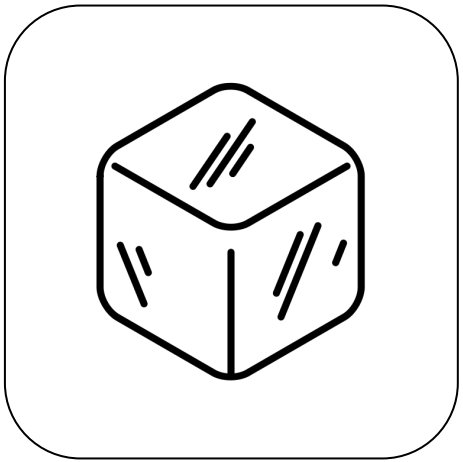
Privacy-Preserving Joins in Untrusted Environments

Kajetan Maliszewski
PhD Defense
4. April 2025

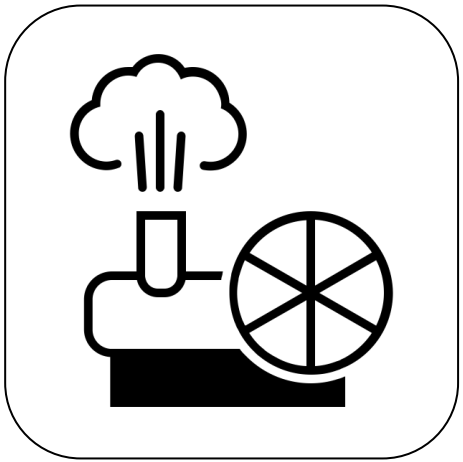
Three states of data matter

At-rest

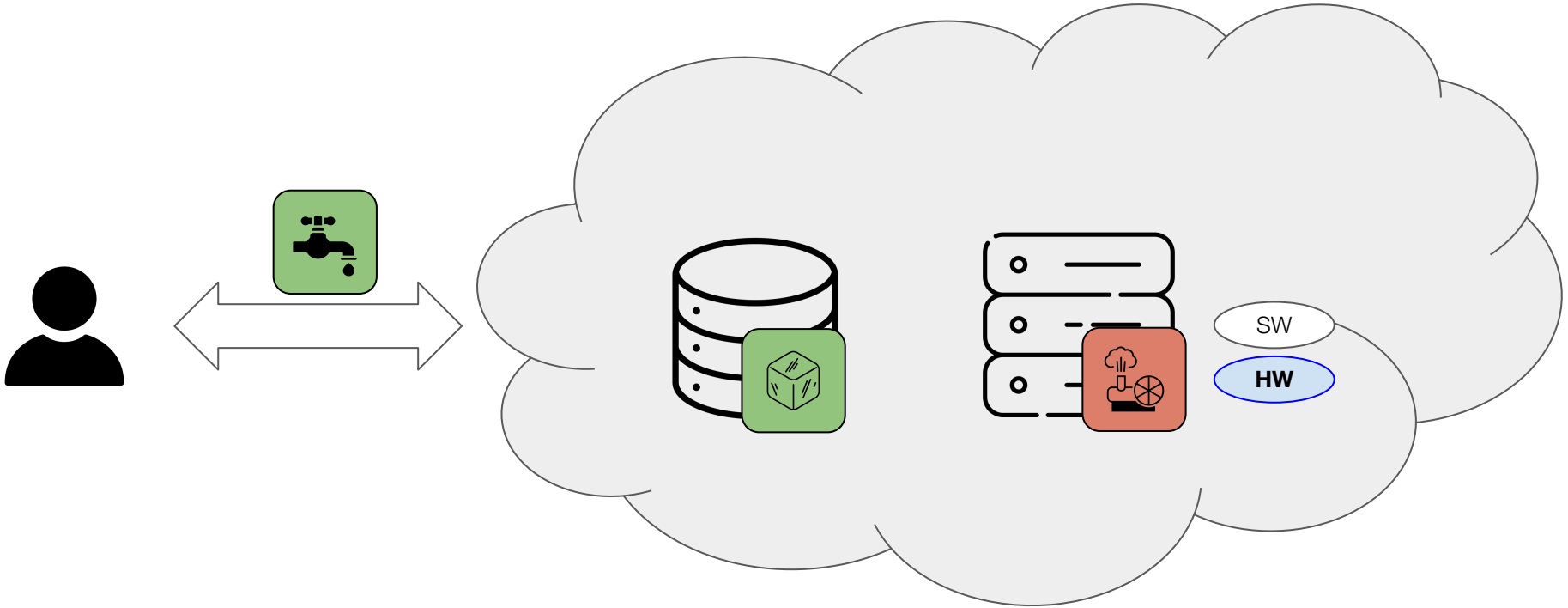


In-transit

In-use

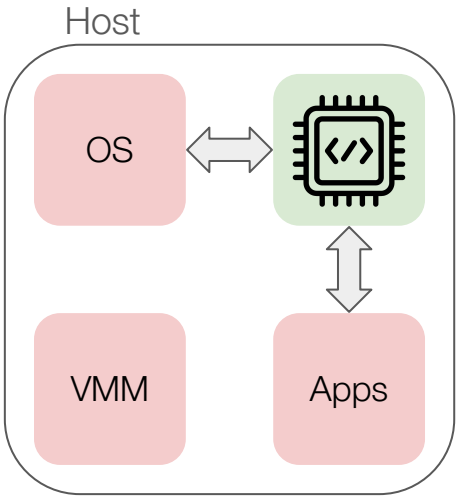


Three states of customer’s sensitive data in public cloud



Trusted Execution Environments (TEEs)

New hardware design:

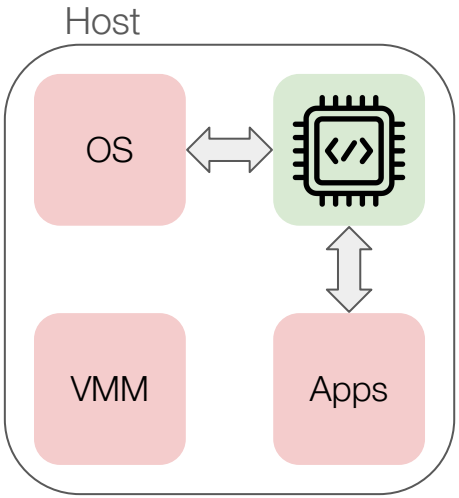


Advantages over other PETs:

- Performance
- Enhanced functionality
- Ease of deployment

Adoption of TEEs has been a bumpy road

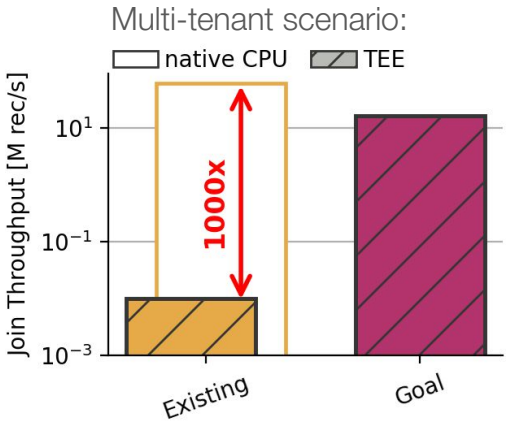
Simple things get complex:



Disadvantages over plain CPU:

- Challenging hardware architecture
- Performance bottlenecks
- Security considerations

Joins in TEEs can significantly underperform



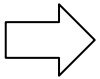
Thesis Goal

Analyze and improve the privacy-performance trade-off of TEEs in batch and stream join workloads

Contributions

Problem

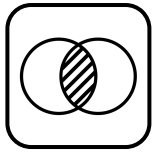
Adoption of TEEs



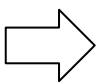
Solution

Equi-joins in TEEs

VLDB 2022

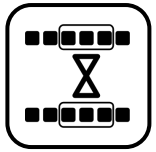


Efficient Query Processing in TEEs

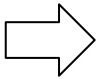


Cracking-Like Join

VLDB 2023



Privacy-Preserving Stream Processing



Privacy-Preserving Stream Joins

under submission

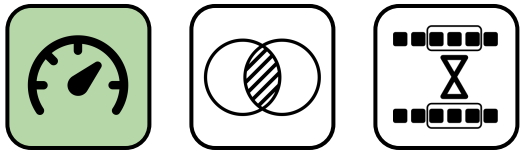
Honest-but-curious adversary

P1/P2
"operational" confidentiality in batch processing

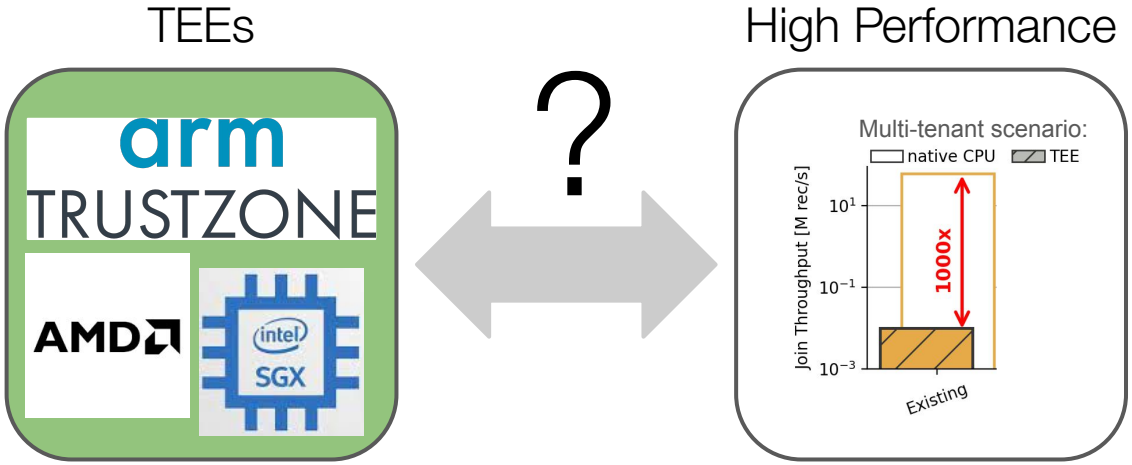
P3
+ side-channel protection in stream processing

Agenda

1. Motivation
2. **Understanding equi-joins in TEEs**
3. Relational Join for TEEs
4. Privacy-preserving stream joins
5. Conclusions



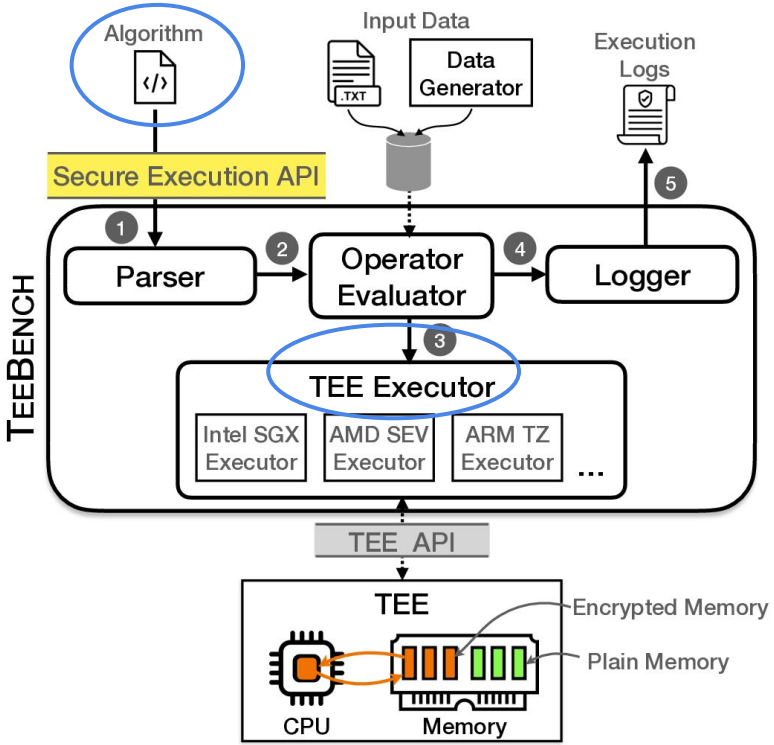
Performance of equi-joins in TEEs is an open challenge



We benchmarked four families of join algorithms

Family	Join Algorithm	
hash-based	CHT	Concise Hash Table
	PHT	Parallel Hash Table
sort-merge	PSM	Parallel Sort-Merge
	MWAY	Multi-Way Sort-Merge
radix-based	RHT	Radix Hash Table
	RHO	Radix Hash Optimized
	RSM	Radix Sort-Merge
nested-based	INL	Index Nested Loop

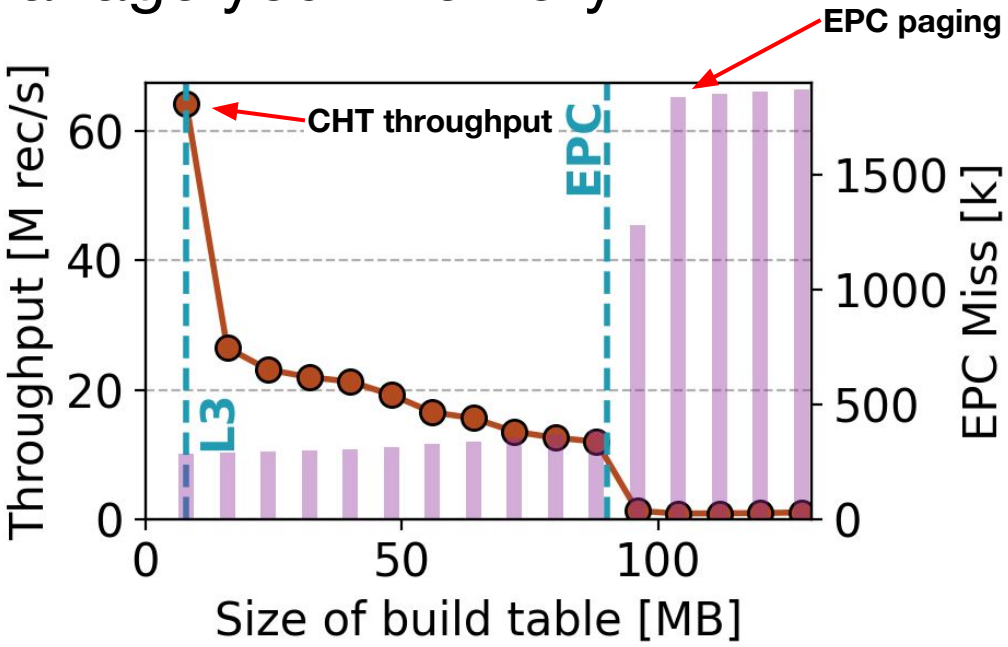
TeeBench is a fair referee for enclave benchmarks



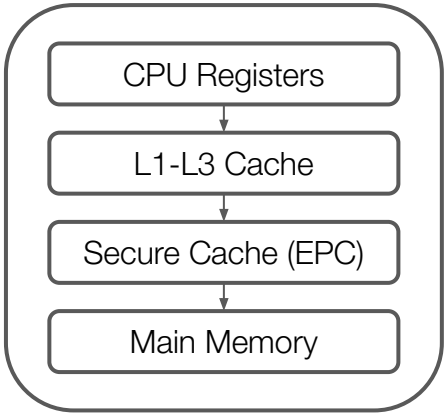
Plug & Play experience

We used TeeBench in all subsequent works

How to manage your memory?



SGX memory model

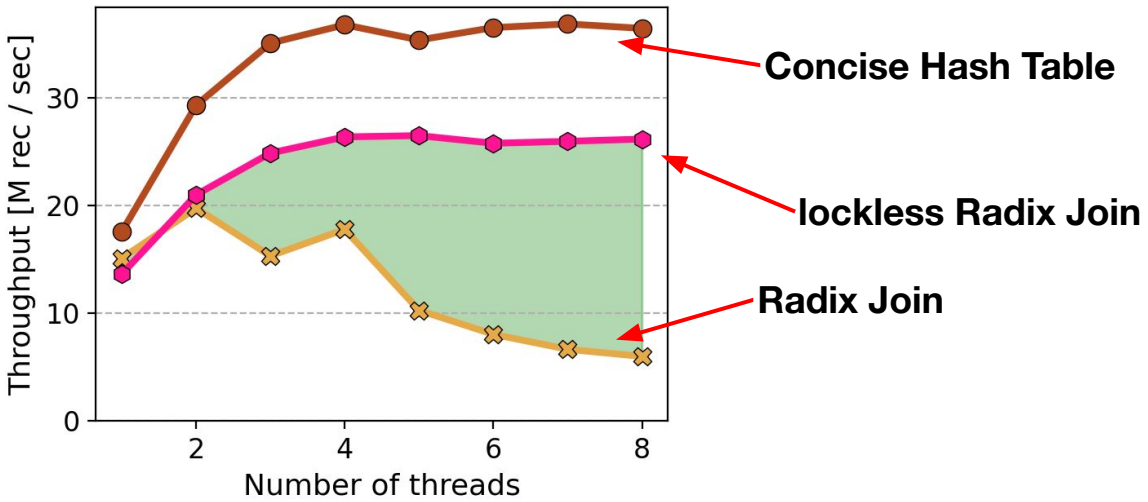


Lesson learned
Never exceed EPC memory

How to do multi-threading?

Mutex is a new bottleneck

Non-blocking primitives avoid OS interaction



Lesson learned
Design wait-free algorithms

We call for TEE-native solutions

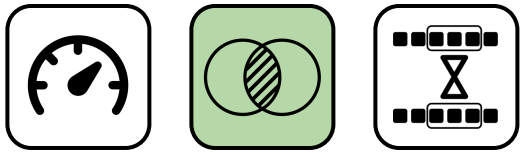
We provide 7 lessons learned

None of the joins is a good choice

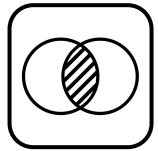
TEE-native approaches offer promising results

Agenda

1. Motivation
2. Understanding equi-joins in TEEs
3. **Relational Join for TEEs**
4. Privacy-preserving stream joins
5. Conclusions



Desiderata for TEE-native processing

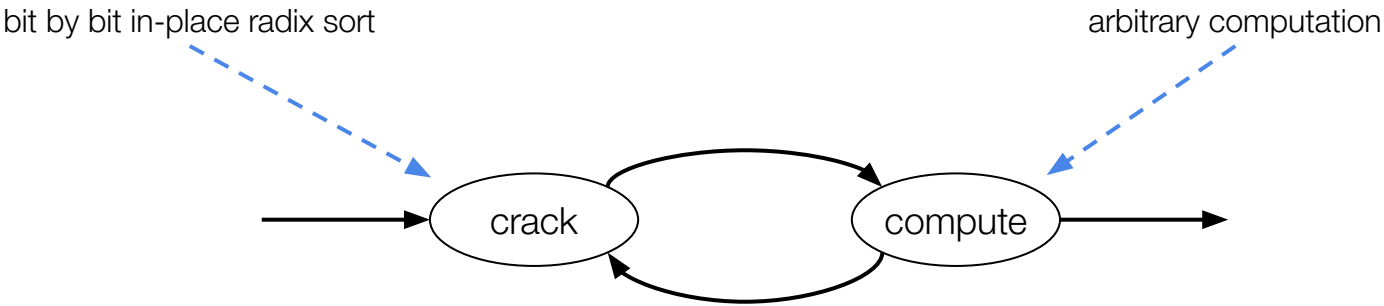


Sequential access patterns

Low memory consumption

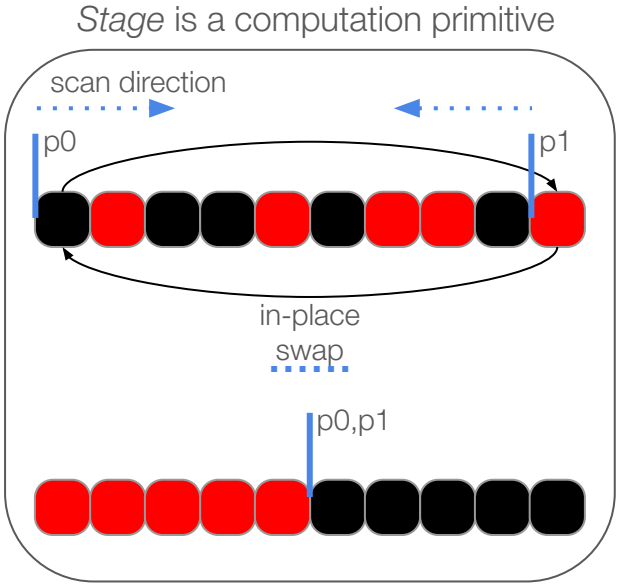
Wait-free algorithms

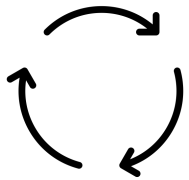
Cracking-Like Philosophy

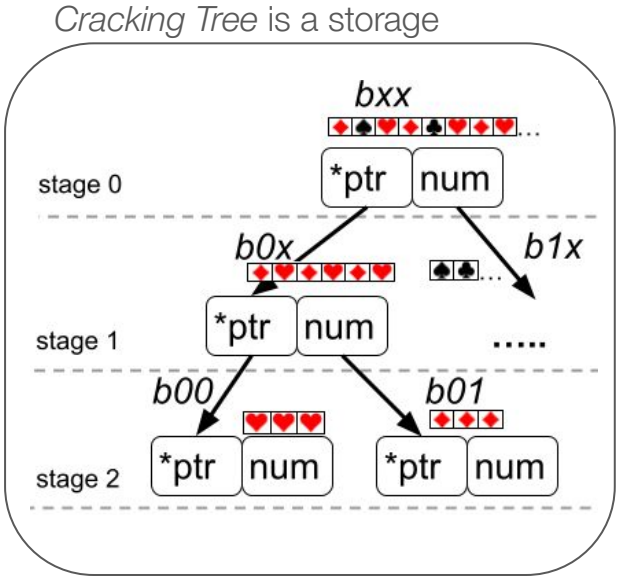


- Partition the data
- Perform sequential scans
- Consume little memory
- TBD: Barrier-free execution

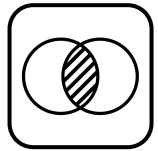
Two new primitives perform the Cracking-Like Join





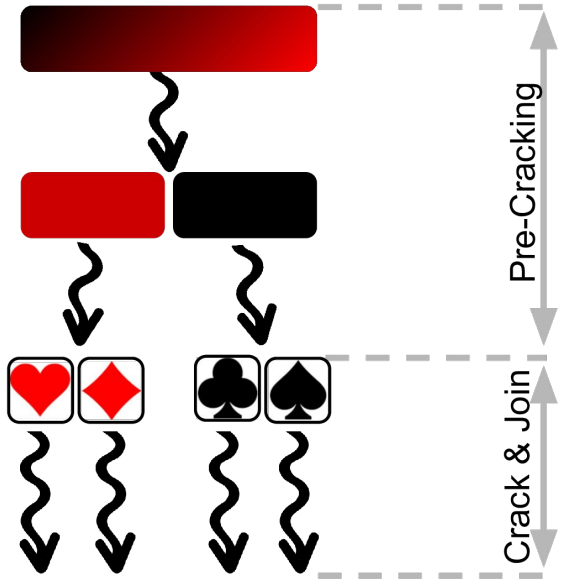


Cracking-Like Join algorithm

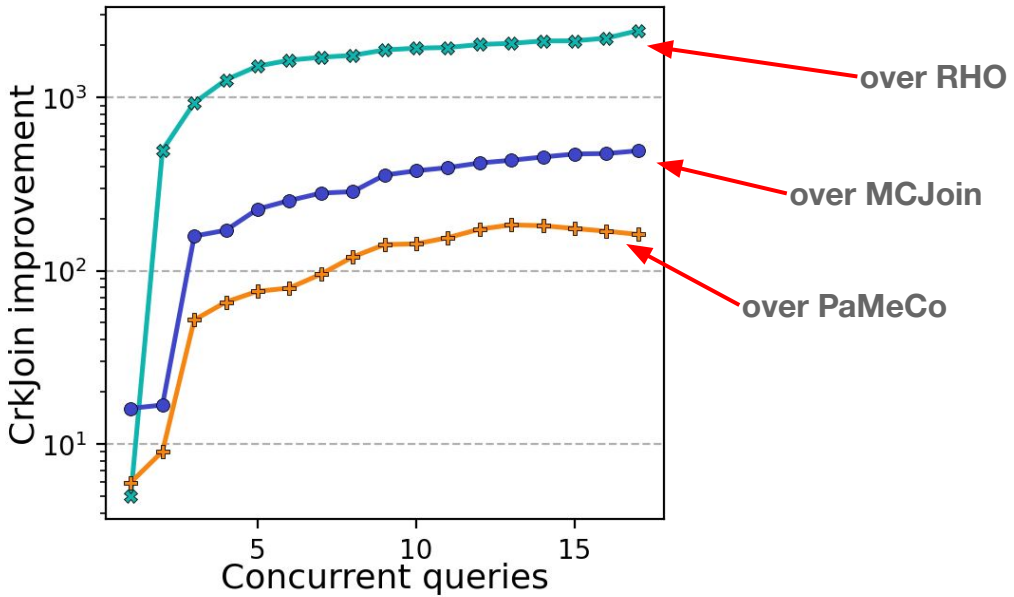


```
1: procedure CrkJoin(Relation R, Relation S, int bits):
2:     rootR ← InitCrackingTree(R)
3:     rootS ← InitCrackingTree(S)
4:     for each p ∈ [0..(2^bits − 1)] do
5:         CrackStageAndBuild(rootR, p)
6:         CrackStageAndProbe(rootS, p)
```

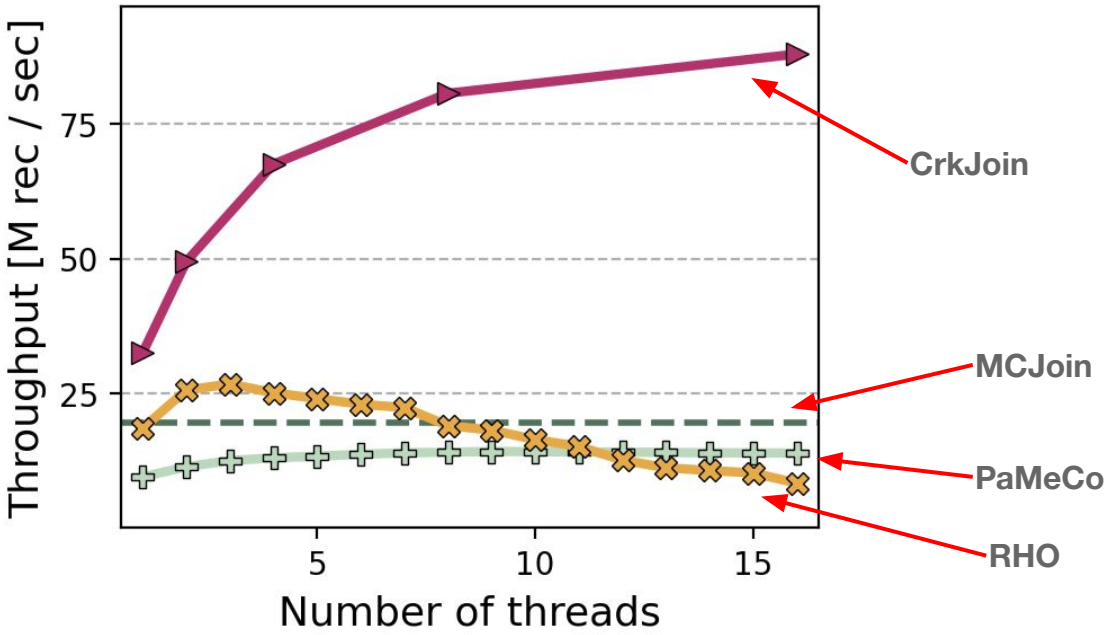
Partitioned relation forms independent chunks



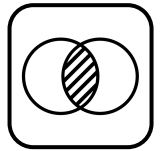
CrkJoin outperforms the baselines in a multi-tenant scenario



CrkJoin scales in multi-core architectures



Findings summary



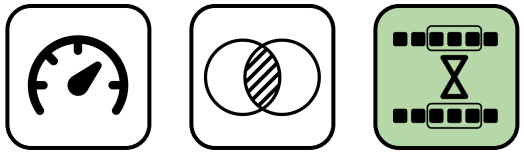
Efficient query processing in TEEs is possible

Our algorithm achieves up to 1000x higher performance

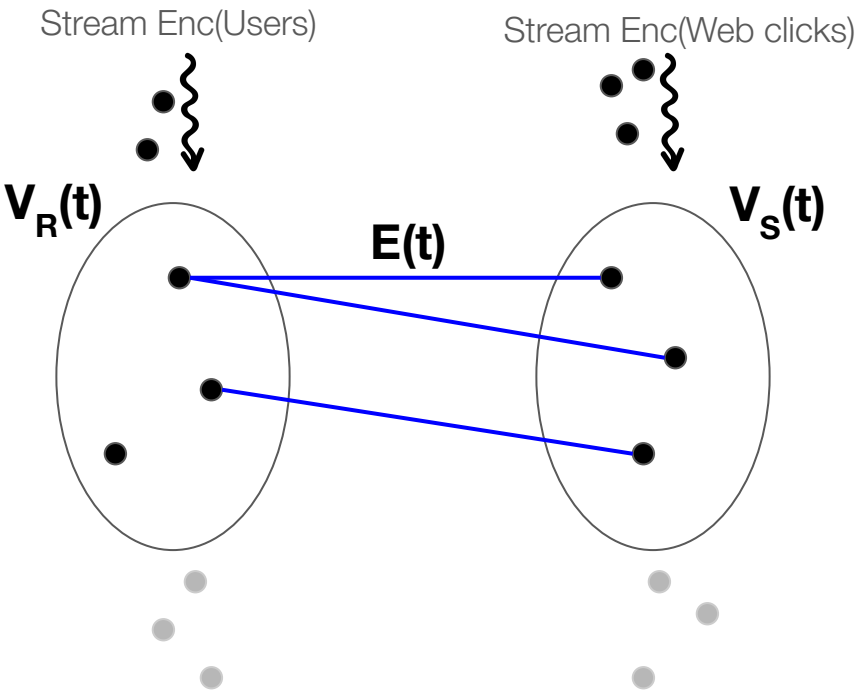
CrkJoin scales to multiple cores

Agenda

1. Motivation
2. Understanding equi-joins in TEEs
3. Relational Join for TEEs
4. **Privacy-preserving stream joins**
5. Conclusions

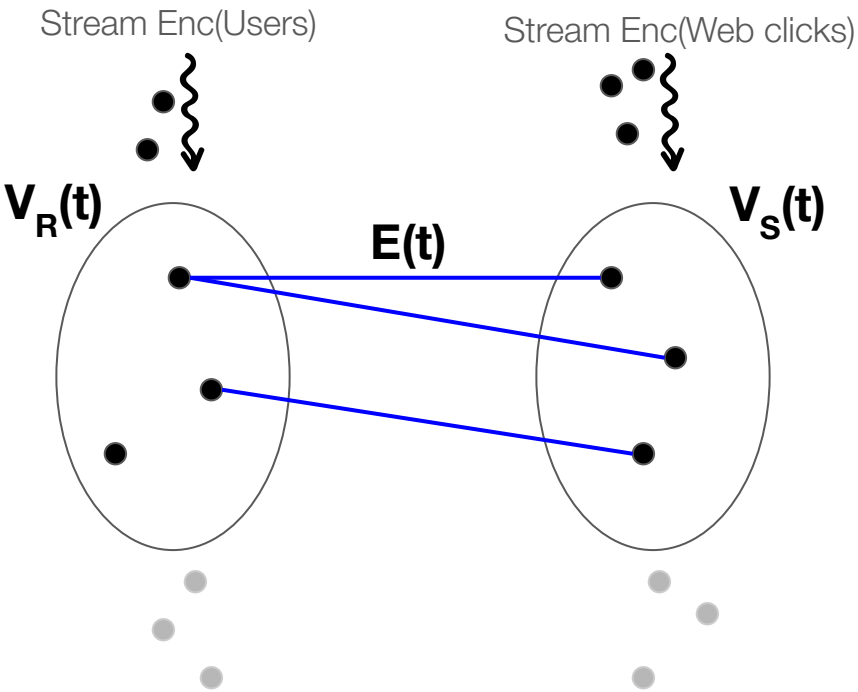


Stream join is a dynamic bipartite graph problem



$G(t) = (V_R(t), V_S(t), E(t))$

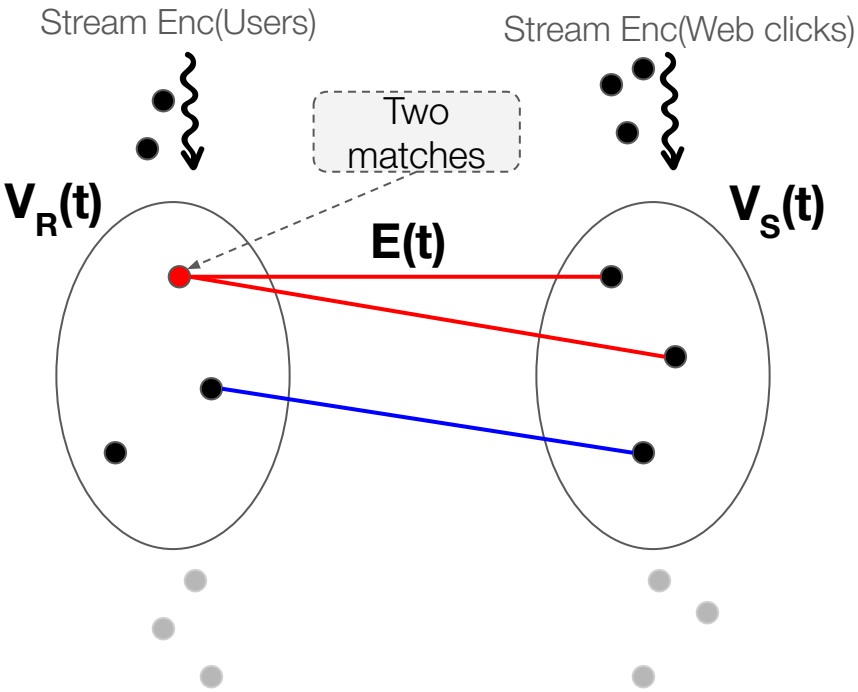
Stream join is a dynamic bipartite graph problem



$G(t) = (V_R(t), V_S(t), E(t))$

We protect data-dependent information (i.e., graph componentes)

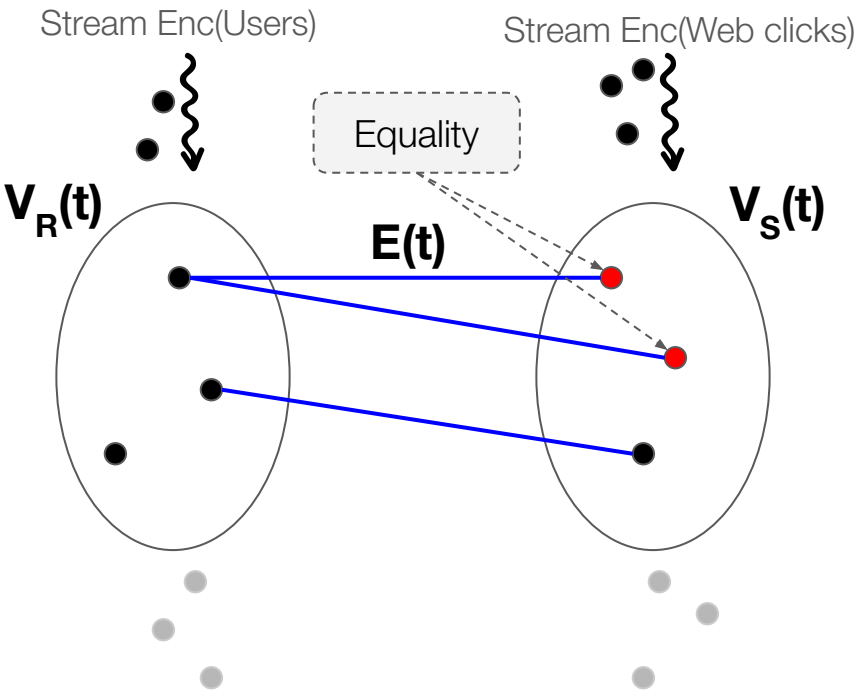
Stream join is a dynamic bipartite graph problem



$G(t) = (V_R(t), V_S(t), E(t))$

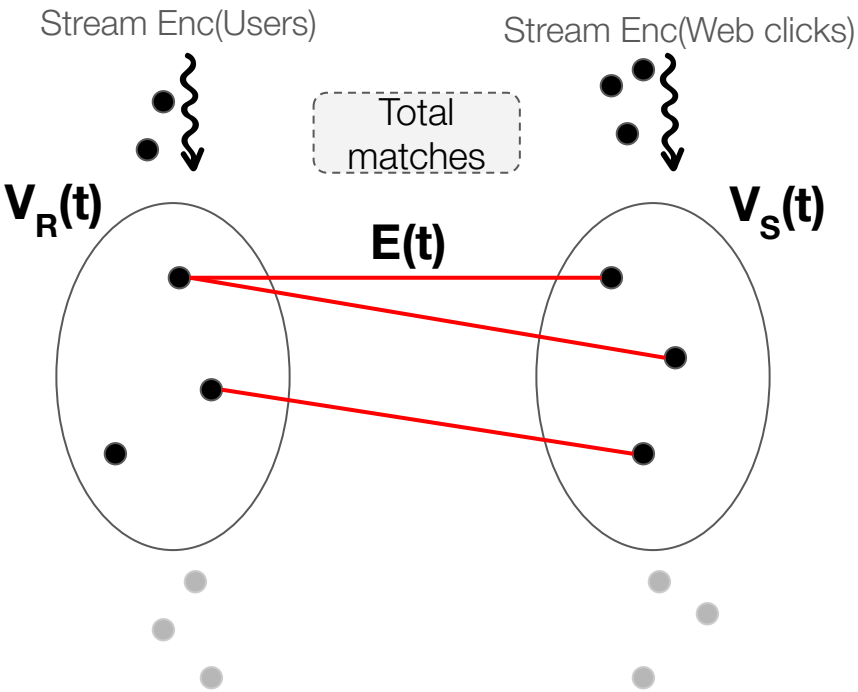
We protect data-dependent information (i.e., graph componentes)

Stream join is a dynamic bipartite graph problem



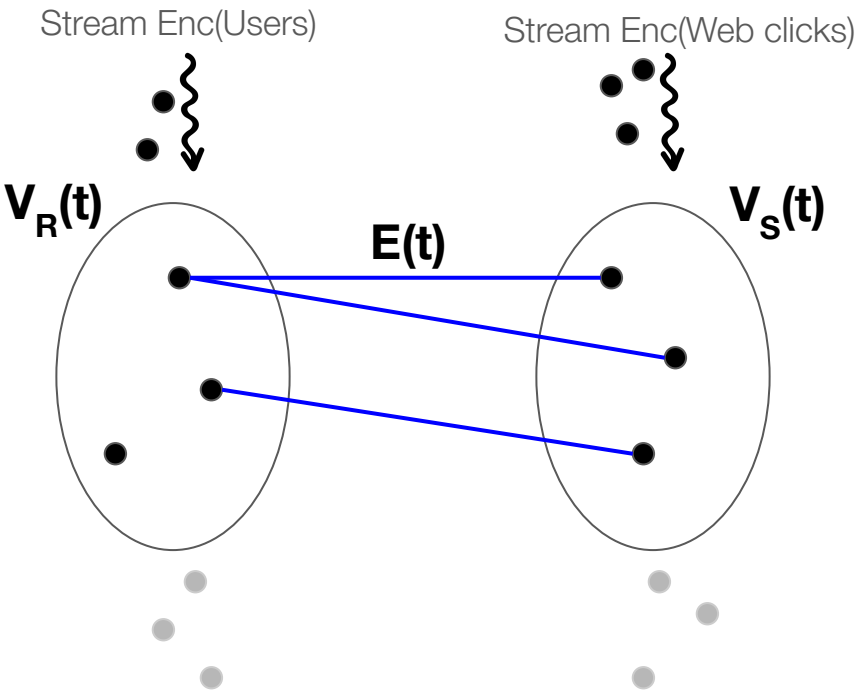
We protect data-dependent information (i.e., graph componentes)

Stream join is a dynamic bipartite graph problem



We protect data-dependent information (i.e., graph componentes)

Stream join is a dynamic bipartite graph problem



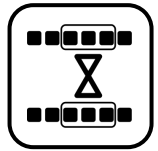
$G(t) = (V_R(t), V_S(t), E(t))$

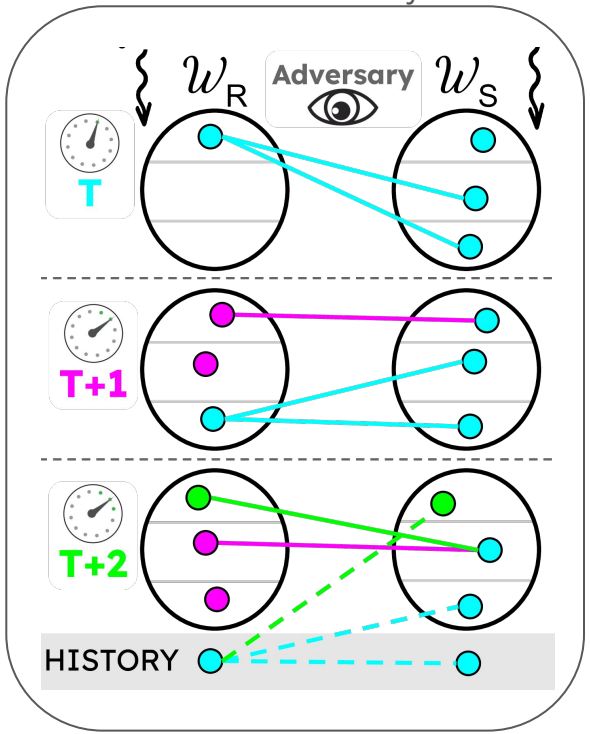
We protect data-dependent information (i.e., graph componentes)

We allow for data-independent leakage

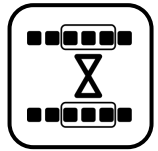
Stream constraints

We introduce a taxonomy of five leakage functions



	What the adversary sees	Informal definition
L0: deterministic confidentiality		Leak the entire join graph and its history. Comparable with deterministic encryption

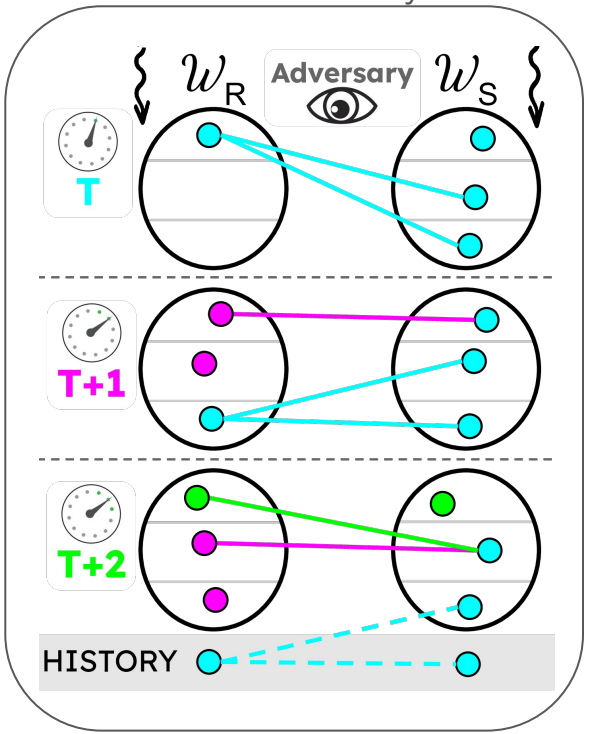
We introduce a taxonomy of five leakage functions



L0: deterministic confidentiality

L1: randomized confidentiality

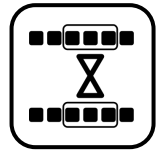
What the adversary sees



Informal definition

Leak current join graph and volume patterns, while hiding the graph history. Similar to randomized encryption.

We introduce a taxonomy of five leakage functions

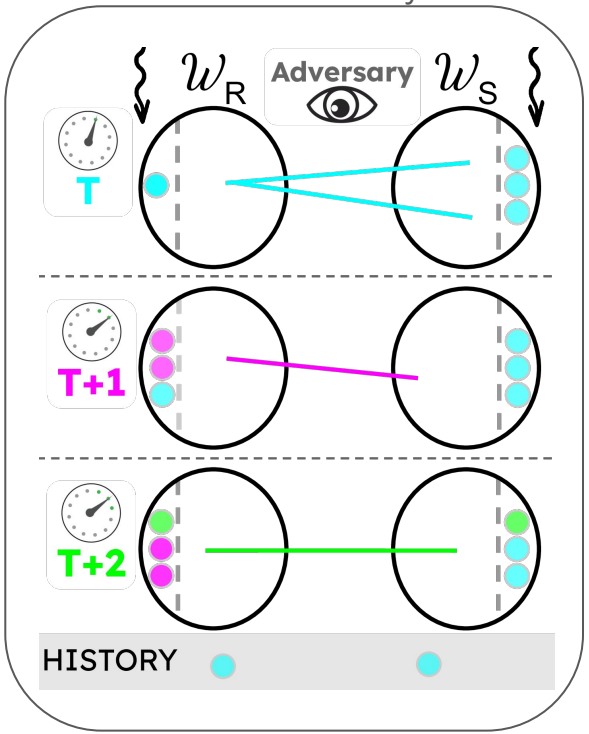


L0: deterministic confidentiality

L1: randomized confidentiality

L2: oblivious lookups

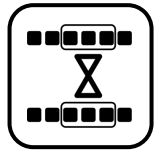
What the adversary sees



Informal definition

For every tuple leak only the volume pattern.

We introduce a taxonomy of five leakage functions



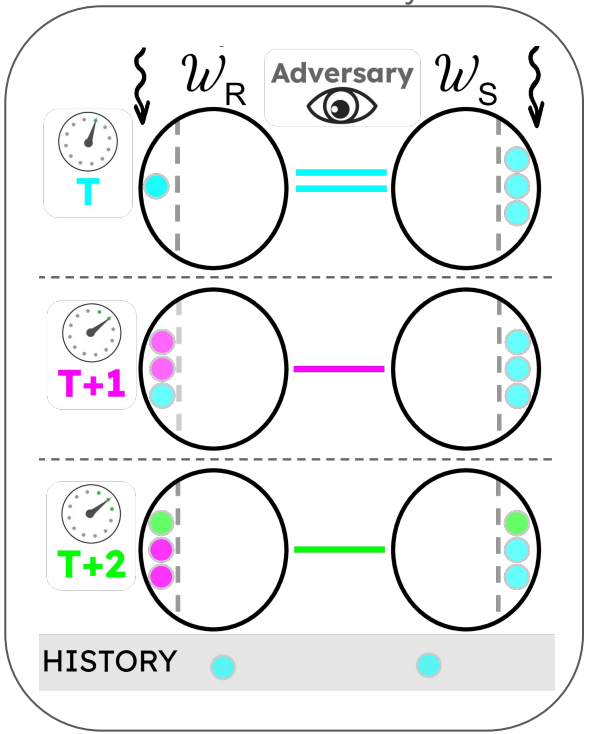
L0: deterministic confidentiality

L1: randomized confidentiality

L2: oblivious lookups

L3: oblivious batching

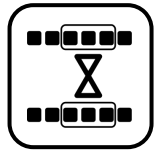
What the adversary sees



Informal definition

For every batch of tuples leak the intermediate join result cardinality.

We introduce a taxonomy of five leakage functions



L0: deterministic confidentiality

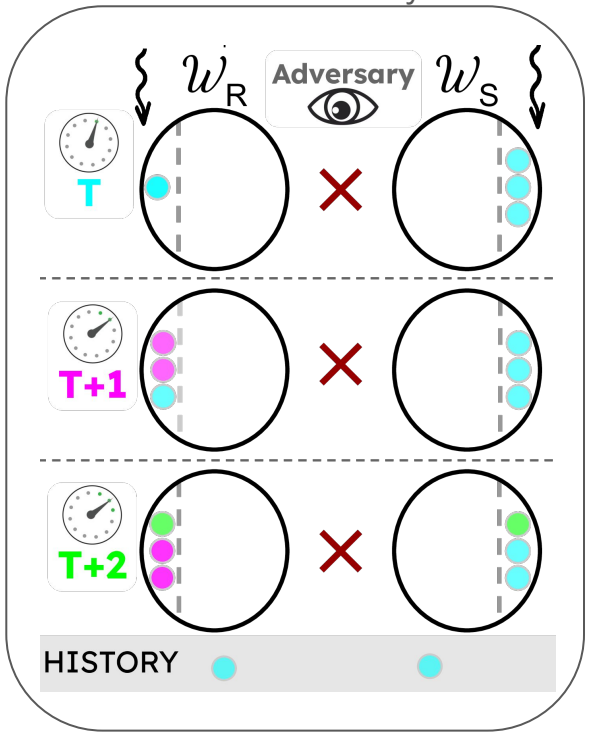
L1: randomized confidentiality

L2: oblivious lookups

L3: oblivious batching

L4: full obliviousness

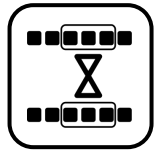
What the adversary sees



Informal definition

Leak no information about the data or access patterns.

We propose two families of stream joins secure against leakage functions



Oblivious Symmetric Hash Joins

- SHJ-L0
- SHJ-L1
- SHJ-L2
- SHJ-L3
- SHJ-L4

Oblivious Computation Approach Joins

FK joins:

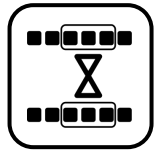
- OCA-L2
- MERG-L3/L4
- SORT-L3/L4

non-FK joins:

- JOIN-L2/L3

micro-batching mode

How to avoid oblivious sorting?



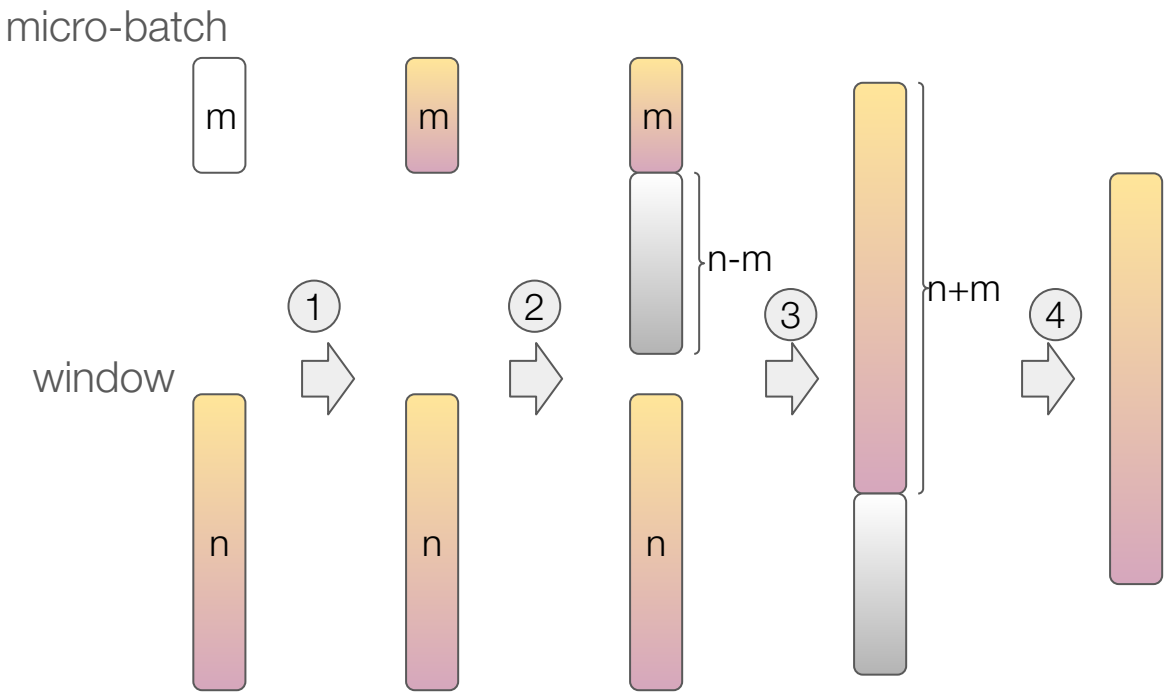
Problem
Oblivious joins heavily rely on OSort

Solution
Maintain the collections sorted

(R ∪ S)

	key	table
(1,1)	1	R
(1,1)	1	S
(2,2)	1	S
	2	R
	2	S
	3	R

How to obliviously append to a sorted collection?



OblivAppend

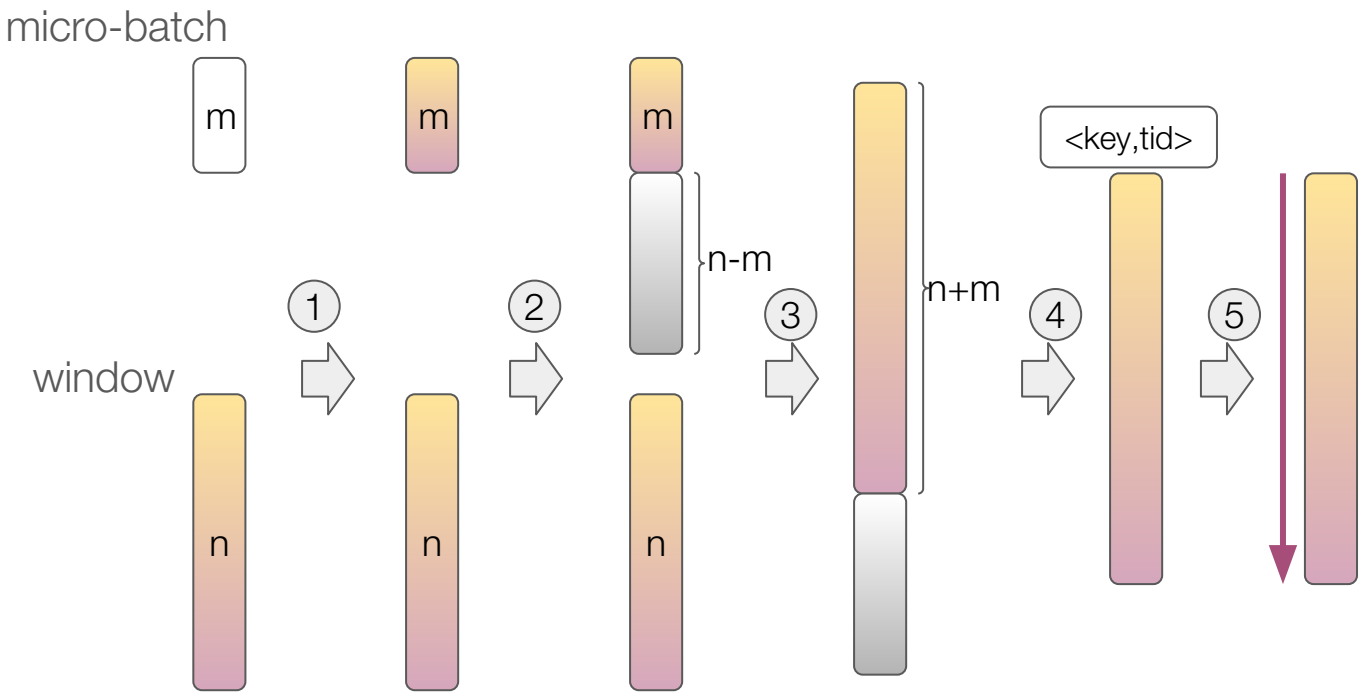
1. OSort micro-batch
2. Append dummies
3. OMerge
4. Trim dummies

Problem setup

1. window is sorted
2. batch is unsorted
3. n >> m
4. OSort is $O(n\log^2 n)$

If n >> m: **O(nlogn)**

OblivAppend as a FK-join building block



OblivAppend

1. OSort micro-batch
2. Append dummies
3. OMerge
4. Trim dummies

FK join

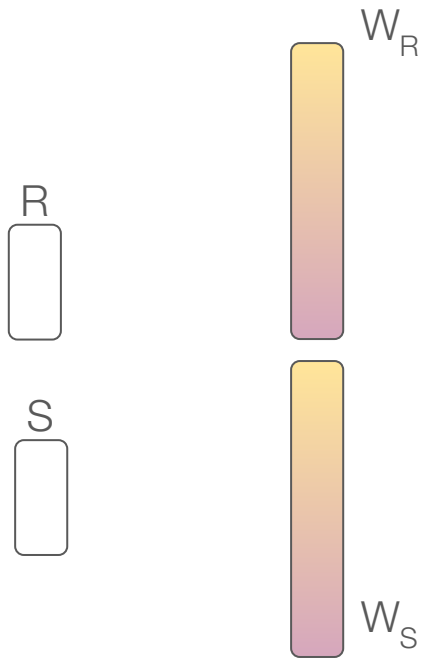
5. Emit join matches

MERG-L4 foreign-key join

Challenges: window correctness, result uniqueness, worst-case padding

FK restrictions allows us to split the join:

$$R \bowtie S = ((R \cup \mathcal{W}_R) \bowtie S) \cup (R \bowtie \mathcal{W}_S)$$

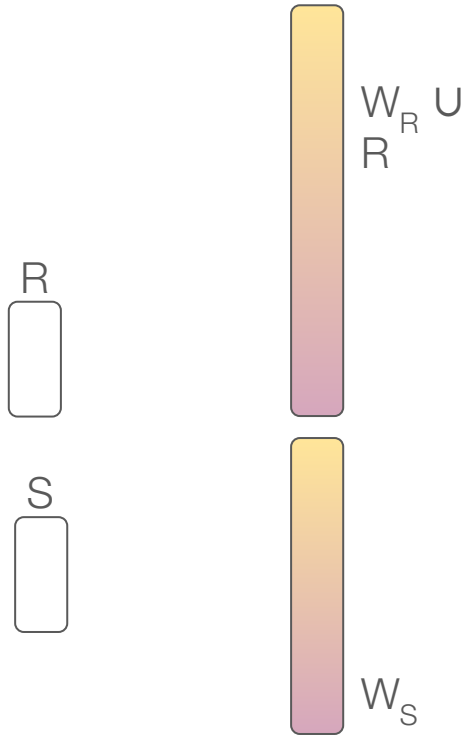


MERG-L4 foreign-key join

Challenges: window correctness, result uniqueness, worst-case padding

FK restrictions allows us to split the join:

$$R \bowtie S = ((\underbrace{R \cup \mathcal{W}_R}_{1}) \bowtie S) \cup (R \bowtie \mathcal{W}_S)$$



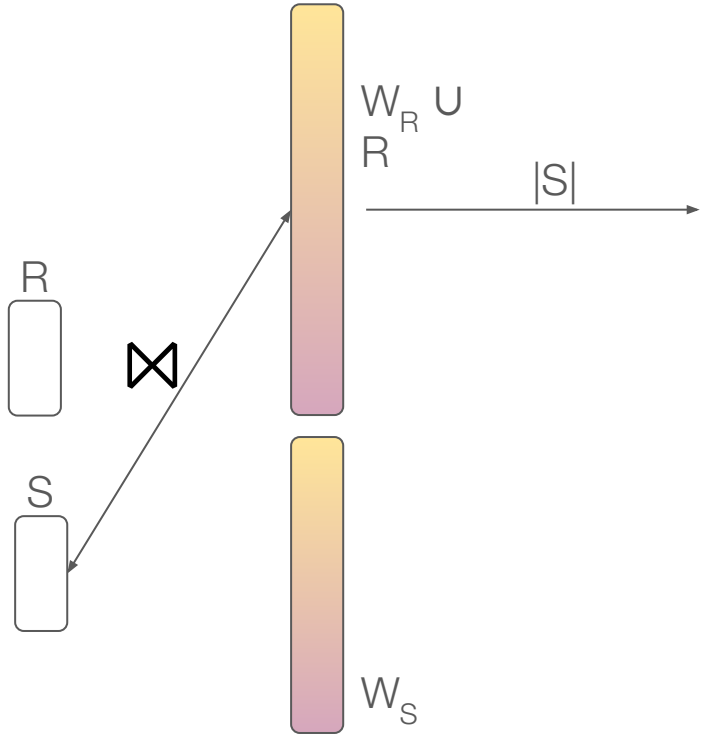
MERG-L4 foreign-key join

Challenges: window correctness, result uniqueness, worst-case padding

FK restrictions allows us to split the join:

$$R \bowtie S = ((R \cup \mathcal{W}_R) \bowtie S) \cup (R \bowtie \mathcal{W}_S)$$

2



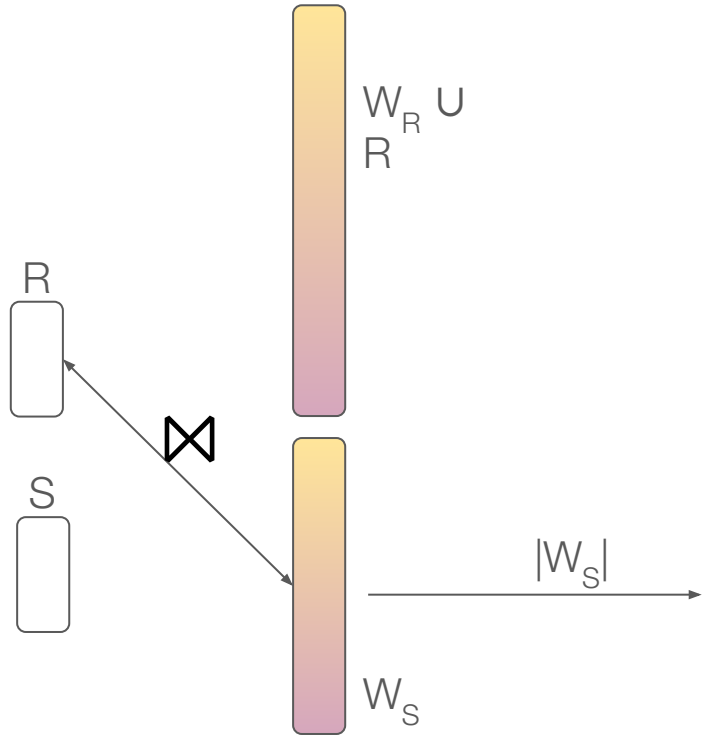
MERG-L4 foreign-key join

Challenges: window correctness, result uniqueness, worst-case padding

FK restrictions allows us to split the join:

$$R \bowtie S = ((R \cup \mathcal{W}_R) \bowtie S) \cup (R \bowtie \mathcal{W}_S)$$

3



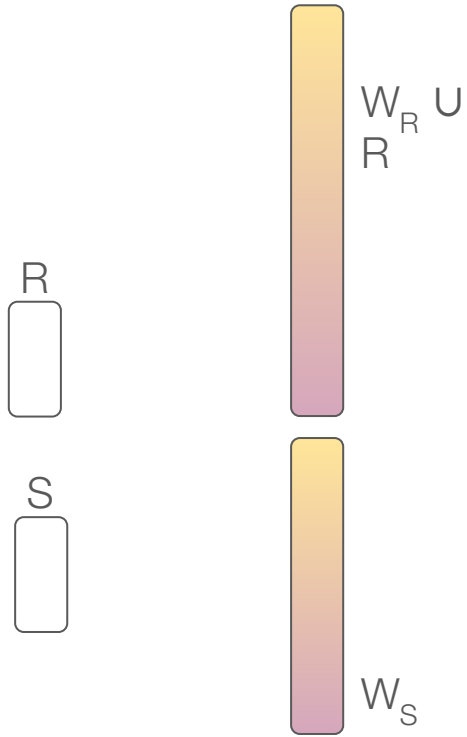
MERG-L4 foreign-key join

Challenges: window correctness, result uniqueness, worst-case padding

FK restrictions allows us to split the join:

$$R \bowtie S = ((R \cup \mathcal{W}_R) \bowtie S) \cup (R \bowtie \mathcal{W}_S)$$

4 Housekeeping

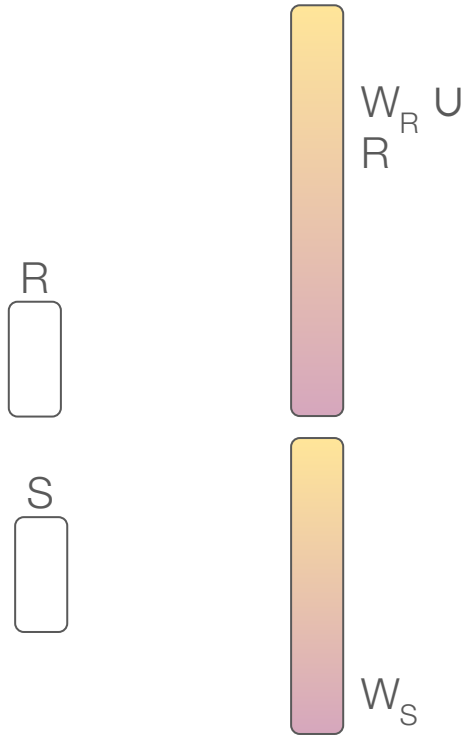


MERG-L4 foreign-key join

Challenges: window correctness, result uniqueness, worst-case padding

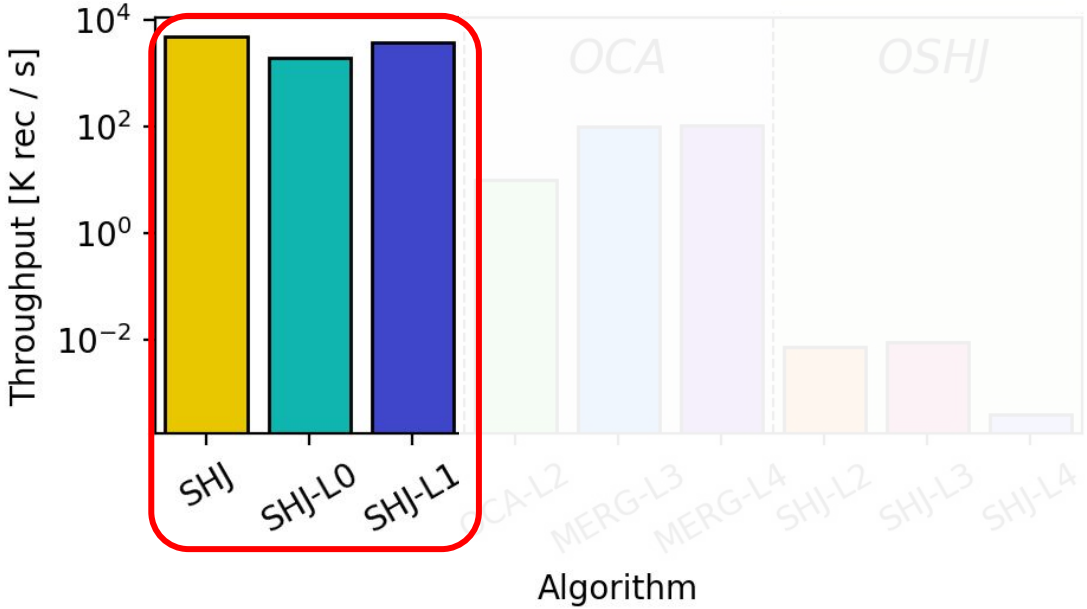
FK restrictions allows us to split the join:

$$R \bowtie S = ((R \cup \mathcal{W}_R) \bowtie S) \cup (R \bowtie \mathcal{W}_S)$$



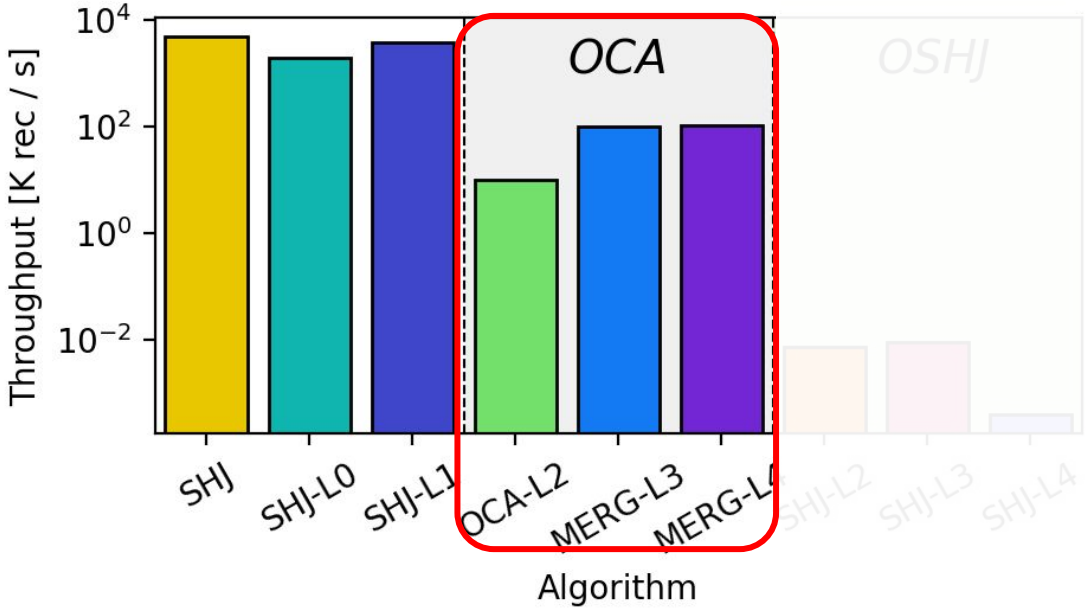
Security: all operations are oblivious
Performance: O(nlogn)

Privacy cost in oblivious stream joins



Privacy without obliviousness has negligible overhead

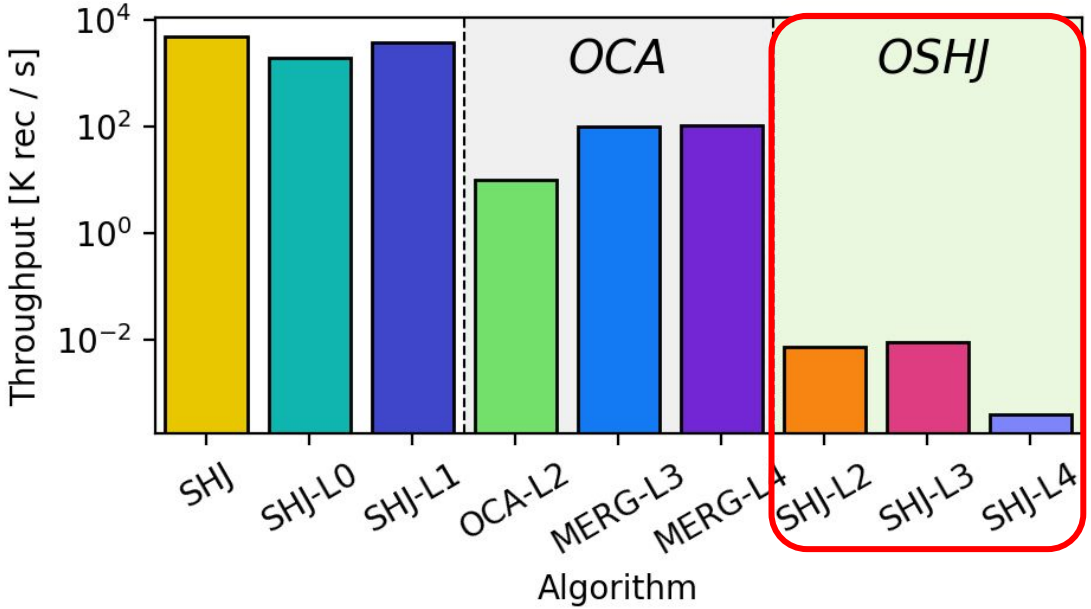
Privacy cost in oblivious stream joins



Privacy without obliviousness has negligible overhead

OCA joins perform well, while maintaining low latency

Privacy cost in oblivious stream joins

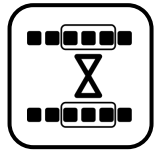


Privacy without obliviousness has negligible overhead

OCA joins perform well, while maintaining low latency

Oblivious index-based solutions significantly underperform

Contributions summary



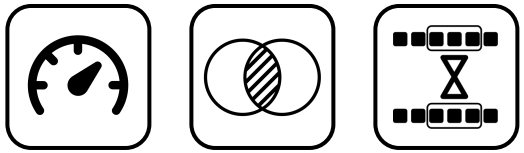
Oblivious stream join problem definition

OSJ taxonomy

Two families of oblivious stream joins

Agenda

1. Motivation
2. Understanding equi-joins in TEEs
3. Relational Join for TEEs
4. Privacy-preserving stream joins
5. **Conclusions**



Conclusions

- TEEs offer a more private future
- Performance and security issues to solve on the way
- Solutions to three core aspects of management of sensitive data:
  - Adoption of TEEs
  - Efficient Query Processing in TEEs
  - Privacy-Preserving Stream Processing
- Future work
- Our work makes privacy-preserving data processing more accessible

VLDB'22 | VLDB'23 | under submission | demo@ SIGMOD'23 | PhD@ VLDB'20