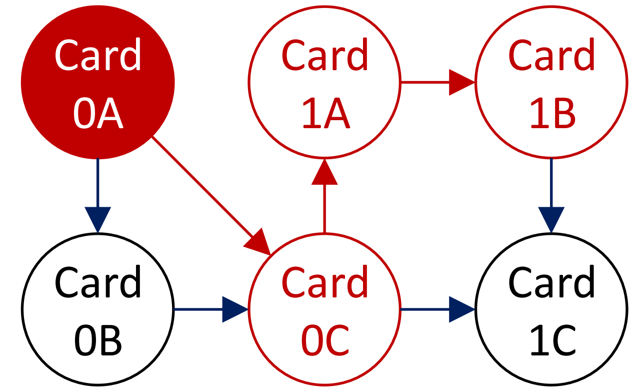
GraphAce: Secure Two-Party Graph Analysis Achieving Communication Efficiency

Jiping Yu, Kun Chen, Yunyi Chen, Xiaoyu Fan,
Xiaowei Zhu, Cheng Hong, Wenguang Chen

Background

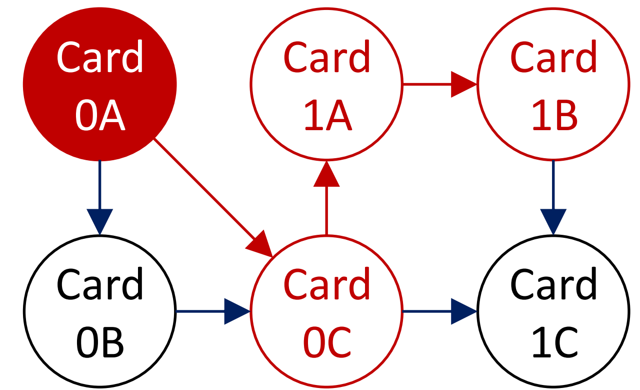
Graph Analysis

- Iterative workload on a static graph
  - Immutable graph topology
  - Mutable vertex attributes



Graph Analysis

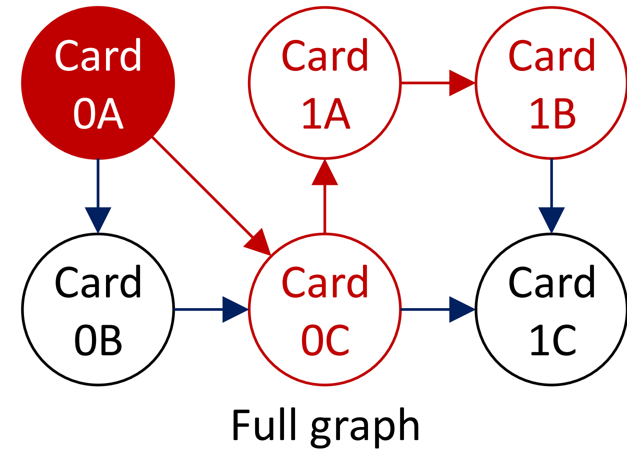
- Iterative workload on a static graph
  - Immutable graph topology
  - Mutable vertex attributes



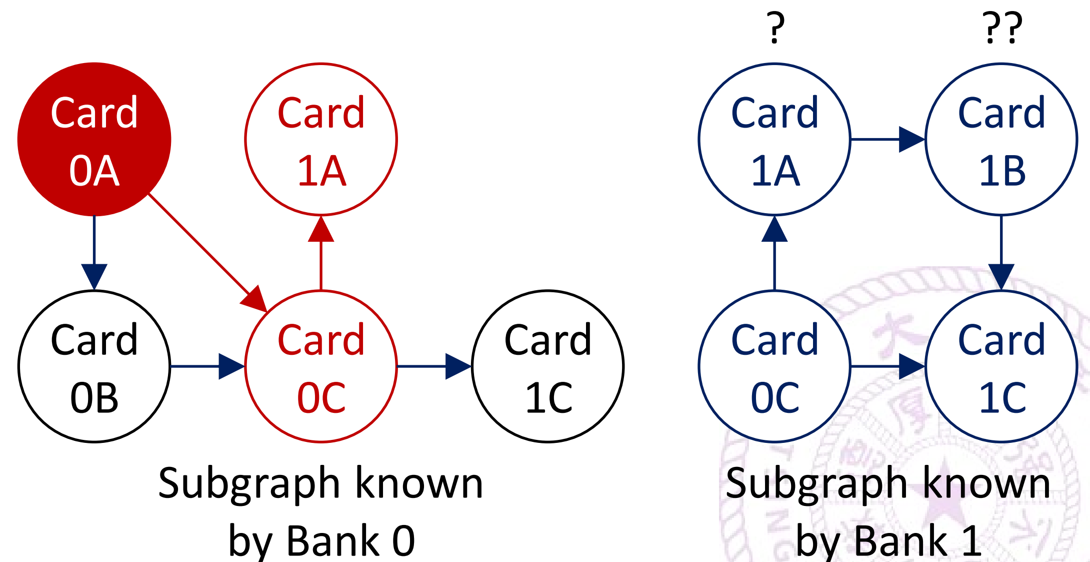
- Example: financial graph
  - Card (account) = vertex; transaction = edge
  - Assume Card 0A was marked high-risk
  - The risk of Card 1B should increase (three-hop via Card 0C – Card 1A)

Secure Two-Party Graph Analysis

- In reality, graph data arise from different entities
  - Each bank knows a subgraph



Full graph

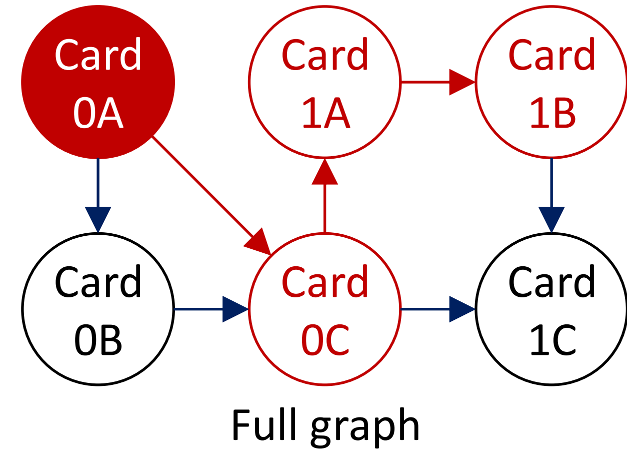


Subgraph known by Bank 0

Subgraph known by Bank 1

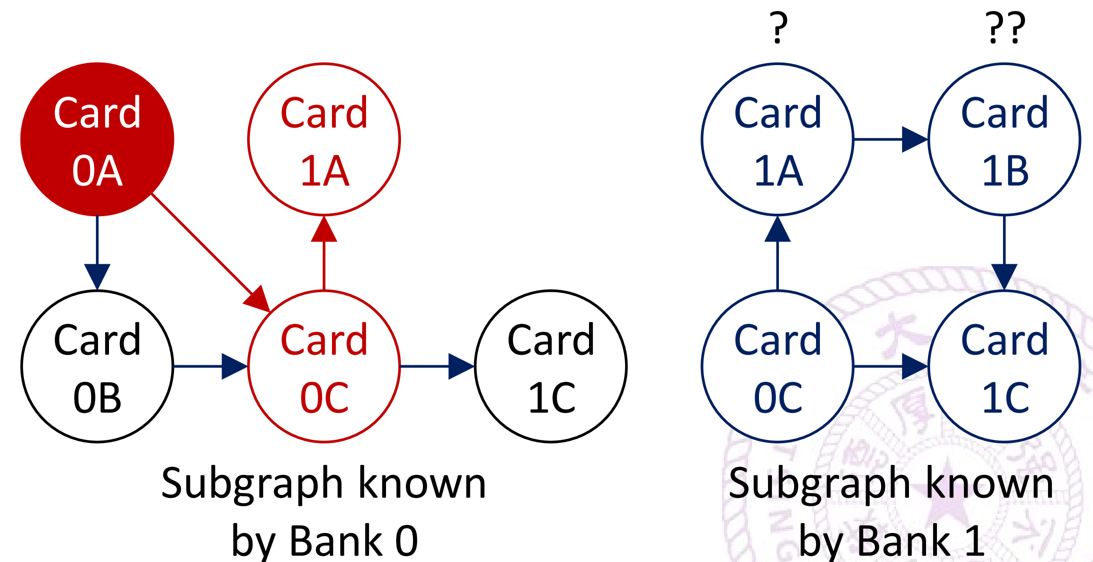
Secure Two-Party Graph Analysis

- In reality, graph data arise from different entities
  - Each bank knows a subgraph



Full graph

- Analyzing the full graph is better than individual analyses
  - If banks operate individually, Bank 1 cannot raise the risk of Card 1B

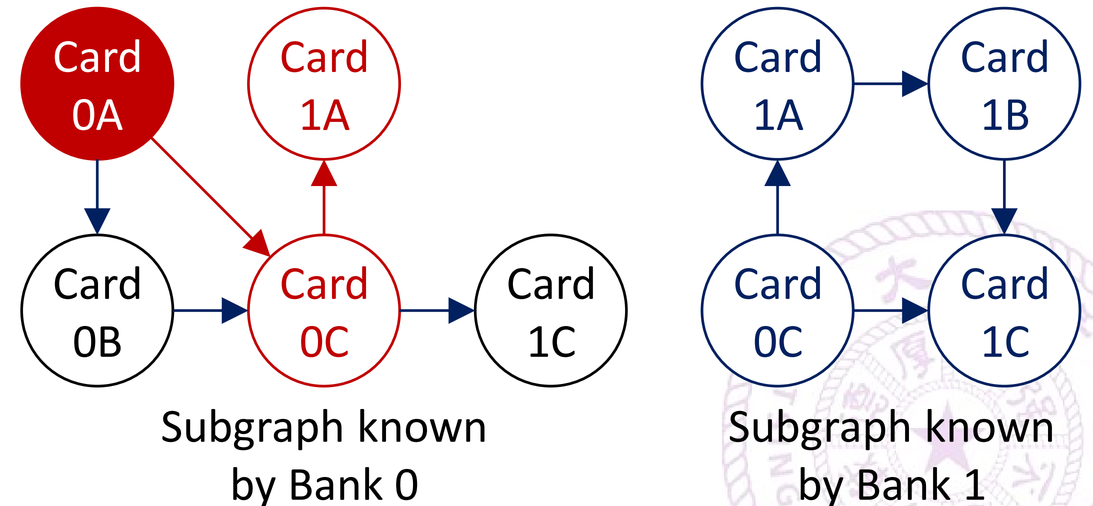


Subgraph known by Bank 0

Subgraph known by Bank 1

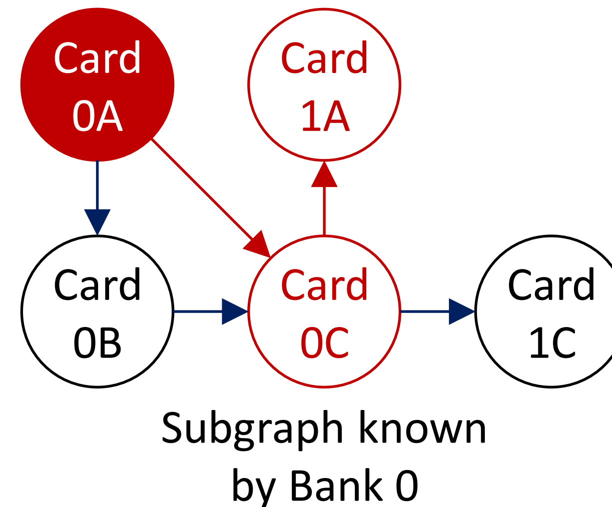
Secure Two-Party Graph Analysis

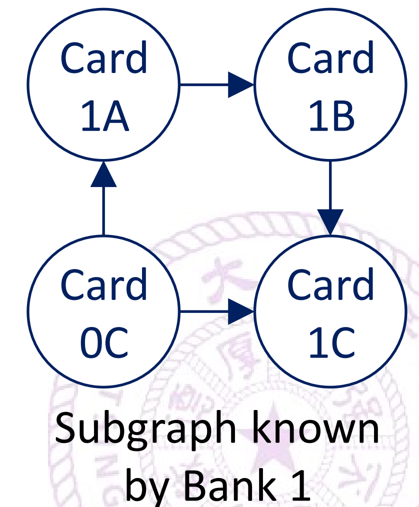
- Subgraphs cannot be directly exchanged in many cases
  - Each party may consider its subgraph valuable
  - Laws may prohibit transferring sensitive data
  - ……



Secure Two-Party Graph Analysis

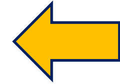
- Subgraphs cannot be directly exchanged in many cases
  - Each party may consider its subgraph valuable
  - Laws may prohibit transferring sensitive data
  - ……
- We need secure analysis!
  - Correctness: each party learns analysis results of its vertices
  - Security: each party learns nothing beyond the above results
  - (Specifically, about the other party's graph, or intermediate values)





Related Work

	Communication per iteration	Parties
GraphSC [NWI+15]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log^2\lvert V\rvert\big)$	2-party, dishonest majority



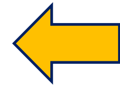
- GraphSC
  - First secure graph analysis framework with sub-quadratic complexity
  - Use bitonic sorting + scanning to convert graph analysis into circuits
    - Resulting in the $\log^2$ factor due to sorting
  - Use GC (garbled circuits) for secure 2-party circuit evaluation

----
The complexity assumes $\lvert E\rvert \leq \lvert V\rvert^2$.

[NWI+15] Nayak, K., Wang, X. S., Ioannidis, S., Weinsberg, U., Taft, N., & Shi, E. 2015. GraphSC: Parallel secure computation made easy. In 2015 IEEE symposium on security and privacy.

Related Work

	Communication per iteration	Parties
GraphSC [NWI+15]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log^2\lvert V\rvert\big)$	2-party, dishonest majority
[AFO+21]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log\lvert V\rvert\big)$	3-party, honest majority



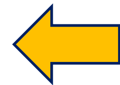
- Araki et al.
  - Based on GraphSC
  - Optimize sorting into a 3-party permuting protocol with SS (secret sharing)
    - Bottleneck becomes parallel scanning ($\log$), instead of sorting ($\log^2$)
  - Assuming honest majority (no two parties collude)

The complexity assumes a vertex may connect to $\Theta(\lvert V\rvert)$ vertices.

[AFO+21] Araki, T., Furukawa, J., Ohara, K., Pinkas, B., Rosemarin, H., & Tsuchida, H. 2021. Secure graph analysis at scale. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.

Related Work

	Communication per iteration	Parties
GraphSC [NWI+15]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log^2\lvert V\rvert\big)$	2-party, dishonest majority
[AFO+21]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log\lvert V\rvert\big)$	3-party, honest majority
Graphiti [KKPG24]	$\Theta(\lvert V\rvert + \lvert E\rvert)$ *	2-party + 1 helper, honest majority



- Graphiti (concurrent work)
  - Adopt communication-free prefix sum over SS (secret sharing)
    - Get rid of parallel scanning ($\log$)
    - Only need secure permuting (linear)
  - Still assuming honest majority (any party does not collude with the helper)

----

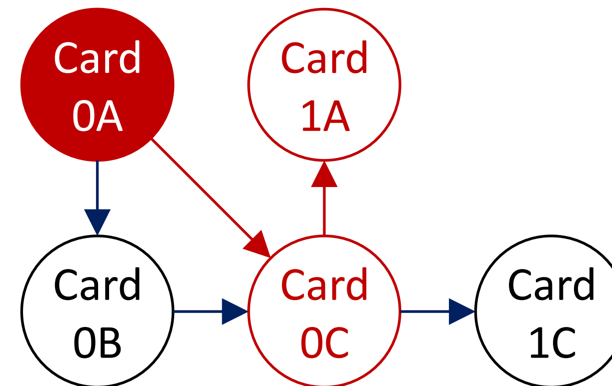
* The first iteration needs $\Theta\big((\lvert V\rvert + \lvert E\rvert)\log\lvert V\rvert\big)$ communication.

[KKPG24] Koti, N., Kukkala, V. B., Patra, A., & Raj Gopal, B. 2024. Graphiti: Secure Graph Computation Made More Scalable. In Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security.

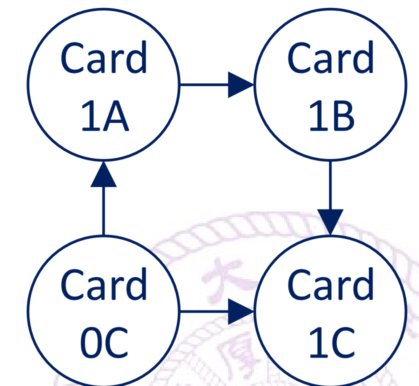
Related Work

	Communication per iteration	Parties	End to end?
GraphSC [NWI+15]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log^2\lvert V\rvert\big)$	2-party, dishonest majority	No
[AFO+21]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log\lvert V\rvert\big)$	3-party, honest majority	No
Graphiti [KKPG24]	$\Theta(\lvert V\rvert + \lvert E\rvert)$	2-party + 1 helper, honest majority	No

- Previous studies are not end-to-end frameworks
  - They assume the graph is already garbled (or secret-shared) among parties
  - Did not discuss the conversion from local subgraphs to garbled full graph
  - Did not discuss how to distribute results to parties



Subgraph known by Bank 0



Subgraph known by Bank 1

Related Work

	Communication per iteration	Parties	End to end?
GraphSC [NWI+15]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log^2\lvert V\rvert\big)$	2-party, dishonest majority	No
[AFO+21]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log\lvert V\rvert\big)$	3-party, honest majority	No
Graphiti [KKPG24]	$\Theta(\lvert V\rvert + \lvert E\rvert)$	2-party + 1 helper, honest majority	No

- For secure two-party graph analysis,
- Can we ……
  - Support end-to-end analysis
  - Without additional non-colluding party
  - Using even less communication?

Related Work

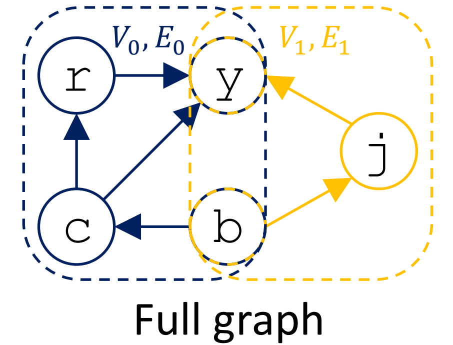
	Communication per iteration	Parties	End to end?
GraphSC [NWI+15]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log^2\lvert V\rvert\big)$	2-party, dishonest majority	No
[AFO+21]	$\Theta\big((\lvert V\rvert + \lvert E\rvert)\log\lvert V\rvert\big)$	3-party, honest majority	No
Graphiti [KKPG24]	$\Theta(\lvert V\rvert + \lvert E\rvert)$	2-party + 1 helper, honest majority	No
GraphAce	$\Theta(\lvert V\rvert)$	2-party, dishonest majority	Yes

- Yes! GraphAce ……
  - Supports end-to-end analysis
  - Without additional non-colluding party
  - Completely eliminate any communication about $\lvert E\rvert$

Design of GraphAce

Problem Formulation

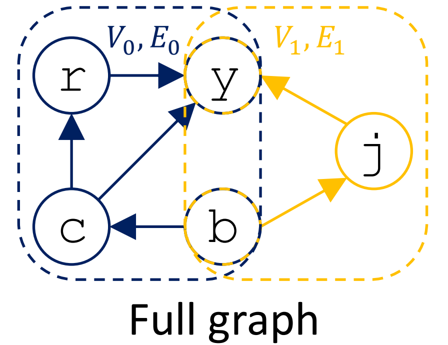
- Each Party b inputs vertex set V_b and edge set E_b
  - Including weights



Full graph

Problem Formulation

- Each Party b inputs vertex set V_b and edge set E_b
  - Including weights

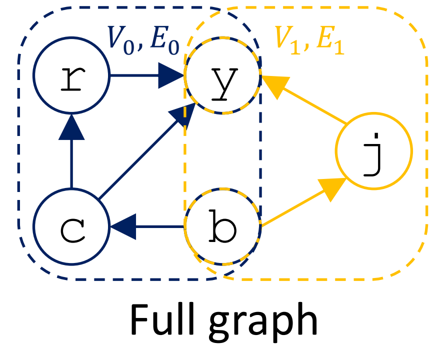


Full graph

- Public information
  - Specification of graph analysis application (under GAS model)
  - $\lceil \log_2 |V_0| \rceil, \lceil \log_2 |V_1| \rceil$

Problem Formulation

- Each Party b inputs vertex set V_b and edge set E_b
  - Including weights



Full graph

- Public information
  - Specification of graph analysis application (under GAS model)
  - $\lceil \log_2 |V_0| \rceil, \lceil \log_2 |V_1| \rceil$

- Security: parties learns nothing beyond final analysis results
  - Specifically, Party 0 learns nothing about:
  - V_1, E_1 (vertices, edges, or weights)
  - Any intermediate results

Insights from Plaintext Graph Analysis

- Observation: (Insecure) two-party graph analysis is equivalent to distributed (plaintext) graph analysis over two machines

Insights from Plaintext Graph Analysis

- Observation: (Insecure) two-party graph analysis is equivalent to distributed (plaintext) graph analysis over two machines

- However, almost all distributed plaintext systems do not require per-iteration communication about $|E|$

Insights from Plaintext Graph Analysis

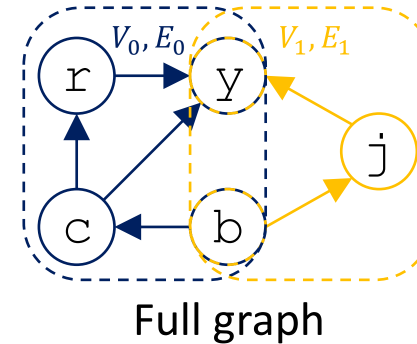
- Observation: (Insecure) two-party graph analysis is equivalent to distributed (plaintext) graph analysis over two machines

- However, almost all distributed plaintext systems do not require per-iteration communication about $|E|$

- Inspiration: resetting to plaintext solution, then successively addressing its security concerns
  - (and try not to introduce $|E|$ communication)

Insecure Two-Party Graph Analysis

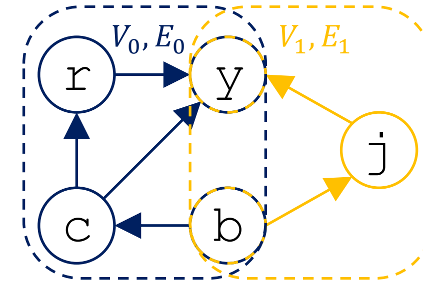
- An example algorithm
  - $w^t(v) = \sum_{(u,v)\in E} w^{t-1}(u)$
  - (sum the weights from incoming edges)
  - For vertex y,
    $w^t(\mathrm{y}) = w^{t-1}(\mathrm{r}) + w^{t-1}(\mathrm{c}) + w^{t-1}(\mathrm{j})$



Full graph

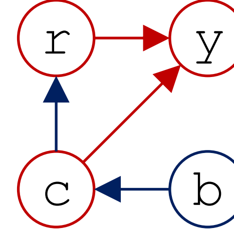
Insecure Two-Party Graph Analysis

- An example algorithm
  - $w^t(v) = \sum_{(u,v)\in E} w^{t-1}(u)$
  - (sum the weights from incoming edges)
  - For vertex $\mathtt{y}$,
    $w^t(\mathtt{y}) = w^{t-1}(\mathtt{r}) + w^{t-1}(\mathtt{c}) + w^{t-1}(\mathtt{j})$
- Local step
  - Party 0 computes $w^{t-1}(\mathtt{r}) + w^{t-1}(\mathtt{c})$
  - Party 1 computes $w^{t-1}(\mathtt{j})$

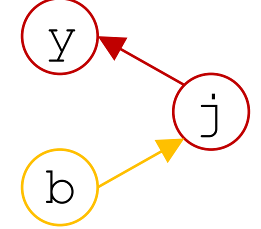


Full graph

$s_0^t(\mathtt{y}) = w^{t-1}(\mathtt{r}) + w^{t-1}(\mathtt{c})$



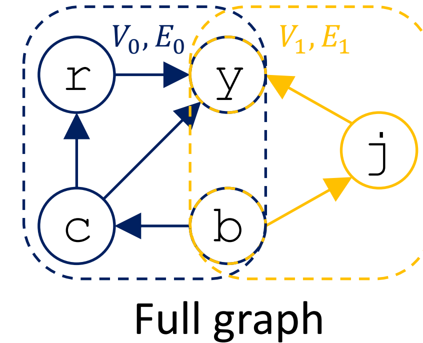
$s_1^t(\mathtt{y}) = w^{t-1}(\mathtt{j})$



Local message passing (LMP)

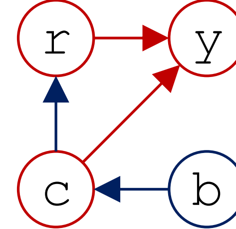
Insecure Two-Party Graph Analysis

- An example algorithm
  - $w^t(v) = \sum_{(u,v)\in E} w^{t-1}(u)$
  - (sum the weights from incoming edges)
  - For vertex y,
    $w^t(\mathrm{y}) = w^{t-1}(\mathrm{r}) + w^{t-1}(\mathrm{c}) + w^{t-1}(\mathrm{j})$
- Local step
  - Party 0 computes $w^{t-1}(\mathrm{r}) + w^{t-1}(\mathrm{c})$
  - Party 1 computes $w^{t-1}(\mathrm{j})$
- Cross-party step
  - Interchange the values
  - Sum up

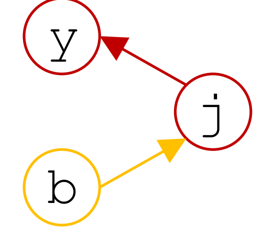


Full graph

$s_0^t(\mathrm{y}) = w^{t-1}(\mathrm{r}) + w^{t-1}(\mathrm{c})$

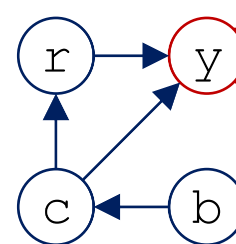


$s_1^t(\mathrm{y}) = w^{t-1}(\mathrm{j})$



Local message passing (LMP)

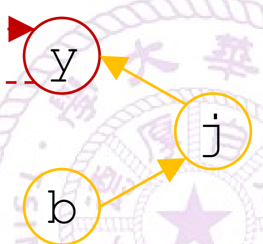
$w^t(\mathrm{y}) = s_0^t(\mathrm{y}) + s_1^t(\mathrm{y})$



Send $s_0^t(\mathrm{y})$

Send $s_1^t(\mathrm{y})$

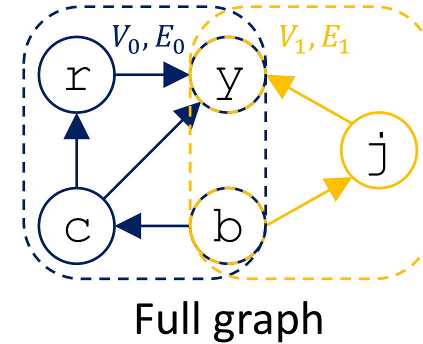
$w^t(\mathrm{y}) = s_0^t(\mathrm{y}) + s_1^t(\mathrm{y})$



Cross-party interchange (CPI)

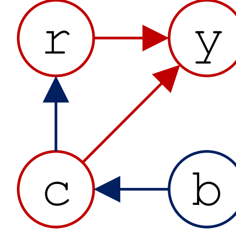
Insecure Two-Party Graph Analysis

- $\Theta(1)$ communication for each vertex
  - Independent of the degree; only send local sum

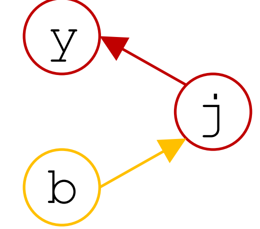


Full graph

$s_0^t(y) = w^{t-1}(r) + w^{t-1}(c)$

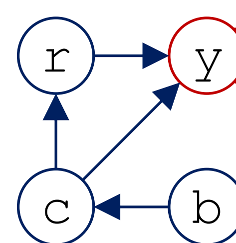


$s_1^t(y) = w^{t-1}(j)$



Local message passing (LMP)

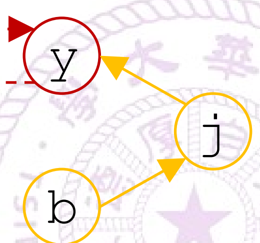
$w^t(y) = s_0^t(y) + s_1^t(y)$



Send $s_0^t(y)$

Send $s_1^t(y)$

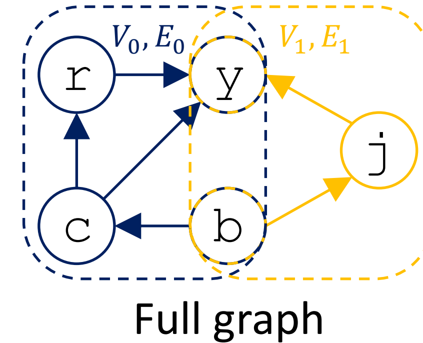
$w^t(y) = s_0^t(y) + s_1^t(y)$



Cross-party interchange (CPI)

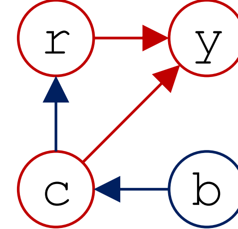
Insecure Two-Party Graph Analysis

- $\Theta(1)$ communication for each vertex
  - Independent of the degree; only send local sum
- $\Theta(|V|)$ for all vertices, per iteration

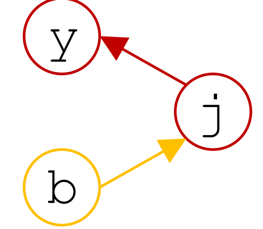


Full graph

$s_0^t(\mathrm{y}) = w^{t-1}(\mathrm{r}) + w^{t-1}(\mathrm{c})$

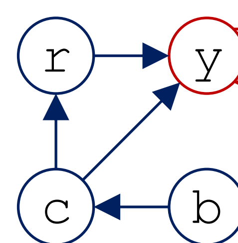


$s_1^t(\mathrm{y}) = w^{t-1}(\mathrm{j})$



Local message passing (LMP)

$w^t(\mathrm{y}) = s_0^t(\mathrm{y}) + s_1^t(\mathrm{y})$



Send $s_0^t(\mathrm{y})$

Send $s_1^t(\mathrm{y})$

$w^t(\mathrm{y}) = s_0^t(\mathrm{y}) + s_1^t(\mathrm{y})$

y

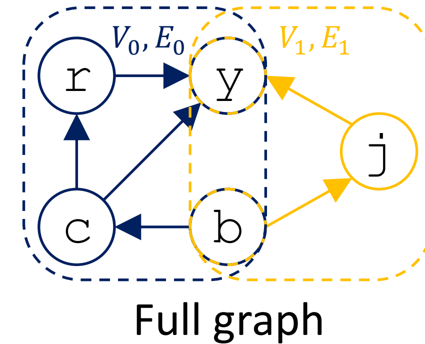
j

b

Cross-party interchange (CPI)

Insecure Two-Party Graph Analysis

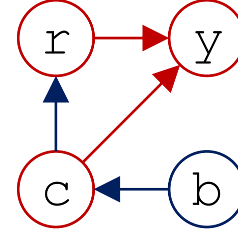
- $\Theta(1)$ communication for each vertex
  - Independent of the degree; only send local sum



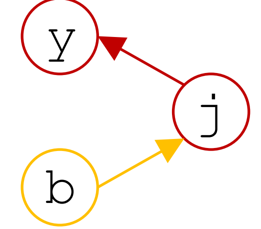
Full graph

- $\Theta(|V|)$ for all vertices, per iteration

$s_0^t(\mathrm{y}) = w^{t-1}(\mathrm{r}) + w^{t-1}(\mathrm{c})$



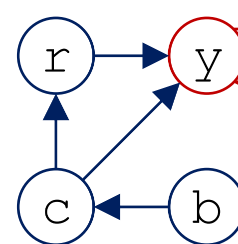
$s_1^t(\mathrm{y}) = w^{t-1}(\mathrm{j})$



Local message passing (LMP)

- Security problems
  - Weight privacy: intermediate results are sent in plaintext
  - Obliviousness: sending local sums leaks the vertex sets V_0, V_1 (the other party knows which vertices are sent)

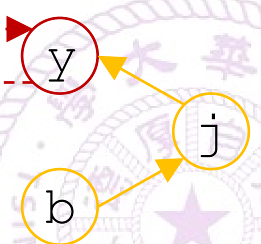
$w^t(\mathrm{y}) = s_0^t(\mathrm{y}) + s_1^t(\mathrm{y})$



Send $s_0^t(\mathrm{y})$

Send $s_1^t(\mathrm{y})$

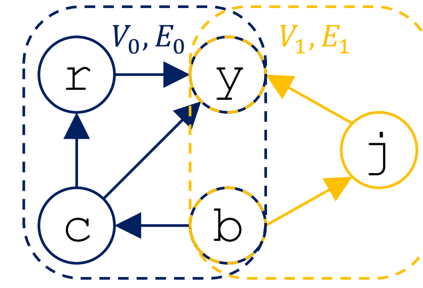
$w^t(\mathrm{y}) = s_0^t(\mathrm{y}) + s_1^t(\mathrm{y})$



Cross-party interchange (CPI)

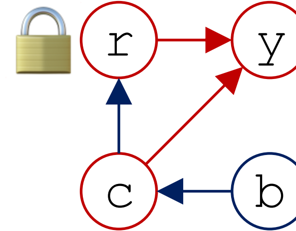
Protect Weights with Mixed Primitives

- Homomorphic Encryption (HE)
  - For local computation
  - $[\![x]\!]_b^H$ denotes encrypted x, with Party $(1-b)$'s private key, store in Party b
  - (so that no party can directly decrypt)

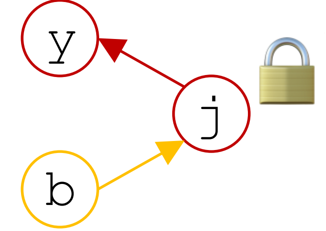


Full graph

$[\![s_0^t(\mathrm{y})]\!]_0^H = [\![w^{t-1}(\mathrm{r})]\!]_0^H \boxplus [\![w^{t-1}(\mathrm{c})]\!]_0^H$



$[\![s_1^t(\mathrm{y})]\!]_1^H = [\![w^{t-1}(\mathrm{j})]\!]_1^H$

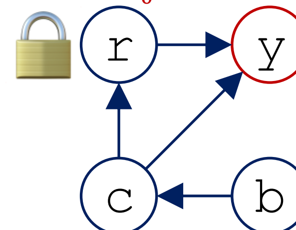


Local message passing (LMP)

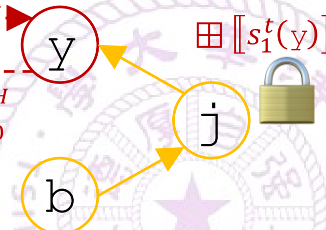
$[\![w^t(\mathrm{y})]\!]_0^H = [\![s_0^t(\mathrm{y})]\!]_0^H \boxplus [\![s_1^t(\mathrm{y})]\!]_0^H$

Convert $[\![s_0^t(\mathrm{y})]\!]_0^H$ to $[\![s_0^t(\mathrm{y})]\!]_1^H$

$[\![w^t(\mathrm{y})]\!]_1^H = [\![s_0^t(\mathrm{y})]\!]_1^H \boxplus [\![s_1^t(\mathrm{y})]\!]_1^H$



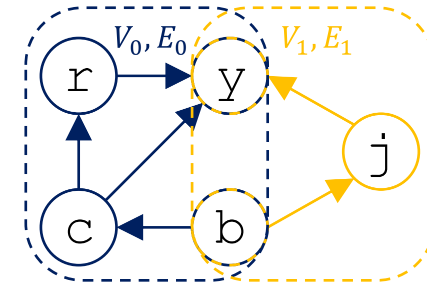
Convert $[\![s_1^t(\mathrm{y})]\!]_1^H$ to $[\![s_1^t(\mathrm{y})]\!]_0^H$



Cross-party interchange (CPI)

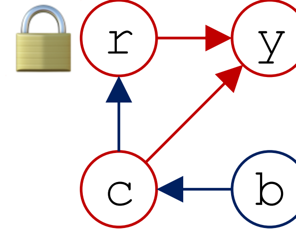
Protect Weights with Mixed Primitives

- Homomorphic Encryption (HE)
  - For local computation
  - $[\![x]\!]_b^H$ denotes encrypted x, with Party $(1-b)$'s private key, store in Party b
  - (so that no party can directly decrypt)
- Secret-Sharing (SS)
  - Enables cross-party conversions
  - (e.g., from $[\![s_0^t(\mathrm{y})]\!]_0^H$ to $[\![s_0^t(\mathrm{y})]\!]_1^H$)

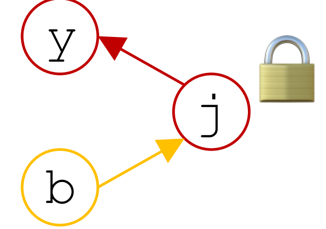


Full graph

$[\![s_0^t(\mathrm{y})]\!]_0^H = [\![w^{t-1}(\mathrm{r})]\!]_0^H \boxplus [\![w^{t-1}(\mathrm{c})]\!]_0^H$



$[\![s_1^t(\mathrm{y})]\!]_1^H = [\![w^{t-1}(\mathrm{j})]\!]_1^H$

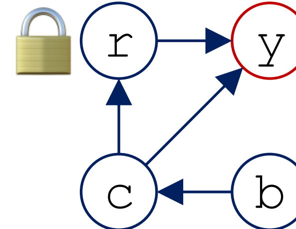


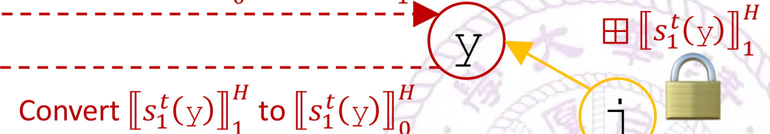
Local message passing (LMP)

$[\![w^t(\mathrm{y})]\!]_0^H = [\![s_0^t(\mathrm{y})]\!]_0^H \boxplus [\![s_1^t(\mathrm{y})]\!]_0^H$

Convert $[\![s_0^t(\mathrm{y})]\!]_0^H$ to $[\![s_0^t(\mathrm{y})]\!]_1^H$

$[\![w^t(\mathrm{y})]\!]_1^H = [\![s_0^t(\mathrm{y})]\!]_1^H \boxplus [\![s_1^t(\mathrm{y})]\!]_1^H$





Convert $[\![s_1^t(\mathrm{y})]\!]_1^H$ to $[\![s_1^t(\mathrm{y})]\!]_0^H$

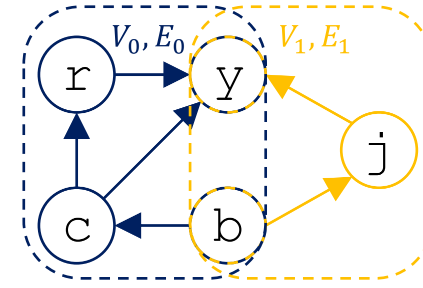
j

b

Cross-party interchange (CPI)

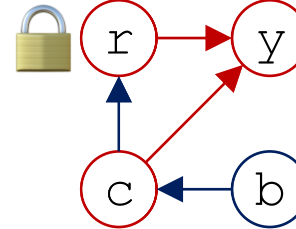
Protect Weights with Mixed Primitives

- Homomorphic Encryption (HE)
  - For local computation
  - $[\![x]\!]_b^H$ denotes encrypted x, with Party $(1-b)$'s private key, store in Party b
  - (so that no party can directly decrypt)
- Secret-Sharing (SS)
  - Enables cross-party conversions
  - (e.g., from $[\![s_0^t(\mathrm{y})]\!]_0^H$ to $[\![s_0^t(\mathrm{y})]\!]_1^H$)
- Local and cross-party steps now operate on ciphertext, rather than plaintext

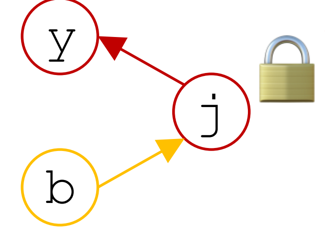


Full graph

$[\![s_0^t(\mathrm{y})]\!]_0^H = [\![w^{t-1}(\mathrm{r})]\!]_0^H \boxplus [\![w^{t-1}(\mathrm{c})]\!]_0^H$



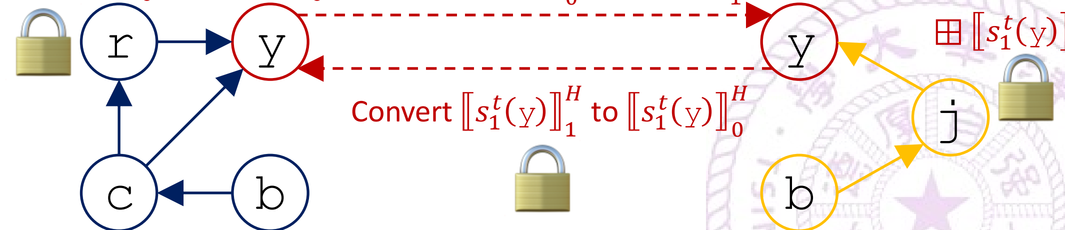
$[\![s_1^t(\mathrm{y})]\!]_1^H = [\![w^{t-1}(\mathrm{j})]\!]_1^H$



Local message passing (LMP)

$[\![w^t(\mathrm{y})]\!]_0^H = [\![s_0^t(\mathrm{y})]\!]_0^H \boxplus [\![s_1^t(\mathrm{y})]\!]_0^H$

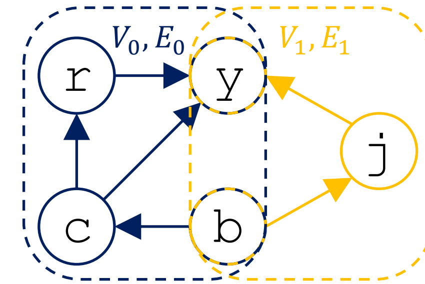
$[\![w^t(\mathrm{y})]\!]_1^H = [\![s_0^t(\mathrm{y})]\!]_1^H \boxplus [\![s_1^t(\mathrm{y})]\!]_1^H$



Cross-party interchange (CPI)

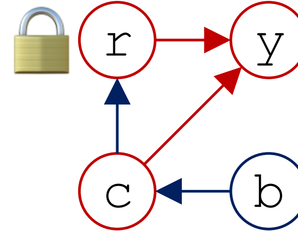
Protect Weights with Mixed Primitives

- Mix is necessary!
  - SS only? Would introduce per-edge communication (cannot operate locally)
  - HE only? Cannot operate cooperatively

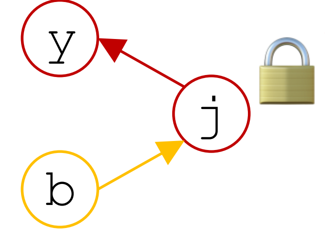


Full graph

$[\![s_0^t(\mathrm{y})]\!]_0^H = [\![w^{t-1}(\mathrm{r})]\!]_0^H \boxplus [\![w^{t-1}(\mathrm{c})]\!]_0^H$

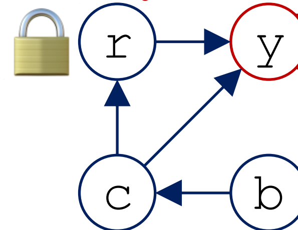


$[\![s_1^t(\mathrm{y})]\!]_1^H = [\![w^{t-1}(\mathrm{j})]\!]_1^H$



Local message passing (LMP)

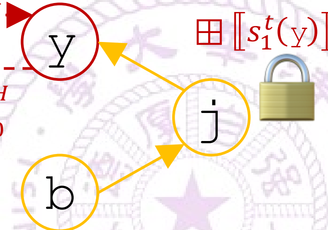
$[\![w^t(\mathrm{y})]\!]_0^H = [\![s_0^t(\mathrm{y})]\!]_0^H \boxplus [\![s_1^t(\mathrm{y})]\!]_0^H$



Convert $[\![s_0^t(\mathrm{y})]\!]_0^H$ to $[\![s_0^t(\mathrm{y})]\!]_1^H$

Convert $[\![s_1^t(\mathrm{y})]\!]_1^H$ to $[\![s_1^t(\mathrm{y})]\!]_0^H$

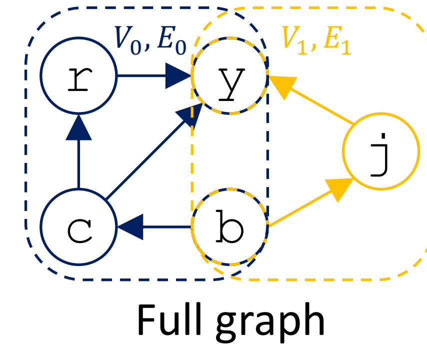
$[\![w^t(\mathrm{y})]\!]_1^H = [\![s_0^t(\mathrm{y})]\!]_1^H \boxplus [\![s_1^t(\mathrm{y})]\!]_1^H$



Cross-party interchange (CPI)

Protect Weights with Mixed Primitives

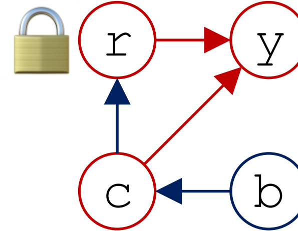
- Mix is necessary!
  - SS only? Would introduce per-edge communication (cannot operate locally)
  - HE only? Cannot operate cooperatively



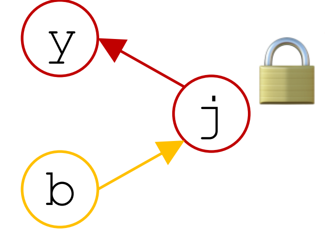
Full graph

- Solves weight privacy
  - $\Theta(|V|)$ communication per iteration
  - Still need to address obliviousness

$[\![s_0^t(\text{y})]\!]_0^H = [\![w^{t-1}(\text{r})]\!]_0^H \boxplus [\![w^{t-1}(\text{c})]\!]_0^H$

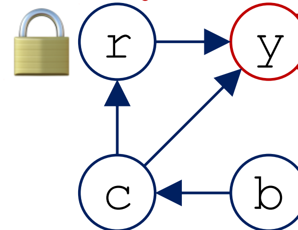


$[\![s_1^t(\text{y})]\!]_1^H = [\![w^{t-1}(\text{j})]\!]_1^H$



Local message passing (LMP)

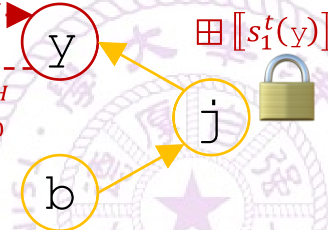
$[\![w^t(\text{y})]\!]_0^H = [\![s_0^t(\text{y})]\!]_0^H \boxplus [\![s_1^t(\text{y})]\!]_0^H$



Convert $[\![s_0^t(\text{y})]\!]_0^H$ to $[\![s_0^t(\text{y})]\!]_1^H$

Convert $[\![s_1^t(\text{y})]\!]_1^H$ to $[\![s_1^t(\text{y})]\!]_0^H$

$[\![w^t(\text{y})]\!]_1^H = [\![s_0^t(\text{y})]\!]_1^H \boxplus [\![s_1^t(\text{y})]\!]_1^H$

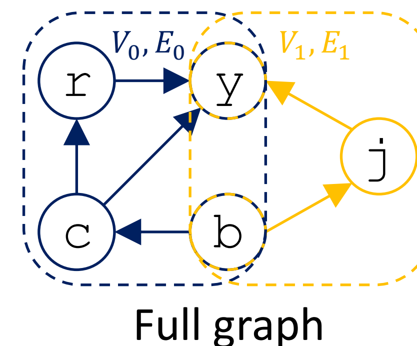


Cross-party interchange (CPI)

Achieve Obliviousness with ChaosTable

- Assign a random unique <u>position</u> for each vertex
  - Vertex in $V_0 \cap V_1$ has the same position as received by each party
  - How to assign? Introduced later

P_0 receives
b : 8
c : 5
r : 7
y : 2



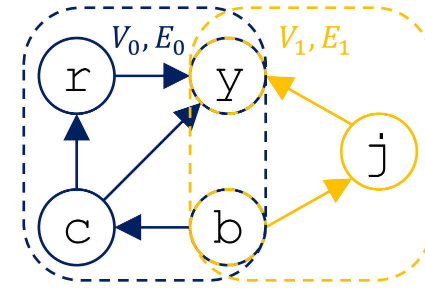
Full graph

P_1 receives
b : 8
j : 3
y : 2

Achieve Obliviousness with ChaosTable

- Assign a random unique position for each vertex
  - Vertex in $V_0 \cap V_1$ has the same position as received by each party
  - How to assign? Introduced later
- Operations reference vertices by positions, rather than name
  - Cross-party step works exhaustively for all possible positions, hiding the vertex set
  - $\Theta(|V|)$ communication, as long as the range of positions is $\Theta(|V|)$

P_0 receives
b : 8
c : 5
r : 7
y : 2



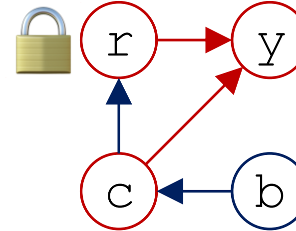
P_1 receives
b : 8
j : 3
y : 2

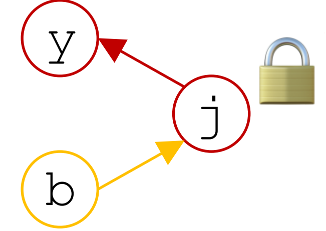
Full graph

y = 2, r = 7, c = 5

$[\![s_0^t(2)]\!]_0^H = [\![w^{t-1}(7)]\!]_0^H \boxplus [\![w^{t-1}(5)]\!]_0^H$

$[\![s_1^t(2)]\!]_1^H = [\![w^{t-1}(3)]\!]_1^H$



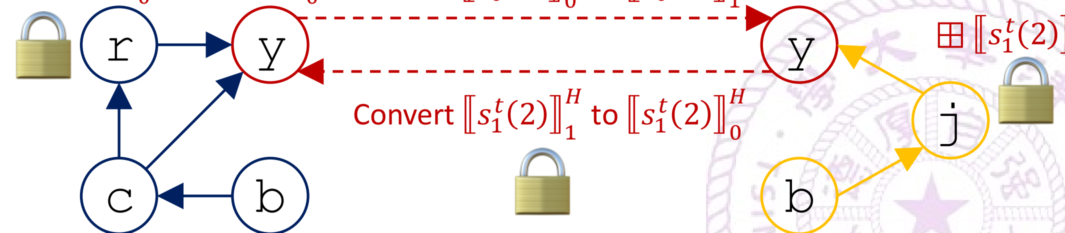


Local message passing (LMP)

$[\![w^t(2)]\!]_0^H = [\![s_0^t(2)]\!]_0^H \boxplus [\![s_1^t(2)]\!]_0^H$

Convert $[\![s_0^t(2)]\!]_0^H$ to $[\![s_0^t(2)]\!]_1^H$

$[\![w^t(2)]\!]_1^H = [\![s_0^t(2)]\!]_1^H \boxplus [\![s_1^t(2)]\!]_1^H$



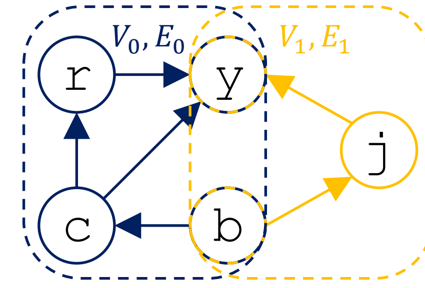
Convert $[\![s_1^t(2)]\!]_1^H$ to $[\![s_1^t(2)]\!]_0^H$

Cross-party interchange (CPI)

Achieve Obliviousness with ChaosTable

- Assign a random unique <u>position</u> for each vertex
  - Vertex in $V_0 \cap V_1$ has the same position as received by each party
  - How to assign? Introduced later
- Operations reference vertices by positions, rather than name
  - Cross-party step works exhaustively for all possible positions, hiding the vertex set
  - $\Theta(|V|)$ communication, as long as the range of positions is $\Theta(|V|)$
- Solves obliviousness!

P_0 receives
b : 8
c : 5
r : 7
y : 2

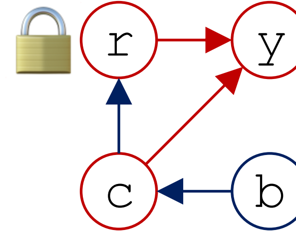


Full graph

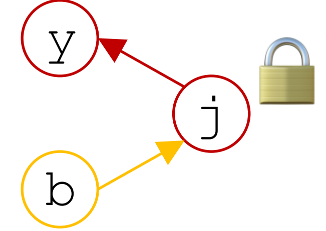
P_1 receives
b : 8
j : 3
y : 2

y = 2, r = 7, c = 5

$[\![s_0^t(2)]\!]_0^H = [\![w^{t-1}(7)]\!]_0^H \boxplus [\![w^{t-1}(5)]\!]_0^H$



$[\![s_1^t(2)]\!]_1^H = [\![w^{t-1}(3)]\!]_1^H$

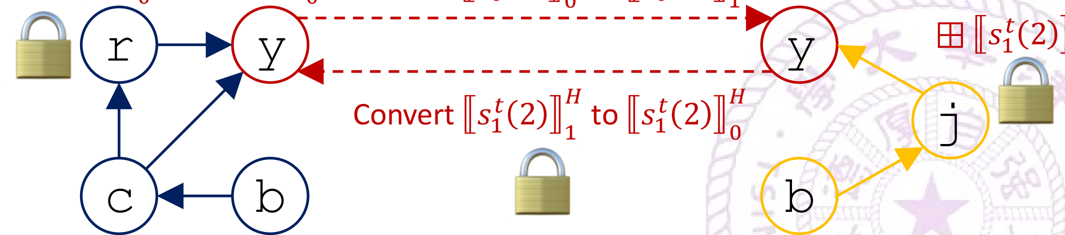


Local message passing (LMP)

$[\![w^t(2)]\!]_0^H = [\![s_0^t(2)]\!]_0^H \boxplus [\![s_1^t(2)]\!]_0^H$

Convert $[\![s_0^t(2)]\!]_0^H$ to $[\![s_0^t(2)]\!]_1^H$

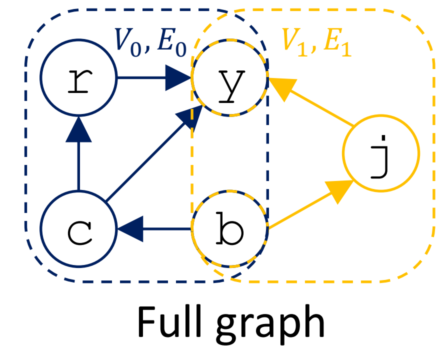
$[\![w^t(2)]\!]_1^H = [\![s_0^t(2)]\!]_1^H \boxplus [\![s_1^t(2)]\!]_1^H$

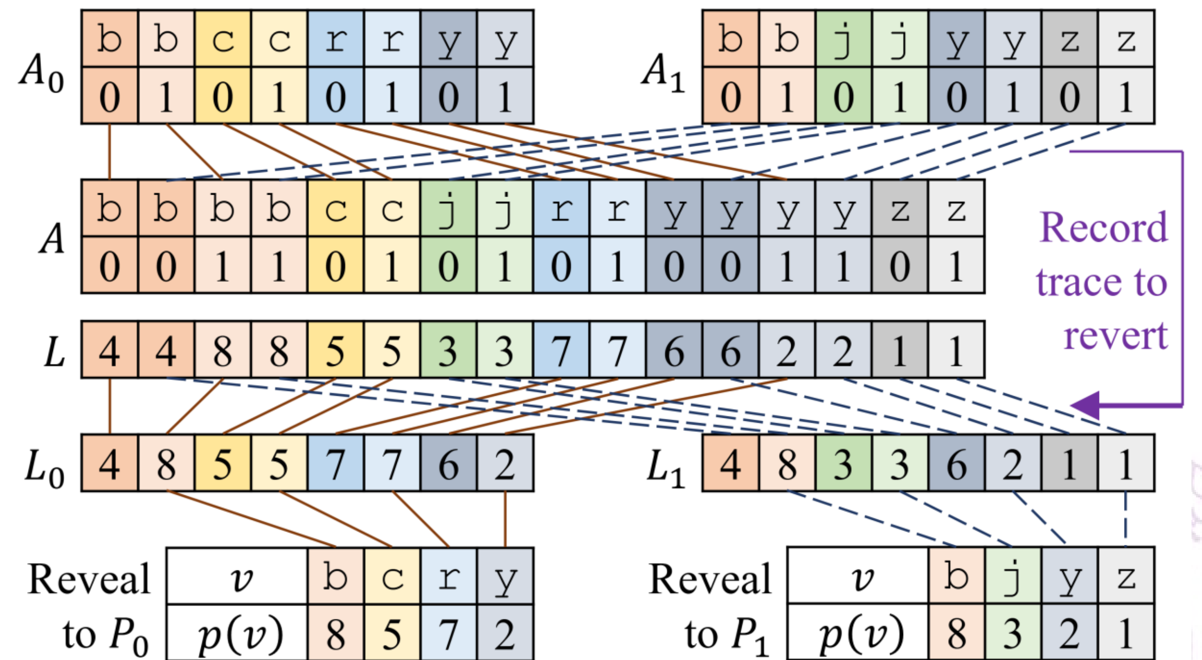


Cross-party interchange (CPI)

Building ChaosTable

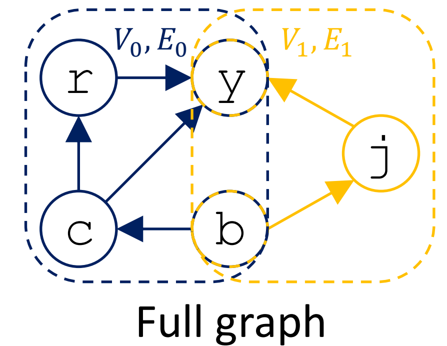
- Each party prepares an array (A_0, A_1)
  - For each vertex v, add $(v, 0)$ and $(v, 1)$ to the array
  - Add padding (z here) so its length becomes power-of-2

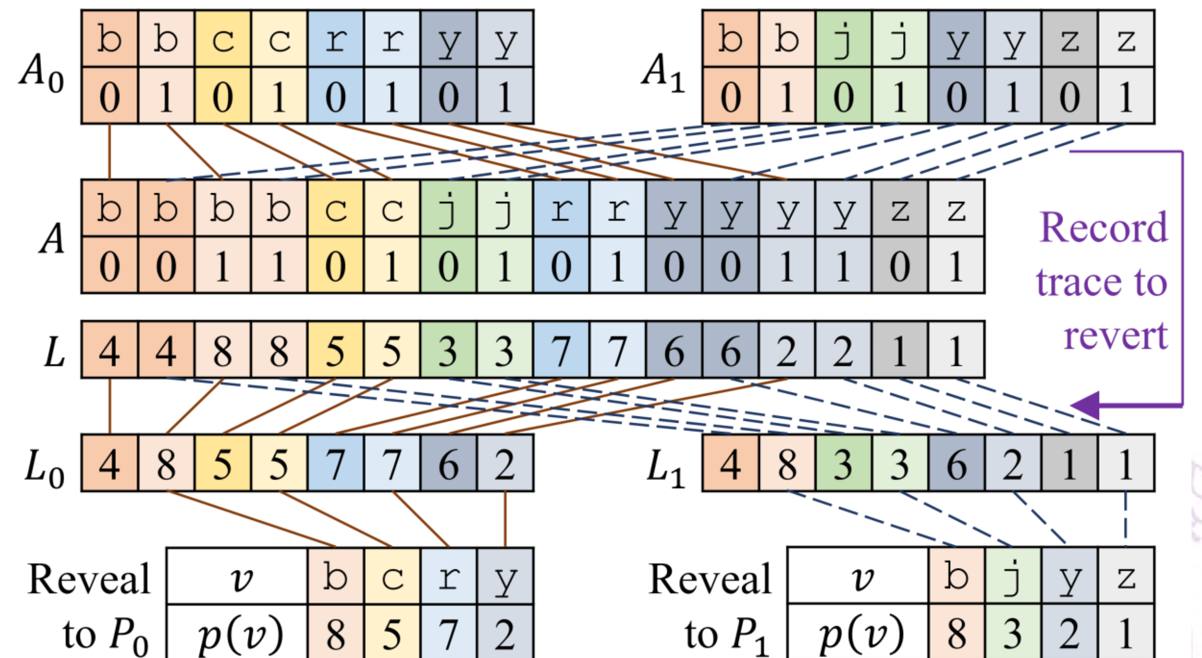




Building ChaosTable

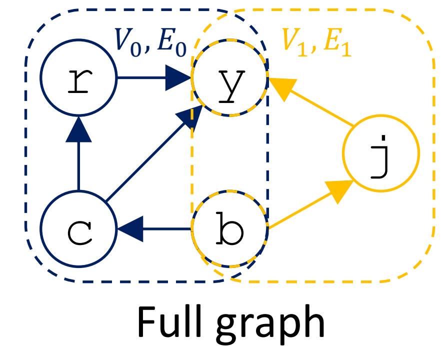
- Each party prepares an array (A_0, A_1)
  - For each vertex v, add $(v, 0)$ and $(v, 1)$ to the array
  - Add padding (z here) so its length becomes power-of-2
- Merge two sorted arrays (A)

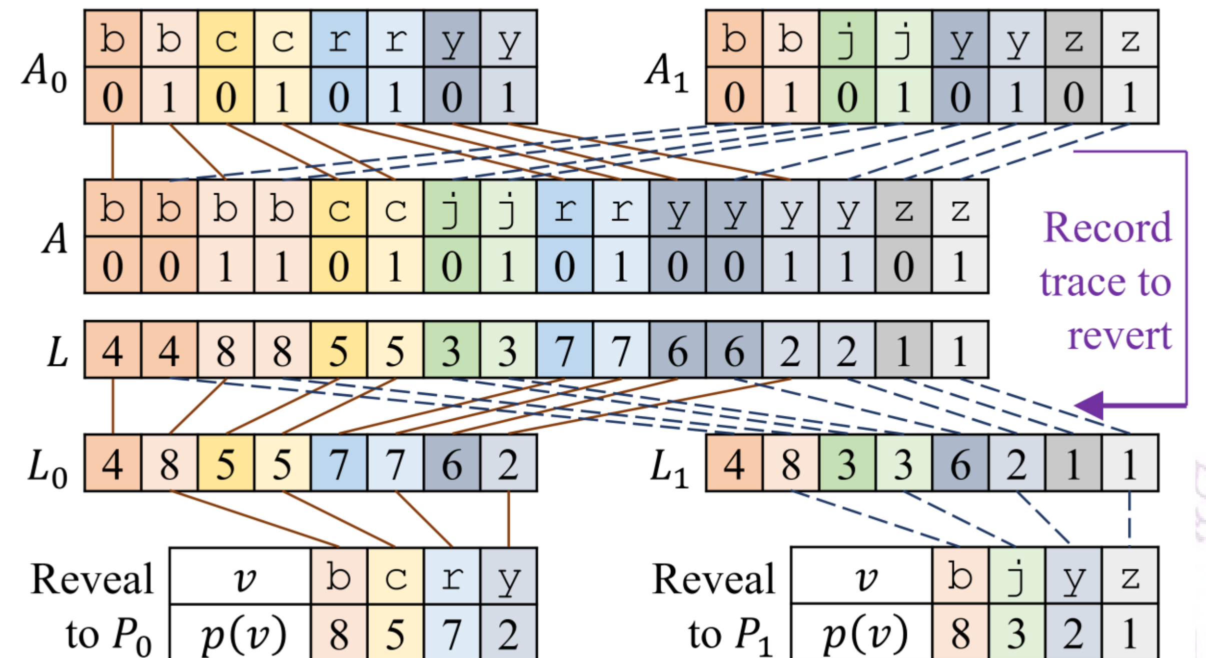




Building ChaosTable

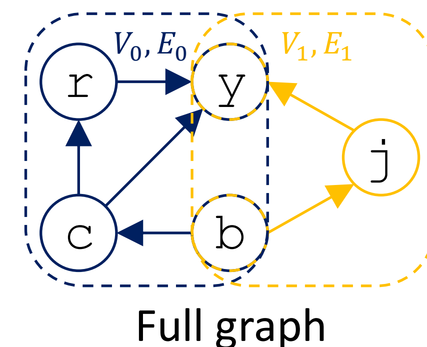
- Each party prepares an array (A_0, A_1)
  - For each vertex v, add $(v, 0)$ and $(v, 1)$ to the array
  - Add padding (z here) so its length becomes power-of-2
- Merge two sorted arrays (A)
- Generate a random array (L)
  - Every two elements are equal



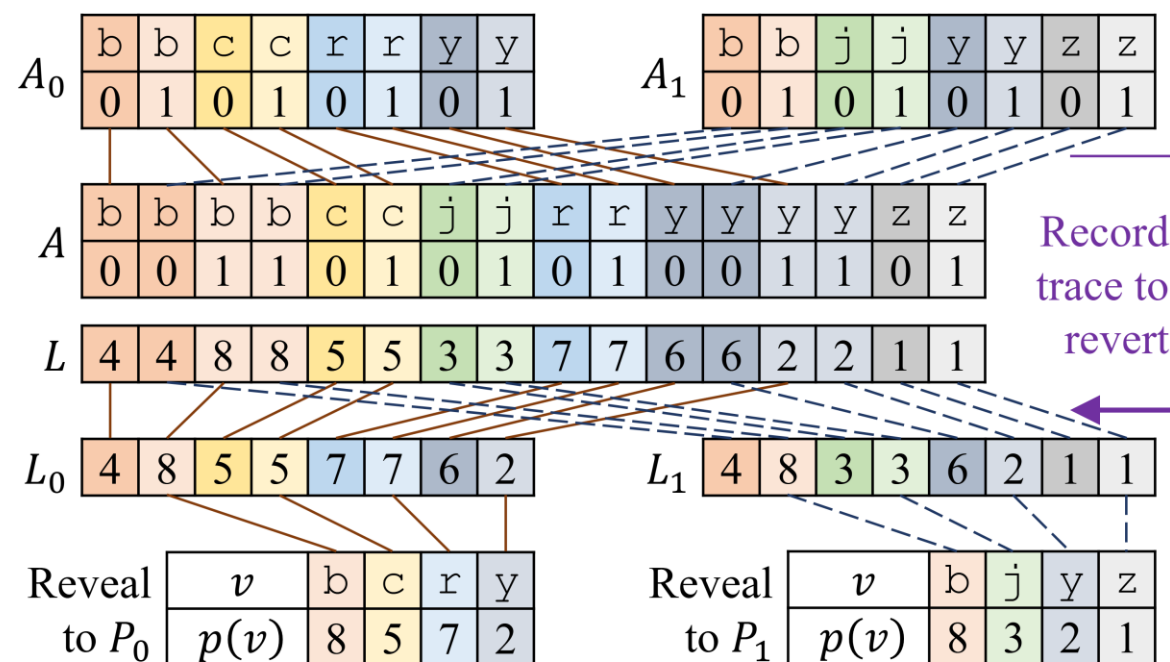


Building ChaosTable

- Each party prepares an array (A_0, A_1)
  - For each vertex v, add $(v, 0)$ and $(v, 1)$ to the array
  - Add padding (z here) so its length becomes power-of-2
- Merge two sorted arrays (A)
- Generate a random array (L)
  - Every two elements are equal
- Revert the merge (L_0, L_1)
  - i.e., $\exists\pi: \pi(L_0||L_1) = L, \pi(A_0||A_1) = A$

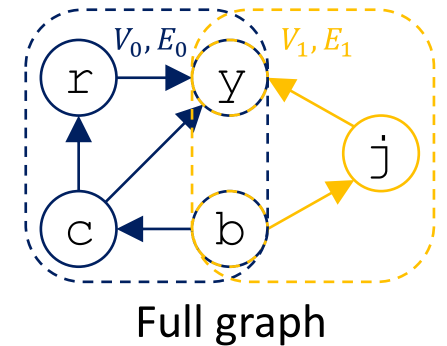


Full graph

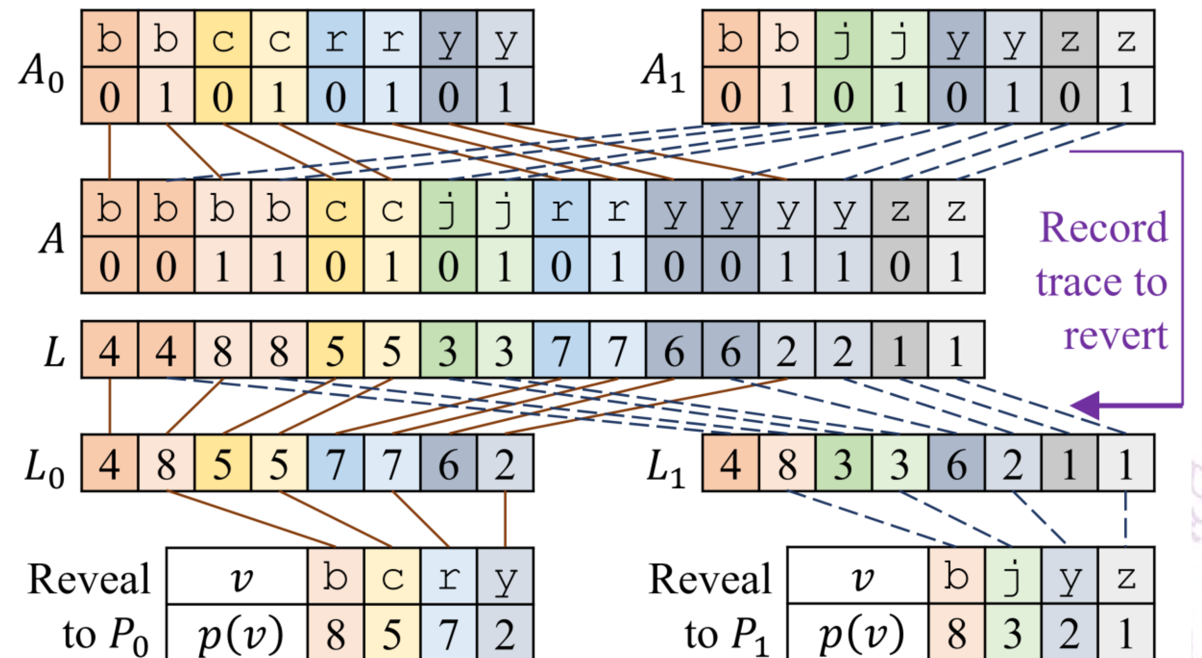


Building ChaosTable

- Each party prepares an array (A_0, A_1)
  - For each vertex v, add $(v, 0)$ and $(v, 1)$ to the array
  - Add padding (z here) so its length becomes power-of-2
- Merge two sorted arrays (A)
- Generate a random array (L)
  - Every two elements are equal
- Revert the merge (L_0, L_1)
  - i.e., $\exists\pi: \pi(L_0||L_1) = L, \pi(A_0||A_1) = A$
- Reveal even-indexed elements of L_b to Party b



Full graph



A_0	b	b	c	c	r	r	y	y
	0	1	0	1	0	1	0	1

A_1	b	b	j	j	y	y	z	z
	0	1	0	1	0	1	0	1

A	b	b	b	b	c	c	j	j	r	r	y	y	y	y	z	z
	0	0	1	1	0	1	0	1	0	1	0	0	1	1	0	1

L	4	4	8	8	5	5	3	3	7	7	6	6	2	2	1	1

L_0	4	8	5	5	7	7	6	2

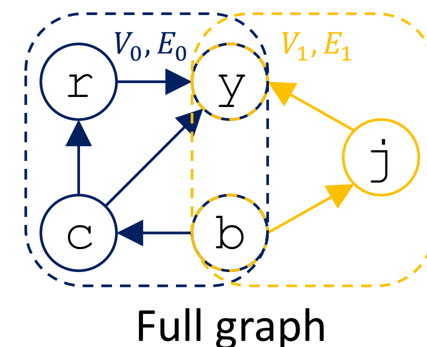
L_1	4	8	3	3	6	2	1	1

Reveal to P_0				
v	b	c	r	y
$p(v)$	8	5	7	2

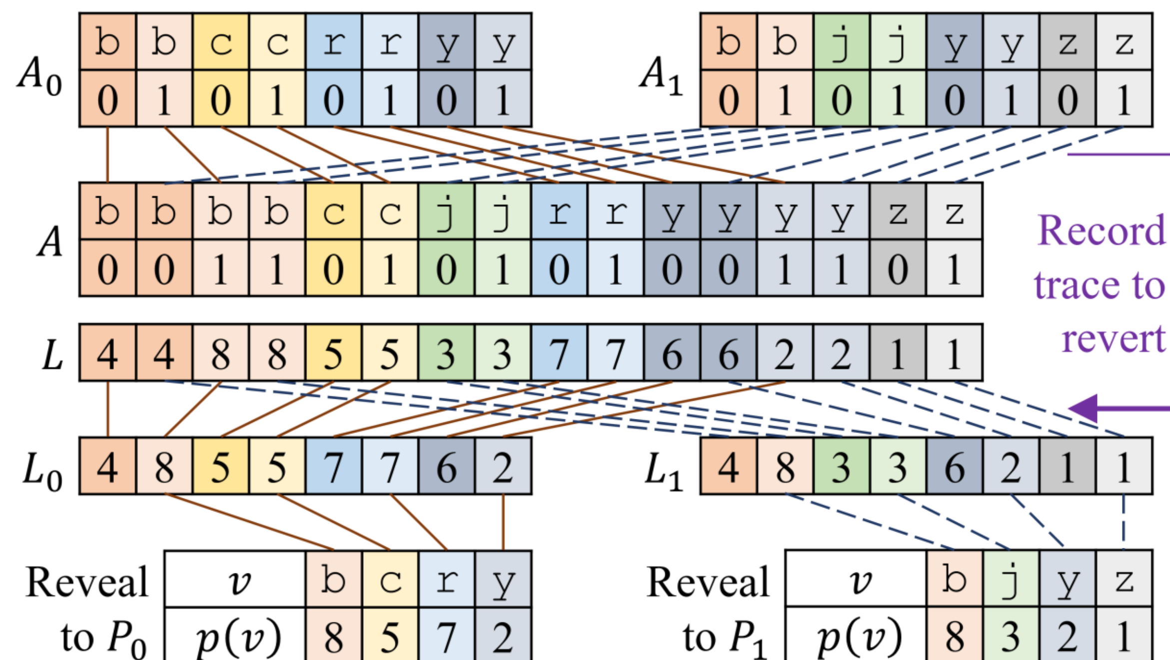
Reveal to P_1				
v	b	j	y	z
$p(v)$	8	3	2	1

Building ChaosTable

- Each party prepares an array (A_0, A_1)
  - For each vertex v, add $(v, 0)$ and $(v, 1)$ to the array
  - Add padding (z here) so its length becomes power-of-2
- Merge two sorted arrays (A)
- Generate a random array (L)
  - Every two elements are equal
- Revert the merge (L_0, L_1)
  - i.e., $\exists\pi: \pi(L_0||L_1) = L, \pi(A_0||A_1) = A$
- Reveal even-indexed elements of L_b to Party b
- All steps use secure protocols
  - Based on secret-sharing

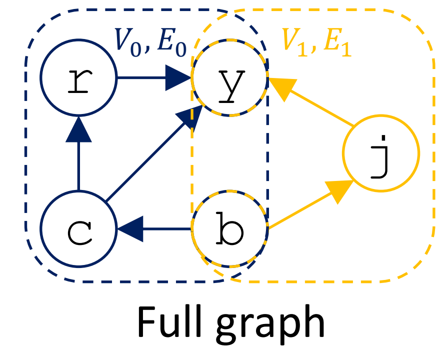


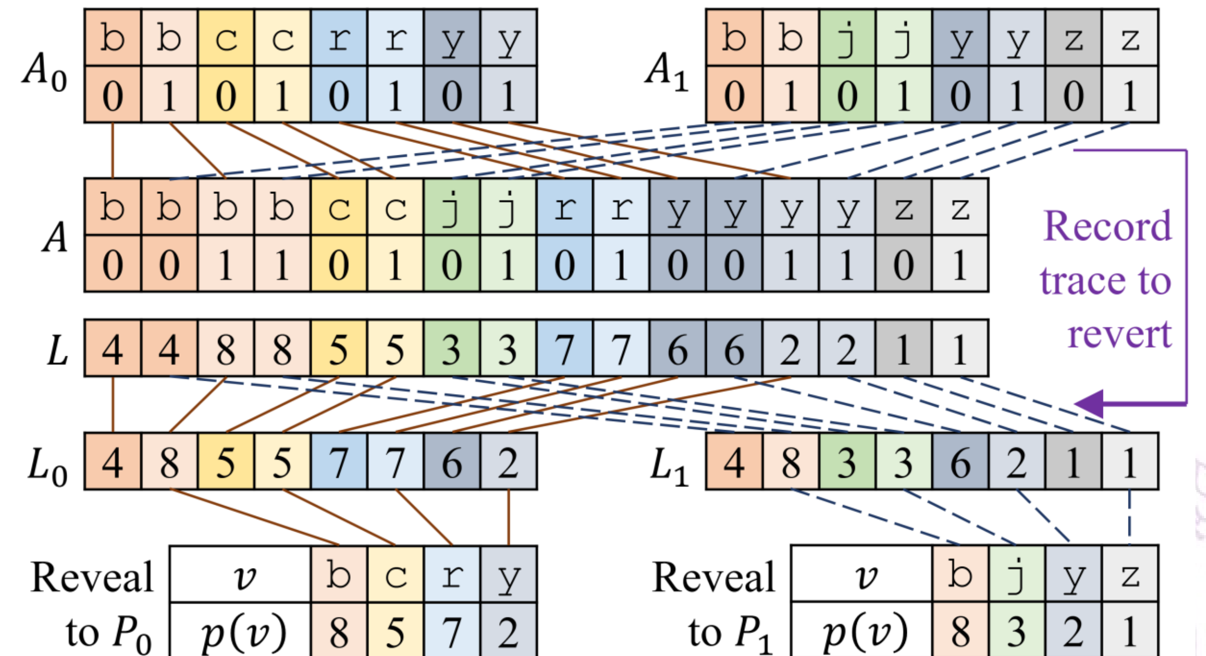
Full graph



Building ChaosTable

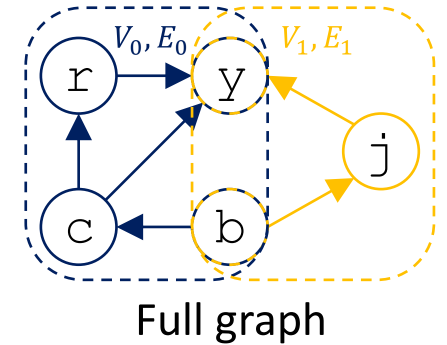
- Correctness
  - Different vertices receive different positions
  - An vertex known by both parties receives a unique position

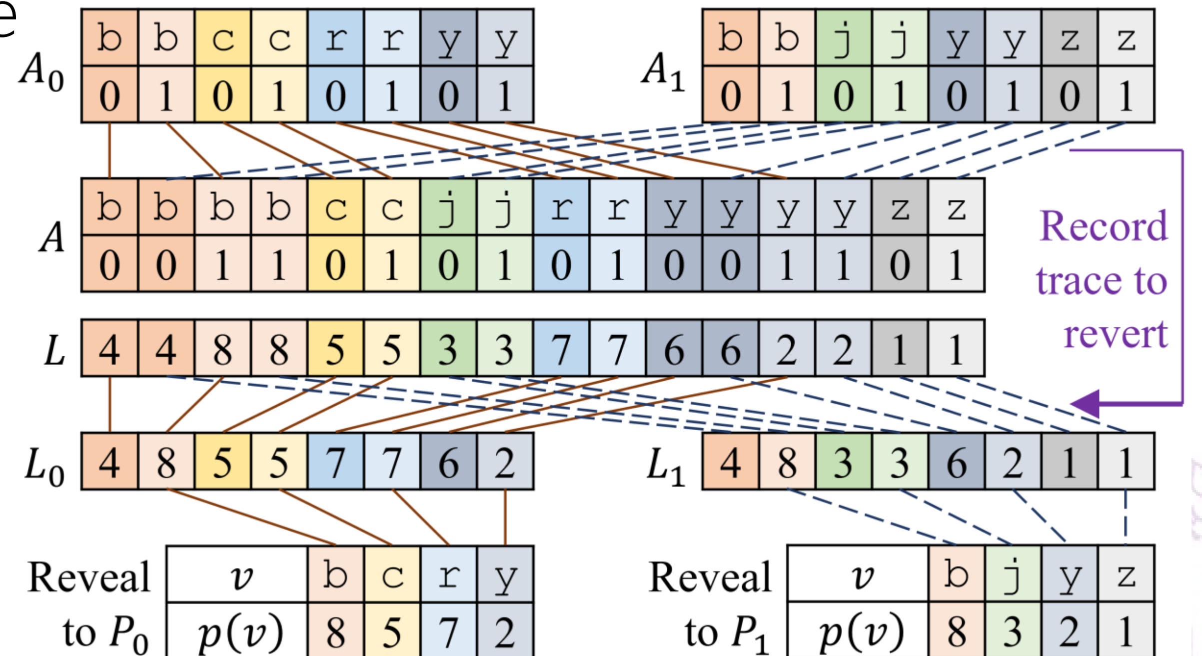




Building ChaosTable

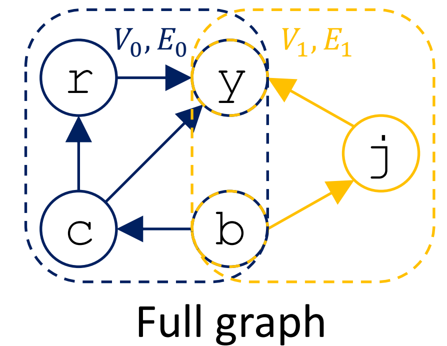
- Correctness
  - Different vertices receive different positions
  - An vertex known by both parties receives a unique position
- Randomness
  - Each party's received results resemble random sampling (w/o replacement)

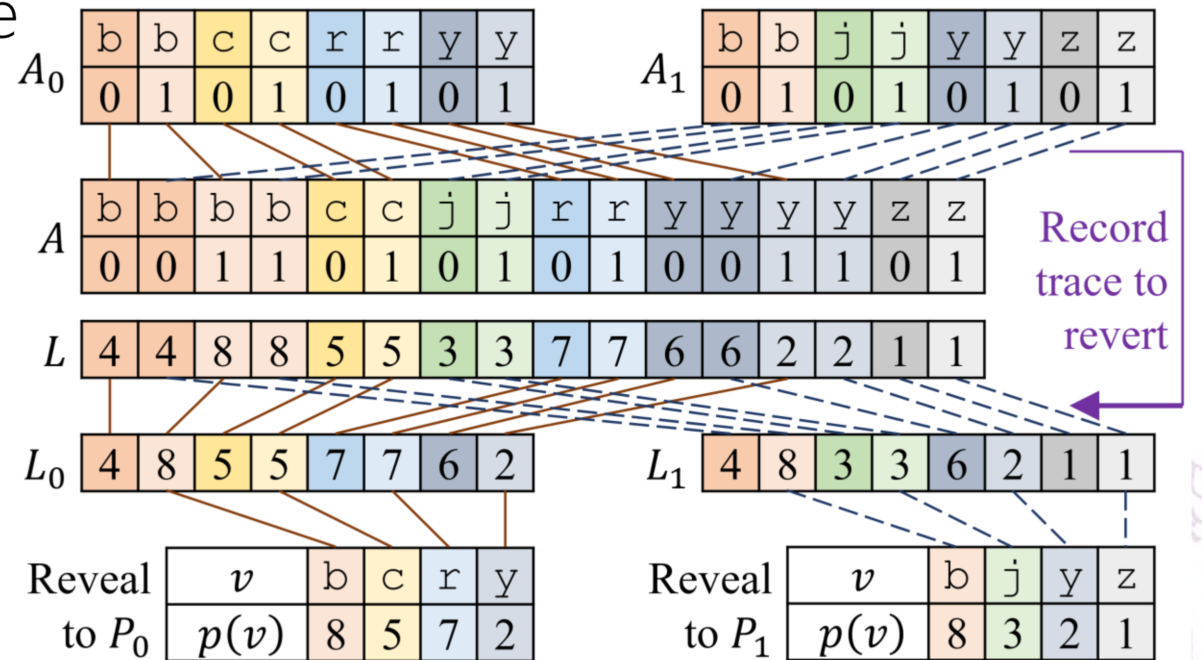




Building ChaosTable

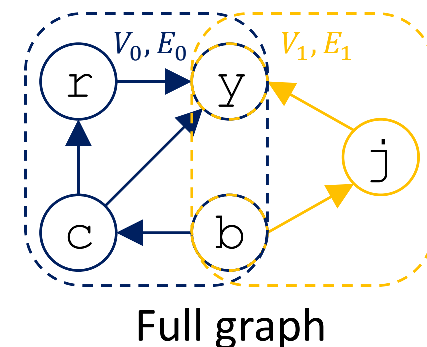
- Correctness
  - Different vertices receive different positions
  - An vertex known by both parties receives a unique position
- Randomness
  - Each party's received results resemble random sampling (w/o replacement)
- Linearity
  - Range of results is $\Theta(|V|)$
  - Critical for ensuring GraphAce's communication complexity!

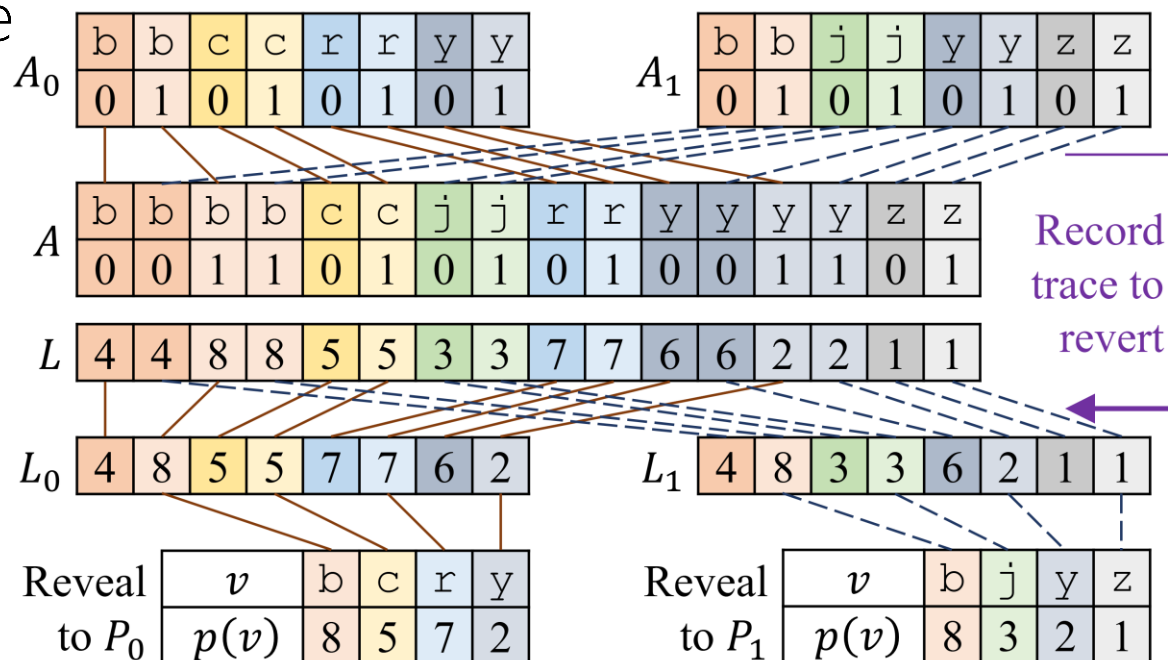




Building ChaosTable

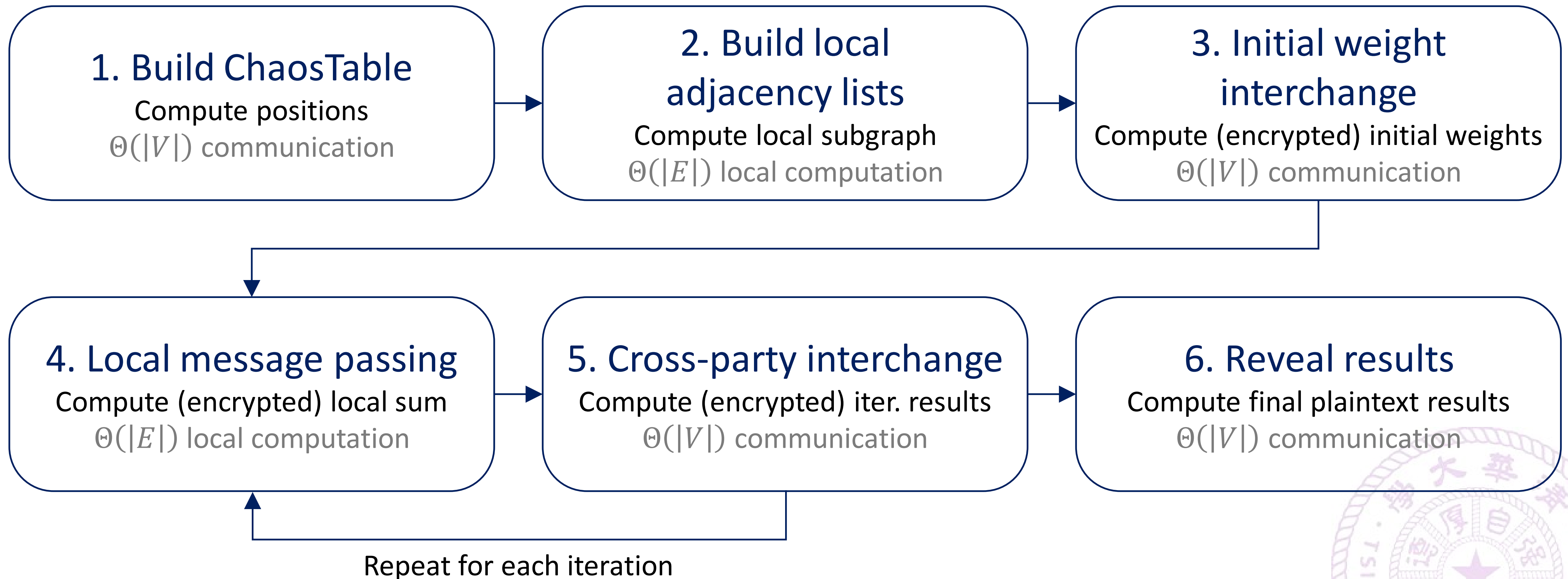
- Correctness
  - Different vertices receive different positions
  - An vertex known by both parties receives a unique position
- Randomness
  - Each party's received results resemble random sampling (w/o replacement)
- Linearity
  - Range of results is $\Theta(|V|)$
  - Critical for ensuring GraphAce's communication complexity!
- Efficiency
  - Need $\Theta(|V|)$ time to build





Full Workflow

- (Generalizing the previous example algorithm)



Security (Outline)

- In GraphAce, any message sent over network is either ciphertext, or (independent) random numbers from known distribution

Security (Outline)

- In GraphAce, any message sent over network is either ciphertext, or (independent) random numbers from known distribution

- Thus, we can prove the security of GraphAce against (static) semi-honest adversaries, with a simulation-based method

Evaluation

Experimental Setup

- **Mainly compare with GraphSC**
  - The only existing 2-party solution

Experimental Setup

- Mainly compare with GraphSC
  - The only existing 2-party solution
- Workload
  - 5 graphs: randomly generated, of various scales
  - 5 applications: PageRank (PR), Sparse Matrix-Vector multiplication (SpMV), Semi-Supervised Label Propagation (SSLP), implemented with Paillier HE, Weakly Connect Components (WCC), Multiple-Source Shortest Path (MSSP), implemented with TFHE (due to complex operations required in HE)

Graph	$\|V_0\|, \|V_1\|$	$\|V\|$	$\|E\|$
Scale 14	128	192	16,192
Scale 17	1,024	1,536	129,536
Scale 20	8,192	12,288	1,036,288
Scale 24	131,072	196,608	16,580,608
Scale 27	1,048,576	1,572,864	132,644,864

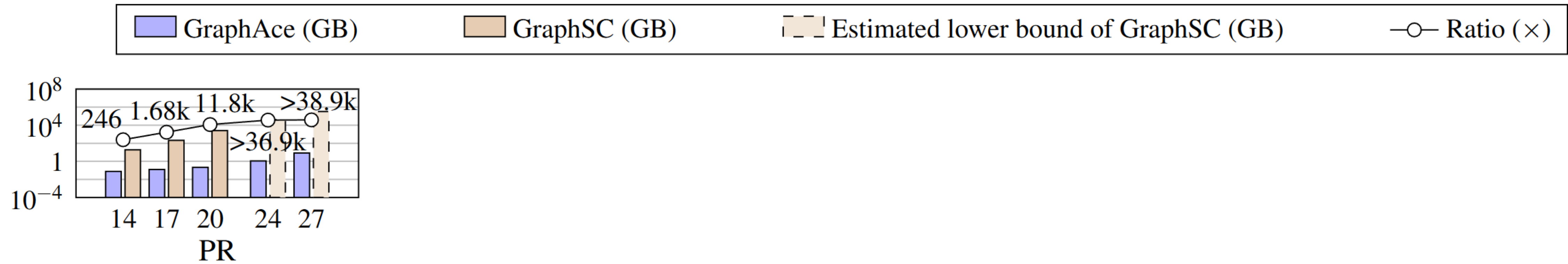
Experimental Setup

- Mainly compare with GraphSC
  - The only existing 2-party solution
- Workload
  - 5 graphs: randomly generated, of various scales
  - 5 applications: PageRank (PR), Sparse Matrix-Vector multiplication (SpMV), Semi-Supervised Label Propagation (SSLP), implemented with Paillier HE, Weakly Connect Components (WCC), Multiple-Source Shortest Path (MSSP), implemented with TFHE (due to complex operations required in HE)

Graph	$\lvert V_0\rvert, \lvert V_1\rvert$	$\lvert V\rvert$	$\lvert E\rvert$
Scale 14	128	192	16,192
Scale 17	1,024	1,536	129,536
Scale 20	8,192	12,288	1,036,288
Scale 24	131,072	196,608	16,580,608
Scale 27	1,048,576	1,572,864	132,644,864

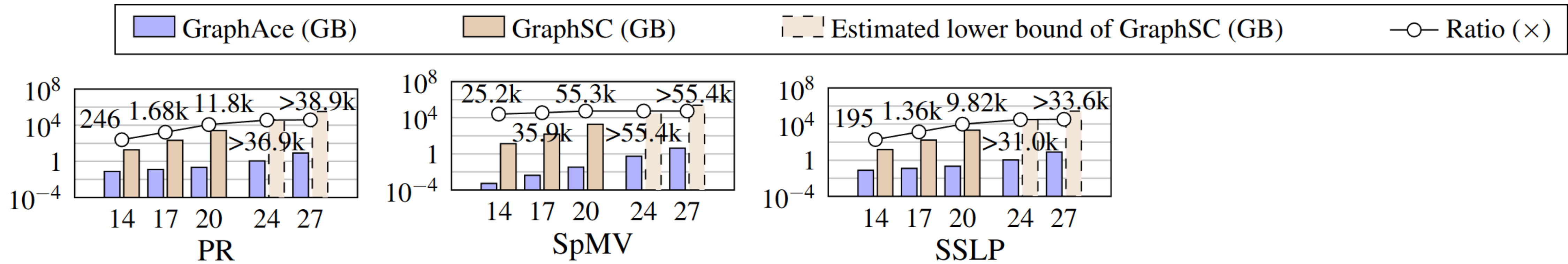
- Environment
  - CPU: 40 cores per party
  - 3 network settings: single datacenter (1-DC, 1000 Mbps, 2 ms latency), inter-datacenter (Inter-DC, 100 Mbps, 20 ms), inter-continent (10 Mbps, 200 ms)

Communication Efficiency



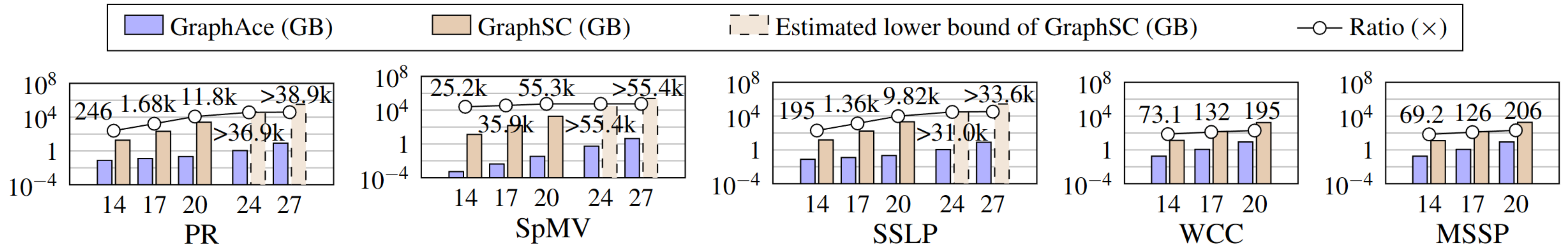
- Better saving ratio for larger graphs
  - GraphAce offers better complexity (linear vs additional log-factors)

Communication Efficiency



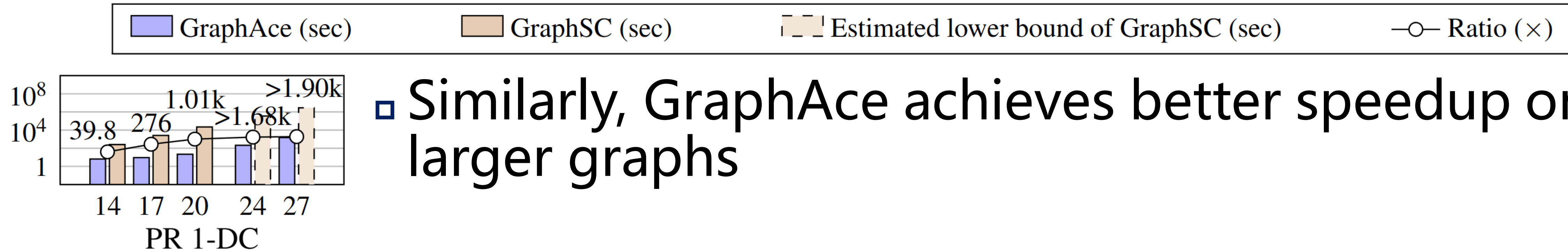
- Better saving ratio for larger graphs
  - GraphAce offers better complexity (linear vs additional log-factors)
- >10k× for PR / SpMV / SSLP
  - Dashed bar: estimated from linear extrapolation because GraphSC did not finish in 24 hours

Communication Efficiency



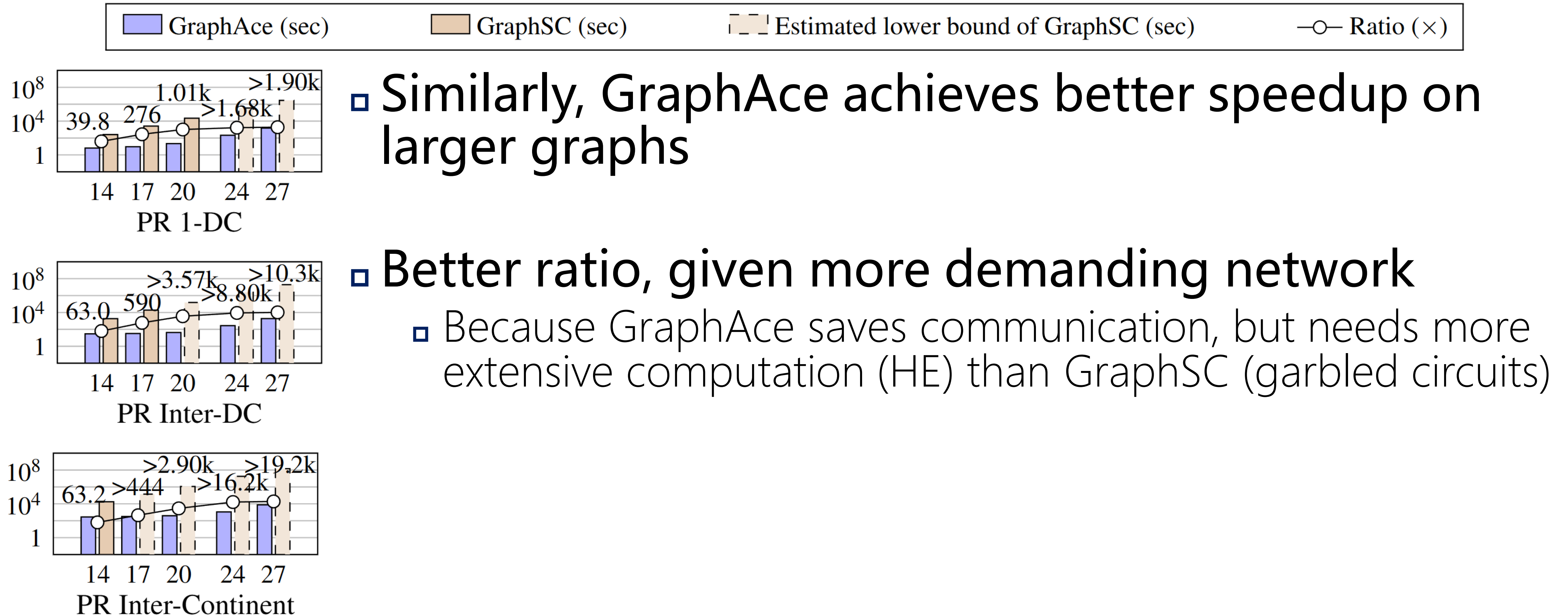
- Better saving ratio for larger graphs
  - GraphAce offers better complexity (linear vs additional log-factors)
- >10k× for PR / SpMV / SSLP
  - Dashed bar: estimated from linear extrapolation because GraphSC did not finish in 24 hours
- ~200× for WCC / MSSP
  - FHE needs larger ciphertext, increasing GraphAce's constant factor

Speedup



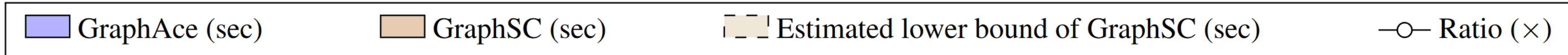
- Similarly, GraphAce achieves better speedup on larger graphs

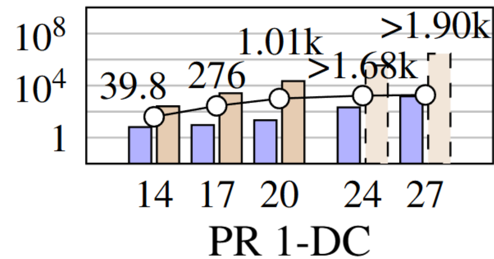
Speedup

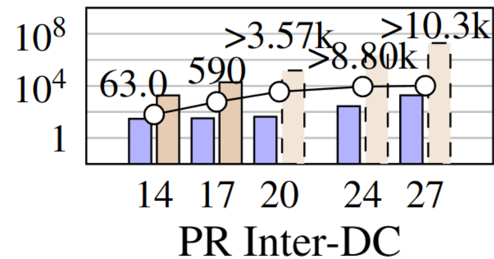


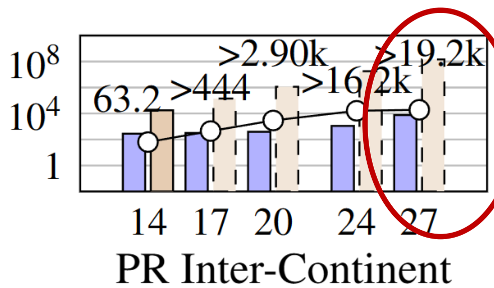
- Similarly, GraphAce achieves better speedup on larger graphs
- Better ratio, given more demanding network
  - Because GraphAce saves communication, but needs more extensive computation (HE) than GraphSC (garbled circuits)

Speedup



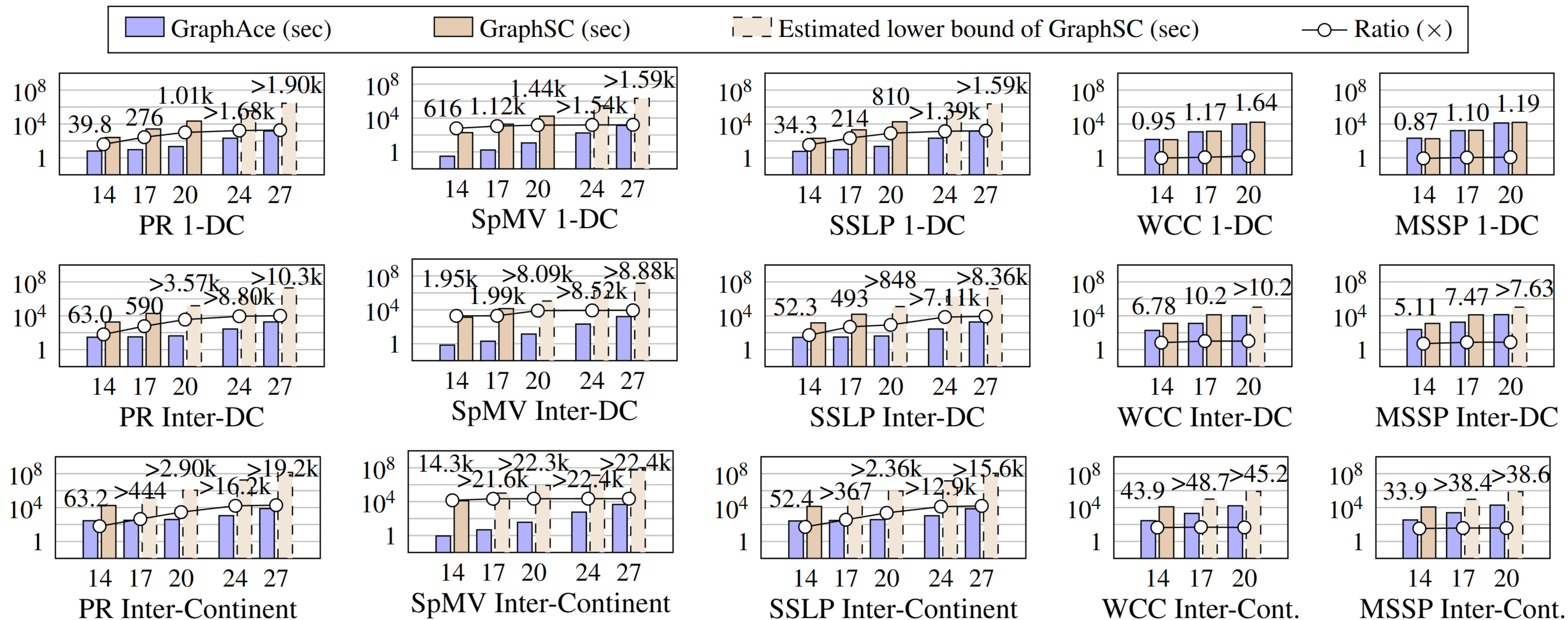






- Similarly, GraphAce achieves better speedup on larger graphs
- Better ratio, given more demanding network
  - Because GraphAce saves communication, but needs more extensive computation (HE) than GraphSC (garbled circuits)
- Example: PR Inter-Continent Scale 27
  - GraphSC would require >4.5 years per iteration
  - GraphAce only needs 2.1 hours

Speedup



- Similarly, better speedup for PR / SpMV / SSLP, than WCC / MSSP

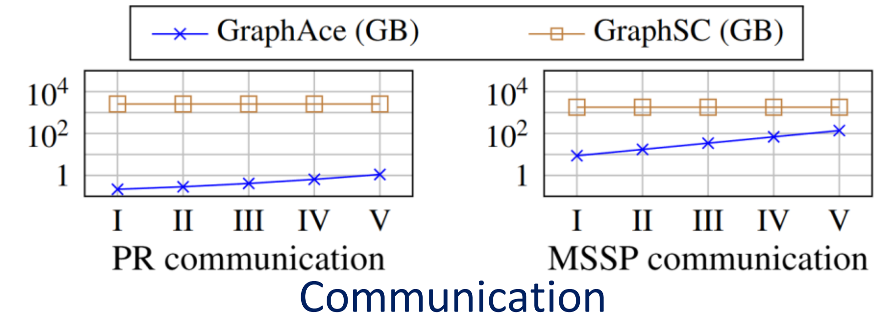
Influences of Average Degree

Graph	$\|V_0\|$, $\|V_1\|$	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
I	8,192	12,288	1,036,288	84.33
II	16,384	24,576	1,024,000	41.67
III	32,768	49,152	999,424	20.33
IV	65,536	98,304	950,272	9.67
V	131,072	196,608	851,968	4.33

- 5 graphs of the same $|V| + |E|$, different $|E|/|V|$
  - Ranging from 4.33 to 84.33, covering most practical cases
  - GraphSC remains the same performance
  - GraphAce needs more work for larger $|V|$

Influences of Average Degree

Graph	$\lvert V_0\rvert$, $\lvert V_1\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E\rvert/\lvert V\rvert$
I	8,192	12,288	1,036,288	84.33
II	16,384	24,576	1,024,000	41.67
III	32,768	49,152	999,424	20.33
IV	65,536	98,304	950,272	9.67
V	131,072	196,608	851,968	4.33

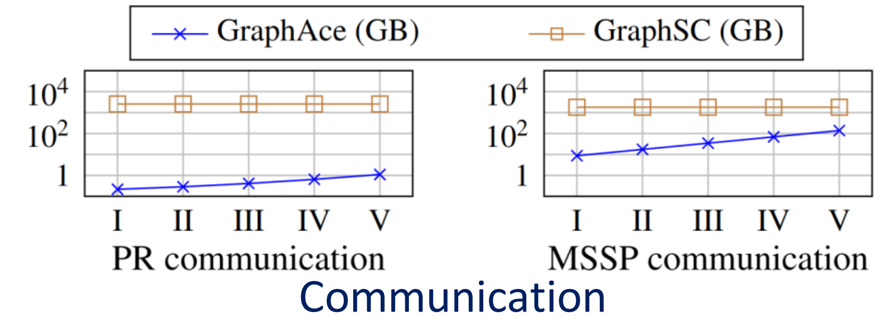


Communication

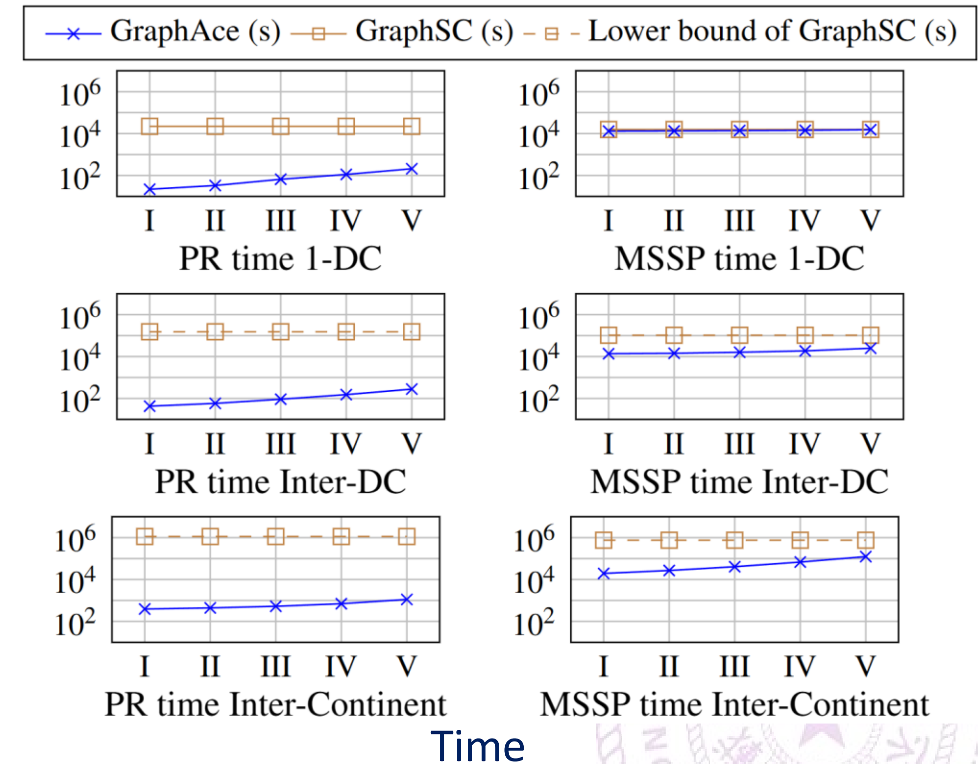
- 5 graphs of the same $|V| + |E|$, different $|E|/|V|$
  - Ranging from 4.33 to 84.33, covering most practical cases
  - GraphSC remains the same performance
  - GraphAce needs more work for larger $|V|$
- GraphAce consistently saves communication

Influences of Average Degree

Graph	$\lvert V_0\rvert, \lvert V_1\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E\rvert/\lvert V\rvert$
I	8,192	12,288	1,036,288	84.33
II	16,384	24,576	1,024,000	41.67
III	32,768	49,152	999,424	20.33
IV	65,536	98,304	950,272	9.67
V	131,072	196,608	851,968	4.33



Communication



Time

- 5 graphs of the same $|V|+|E|$, different $|E|/|V|$
  - Ranging from 4.33 to 84.33, covering most practical cases
  - GraphSC remains the same performance
  - GraphAce needs more work for larger $|V|$
- GraphAce consistently saves communication and time

Artifacts

- Available at https://zenodo.org/records/15009743

```
# For Ubuntu system
sudo apt install docker.io xz-utils
wget -O graphace.tar.xz "https://zenodo.org/records/15009743/files/graphace.tar.xz"
cat graphace.tar.xz | xz -d | tar x
cd graphace
sudo docker build -t graphace .
sudo docker run --cap-add SYS_NICE --cap-add NET_ADMIN \
  -v ./results:/app/results -it graphace python3 run.py --preset toy
```

Artifacts

- Available at https://zenodo.org/records/15009743

```
# For Ubuntu system
sudo apt install docker.io xz-utils
wget -O graphace.tar.xz "https://zenodo.org/records/15009743/files/graphace.tar.xz"
cat graphace.tar.xz | xz -d | tar x
cd graphace
sudo docker build -t graphace .
sudo docker run --cap-add SYS_NICE --cap-add NET_ADMIN \
  -v ./results:/app/results -it graphace python3 run.py --preset toy
```

- See our Artifacts Appendix for further information

Thank you!