Privacy-Preserving Stream Joins

Kajetan Maliszewski
BIFOLD & TU Berlin
maliszewski@tu-berlin.de

Ioannis Demertzis
UC Santa Cruz
idemertz@ucsc.edu

Volker Markl
BIFOLD & TU Berlin
volker.markl@tu-berlin.de

ABSTRACT

A growing number of cloud applications process sensitive streaming data. Yet, the existing stream processing systems lack support for privacy-preserving operations, particularly joins. Despite using trusted hardware, these systems remain vulnerable to access pattern attacks when executing on untrusted cloud infrastructure.

We formally define the unexplored problem of oblivious stream joins in secure environments and introduce the first configurable leakage framework for such operators over encrypted data. Our framework defines five leakage functions that enable new privacy-performance tradeoffs in unbounded environments. We address the lack of support for secure stream join queries and propose two families of oblivious algorithms for non-foreign key and foreign-key joins: index-based, using ORAM indices, and computation-based, using oblivious primitives. Our solutions achieve this with new oblivious tools: ObliWind, a B-Tree-based doubly-oblivious window management data structure, and OAppend, an append primitive that maintains item order without revealing access patterns. Through extensive evaluation, we demonstrate that our fastest foreign-key stream join achieves performance within an order of magnitude of insecure baselines while offering strong privacy guarantees. Compared to ORAM-based approaches, we achieve a 3- to 6-orders-of-magnitude performance improvement, making privacy-preserving stream joins practical for real-world deployments.

PVLDB Reference Format:
Kajetan Maliszewski, Ioannis Demertzis, and Volker Markl.
Privacy-Preserving Stream Joins. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/kai-chi/obliv-stream-join.

1 INTRODUCTION

Modern cloud applications frequently handle real-time streaming data, such as healthcare devices analyzing physiological signals [19] and IoT devices tracking user locations. While many organizations have implemented privacy measures for static data [12, 76, 106, 118], these measures do not address the challenges of stream processing, such as continuous queries. Moreover, existing stream processing systems (SPSs) [11, 25, 121] lack privacy-preserving mechanisms, particularly when joining multiple streams [82], which remains among the most complex and resource-intensive tasks.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Secure Streaming: In cloud computing, data must be protected in all states: *at-rest*, *in-transit*, and *in-use* [2]. However, the existing SPSs [25, 121] protect only the first two states (via encryption and transport layer security), exposing data *in-use* to malicious observers (e.g., cloud administrators). Moreover, despite the three-state protection, query pattern leakage [26, 61, 81] and side-channel attacks [45, 65, 108] reveal additional information about the data. For example, consider GPS devices that provide real-time map services by joining stream location data with map data. Even with end-to-end encryption, malicious observers can monitor join patterns via memory accesses [15] - which encrypted location records access which regions and at what frequency. This metadata leakage reveals sensitive information: high join frequencies expose popular locations, join patterns identify users with comparable behaviors, and cross-stream patterns reveal user groups. Thus, while raw data remain encrypted, query execution metadata expose the very patterns encryption aims to protect. Therefore, there is a clear need for privacy-preserving SPSs that protect users' activities and behaviors.

Oblivious Computation: Many privacy-oriented solutions use Trusted Execution Environments (TEEs), hardware-isolated computing environments, as fundamental building blocks for data confidentiality [28, 43, 122]. TEEs work by creating encrypted memory regions accessible only by the CPU. Yet, as shown in the previous example and related work [26, 51, 56, 61, 63, 81], TEEs alone leak sensitive information from their encrypted memory. To solve this problem, practitioners employ *oblivious* computation, which meticulously tracks access patterns [47, 84, 115], ensuring independence from the input data. For example, *Signal* (an instant messaging service) operates an oblivious contact discovery atop TEEs [33], and *TikTok* utilizes an oblivious friends-matching mechanism [106]. Nonetheless, no existing system supports oblivious stream joins.

Limitations of Existing Works: While oblivious static joins have received significant attention [31, 43, 66, 122], no prior work has addressed oblivious stream joins in the *trusted hardware* model. Yet, naively using static joins in streaming leaks intermediate cardinalities. Therefore, we benchmarked two existing stream join algorithms - insecure Symmetric Hash Join (*SHJ*) and TEE-based Nested-Loop stream Join with worst-case padding (*NLJ-L4*). While *SHJ* is a popular index-based stream join algorithm [25, 60, 99], *NLJ-L4* is a strawman solution for stream joins that hides access patterns [73]. The results (Section 7.1) show a three orders of magnitude *performance* gap between the algorithms caused by *NLJ-L4*'s high complexity. Additionally, *SHJ* and *NLJ-L4* represent opposite sides of the privacy scale, uncovering a *privacy* gap in existing work. Moreover, in multi-join query plans, the overhead from *NLJ-L4*'s padding exhibits multiplicative growth, saturating downstream operators with dummy tuples. We strongly believe that users would benefit from stream joins with a tunable privacy-performance trade-off. Tunable privacy has been explored in various data processing scenarios [29, 38, 71], but it has not yet been applied to stream joins.

Therefore, we reveal a clear need for stream join algorithms that improve both *SHJ*'s security and *NLJ-L4*'s performance.

Challenges of Oblivious Stream Joins: Designing oblivious stream joins presents several challenges. First, achieving full obliviousness requires a costly cross join (i.e., Cartesian product) to hide output cardinality, yet no intermediate streaming leakage models exist. This reveals a gap in current security models, which do not align with the streaming paradigm. Second, stream applications prioritize low latency, often measured as throughput, and eager execution. However, existing oblivious techniques, largly based on Oblivious RAM (ORAM) [33, 79, 107] or assuming oblivious memory [43, 122], suffer from high latency [7] and complex parallelization [119]. TEEs, though widely adopted, lack native oblivious memory [53], rendering many prior techniques impractical. Third, stream joins operate over windows. It is unclear how to manage these efficiently while maintaining strong privacy. Overall, the privacy-performance tradeoff, already a challenge in static settings, is amplified in streaming, where real-time constraints and continuous updates make traditional approaches [64, 74, 117] unsuitable.

Privacy-Preserving Stream Joins: In this paper, we take the first step toward enabling fully private stream processing on untrusted infrastructure by introducing *oblivious stream joins* – a critical, yet unexplored, building block for secure streaming systems. Existing stream processing engines lack support for joins that protect against access-pattern leakage, leaving users vulnerable even when leveraging trusted hardware. To address this challenge, we propose a formal security framework tailored to streaming, consisting of five leakage functions ($\mathcal{L}_0$-$\mathcal{L}_4$), which represent a spectrum of privacy guarantees: from weak confidentiality to strong encryption with obliviousness. Our framework enables fine-grained control over the privacy-performance tradeoff, filling a critical gap in stream processing security. We instantiate the framework with the first suite of oblivious stream join algorithms, spanning index-based (SHJ) and computation-based (OCA) families. Our index-based joins rely on ObliWind, a novel B-Tree-backed, doubly oblivious sliding window data structure that supports efficient deletions and timestamp ordering. To move beyond the limitations of ORAM, we introduce OAppend, a lightweight oblivious primitive for ordered appends that underpins the OCA joins. Our flagship join, *FK-MERG-L4*, achieves full obliviousness while maintaining throughput within an order of magnitude of insecure baselines. Through extensive empirical evaluation, we demonstrate that our algorithms bridge the performance-privacy gap between insecure *SHJ* and the strawman *NLJ-L4*, outperforming ORAM-based solutions by 3-6 orders of magnitude. Our work lays the foundation for a new generation of practical and secure streaming systems that protect not only the data content, but also behavioral patterns emerging from that data.

In summary, we make the following contributions:

(1) We formulate the previously unexplored problem of oblivious stream joins (Section 3).
(2) We propose a formal security framework for joins in unbounded environments based on a novel leakage taxonomy (Section 4).
(3) We contribute two new primitives foundational to stream processing: ObliWind, an oblivious window data structure, and OAppend, an oblivious append that maintains sorted collections.
(4) We present the first two families of oblivious stream joins: index-based (Section 5), and oblivious-primitive-based (Section 6).
(5) We evaluate all algorithms, demonstrating significant improvement over baselines and the strawman solution (Section 7).

2 PRELIMINARIES

We now provide the necessary background. We discuss the stream processing and security foundations (Section 2.1), the threat model (Section 2.2), and related work (Section 2.3).

2.1 Background

Stream processing analyzes unbounded, continuous data in real-time, providing millisecond-level insights for applications such as financial transactions, social media feeds, and IoT sensors. We adopt the terminology from Verwiebe et al. [110]. A data stream $\mathcal{S}$ is an infinite sequence of tuples containing a timestamp τ, a key, and a payload. We define finite subsets of streams as windows $\mathcal{W}$ of size w, enabling operations that would otherwise be impossible to define, such as joins and aggregations. We focus on *sliding windows*, i.e., windows that contain tuples from an earlier time until the most recent tuple. Sliding windows support two modes: *time-based*, with tuples from the last τ time units, and *count-based*, containing the last k tuples. Stream tuples are processed *tuple-at-a-time* or in *(micro-)batches*. In the former, a tuple is joined as soon as it arrives, resulting in low latency and higher processing costs [25]. In the latter, tuples form batches that are joined at fixed intervals [16, 49, 78, 121], achieving higher average throughput but also higher latency. Streams can be consumed via push- and pull-based models. Typically, the pull-based model is used for incremental view maintenance [54] or dashboards, while the push-based model, which is our focus, for real-time cases [113]. Modern SPSs [16, 25] protect data via encryption. Despite this mechanism, user data remain vulnerable to side-channel attacks, including memory access patterns [50, 56], control flow execution paths [65], or output rates [89]. To mitigate these risks, SPSs should implement countermeasures, such as oblivious processing, constant-time algorithms, and padding. Our paper addresses this gap.

Stream join operators continuously combine rows from multiple streams on a join predicate, producing incremental results. Unlike relational joins [97] that combine static tables, stream joins process unbounded data using windows to delineate the working dataset. A *foreign key* join (FK) occurs when one stream contains unique keys (e.g., unique customers referenced by many orders). In contrast, a *non-foreign key* join (non-FK) is a generic join where streams may contain duplicates (e.g., joining on matching dates). Non-FK are more complex due to lacking relationship restrictions. The join type determines worst-case padding (i.e., the maximum output size): FK joins pad to the foreign key window size, while non-FK joins pad to the windows' Cartesian product, significantly affecting the performance. Finally, binary joins can be represented as bipartite *join graphs* [15], where tuples form two disjoint vertex sets and edges represent join matches.

Symmetric hash join (SHJ) is the primary join algorithm used in stream processing. SHJ maintains hash tables for both streams and performs the three-step procedure [58]. For each incoming tuple, it: (i) adds the tuple to its stream's hash table, (ii) probes the

opposite stream's table, and (iii) invalidates expired tuples. SHJ is often the only join implementation in SPSs [11, 16, 25] thanks to its $O(1)$ amortized time complexity for hash table operations.

Nested-loop join (NLJ) is a relational join that uses nested iterations through the joined tables. Despite quadratic performance, all major databases implement it due to simplicity and well-understood cost. By nature, NLJ executes privately (i.e., all data/code access patterns remain hidden) thanks to a deterministic sequential scan per tuple. In the MPC setup, NLJ is often used as a secure join [22, 67, 123]. NLJ also achieves full privacy when executed inside a TEE with worst-case padding. As a baseline stream join derived from prior MPC work, *NLJ-L4* (Section 7.1) sequentially scans the opposite window for each incoming tuple. It emits join results if it finds a match and, for full privacy, dummy tuples in case of no match.

Trust models. Secure computation includes Multi-Party Computation (MPC) [44], where multiple data owners compute without sharing inputs, and outsourced computation, where computation is offloaded to an untrusted party. For single-cloud solutions, we focus on outsourced computation with two trust models: *trusted hardware* (secure co-processors/CPUs in untrusted machines) [43, 91, 120, 122] or *trusted proxy* (separating computation from storage) [31, 36, 95, 102, 103, 112, 119]. While both protect data during computation, *trusted hardware* is more relevant to the public cloud as it provides a uniform environment and is fully supported by all major cloud providers. We assume the *trusted hardware* model.

Trusted Execution Environments (TEEs) , also called *enclaves* [59, 75, 88], are CPU-isolated execution environments that provide code and data confidentiality and integrity. Intel SGX [75] is the most widely deployed TEE in the public cloud. SGX encrypts data in isolated Enclave Page Cache (EPC) memory, with SGXv2 supporting up to hundreds of GBs. SGX persistently stores encrypted data outside enclaves via *sealing*, while also verifying code integrity and validating hardware configuration via *remote attestation*.

Oblivious processing [6, 15, 70] ensures that data/code access patterns and intermediate results remain hidden. An algorithm is oblivious if its access patterns are computationally indistinguishable across all inputs of the same size, i.e., a polynomial-time adversary cannot distinguish the memory address sequence accessed during computation for any two equal-length inputs. Obliviousness is crucial when offloading computation to untrusted parties, such as cloud providers, where it mitigates most of the side-channel attacks (e.g., access pattern, power consumption, or timing attacks) [45]. Oblivious computation has been applied to distributed analytics [122], relational databases [43], and graph processing [28], but not yet to stream processing. *Double-obliviousness* [79] extends these guarantees to both TEE-protected and external memory.

Oblivious sort sorts an array without revealing information about the process or the data. Two approaches are sorting networks (e.g., bitonic sort [21]) that use deterministic oblivious swaps, and bucket oblivious sort [17], combining oblivious shuffle with non-oblivious comparison sort. Bitonic sort is popular due to its practical efficiency and simple implementation. Despite its $O(n \log^2 n)$ running time being higher than other constructions (e.g., AKS [8] achieves $O(n \log n)$), it remains competitive due to small constants.

Oblivious compaction [18, 48] permutes elements, placing marked items before unmarked ones. Oblivious compaction is often used to filter out "dummy" elements introduced to hide previous memory operations. While naively sorting on marked flags yields correct results, Sasy et al. [96] proposed a faster, $O(NlogN)$ algorithm as a deterministic set of oblivious conditional swaps. Their method splits the input array into left and right sides, recursively compacting the left side, while computing the number of "marked" items in it. It then compacts the right side and conditionally swaps pairwise with the remaining elements from the left side.

Oblivious shuffle produces a random permutation without revealing access patterns. While assigning random identifiers and sorting works, specialized algorithms achieve higher performance. ORShuffle [96] is currently the fastest oblivious shuffle, using random split via oblivious compaction before recursively shuffling each half. Despite $O(nlog^2 n)$ complexity, it is practically efficient.

Oblivious RAM (ORAM) [47, 84] enables clients to access data stored on untrusted servers without revealing access patterns. Unlike traditional encryption, which only hides data values, ORAM shuffles and re-encrypts data after each operation, making repeated accesses appear random and unpredictable. Path ORAM [101] and Circuit ORAM [114] are among the widely used constructions. ORAMs are still facing numerous practical challenges, particularly in streaming, including high latency and limited parallelization [27].

Oblivious maps [28, 33, 79, 107] typically use oblivious AVL trees [4] inside TEEs with ORAM-stored nodes [115]. While AVL's self-balancing property is beneficial, achieving full obliviousness requires worst-case padding for all operations. This makes deletions particularly expensive due to greater constants than insertions. While [107] reports deletions as being 5× more expensive than insertions, many works [28, 79] disregard them completely.

2.2 Threat Model

We combine TEEs and obliviousness in a workflow similar to that of prior works [28, 31, 43, 64, 79, 107, 122]. We protect against adversaries with OS and infrastructure control who perform memory snapshots, monitor access patterns, network communication, and CPU instructions. However, the adversary cannot break into the TEE. Following prior work [31, 43, 64, 122], we consider certain input stream information public, specifically: window sizes and tuple counts (which reflect input/output sizes in static joins), as well as their timestamps. Therefore, the attacker can reconstruct the windows using public information. While mitigation techniques exist (e.g., fixed-size batching [36] and traffic obfuscation [109]), they incur very high costs. Thus, hiding this metadata is left to the stream provider and our framework supports both padded and unpadded streams. Further, we prevent the attacker from learning the values of any individual tuple, and we formally define which elements of the join graph are leaked using leakage functions. The adversary tries to learn these facts by observing memory and code access patterns. We comply with this threat model by executing operators within TEEs to ensure data confidentiality, integrity, and isolation. We design data-oblivious algorithms that hide memory access patterns and ensure deterministic code execution through code padding and dummy operations. We implemented our solution on Intel SGXv2 [75], the most popular cloud hardware enclave. Other TEEs (e.g., AMD SEV [59]) would serve equally well.

2.3 Related Work

Privacy-preserving data processing has evolved from weak encryption schemes (i.e., DET- and order-preserving encryption), used by CryptDB [90] and Always Encrypted [10]. Yet, a line of attacks [61, 81] has shown that they fail to provide the promised privacy. This led to TEE-based solutions: EnclaveDB [91] is a TEE database engine, while Cipherbase [13] executes secure transactions on FPGAs. Other attacks [62, 69] have proven that TEEs are vulnerable to access pattern leakage. This resulted in works that combine TEEs and *oblivious data processing*, often ORAM-based [47, 84]: Opaque [122] supports distributed analytics, ObliDB [43] is a database engine, Obladi [36] introduces oblivious transactions, and Graphos [28] implements oblivious graph algorithms. However, none have considered oblivious stream processing.

Other works have studied **oblivious relational (static) joins**. Chang et al. [31] introduced an index NLJ for trusted proxy model without support for ad-hoc updates, a central component of stream joins. Our OCA algorithms address this limitation through OAppend, enabling incremental updates without full reconstruction. Krastnikov et al. [64] proposed the first non-quadratic join for non-FK setups, also with no support for dynamic workloads. ObliDB [43] included a partially-oblivious (i.e., assuming some oblivious memory) hash join using TEEs. Opaque [122] presented a simple and efficient FK join using a single oblivious sort and a sequential scan. Our joins improve [122] by replacing the oblivious sort with a more efficient OAppend. Finally, Mavrogiannakis et al. [74] scale oblivious joins to parallel environments. Yet, we are the first to study encrypted stream joins in the trusted hardware model.

Stream Joins is a well-studied topic in the database community. Kang et al. [58] introduced the three steps of a sliding window join. Joins were proposed for parallel [46] and distributed [68] execution, FPGAs [105], approximate joins [37, 100], and non-index setups [39]. In addition, industry has proposed systems like Apache Flink [25] and Spark Structured Streaming [16]. While other systems have addressed geo-distribution [9] and adaptive stream synchronizations [57] , none consider execution in untrusted environments.

Others have worked on **privacy in dynamic environments**. Whisper [93] computes aggregate statistics in the MPC model (similar to [3, 35]). Ostrovsky et al. [85] and Bethencourt et al. [24] investigated private searching on streaming data. Lastly, Streambox-TZ [87] offloads sensitive computation to a TEE but leaks data access patterns. In summary, existing privacy-preserving stream systems operate in different threat models (i.e., DP - targeting public datasets, and MPC - assuming non-collusion) and focus only on aggregations, a simpler database operator. This renders them indirect competitors. Recall that obliviousness protects the computation but not its result. While out of scope, Differential Privacy (DP) [42] and padding [38] can complement obliviousness, reducing query result leakage [23, 30, 32, 40, 41, 92].

3 OBLIVIOUS STREAM JOIN PROCESSING

We present the first formal security scheme for private stream joins (Section 3.1) and define its real/ideal security game (Section 3.2). Then, we formally define the problem of this paper (Section 3.3).

3.1 Oblivious Stream Join Definitions

We introduce an oblivious stream join (OSJ) scheme that follows our threat model. OSJ is a one-party protocol and consists of two algorithms: OSJ-Init and OSJ-Join. The former initializes the server's state, and the latter performs a join operation:

- $EM \leftarrow$ OSJ-Init$(1^\lambda, \mathcal{D})$: takes as input a security parameter λ and the data collection $\mathcal{D}$. It outputs the server's encrypted state EM, storing the stream windows content.
- $(\mathbf{T}, EM) \leftarrow$ OSJ-Join$(\mathcal{B}, EM)$: the client's input is a batch $\mathcal{B}$ of any number of tuples. The output consists of the join result tuples $\mathbf{T}$ and the updated encrypted state EM.

3.2 Security Game

The security proof of OSJ is based on a simulation of accesses constructed using only public information (similar to standard ORAM security [47]). We define a real/ideal security game, where the attacker interacts with a machine running the real OSJ protocol ("real") or a machine running an OSJ simulator ("ideal"). The simulator utilizes only publicly available information obtained from the execution trace defined by a leakage function. Intuitively, our OSJ construction is secure if the attacker has no advantage in guessing which world ("real" or "ideal") it operates in.

Definition 1. *Suppose (*OSJ-Init, OSJ-Join*) is an OSJ scheme based on the above definition, let $\lambda \in \mathbb{N}$ be the security parameter, and consider experiments $\textbf{\textit{Real}}(\lambda)$ and $\textbf{\textit{Ideal}}_{\mathcal{L}}(\lambda)$ presented in Algorithm 1, where $\mathcal{L}$ is a leakage function. OSJ is $\mathcal{L}$-secure if for all polynomial-size adversaries* Adv*, there exist polynomial-time simulators* IdealOSJ-Init *and* IdealOSJ-Join*, such that:*

$$\left| Pr\left[\textbf{\textit{Real}}^{OSJ}_{\Pi,Adv}(\lambda) = 1\right] - Pr\left[\textbf{\textit{Ideal}}^{OSJ}_{Sim,Adv,\mathcal{L}}(\lambda) = 1\right]\right| \leq negl(\lambda)$$

Algorithm 1 OSJ real-ideal security experiments.

$bit \leftarrow \textbf{Real}^{OSJ}_{\Pi,Adv}(\lambda)$:

1: $(st_A, \mathcal{D}) \leftarrow Adv(1^\lambda)$
2: $EM \leftarrow$ OSJ-Init$(1^\lambda, \mathcal{D})$
3: **for each** $k = 1$ *to* q **do** ▷ q: polynomial #batches
4: $(\mathcal{B}_k, st_A) \leftarrow Adv(EM, \mathbf{T}_1, ..., \mathbf{T}_{k-1}, st_A)$
5: $(\mathbf{T}_k, EM) \leftarrow$ OSJ-Join$(\mathcal{B}_k, EM)$
return $bit \leftarrow Adv(\mathbf{T}_1, ..., \mathbf{T}_k, st_A)$

$bit \leftarrow \textbf{Ideal}^{OSJ}_{Sim,Adv,\mathcal{L}}(\lambda)$:

1: $(st_A, \mathcal{D}) \leftarrow Adv(1^\lambda)$
2: $EM \leftarrow$ IdealOSJ-Init$(1^\lambda, \mathcal{D})$
3: **for each** $k = 1$ *to* q **do** ▷ q: polynomial #batches
4: $(\mathcal{B}_k, st_A) \leftarrow Adv(EM, \mathbf{T}_1, ..., \mathbf{T}_{k-1}, st_A)$
5: $(\mathbf{T}_k, EM) \leftarrow$ IdealOSJ-Join$(\mathcal{L}(\mathcal{B}_k), st_A)$
return $bit \leftarrow Adv(\mathbf{T}_1, ..., \mathbf{T}_k, st_A)$

3.3 Problem Definition

We perform a binary equi-join over two encrypted streams, $\mathcal{S}_R$ and $\mathcal{S}_S$, containing tuples $< \tau, k, p >$, with timestamp, key, and payload, respectively. We define two count-based, sliding windows, $\mathcal{W}_R$ and $\mathcal{W}_S$, of sizes w_R and w_S, and a *leakage function* $\mathcal{L}$ that defines what

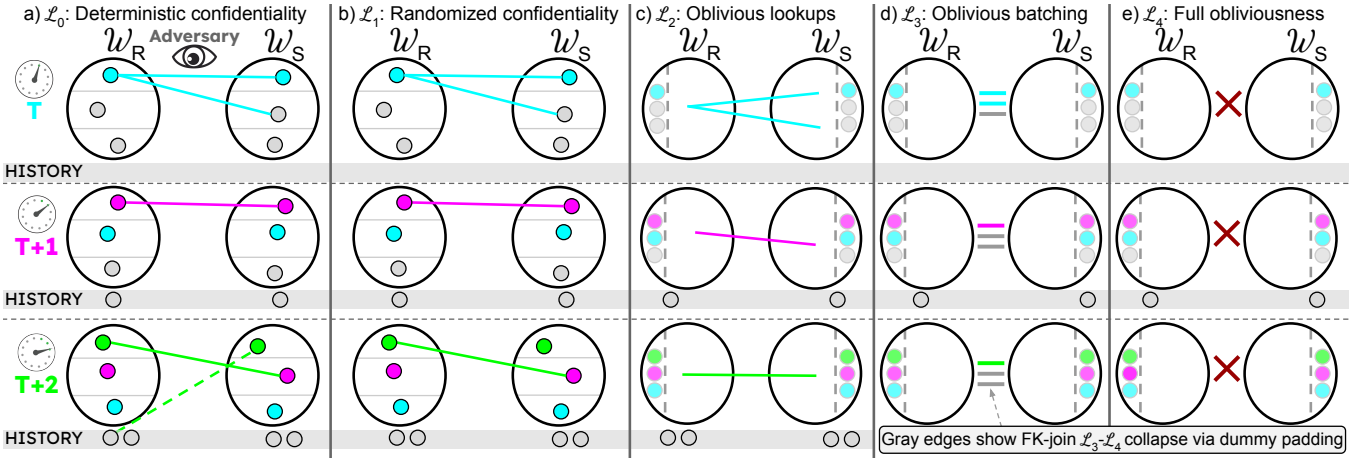

Figure 1: Stream join graph elements visible to adversaries over time per leakage profile. Colors indicate time spans between batch additions, dashed lines show indirectly learned matches, and gray edges represent optional dummy tuples leading to $\mathcal{L}_4$.

a secure protocol is allowed to leak. Given $\mathcal{W}_R$, $\mathcal{W}_S$, and $\mathcal{L}$, an OSJ algorithm appends tuple pairs (r, s) exactly once to an append-only join result set iff the tuples satisfy the join predicate P and at the time when r arrives, s is in $\mathcal{W}_S$ (similarly, when s arrives, r is in $\mathcal{W}_R$). In addition, the algorithm is $\mathcal{L}$-secure according to Def. 1. Formally: $\mathcal{S}_R \bowtie_{\mathcal{L}} \mathcal{S}_S = \{(r, s) | P(r, s), r \in \mathcal{W}_R, s \in \mathcal{W}_S\}$

Our definition aligns with foundational streaming research [46, 58, 105] and standard SPSs [9, 11, 16, 25, 57].

Limitations. We process streams in the order of arrival. Our solutions are easily generalizable to other window types as long as window management is independent of the data values (e.g., session, tumbling, or fixed-band windows). We leave out for future work windows with conditional logic (e.g., threshold windows), where branching can leak access patterns.

4 LEAKAGE TAXONOMY

Privacy-preserving stream processing is fundamentally challenging because it alters the underlying assumptions of secure computation. Oblivious joins must now operate over *unbounded inputs*, under *strict latency constraints*, and with *evolving state*, rendering traditional static security models inadequate. Moreover, the lack of formal definitions for oblivious stream joins hinders fair comparison between solutions that often assume different threat models. This creates a clear need for a unified framework to reason about privacy guarantees in streaming contexts.

We address this gap by introducing a leakage taxonomy framework for privacy-preserving stream joins. At its core, the framework formalizes five leakage functions that capture adversarial observability over time, allowing to precisely describe what an attacker infers in different scenarios. We define these functions in terms of a dynamic bipartite graph [86], referred to as the *join graph* [15], which evolves as new data arrive and join matches are recomputed. While our framework establishes foundational primitives for oblivious binary stream joins, it naturally extends to multi-way joins and other downstream operators with similarly defined leakage functions. We defer these extensions to future work.

Formally, graph $G(t) = (V_R(t), V_S(t), E(t))$ is a join graph at time t, where V_R and V_S are vertex sets of encrypted tuples stored in windows $\mathcal{W}_R$ and $\mathcal{W}_S$ of unbounded streams $\mathcal{S}_R$ and $\mathcal{S}_S$, and E is the set of edges denoting join matches between them. Each edge $e \in E$ connects a tuple from V_R to one in V_S. We denote edges incident to a vertex v as $E_v(t)$, the degree of v as $deg_v(t)$, and vertices connected to (i.e., matched tuples) v as $J_v(t)$. Tuples are added in batches $\mathcal{B}$, reflecting the behavior of many SPSs [78, 121].

We propose five leakage functions ($\mathcal{L}_0$-$\mathcal{L}_4$), each capturing progressively stronger privacy guarantees. They range from a leakage equivalent to deterministic (DET) encryption that reveals complete join information, to a leakage with full obliviousness that leaks nothing beyond public parameters. Figure 1 visualizes the intuition behind $\mathcal{L}_0$ through $\mathcal{L}_4$. At a high level, $\mathcal{L}_0$ and $\mathcal{L}_1$ protect only data content via weak and strong data encryption, while exposing access patterns. $\mathcal{L}_2$ hides tuple-level access patterns but leaks volume information, while $\mathcal{L}_3$ generalizes this to batch-level leakage. Finally, $\mathcal{L}_4$ hides all access patterns, achieving maximal privacy. $\mathcal{L}_0$-$\mathcal{L}_2$ execute per-tuple, producing one output tuple per match, while $\mathcal{L}_3$-$\mathcal{L}_4$ execute per batch, outputting the batch join result ($\mathcal{L}_3$) or batch Cartesian product ($\mathcal{L}_4$).

The existing access pattern attacks [26, 50, 51, 56, 61, 63, 81] have proven $\mathcal{L}_0$- and $\mathcal{L}_1$-secure static solutions insufficient, as they leak enough to recover plaintext join results. On the other hand, while $\mathcal{L}_4$ offers the strongest protection, its worst-case complexity is often prohibitive. However, FK join constraints allow for bounded output cardinality, making $\mathcal{L}_4$-secure FK joins practical. In subsequent sections, we demonstrate that, through careful design, efficient $\mathcal{L}_4$-secure FK joins can be achieved.

Table 1 classifies prior secure static joins within this taxonomy as originally presented (e.g., with/out padding). Note that while we also classify MPC and *trusted proxy* works , it is not adequate to directly compare their performance due to fundamental differences in communication (multi- vs. single-party), cryptographic primitives (secret sharing vs. TEE isolation), and security assumptions (non-collusion vs. hardware trust). We now formally define the leakage functions in terms of graph $G(t)$.

Table 1: Examples and classification of related secure static join work in their default mode, and the complexity per tuple of all our proposed stream joins. We denote window size as N, batch size as m, and batch output size as T.

	Non-FK Joins			FK Joins		
LF	**Static join prior work**	**Our stream join**	**Complexity**	**Static join prior work**	**Our stream join**	**Complexity**
$\mathcal{L}_0$	CryptDB [90], Always Encrypted [10]	*SHJ-L0*	$O(\log N)$	-	-	-
$\mathcal{L}_1$	EnclaveDB [91]	*SHJ-L1*	$O(\log N)$	CrkJoin [72]	-	-
$\mathcal{L}_2$	Chang et al. [31]	*SHJ-L2*	$O(\log^2 N)$	ObliDB [43] (no padding)	*FK-EPHI-L2*	$O((N \log N)/m)$
$\mathcal{L}_3$	Sovereign joins [5], Arasu et al. [15], Krastnikov et al [64],Shrinkwrap [23]	*SHJ-L3* *NFK-JOIN-L3*	$O(T \log^2 N)$ $O((N \log^2 N)/m)$	Opaque [122] (no padding)	*FK-MERG-L3* *FK-SORT-L3*	$O((N \log N)/m)$ $O((N \log^2 N)/m)$
$\mathcal{L}_4$	SMCQL [22], Secrecy [67], Conclave [112], Secure Yannakakis [116]	*SHJ-L4*	$O(N \log^2 N)$	Opaque [122], ObliDB [43]	*FK-MERG-L4* *FK-SORT-L4*	$O((N \log N)/m)$ $O((N \log^2 N)/m)$

4.1 $\mathcal{L}_0$: Deterministic Confidentiality

Hint: Leak the full join graph and its history.

For each batch $\mathcal{B}$ between t_0 and t_i, the adversary learns the edges $E(t)$ (exact join matches) and, for each vertex $b \in \mathcal{B}$, its degree $deg_b(t)$ (i.e., volume pattern) and join indices $J_b(t)$. The adversary can correlate current data with expired tuples (e.g., dashed line in Figure 1a) and reconstruct historical join results. For instance, the DET-encrypted equi-joins in Always Encrypted [10] and CryptDB [90] are $\mathcal{L}_0$-secure. Formally:

$$\mathcal{L}_0(G(t_i)) = \left\langle (E(t_j), (deg_b(t_j), J_b(t_j))_{b \in \mathcal{B}_j})_{j \in \{0..i\}} \right\rangle$$

4.2 $\mathcal{L}_1$: Randomized Confidentiality

Hint: Leak the current join graph without historical links.

$\mathcal{L}_1$ improves upon $\mathcal{L}_0$ by removing the adversary's ability to link current tuples to historical data. It reveals only the current join graph (i.e., results and degrees from the latest batch), but not if tuples match previously seen data. This is equivalent to RND encryption within a TEE, where each batch is encrypted independently. Figure 1b illustrates that $\mathcal{L}_1$ protects links to past data (i.e., the absence of a green dashed line in $T + 2$). Effectively, the adversary has no benefit in storing historical data. Solutions based on TEEs with RND encryption, such as Intel SGXv1, are $\mathcal{L}_1$-secure, for example, EnclaveDB [91] and CrkJ [72]. Formally:

$$\mathcal{L}_1(G(t_i)) = \left\langle (E(t_j), (deg_b(t_j), J_b(t_j))_{b \in \mathcal{B}_j})_{j \in \{i-1..i\}} \right\rangle$$

4.3 $\mathcal{L}_2$: Oblivious Lookups

Hint: For every tuple lookup, leak only the volume pattern.

$\mathcal{L}_2$ introduces oblivious data access, leaking only the number of join matches (i.e., tuple degree) per tuple at insertion time, without revealing which tuples matched. Match identities, join indices, and correlations across batches remain private. Figure 1c visualizes it as edges not incident to any vertex. This leakage profile commonly arises in ORAM-based systems [28, 31, 33, 43, 79, 107]. Formally:

$$\mathcal{L}_2(G(t_i)) = \left\langle |E(t_i)|, (deg_b(t_i))_{b \in \mathcal{B}_i} \right\rangle$$

4.4 $\mathcal{L}_3$: Oblivious Batching

Hint: Leak intermediate output cardinality per micro-batch.

$\mathcal{L}_3$ further reduces leakage by revealing only the total number of join results per batch, instead of per tuple. While individual tuple contributions remain hidden, the aggregate output volume is exposed to the adversary. Note that $\mathcal{L}_3$ is easily tunable: for a single-tuple batch, it becomes $\mathcal{L}_2$, while when padding the result to worst-case cardinality, it achieves full obliviousness. This profile is typically assumed in static oblivious joins [64, 74]. Formally:

$$\mathcal{L}_3(G(t_i)) = \left\langle |E(t_i)|, deg_{\mathcal{B}_i}(t_i) \right\rangle$$

where $deg_{\mathcal{B}_i}(t_i) = \sum_{b \in \mathcal{B}_i} deg_b(t_i)$.

4.5 $\mathcal{L}_4$: Full Obliviousness

Hint: Leak no information beyond public metadata.

$\mathcal{L}_4$ represents the strongest security level with full data and access patterns obliviousness. The adversary learns nothing about the join graph beyond public parameters specified in Section 2.2. Achieving $\mathcal{L}_4$ requires worst-case padding and deterministic, data-independent execution. Existing instances of $\mathcal{L}_4$ include Secrecy's nested-loop join [67] and Opaque's FK join [122]. Formally:

$$\mathcal{L}_4(G(t_i)) = \emptyset$$

5 ORAM-BASED SYMMETRIC HASH JOINS

Modern SPSs universally adopt SHJ [11, 25, 121], making it the natural starting point for privacy-preserving equivalents. Our design goal is to enhance the existing stream join approach with obliviousnes. We contribute five SHJ-style algorithms: $\mathcal{L}_0/\mathcal{L}_1$-secure algorithms (Section 5.1), followed by ORAM-based $\mathcal{L}_2$-secure (Section 5.2) and $\mathcal{L}_3/\mathcal{L}_4$-secure joins (Section 5.3).

5.1 $\mathcal{L}_0$- and $\mathcal{L}_1$-Secure Symmetric Hash Joins

$\mathcal{L}_0$- and $\mathcal{L}_1$-secure algorithms are widely adopted in commercial systems [14, 80] and academic prototypes [20, 104, 111], e.g., CryptDB [90] ($\mathcal{L}_0$) and Always Encrypted [10] ($\mathcal{L}_0/\mathcal{L}_1$). Despite numerous attacks proving them insecure [26, 50, 51, 56, 61, 63, 81], cloud providers continue to offer them in their portfolios [77].

We implement *SHJ-L0* and *SHJ-L1*, two stream joins that match the guarantees of existing systems and establish performance baselines akin to current industry practices. While identical in the high-level structure, they differ in implementation and deployment. Both algorithms follow the canonical SHJ procedure (see Section 2.1) for unbounded streams [58]. They: (i) search the opposite stream's window for matching tuples, (ii) insert the tuple into its window, and (iii) expire old tuples. Since we introduce no significant changes to the original idea, we defer the pseudocode to Appendix [1].

We represent windows as maps (for fast lookups) and min-heaps (for expiration ordering). Specifically, window search performs a map lookup, insertion updates both the map and heap, and expiration executes heap extraction and map deletion. *SHJ-L0* processes DET-encrypted tuples outside trusted hardware. The streams are

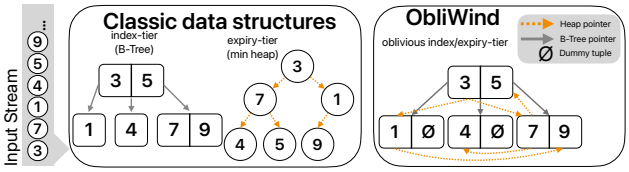

Figure 2: Classic data structures (left) built from an input stream and ObliWind (right) that obliviously combines both.

encrypted with AES-256 with a fixed initialization vector, allowing equality checks. In contrast, *SHJ-L1* processes RND-encrypted tuples inside a TEE, eliminating historical correlations.

Security. *SHJ-L0* with DET-encryption enables the adversary to observe the full join graph and correlate current tuples with expired ones, consistent with $\mathcal{L}_0$ leakage. *SHJ-L1*, running inside TEEs with RND encryption, leaks the current join graph and prevents correlating it with historical data, matching $\mathcal{L}_1$ leakage. Formal simulation-based proofs are provided in Appendix [1].

Performance. The dominant cost in both joins arises from heap extraction, which incurs $O(logN)$ time per tuple for window size N. Consequently, both joins achieve $O(logN)$ complexity per tuple.

5.2 $\mathcal{L}_2$-Secure Symmetric Hash Join

$\mathcal{L}_2$-security hides access patterns for individual tuples. To achieve this, we must go beyond data encryption and protect data structures that manage the stream windows. Previous work addresses a different threat model and requires full rebuilding per batch [31]. This motivates a tuple-oblivious variant of SHJ, which we refer to as *SHJ-L2*. *SHJ-L2* follows the SHJ structure (as in the previous section), but replaces classic data structures (maps and heaps) with their oblivious, ORAM-based equivalents. This modification, although conceptually minor, substantially changes the security profile: instead of leaking full access patterns ($\mathcal{L}_0$-$\mathcal{L}_1$), *SHJ-L2* leaks only the degree of a tuple at insertion time, aligning with $\mathcal{L}_2$.

Prior oblivious maps [28, 79, 107] lack critical features required by sliding windows: efficient deletions and timestamp ordering. First, tuple deletion in existing ORAM-based maps is either prohibitively expensive [107] or not supported [28, 79]. Second, tuples expire based on arrival order (e.g., via timestamps), which is not natively supported by existing maps.

To address these challenges, we introduce ObliWind, a novel doubly-oblivious data structure specifically designed for sliding stream windows. ObliWind (Figure 2) is a composite, two-tier data structure; It consists of an *index tier* for lookups and duplicate placement, and an *expiry tier* for time-ordered eviction. The key idea of the index tier is an oblivious B-Tree [115] atop Path-DORAM (similar to AVL tree in Oblix [79]). On the other hand, the expiry tier is a doubly-oblivious min-heap. This design enables ObliWind to perform all sliding window operations (i.e., search, insert, and expire) obliviously and supports efficient deletions through B-Tree's improved amortized performance (over AVL trees [28, 79, 107] with higher rebalancing costs). While join keys can have duplicates (e.g., many orders match a single customer), we need to be able to uniquely identify any tuple so that it can expire at the right time. Therefore, a tuple $< \tau, k, p >$ (timestamp, key, payload, respectively) is stored under a composite key $< k, \tau >$. In addition, by placing the timestamp second in the composite key, we ensure that tuples stored in a sorted tree (i.e., B-Tree) are sorted by key and timestamp. Thus, ObliWind addresses all the needs of oblivious stream windows , enabling ad-hoc updates without full rebuilt per batch (as in [31]).

We implement the *index tier* as a doubly-oblivious B-Tree [115] atop Path-ORAM [101] to support secure index operations. We ensure all operations are oblivious, including tree rebalancing. Similar to Oblix [79], we extend the B-Tree with order-statistic tree [34] techniques to obliviously count duplicate elements, augmenting each B-Tree node with its subtree size. We modify insertion and deletion such that the size information remains consistent across operations. These modifications are minor and do not affect ObliWind's security. Similarly, we use a doubly-oblivious version of Path Oblivious Heap (POH) [98] for the *expiry tier*. POH supports all classic heap operations. We order the elements in POH by timestamp, enabling efficient and private search of the oldest element.

Below, we describe all the supported operations in ObliWind.

- $search(k, \tau)$: finds an element with key k and timestamp τ. It executes one oblivious index read. If $\tau = 0$, it returns the oldest element with k (i.e., with the smallest timestamp).
- $size(k)$: returns the number of elements with key k. We run two oblivious index reads: with k and $\tau = 0$ (finds the oldest element with k), and with $k + 1$ and $\tau = 0$ (finds the oldest element with *next* key greater than k). The difference in their indices is the number of elements with k.
- $insert(\tau, k, p)$: inserts a new element with a composite key $< k, \tau >$ into the index and heap. B-Tree insertion ensures the tree's correctness by adjusting the size information in visited nodes and obliviously rebalancing the tree.
- $expire()$: extracts the minimum element using heap pointers and deletes it from the index. Deletion readjusts the size information across nodes and rebalances the B-Tree.

We implement *SHJ-L2* inside an Intel SGXv2 enclave with ObliWind stream windows. Notably, ObliWind integrates an in-enclave ORAM server. This change removes the need for TEE-client-server encryption, as all data is now stored in EPC. Additionally, all operations perform fully within the enclave, avoiding costly transitions. Although this increases the TCB, in-enclave components are a standard practice in secure systems [10, 91, 122] and have no security implications. Compared to Omix++ [28], a recent AVL-based oblivious map, ObliWind improves search performance by 51%, while significantly expanding functionality.

Security. All access patterns in ObliWind are protected through a doubly-oblivious design. *SHJ-L2* leaks only the tuple's degree at insertion time (i.e., how many matches it produces), which aligns with $\mathcal{L}_2$. See Appendix [1] for a formal proof.

Performance. All ObliWind operations incur an $O(\log N)$ cost, and while ORAM-stored, this compounds to $O(\log^2 N)$ per tuple.

5.3 $\mathcal{L}_3$- and $\mathcal{L}_4$-Secure Symmetric Hash Joins

While ORAM indices enabled tuple-level obliviousness ($\mathcal{L}_2$), their point query design limits them for batch processing. This raises a central question: how to leverage index-based joins and achieve batch-level ($\mathcal{L}_3$) or full ($\mathcal{L}_4$) obliviousness without revealing intermediate results and without new ORAM constructions?

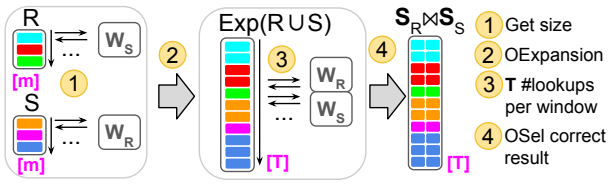

Figure 3: *SHJ-L3* obliviously multiplies index queries to achieve $\mathcal{L}_3$. Collection sizes (in purple) are publicly known.

We propose a novel strategy that hides the tuple-level volume pattern of point queries. Instead of retrieving all tuple join matches in a single query, which reveals the per-tuple degree, we perform a publicly-known number of single-tuple index queries, matching the final join output size. This hides the contributions of a single tuple, thereby enhancing privacy of existing solutions [31].

The core idea is a three-phase procedure (Figure 3). First, we compute the join cardinality per tuple in the input batches R and S. Specifically, we query each tuple's opposite window for the size of its key (1). We aggregate these values into a public total T, which determines the number of index lookups. Second, we obliviously expand the input batches into an array of length T (2). Tuples are duplicated according to their join size (e.g., a tuple with three matches appears three times). This operation, denoted OExpansion, is a well-studied oblivious operator [64, 74] that prevents from learning individual tuple degrees. Finally, for each tuple in the expanded array, we perform one lookup in both windows (3). The correct join match is selected using OSel [28], which obliviously selects the correct tuple and emits the result (4). After execution, we insert the batches into the windows and expire old tuples. Similar to the previous section, we represent windows with ObliWind.

Algorithm 2 shows the pseudocode of *SHJ-L3*. Its key novelty sits in hiding the intermediate results of index queries. First, we determine each the output cardinality of each tuple (lines 4-9). Then, we expand the input (line 11) and query both windows T times while obliviously selecting the join results (lines 12-16). Finally, we retire tuples using procedures from the previous section.

To achieve $\mathcal{L}_4$, we extend this approach with worst-case padding. In line 10, we pad the total number of window lookups T to the maximum join output size (i.e., the window size per input tuple). This ensures that no information is leaked, including the total join cardinality. We call the resulting algorithm *SHJ-L4*.

Security. ObliWind handles all window operations: doubly-oblivious search, insertion, expiration, and size. *SHJ-L3* leaks the join result size per batch, while hiding per-tuple access patterns, conforming with $\mathcal{L}_3$. *SHJ-L4* hides the output size via worst-case padding, achieving $\mathcal{L}_4$-security. Formal proofs are provided in Appendix [1].

Performance. The dominant cost comes from ObliWind, with each operation incurring $O(\log^2 N)$ complexity. For output size T, *SHJ-L3* completes in $O(T \log^2 N)$, and *SHJ-L4* in $O(N \log^2 N)$.

6 OBLIVIOUS COMPUTATION APPROACH JOINS

Previosuly, we showed that achieving all leakage profiles via oblivious indices is possible. However, these algorithms carried a heavy performance baggage - the underlying ORAM. Even the most efficient enclave ORAMs [28, 107] are unsuitable for streaming due to high latency and low scalability in terms of collection size. Moreover, while fully oblivious non-FK joins pad to a prohibitive quadratic size, worst-case padding becomes acceptable in FK joins (Figure 1d-e), where the complexity is substantially reduced. In this section, we propose non-ORAM FK joins that exploit the semantical assumptions, as well as their extensions to non-FK schemas. We propose Oblivious Computation Approach (OCA) $\mathcal{L}_3$- and $\mathcal{L}_4$-secure algorithms (Section 6.1), followed by a $\mathcal{L}_2$-secure join (Section 6.2), and extensions to non-FK stream joins (Section 6.3).

Algorithm 2 Pseudocode of *SHJ-L3* and *SHJ-L4*.

Input: batches R and S of m tuples per stream
Output: join results without/with padding

```
1:  procedure Join(tuples R, tuples S):
2:      T = 0; sizes ← ∅
3:      W_R.insert(R)
4:      for i ∈ [0, m − 1] do
5:          sizes[i] ← W_S.size(R[i])
6:          T += sizes[i]
7:      for i ∈ [0, m − 1] do
8:          sizes[i + m] ← W_R.size(S[i])
9:          T += sizes[i + m]
10:     WorstCasePadding(sizes, T)
11:     res.1 ← OExpansion(R ∪ S, sizes)
12:     for i ∈ [0..T − 1] do
13:         p ← res[i].1
14:         p_1 ← W_R.search(p)
15:         p_2 ← W_S.search(p)
16:         res[i].2 ← OSel(p.tid, p_1, p_2)
17:     W_S.insert(S)
18:     W_R.expire()
19:     W_S.expire()
20:     return res
```

6.1 $\mathcal{L}_3$- and $\mathcal{L}_4$-Secure OCA FK Joins

The existing oblivious static joins rely on (multiple) oblivious sorts of the entire dataset [15, 64, 122] or full rebuilding [31]. Despite recent advancements [17, 52, 83], oblivious sorting remains a performance barrier. Moreover, in streaming, almost all tuples in a window are already sorted from the previous batch. Thus, to achieve high performance in streaming, we must avoid full window sorts.

We observed that representing windows as sorted collections avoids the costly sorting process. In a FK join, having the windows sorted allows us to proceed directly to the linear scan of the windows and join the primary key with foreign key records (similar to Opaque [122]). Therefore, the challenge is how to efficiently and obliviously maintain a sorted collection in a micro-batch scenario.

We propose OAppend (Algorithm 3) that obliviously appends a batch to a sorted collection while preserving its order. Our goal is to reduce the complexity of a full oblivious sort [122] and avoid full rebuilding (as in [31]). First, we obliviously sort the micro-batch; Its cost is negligible as the batch size is small (i.e., fits in L1 cache). Then, we pad it with dummy infinity tuples such that the batch and

Algorithm 3 OAppend operation.

Input: batch T of m tuples, sorted collection C of size n
Output: sorted collection of $(n+m)$ tuples

1: **procedure** OAppend(tuples T, array C):
2: OSort(T)
3: $T_{exp} \leftarrow$ Padd(T, n)
4: $C \leftarrow$ OMerge(C, T_{exp}) ▷ $O(2nlog2n)$
5: **return** $C[0..m+n)$

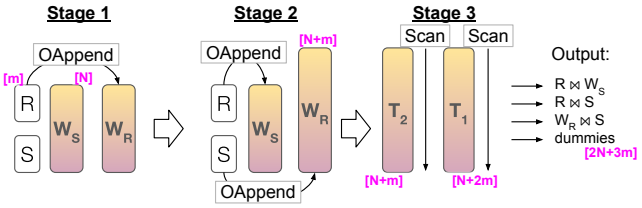

Figure 4: *FK-MERG-L4* obliviously joins unsorted (white) batches with sorted (colored) windows. Purple text indicates size of a collection.

window are both sorted and of equal sizes. In this setup, we can utilize an oblivious merge [21], a simpler (i.e., $O(N \log N)$) primitive that merges two sorted sequences into a sorted collection. Therefore, we obliviously merge the batch with the window and trim the dummy records. We prove its obliviousness in Appendix [1].

Using OAppend, we now maintain windows as sorted collections. With that, we propose *FK-MERG-L3* and *FK-MERG-L4*, $\mathcal{L}_3$-, and $\mathcal{L}_4$-secure FK stream join algorithms. The high-level intuition is to append the input batch to opposite stream's sorted window and obtain the join matches with a full sequential scan. With every scanned tuple, we emit a join match or a dummy tuple. Our algorithm improves Opaque's static join [122] complexity to $O((N \log N)/m)$, while adapting to unbounded environments. Note that the result is padded to the FK worst-case, achieving $\mathcal{L}_4$-security.

We further optimize the algorithms by performing two smaller joins with worst-case padding. The key idea is to join both batches with windows from the opposite streams separately. Thus, our join becomes: $((R \cup \mathcal{W}_R) \bowtie S) \cup (R \bowtie \mathcal{W}_S)$, where R is the PK-batch and S is the FK-batch. We benefit twofold from this optimization; We perform two smaller joins and achieve result uniqueness.

Figure 4 visualizes how we achieve worst-case padding and result uniqueness. In Stage 1, we obliviously append batch R to $\mathcal{W}_R$. Next, we append R and S to their opposite windows, resulting in T_1 and T_2. Finally, we sequentially scan both collections while emitting join matches. All intermediate sizes (purple text) are publicly known.

Algorithm 4 details *FK-MERG-L3* and *FK-MERG-L4*. We begin by appending batch R to its window (line 2). Next, we join batch S with $\mathcal{W}_R$ by merging them into T_1 (line 3) and performing a Scan on T_1 (line 4). Scan sequentially scans a collection, while tracking the last PK-tuple and joining it with the current FK-tuple (similar to [122]). For each scanned tuple, it emits a join match or a dummy tuple. Then, we perform the second of the smaller joins in lines 5-6. Finally, we append S to its window (line 7) and retire old tuples from both windows. We implement ORetire to expire tuples from

Algorithm 4 Pseudocode of *FK-MERG-L3* and *FK-MERG-L4* stream joins.

Input: batch of m tuples per stream
Output: join results

1: **procedure** *FK-MERG-L4*(tuples R, tuples S):
2: $\mathcal{W}_R \leftarrow$ OAppend($R, \mathcal{W}_R$)
3: $T_1 \leftarrow$ OAppend($S, \mathcal{W}_R$)
4: res = Scan(T_1)
5: T_2 = OAppend($R, \mathcal{W}_S$)
6: $res.append$(Scan(T_2))
7: $\mathcal{W}_S \leftarrow$ OAppend($S, \mathcal{W}_S$)
8: ORetire($\mathcal{W}_R$)
9: ORetire($\mathcal{W}_S$)
10: OCompaction(res)
11: **return** res

windows by: (i) marking old tuples via sequential scan based on the timestamp, (ii) obliviously compacting the windows on the marked bit, and (iii) trimming old tuples. Additionally, *FK-MERG-L3* compacts the result to remove dummy tuples (line 10). The order of operations is essential to maintain the result uniqueness.

Security. Both algorithms utilize OAppend, oblivious merge, sort, and sequential scan, all proven secure. Two smaller joins leak no information thanks to worst-case padding. With compaction, the leakage includes total volume pattern $|E|$ and volume pattern per batch deg_B, matching $\mathcal{L}_3$. Without compaction, the algorithm leaks no information, which matches $\mathcal{L}_4$. Hence, *FK-MERG-L4* is $\mathcal{L}_4$-secure and *FK-MERG-L3* is $\mathcal{L}_3$-secure. Formal proofs are in Appendix [1].

Performance. The sorting cost of an m-tuple batch is negligible. Other operations are $O(NlogN)$ for N-sized windows, bringing the complexity of both algorithms to $O(NlogN/m)$ per tuple.

6.2 $\mathcal{L}_2$-Secure OCA FK Join

In oblivious computation, data is constantly relocated to avoid correlations across operations. Yet, a single data access reveals no access patterns. Therefore, we can define ephemeral (i.e., one-time) data structures and access all their records exactly once without leaking any patterns. Further, Asharov et al. [17] observed that an oblivious shuffle breaks cross-execution correlations. This opens an exciting opportunity: an oblivious shuffle and a non-oblivious procedure yields an oblivious algorithm (e.g., sorting [17]). These two observations, oblivious shuffling and rebuilding ephemeral structures, indicate a strong periodicity also seen in the streaming environment: While the former provides a "fresh start", the latter allows for one-time obliviousness. This "access-once-before-reshuffling" property is similar to square-root and hierarchical ORAMs [47].

We propose *FK-EPHI-L2* ("foreign-key-ephemeral-index"), a micro-batch, $\mathcal{L}_2$-secure FK stream join algorithm. Its core idea is a three-step procedure. First, obliviously shuffle both windows with the incoming batches. Second, build an ephemeral, non-oblivious hash table on the FK-relation (i.e., with duplicates). Third, probe it with unique requests from the primary-key relation (i.e., without duplicates), collect the join matches, and destroy the index. The remaining part of the algorithm requires to deduplicating join matches and retiring old records obliviously .

Algorithm 5 shows the pseudocode of *FK-EPHI-L2*. We begin by appending batches to their windows and obliviously shuffling both collections (lines 2-5). Next, we build a non-oblivious hash table on $\mathcal{W}_S$ (line 6) and probe it with unique records from $\mathcal{W}_R$ (lines 8-9). We collect the matches in collection *res*. Similar to Apache Spark [16], we deduplicate join results (line 10). We implement OSelection for deduplication. It works similarly to ORetire (see Section 6.1) using timestamps to mark new (i.e., to-be-emitted) records. Finally, *FK-EPHI-L2* retires old tuples using ORetire (lines 11-12).
Security. All operations in *FK-EPHI-L2* are oblivious, excluding index building and probing. However, index accesses are distinct, leaking only the volume pattern per access. There are no repeated access patterns that leak correlations between tuples. Hence, we leak the degree of each tuple but no cross-batch or cross-tuple correlations, fitting $\mathcal{L}_2$. Formal proof can be found in Appendix [1].
Performance. Shuffle/compaction incurs the highest cost, at $O(N \log N/m)$ per tuple for an N-sized window and m-sized batch.

Algorithm 5 Pseudocode of *FK-EPHI-L2* stream join.

Input: batch of m tuples per stream
Output: join results

```
1:  procedure FK-EPHI-L2(tuples R, tuples S):
2:      W_R.insert(R)
3:      W_S.insert(S)
4:      OShuffle(W_R)
5:      OShuffle(W_S)
6:      ht ← NonObliviousHashTableSetup(W_S)
7:      res ← ∅
8:      for t ∈ W_R do
9:          res.append(ht.search(t))
10:     res ← OSelection(res)
11:     ORetire(W_R)
12:     ORetire(W_S)
13:     return res
```

6.3 OCA Non-FK Joins

Non-FK oblivious joins are more complex as they lack cardinality bounds lower than the Cartesian product. Without ORAM, the strawman oblivious, generic stream join involves a full window join for per batch. Thus, we now face the problem of generic static oblivious join [15] executed periodically, requiring result uniqueness. This introduces a new challenge as the existing works [15, 31, 64] are incapable of tracking new results after an incremental change.

We propose *NFK-JOIN-L3*, the first non-ORAM doubly-oblivious generic stream join algorithm. The main idea is a full oblivious join between windows while tracking tuples from the most recent batch. The key insight is to prune the results and emit only new matches.

To achieve this goal, we modify an oblivious static join to streaming. In detail, we mark tuples from the recent batch and append them to their respective windows. We then obliviously join the windows and, using the mark, determine if a join match should be emitted; We only emit a match if it contains at least one marked tuple. Effectively, we still perform a full join on both windows, but during the join process, we trim redundant tuples.

We base our implementation on the state-of-the-art oblivious static join introduced by Krastnikov et al. [64]. Precisely, we modify its group dimension calculation stage; After concatenating and sorting windows, we perform a downward scan and mark tuples from $\mathcal{W}_S$ that join with at least one new tuple from $\mathcal{W}_R$. Next, we do the opposite - we perform an upward scan and mark tuples from $\mathcal{W}_R$ that join with at least one new tuple from $\mathcal{W}_S$. We now proceed to the original dimension calculation but increase the group dimensions for tuples that have been marked in the previous scans (i.e., that potentially contribute a new result tuple). From this point, we continue with the remainder of the original algorithm. At the end of the join algorithm, we have collected the join matches where at least one of the tuples comes from the most recent batch.
Security. *NFK-JOIN-L3* is based on an already-proven $\mathcal{L}_3$-secure join [64]. The modification adds two full sequential scans with no leakage. All remaining operations are oblivious, matching $\mathcal{L}_3$.
Performance. *NFK-JOIN-L3* has $O(N \log^2 N/m)$ complexity per tuple, with oblivious sort being the dominating factor.

7 EXPERIMENTAL EVALUATION

The primary goal of the algorithms is to perform stream join queries under formally defined leakage profiles efficiently . We formally proved the security goal in the previous sections. We now empirically evaluate their performance. First, we estimate the performance cost associated with each leakage profile (Section 7.2). Second, we compare the performance of FK joins (Section 7.3) and Non-FK joins (Section 7.4). For each experiment set, we manipulate the key stream parameters, i.e., batch size and window size.

7.1 Setup

Platform. We ran all experiments on a machine with 2 x Intel Xeon 4416+ CPUs, 128 GB EPC, and 188 GB RAM. It ran Ubuntu 22.04.3 LTS OS with SGX SDK 2.24.100 and gcc 11.4.0.. We implemented all algorithms in C++ as a standalone library that isolates oblivious operations, enabling tight security control and precise performance measurements without production overhead,e.g., when integrating into systems like Apache Flink [25]. We conducted the experiments using TeeBench [73] with stream query support. While out of scope, the compiled code should be externally verified [64, 96].
Baselines. Despite a rich history of secure static joins (Table 1), to the best of our knowledge, there exist no direct competitors for oblivious stream joins. Thus, our core baselines are *SHJ* and *NLJ-L4*, which delineate the performance-privacy tradeoff, from high performance and no privacy to strong privacy and low performance. Recall, that *NLJ-L4* is a strawman oblivious solution enabling indirect comparison to MPC nested-loop joins [22, 67, 123]. Direct comparisons with other MPC works [116] and *trusted proxy* [31] solutions yield unfair results due to differences in communication, cryptographic primitives, and security assumptions. To further emphasize the improvements of our algorithms, we implemented an additional *SORT* family of oblivious stream FK joins. The main naive idea is to obliviously sort entire windows per join invocation and sequentially scan the sorted collections while emitting join matches. These algorithms resemble Opaque's static join [122] and

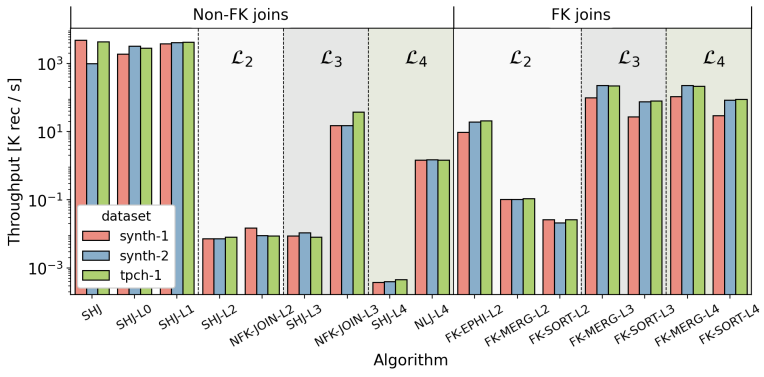

Figure 5: The cost of oblivious stream joins.

Table 2: Memory consumption and average latency of stream joins for dataset *synth-1*.

Join	Algorithm	Peak memory usage [MB]	Latency [μs/tuple]
Non-Foreign Key	*SHJ*	28	4
	SHJ-L0	53	6
	SHJ-L1	29	3
	SHJ-L2	1798	$132 * 10^3$
	SHJ-L3	1798	$151 * 10^3$
	SHJ-L4	1798	$3554 * 10^3$
	NFK-JOIN-L2	30	$67 * 10^3$
	NFK-JOIN-L3	30	66
	NLJ-L4	16	613
Foreign Key	*FK-EPHI-L2*	33	226
	FK-MERG-L2	50	$9 * 10^3$
	FK-MERG-L3	50	18
	FK-MERG-L4	50	13
	FK-SORT-L2	48	$32 * 10^3$
	FK-SORT-L3	48	32
	FK-SORT-L4	48	29

are the baselines closest to prior work. We refer to them as *FK-SORT-L4* and *FK-SORT-L3*. Similar to *MERG* algorithms, *FK-SORT-L4* achieves $\mathcal{L}_4$-security. *FK-SORT-L3* trims the dummy tuples via oblivious compaction, achieving $\mathcal{L}_3$-security. Finally, we introduce two $\mathcal{L}_2$ baselines: *FK-MERG-L2* and *FK-SORT-L2*, single-tuple batch versions of their $\mathcal{L}_3$ equivalents. Both algorithms leak per-tuple degree, obtaining $\mathcal{L}_2$-security. Our main performance metric is the maximum sustainable throughput [60]. Since ORAM-based joins are not easily parallelizable, we report single-threaded performance. We abort experiments that exceed 6h.

Datasets. By definition of oblivious computation, the performance and execution patterns depend solely on data size and are indistinguishable across inputs. Thus, any real and synthetic datasets of the same size will perform equally in performance and privacy [74]. Similar to related streaming work [37, 46, 94, 100, 105], we used two synthetic datasets: *synth-1* and *synth-2*. *synth-1* consists of two symmetric streams with 1000 [tuples/s] rates, and *synth-2* of two asymmetric streams with 1000 and 4000 [tuples/s] rates. As in [68], we generated TPC-H data to form *tpch-1* dataset, which joins the *customer* and *orders* tables on *c_custkey*. All tuples have the *<timestamp,key,payload>* format. We generated datasets with FK relationships for all FK joins [9]. Default parameters include a 2^{16}-tuple window size and a 1-second batch size (as in [16]). While OCA joins support larger states, we use smaller windows to enable comparison with ORAM-based joins. All runs are warmed up by filling up the windows to avoid the startup effect (i.e., empty windows not retiring old tuples). If results are consistent across datasets, for clarity, we report single-dataset numbers.

Implementation. *SHJ*, *SHJ-L0*, and *SHJ-L1* use bucket-chaining hash tables. *SHJ-L2*, *SHJ-L3*, and *SHJ-L4* store tuples in ObliWind. We implemented ObliWind as an order-statistic oblivious B-Tree multimap on top of Path-ORAM [101]. Additionally, we utilized oblivious expansion from [64]. The OCA joins employ modified oblivious sort, shuffle, and compaction from [83], ObliviousMerge with bitonic merge [21], and standard library map for *FK-EPHI-L2*.

7.2 Privacy Cost in Oblivious Stream Joins

Given the absence of prior oblivious stream joins, a central question we address is: *What is the performance cost associated with each leakage level?* Based on related work [28, 64, 122], we expect ORAM-based approaches to underperform, whereas OCA joins will offer competitive throughput. To evaluate this, we benchmark the throughput of all proposed algorithms across three test datasets.

Figure 5 highlights three key findings, which align with our intuition. First, basic encryption ($\mathcal{L}_0$ and $\mathcal{L}_1$) incurs negligible overhead, with *SHJ-L0* and *SHJ-L1* matching native CPU performance. Second, the OCA FK joins achieve high performance across all leakages. For example, *FK-MERG-L4* is only 4.3× slower than *SHJ* on *synth-2*, while ensuring full confidentiality and obliviousness. Third, while ORAM-based joins (*SHJ-L2* to *SHJ-L4*) support generic joins, they are 3-6 orders of magnitude slower than OCA joins and 6-8 orders of magnitude slower than insecure *SHJ*, due to costly ORAM access. Thus, ORAM's generality comes at a prohibitive performance cost.

Next, we evaluated peak memory usage (via OS and *sgx-gdb*) and per-tuple latency for all algorithms. For tuple-at-a-time modes, latency reflects the time from join invocation to the emission of all matches. For micro-batch modes, it is the batch processing time divided by the batch size (assuming no batch fill delay). Table 2 summarizes the results; similar trends hold across datasets, which are omitted for brevity. Non-ORAM algorithms exhibit moderate memory usage, while ORAM-based joins consume nearly 2 GB even with 2^{16}-tuple windows, indicating poor scalability and posing a challenge for TEEs with limited secure memory (e.g., 8 GB in some SGXv2 [55]). Latency results reveal a similar pattern: *FK-MERG-L4* achieves 13μs per tuple, whereas oblivious SHJ joins incur $0.13 - 3.5s$ – far too slow for real-time processing. Additionally, *FK-MERG-L2* and *FK-SORT-L2* perform poorly in tuple-at-a-time mode, reinforcing that these designs are better suited to micro-batching.

OCA algorithms outperform ORAM-based baselines by 3-6 orders of magnitude, while offering low latency.

7.3 Oblivious Foreign Key Stream Joins

We evaluate FK joins, a common streaming use case [9], by varying window and batch sizes to assess robustness across diverse scenarios.. Figures 6a-c show throughput as a function of window size, revealing several findings.

The *MERG* family consistently outperforms *SORT* across all leakage levels, e.g., *FK-MERG-L4* achieves 4.6× higher throughput than *FK-SORT-L4*. This confirms OAppend's efficiency, which replaces an oblivious sort with two faster oblivious merges. Micro-benchmarks

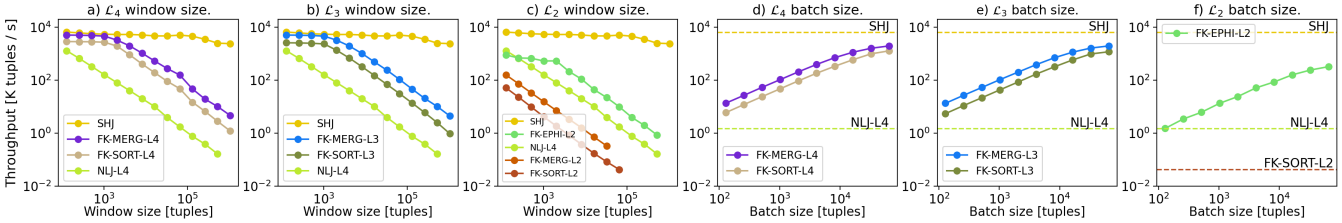

Figure 6: Performance of FK stream join algorithms for $\mathcal{L}_2$-$\mathcal{L}_4$ leakages as a function of (a-c) window size and (d-f) batch size.

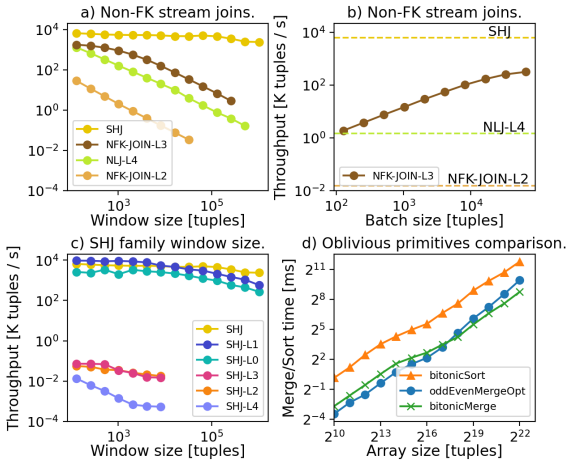

Figure 7: (a-c) Performance of non-FK stream join algorithms. (d) Oblivious merge and sort performance.

(Figure 7d) show that Odd-Even and Bitonic Merge consistently outperform Bitonic Sort [21] by > 2×, supporting this design choice.

FK-MERG-L4 outperforms *FK-MERG-L3* despite stronger privacy, due to the latter's compaction overhead, resembling prior work (Opaque [122]), where stronger privacy yielded better performance.

Among $\mathcal{L}_2$-secure joins, *FK-EPHI-L2* achieves the highest throughput. *FK-MERG-L2* and *FK-SORT-L2*, built for micro-batches, significantly underperform in tuple-at-a-time mode. Still, *FK-EPHI-L2* lags $\mathcal{L}_3$ joins by an order of magnitude due to costly index construction, questioning when $\mathcal{L}_2$ is justified. Given weaker privacy and lower performance, stronger leakage profiles appear preferable.

Finally, we examined the effect of increasing batch size. Intuitively, it improves throughput as OAppend pads with fewer dummy tuples instead processing real ones. However, large batches can hurt responsiveness by increasing the latency of some tuples. Figures 6d-f confirm our intuition. Increasing batch size improves throughput across all algorithms. Notably, *FK-MERG-L4* achieves 450× higher throughput than *NLJ-L4* and is within 3.5× of *SHJ*.

FK-MERG-L4 achieves the highest privacy and performance among FK joins with only a 3.5× slowdown compared to *SHJ*.

7.4 Oblivious Non-Foreign Key Stream Joins

We now evaluate oblivious non-FK joins to quantify the cost of generalizing beyond FK constraints. Figures 7a-c report throughput across window and batch sizes for the *SHJ* and *JOIN* families.

NFK-JOIN-L3 avoids ORAM's performance penalty, while *NFK-JOIN-L2* upholds previous $\mathcal{L}_2$ findings (Figure 7a). Comparing across families and join types, *NFK-JOIN-L3* performs 3-10× worse than *FK-MERG-L3*, underscoring FK efficiency gains. Surprisingly, even naive *NLJ-L4* outperforms oblivious *SHJ* joins, highlighting the severity of ORAM overhead. Lastly, *NFK-JOIN-L3* benefits from larger batches (Figure 7b), illustrating the latency-throughput tradeoff. Results for oblivious *SHJ* joins for large batch sizes are omitted due to infeasible runtimes.

Figure 7c illustrates that non-oblivious *SHJ-L0* and *SHJ-L1* maintain stable performance due to constant-time index operations, closely matching insecure *SHJ* with minimal encryption overhead. In contrast, oblivious variants (*SHJ-L2/L3/L4*) degrade severely as the window size increases due to expensive ORAM access. Recall that the SHJ family does not aim at improving ORAM's performance. Despite ObliWind's optimizations, ORAM-based joins remain unsuitable for latency-sensitive workloads. Additionally, the gap between *SHJ-L3* and *SHJ-L4* shows the impact of worst-case padding. Overall, ORAM's performance penalty renders it impractical.

NFK-JOIN-L3 pays an order of magnitude performance price for enabling generic stream joins compared to FK-join algorithms.

8 CONCLUSIONS

The existing SPSs lack privacy protection in untrusted environments and remain vulnerable to access pattern leakage despite using trusted hardware. We have introduced a formal security framework that defines five leakage functions, proposing new privacy-performance tradeoffs for stream join queries. We have contributed two families of stream join algorithms: an index-based (SHJ) approach and a computation-based approach (OCA). Our experiments demonstrate that OCA algorithms outperform ORAM-based joins by several orders of magnitude and operate within an order of magnitude of insecure baselines while offering strong privacy. The results demonstrate that privacy-preserving joins, particularly for foreign-key queries, are practical for real-world applications.

Future Work. While we present fundamental ideas for oblivious stream joins, we left out several promising research directions. First, we can combine privacy-preserving techniques, such as obliviousness with differential privacy. This enables to gradually transition between leakage profiles. Second, our techniques may apply to other stream operators, such as aggregation and group-by. The critical question is which components are reusable. Third, modern SPSs offer advanced features such as out-of-order execution and fault tolerance. It is crucial to support them in an oblivious manner.

REFERENCES

[1] 2025. . https://github.com/kai-chi/obliv-stream-join/blob/main/paper/full_manuscript.pdf

[2] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, et al. 2020. The Seattle report on database research. *ACM Sigmod Record* 48, 4 (2020), 44–53.

[3] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. 2022. Prio+: Privacy preserving aggregate statistics via boolean shares. In *International Conference on Security and Cryptography for Networks*. Springer, 516–539.

[4] Georgii M Adelson-Velskii and Evgenii Mikhailovich Landis. 1962. An algorithm for the organization of information. In *Soviet Mathematics Doklady*, Vol. 3. 4.

[5] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. 2006. Sovereign joins. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 26–26.

[6] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFUSCURO: A commodity obfuscation engine on Intel SGX. In *Network and Distributed System Security Symposium*.

[7] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[8] Miklós Ajtai, János Komlós, and Endre Szemerédi. 1983. An 0 (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 1–9.

[9] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. 2013. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*. 577–588.

[10] Panagiotis Antonopoulos, Arvind Arasu, Kunal D Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, et al. 2020. Azure SQL database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1511–1525.

[11] Apache. 2024. *Apache Storm*. Retrieved October 27, 2024 from https://storm.apache.org/

[12] Apple. 2024. *Apple Platform Security*. Retrieved October 27, 2024 from https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf

[13] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security with Cipherbase.. In *CIDR*.

[14] Arvind Arasu, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, and Ravi Ramamurthy. 2015. Transaction processing on confidential data using cipherbase. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 435–446.

[15] Arvind Arasu and Raghav Kaushik. 2014. Oblivious Query Processing. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*.

[16] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*.

[17] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket oblivious sort: An extremely simple oblivious sort. In *Symposium on Simplicity in Algorithms*. SIAM, 8–14.

[18] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: optimal oblivious RAM. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30*. Springer, 403–432.

[19] H Ceren Ates, Ali K Yetisen, Firat Güder, and Can Dincer. 2021. Wearable devices for the detection of COVID-19. *Nature Electronics* 4, 1 (2021), 13–14.

[20] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*.

[21] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*.

[22] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks. *Proc. VLDB Endow.* 10, 6 (2017), 673–684.

[23] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018).

[24] John Bethencourt, Dawn Song, and Brent Waters. 2009. New techniques for private stream searching. *ACM Transactions on Information and System Security (TISSEC)* 12, 3 (2009), 1–32.

[25] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* (2015).

[26] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*.

[27] Anrin Chakraborti and Radu Sion. 2019. ConcurORAM: High-Throughput Stateless Parallel Multi-Client ORAM. In *NDSS*.

[28] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2023. GraphOS: Towards Oblivious Graph Processing. *Proceedings of the VLDB Endowment* (2023).

[29] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. 2019. Foundations of Differentially Oblivious Algorithms. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*.

[30] T-H Hubert Chan, Kai-Min Chung, Bruce Maggs, and Elaine Shi. 2022. Foundations of differentially oblivious algorithms. *ACM Journal of the ACM (JACM)* (2022).

[31] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards practical oblivious join. In *Proceedings of the 2022 International Conference on Management of Data*. 803–817.

[32] Shumo Chu, Danyang Zhuo, Elaine Shi, and TH Hubert Chan. 2021. Differentially Oblivious Database Joins: Overcoming the Worst-Case Curse of Fully Oblivious Algorithms. In *2nd Conference on Information-Theoretic Cryptography*.

[33] Graeme Connell. 2012. Technology Deep Dive: Building a Faster ORAM Layer for Enclaves. (2012).

[34] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[35] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*. 259–282.

[36] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 727–743.

[37] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. 2003. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 40–51.

[38] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX security symposium (USENIX Security 20)*. 2433–2450.

[39] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. 2002. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 299–310.

[40] Wei Dong, Zijun Chen, Qiyao Luo, Elaine Shi, and Ke Yi. 2024. Continual Observation of Joins under Differential Privacy. *Proceedings of the ACM on Management of Data* (2024).

[41] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N Rothblum. 2010. Differential privacy under continual observation. In *Proceedings of the forty-second ACM symposium on Theory of computing*.

[42] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* (2014).

[43] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: oblivious query processing for secure databases. *Proceedings of the VLDB Endowment* (2019).

[44] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. 2018. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security* (2018).

[45] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2021. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–36.

[46] Buğra Gedik, Rajesh R Bordawekar, and Philip S Yu. 2009. CellJoin: a parallel stream join operator for the cell processor. *The VLDB journal* 18 (2009), 501–519.

[47] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[48] Michael T Goodrich. 2011. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 379–388.

[49] Google. 2024. *Outputs from Azure Stream Analytics*. Retrieved October 22, 2024 from https://cloud.google.com/products/dataflow

[50] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2019. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*.

[51] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. 2016. Breaking web applications built on top of encrypted data.

In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1353–1364.

[52] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. 2023. Efficient Oblivious Sorting and Shuffling for Hardware Enclaves. *Cryptology ePrint Archive* (2023).

[53] Shay Gueron. 2016. A memory encryption engine suitable for general purpose processors. *Cryptology ePrint Archive* (2016).

[54] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1259–1274.

[55] Intel. 2025. *Intel Xeon Silver 4309Y Processor*. Retrieved March 27, 2025 from https://www.intel.com/content/www/us/en/products/sku/215275/intel-xeon-silver-4309y-processor-12m-cache-2-80-ghz/specifications.html

[56] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: ramification, attack and mitigation.. In *NDSS*.

[57] Gabriela Jacques-Silva, Ran Lei, Luwei Cheng, Guoqiang Jerry Chen, Kuen Ching, Tanji Hu, Yuan Mei, Kevin Wilfong, Rithin Shetty, Serhat Yilmaz, et al. 2018. Providing streaming joins as a service at facebook. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1809–1821.

[58] Jaewoo Kang, Jeffrey F Naughton, and Stratis D Viglas. 2003. Evaluating window joins over unbounded streams. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE, 341–352.

[59] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. (2016).

[60] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, 1507–1518.

[61] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[62] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* (2020).

[63] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[64] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient oblivious database joins. *Proceedings of the VLDB Endowment* (2020).

[65] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. 557–574.

[66] Yaping Li and Minghua Chen. 2008. Privacy preserving joins. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 1352–1354.

[67] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. {SECRECY}: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1031–1056.

[68] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 811–825.

[69] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[70] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 359–376.

[71] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. 2023. Waffle: An online oblivious datastore for protecting data access patterns. *Proceedings of the ACM on Management of Data* (2023).

[72] Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. Cracking-Like Join for Trusted Execution Environments. *Proceedings of the VLDB Endowment* (2023).

[73] Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, Jonas Traub, and Volker Markl. 2021. What is the price for joining securely? benchmarking equi-joins in trusted execution environments. *Proceedings of the VLDB Endowment* (2021).

[74] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. 2025. OBLIVIATOR: Oblivious Parallel Joins and other Operators in Shared Memory Environments. *Cryptology ePrint Archive* (2025).

[75] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*.

[76] Meta. 2023. *The Labirynth Encrypted Message Storage Protocol*. Technical Report.

[77] Microsoft. 2016. *Always Encrypted with secure enclaves*. Retrieved November 11, 2024 from https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-enclaves

[78] Microsoft. 2024. *Outputs from Azure Stream Analytics*. Retrieved October 22, 2024 from https://learn.microsoft.com/en-us/azure/stream-analytics/stream-analytics-define-outputs

[79] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 279–296.

[80] MongoDB. 2024. *Queryable Encryption*. Retrieved August 6, 2024 from https://www.mongodb.com/docs/manual/core/queryable-encryption/

[81] Muhammad Naveed, Seny Kamara, and Charles V Wright. 2015. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[82] Netflix. 2016. *Evolution of the Netflix Data Pipeline*. Retrieved November 11, 2024 from https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905

[83] N. Ngai, I. Demertzis, J. Ghareh Chamani, and D. Papadopoulos. 2024. Distributed & Scalable Oblivious Sorting and Shuffling. In *2024 IEEE Symposium on Security and Privacy (SP)*.

[84] Rafail Ostrovsky. 1990. Efficient computation on oblivious RAMs. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 514–523.

[85] Rafail Ostrovsky and William E Skeith III. 2005. Private searching on streaming data. In *Annual International Cryptology Conference*. Springer, 223–240.

[86] Serafeim Papadias, Zoi Kaoudi, Varun Pandey, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2024. Counting butterflies in fully dynamic bipartite graph streams. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2917–2930.

[87] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. 2019. {StreamBox-TZ}: Secure stream analytics at the edge with {TrustZone}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 537–554.

[88] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)* (2019).

[89] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. 2020. Practical volume-based attacks on encrypted databases. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*.

[90] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles*. 85–100.

[91] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.

[92] Lina Qiu, Georgios Kellaris, Nikos Mamoulis, Kobbi Nissim, and George Kollios. 2023. Doquet: Differentially Oblivious Range and Join Queries with Private Data Structures. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4160–4173.

[93] Mayank Rathee, Yuwen Zhang, Henry Corrigan-Gibbs, and Raluca Ada Popa. 2024. Private Analytics via Streaming, Sketching, and Silently Verifiable Proofs. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 194–194.

[94] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency handshake join. *Proceedings of the VLDB Endowment* 7, 9 (2014), 709–720.

[95] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. 2016. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–217.

[96] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast fully oblivious compaction and shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2565–2579.

[97] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*. 1961–1976.

[98] Elaine Shi. 2020. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 842–858.

[99] Apache Spark. 2024. *StreamingSymmetricHashJoinExec*. Retrieved October 22, 2024 from https://github.com/apache/spark/blob/d24393b0b5edd8b2f1a224042c03c36eb7560cff/sql/core/src/main/scala/org/apache/spark/sql/execution/streaming/StreamingSymmetricHashJoinExec.scala#L692

[100] Utkarsh Srivastava and Jennifer Widom. 2004. Memory-limited execution of windowed stream joins. In *VLDB*, Vol. 4. 324–335.

[101] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[102] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 253–267.

[103] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. 2012. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual*

Computer Security Applications Conference. 229–238.
[104] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment* (2021).
[105] Jens Teubner and Rene Mueller. 2011. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 625–636.
[106] TikTok. 2023. *PETAce in Action: Enhancing privacy during Friends Matching in TikTok*. Retrieved October 27, 2024 from https://developers.tiktok.com/blog/tiktok-practices-in-privacy-enhancing-technologies
[107] Afonso Tinoco, Sixiang Gao, and Elaine Shi. 2023. {EnigMap}:{External-Memory} Oblivious Map for Secure Enclaves. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4033–4050.
[108] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*. 1041–1056.
[109] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 137–152.
[110] Juliane Verwiebe, Philipp M Grulich, Jonas Traub, and Volker Markl. 2023. Survey of window types for aggregation in stream processing systems. *The VLDB Journal* 32, 5 (2023), 985–1011.
[111] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a scalable encrypted database with full SQL query support. *Proceedings on Privacy Enhancing Technologies* (2019).
[112] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*.
[113] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation Without Joins. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1046–1058.
[114] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 850–861.
[115] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 215–226.
[116] Yilei Wang and Ke Yi. 2021. Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*.
[117] Yilei Wang, Xiangdong Zeng, Sheng Wang, and Feifei Li. 2025. Jodes: Efficient Oblivious Join in the Distributed Setting. *arXiv preprint arXiv:2501.09334* (2025).
[118] WhatsApp. 2016. *end-to-end encryption*. Retrieved October 27, 2024 from https://blog.whatsapp.com/end-to-end-encryption
[119] Peter Williams, Radu Sion, and Alin Tomescu. 2012. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 977–988.
[120] Xinying Yang, Cong Yue, Wenhui Zhang, Yang Liu, Beng Chin Ooi, and Jianjun Chen. 2024. SecuDB: An In-enclave Privacy-Preserving and Tamper-resistant Relational Database. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3906–3919.
[121] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*.
[122] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.
[123] Peizhao Zhou, Xiaojie Guo, Pinzhi Chen, Tong Li, Siyi Lv, and Zheli Liu. 2024. Shortcut: Making MPC-based Collaborative Analytics Efficient on Dynamic Databases. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 854–868.

A ALGORITHMS

A.1 Symmetric Hash Join

Symmetric Hash Join (SHJ) is a canonical stream join operator widely adopted in modern SPSs. It processes unbounded streams by maintaining two sliding windows (i.e., one per stream) and probing one window with each incoming tuple from the other stream. SHJ (Algorithm 6) processes each incoming tuple in three steps [58]. For each tuple in stream $\mathcal{S}_R$: First, it inserts the tuple to its sliding window (line 3). Second, it probes the opposite window for join matches (line 4). Third, it expires outdated tuples from its window (line 5). Similarly, it processes tuples in $\mathcal{S}_S$ in lines 7-9.

Algorithm 6 Pseudocode of *SHJ-L0*, *SHJ-L1*, and *SHJ-L2*. The first two algorithms protect data with data encryption, while the last join adds window obliviousness.

Input: tuple p
Output: join results
1: **procedure** JOIN(tuple p):
2: **if** $p \in \mathcal{S}_R$ **then**
3: $\mathcal{W}_R.insert(p)$
4: $res \leftarrow (t, \mathcal{W}_S.search(p))$
5: $\mathcal{W}_R.expire()$
6: **else if** $p \in \mathcal{S}_S$ **then**
7: $\mathcal{W}_S.insert(p)$
8: $res \leftarrow (\mathcal{W}_R.search(p), p)$
9: $\mathcal{W}_S.expire()$
10: **return** res

We define three variants of SHJ that follow these three steps without modifications: *SHJ-L0*, *SHJ-L1*, and *SHJ-L2*. Each of the algorithms targets a progressively stronger security guarantee under our leakage taxonomy (Section 4). Their core logic follows the same control flow but differs in window representation, deployment context, and cryptographic protections (see Sections 5.1 and 5.2).

B SECURITY ANALYSIS

We prove that our algorithms are secure according to Definition 1 given a leakage function $\mathcal{L}$. In each proof, we show that for a polynomial-time adversary, there exists a polynomial-time simulator $\mathcal{S}$ such that the real and ideal experiments in Algorithm 1 are computationally indistinguishable. We define a polynomial-time simulator $\mathcal{S}$ that invokes the ideal experiment. $\mathcal{S}$ has access only to public information (as defined in Section 2.2) and the leakage defined by one of the leakage functions (Section 4). The adversary tries to guess if the executed experiment is real or ideal. Our join algorithms are secure if a polynomial-time adversary has no more than negligible probability in guessing the experiment.

B.1 *SHJ-L0* and *SHJ-L1*

THEOREM 1. *Algorithm 6 for SHJ-L0 is $\mathcal{L}_0$-secure according to Def. 1.*

PROOF. Simulator $\mathcal{S}$ executes SIMSHJL$_0$ (Algorithm 7) with $\mathcal{L}_0$ as input. First, it generates a deterministic ciphertext tuple p and window states W_R and W_S that match equality patterns from $\mathcal{L}_0$. It then appends the tuple to its respective window (public information) and produces join matches that match E. Finally, it retires a tuple if necessary, which is also public information. Therefore, $\mathcal{S}$ achieves equality patterns consistent with DET-encryption, exact same join results, and identical window states and access patterns. □

THEOREM 2. *Algorithm 6 for SHJ-L1 is $\mathcal{L}_1$-secure according to Def. 1.*

Algorithm 7 Simulation of *SHJ-L0* and *SHJ-L1*.

```
1:  procedure SimShjL0/1(ℒ):
2:      n be window size parameter
3:      p ← 𝒜(1^λ, ℒ)
4:      𝒲_R, 𝒲_S ← 𝒜(1^λ, ℒ)
5:      if p ∈ 𝒮_R then
6:          𝒲_R.insert(p, c)
7:          res ← {(p, s)|s ∈ W_S ∧ (p, s) ∈ ℒ.E}
8:          if |W_R| > n then
9:              W_R.retire()
10:     else
11:         𝒲_S.insert(p, c)
12:         res ← {(r, p)|r ∈ W_R ∧ (r, p) ∈ ℒ.E}
13:         if |W_S| > n then
14:             W_S.retire()
15:     return res
```

Proof. Simulator $\mathcal{S}$ executes SimShjL$_1$ with $\mathcal{L}_1$ as input. The proof is similar to the previous one. The difference is in the window state generation process. In this case, $\mathcal{L}_1$ does not reveal any patterns with t. Hence, the W_R and W_S are generated without the connection to the historical data. Otherwise, $\mathcal{S}$ successfully reproduces the join matches and the access patterns of *SHJ-L1*. □

B.2 *SHJ-L2*

Theorem 3. *Algorithm 6 for SHJ-L2 is $\mathcal{L}_2$-secure according to Def. 1 assuming a doubly-oblivious hash table supporting oblivious insert, search, and retire operations.*

Proof. $\mathcal{S}$ executes Algorithm 8. First, it generates a random tuple (line 4). It identifies to which stream the tuple belongs, which is a public parameter. Second, it simulates oblivious insertion and oblivious search on respective windows (lines 6-7 and 12-13). Next, it generates deg^t random join matches without revealing which specific tuples match (lines 8 and 14). Finally, it simulates the retire operation on a window using give access patterns. $\mathcal{S}$ is capable of producing the correct number of join results without revealing any additional access patterns. □

Algorithm 8 Simulation of *SHJ-L2*.

```
1:  procedure SimShjL2(ℒ):
2:      Let n be the window size parameter
3:      t ← 𝒜(1^λ, ℒ)
4:      𝒲_R, 𝒲_S ← 𝒜(1^λ, ℒ)
5:      if t ∈ 𝒮_R then
6:          SimOHashTable.insert(ℒ, 𝒲_R)
7:          SimOHashTable.search(ℒ, 𝒲_S)
8:          res ← Generate ℒ.deg^t random matches
9:          if |𝒲_R| > n then
10:             SimOHashTable.retire(ℒ, 𝒲_R)
11:     else
12:         SimOHashTable.insert(ℒ, 𝒲_S)
13:         SimOHashTable.search(ℒ, 𝒲_R)
14:         res ← Generate ℒ.deg^t random matches
15:         if |𝒲_S| > n then
16:             SimOHashTable.retire(ℒ, 𝒲_S)
17:     return res
```

B.3 *SHJ-L3* and *SHJ-L4*

Theorem 4. *The approach introduced in Section 5.2 for selection is oblivious as to Def. 1, assuming a compaction oblivious to Def. 1.*

Proof. Algorithm 9 simulates the selection approach. It performs a scan, an oblivious compaction (proven oblivious in [96]), and truncation. All operations depend only on the input size n and number of rows to discard m (both public information), making SimOSelection oblivious. □

Algorithm 9 Simulation of ObliviousSelection.

```
1:  procedure SimOSelection(ℒ):
2:      Parse ℒ to retrieve m
3:      Randomly mark m rows
4:      T ← SimOCompact(ℒ)
5:      Truncate T to have size m
6:      return T
```

Theorem 5. *Algorithm 2 for SHJ-L3 is $\mathcal{L}_3$-secure according to Def. 1 assuming assuming an oblivious hash table, expansion, selection, conditional select, and compaction.*

Proof. Simulator $\mathcal{S}$ executes SimShjL$_3$ (Algorithm 10). First, it generates *res*, R, and S. Then, it simulates m insertions to an oblivious map (line 9). Next, it loops over each tuple in both batches (lines 10-12 and 13-15), where it simulates size retrieval and stores a random number in *sizes*. Line 16 normalizes *sizes* to *out_size*. In line 17, $\mathcal{S}$ simulates oblivious expansion. In lines 18-21, $\mathcal{S}$ loops over *res* simulating two lookups in the oblivious hash tables and a conditional select. Next, it simulates insertion into the hash table. In line 23, $\mathcal{S}$ simulates oblivious selection (proven oblivious in Theorem 4). None of these leaks access patterns. Lines 23-24 perform two *retire* operations on oblivious hash tables. □

Theorem 6. *Algorithm 2 for SHJ-L4 is $\mathcal{L}_4$-secure according to Def. 1 assuming assuming an oblivious hash table, expansion, conditional select, and compaction.*

Proof. $\mathcal{S}$ executes SimShjL$_4$ (Algorithm 10). The proof is almost the same as of Theorem 5. The differences are in line 4, where *out_size* is initialized to the worst-case output m^2. In line 16, this causes *sizes* to sum up to m^2, and the loop in lines 18-21 to execute m^2 times. None of these operations leak access patterns. □

B.4 *FK-EPHI-L2* FK join

Theorem 7. *Algorithm 5 is $\mathcal{L}_2$-secure according to Def. 1 assuming shuffle, selection and compaction algorithms are oblivious to Def. 1.*

Proof. Simulator $\mathcal{S}$ executes SimOcaL2 in Algorithm 11. $\mathcal{S}$ starts by generating two random batches R and S. It appends the batches to their respective windows (lines 4-5) and obliviously shuffles both windows (lines 6-7). In lines 8-9, it builds a non-comparison hash index on the set of non-unique elements W_S. Next, $\mathcal{S}$ runs a

Algorithm 10 Simulation of *SHJ-L3* and *SHJ-L4*.

1: **procedure** SimShjL$_{3/4}$($\mathcal{L}$):
2: Let n be the window size and m the batch size parameters
3: $\mathcal{W}_R, \mathcal{W}_S$ be the states of the windows
4: $out\_size \leftarrow \mathcal{L}_3.deg^{\mathcal{B}}$
5: $out\_size \leftarrow m^2$
6: $res \leftarrow \mathcal{A}(1^\lambda, out_size)$
7: $R, S \leftarrow \mathcal{A}(1^\lambda, m)$
8: Initialize empty array *sizes*
9: $SimOHashTable.insert(\mathcal{L}, \mathcal{W}_R, R)$
10: **for** $i \in [0, m-1]$ **do**
11: $SimOHashTable.size(\mathcal{L}, \mathcal{W}_S)$
12: Randomly assign $sizes[i]$
13: **for** $i \in [0, m-1]$ **do**
14: $SimOHashTable.size(\mathcal{L}, \mathcal{W}_R)$
15: Randomly assign $sizes[i+m]$
16: Normalize *sizes* to out_size
17: $SimOExpansion(\mathcal{L}, R \cup S, sizes)$
18: **for** $i \in [1, out\_size]$ **do**
19: $SimOHashTable.search(\mathcal{L}, \mathcal{W}_R)$
20: $SimOHashTable.search(\mathcal{L}, \mathcal{W}_S)$
21: $SimOSel(\mathcal{L})$
22: $SimOHashTable.insert(\mathcal{L}, \mathcal{W}_S, S)$
23: $res \leftarrow SimOSelection(\mathcal{L})$
24: $SimOHashTable.retire(\mathcal{L}, \mathcal{W}_R)$
25: $SimOHashTable.retire(\mathcal{L}, \mathcal{W}_S)$
26: **return** *res*

series of unique lookups from W_R in the hash table (lines 10-11). This leaks the volume pattern deg^b for each tuple b. Lines 12-14 trim result and both windows using oblivious selection (proven in Theorem 4). Finally, the simulator returns *res*, which leaks the overall volume pattern $|E|$. The total leakage equals to $\mathcal{L}_2$. □

Algorithm 11 Simulation of *FK-EPHI-L2*.

1: **procedure** SimOcaL$_2$($\mathcal{L}$):
2: m and n be public parameters that denote batch size and window size, respectively
3: $R, S \leftarrow \mathcal{A}(1^\lambda, m)$
4: $W_R \leftarrow W_R \cup R$
5: $W_S \leftarrow W_S \cup S$
6: $W_R \leftarrow SimOShuffle(\mathcal{L})$
7: $W_S \leftarrow SimOShuffle(\mathcal{L})$
8: **for** $i \in [0, (m+n-1)]$ **do**
9: $HashTable.insert(W_S[i])$
10: **for** $i \in [0, (m+n-1)]$ **do**
11: $res \leftarrow res || HashTable.search(W_R[i])$
12: $res \leftarrow SimOSelection(\mathcal{L})$
13: $\mathcal{W}_R \leftarrow SimOSelection(\mathcal{L})$
14: $\mathcal{W}_S \leftarrow SimOSelection(\mathcal{L})$
15: **return** *res*

Algorithm 12 Simulation of ObliviousAppend.

1: **procedure** SimOAppend($\mathcal{L}$):
2: m and n be public parameters that denote batch size and window size, respectively
3: $T \leftarrow SimOSort(\mathcal{L})$
4: append $(n-m)$ dummies to T
5: $C \leftarrow SimOMerge(\mathcal{L})$
6: **return** $C[0..m+n)$

B.5 *FK-MERG-L3* and *FK-MERG-L4* FK join

Theorem 8. *Algorithm 3 is oblivious according to Def. 1 assuming Sort and Merge used algorithms are oblivious to Def. 1.*

Proof. Simulator $\mathcal{S}$ executes SimOAppend (Algorithm 12). First, $\mathcal{S}$ obliviously sorts the batch and appends $(n-m)$ dummy rows to the result, where m and n are public information. Second, it obliviously merges the batch with the array. Finally, it returns the first $(m+n)$ rows of the merged collection. □

Theorem 9. *Algorithm 4 for FK-MERG-L3 is $\mathcal{L}_3$-secure according to Def. 1, assuming Append and Compaction used algorithms are oblivious to Def. 1.*

Proof. Simulator $\mathcal{S}$ executes SimJoin$_3$ (Algorithm 13). It starts with two oblivious appends (lines 2-3) that depend on the batch and window sizes, both public information. Then, it performs a scan and marks the rows that satisfy the join predicate. Next, it executes two appends and another scan (lines 4-8). $\mathcal{S}$ now performs two selects, which internally consist of a full sequential scan, compaction, and truncation to the window size (public information). It obliviously compacts the result array (line 12) to bring join matches to the top. Finally, it truncates the array to the desired output size. The leakage consists of the output size, proving it $\mathcal{L}_3$-secure. □

Algorithm 13 Simulation of Join$_3$ for *FK-MERG-L3* and *FK-MERG-L4*.

1: **procedure** SimJoin$_3$($\mathcal{L}$):
2: m and n be public parameters that denote batch size and window size, respectively
3: Generate R and S of m random tuples
4: Generate T_1 of m and T_2 of n random tuples
5: $W_R \leftarrow SimOAppend(\mathcal{L})$
6: $T_1 \leftarrow SimOAppend(\mathcal{L})$
7: $D_{out} \leftarrow \mathcal{L}$
8: $D_{out} \leftarrow m^2$
9: Randomly mark x rows, where $0 \le x \le D_{out}$
10: $T_2 \leftarrow SimOAppend(\mathcal{L})$
11: Randomly mark $(D_{out} - x)$ rows
12: $W_S \leftarrow SimOAppend(\mathcal{L})$
13: $W_R \leftarrow SimOSelect(\mathcal{L})$
14: $W_S \leftarrow SimOSelect(\mathcal{L})$
15: $res \leftarrow T_1 || T_2$
16: $res \leftarrow SimOCompact(\mathcal{L})$
17: Truncate *res* to the output size denoted by $\mathcal{L}$
18: **return** *res*

Theorem 10. *Algorithm 4 for FK-MERG-L4 is $\mathcal{L}_4$-secure according to Def. 1.*

Proof. The proof is the same as for Theorem 9 except for the two final steps of compaction and truncation. All remaining operations are oblivious, proving it $\mathcal{L}_4$-secure. □