RESULTS reproduced

Artifacts Available acm V1.1

# What Is the Price for Joining Securely? Benchmarking Equi-Joins in Trusted Execution Environments

Kajetan Maliszewski* Jorge-Arnulfo Quiané-Ruiz*,∓ Jonas Traub* Volker Markl*,∓

*Technische Universität Berlin (TU Berlin) ∓German Research Center for Artificial Intelligence (DFKI)
[maliszewski,jorge.quiane,jonas.traub,volker.markl]@tu-berlin.de

## ABSTRACT

Protection of personal data has been raised to be among the top requirements of modern systems. At the same time, it is now frequent that the owner of the data and the owner of the computing infrastructure are two entities with limited trust between them (e. g., volunteer computing or the hybrid-cloud). Recently, trusted execution environments (TEEs) became a viable solution to ensure the security of systems in such environments. However, the performance of relational operators in TEEs remains an open problem. We conduct a comprehensive experimental study to identify the main bottlenecks and challenges when executing relational equi-joins in TEEs. For this, we introduce TEEBENCH, a framework for unified benchmarking of relational operators in TEEs, and use it for conducting our experimental evaluation. In a nutshell, we perform the following experimental analysis for eight core join algorithms: off-the-shelf performance; the performance implications of data sealing and obliviousness; sensitivity and scalability. The results show that all eight join algorithms significantly suffer from different performance bottlenecks in TEEs. They can be up to three orders of magnitude slower in TEEs than on plain CPUs. Our study also indicates that existing join algorithms need a complete, hardware-aware redesign to be efficient in TEEs, and that, in secure query plans, managing TEE features is equally important to join selection.

**PVLDB Reference Format:**
Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, Jonas Traub, Volker Markl. What Is the Price for Joining Securely? Benchmarking Equi-Joins in Trusted Execution Environments. PVLDB, 15(3): 659 - 672, 2022.
doi:10.14778/3494124.3494146

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/agora-ecosystem/tee-bench.

## 1 INTRODUCTION

An increasing number of applications deal with sensitive data. For example, hospitals process health records of their patients, or social media companies collect information about their users. A common aspect across these applications is that data is in one of three states: at rest, in transit, or in use. However, the current generation of data processing systems protects data in only two of these states: at rest (e. g., using data encryption) and in transit (e. g., using transport layer security). They have assumed that the machine running the code is trusted, thereby protection of data in use is not necessary.

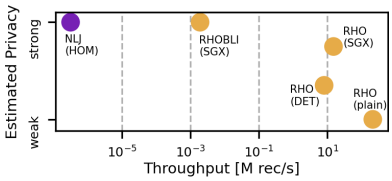


**Figure 1: Secure joins throughput.**

Several processing models, e. g., volunteer computing [4] and the hybrid-cloud [44, 70], often involve cooperation between entities that do not fully trust each other. For example, hospitals might want to process their patient data in the public cloud. These models challenge the assumption of existing systems because now data has to be protected in all three states. Moreover, researchers have found that *joining* different datasets is a daily routine across many fields [22]. Crossing the boundary of a single organization just adds to the complexity of the problem: Joins can be performed on infrastructure owned by entities other than the data owners, potentially, with no trust between them. Although one might think of encrypted databases [53, 55] as a solution, the underlying encryption schemes support a limited set of operations and can add large overheads [49]. Thus, encrypted databases have not been adopted on a broad scale leaving a void in secure data processing.

The hardware community has proposed Trusted Execution Environments (TEEs) to fill this gap. A TEE is a secure area inside the CPU that protects code and data on the hardware level. While TEEs provide the right level of security, their performance in executing relational operators remains an open challenge. Additionally, the performance of the existing data processing systems that utilize TEEs is far from satisfactory. Opaque [72] comes with an overhead of up to 46x, while ObliDB [21] uses Oblivious RAM (ORAM) which does not scale [10]. To illustrate the problem, we examined the performance of a radix-based equi-join using several security techniques: (i) insecure (on a plain CPU), (ii) in a secure enclave, (iii) in a TEE with ORAM enabled, (iv) using deterministic encryption, and (v) using homomorphic encryption (nested loop equi-join).[1] In addition, we estimated the privacy of each technique based on the leakage profile. Figure 1 shows the results of this experiment. We observe that throughput is reduced by one order of magnitude when joining inside a TEE compared to a plain CPU. The performance is further reduced by four orders of magnitude when the join uses ORAM (i. e., hides data access patterns). These penalties occur when we blindly execute existing solutions on this novel

[1]The experimental setup is described in Section 3 and the dataset in Section 6.2.


Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494146

hardware. Likewise, two encryption techniques did not present a viable option. The homomorphic encryption join (HOM) [67] has the highest privacy profile but the performance of the current encryption schemes is still very low. In contrast, using deterministic encryption (DET) for equi-joins, we only pay the price of processing longer keys (e. g., in our case the RSA-encrypted keys were 64 bytes long). However, DET is known to leak large amounts of information if not combined with stronger encryption schemes [50].

We believe that the gap between TEEs and plain CPUs can be significantly reduced. However, no study precisely shows where the loss of performance comes from. Therefore, we first need to thoroughly understand how relational operators perform inside a TEE. We focus on equi-joins, as they are at the core of many fields and emerging data processing platforms [23, 47, 65]. For example, Agora [65] requires secure computation to join datasets on untrusted nodes. The join operator is among the most resource-intensive operators, making it a common study subject. Previous works have benchmarked join algorithms for plain CPU [58] and streaming scenarios [71], as well as the general performance of TEEs [66, 68]. Unfortunately, none of them has provided insights into the performance of the join operator in a TEE. As more works on secure join processing emerge [1, 6, 42, 45], we need a detailed analysis of the classic and new equi-join algorithms on TEEs.

We believe that the lack of such a comprehensive analysis slows down the adoption of confidential computing in data processing systems. Yet, performing such an analysis is challenging because no framework allows us to thoroughly compare different join algorithms (as well as any other relational operator). Our first contribution is then a unified benchmark framework to perform a detailed comparison among different equi-join algorithms in TEEs:

**(1)** We introduce TEEBENCH, a framework for benchmarking relational joins and other operators in TEEs (Section 4). It allows for the effortless execution of any join algorithm in a secure enclave. To our knowledge, it is the first open-source framework that allows for an experimental evaluation of any relational operator across multiple TEEs. We use TEEBENCH to conduct all our experiments.

Our experimental study makes the following contributions:

**(2)** We start with a general performance comparison. We measure the throughput of all eight algorithms for synthetic and real datasets. We then look deeper into the stages of the algorithms and report hardware performance counters (Section 5).
**(3)** We proceed to study features related to TEEs and secure processing (Section 6). We measure the cost of data sealing and investigate if chunking can reduce it. We estimate the cost of obliviousness and compare its performance to related work [21, 42, 72]. We also limit the interaction with the OS using lockless synchronization.
**(4)** We continue with sensitivity and scalability analysis of the join algorithms in TEEs (Section 7). We use diverse input data to stress the algorithms and measure the throughput using more threads.
**(5)** Finally, we discuss and benchmark alternative TEEs (Section 8).

In summary, our comprehensive study presents seven main lessons learned (Section 9). Above all, we are the first to find that:

**(a)** Existing algorithms need a complete, hardware-aware redesign that adapts to the secure memory model (i. e., considers the EPC) and the runtime environment. For example, lockless synchronization improves the throughput by up to 34%.
**(b)** Chunking reduces the overall cost of sealing by up to 40%.
**(c)** Radix partitioning adjusts well to the memory model of Intel SGX and alleviates EPC paging.

## 2 BACKGROUND

We now (i) explain the technology of TEEs in Section 2.1, (ii) discuss data access patterns and how they can be exploited in Section 2.2, (iii) describe our threat model in Section 2.3, and (iv) present the eight join algorithms that we consider in our study in Section 2.4.

### 2.1 Trusted Execution Environments (TEEs)

A TEE is an isolated area in the CPU that protects data and computation on untrusted machines, e. g., in a public cloud. TEEs provide two main properties: (i) data confidentiality, and (ii) code and data integrity. These properties ensure that all data and executed code are encrypted such that they can never be accessed by a malicious OS, administrator, or hypervisor. TEEs are implemented in hardware using the concept of *enclaves*, extensions to the CPU instruction set, designed for confidential computing. They provide mechanisms that enable the above-mentioned properties: *protected memory*, *isolated processing*, and *sealing*. Although these concepts are general, manufacturers implement them in different ways.

**TEE Models.** There are two TEE models: *process-based* (e. g., Intel SGX) and *VM-based* (e. g., AMD SEV). The *process-based* model divides a secure process into trusted and untrusted parts. While the trusted part resides in the protected memory and handles isolated processing, the untrusted part communicates with the OS and handles the I/O from the trusted part. This model keeps the size of the Trusted Computing Base (TCB) small and meticulously monitors all traffic in and out of the enclaves. However, porting applications requires a substantial amount of work and expert knowledge. The *VM-based* model contains a traditional Virtual Machine (VM) with its memory encrypted using hardware encryption keys. This model offloads the integration burden to the VM-provider and makes the adoption of TEEs easier. However, it requires running the OS inside the VM, which increases the TCB and the attack surface.

In addition, while SGX never allocates more than 256 MB for its protected memory, SEV allocates the memory needed to run the entire VM which can easily be in the range of gigabytes. Yet, AMD SEV has been reported to have negligible performance overhead [29]. Despite this, Intel has the dominant position on the market (93% market share [54]) and is widely adopted by the industry. Therefore, it is crucial to properly identify the bottlenecks of *process-based* enclaves (such as Intel SGX). We thus consider Intel SGX.

**Protected Memory.** An enclave stores code and data in an isolated virtual address space, called *Enclave Page Cache* (EPC). The EPC is accessed exclusively by enclave processes. Before storing in the EPC, the on-die Memory Encryption Engine (MME) encrypts the data. The MME ensures that data is available in unencrypted form only in the CPU cache. As of today, EPC memory sizes are severely limited (at most 256 MB in current SGX implementations). Moreover, it is shared between all the enclaves running on the same CPU and stores both its metadata and users' data. Because one often exceeds this limit, SGX provides a mechanism for evicting EPC pages to the main memory by encrypting them first and storing them in an *Eviction Tree* [17]. An EPC miss costs as much as 40K CPU cycles [64]. Even

though there have been attempts to improve the performance of EPC [52, 64], it has remained a significant bottleneck. The ongoing work on Total Memory Encryption (TME) [35] might mitigate the problem, but, till today, little is known how it will work with SGX.

**Isolated Processing.** The threads running in an enclave use protected virtual address space. Interactions with enclaves are possible only through predefined calls: entry points (ECALLs) and exit points (OCALLs), each executing a CPU instruction – EENTER and EEXIT, respectively. They entail steps necessary to preserve privacy, such as flushing the memory address translation cache (TLB), handling page faults, or context switching. A single OCALL can cost up to 15K CPU cycles [63], hence it is advised to minimize them.

**Sealing.** An enclave can encrypt the data such that only the enclave's owner can decrypt it. This mechanism is called *sealing* [3]. It enables the enclaves to use the OS or the outside network to securely store their data as well as to exchange encrypted data with outside services. We observed that sealing one megabyte of data can cost up to 6M CPU cycles.

## 2.2 Data Access Patterns

Enclaves are dedicated to protecting data in use in all but one aspect - *access patterns* [31]. The protected memory area resides in the main memory, so a malicious administrator can still track the exact memory addresses being accessed. Over time, the attacker can learn facts from encrypted data and information about the queries the users are sending. Numerous works have shown astonishing effects exploiting this vulnerability [15, 30, 37, 38, 46, 56, 69].

There are two ways to hide access patterns: using Oblivious RAM (ORAM) and accessing memory in a way indistinguishable between two runs. ORAM [28] is an abstraction that uses obfuscating techniques to hide memory reads and writes of a program. Although being a generic solution, ORAM introduces polylogarithmic computation complexity and size overhead [10], making it impractical for real-size datasets. In contrast, oblivious algorithms may not introduce large hidden constants or such elevated complexity as ORAM. However, an oblivious algorithm hides access patterns in its specific way and the ideas cannot be generalized. This dramatically increases the effort when hiding access patterns in a complex system. In our experiments, we compare the performance of state-of-the-art examples of these two approaches [21, 42, 72].

## 2.3 Threat Model

TEEs protect against a malicious system administrator, hypervisor, or the OS. We expect that the attacker can monitor network communication, learn memory access patterns, and tamper with untrusted memory. She can perform full snapshots of the memory and learn facts about the data from the past (e. g., previous unencrypted versions of the database) or outside sources (e. g., distribution of diseases in publicly available medical data). We do not consider side-channel attacks on TEEs [51] as the majority of them are implementation-specific. Each of the currently available TEEs has disclosed attacks [14, 51] and cannot be considered fully secure. However, some of these attacks can already be mitigated [59, 60]. We expect the manufacturers to fix these vulnerabilities. TEE-based systems can support a wide range of security models [5, 21, 48, 63, 72]. Fewer privacy leaks mean a higher computational price. Hence, users might decide on practical trade-offs and choose to allow leakages to gain performance. For instance, obliviousness can severely hurt performance [21, 48, 61, 72] but benefits users only slightly as exploiting access patterns is in most cases impractical. In fact, recent works [5, 63] assume operational data confidentiality, i. e., they boost performance by allowing access patterns leakage. Users might also focus on code integrity rather than data confidentiality, e. g., when running systems compliant with GDPR [61]. In such cases, users might choose to execute the code in a TEE using plain data to avoid high encryption costs.

**Table 1: Join algorithms considered in this evaluation.**

| class | acronym | name | code | color |
|---|---|---|---|---|
| hash-based | CHT | Concise Hash Table [9] | [58] | |
| | PHT | Parallel Hash Table [11] | [8] | |
| sort-merge | PSM | Parallel Sort-Merge | Own | |
| | MWAY | Multi-Way Sort-Merge | [41] | |
| radix-based | RHT | Radix Hash Table [41] | [8] | |
| | RHO | Radix Hash Optimized [8] | [8] | |
| | RSM | Radix Sort-Merge | Own | |
| nested-based | INL | Index Nested Loop | Own | |

## 2.4 Join Algorithms

Recall that our goal is to provide an extensive experimental performance study of state-of-the-art join algorithms running in a TEE. These algorithms can be divided into four classes: (i) hash-based, (ii) sort-merge, (iii) radix-based, and (iv) nested-based (Table 1).

**Hash-based.** We consider two hash-based algorithms: the *Concise Hash Table* (**CHT**) [9] and the *Parallel Hash Table* (**PHT**) [11]. The CHT join is based on a linear probing hash table redesigned to consume less memory and to have faster access. It increases the fill factor to 100% and avoids collisions with a sparse bitmap with embedded population counts. Effectively, it reduces memory usage by up to three orders of magnitude. This makes CHT an attractive option for secure enclaves, which operate on limited secure memory. The PHT join follows a "no-partitioning" design of a hash join algorithm. It builds a shared hash table and uses all available threads to probe the table. We picked PHT for the benchmark as a robust and efficient implementation of the canonical hash join algorithm.

**Sort-merge.** We consider two sort-merge algorithms: the *Multi-Way Sort-Merge* (**MWAY**) [41] and the *Parallel Sort-Merge* (**PSM**). The MWAY algorithm is optimized for NUMA systems. It splits both relations into a few partitions. Each partition is locally sorted using sorting networks. The partitions are then multi-way merged. Finally, the sorted relations are joined with a single-pass merge join. The PSM join comprises two phases: the sort and merge phases. First, the algorithm uses a parallel three-way quicksort [34] to efficiently sort both relations. Second, the algorithm scans both tables sequentially and merges the matching results.

**Radix-based.** We consider three implementations for the radix-based class: (i) *Radix Hash Table* (**RHT**) [41], (ii) *Radix Hash Optimized* (**RHO**) [8], and (iii) *Radix Sort Merge* (**RSM**). All three algorithms have two phases: the partition and join phases. They share

**Table 2: Datasets used in most of the experiments.**

| | *cache-fit* | *cache-exceed* |
|---|---|---|
| size of key/payload | 4/4 bytes | 4/4 bytes |
| R cardinality | 1.3M | 5.2M |
| S cardinality | 13M | 52M |
| \|R\|:\|S\|ratio | 1:4 | 1:4 |
| total input size | 50 MB | 500 MB |

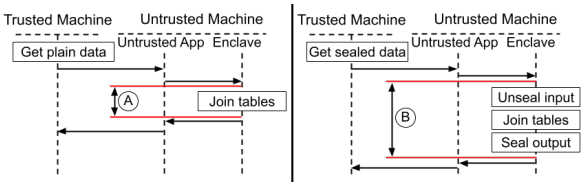


**Figure 2: Join in TEEs using *plain* and *sealed* data.**

the same partitioning algorithm but differ in the join algorithm. Partitioning follows the design by Kim et al. [41]. The relations are reordered into multiple *clusters* based on pre-computed histograms. This reduces TLB misses and improves cache locality. RHT uses a histogram-based hash table with tuples reordering to join the partitions. The hash table comprises a histogram of hash values and the outer relation re-ordered by hash values. RHO improves on RHT. It builds the hash table using an optimized implementation of bucket chaining. It then stores the buckets and synchronization latches as a contiguous array, removing the need for pointer dereferencing and reducing cache misses. We include RHO in our benchmark as one of the most performant radix joins. We also include RHT to see if the difference between RHO and RHT translates to TEE. RSM is our implementation that matches the tuples with a sort-merge algorithm in the join phase. It quicksorts both partitions and scans them to find matches. We benchmark RSM to see if a sort-merge algorithm can prevail over hash join for small data chunks.

**Nested-based.** We consider *Index Nested Loop* (**INL**) because it is a popular choice for join queries with low selectivities. INL first builds an index on the outer table using a B+Tree. Second, it uses multiple threads to probe each element of the inner table against the index. We assume the index for the outer table already exists. In addition, we measured the throughput of the classic *nested-loop* join (**NLJ**). We decided to exclude it from the evaluation due to its extremely low performance in all experiments.

# 3 METHODOLOGY

We now describe the setup and methodology of our evaluation. We define the datasets in Section 3.1, outline the hardware in Section 3.2, and detail how we measure the join's performance in Section 3.3.

## 3.1 Datasets

We use two synthetic datasets (Table 2) designed to cover key TEE scenarios: dataset *cache-fit* that fits into the secure memory and dataset *cache-exceed* that significantly exceeds the secure memory. Although limited in size, we believe that these two datasets trigger the major behaviors of TEEs. The datasets comprise <*key,payload*> tuples, where both attributes are 4 bytes wide, similar to related work [8]. The keys follow a primary key-foreign key relationship, as it is the most common use case for a join operator in DBMSes. The keys are distributed uniformly in both datasets.

We also experiment with real data. We use two tables from the IMDb dataset: *name.basics* and *title.basics* [32]. The cardinalities of the tables are 10.5 million and 42.2 million, respectively, which after filtering out unused columns sums up to 402 MB. While real data validates the relevance of the results, synthetic data allows us to vary the input characteristics and to cover more diverse scenarios.

## 3.2 Platform

We ran our experiments on a machine with a 1.8 GHz quad-core Intel Core i7-8565U CPU. The CPU has the following caches: 32 kB L1d, 32 kB L1i, 256 kB L2, and 8 MB L3. The TLB has 64 entries for both L1 TLB and L2 TLB with 4 kB pages. The machine has 38 GB of main memory and 128 MB of EPC. It runs a 64-bit Ubuntu 18.04.02 OS and a custom SGX driver v2.11 [26].All algorithms were written in C/C++ and compiled with gcc 7.5.0. The experimental setup extends the *Sample Enclave* example code provided by Intel [33].

## 3.3 Join Processing Study

We study the performance of relational equi-joins. We join two relations R and S, where R is the outer and S is the inner relation, on a join predicate *R.key = S.key*. We run some experiments securely (SGX version) and insecurely (plain CPU version). Note that the code needs a separate compilation for each platform. The base metric in most of the experiments is *throughput*, which we calculate as *the sum of input cardinalities divided by the join execution time.*

Figure 2 shows two ways to measure the execution time of a join. One isolates the performance of the algorithm to the highest extent (measurement Ⓐ). We feed the enclave with plain data and do not materialize the output. A similar methodology has been followed in many related works [11, 19, 41, 58]. Although this scenario violates data confidentiality, it allows focusing precisely on the performance of the algorithm. In addition, it is useful when users seek to comply with regulations (e. g., GDPR) and prove that specific computation was performed over data (i. e., code integrity) without considering its confidentiality. We then adapt the methodology to the secure environment (measurement Ⓑ). We provide sealed input data and expect sealed output. We thus measure the cost of sealing/unsealing, joining, and output materialization. In both cases, we measure the execution only on the untrusted machine. As the communication between machines is not measured, we can simulate both *trusted* and *untrusted* machines on a single-node setup.

Note that an experiment begins with the generation of data and the creation of a new enclave. This ensures unpolluted EPC and increases the repeatability of the experiments. The same data is generated for each run by fixing the seed value. We repeated each experiment five times and reported the average of the results. We leave out the study of inequality joins [39, 40] to future work.

# 4 FRAMEWORK

Although confidential computing has started to attract the attention of the database community [21, 42, 72], comparing different algorithms (or systems) in TEEs is not trivial. First, when comparing two algorithms one has to make sure to provide the same level of

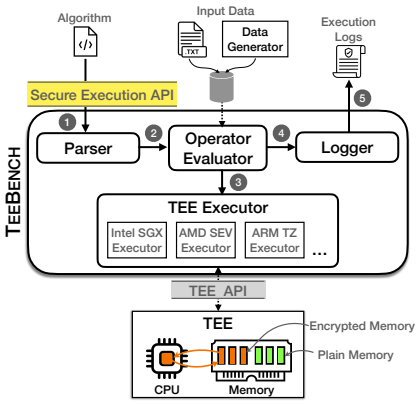


**Figure 3: General architecture of TEEBENCH.**

security (e. g., whether both algorithms are oblivious). Doing so is hard because the technology is young and the community has not worked out single standards and definitions (e. g., there are multiple definitions of *obliviousness* [42]). Second, it is unclear how to take precise measurements inside secure enclaves, because they have no notion of time and have to call the resources of the OS. As a result, different works often take different approaches to measure the processing time. This introduces an ambiguity that leads to the inability to compare results across algorithms and systems.

Thus, to be able to perform our experimental evaluation, it is crucial to first come up with a way to compare algorithms (composed of relational operators) fairly. Achieving this goal is challenging for several reasons. First, enclaves have no notion of time as they do not support system calls. Second, there are new hardware events, e. g., swapping encrypted pages, that require new ways of detection and measurement. Third, enclave implementations differ across platforms, thereby the tool has to operate on different architectures. Fourth, enclaves introduce new memory models and hardware limitations that hinder the integration with current DBMSes.

We have developed TEEBENCH, a unified benchmarking framework to tackle the above challenges. It is open-source under the Apache 2.0 License [27].Using the framework, one can easily port algorithms and compare them across different TEEs. TEEBENCH provides a full benchmarking experience: from generating test data and algorithm execution to the presentation of the results and collection of the logs. Figure 3 illustrates the architecture of TEEBENCH. The framework is composed of four main components: Parser, Operator Evaluator, TEE Executor, and Logger. In detail, TEEBENCH executes programs (algorithms) as follows. As the first step, a user provides an algorithm for a relational operator (1) using Standard Template Libraries (STL) [18] to remain independent of any CPU architecture. To help the user, we provide a Secure Execution API, which can be used to provide advanced insights into the execution. This API provides a set of calls for precise time measurements and collection of enclave's hardware counters (e. g., EPC misses) among others. Additionally, it provides an interface to a robust implementation of Path ORAM [62] from ObliDB [21], which enables porting memory accesses of any algorithm to ORAM. As a result, our framework can transform any relational operator to its oblivious equivalent. Altogether, TEEBENCH enables users to quickly "secure" their algorithm and to easily compare it on hardware platforms from different manufacturers. For instance, we used the framework to compare the performance of Intel SGX and AMD SEV. Furthermore, it can operate on both native CPU and TEE, offering insights into the differences between the two environments. The Parser receives the user's algorithm and checks the correctness of both the input and the runtime configuration. Then, the Parser compiles the algorithm for the platform selected by the user and outputs a binary (2). Next, the Operator Evaluator executes the binary file. During the execution, it imports the input data provided by the user or calls the Data Generator. TEEBENCH provides an extended version of the data generator originally implemented by Balkesen et al. [8]. The generator includes new features, such as reading input files or manipulating join selectivity. The Operator Evaluator coordinates the execution via the TEE Executor (3). The latter abstracts the interaction with the TEE. Hence, the user remains oblivious to the underlying hardware. The Operator Evaluator outputs the aggregates of the different execution measurements (4). The Logger gets the aggregated data and provides a unified way of reporting and logging the results directly to the user or external memory (5).

We thoroughly examined TEEBENCH's usability for `joins` on native CPU and TEEs (both Intel and AMD architectures). While TEEBENCH successfully tackles the first three before-mentioned challenges, the fourth requires building an enclave-native database engine which is out of the scope of this work.

## 5 OFF-THE-SHELF PERFORMANCE

We first study which of the join algorithms has the highest throughput in TEEs and what their bottlenecks are. We take the algorithms *off-the-shelf* and measure their performance: We measure their throughput (Section 5.1), analyze the CPU cycles of different phases (Section 5.2), look at how they behave for significantly larger inputs (Section 5.3), and inspect the hardware counters (Section 5.4). Table 1 contains the join acronyms used in the experiments. In all figures, we use colors to indicate an algorithm (e. g., CHT, RSM) and hatching patterns to represent the mode (e. g., SGX, plain CPU) in which an algorithm is executed.

**Synopsis:** We show that EPC paging is the thinnest performance bottleneck and should be avoided at all costs. Partitioning of data can be expensive but it alleviates the paging cost to a high degree. As a result, the (radix-based) RHO algorithm has the highest throughput for datasets exceeding the EPC due to its partitioning mechanism. In contrast, for small datasets fitting the EPC, the (hash-based) CHT algorithm has the highest throughput.

### 5.1 General Throughput

We take all eight join algorithms off-the-shelf and measure their performance with regards to the overall throughput. We run the same algorithms using both a plain CPU and a TEE. We use both the *cache-fit* and *cache-exceed* datasets.

Figure 4 shows the results of these experiments. The most apparent result is the significant difference in performance between the insecure and secure processing modes. We observe that the throughput of secure computation is up to three orders of magnitude smaller than the one of an insecure computation. While

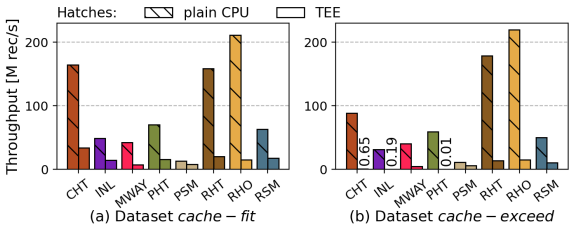


**Figure 4: Throughput of join algorithms.**

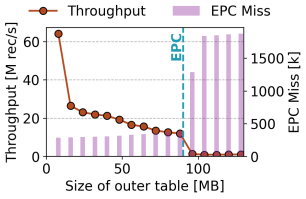


**Figure 5: CHT's throughput and EPC paging**

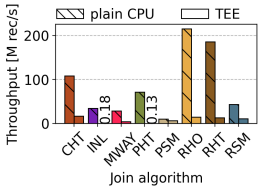


**Figure 6: Join algorithms' throughput with IMDb.**

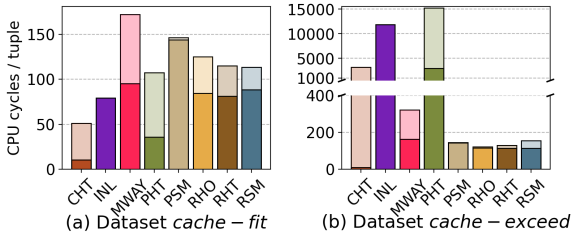


**Figure 7: CPU cycles per tuple for each stage in joins. Dark color indicates the first stage (build, sort, or partition) and light color indicates the second stage (probe, merge, or join).**

the plain CPU (insecure mode) can achieve a throughput above 210M rec/s (e. g., RHO), the TEE (secure mode) achieves 35M rec/s (e. g., CHT) at best. These costs include the maintenance of data confidentiality as well as code and data integrity. The results clearly show the massive processing cost added by TEEs.

In detail, we observe that CHT has the highest throughput for dataset *cache-fit* due to its highly optimized hash table. The table achieves almost 100% occupancy with almost no collisions, which results in a low memory footprint and very fast access. However, when the hash table grows too big it starts to utilize higher levels of CPU caches, EPC, and, eventually, main memory. This causes the low performance of CHT for the *cache-exceed* dataset. For such large datasets, it makes sense to split the input dataset into small pieces and ensure the data is kept high in the memory hierarchy and never has to be paged out of EPC. Radix-based joins do exactly this. They reduce memory access time by improving data locality [12]. This is why radix-based joins outperform other families for the *cache-exceed* dataset. The sort-merge family, in contrast, falls behind due to two reasons. While PSM sorts entire relations, which causes excessive random memory access, MWAY allocates additional memory for partitions, which leads to EPC paging. Moreover, this family suffers from the lack of SIMD instructions in TEEs [7, 16, 41].

We also observe that some secure algorithms demonstrate a significant difference in throughput for different input sizes. Although the radix-based and sort-merge joins do not experience major differences between *cache-fit* and *cache-exceed*, the performance of hash-based and nested-based joins is drastically reduced. For instance, PHT processes 16.11M rec/s for dataset *cache-fit* while it only takes 0.13M rec/s for dataset *cache-exceed*. Both CHT and PHT process the input data in one batch and are hardware-oblivious. As the input relations grow, they allocate more memory, which expands to the next levels in the memory hierarchy, eventually exceeding the EPC. Similarly, INL allocates memory for building an index. If the size of the index exceeds the EPC (e. g., for dataset *cache-exceed*) the performance immediately drops.

As EPC paging is a bottleneck [64] on the joins' throughput, we fix the size of the inner relation to 100MB and scale the size of the outer relation. In this setup, we measure the throughput of CHT as well as its EPC misses. We expect CHT to show a possible relationship between paging and throughput in the clearest form because it gradually allocates memory for the hash table. Figure 5 illustrates the results of this experiment. It is apparent, that when EPC paging grows, throughput reduces. A barrier of 93MB (i. e., EPC operational size) clearly marks the biggest difference in throughput. Once we cross the EPC size, the EPC misses increase which drastically decreases the throughput of CHT. With an estimated cost of 40k CPU cycles for an EPC miss [64], paging becomes a severe bottleneck. The results suggest that enclave-native algorithms can do several optimizations to reduce EPC paging: (i) they should reduce the memory consumption, even at the cost of more CPU processing; (ii) they should align the data structures to the SGX memory model, e. g., better exploit 4kB pages.

Finally, we measure the throughput of the joins using the IMDb dataset. The size of the dataset is comparable to the size of *cache-exceed*, thus we anticipate similar throughputs. Figure 6 shows the results of this experiment. These results confirm our intuition. All, except CHT, resemble the *cache-exceed* dataset. CHT performs better due to the smaller cardinality of the outer table, which allows it to allocate less memory and limit EPC paging (similar to Figure 5).

We can already observe that inaccurate join selection can lead to heavy performance degradation. These results underline the importance of a thorough examination of join algorithms on TEEs.

**Finding:** CHT is the fastest for *cache-fit* and RHO for *cache-exceed*. EPC paging largely determines the performance of a join.

## 5.2 Measuring Phases of Joins

We now take a deeper look into each join. We seek to find out where the CPU cycles are spent and identify the most CPU-intensive operations. This will help us to reduce their impact. We split the join families into stages: hash-based to build and probe; sort-merge to sort and merge; and radix-based to partition and join. We measure the CPU cycles to process one input tuple for each stage, which allows us to compare stages between algorithms and datasets.

Figure 7 illustrates the results of these experiments. We observe that the sort phase of PSM has a higher cost than any other stage for dataset *cache-fit*. This is because of a high number of random

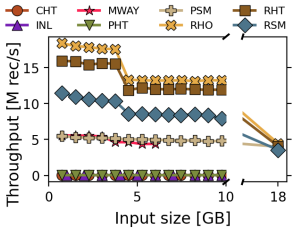


**Figure 8: Scaling dataset *cache-exceed*.**

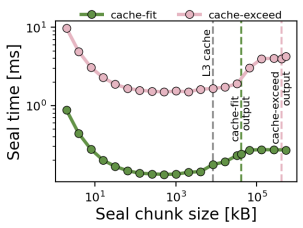


**Figure 9: Chunking impact.**

memory accesses. However, the cost remains similar for dataset *cache-exceed* due to its *shallow copying* of the input data: PSM references the data to the outside of the enclave and avoids storing it in EPC. We see that although MWAY reduced the cost of sorting, it paid a big price to merge the partitions. In addition, MWAY does not perform well due to using more memory than the EPC has.

We see that CHT uses the least CPU cycles for dataset *cache-fit*. This is due to its concise data structures (see Section 2.4), which avoid collisions in the hash table and reduce cache misses. However, the hash joins and INL perform poorly for *cache-exceed*. Processing all the data in one chunk leads to data structures that exceed the EPC capacity and causes swapping to main memory. In our experiment for dataset *cache-exceed*, CHT, PHT, and INL cross this point. EPC thrashing brings their cost per tuple up to 15K CPU cycles.

In contrast to "no-partitioning" joins, radix-based joins partition the data into small chunks. It improves data locality and cache hit ratio, and avoids swapping to main memory. As a result, radix joins are the fastest algorithms for *cache-exceed*. More importantly, they show a performance independent of the dataset size.

**Finding:** Processing large chunks of data at once leads to more CPU cycles when joining tuples in TEEs.

## 5.3 Scaling Input Data

We believe that datasets *cache-fit* and *cache-exceed* expose many peculiarities of TEEs. Nevertheless, real datasets for in-memory databases often have higher cardinalities. We scale the *cache-exceed* dataset to 18 GBs – the maximum size before severe disk swapping occurs. We measure the throughput to learn if larger data alter the performance. The results show that, for the most part, the performance does not change radically (Figure 8). While the results of sort-merge and hash-based families are consistent with the *cache-exceed* dataset (Figure 4), the performance of radix-based joins reduces at 600M records. This marks when the partition size overflows the L2 cache. Adding radix bits could improve the performance. It results in more partitions but with smaller sizes. The throughput further decreases for 18 GBs because disk swapping starts. Note that enclaves often fail when disk swapping occurs. Some algorithms were not able to join large data due to unexpected behavior of enclaves or did not complete in the given time (i. e., 1h).

**Finding:** The performance of joins is stable for large data.

**Table 3: Hardware performance counters for secure joins.**

**Dataset *cache-fit***

| | 1st phase | | | | | | 2nd phase | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| alg | L2Miss [k] | L3Miss [k] | L2HitRate | L3HitRate | IR [M] | EPCMiss [k] | L2Miss [k] | L3Miss [k] | L2HitRate | L3HitRate | IR [M] | EPCMiss [k] |
| CHT | 187 | 14 | 0.57 | 0.87 | 56 | 6.3 | 71 | 30 | 0.34 | 0.5 | 9 | 0 |
| INL | 206 | 39 | 0.45 | 0.8 | 26 | 18 | - | - | - | - | - | - |
| MWAY | 2121 | 232 | 0.58 | 0.83 | 573 | 28.1 | 33 | 9 | 0.39 | 0.68 | 6 | 18 |
| PHT | 40 | 10 | 0.23 | 0.75 | 4 | 7.1 | 113 | 41 | 0.35 | 0.61 | 13 | 0 |
| PSM | 198 | 20 | 0.36 | 0.88 | 22 | 1.1 | 8 | 1 | 0.29 | 0.89 | 3 | 0 |
| RHO | 1228 | 75 | 0.6 | 0.88 | 347 | 14.3 | 1546 | 4 | 0.67 | 0.99 | 213 | 0 |
| RHT | 1368 | 82 | 0.6 | 0.88 | 364 | 14.3 | 596 | 5 | 0.62 | 0.98 | 80 | 0.7 |
| RSM | 1268 | 74 | 0.61 | 0.88 | 353 | 14.3 | 56 | 1 | 0.33 | 0.96 | 5 | 0 |

**Dataset *cache-exceed***

| | 1st phase | | | | | | 2nd phase | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| alg | L2Miss [k] | L3Miss [k] | L2HitRate | L3HitRate | IR [M] | EPCMiss [k] | L2Miss [k] | L3Miss [k] | L2HitRate | L3HitRate | IR [M] | EPCMiss [k] |
| CHT | 2296 | 279 | 0.56 | 0.81 | 596 | 57 | 674096 | 63670 | 0.53 | 0.82 | 185244 | 6602 |
| INL | 2575549 | 384643 | 0.50 | 0.75 | 717565 | 97226 | - | - | - | - | - | - |
| MWAY | 37640 | 3077 | 0.6 | 0.87 | 9959 | 516 | 7844 | 493 | 0.60 | 0.87 | 2612 | 128 |
| PHT | 584642 | 69772 | 0.51 | 0.79 | 180787 | 7011 | 2321834 | 280160 | 0.52 | 0.79 | 748240 | 27779 |
| PSM | 1827 | 148 | 0.31 | 0.91 | 173 | **1** | 19 | 3 | 0.28 | 0.84 | 2 | 0 |
| RHO | 18927 | 1627 | 0.58 | 0.85 | 5734 | 256 | 118 | 4 | 0.31 | 0.95 | 9 | 1 |
| RHT | 19164 | 1692 | 0.57 | 0.85 | 5785 | 256 | 1461 | 100 | 0.64 | 0.89 | 413 | 22 |
| RSM | 19536 | 1605 | 0.57 | 0.86 | 5763 | 256 | 332 | 18 | 0.26 | 0.94 | 27 | 1 |

## 5.4 Measuring Hardware Counters

We now take a look into the CPU to understand what happens on the hardware level. Our goal is to understand the performance bottlenecks that each algorithm has and show how to mitigate them. We use hardware performance counters to get insights from the CPU. In addition to the commonly used metrics (e. g., cache misses), we count invocations of the EWB CPU instruction. EWB is executed per EPC page eviction, thus, indicating the EPC misses.

Table 3 shows the results of this experiment. We can draw several observations from these results. First, stages with *EPC thrashing* (marked red) are among the longest-running ones, i. e., have the highest execution time. Swapping from EPC is very expensive, hence, EPC misses severely reduce performance (see *EPCMiss* column). Second, radix-based joins reach 256K EPC misses, for dataset *cache-exceed*, during the partition stage (marked yellow). Yet, they successfully reduce paging in the join phase and achieve a very high L3 hit rate (marked green). This leads to good performance independently of the input size (see Figure 4). Third, PSM achieves minimal paging (marked bold) because of *shallow copying* of the input data. This shows that in-place processing can mitigate EPC paging. However, random memory access leads PSM to be one of the worst-performing algorithms. Moreover, when working with sealed input (see Figure 2), PSM allocates secure memory for decrypted data, which can lead to EPC paging and throughput reduction.

**Finding:** Partitioning helps to reduce EPC misses.

# 6 SECURE ENCLAVE OPERATIONS

In a secure query plan, a join operator does more than finding matches. We study the performance implications of three actions an operator may take: it might (i) receive and send encrypted (*sealed*) data (Section 6.1), (ii) hide data access patterns (Section 6.2), and (iii) synchronize threads in a parallel execution (Section 6.3).

**Synopsis:** For our test datasets, the price for data sealing and materialization was up to 5× the overall cost of a join operator. Sealing in chunks that fit in the L3 cache can reduce the cost by

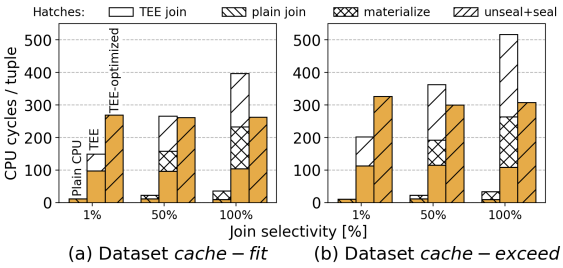


**Figure 10: Materialization and data sealing impact on RHO.**

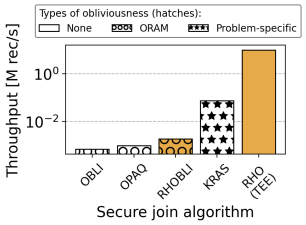


**Figure 11: Throughput of secure and oblivious joins.**

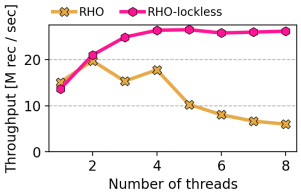


**Figure 12: Lockless RHO.**

up to 40%. We see that problem-specific obliviousness techniques are the only way to hide access patterns at scale. We also observe that lockless primitives can improve the throughput by up to 34%.

## 6.1 Data Sealing

*Sealing* enables secure data exchange with outside services (see Section 2.1). We study its impact on the overall performance of the join operator. We measure the CPU cycles the operator spends on join and sealing for three different join selectivities: 1%, 50%, 100%. We also look into the cost of materializing the output data. We then check if splitting into chunks can reduce sealing costs. Finally, we implement chunk sealing in RHO and measure its performance.

Figure 10 combines the results for data materialization and data sealing (*TEE* bars) in one graph and compares the results to plain CPU join (*Plain CPU* bars). Overall, we observe that sealing adds a high computational cost to the operator. However, this cost depends on the join selectivity. The sealing cost comprises unsealing the input and sealing the output. Although for the three cases there is a fixed cost for unsealing, the cost of sealing the output mainly depends on the size of the join result. Clearly, the higher selectivity, the higher sealing costs. In our setup, the sealing cost can reach as much as 2.5× the join cost. Data sealing is expensive because of two reasons: it allocates secure memory for decrypted input and encrypted output; and it encrypts and decrypts the data. We observed in our experiments that data sealing led to a 200% increase in EPC misses, compared to processing unencrypted data with materialization. This, in turn, reduces the overall throughput.

We observed that the cost of sealing is correlated with data transfer between enclaves. To reduce this cost, we can apply techniques akin to the data movement problem, e. g., filter pushdown, pipelining operators, or improving locality. Another way to reduce memory consumption is to split the relation into chunks and seal them separately. We ran a micro-benchmark to verify this idea. Figure 9 shows that well-tuned chunking reduces the seal time. For both datasets, the best results showed chunks that fit into the L3 cache. While smaller chunks require more CPU, larger chunks generate first cache misses and then EPC misses.

Partitioning joins can easily seal in chunks. We implemented this feature for RHO to verify if it improves the performance. We included the results in Figure 10 (*TEE-optimized* bar). We observe that the performance is now independent of selectivity. The matches are distributed uniformly between partitions so most of the partitions seal at least some data. Although the cost of chunking does not compensate for small selectivities, it greatly improves the performance for the higher ones. In fact, chunking reduced the cost per tuple by 40% for the highest selectivity. The results also reveal that materialization adds a large cost in TEE and plain CPU. For low selectivity, there are a few records to materialize and the penalty is small. However, the cost of materialization quickly grows to 1.5× the join cost for the highest selectivity. Output materialization is expensive due to memory allocation and frequent memory writes.

**Finding:** Sealing in chunks lifts the performance by up to 40%.

## 6.2 Obliviousness

Hiding access patterns is the cherry on top of confidential computing. It is the last element that ensures no privacy leakage while data is processed. However, TEEs do not protect against traffic analysis attacks [31]. Consequently, users are left with porting their algorithms to ORAM or redesigning their memory access patterns (see Section 2.2). We ran an experiment where we compare the secure RHO algorithm (*RHO TEE*) with: (i) TEE RHO atop ORAM (*RHOBLI*); (ii) ObliDB hash join (OBLI) [21], (iii) Opaque sort-merge join (OPAQ) [72], and (iv) the algorithm proposed by Krastnikov et al. [42] (KRAS), which hides access patterns with a problem-specific design. As ORAM has a very large memory consumption, we used a modified version of the *cache-fit* dataset with the cardinalities of the tables reduced to 200k and 400k for R and S, respectively.

Figure 11 shows the results of this experiment. We see a two orders of magnitude difference between the ORAM-based (OBLI, OPAQ, and RHOBLI) and problem-specific (KRAS) solutions. ORAM-based joins perform poorly because of the polylogarithmic memory cost and many OCALLs per tuple. This makes ORAM "useless" for real-world applications. On the other hand, problem-specific designs introduce complex calculations but require less memory and tend to bound the execution to the secure environment only.

In contrast with earlier findings [21], we observe that the join used in Opaque performs better than the join used in ObliDB. In our experiment, OPAQ is faster than OBLI by 45%. This is because our dataset is significantly larger than the ones used in [21]. Although OBLI performs better for datasets in the ranges of a few thousand records, OPAQ outperforms it for ranges of a few hundred thousand (600k records in our case). To confirm the discrepancies, we modified the code for the original experiment [24] to match the parameters of our experiment and obtained results alike to our findings. These results underline the need for thorough and diverse experiments of novel algorithms. Last, we observe that RHOBLI

outperforms OPAQ and OBLI due to a more efficient data representation. OPAQ and OBLI store records as arrays of chars, which generates a large overhead. Nonetheless, porting secure RHO to ORAM (i. e., RHOBLI) cuts the throughput by four orders of magnitude, which is unacceptable in a realistic scenario. Note, that our threat model excludes access pattern attacks and further investigation of obliviousness is out of the scope of this paper.

**Finding:** Adding obliviousness decreases the performance of TEE joins by two to five orders of magnitude. ORAM-based solutions are not practical for larger datasets.

### 6.3 Lockless Synchronization

During parallel execution, threads synchronize to use global data structures. Process-based enclaves run on top of an OS, which is responsible for thread management and synchronization. This can easily become a bottleneck (see Section 2.1). This is the case of Intel SGX whose threads wait for mutexes outside of the enclave, generating many OCALLs. We could observe this behavior in the radix-based joins, which use mutexes to access task queues. We believe that lockless primitives can reduce such a synchronization bottleneck. We, thus, study the impact of mutexes and atomic variables on the performance of **RHO**, a radix-based join. We implemented **RHO-lockless**, which uses atomic variables for synchronization.

Figure 12 shows the results of the experiment for dataset *cache-fit*. Here, the partitions are small and the tasks last short. This makes thread management a relatively high cost for RHO. The results indicate that in the case of RHO more threads do not pay off. Once we reach the number of physical cores (4 cores), the threads start to compete for resources. However, RHO-lockless behaves differently. It scales very well with the number of threads, including hyper-threads. For a large part of the execution, the threads remain inside the enclave, which vastly reduces the cost of thread synchronization. In fact, RHO-lockless was able to reduce the number of CPU context switches by 99%. Overall, the throughput improves by 34%. This effect can be seen in scenarios where more severe bottlenecks do not occur, such as EPC paging. That is why RHO and RHO-lockless perform very similarly for dataset *cache-exceed*. Thus, we decided to leave out the results for dataset *cache-exceed*.

**Finding:** Lockless primitives significantly reduce OCALLS.

## 7 THE FIVE TWISTS

We now introduce five twists to see how the algorithms behave with different data and hardware characteristics. First, we vary the join selectivity and break the primary-foreign-key relation (Section 7.1). We anticipate higher throughput of INL for low selectivity unless it triggers EPC paging. Second, we introduce data skew (Section 7.2). High skew improves data locality by accessing the same records more frequently. We anticipate that joins with low cache hit rate (e. g., hash joins) can benefit from high skew. Third, we feed the joins with sorted data (Section 7.3). Here, PSM might improve by skipping the sort phase. Fourth, we scale the relations independently to each other (Section 7.4). This tells us if the ratio between the input tuples changes the throughput. We expect the hash-based joins to be the most sensitive because they need memory for the entire hash table.

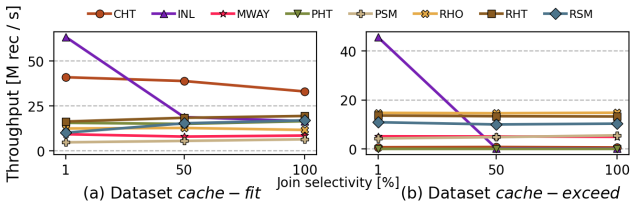


**Figure 13: Throughput when varying join selectivity.**

Fifth, we study the scalability of the join algorithms concerning the number of threads (Section 7.5).

**Synopsis:** Hash-based joins benefit from data skew, low join selectivity, and pre-sorted input. While PSM also benefits from sorted input, INL improves with low selectivity. Radix-based joins have a constant performance independently from the nature of the data. Join algorithms scale until they saturate physical CPU cores and scaling them beyond this point can have a detrimental effect.

### 7.1 Selectivity Experiment

We study the performance when varying the join selectivity. We call *join selectivity* the ratio between the number of tuples from the inner table that find a match and the cardinality of the inner table. Recall we generated the datasets with a primary-foreign-key relationship and the assumption that every tuple from the inner table finds a match. We now generate tables with the same cardinalities but with no guarantee if tuples from the outer table will be matched. We vary the selectivity from 1% to 100% for both datasets.

Figure 13 illustrates that the nested-based and hash-based families have better performance for low selectivity. INL improves highly due to the low cost of the index scan for small selectivities. Interestingly, it is the fastest algorithm for both datasets with 1% selectivity because it successfully avoids EPC paging. In the case of CHT, the CPU better predicts the right branch. In contrast, we do not observe the same benefit for the radix joins. The branch predictor starts from scratch for every partition, which makes it tough to reduce mispredictions. Except for INL, we observe throughputs similar to Section 5.1, i. e., while CHT outperforms others for *cache-fit*, RHO is the fastest for *cache-exceed*. The results are alike because the time spent on matching tuples is minimal. The majority of the execution time is spent processing the input. Recall that we do not materialize the join results to better isolate the join operation.

**Finding:** INL outperforms other algorithms for very small selectivities due to fast index scan.

### 7.2 Data Skew

Data skewness is common in real datasets. We evaluate it by considering the Zipfian distribution to represent non-uniform datasets. We generated the test datasets with the Zipf factor from 0.5 to 0.99 and measured the throughput of the joins.

We observe that CHT reacts the most to data skew (Figure 14). While the throughput of other algorithms varies slightly, CHT improves up to an order of magnitude. A high concentration of keys allows the CPU to access cached entries more often. For example,

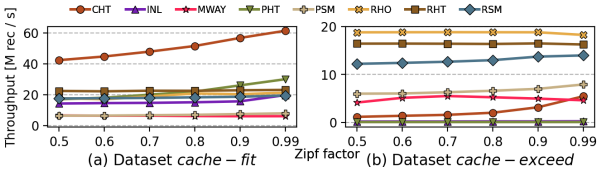


**Figure 14: Skewed data distribution.**

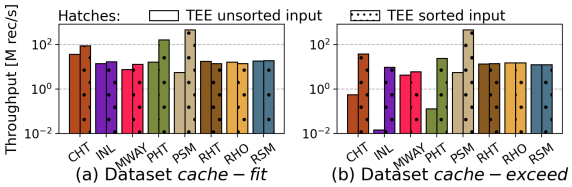


**Figure 15: Throughput with sorted and unsorted input.**

for *cache-exceed*, we observed that CHT reduces L3 cache misses and EPC paging ten-fold when the Zipf factor increases from 0 to 0.99. PHT and INL show similar behavior for *cache-fit*. However, for the larger dataset even highly skewed data does not reduce EPC paging enough to revive the algorithm. Although EPC paging decreased more than 60%, it was not enough for a noticeable change.

Sort-merge and radix-based families do not react strongly to data skew. The performance of PSM depends almost entirely on sorting. The parallel quicksort [34] used in PSM performs similarly for both uniform and skewed data. Hence, the performance of PSM is stable. In the case of radix joins, the cache hit rate is already high (i. e., up to 99%), hence, there is no space for reducing cache misses.

In essence, the algorithms perform similarly with or without data skew (see Section 5.1). Although hash-based family benefits from high skew, the overall picture does not change.

**Finding:** Hash joins benefit from high skew but retain the lowest throughput for dataset *cache-exceed*.

## 7.3 Pre-Sorted Relations

When executing more complex queries, it is not uncommon to join already sorted data. This occurs, for instance, when joining on a sorted attribute (e. g., clustered index), or after some operators in the query plan (e. g., a join that produces sorted output). Thus, we examine the throughput of the join algorithms for pre-sorted relations. We expect the sort-merge join to benefit the most from it. PSM can skip its sort phase and directly merge the tuples.

Figure 15 shows that indeed PSM (as well as hash-based and nested-based) join algorithms tremendously benefit from having pre-sorted relations. As expected, PSM achieves its high throughput due to skipping the sort phase and directly merging the tuples. The merging stage comprises only a fast linear scan of both tables. Surprisingly, we also observe a resurgent performance of the hash-based and nested-based families for dataset *cache-exceed*. This is because these algorithms re-use the cached records optimally, i. e., a record is fetched from the hash table (or index) only once and used for all the records that will match. In our experiments, we

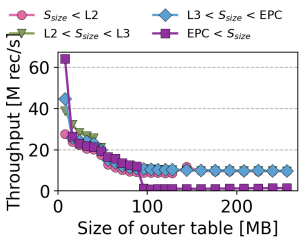


**Figure 16: CHT's throughput – scaling the outer relation.**

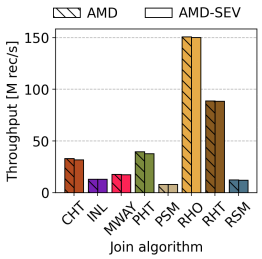


**Figure 17: Throughput with IMDb on AMD CPU.**

also observed that processing sorted input reduces memory page faults (which is the initial step of EPC paging) by 99%. This reduces the major bottleneck of these algorithms and enables CHT, PHT, and INL to process datasets *cache-fit* and *cache-exceed* with a similar throughput. On the other side, the radix-based family simply ignores whether the input is sorted.

**Finding:** All but radix-based joins benefit from sorted input.

## 7.4 Scaling Relations

We fix the size of the inner relation (S) and measure the throughput while scaling the outer relation (R). We considered four sizes of S, which cover the edge cases of the SGX caches: (1) $S_{size}$ < L2 (200 kB), (2) L2 < $S_{size}$ < L3 (6.4 MB), (3) L3 < $S_{size}$ < EPC (16 MB), and (4) EPC < $S_{size}$ (100 MB).

We observed that the hash-based joins severely drop as the outer table grows. CHT and PHT need memory for the hash table and as its size increases, the throughput decreases (Figure 16). Initially, the performance drops for all four cases, because the amount of allocated memory depends only on the size of the outer table. However, the drop for *EPC* < $S_{size}$ (—■—) at 90 MB marks where secure memory runs out and EPC thrashing starts. Interestingly, the other three cases experience no such drops. They maintain the throughput due to a smaller inner relation, such that, as it scans the inner relation, it accesses a fraction of the hash table. The rest is swapped out from EPC and never accessed. Thus, it is equally important to monitor the amount of allocated memory and the fraction of it being in use. We do not present the results of the other algorithms because we observed no effect on the throughput. While partitioning in radix joins reduces the memory allocated to a minimum, sort-merge escapes EPC paging due to *shallow copying* (Section 5.4).

We also ran the opposite to the previous experiment: We fix the size of the outer relation to the same four sizes and measure the throughput while scaling the inner relation. However, we observed no behavior related to TEEs. The throughput of the joins was primarily determined by the memory consumption with EPC as a reflection of another cache level.

**Finding:** Hash joins are susceptible to the outer relation's size.

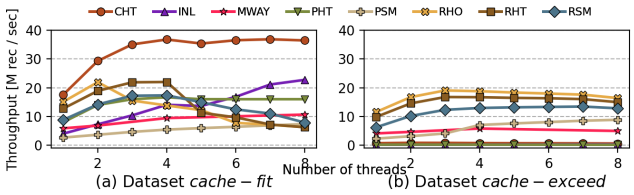


**Figure 18: Scaling joins with the number of threads.**

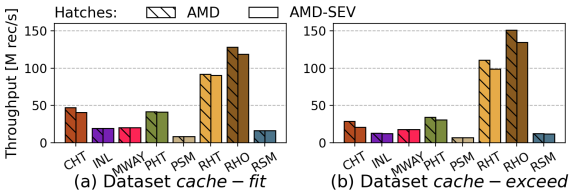


**Figure 19: Join algorithms on AMD and AMD SEV.**

## 7.5 Scaling Threads

We start by varying the number of threads used by a join algorithm. Figure 18 illustrates the results of this experiment. For the *cache-fit* dataset, the throughput of hash-based and sort-merge joins increases with the number of used threads (including hyper-threads). Both sorting the relations and accesses to hash tables are sources of random memory access. They cause the threads to wait for the data and create idle periods, which can be filled using hyper-threads. In addition, the nested-based algorithm scales almost linearly thanks to fast, concurrent access to the index. Interestingly, radix-based joins improve only until reaching the number of physical cores (i.e., three, as one core is taken by the untrusted app) and then decline. There are two reasons behind this performance drop. First, radix-based joins depend on task queues, which synchronize outside of the enclave using mutexes. This leads to excessive OCALLs, which were identified as bottlenecks (see Section 2.1). This could be partially mitigated by using busy-waiting primitives that avoid context-switching (see Section 6.3). Second, the cost of thread maintenance dominates when processing small datasets, such as *cache-fit*.

In contrast, for *cache-exceed* (Figure 18(b)), the radix-based family does not suffer significant performance drops using hyper-threads. Yet, they peak using only three threads. As the CPU caches are optimized, radix joins cannot benefit much from hyper-threads. PSM performs similarly for both datasets: It does not trigger EPC paging. Finally, hash- and nested-based joins maintain low throughput because multi-threading does not alleviate EPC thrashing.

Lastly, we tested if CPU affinity, i. e., pinning a thread to a core, changes the performance. CPU affinity often improves cache efficiency and has been widely used in join implementations [7, 8, 41, 58, 71]. As of today, SGX SDK does not support this functionality, which we implemented in our release [25]. In the experiments, we observed that PSM is the only algorithm that benefits from CPU affinity. It cuts L2 cache misses by half because threads never jump between physical cores. Other algorithms do not improve because their costs are dominated by EPC paging. Therefore, CPU affinity helps only if other bottlenecks are eliminated.

**Finding:** All algorithms benefit from more threads (similar to plain CPU) but radix joins only until reaching the number of physical cores. Other families benefit from hyper-threads due to a higher cache miss ratio.

# 8 OTHER HARDWARE PLATFORMS

We turn our attention to *VM-based* enclaves. We mentioned in Section 2.1 that the focus of this work is the *process-based* architecture. However, we look into the performance of other designs to better understand secure enclaves as a whole. An alternative to Intel SGX for the public cloud is AMD Secure Encrypted Virtualization (SEV) [2]. SEV builds on the *VM-based* model, which costs and benefits were discussed in Section 2.1. Note that there are successful attacks against the SEV's encryption mechanisms [13, 14]. Moreover, some versions lack memory integrity protection. This boils down to (i) users being incapable of proving that their code runs in a secure enclave and (ii) hypervisors being able to manipulate the encrypted pages without the enclave noticing. Despite its limitations, SEV has been reported to have low performance overhead [29]. Although the technology does not satisfy our threat model, it is worth seeing how it performs under similar conditions. We compared AMD SEV with native AMD (similarly to Section 5.1). For these experiments, we used two Google Cloud VMs with AMD EPYC 7B12 CPU, 8GB RAM, and Ubuntu 18.04.5 OS. One machine had SEV disabled.

The results (Figure 19) are interesting in several ways. First, the performance of native AMD execution is similar to native Intel execution (Figure 4). RHT and RHO (both radix-based) clearly outperform other algorithms. Second, the performance of AMD SEV is only slightly worse than the native. This is due to offloading the memory encryption cost to a secure coprocessor. Third, the differences between datasets *cache-fit* and *cache-exceed* are small. AMD SEV is not as sensitive to memory consumption as Intel SGX because it uses the entire main memory for storing encrypted pages. However, this comes at the price of a much larger attack surface.

We examined the performance of AMD SEV for a real dataset. Similar to Section 5.1, we used the IMDb dataset. The results for the real dataset (Figure 17) resemble the results for synthetic datasets. We observe that the performance of AMD SEV is, again, only slightly lower than plain AMD. The *VM-based* model has the potential for killing two birds with one stone - providing native performance and hardware security. However, it is still considered a niche in the CPU market. The research community has proposed techniques to mitigate the two security flaws - memory integrity protection [64] and remote attestation [14].

**Synopsis:** AMD SEV has a performance comparable to native execution. However, it lacks critical security features and has not been widely adopted.

# 9 THE 7 LESSONS LEARNED

Our experimental study left us with seven lessons learned:

**(1) CHT for small data and RHO for large.** Hash joins performed better in all experiments with dataset *cache-fit*. For instance, in Section 5.1 secure CHT achieved only an order of magnitude

worse performance than insecure CHT. However, for *cache-exceed*, radix joins (RHT and RHO) always outperform other algorithms.
**(2) RHO is a safe choice.** RHO maintained similar performance across all inputs. Across all experiments, it is the fastest algorithm for *cache-exceed* and performs relatively well for small data. If it cannot be ensured that the data fit the EPC, RHO is a safe bet.
**(3) Avoid EPC paging at all costs.** Although paging expands the very small EPC memory, it comes with a prohibitive cost. In all experiments, even low paging had a major impact on the performance. In the cases of CHT and PHT, EPC paging made the algorithms unusable for dataset *cache-exceed*. We recommend designing the relational operators without exceeding the EPC. For recent SGX implementations, this limits the memory usage to at most 90 MB.
**(4) Use all available physical cores but no locks**. Multithreading can significantly improve throughput. However, while hyperthreads may cause drastic performance cuts locks lead to excessive OCALLs. We recommend using only physical cores and no locks.
**(5) Consider secure enclave operations**. Data sealing and tuple reconstruction can increase the cost of the operator 5×, which can be reduced with chunking. An optimizer building TEE query plans should increase pipelining and late encryption and materialization.
**(6) Double-check before hiding data access patterns.** Obliviousness can cut the performance of a join operator by five orders of magnitude. Besides, learning facts from access patterns is tedious and impractical for most cases. Decide for obliviousness only if your use-case requires no data access patterns leakage.
**(7) Stay hardware-conscious.** TEEs are rapidly changing. While AMD improves the security of SEV, Intel developed TME [35]. TME encrypts the entire main memory and, potentially, removes EPC paging. Yet, till today, little is known how TME works with SGX.

## 10 RELATED WORK

There are three types of related work: (i) works that focus on relational joins; (ii) works that combine TEEs and data processing systems; and (iii) tooling for performance measurement on TEEs.
**Relational Joins** is a fundamental topic that has been studied for decades [7–9, 11, 12, 19, 20, 41, 43, 58, 71]. DeWitt et al. [19] analyzed the costs of in-memory joins long ago. However, Boncz et al. [12] were the first to measure the influence of the memory address-translation cache (TLB) on the join performance and proposed radix partitioning as a solution. Kim et al. [41] included modern hardware features, such as wider SIMD instructions, in the hash and sort joins. Later, Balkensen et al. [7] improved their sort-merge join by reducing thread synchronization and memory bandwidth. Blanas et al. [11] demonstrated high performance of no-partitioning hash joins. Balkesen et al. [8] improved [11] through hardware-conscious design, i. e., reconsidering TLB and cache misses. Lang et al. [43] further advanced the hash join algorithm for NUMA architectures. Barber et al. [9] designed a concise hash table to reduce the memory consumption of hash joins. We consider these latest advances in join algorithms in our evaluation [8, 9, 11, 41]. Secure implementations of relational join have also been studied [1, 6, 42, 45]. Agrawal et al. [1] proposed joins that hide access patterns but have high complexity. Li et al. [45] reported a security issue in [1], which they fixed but the complexity remained high. Arasu et al. [6] and Krastnikov et al. [42] proposed joins for oblivious processing and achieved much lower complexity. We include the latest of them [42] in our study. Others have also evaluated join algorithms on plain CPU [58] and in a streaming scenario [71]. However, no existing work studies how state-of-the-art algorithms perform in TEEs.
**TEE-based DBMSes** are quickly emerging as data protection has become a key factor in designing modern systems [5, 21, 42, 48, 57, 63, 72]. Opaque [72] is a secure analytics platform for a distributed setup and ObliDB [21] is a secure database engine with obliviousness guarantees. However, both assume an oblivious memory area provided by the hardware. This is not the case for some TEEs (e. g., Intel SGX). EnclaveDB [57] is a secure DBMS engine based on Intel SGX but with no data access patterns protection. Oblix [48] is a *doubly*-oblivious search index, i. e., it does not require the enclave's internal memory to be oblivious to hide data access patterns. Always Encrypted [5] is an add-on for Microsoft SQL Server that provides data confidentiality for high-sensitivity columns. Enclage [63] is the first enclave-native storage engine. Only the first two works [21, 72] introduce novelties to the field of join processing. We include join algorithms from both in our study. In summary, TEEs have attracted the attention of both the systems and database communities. However, none of these works provides insights into how different join algorithms perform in TEEs.
**Profiling Intel SGX** has not been the focus of the study of many works and hence only a few tools are currently available to address this need. The SGX SDK comes with *sgx-gdb*, an extension for the GNU Debugger that reports the total memory consumption of enclaves. VTune Profiler [36] supports the analysis of *sgx-hotspots* that mark frequently executed pieces of code. *sgx-perf* [68] is a shared library for performance analysis. Finally, Graphene-SGX [66] is a library OS that runs unmodified applications in secure enclaves and provides statistics on executed programs. Although these tools are designed to be general, our profiling framework is specialized for analyzing the performance of relational operators.

## 11 CONCLUSION

We highlighted the importance of confidential computing in data processing and pointed out the lack of tools to understand the performance problem of TEEs. We introduced TEEBENCH, a framework for benchmarking any relational operator in multiple TEEs. Using the framework, we conducted a deep study of major join algorithms running in secure enclaves. Through our experiments, we observed that existing join algorithms underperform in secure enclaves. We showed that they can be up to three orders of magnitude slower in TEEs than on plain CPUs. We identified surprising findings, such as risks of using no-partitioning joins and adaptability of RHO. To achieve performance close to native, we need novel, hardware-aware algorithms. Our seven lessons learned can help to design efficient algorithms for secure enclaves and improve the adoption of confidential computing in databases. As future work, we are designing a secure join algorithm that utilizes the full potential of TEEs and integrating the findings into a database optimizer.

## ACKNOWLEDGMENTS

This work was funded by the German Ministry for Education and Research as BIFOLD – Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

## REFERENCES


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

[2] AMD. 2021. *AMD Secure Encrypted Virtualization (SEV)*. Retrieved May 18, 2021 from https://developer.amd.com/sev/

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13. ACM New York, NY, USA, 7.

[4] David P Anderson and Gilles Fedak. 2006. The computational and storage potential of volunteer computing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, Vol. 1. IEEE, 73–80.

[5] 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.

[6] Arvind Arasu and Raghav Kaushik. 2013. Oblivious query processing. In *Proceedings of the 17th International Conference on Database Theory*. 26–37.

[7] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[8] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362–373.

[9] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *Proceedings of the VLDB Endowment* 8, 4 (2014), 353–364.

[10] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 837–849.

[11] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 37–48.

[12] Peter A Boncz, Stefan Manegold, and Martin L Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases*. 54–65.

[13] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. *arXiv preprint arXiv:2108.04575* (2021).

[14] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. 2019. Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[15] 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*. 668–679.

[16] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1313–1324.

[17] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[18] cplusplus. 2021. . Retrieved May 31, 2021 from https://www.cplusplus.com/reference/stl/

[19] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*. 1–8.

[20] Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2014. NADEEF/ER: generic and interactive entity resolution. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1071–1074.

[21] Saba Eskandarian and Matei Zaharia. 2020. Oblidb: Oblivious query processing using hardware enclaves. *Proceedings of the VLDB Endowment* 13, 2 (2020), 169–183.

[22] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1001–1012.

[23] Raul Castro Fernandez, Pranav Subramaniam, and Michael J Franklin. 2020. Data market platforms: Trading data assets to solve data problems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1933–1947.

[24] Github. 2021. *An Oblivious General-Purpose SQL Database for the Cloud*. Retrieved May 18, 2021 from https://github.com/SabaEskandarian/ObliDB

[25] Github. 2021. *Intel SGX for Linux*. Retrieved Sep 14, 2021 from https://github.com/agora-ecosystem/linux-sgx/tree/2.11-exp-multithreading

[26] Github. 2021. *Intel SGX Linux Driver*. Retrieved Sep 14, 2021 from https://github.com/agora-ecosystem/linux-sgx-driver/tree/ewb-monitoring

[27] Github. 2021. *TEE Relational Operator Benchmark Suite*. Retrieved Sep 14, 2021 from https://github.com/agora-ecosystem/tee-bench

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

[29] Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2018. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 133–142.

[30] 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.

[31] Shay Gueron. 2016. A memory encryption engine suitable for general purpose processors. *IACR Cryptol. ePrint Arch.* (2016).

[32] IMDb. 2020. *IMDb Datasets*. Retrieved December 18, 2020 from https://www.imdb.com/interfaces/

[33] Intel. 2020. . Retrieved May 31, 2021 from https://download.01.org/intel-sgx/sgx-linux/2.11/

[34] Intel. 2020. *An Efficient Parallel Three-Way Quicksort Using Intel C++ Compiler And OpenMP 4.5 Library*. Retrieved May 18, 2021 from https://software.intel.com/content/www/us/en/develop/articles/an-efficient-parallel-three-way-quicksort-using-intel-c-compiler-and-openmp-45-library.html

[35] Intel. 2021. *Intel Architecture Memory Encryption Technologies*. Retrieved May 18, 2021 from https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf

[36] Intel. 2021. *Intel VTune Profiler*. Retrieved May 18, 2021 from https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html

[37] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access pattern disclosure on searchable encryption: ramification, attack and mitigation.. In *Ndss*, Vol. 20. 12.

[38] 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*. 1329–1340.

[39] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2015. Lightning Fast and Space Efficient Inequality Joins. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2074–2085.

[40] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2017. Fast and scalable inequality joins. *VLDB J.* 26, 1 (2017), 125–150.

[41] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389.

[42] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient oblivious database joins. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2132–2145.

[43] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively parallel NUMA-aware hash joins. In *In Memory Data Management and Analysis*. Springer, 3–14.

[44] Jin Li, Yan Kit Li, Xiaofeng Chen, Patrick PC Lee, and Wenjing Lou. 2014. A hybrid cloud approach for secure authorized deduplication. *IEEE Transactions on Parallel and Distributed Systems* 26, 5 (2014), 1206–1216.

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

[46] Chang Liu, Liehuang Zhu, Mingzhong Wang, and Yu-an Tan. 2014. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences* 265 (2014), 176–188.

[47] Kajetan Maliszewski. 2020. Secure Data Processing at Scale. *Proceedings of the VLDB PhD Workshop* (2020).

[48] 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.

[49] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. 2011. Can homomorphic encryption be practical?. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. 113–124.

[50] 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*. 644–655.

[51] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A survey of published attacks on intel SGX. *arXiv preprint arXiv:2006.13598* (2020).

[52] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the Twelfth European*

*Conference on Computer Systems*. 238–253.

[53] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 359–374.

[54] PassMark. 2021. . Retrieved Sep 6, 2021 from https://www.cpubenchmark.net/market_share.html

[55] 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.

[56] David Pouliot and Charles V Wright. 2016. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 1341–1352.

[57] 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.

[58] 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.

[59] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs.. In *NDSS*.

[60] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 317–328.

[61] Jatinder Singh, Jennifer Cobbe, Do Le Quoc, and Zahra Tarkhani. 2020. Enclaves in the Clouds: Legal considerations and broader implications. *Queue* 18, 6 (2020), 78–114.

[62] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 299–310.

[63] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1019–1032.

[64] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 665–678.

[65] Jonas Traub, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Agora: Bringing together datasets, algorithms, models and more in a unified ecosystem [vision]. *ACM SIGMOD Record* 49, 4 (2021), 6–11.

[66] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 645–658.

[67] Shiyuan Wang, Divyakant Agrawal, and Amr El Abbadi. 2012. Is homomorphic encryption the holy grail for database queries on encrypted data? (2012).

[68] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*. 201–213.

[69] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

[70] Kehuan Zhang, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. 2011. Sedic: privacy-aware data intensive computing on hybrid clouds. In *Proceedings of the 18th ACM conference on Computer and communications security*. 515–526.

[71] Shuhao Zhang, Yancan Mao, Jiong He, Philipp M Grulich, Steffen Zeuch, Bingsheng He, Richard TB Ma, and Volker Markl. 2021. Parallelizing Intra-Window Join on Multicores: An Experimental Study. In *Proceedings of the 2021 International Conference on Management of Data*. 2089–2101.

[72] 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 *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. 283–298.