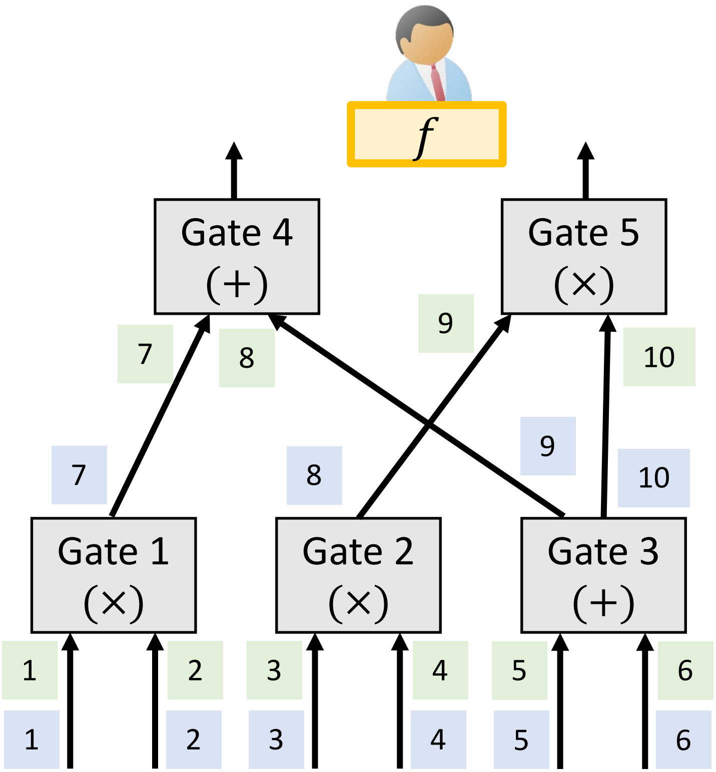
1 Assign an incoming label and an outgoing label

2 Compute a mapping: $\pi: in \rightarrow out$



3 Share random masks: $[imask_w], [omask_w]$

PFE: Hiding Wire Configuration [MS13]

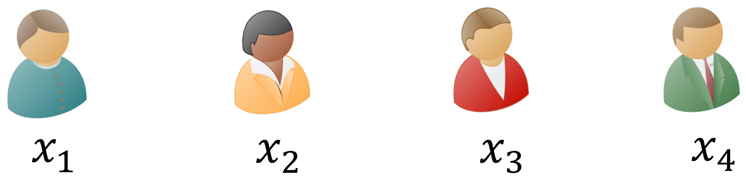


Pre-Processing Phase

For every wire w:

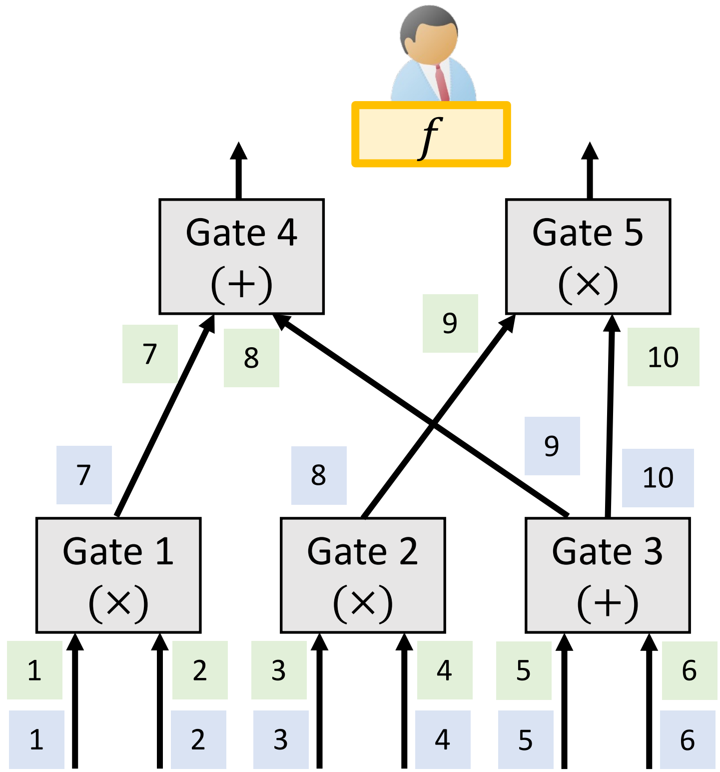
1. Assign an incoming label and an outgoing label

2. Compute a mapping: $\pi: in \rightarrow out$



3. Share random masks: $[imask_w], [omask_w]$

PFE: Hiding Wire Configuration [MS13]

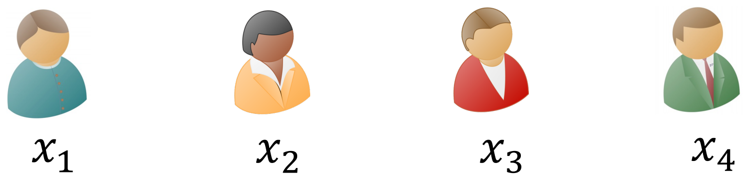


Pre-Processing Phase

For every wire w:

1. Assign an incoming label and an outgoing label

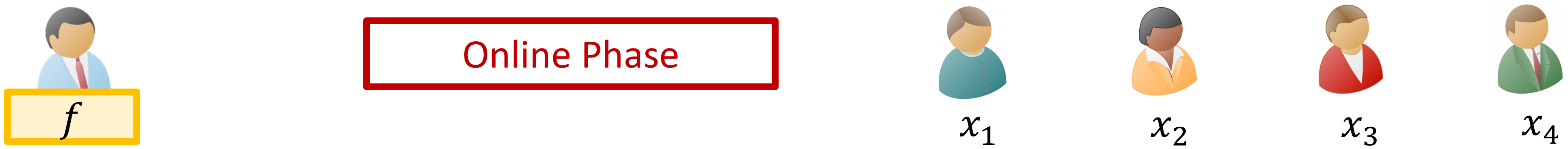
2. Compute a mapping: $\pi: in \rightarrow out$



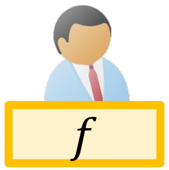
3. Share random masks: $[imask_w], [omask_w]$

4. function holder learns: $\Delta_w = imask_w - omask_{\pi(w)}$

PFE: Hiding Wire Configuration [MS13]

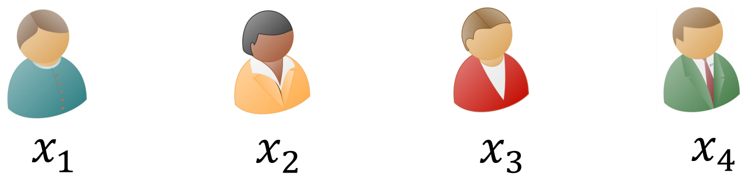


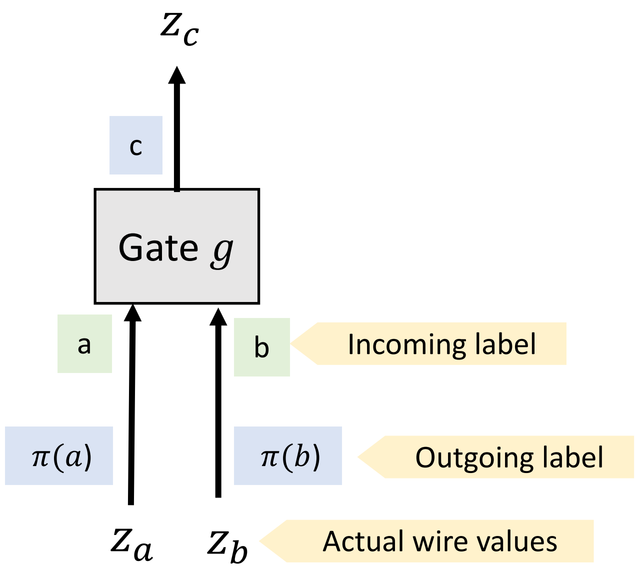
PFE: Hiding Wire Configuration [MS13]



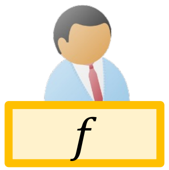
Online Phase

For every gate g:



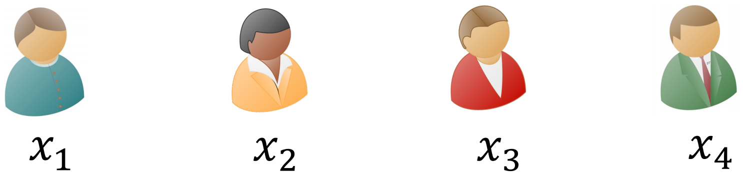


PFE: Hiding Wire Configuration [MS13]

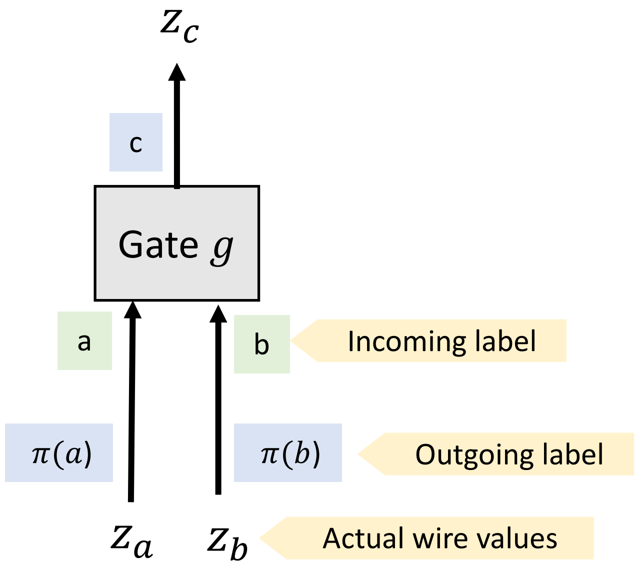


Online Phase

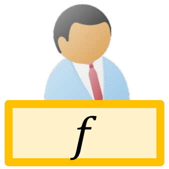
For every gate g:



After computing g

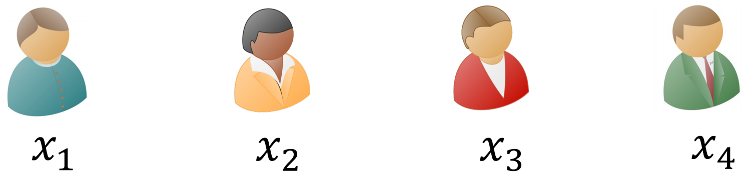


PFE: Hiding Wire Configuration [MS13]



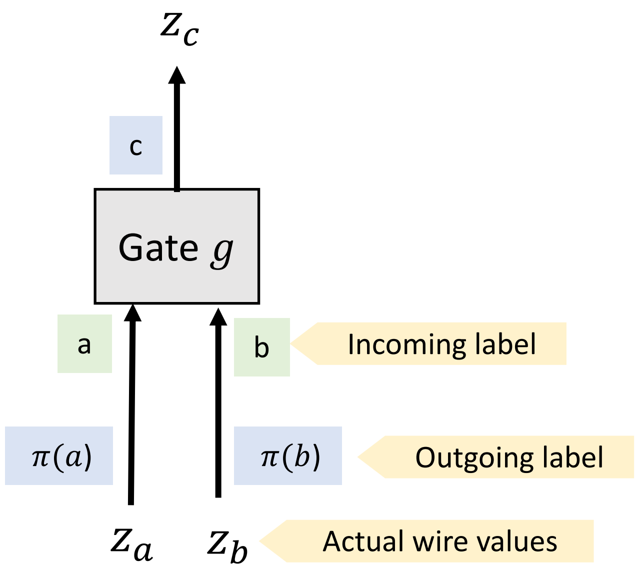
Online Phase

For every gate g:



After computing g

Compute $[u_c] = [z_c] + [omask_c]$
and Reconstruct u_c



PFE: Hiding Wire Configuration [MS13]

f

Online Phase

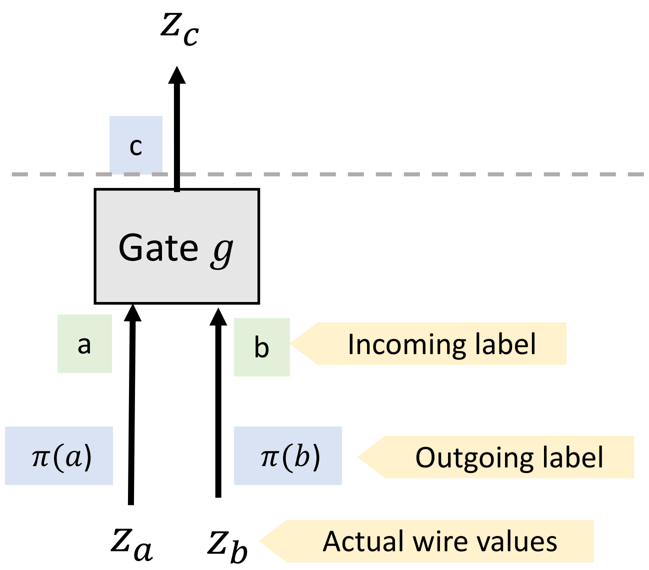
For every gate g:

x_1 x_2 x_3 x_4

After computing g

Compute $[u_c] = [z_c] + [omask_c]$
and Reconstruct u_c

For computing g



PFE: Hiding Wire Configuration [MS13]

f

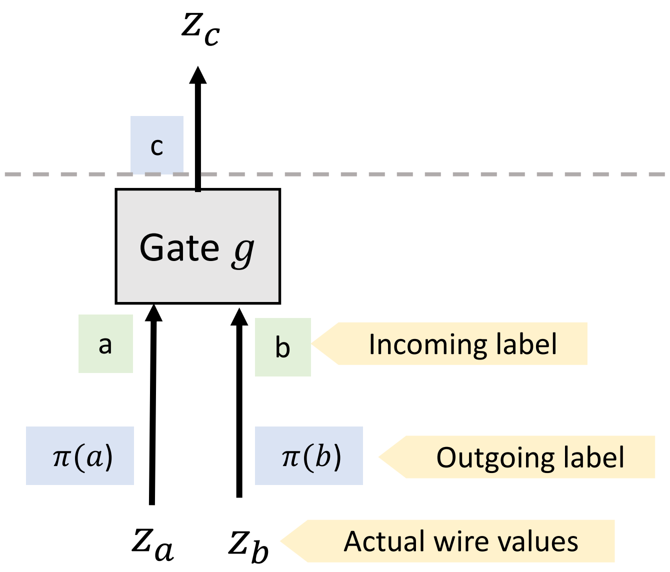
Online Phase

For every gate g:

x_1 x_2 x_3 x_4

After computing g

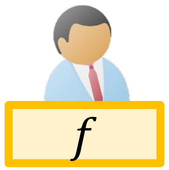
Compute $[u_c] = [z_c] + [omask_c]$
and Reconstruct u_c



For computing g

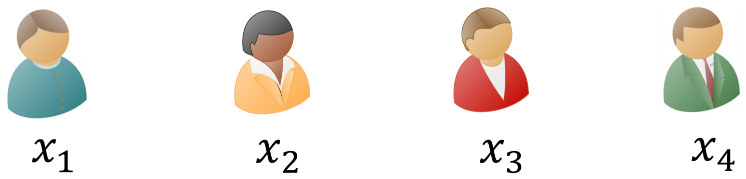
Compute and send
$A = u_{\pi(a)} + \Delta_a$
$B = u_{\pi(b)} + \Delta_b$

PFE: Hiding Wire Configuration [MS13]



Online Phase

For every gate g:

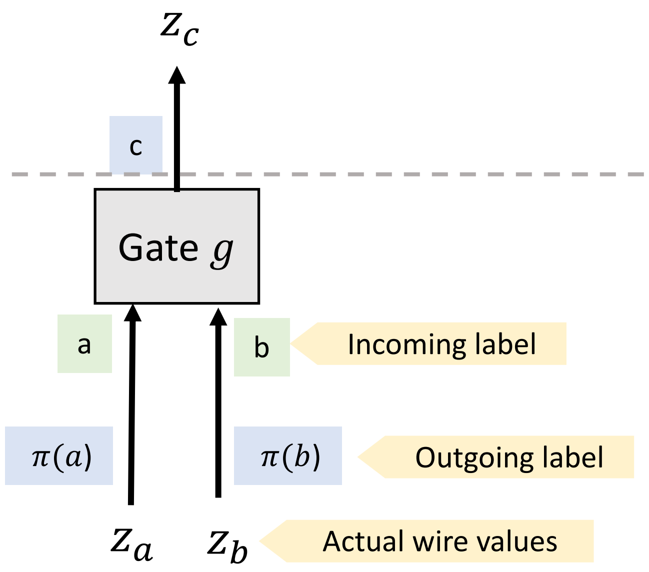


After computing g

Compute $[u_c] = [z_c] + [omask_c]$
and Reconstruct u_c

For computing g

Compute and send
$A = u_{\pi(a)} + \Delta_a$
$B = u_{\pi(b)} + \Delta_b$

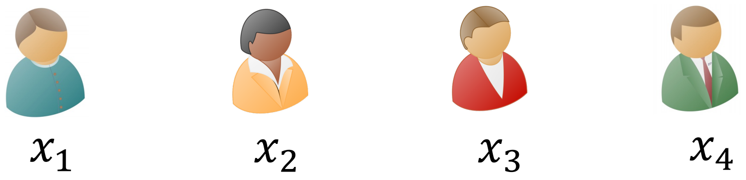


Compute
$[z_a] = A - [imask_a]$
$[z_b] = B - [imask_b]$

Talk Outline

- Overview of the [MS13] PFE protocol
- Our semi-honest non-constant round protocol for conditional branches
- Performance Evaluation
- Remarks on additional results

Conditional Branches: Hiding Gate Functions



Let's assume parties have secret sharing of unary representation of α, i.e., $[b_1], \ldots, [b_k]$

Index of the active branch

Conditional Branches: Hiding Gate Functions

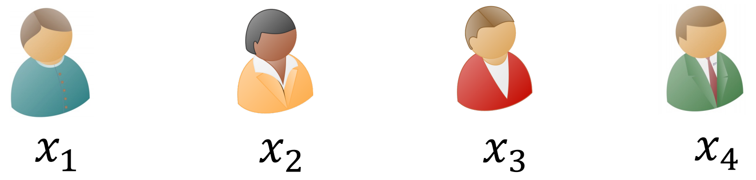
x_1 x_2 x_3 x_4

Let's assume parties have secret sharing of unary representation of α, i.e., $[b_1], \ldots, [b_k]$

Type of gate g in Branch 1

For every gate g: $[type_g] = [b_1].\, type_{1,g} + \cdots + [b_k].\, type_{k,g}$

Conditional Branches: Hiding Gate Functions



Let's assume parties have secret sharing of unary representation of α, i.e., $[b_1], \ldots, [b_k]$

For every gate g:

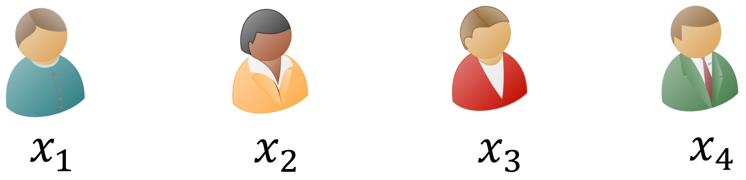
$$[type_g] = [b_1].\, type_{1,g} + \cdots + [b_k].\, type_{k,g}$$

Given shares $[L_g], [R_g]$ of left and right input wires, compute

$$[type_g].\,([L_g].\,[R_g]) + (1 - [type_g]).\,([L_g] + [R_g])$$

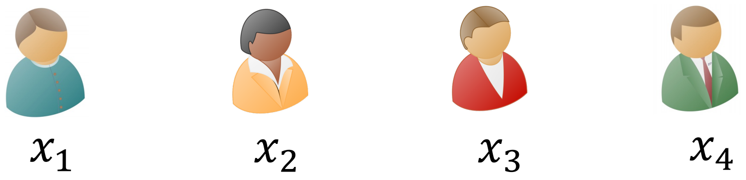
Conditional Branches: Hiding Wire Configuration

Pre-Processing Phase



Conditional Branches: Hiding Wire Configuration

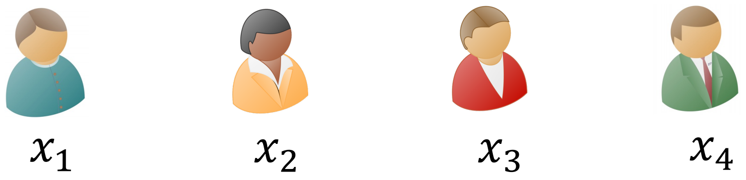
Pre-Processing Phase



For every wire w in every branch i: Assign an incoming label and an outgoing label

Conditional Branches: Hiding Wire Configuration

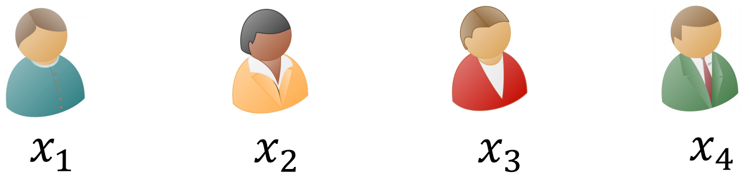
Pre-Processing Phase



For every wire w in every branch i:	Assign an incoming label and an outgoing label
For every branch i:	Compute mappings: $\pi_i : in \rightarrow out$

Conditional Branches: Hiding Wire Configuration

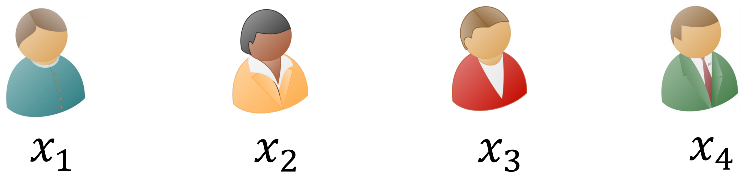
Pre-Processing Phase



For every wire w in every branch i:	Assign an incoming label and an outgoing label
For every branch i:	Compute mappings: $\pi_i : in \rightarrow out$
	Share a set of random masks: $[imask_w], [omask_w]$

Conditional Branches: Hiding Wire Configuration

Pre-Processing Phase



For every wire w in every branch i: Assign an incoming label and an outgoing label

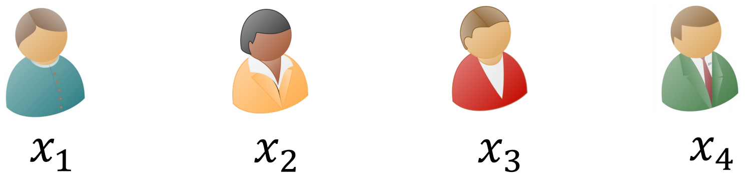
For every branch i: Compute mappings: $\pi_i : in \rightarrow out$

Share a set of random masks: $[imask_w], [omask_w]$

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

Conditional Branches: Hiding Wire Configuration

Pre-Processing Phase



For every wire w in every branch i: Assign an incoming label and an outgoing label

For every branch i: Compute mappings: $\pi_i : in \rightarrow out$

Share a set of random masks: $[imask_w], [omask_w]$

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

$$[\Delta_w] = [imask_w] - [omask_{\pi_\alpha(w)}]$$

(Δ values for the active branch)

Conditional Branches: Hiding Wire Configuration

Pre-Processing Phase

For every wire w in every branch i: | Assign an incoming label and an outgoing label

For every branch i: | Compute mappings: $\pi_i : in \rightarrow out$

Share a set of random masks: $[imask_w], [omask_w]$

This computation depends on the size of all branches

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

$$[\Delta_w] = [imask_w] - [omask_{\pi_\alpha(w)}]$$

Conditional Branches: Hiding Wire Configuration

Pre-Processing Phase

x_1 x_2 x_3 x_4

For every wire w in every branch i:	Assign an incoming label and an outgoing label
For every branch i:	Compute mappings: $\pi_i : in \to out$
	Share a set of random masks: $[imask_w], [omask_w]$
This computation depends on the size of all branches	$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$
How can we compute this efficiently?	$[\Delta_w] = [imask_w] - [omask_{\pi_\alpha(w)}]$

Oblivious Inner Product

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

Oblivious Inner Product

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

Using Threshold Linearly Homomorphic Encryption!

Oblivious Inner Product

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

Using Threshold Linearly Homomorphic Encryption!

Encrypt $b_1, \dots, b_k$:

$$< b_1 > \xleftarrow{Enc} [b_1], \dots, < b_k > \xleftarrow{Enc} [b_k]$$

Oblivious Inner Product

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

Using Threshold Linearly Homomorphic Encryption!

Encrypt $b_1, \ldots, b_k$:

$$< b_1 > \xleftarrow{Enc} [b_1], \ldots, < b_k > \xleftarrow{Enc} [b_k]$$

Each party p computes for every wire w:

$$< omask^{(p)}_{\pi_\alpha(w)} > = < b_1 >. [omask_{\pi_1(w)}]^{(p)} + \cdots + < b_k >. [omask_{\pi_k(w)}]^{(p)}$$

Oblivious Inner Product

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

Using Threshold Linearly Homomorphic Encryption!

Encrypt $b_1, \dots, b_k$:

$$< b_1 > \overset{Enc}{\longleftarrow} [b_1], \dots, < b_k > \overset{Enc}{\longleftarrow} [b_k]$$

Each party p computes for every wire w:

$$< omask_{\pi_\alpha(w)}^{(p)} > = < b_1 >. [omask_{\pi_1(w)}]^{(p)} + \cdots + < b_k >. [omask_{\pi_k(w)}]^{(p)}$$

Aggregate:

$$< omask_{\pi_\alpha(w)} > = \sum_p < omask_{\pi_\alpha(w)}^{(p)} >$$

Oblivious Inner Product

$$[omask_{\pi_\alpha(w)}] = [b_1][omask_{\pi_1(w)}] + \cdots + [b_1][omask_{\pi_1(w)}]$$

Using Threshold Linearly Homomorphic Encryption!

Encrypt $b_1, \ldots, b_k$:

$$< b_1 > \overset{Enc}{\longleftarrow} [b_1], \ldots, < b_k > \overset{Enc}{\longleftarrow} [b_k]$$

Each party p computes for every wire w:

$$< omask^{(p)}_{\pi_\alpha(w)} > = < b_1 >. [omask_{\pi_1(w)}]^{(p)} + \cdots + < b_k >. [omask_{\pi_k(w)}]^{(p)}$$

Aggregate:

$$< omask_{\pi_\alpha(w)} > = \sum_p < omask^{(p)}_{\pi_\alpha(w)} >$$

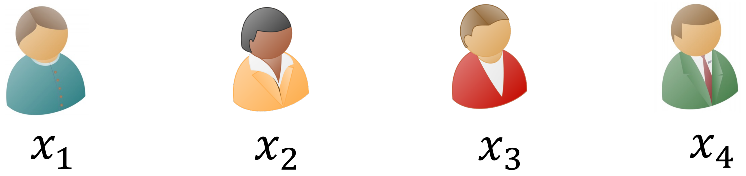
Decrypt mask to shares

$$[omask_{\pi_\alpha(w)}] \overset{Dec}{\longleftarrow} < omask_{\pi_\alpha(w)} >$$

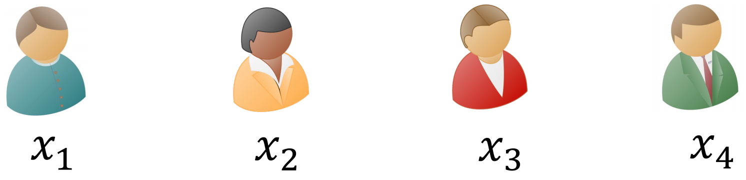
Communication only depends on the size of one branch

Conditional Branches: Hiding Wire Configuration

Online Phase

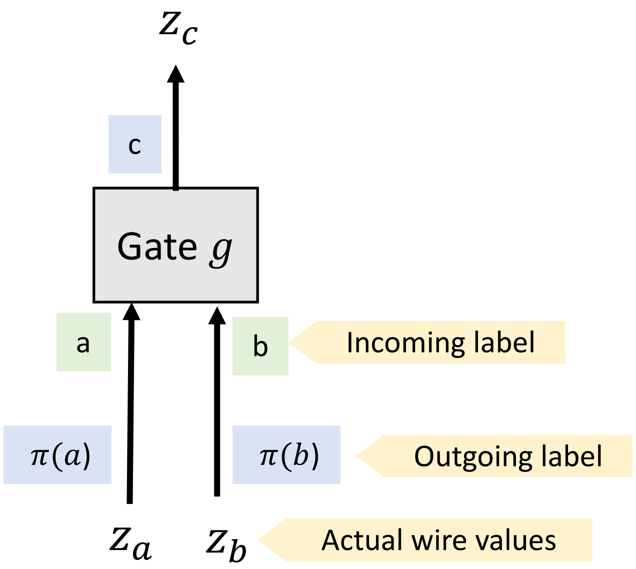


Conditional Branches: Hiding Wire Configuration



Online Phase

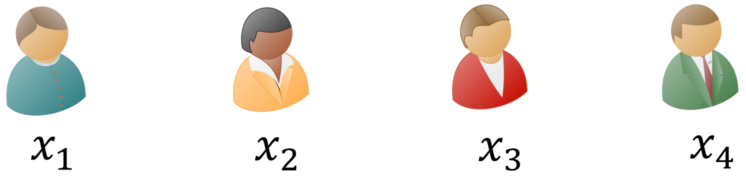
For every gate g:



Conditional Branches: Hiding Wire Configuration

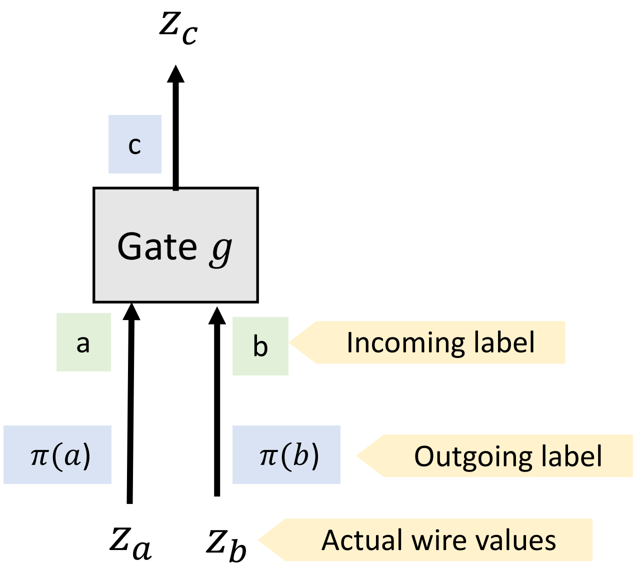
Online Phase

For every gate g:



After computing g

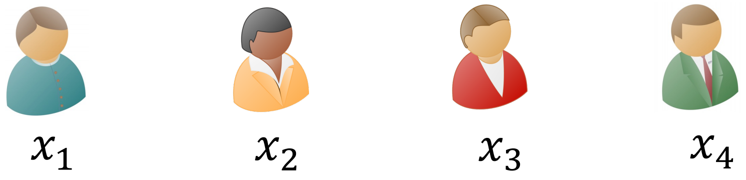
Compute $[u_c] = [z_c] + [omask_c]$
and Reconstruct u_c



Conditional Branches: Hiding Wire Configuration

Online Phase

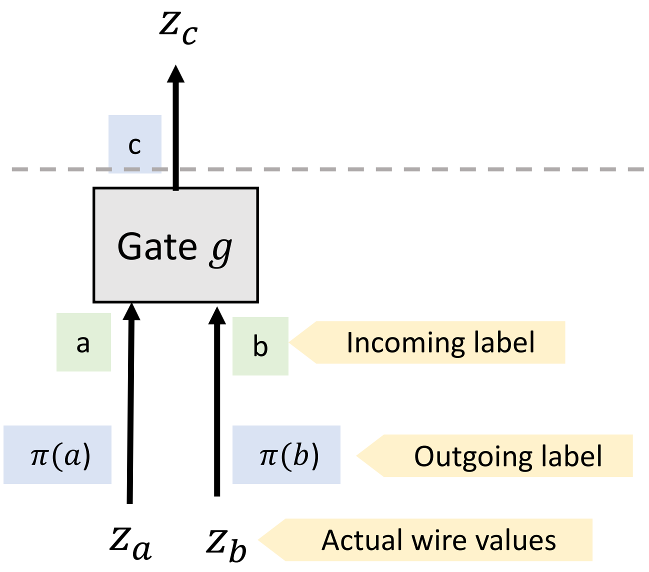
For every gate g:



After computing g

Compute $[u_c] = [z_c] + [omask_c]$
and Reconstruct u_c

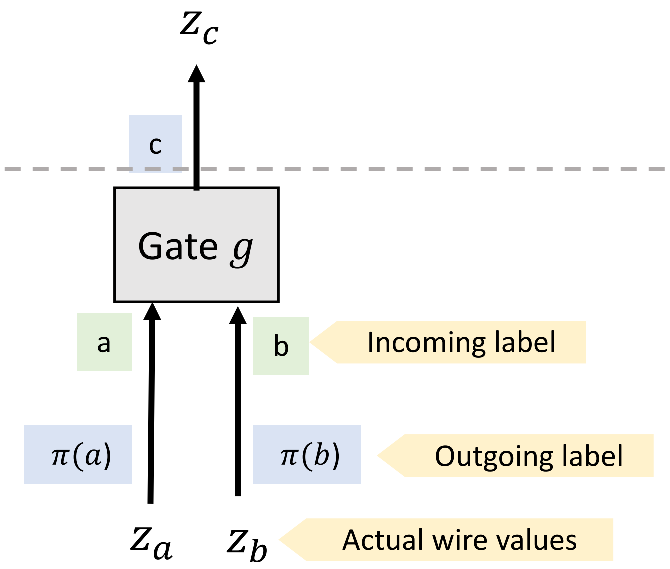
For computing g

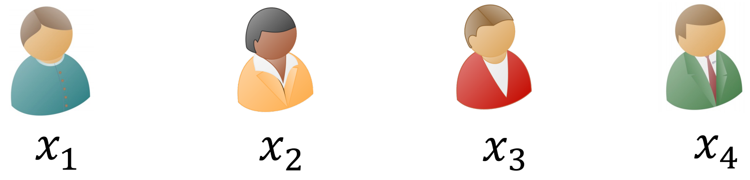


Conditional Branches: Hiding Wire Configuration

Online Phase

For every gate g:





After computing g

Compute $[u_c] = [z_c] + [omask_c]$
and Reconstruct u_c

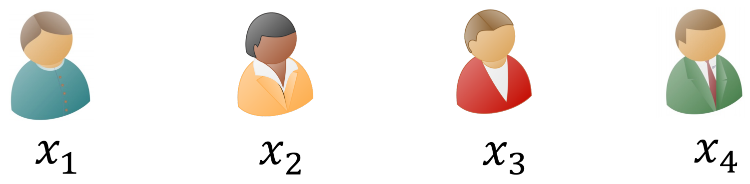
For computing g

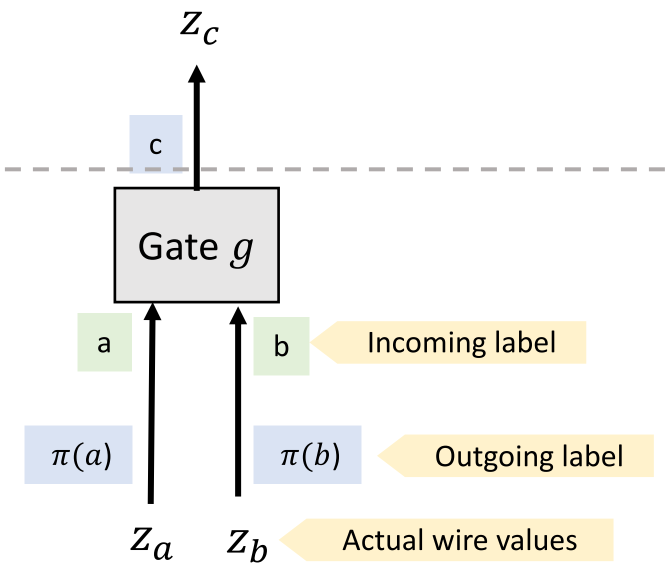
Compute
$A = u_{\pi(a)} + [\Delta_a]$
$B = u_{\pi(b)} + [\Delta_b]$

Conditional Branches: Hiding Wire Configuration

Online Phase

For every gate g:





After computing g

Compute $[u_c] = [z_c] + [omask_c]$
and Reconstruct u_c

For computing g

Compute
$$A = u_{\pi(a)} + [\Delta_a]$$
$$B = u_{\pi(b)} + [\Delta_b]$$

Compute
$$[z_a] = A - [imask_a]$$
$$[z_b] = B - [imask_b]$$

Talk Outline

- Overview of the [MS13] PFE protocol
- Our semi-honest non-constant round protocol for conditional branches
- Performance Evaluation
- Remarks on additional results

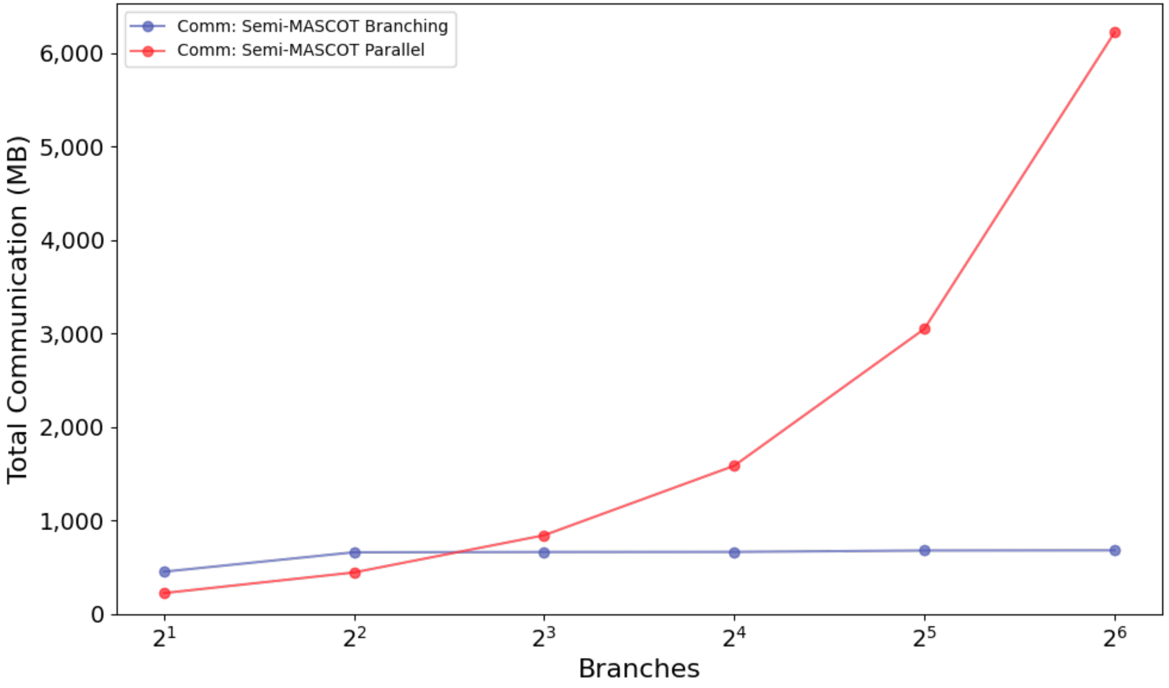
Comparing with MASCOT ($O(n^2|C|)$ Protocol)

Comparing with MASCOT ($O(n^2|C|)$ Protocol)

We use an implementation of semi-honest MASCOT from the MP-SPDZ library

Comparing with MASCOT ($O(n^2|C|)$ Protocol)

We use an implementation of semi-honest MASCOT from the MP-SPDZ library

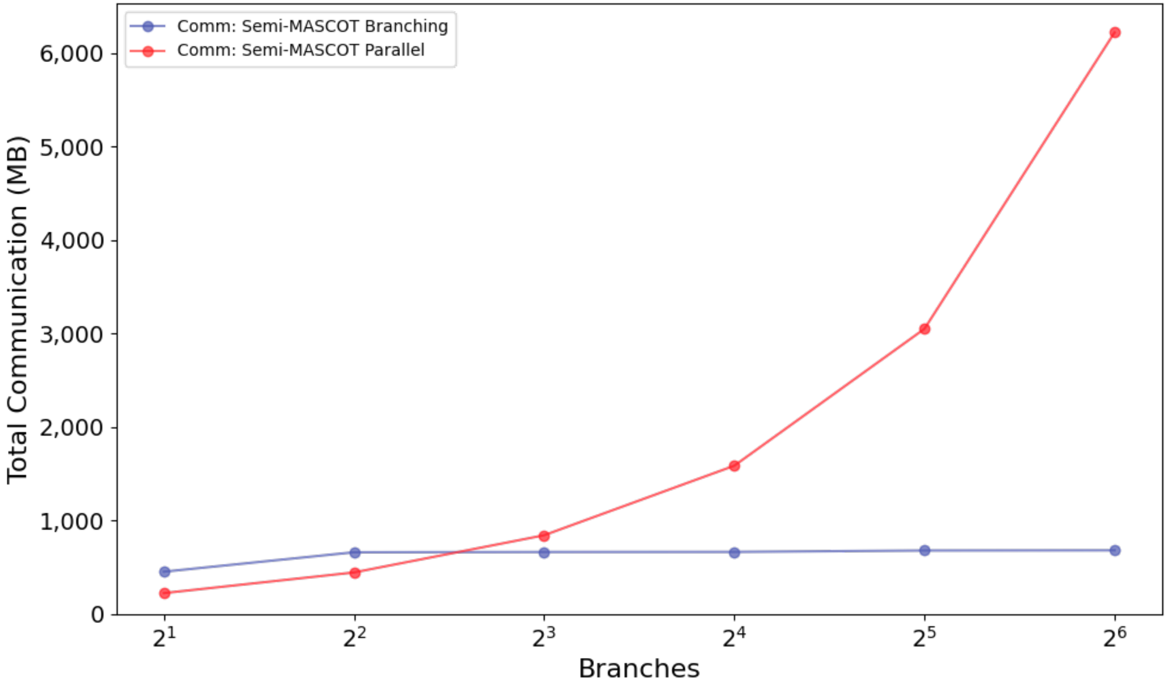


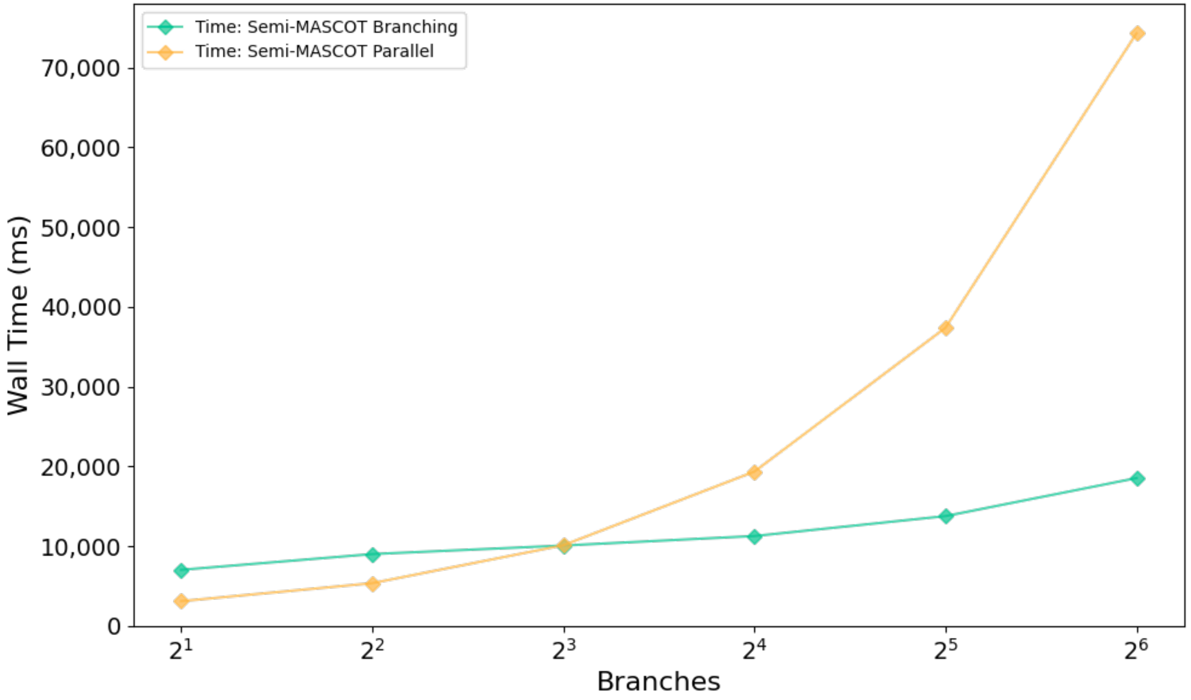
Communication

3 parties, 2^{16} gates in each branch

Comparing with MASCOT ($O(n^2|C|)$ Protocol)

We use an implementation of semi-honest MASCOT from the MP-SPDZ library





Communication

Run-Time

3 parties, 2^{16} gates in each branch

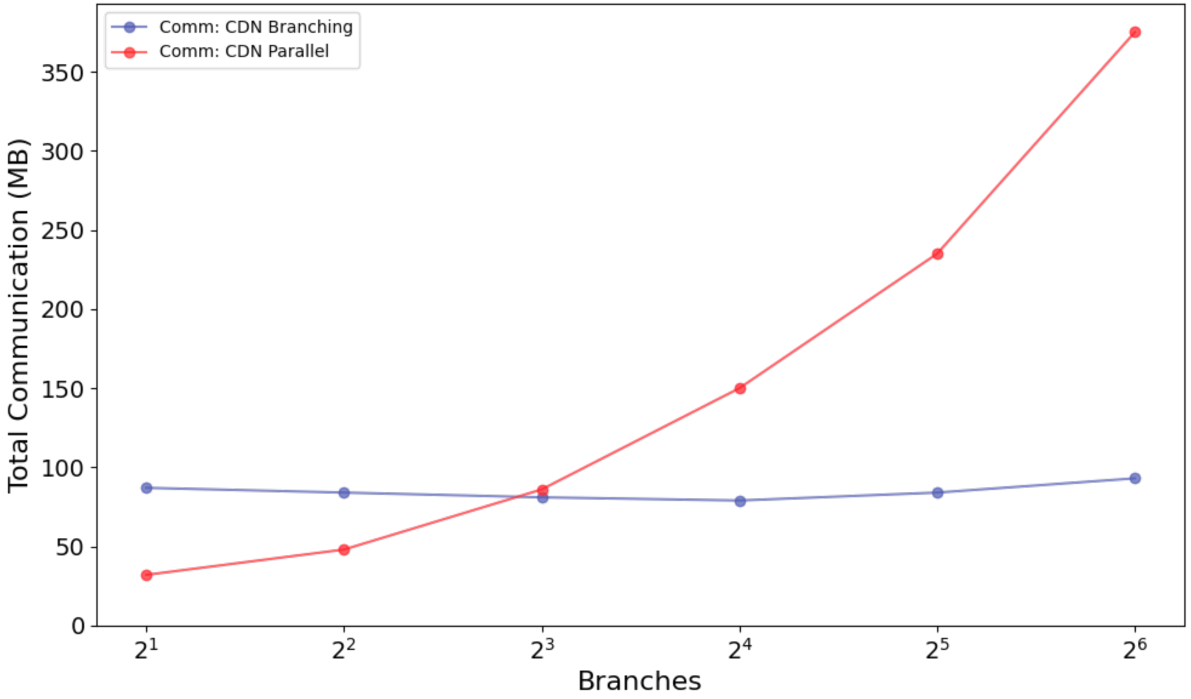
Comparing with [CDN01] ($O(n|C|)$ Protocol)

Comparing with [CDN01] ($O(n|C|)$ Protocol)

We implement the [CDN01] Protocol

Comparing with [CDN01] $(O(n|C|)$ Protocol)

We implement the [CDN01] Protocol

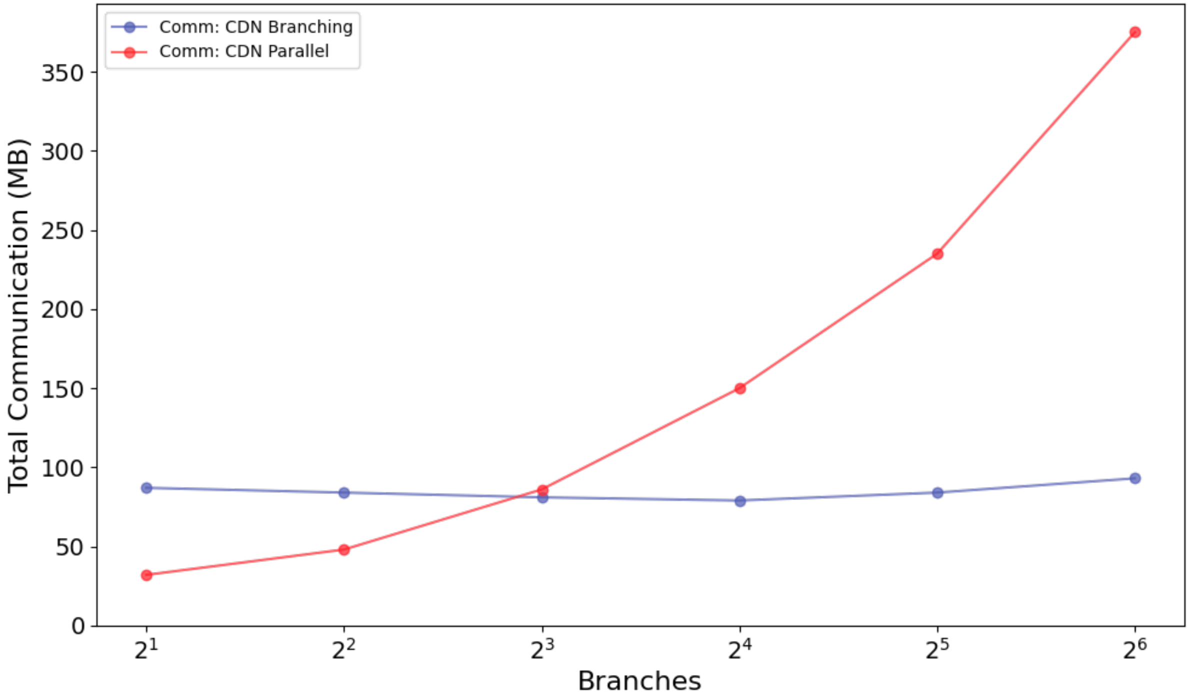


Communication

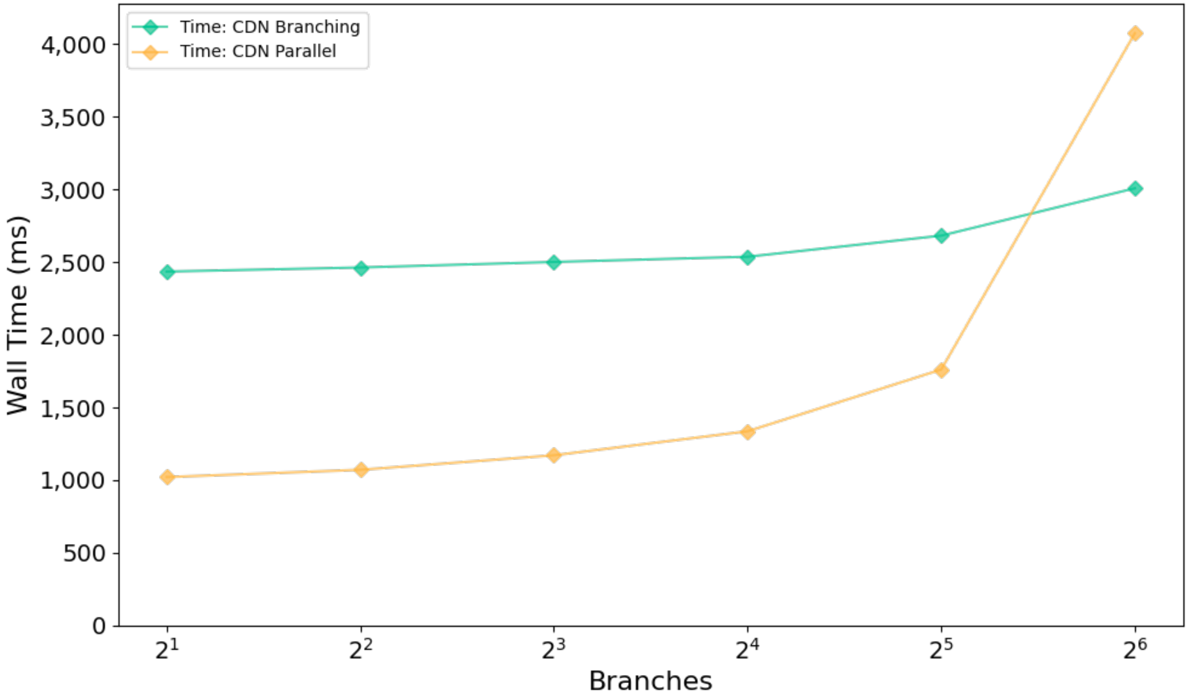
3 parties, 2^{16} gates in each branch

Comparing with [CDN01] ($O(n|C|)$ Protocol)

We implement the [CDN01] Protocol



Communication



Run-Time

3 parties, 2^{16} gates in each branch

Talk Outline

- Overview of the [MS13] PFE protocol
- Our semi-honest non-constant round protocol for conditional branches
- Performance Evaluation
- Remarks on additional results

Additional Results

This protocol can be extended to malicious security

Additional Results

This protocol can be extended to malicious security

Incurs additional multiplicative overhead dependent on the statistical security parameter

Additional Results

This protocol can be extended to malicious security

Incurs additional multiplicative overhead dependent on the statistical security parameter

A constant round variant based on multi-party garbling

Additional Results

This protocol can be extended to malicious security

Incurs additional multiplicative overhead dependent on the statistical security parameter

A constant round variant based on multi-party garbling

Naïve multi-party garbling using the above approach results in non-black box use of cryptography

Additional Results

This protocol can be extended to malicious security

Incurs additional multiplicative overhead dependent on the statistical security parameter

A constant round variant based on multi-party garbling

Naïve multi-party garbling using the above approach results in non-black box use of cryptography

We present an alternate solution using the linearly key-homomorphic PRFs based garbling approach from [BLO17]

Conclusion

A multi-party protocol for securely computing conditional branches, where the total communication only depends on the size of the largest branch

Extensions to malicious security and a semi-honest constant round protocol

Thank You!