Cracking-Like Join for Trusted Execution Environments

Kajetan Maliszewski
Technische Universität Berlin
maliszewski@tu-berlin.de

Jorge-Arnulfo Quiané-Ruiz*
ITU Copenhagen
joqu@itu.dk

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

ABSTRACT

Data processing on non-trusted infrastructures, such as the public cloud, has become increasingly popular, despite posing risks to data privacy. However, the existing cloud DBMSs either lack sufficient privacy guarantees or underperform. In this paper, we address both challenges (privacy and efficiency) by proposing CrkJoin, a join algorithm that leverages Trusted Execution Environments (TEE). We adapted CrkJoin to the architecture of TEEs to achieve significant improvements in latency of up to three orders of magnitude over baselines in a multi-tenant scenario. Moreover, CrkJoin offers at least 2.9x higher throughput than the state-of-the-art algorithms. Our research is unique in that it focuses on both privacy and efficiency concerns, which has not been adequately addressed in previous studies. Our findings suggest that CrkJoin makes joining in TEEs practical, and it lays a foundation towards a truly private and efficient cloud DBMS.

PVLDB Reference Format:
Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. Cracking-Like Join for Trusted Execution Environments. PVLDB, 16(9): 2330-2343, 2023.
doi:10.14778/3598581.3598602

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

1 INTRODUCTION

Many applications, particularly in the medical and financial domains [53], prioritize data confidentiality and are subject to strict governmental regulations (e. g., GDPR or CCPA) [20, 24, 64]. This leads to on-premise solutions that do not make use of the cost, management, and implementation benefits of the public cloud. Therefore, there is a growing need for data systems that clearly define privacy boundaries for both datasets and query execution. One important query processing operation on relational data, which requires privacy preservation, is the relational join operator [26].

The research community has proposed software- and hardware-based solutions to the problem of joining data securely. While software solutions employ different encryption schemes [2, 27], hardware solutions exploit dedicated hardware modules co-located [31] or within [18, 35] a CPU. Software-based solutions, e. g., encrypted databases [7, 59, 71], encrypt the data on trusted machines and process the ciphertext on untrusted machines. However, they have not been widely adopted by the industry because of large overheads [29, 54]. Due to this hardware manufacturers have introduced Trusted Execution Environments (TEEs) [18, 35], which are CPU extension that provides hardware guarantees for the confidentiality and integrity of code and data. TEEs offload the encryption burden to an on-chip component (e. g., Memory Encryption Engine [30]) and execute the code using optimized CPU instructions. As a result, TEEs significantly outperform software-based solutions.

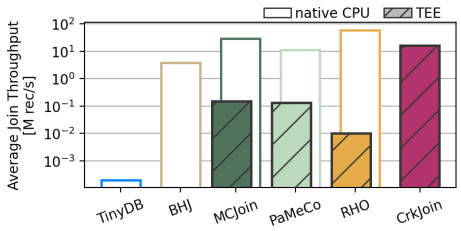

Figure 1: TEE vs. CPU: a performance problem.

Although TEEs have the potential to bridge security and high performance, they are far from ideal; If used incorrectly, their associated costs can quickly get out of hand. This can lead to drastic performance regressions, easily by orders of magnitude, compared to sole CPU performance [6, 18, 47, 67]. The key to high performance of a TEE algorithm is memory consumption that fits the secure cache and little (or no) interaction with the OS [47, 66]. However, state-of-the-art TEE-based join algorithms have not exploited TEEs' architecture to reach truly high performance. For example, while EnclaveDB [60] ignores the increased costs of secure threads, ObliDB's [23] join algorithm does not scale to larger datasets [47]. Moreover, general memory-constrained join algorithms [12, 14, 44] are unsuitable for TEEs, as there are further unconsidered bottlenecks, such as memory access patterns and thread management.

To illustrate the adequacy and efficiency problem of state-of-the-art join algorithms on TEEs, we ran an experiment with a public cloud scenario, where a machine runs multiple, concurrent queries:[1] We assumed a degree of parallelism of eight, which is a common default value for industry warehouses [65]. We ran the experiment for native CPU (insecure) and Intel SGX (a popular TEE). Figure 1 shows the average throughput of the most important TEE baselines [47]. Overall, we observe that all existing solutions underperform on SGX by a few orders of magnitude. RHO [9], a highly-optimized radix join, performs well in the insecure setting, but its performance drops sharply for TEEs: It is four orders of magnitude slower due to the architectural differences [15, 18]. TinyDB [44] (i. e., nested-loop join) and BHJ (Section 2.3) consume little to no memory, a scarce resource in TEEs. Yet, due to their complexity, they fail to complete the task in the given time on SGX.

*Work done while at Technische Universität Berlin.

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598602

[1] The experimental setup is described in Sections 7.1 and 7.2

MCJoin [12] and PaMeCo [11] adapt to memory-constrained hardware. They improve the SGX performance of RHO by one order of magnitude by avoiding Enclave Page Cache (EPC) paging. However, excessive thread creation (a major TEE bottleneck) prevents them from achieving higher performance; Their CPU versions perform two orders of magnitude better. In summary, we observe that a hardware-conscious design is the right direction and that none of the existing joins properly addresses all TEE's bottlenecks.

It is thus crucial to design an algorithm with the TEEs' architecture in mind to enable efficient and private data processing systems. Our algorithm (Cracking-Like Join—CrkJoin for short) shows that, when deeply considering the underlying hardware, one can achieve far superior performance. In the above experiment, CrkJoin outperforms the TEE baselines by at least two orders of magnitude; It is in the same order of magnitude as the baselines' CPU performance.

Yet, designing such a TEE algorithm is challenging for several reasons. First, TEEs introduce limitations to the programming model. Many widely-used functionalities (e. g., system calls) and libraries pose a security thread and are disabled in TEEs. Therefore, we need to adapt building systems to fit the new environment. Second, based on our study of the TEE bottlenecks [47], achieving superior performance requires novel designs that obey the strict rules of secure enclaves, e. g., drastically reduced OS interaction. Third, it is not obvious which new designs will perform best in TEEs; The community has not established guidelines for building algorithms.

CrkJoin (our proposal) tackles the above challenges by fully exploiting the underlying TEE architecture. It achieves a very low memory footprint by incrementally organizing relations in-place and comes with a no-synchronization design to take advantage of multiple cores. CrkJoin draws inspiration from the gradual tuple reorganization pioneered in database cracking [32] but exploits reorganization within a single operator rather than across hundreds of queries. Our experimental evaluation shows that CrkJoin improves the query latency by up to three orders of magnitude compared to state-of-the-art join algorithms in a multi-tenant scenario.

In summary, we make the following contributions:

(1) We revisit the bottlenecks of Intel SGX (a popular TEE architecture) and derive the desiderata for TEE-native data processing (Section 2). We identify access patterns, memory consumption, and thread contention as the key elements of efficient TEE algorithms.
(2) We introduce the "cracking-like" philosophy, a set of processing principles for TEEs (Section 3). Its goal is to set up the base for TEE-native algorithms. The philosophy follows an iterative process that gradually reorganizes the relations. It always obeys the desiderata.
(3) We propose a cracking-like join algorithm (CrkJoin), the first TEE-native join algorithm (Section 4) with a unique threading model (Section 5). Our algorithm instantiates the new processing philosophy, therefore, complying with the desiderata.
(4) We implement CrkJoin (Section 6) and extensively evaluate it on Intel SGX (Section 7). We show that CrkJoin significantly outperforms the state-of-the-art join algorithms and demonstrates robustness and good scalability in multi-threaded environments.

2 DATA PROCESSING IN ENCLAVES

We now provide an overview of TEEs (Section 2.1) and further motivate the need for a TEE-native join algorithm. We revise the bottlenecks of a popular TEE architecture (Section 2.2) and discuss why we can use neither existing join algorithms nor a naive approach for memory-constrained joins (Section 2.3). Finally, we derive the desiderata for TEE-native data processing (Section 2.4).

2.1 Trusted Execution Environments

A *Trusted Execution Environment* (TEE) [3, 19, 35, 41, 57] is a set of instructions inside a CPU, which guarantees code and data confidentiality and integrity. These guarantees protect against malicious processes with high privilege levels, including the OS or the hypervisor. Therefore, TEEs are a suitable safety and privacy measure for untrusted environments, such as the public cloud. TEEs are implemented in hardware and allow users' code to create secure threads and allocate memory in a secure memory area, i. e., in Enclave Page Cache (EPC), which can be as low as 90 or 256 MB. TEEs instantiate as *enclaves*. They communicate with the OS through ECALLs (enclave calls) and OCALLs (outside calls). These calls are defined by the application developer and carefully designed to expose no sensitive information to the OS. Intel Software Guard Extensions (SGX) [18, 51] is the most widely adopted TEE in public cloud setups. Therefore, we focus on the architecture of SGX. We particularly focus on SGXv1 because it is the dominant enclave in production as of today. Yet, our findings apply also to SGXv2 [50].
Threat Model. We inherit the security and privacy guarantees of TEEs. Similarly to [4, 66], we protect against a *strong adversary*, i. e., an actor with privileged OS and infrastructure access. The actor can perform memory snapshots, monitor its access patterns, and network communication. However, they can neither access enclave's state nor analyze enclave's computation. TEEs are infamous for side-channel attacks [17, 25, 28]. Yet, the community has proposed mitigations to some vulnerabilities [62, 63]. We exclude these attacks as we expect them to be fixed. We assume operational data confidentiality (as in [4, 66]). We allow access patterns leakage as these attacks are impractical and costly to mitigate [23, 52, 64, 72]. Instead, we focus on practicality and high performance.

2.2 Revisiting the Bottlenecks of Intel SGX

Previous works have studied the performance of SGX [6, 47, 67, 69] and identified two major bottlenecks: EPC misses and OS interaction. Yet, we further identify bottlenecks relevant to data processing – *Secure Memory Access* and *Multi-threading* – that motivated the design for the algorithm presented in the following sections.
Secure Memory Access on SGX is expensive. Others showed that EPC introduces high access costs compared to traditional main memory [6]; An EPC miss reaches 40K CPU cycles [67] compared to 170 CPU cycles for main memory access on our machine. To illustrate this further, we have conducted an experiment to compare access cost of SGX and plain CPU for data structures commonly used in databases: (i) sequential table scan, (ii) random table access, (iii) hash map access, (iv) index (B-tree) scan, and (v) index search.

Figure 2a shows the relative SGX performance compared to the native CPU, i. e., it shows the performance degradation as a function of data size. We observe that all operations suffer significantly not only from out-of-cache accesses but also partly from L3 misses; They perform up to one order of magnitude worse between L3 and EPC (yellow zone), and over two orders of magnitude worse

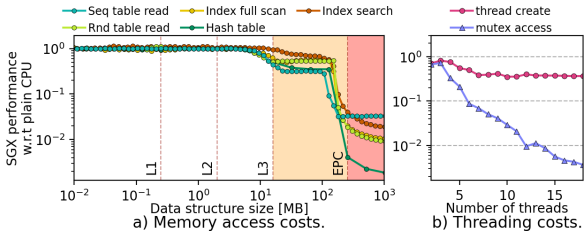

Figure 2: Bottlenecks of Intel SGX.

Table 1: Memory access time for 256MB structures.

Operation	CPU Time [ns/elem]	SGX Time [ns/elem]
seq read	1.31	39.55
rnd read	79.50	4206.32
index scan	4.25	253.50
index search	317.51	8284.86
hash table	16.24	3967.81

after exceeding the EPC (red zone). This tells us that, while the data resides in caches, we are allowed to access them randomly without a performance penalty. Additionally, the absolute values of the measurements (Table 1) show that we should avoid any random access (i. e., random access, index search, and hash table access) even within the EPC. For example, index search, which uses binary search to find a key, is the slowest operation and reaches the access time close to $9\mu s/elem$. Although SGXv2 comes with a larger EPC, it does not solve the memory access problem. Instead, it extends the underperforming yellow zone and only delays further degradation by entering the red zone to larger datasets [22]. This experiment shows that we can achieve good memory access performance in TEEs with two rules: (i) access out-of-cache data sequentially, and (ii) keep random access data structures only in the CPU caches.

Multi-threading on SGX is nontrivial. Previous studies showed that TEEs struggle to achieve optimal performance during parallel execution [6, 47], caused by SGX threading primitives eventually being served by the OS. For instance, a mutex performs an OCALL to enter non-busy waiting (i. e., sleep). A naive mitigation would be to execute only busy waiting. Yet, non-critical services should avoid busy waiting as it leads to resource contention. SGX thread creation and mutex access both require a system call. We ran an experiment to illustrate the cost of these two operations compared to plain CPU when varying the number of threads. First, we measured the cycles taken to create empty threads. Second, we created threads that perform additions on a counter protected by a mutex. These experiments isolate the investigated operations to the highest extent. We detail the experimental setup in Section 7.1.

Figure 2b shows the results. We observe that with more threads the performance of SGX mutexes deteriorates. This is caused by the growing number of OCALLs that needs to be served by the OS, which in turn leads to expensive context switches and TLB flushes [18]. Although with relatively constant performance, thread creation leads to up to 3× worse performance on SGX. These results lead us to the following takeaways: Users should (i) limit excessive thread creation, and (ii) carefully use synchronization primitives.

The **takeaways** from both experiments indicate why the canonical joins underperform on SGX. In a previous study [47], we showed

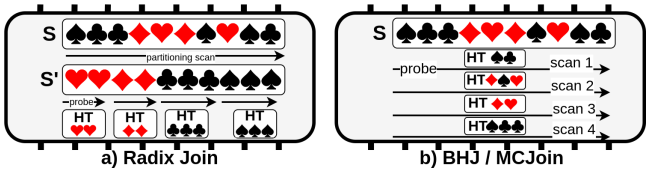

Figure 3: Probe table (S) scans (arrows) and temporal memory consumption (white boxes) of existing join algorithms. Card suits represent partitions.

that hash join and radix join perform better than other state-of-the-art joins on SGX. Nonetheless, while scaling the number of threads and the dataset size, we exposed their limitations. We can now conclude that these limitations come from not adhering to SGX's architecture; While radix join uses large amount of memory to store partitioned relations, hash join performs random accesses to the hash table that, in most cases, lead to EPC trashing. Figure 3a shows how radix join scans the probe relation and uses the memory for probing and storing hash tables of the build relation. Although it scans the tables only a few times, it needs double the dataset size of memory to store partitioned tables. We leave out the build relation of the figure to improve the figure's clarity.

2.3 Limitations of a Naive Approach

We now analyze a naive approach to tackle the two aforementioned bottlenecks and explain why it does not solve the problem either.

One (naive) approach to cut memory consumption is to modify the block-nested loop join. We can split the build relation to multiple blocks and build a hash table on one block at a time. This technique reduces the size of the hash table to a fraction of its original size. However, we introduce a full scan of the probe relation for each hash table, effectively, repeating the probe operation many times. Moreover, if we try to fit the hash table into the CPU caches (i. e., typically up to 20 MB), we reach hundreds of full scans of the probe relation. We implemented Block-Nested Hash Join (BHJ) that uses this method. BHJ builds a hash table on a block of the build relation and probes it against the entire probe relation. It then repeats the operations for the remaining blocks of the build relation. MCJoin [12] and PaMeCo [11] work in a similar block-nested fashion. However, they use compression to fit more tuples in a single block and reduce the size of the data structures. Figure 3b shows how BHJ and MCJoin scan the probe relation. Although they reduce the memory consumption, they also introduce multiple full scans of the relation. In addition, the block-nested approach recreates threads for each scan, which leads to a threading bottleneck as identified in the previous section.

On the one hand, the existing solutions mitigate the memory bottleneck. On the other, they introduce new bottlenecks. We argue that the mitigations do not have to collide: Algorithms can achieve high performance with a design that deeply exploits TEEs' design.

2.4 Desiderata

Based on the previous discussions, we identify the desiderata for TEE-native data processing that fully exploits TEE's design.

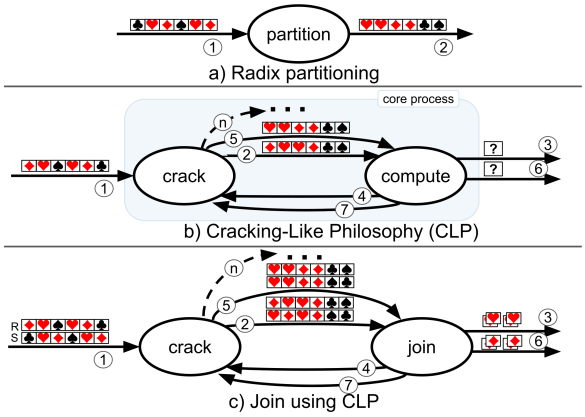

Figure 4: Stages in radix partitioning and CLP. Numbers indicate the order of execution and card suits indicate partitions. While the former does everything in one stage, the latter iteratively reorganizes the data in small stages. CLP can be a foundation for a join algorithm.

D1. Access patterns. In TEEs, the contrast between the performance of sequential and random accesses magnifies. The absolute values of the data used in Figure 2a tell us that while native CPU random access is up to 65× slower than sequential scan, for TEEs, random access is more than 260× slower than sequential scan. Across platforms, sequential scan maintains its performance within one order of magnitude but the performance of random access quickly reduces by two orders of magnitude compared to plain CPU. Therefore, TEE operators should avoid random access even if the price is to read more data via multiple sequential accesses.
D2. Low memory consumption. Large data structures trigger expensive EPC paging. This has been identified as a main bottleneck both in Figure 2a and in related work [45, 66, 67]. It is thus important to significantly reduce memory consumption to avoid EPC paging.
D3. Wait-free algorithms. Threads are inherently managed by the OS. However, communication between secure enclaves and the OS can be both insecure and inefficient. Simple mitigations, such as using spinlocks instead of mutexes, are impractical: Blocking primitives quickly lead to thread contention. Wait-free thread designs are thus crucial to minimize the interaction with the OS.

3 CRACKING-LIKE PHILOSOPHY

TEEs are challenging as they drastically change the code execution. With such limited resources, to some, TEEs can appear as machines from the past. As discussed earlier, programs must eliminate random I/Os, consume little memory, and use barrier-free data structures to fully benefit from the hardware. Algorithms that do not adhere to these rules pay a significant performance price [6, 47, 66, 67]. Yet, the same studies demonstrate techniques, e. g., data partitioning or compression, that greatly reduce the penalty.

We introduce Cracking-Like Philosophy (CLP), a TEE-native processing philosophy to tackle all aforementioned SGX bottlenecks by iteratively organizing relations. We call the philosophy *cracking-like* because it self-organizes the data, similar to database cracking [32]. However, in contrast to database cracking, CLP does not need hundreds of queries to benefit query processing [33]: It exploits tuple reorganization within a single query. The core idea is to create *partitions*, i. e., groups of tuples, that share the same bit mask in their keys. Specifically, it gradually partitions (i. e., physically reorganizes) in-place the input tuples using simple and optimized operations. In each iteration, it performs one step of partitioning on a relation slice. We call this process relation *cracking*.

It is worth noting that CLP aims at using only sequential relation scans, consequently, addressing two bottlenecks: random I/Os and memory consumption. Furthermore, in each iteration, it builds self-contained partitions that can be processed by independent threads, which tackles the third bottleneck: wait-free execution. Within cracking iterations, we perform computation pertinent to the algorithm, e. g., a join algorithm finds matches between tuples.

Other partitioning algorithms, such as radix partitioning (Figure 4a), process relations at once, allocating a large amount of memory and performing random writes. In contrast, cracking selects slices of relations with relevant tuples and partitions them into small, incremental stages. A tuple is relevant if it belongs to the currently processed partition. Figure 4b illustrates the CLP approach, where the numbers indicate the order of steps and the card suits represent partitions. CLP is composed of an iterative core process (blue box). It iterates over each partition and alternates between two main phases: (i) it *cracks* the relation, i. e., it performs one partitioning iteration (crack node), and (ii) it *computes* the output, i. e., it executes the relational algorithm of choice on a single partition (compute node). We discuss the details of cracking in Section 4.

CLP applies easily as the main building block of relational algorithms. For example, a join algorithm implements a matching function in the compute phase (Figure 4c) and iteratively outputs pairs of tuples. The algorithm inherently mitigates the SGX bottlenecks with cracking. It then piggybacks on the partitioning scans to perform a join operation on a single partition that always fits in CPU caches. Although we now focus on joins, one can implement all relational algorithms following the CLP.[2] For instance, aggregations and group-bys are intrinsically partitioned; They, thus, fit this philosophy. Further, selection and projection can execute the compute phase within the cracking phase. Note that, the database optimizer should consider that CLP does not provide order-preserving guarantees due to its in-place nature. Even though CLP operators scan some tuples multiple times, they do it to keep the memory footprint at a minimum. As we will see next, the tradeoff between the number of scans and the memory access and consumption pays off.

4 CRACKING-LIKE JOIN ALGORITHM

We now instantiate the CLP into a join algorithm – We propose CrkJoin. The algorithm follows the ideas presented in Section 3: Its core is an iterative process that gradually *cracks* the relations with a TEE-native partitioning algorithm and *computes* tuple matches within those partitions. Yet, strictly following the CLP is far from trivial. SGX's bottlenecks (Section 2.2) impose severe limitations on new designs. Algorithms must avoid costly operations (e. g., random access), exploit hardware accelerators (e. g., prefetchers), and drastically cut memory consumption. We, thus, call for new primitives that achieve superior performance on this novel hardware.

[2] We reserve the implementation of these operators for future work.

In the following, we explain how CrkJoin overcomes these challenges. We first define cracking relations and its primitives (Section 4.1). Next, we depict how it performs a join operation (Section 4.2). We, then, explain the dynamic selection of partitioning bits (Section 4.3). We finish with a cost analysis (Section 4.4).

4.1 Cracking Relations

Data partitioning is at the heart of CrkJoin when cracking relations. In such a memory-constrained setup, partitioning is crucially important; it is an effective way of avoiding large data structures. Doing it efficiently inside a secure enclave is particularly challenging. A TEE-native partitioning must comply with the consequences of the hardware architecture. In addition to the desiderata (see Section 2.4), enclaves do not support vectorization, which is common in partitioning algorithms [58]. We, thus, have to deeply consider the hardware characteristics in the design of our join algorithm.

CrkJoin implements an in-place partitioning using sequential scans only. This allows it to reduce random access and leads to small memory footprint; the algorithm avoids creating large ancillary structures. Yet, the partitioning algorithm is lazy. It exploits the results from previous iterations to crack the smallest relation slice needed for the current iteration. Therefore, CrkJoin avoids scanning the entire relation per partition (such as MCJoin [12]). As a result, CrkJoin achieves the first two desiderata points. For clarity, we now focus only on single-threaded execution. We release this assumption in Section 5, where we tackle the third desiderata point.

Algorithm 1 shows the high-level structure of CrkJoin. It first initializes the data structures to track the progress of partitioning (lines 2 and 3). It, then, iterates over each partition and, every time, it executes a single *cracking* stage on both relations (lines 5 and 6).

Algorithm 1 CrkJoin algorithm.

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

Running Example. Figure 5 shows a fragment of CrkJoin's execution. Throughout the paper, we use this figure as a running example to better visualize the cracking-like join idea. Figures 5a-b show the data and computation primitives. For the partitioning algorithm, we assign a playing card suit to each partition. We use hearts (♥), diamonds (♦), clubs (♣), and spades (♠) (see table in Figure 5).

Cracking Primitives. Efficiently performing lazy data partitioning, such as *cracking relation*, requires primitives that do not violate the desiderata: no random access, low memory consumption, and a wait-free design. We introduce two primitives: (i) Cracking Tree, a data structure to keep track of the progress of the algorithm, and (ii) *stage*, a basic unit of computation when cracking relations.

The purpose of the *Cracking Tree* (CT) is to keep track of the partitioning progress in a data structure so that it can build atop results from previous stages (Figure 5a). CT is a binary tree: The nodes store pointers to a slice's beginning and the number of tuples it contains. In its structure, CT intrinsically stores the bit mask of a slice. For example, the diamonds represent a mask *b01*. The node containing the diamond tuples, therefore, is located from the root down left (0-bit), and then right (1-bit). Correctly identifying the slice with diamond tuples significantly reduces the size of the scan.

Algorithm 2 Cracking Relation Stage

```
1: procedure CrackStage(CT *root, int partition):
2:     slice ← FindSmallestSlice(root, partition)
3:     p0 ← slice.first, p1 ← slice.last
4:     b ← DetermineBit(slice, partition)
5:     while p0 < p1 do
6:         while CheckBit(p0, b) = 0 do
7:             p0++;
8:         while CheckBit(p1, b) = 1 do
9:             p1--;
10:        if p0 < p1 then
11:            SwapTuples(p0, p1);
12:    slice.AddLeafNodes(slice.first, slice.last, p0);
```

In contrast to CT, a *stage* is a computation primitive. The goal of a stage is to partition a slice of a relation by one bit, which leads to adding two new nodes to a CT. Therefore, a stage outputs the same slice internally partitioned in two. A stage consists of multiple steps, which advance the pointers toward each other and perform a single tuple swap. For instance, slice *b0x* (Figure 5a) is partitioned into *b00* and *b01*. The challenge is to do so efficiently, i. e., without violating the desiderata. We achieve this by implementing stages with sequential scans and in-place swapping. Algorithm 2 shows the pseudo-code of a stage. CrkJoin traverses the CT to find the *slice* containing all relevant tuples (lines 2 and 3). Next, it identifies the bit b to partition on. Finally, it partitions the slice (lines 5-11).

Let us use the running example to visualize CrkJoin's partitioning (Figure 5b). Recall that we use card suits to represent partitions. CrkJoin starts with the slice between p_0 and p_1, pointers to the first and last tuples of the slice (step 1). It, then, scans the slice from both ends and swaps tuples that do not match the partition. Here, it swaps diamond tuples pointed by p_0 with hearts tuples pointed by p_1 (step 2). Once the pointers meet, the slice is internally split into two – hearts and diamonds (step n). CrkJoin stores the information about the new partitions as *b00* and *b01* in the CT (line 12).

Bricks as Building Blocks. We now explain how CrkJoin builds on top of these two primitives to lazily partition a relation. CrkJoin takes the number of partitioning bits, b, as input and devises the number of partitions and their bit masks. In the running example, the number of bits is two (we discuss setting this number in Section 4.3). Therefore, there are four partitions with bit masks from *b00* to *b11*. Although our algorithm might resemble radix partitioning, we do two fundamental things differently: (i) we process a relation iteratively using always one bit of the key, and (ii) we partition in-place and, therefore, do not create a copy of the data.

We start the partitioning by initializing the CT, which we use to record the progress. Then, we iterate over each of the four partitions. For instance, in the running example, we process partitions in the following order: hearts, diamonds, clubs, and spades. Yet, we do not partition the entire relation in each stage. We reuse results from

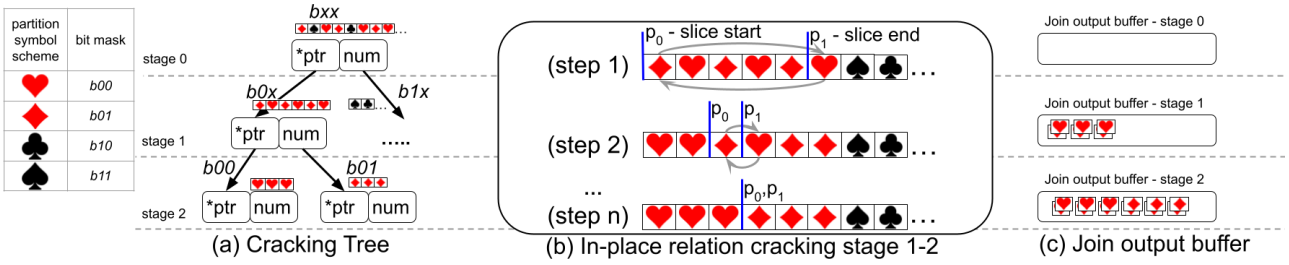

Figure 5: Cracking and joining a relation.

previous stages to find the smallest slice that contains relevant tuples. For example, in Figure 5b, while searching for heart tuples, we see that all of them are within p_0 and p_1 pointers. Hence, we avoid scanning the tuples in the remainder of the relation.

Let us explain the first two stages of the algorithm using again our running example, to better illustrate the relation cracking process. We start with the hearts partition, as its mask, *b00*, is the first in the order. Initially, we have no information about the order of the tuples (stage 0 in Figure 5a). Therefore, we partition the entire relation on the first Most Significant Bit (MSB) of the bit mask. The relation becomes internally partitioned in two – one partition with hearts and diamonds, and one with clubs and spades. We store the relevant pointers in the CT (stage 1). We proceed to the second stage. This time, we process the diamond partition (*b01* mask). Using the CT, we know that all diamond tuples are in the left leaf node (*b0x*). We partition this slice on the second MSB. As a result, we derive two slices – one with hearts and one with diamonds. Again, we store this information in the CT (stage 2). We repeat the process for the remaining partitions. We observe that stages modify the order of the tuples in the slices contained in the CT. Yet, a stage never moves a tuple out from a slice defined by previous stages.

Theorem 1. *A CrackStage (CS) preserves the integrity of a CT.*

Proof Sketch. A CS could break the integrity of a CT if it swaps tuples between already-defined slices or it inserts new faulty slices: **1)** Let a relation slice $A = [p_0, p_1]$ be the area scanned by the CS. By Algorithm 2, we know that p_0 only increases and p_1 only decreases (lines 7 and 9), and that the scan finishes when $p_0 = p_1$. Thus, SwapTuples is invoked only within A.
2) The CS inserts new slices when $p_0 = p_1$. As of specification (Algorithm 2), p_0 (p_1) increases (decreases) only if all tuples to its left (right) are in the correct partition. Therefore, we conclude that all tuples are in correct partitions when $p_0 = p_1$. □

For example, in Figure 5b, we swap diamonds with hearts within the red tuples slice. Yet, we do not swap a red with a black tuple.

Note that while partitioning, CrkJoin performs a join at each iteration. Joining shares the scans with cracking (Algorithm 1 lines 5-6). In the next section, we will see how CrkJoin achieves efficient scan sharing between the cracking and joining phases.

4.2 Joining the Cracking Slices

As in the *cracking* phase CrkJoin reorganized tuples and identified the ongoing smallest slices with relevant tuples, the joining phase is

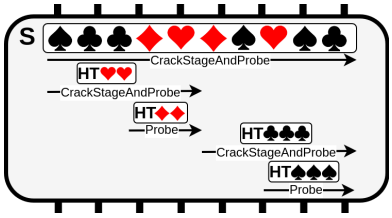

Figure 6: Probe table scans (arrows) and temporal memory consumption (white boxes) of CrkJoin.

simplified: CrkJoin has more organized data throughout the cracking stages and steadily reduces the scans while keeping the memory consumption low (Figure 6). Yet, the remaining challenge is joining two partitions with few cache misses and low memory footprint. Conceptually, we integrated the joining process into the cracking-like process in the *compute* phase. Yet, we take a step further and fuse cracking with joining. Both phases now share a single relation scan. The idea behind the fusion is simple: While examining the tuples for cracking, we also examine them for finding join matches.

Following our findings in [47], we decided to join with a hash join. A hash-based partitioning join performed best across an extensive number of experiments. We defined two functions for the fusion operations: CrackStageAndBuild (Algorithm 1 line 5) and CrackStageAndProbe (Algorithm 1 line 6). The former cracks the slice into two and builds a hash table with relevant tuples. The latter also cracks the slice but then probes the hash table for matches.

In the initial stages, slices are partitioned only on a subset of the bit mask. Therefore, they can contain tuples from other partitions. CrkJoin filters out irrelevant tuples and builds a hash table with the relevant ones. In later stages, a slice may be already fully partitioned. If so, CrkJoin performs a Build or a Probe scan without cracking. In detail, it builds a bucket chaining hash table [49]; This technique performs best in the lack of SIMD instructions [9]. Next, CrkJoin scans the slice of the probe relation and probes the table with the relevant tuples. Notice that, as one of the goals of CrkJoin is to achieve a low memory footprint, it ensures that the hash table fits into CPU caches by adapting the number of partitioning bits.

4.3 Selection of Partitioning Bits

All partitioning algorithms decide how granular they split the data. For instance, while GRACE join [39, 55] selects the number of buckets (i. e., partitions) such that each bucket fits in the main memory, radix join minimizes TLB misses. CrkJoin faces a similar

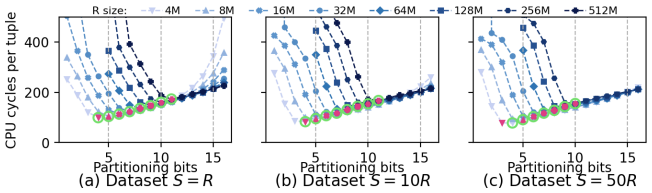

Figure 7: CrkJoin cost vs. partitioning bits. Red points indicate the optimal configuration, green circles indicate dynamically selected configuration.

trade-off; while fewer partitions introduce fewer scans and increase the size of the hash tables, more partitions do the opposite. We evaluated the performance impact of this trade-off on CrkJoin. To achieve that, we measured CPU cycles per input tuples for a range of partitioning bits. We also varied the dataset size (4-512M) and the input's cardinality ratio (1:1, 1:10, and 1:50). The results of the experiment (Figure 7) show that the right number of partitioning bits greatly influences CrkJoin's performance. On one hand, if the number of partitions is too low, we observe excessive random access to the hash tables. These accesses in turn generate cache misses. On the other hand, in the case of too many partitions, the algorithm creates many fine-grained partitions, which are joined one by one. We marked in red the optimal configuration of each run.

We observe two things from this experiment. First, although the cardinality ratio can influence the performance, it does not affect the optimal number of partitions; almost all datasets achieve their minimum for the same number of bits. Hence, the size of the probe relation does not impact the performance. Second, the average size of a partition in all optimal cases is close to 2 MB, i. e., the size of the L2 cache. Therefore, we propose a model that selects the smallest number of partitions such that they do not exceed the L2 cache:

$$b = ceil(\log_2(\frac{|R| \times t_{size}}{L2_{size}})) \quad (1)$$

, where b is the number of partitioning bits, t_{size} is the size of a tuple, and $L2_{size}$ is the size of the L2 cache. Previous approaches [21, 39, 55] have addressed a similar limitation w.r.t. the main memory. Yet, their split functions introduced skew in the bucket distribution and forced the community to search for more complex, dynamic solutions. CrkJoin avoids this problem thanks to radix partitioning, which mitigates most of the commonly occurring skews [15].

We tested the model by selecting the number of bits according to Formula (1). We include the results in Figure 7 as green circles. We see that this simple model is very accurate. It selected the optimal configuration for all but one run ($R_{size} = 8M$ in Figure 7c). However, the selected and the optimal configurations differ by less than 1%, which is negligible. We conclude that CrkJoin is capable of selecting the right number of bits and use this model in all experiments.

4.4 Analysis

The CLP finds the balance between memory access and consumption. Figure 3 shows two table scan approaches: a minimum number of scans (radix join) and a complete scan per block (MCJoin). In the first iteration, CrkJoin performs a complete scan (similar to MCJoin). However, cracking greatly reduces the scans in the next iterations. We now estimate the number of scanned tuples.

Table 2: Comparison of algorithms costs.

	CrkJoin	RHO	MCJoin
table scans	$O(b \times N)$	3	$O(\|R\| + \frac{\|R\|}{m} \times \|S\|)$
memory usage	$\frac{\|R\|}{2^b}$	N	m

Given a relation with N tuples, we crack the relation on b partitioning bits. We represent the cracking process as a binary tree (as in Figure 5a). Note that the depth of the tree is equal to b. The sum of tuples in each level is N. In the worst-case, all tuples belong to the last scanned partition (spades tuples in the running example). Therefore, we scan N tuples for all levels from 0 (i. e., root) until $b - 1$. Thus, the upper bound for the number of scanned tuples is $O(b \times N)$. We summarize the key cost parameters in Table 2.

Let us interpret what this complexity means for common query execution in TEEs and how it compares to related work. We first consider the probe relation. Typically, our algorithm achieves the highest throughput using ten partitioning bits (Section 4.3). In this configuration, CrkJoin carries out up to *ten* table scans. Radix join performs *three* complete scans of the probe relation: to build a histogram, to partition, and to join. On the contrary, MCJoin divides the build relation into blocks that fit the restricted memory m and scans the probe relation $\frac{|R|}{m}$ times (Figure 3b). While running on Intel SGX, the memory limit can be as low as 90MB (Section 2.1), which can fit up to 12M tuples (as in Table 3). Hence, MCJoin on SGX performs *thirteen* complete scans of the probe relation for dataset D. It is worth noting that smaller does not mean better; It is more important how each of these algorithms completes these scans with respect to access patterns and memory consumption.

5 PARALLEL CRACKING-LIKE JOIN

We now release our single-threaded assumption and focus on leveraging multiple cores to speed up CrkJoin. As seen in Section 2.2, benefiting from parallel execution is particularly challenging on SGX: Threads and mutexes are implemented with expensive OCALLs.

Figures 8a-b compare two popular threading models in join algorithms, such as in radix join [9] and PaMeCo [11]. Radix join partitions the data by assigning sub-relations to individual threads. It adds their output to a queue, which coordinates the threads in the join phase. PaMeCo's approach is simple – it parallelizes the build and probe phases within each iteration of the hash join. Its tasks are short. They synchronize after each of the two phases and their sub-phases (i. e., flip-flop, scatter, and join). The main weaknesses of these existing threading models are their strong dependency on synchronization. PaMeCo additionally suffers from excessive thread creation. This is not suitable for SGX due to its expensive OCALLs. Instead, we need a design with no synchronization and with more long-running threads for optimal multi-core performance on SGX.

We, thus, introduce a new threading model that intrinsically addresses the third desiderata, i. e., wait-free execution. Recall that CrkJoin is based on CTs and stages, which always output two slices (Section 4.1). We observe that these two output slices are *independent*, i.e., each contains tuples from a separate partition range. For instance, in stage 1 in Figure 5a, slices *b0x* and *b1x* are independent.

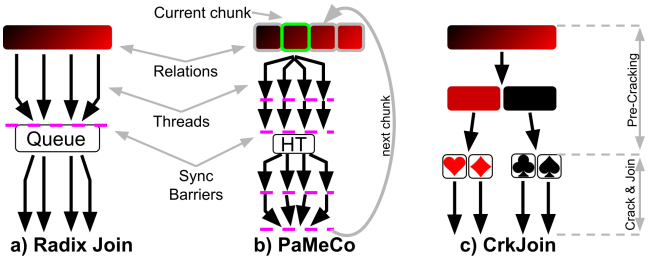

Figure 8: Threading models in join algorithms.

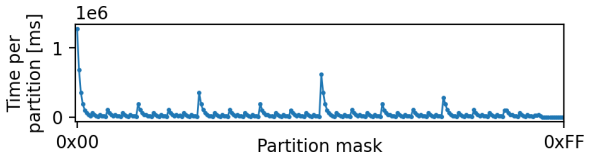

Figure 9: Time per partition.

While the first contains only red tuples, the second contains only black tuples. This presents a clear opportunity for parallelization: Independent slices can be processed by separate threads. We have designed the CT structure in such a way that keeps this independence behaviour. The branches representing independent slices in the CT structure are also independent, i. e., CrkJoin never accesses irrelevant parts of the tree. Clearly, we can independently join each of the slices when the relations are partially-partitioned. The main idea resembles the morsel parallelism proposed by Leis et al. [42] because of its elasticity in distribution that both models perform at runtime. Nonetheless, while the morsel approach splits the relations horizontally into chunks, our parallelism sprouts as an effect of data partitioning. Moreover, in contrast to [42], CrkJoin's threads share no data structures and are not synchronized.

Parallel CrkJoin leverages this threading model to fully comply with the third desiderata. The main idea behind Parallel CrkJoin is to split relations into independent slices and join each of them in a separate thread. Figure 8c illustrates Parallel CrkJoin. We introduce a pre-partitioning step (the *Pre-Cracking* phase in the figure) to reorganize relations such that they form N independent slices. To achieve this, we use our two cracking primitives: stage and CT. That is, we partition the relations with stages and store the results in CTs. We inherit the correctness of the pre-partitioning step by reusing already introduced building blocks. Once we partitioned the relations to N slices, we proceed to the *Crack & Join* phase. We delegate each independent slice to a separate thread and execute single-threaded instances of CrkJoin (ST-CrkJoin, for short) that require no synchronization. Let us illustrate this process using the motivating example. We start with one thread that *cracks* a relation into two. It recursively starts two threads that repeat this task - they crack their slices and create two child threads. This continues until reaching $N/2$ threads, which crack both relations into N slices. In Figure 5a, the first thread cracks slice *bxx*. It then creates two threads that further crack slices *b0x* and *b1x*. The process continues until reaching N slices. At that stage, we run N instances of ST-CrkJoin and complete the join operation. The threads in the *Pre-Cracking* and *Crack & Join* phases do not require synchronization and, thus, comply with the desiderata. By default, CrkJoin assigns equal number of partitions to each thread. Yet, uneven partition ranges can potentially counterbalance data skew. Thanks to its flexible design, CrkJoin can easily adapt to a different strategy: It extends the *Pre-Cracking* phase to achieve a higher partitioning granularity and, depending on the strategy, assigns different ranges to each thread. This mechanism can mitigate input data skew problems.

6 IMPLEMENTATION

We now provide implementation details on CT operations and three types of table scans in CrkJoin.

CT Operations. We recall that a CT is a binary tree, where each node represents a relation slice partially or fully partitioned by the partitioning bits. A node stores a pointer where the slice starts and the length of the slice. The structure of the tree stores the information on the already-partitioned bits. We perform three main operations on a CT (Algorithm 1). First, FindSmallestSlice searches for the smallest relation slice that contains relevant tuples. The operation is a tree traversal that compares bit-by-bit the partitions' bit mask with the existing nodes in the CT. The operation starts at the root and finishes if the current node has either no children or is fully partitioned. Second, we pick the next partitioning bit with DetermineBit. This operation compares the partitioned bits of a slice with the full partitioning mask and determines the next bit to crack the relation. Third, we add new nodes to CT with AddLeafNodes. It stores the results of a cracking stage (i. e., two new slices) in the CT as two new leaf nodes.

Cracking Table Scans. Parallel CrkJoin performs three types of table scans: basic cracking stage, cracking with building, and cracking with probing. We already explained the first scan type in Algorithm 1. CrkJoin uses CrackStage in the pre-partitioning phase. This operation is also the base for the two other scans. CrackStageAndBuild extends CrackStage by examining each scanned tuple as a candidate for joining. This is done after checking the value of the bit the slice is being cracked on (i. e., Algorithm 1 between lines 6 and 7, and 8 and 9). If the tuple is relevant, we add it to a hash table. CrackStageAndProbe examines each scanned tuple for its relevance and probes the hash table for join matches.

We measured the time to process each partition in ST-CrkJoin. The results of the experiment (Figure 9) show a general behavior: The more the data is organized, the smaller the processing time. The sudden spikes indicate that a new tree branch is being explored. In the running example, such a spike is visible when we finish processing branch *b0x* and start processing branch *b1x*.

7 EXPERIMENTAL EVALUATION

Recall the major goal of CrkJoin is high efficiency in untrusted, concurrent environments, such as the public cloud. We present an extensive experimental evaluation of our algorithm that validates this performance goal. We answer the following key questions: How fast is CrkJoin in a real-life environment (Section 7.2)? How well does CrkJoin perform for a single query (Section 7.3)? How well does CrkJoin scale with respect to the number of cores and data size (Section 7.4)? Is CrkJoin robust for query processing (Section 7.5)? How does CrkJoin behave with diverse datasets (Section 7.6)?

Table 3: Dataset used in the experiments.

	Synthetic		TPC-H (SF 100)	
Dataset	***A***	***B***	***C***	***D***
R cardinality	32M	32M	150M	15M
S cardinality	320M	32M	600M	150M
|R| : |S| ratio	1:10	1:1	1:4	1:10
total input size	2.6 GB	0.5 GB	5.6 GB	1.2 GB

7.1 Setup

In the following, we describe the hardware, baselines, datasets, and queries that we used for our experiments.

Platform. We ran all experiments on a 3.4GHz 16-core Intel Xeon E-2278G CPU with 256 kB L1, 2 MB L2, 16 MB L3, 256 MB EPC, and 125 GB RAM. The machine ran Ubuntu 20.02.2 OS and SGX v2.15.101. We wrote all joins in C++ and compiled them with gcc v9.4.0 with *-O3*. We used TeeBench [47] to manage the experiments.

Baselines. We compared the performance of CrkJoin to three memory-constrained join algorithms (*core baselines*), namely Block-nested Hash Join (BHJ), MCJoin [12], and PaMeCo [11]. BHJ is our best-effort implementation of the naive approach discussed in Section 2.3. MCJoin [12] is a single-threaded memory-constrained join algorithm based on BHJ principles. It reduces memory consumption with extensive usage of compression. PaMeCo [11] is a multi-threaded MCJoin. MCJoin and PaMeCo can be configured to consume less memory. We determined experimentally that MCJoin performs optimally with 275 MB and PaMeCo with 90 MB on SGX. We use these values in all experiments. Furthermore, we considered the radix join algorithm (RHO) [9] as a core baseline because our recent study [47] showed that RHO is the most versatile on SGX. In all experiments, algorithms use the optimal number of threads per dataset. We determine these values experimentally in Section 7.4.

For completeness, we also consider TinyDB [44] as a baseline when evaluating CrkJoin for a single query. The TinyDB algorithm is based on the traditional nested-loop join. We call it TinyDB because the TinyDB system [44] used it as a memory-efficient join algorithm. Other memory-constrained database systems [14, 36] also followed the same approach. We avoid reporting the results for TinyDB in other experiments as it significantly underperformed.

Datasets. We used two synthetic datasets (Table 3): *A* and *B*. The datasets represent a column-oriented storage model. They contain *<key, payload>* tuples with 4-bytes attributes. The tables follow a primary key-foreign key relationship, which is the prevalent join use case in DBMSs. The join keys are uniformly distributed unless stated otherwise. The sizes and other characteristics of the datasets are similar to related work [9, 13, 38, 47, 61] to allow comparability across works. Similarly, we extracted TPC-H data (SF 100) to two datasets: *C* and *D*. Dataset *C* joins tables *orders* with *lineitem* on the *o_orderkey* attribute. Dataset *D* joins tables *customer* with *orders* on the *c_custkey* attribute. Both joins are used in many TPC-H queries.

Queries. We focus on relational equi-joins. We join a relation R with a relation S on an equi-join predicate $R.key = S.key$. Overall, in our experiments, we measure the throughput of CrkJoin to evaluate its efficiency. We define throughput as *the sum of input cardinalities divided by the join execution time*. We take each measurement five times and report the median value. We mark queries

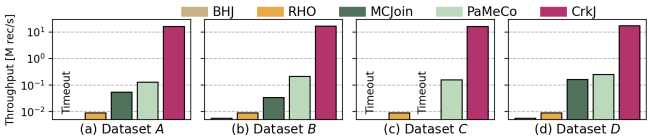

Figure 10: Eight concurrent queries per dataset.

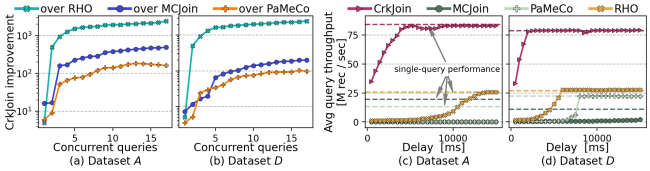

Figure 11: Multi-tenancy performance.

that completed but took more than several hours as *Timeout*. In our micro-benchmarks, we avoided materializing the output and only counted the number of join matches (similar to [9, 47, 61]). Last but not least, we ran a set of TPC-H queries to measure the end-to-end performance with different algorithms. We simplified the queries so that their performance depends primarily on the join operators.

7.2 Multi-Tenancy Environments

In real, untrusted environments (such as the public cloud), infrastructure (especially SaaS) is often shared by multiple users running many queries at a time. Although multi-tenancy in TEEs is not a security threat, users' queries share the resources. We validate the performance when multiple enclaves run joins concurrently on one machine. To validate CrkJoin, we start multiple join operators (Section 7.1) on the test datasets, each operating on separate tables and in a separate enclave to emulate users with no shared privileges.

As a first experiment, we ran eight concurrent join queries at the same time (similar to Figure 1). Figure 10 illustrates the average query throughput achieved by CrkJoin for all datasets. We see that CrkJoin outperforms the baselines consistently for all datasets: It achieves up to three orders of magnitude higher throughput. BHJ and RHO perform significantly worse due to high memory footprint and excessive thread synchronization. MCJoin and PaMeCo perform better thanks to the low-memory design. Yet, they still perform multiple full scans of the probe table, which leads them to two orders of magnitude worse performance than CrkJoin. Moreover, in three experiments, PaMeCo did not benefit much from multi-threading due to a thread management incompatibility with SGX.

As a second experiment, we manipulated the number of concurrent queries. The objective of this experiment was to confirm that the CrkJoin's improvement in Figure 10 was not coincidental. We vary the number of queries until the machine runs out of main memory. We ran this experiment with a set of eight queries for one synthetic dataset (*A*) and one TPC-H dataset (*D*). We present the results in Figures 11a-b. We observe that CrkJoin always outperforms the baselines independently of the number of queries. With more queries, CrkJoin gracefully degrades, while the baselines degrade drastically. CrkJoin outperforms MCJoin and PaMeCo by more than two orders of magnitude and RHO by more than three

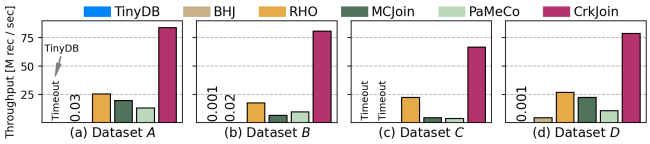

Figure 12: Single-query throughput comparison.

orders of magnitude. We attribute this to CrkJoin's very low memory consumption. The baselines operate on larger data structures and quickly exhaust the EPC memory; we noted EPC thrashing already for two concurrent RHO queries. We excluded BHJ from all experiments in Figure 11 due to timeout errors.

Lastly, we considered a scenario where queries arrive one after the other with a delay between them. Besides this being common in practice, our goal is to evaluate if a simple delay between queries reduces the concurrency and alleviates the memory bottleneck; A delay can lead to queries running individually on a machine. Note that single-query execution represents the optimal performance of each join as there is no contention coming from other operators. In this experiment, we ask how big this delay has to be so that each join converges to the optimal, single-query performance. Figures 11c-d illustrate the results with the dotted lines representing single-query performance. We observe that CrkJoin reacts well when queries arrive one after the other: It is up to three orders of magnitude faster than the baselines. Its throughput quickly converges to the single-query performance; it achieves 90% of the single-query performance for a 3.5-second delay for dataset *A*. On the other hand, the other baselines prove to be impractical. RHO needs at least a 13-second delay, and other baselines more than 15 seconds, to converge to the single-query performance for dataset *A*. We observe similar behavior for dataset *D*, which is unacceptable for the public cloud. Overall, we showed that memory consumption is important in a multi-tenant setup due to contention coming from parallel execution. Thanks to its design, CrkJoin multiplied the improvement factor from one order of magnitude for a single-query to up to three orders of magnitude for a multi-tenant environment. We expect the real-life public cloud setups to be multi-tenant. We recall that the new generation of SGX comes with a larger EPC and would delay EPC thrashing, i. e., entering the red zone defined in Figure 2. However, in the future, we expect hundreds of concurrent queries that will inevitably exceed the EPC.

CrkJoin improves multi-tenancy performance by up to three orders of magnitude compared to baselines.

7.3 Single-Query Performance

We now isolate CrkJoin to evaluate its performance in a confined environment. A single enclave has now all the resources of a single machine to run one join query. Such an isolated environment has been the measurement base case of many related works on join performance [9, 10, 12, 15, 23, 47, 61]. This allows us to evaluate the performance without the race for resources between operators. Concretely, we compared the throughput of CrkJoin to all core baselines and TinyDB using the four test datasets.

Figure 12 shows the results of this experiment. We observe that CrkJoin performs significantly better for all datasets. It outperforms

Table 4: Hardware performance counters.

	Dataset *A*			Dataset *B*		
	IR[M]	L3Miss [k]	L2Miss [k]	IR[M]	L3Miss [k]	L2Miss [k]
CrkJoin	135	190	1552	81	80	620
RHO	26670	4526	114856	4481	786	18790
MCJoin	6251	1692	20772	6205	1511	20399
PaMeCo	4501	2070	19638	4066	728	12789
BHJ	2301	3663	25994	719	532	4748

other algorithms by at least 2.9× (RHO) and up to four orders of magnitude (TinyDB). We attribute this to low memory consumption (in contrast to RHO) and efficient usage of the multi-core architecture. In fact, we see that PaMeCo benefits very little or even deteriorates when using multiple threads compared to its single-threaded version (MCJoin). PaMeCo introduces a threading model that does not scale on Intel SGX. In addition, we observe that both BHJ and TinyDB perform substantially worse than our algorithm. Both introduce an excessive amount of scans of the input tables which dwindle the performance. All algorithms behave similarly across datasets. Yet, we see that MCJoin and PaMeCo benefit when the cardinalities of the input tables differ more: They perform many smaller scans per block of the outer table when the ratio is smaller (e. g., for datasets *B* and *C*), which leads to lower throughput.

Next, we measured the hardware performance counters for both synthetic datasets. Table 4 shows the most relevant results. We see that the L2 and L3 cache misses of CrkJoin were 1-2 orders of magnitude lower compared to the baselines. This indicates that the simplicity of the cracking primitives enables high data locality and successful work of prefetchers. Effectively, it leads to a smaller number of retired instructions and higher throughput. On the contrary, RHO performs excessive random writes during the partitioning phase, which caused a high number of cache misses.

Based on these results and those from Section 7.2, we decided to discard baselines that underperform on SGX to improve the clarity of the remaining experiments. Thus, in the following experiments, we compare CrkJoin only with MCJoin, PaMeCo, and RHO.

In confined environments, CrkJoin outperforms core baselines and TinyDB with improvements from a factor of 2.9 up to three and four orders of magnitude, respectively.

7.4 Scalability

In the following, we evaluate CrkJoin's scalability with respect to the number of concurrent threads and the database size. For the former, we measured the throughput of all the join algorithms while increasing the number of threads. For the latter, we picked the optimal number of threads per join algorithm and scaled up the datasets. These experiments allow us to determine if CrkJoin performs well on multi-core hardware and with real-life size datasets.

Number of Threads. Figure 13 presents the results when increasing the number of threads used by the algorithms for all test datasets. First, we see that, in a single thread setup, CrkJoin outperforms the baselines by 2.8× on average. It achieves so via low memory consumption and by prioritizing sequential scans. Second, CrkJoin improves by up to 15× over the baselines in terms of throughput. It can only achieve so by simultaneously exploiting good memory utilization and an effective wait-free design. It is the only algorithm

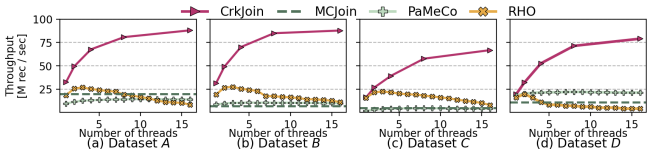

Figure 13: Scalability of joins with the number of threads.

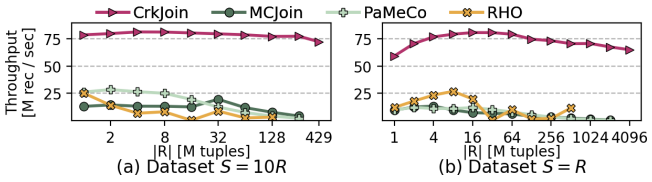

Figure 14: Scaling the input relations for $R \bowtie S$ (R = build).

to scale beyond three threads. We particularly observe that RHO scales only up to three threads and then steadily deteriorates (similar to [47]). This is because RHO's threads synchronize outside of the enclave using expensive OCALLS, which quickly become a bottleneck. For more than eight threads, the performance of RHO goes below the performance of single-threaded MCJoin. Similarly, we see that PaMeCo achieves negligible scalability for more than four threads. When investigating this further, we observed it performs excessive thread management outside the enclave. We selected the optimal number of threads for CrkJoin (16), RHO (3), and PaMeCo (4); and used them across all experiments.

Dataset Size. We now increased the cardinalities of both relations from the synthetic datasets up to the maximum 4-byte values (i. e., over four billion tuples). We observe from Figure 14 that CrkJoin outperforms the baselines by 2.8 – 252× and is the only algorithm able to process the largest datasets. RHO requires creating very large enclaves (i. e., at least the size of the input data) that are impractical and discouraged by Intel [34]. In particular, we observe that CrkJoin shows a stable performance across all inputs. We attribute this stability to fair work distribution between wait-free threads and the correct selection of the number of partitioning bits. CrkJoin cracks on more bits for larger datasets, which prevents it from exceeding the CPU caches for a single partition. We also see that, similarly to CrkJoin, MCJoin has a stable performance. This is because it adapts the number of scans based on the memory limitation and the sizes of the input tables. This is not the case for PaMeCo and RHO. The performance of PaMeCo deteriorates due to two memory events: when the input data exceeds the restricted memory (> 8M tuples) and the EPC (> 64M tuples). The performance of RHO fluctuates due to its static configuration. This confirms the findings from previous works on the volatility of the performance of the radix join [15, 61].

CrkJoin scales efficiently across all CPU cores and outperforms the baselines from 2.8× up to more than two orders of magnitude.

7.5 End-to-End Performance

We, now, turn our attention to the performance of CrkJoin in "the big picture", i. e., when the join operators form part of a query. For this, we selected queries from the TPC-H benchmark that contain primarily joins and no sub-queries; we chose queries 3, 10, 12, and 19 for the evaluation. As we focus on the performance of joins, we simplified the queries by removing operations other than join, selection, and projection. We then manually "compiled" the queries to their C++ representations following the column-store [16] and query compilation [56] principles. Our approach strongly resembles the approach taken in [61], which follows the execution strategy of HyPer [37]. We ran all queries in a confined environment, similar to Section 7.3. Note that all joins in this experiment are pipeline-breakers. We, thus, materialize the output after each operator.

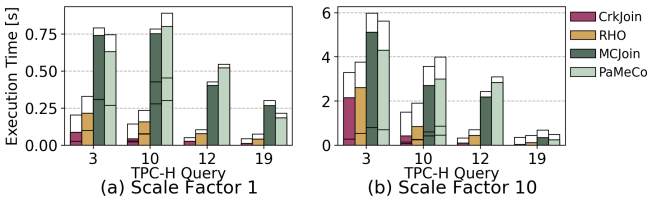

Figure 15: Simplified TPC-H queries. Color bars represent the time spent on join; white bars mark the time of the rest of the query (including the join output materialization).

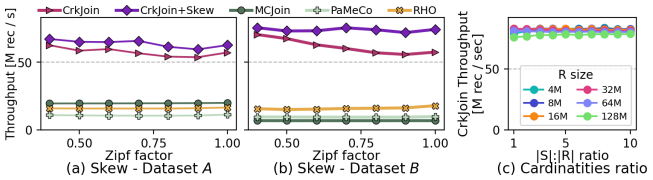

Figure 16: Data twists.

The results of this experiment (see Figure 15) show that queries with CrkJoin outperform all baselines for all cases. The queries are up to 11× faster than PaMeCo, 9× than with MCJoin and 2.2× faster than with RHO. We marked the execution time of individual joins in relation to the overall time. In general, the obtained results align with the results of the single-query results in Sections 7.3 and 7.4, confirming the correctness of the previous results. For a few cases (e. g., Q3), we observe that the difference between CrkJoin and the baselines is smaller than expected. This is caused by the additional memory consumption introduced by selections and projections. Its effect is particularly seen with CrkJoin due to the algorithm's tiny memory consumption; EPC now receives significantly more calls. Overall, the results confirm that CrkJoin maintains its superior performance also when it forms part of a complete query.

CrkJoin speeds up TPC-H queries by up to 11×.

7.6 Data Twists

We now evaluate whether CrkJoin is resilient to diverse data inputs. To achieve that, we followed a similar approach to other studies on join performance [9, 10, 12, 13, 47]: We introduced skew into the synthetic datasets and we varied the cardinalities ratio.

Data Skew. We generated datasets A and B with a skew to validate the performance with non-uniform data distributions. We considered the Zipfian distribution varying the Zipf factor from 0.4

to 1 and measured the throughput of the algorithms. Figure 16a-b shows that CrkJoin is resilient to data skew. CrkJoin outperforms the baselines by 2.2 – 10×. Yet, in contrast to all baseline algorithms, our algorithm slightly decreases its performance for higher skew due to uneven work distribution between threads. Some threads take more time because they process more tuples than others. RHO mitigates it with *task decomposition*, a technique that divides large partitions and distributes them to more threads. We, thus, evaluated a different strategy to assign uneven partition ranges to threads: We considered a simple strategy that balances out the number of the tuples in the probe relation assigned to each thread. While it is conceptually similar to task decomposition, CrkJoin further cracks the largest slices in the CT and distributes these slices among threads. The results (CrkJoin+Skew) show that this strategy improves the throughput by up to 30%, fully mitigating the impact of data skew for dataset *B*. Although the results are promising, designing a full version of this strategy is out of the scope of this paper.

Cardinalities Ratio. Lastly, we manipulated the ratio between the cardinalities of the input tables from 1 : 1 until 1 : 10. The goal was to evaluate the stability of CrkJoin. In this experiment, we excluded the baselines and, instead, introduced more workloads. Figure 16c illustrates the results for six different inputs. Overall, we see that the performance of CrkJoin is balanced independently of the cardinality ratio. This tells us that the build and probe stages achieve similar performance and are not prone to become performance bottlenecks for datasets with other characteristics.

CrkJoin performs in a stable way for data with diverse characteristics and outperforms others by up to 10× for skewed data.

8 RELATED WORK

The two main lines of related work are relational joins and TEE-based systems. Also note that there are other aspects related to our work (e. g., data partitioning or modern hardware), which we already cited throughout the paper whenever necessary.

Relational joins have been studied for decades from many different angles [8, 9, 13, 15, 21, 38, 39, 48, 49, 55]. DeWitt et al. [21] devised the textbook join algorithms. Nakayama el at. [55] and [39] improved the hybrid hash join to avoid disk paging for skewed data. Later, Boncz et al. [15] and [49] considered memory as the main bottleneck and proposed radix partitioning and radix join [48]. A decade later, the community revisited the performance of join algorithms. Kim et al. [38] improved the performance of sort-merge joins using SIMD instructions. Their work was further improved by Balkesen et al. [8]. Meantime, Blanas et al. [13] advocated for non-partitioning joins. Balkesen et al. [9] also proposed the currently fastest implementation of radix join, which we included as one of our baselines. The other works proved unstable and unreliable in secure enclaves [47]. Joins in resource-constrained environments have also been extensively studied [10–12, 14, 36, 44]. TinyDB [44], PicoDBMS [14], and DB2 Everyplace [36] used the nested-loop join as a no-memory solution. MCJoin [12] and PaMeCo [11] are block-nested loop algorithms designed for memory-constrained environments. We compared the performance of CrkJoin to all of these three algorithms. Barber et al. [10] proposed a compressed hash table that reduced the memory footprint of their hash join. Secure relational joins have mostly focused on hiding access patterns [1, 5, 40, 43]. Yet, these works have not considered the performance in TEEs. The performance of join algorithms has been evaluated in benchmarks on plain CPU [61], in streaming engines [70], and in TEEs [46, 47]. The findings from these works helped us to identify the bottlenecks of SGX and select the right baselines.

TEE-based DBMSs have also been extensively researched in recent years as data privacy becomes increasingly important [4, 23, 52, 60, 66, 68, 72]. Always Encrypted [4] enables some operations in secure enclaves with Azure SQL Database. Yet, the paper only mentions hash joins, which proved to be inefficient in enclaves [47]. Opaque [72] is a distributed analytics platform and ObliDB [23] is a secure database engine with obliviousness guarantees. Both propose joins that hide data access patterns but sacrifice the performance to achieve their goal. EnclaveDB [60] is a database engine that ensures confidentiality, integrity, and freshness of data and queries. Sun et al. [66] built the first enclave-native storage engine. Oblix [52] is an oblivious index based on SGX. Operon [68] is a database that preserves data ownership throughout the entire pipeline. However, none of these three works considered the join processing problem. In addition, some works have analyzed and tried to improve the performance of TEEs [6, 67]. Arnautov et al. [6] analyzed Intel SGX to design secure Linux containers on top of it. Taassori et al. [67] proposed a way to reduce the EPC paging overheads. All these works are orthogonal to CrkJoin. Lastly, works on encrypted data processing have strongly influenced how TEE-based systems are being developed [27, 29, 54, 59, 71]. Fully homomorphic encryption [27] enables any operation on encrypted data. However, these schemes are currently impractical in terms of performance [54]. CryptDB [59] is a database that uses many encryption schemes to protect the data. Yet, [29, 71] have pointed out that the existing encrypted databases might have undisclosed vulnerabilities.

9 CONCLUSIONS

We found that joins currently underperform on TEEs due to three bottlenecks: access patterns, limited memory, and threading costs. Related work has focused primarily on security, paying significantly less attention to efficiency. We have introduced the CLP, a set of processing principles that mitigate the bottlenecks of TEEs. We have then proposed CrkJoin, a join algorithm that instantiates this philosophy. We have shown that CrkJoin performed notably better in a multi-tenant cloud scenario, outperforming the baselines by up to three orders of magnitude. CrkJoin also performs at least 2.9× better than baselines in an isolated, single-query environment. Overall, our findings have enabled efficient join processing in TEEs, leading to a more practical and safer cloud. We are excited by the potential of CrkJoin, and, in future work, would like to introduce CrkJoin into a secure database engine and devise its cost models for the database optimizer. We would also like to study inequality joins as well as other relational operators (e. g., aggregation and group-by) as they could also benefit from the principles of the CLP.

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). Dedicated to the memory of Jorge-Arnulfo Quiané-Ruiz.

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] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 563–574.

[3] Amazon. 2022. *AWS Nitro Enclaves*. Retrieved December 21, 2022 from https://aws.amazon.com/ec2/nitro/nitro-enclaves/

[4] 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.

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

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. {SCONE}: Secure linux containers with intel {SGX}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 689–703.

[7] Sumeet Bajaj and Radu Sion. 2013. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3 (2013), 752–765.

[8] 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.

[9] 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.

[10] 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.

[11] Steven Begley, Zhen He, and Yi-Ping Phoebe Chen. 2016. PaMeCo join: A parallel main memory compact hash join. *Information Systems* 58 (2016), 105–125.

[12] Steven Keith Begley, Zhen He, and Yi-Ping Phoebe Chen. 2012. Mcjoin: A memory-constrained join for column-store main-memory databases. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 121–132.

[13] 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.

[14] Christophe Bobineau, Luc Bouganim, Philippe Pucheral, and Patrick Valduriez. 2000. PicoDBMS: Scaling down database techniques for the smartcard. In *VLDB*. 11–20.

[15] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. 1999. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, Vol. 99. 54–65.

[16] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*, Vol. 5. 225–237.

[17] 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*. 1087–1099.

[18] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[19] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[20] Deloitte. 2020. *Cloud banking: More than just a CIO conversation. What will financial services of the future look like with cloud?* Retrieved Jun 17, 2022 from https://www2.deloitte.com/za/en/pages/financial-services/articles/bank-2030-financial-services-cloud.html

[21] 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.

[22] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. 2022. Benchmarking the Second Generation of Intel SGX Hardware. In *Data Management on New Hardware*. 1–8.

[23] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: oblivious query processing for secure databases. *Proceedings of the VLDB Endowment* 13, 2 (2019), 169–183.

[24] European Banking Federation. 2020. *The use of Cloud Computing by Financial Institutions*. Technical Report. Brussels, BE.

[25] 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.

[26] 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.

[27] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Ph. D. Dissertation. Stanford, CA, USA.

[28] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[29] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Why your encrypted database is not secure. In *Proceedings of the 16th workshop on hot topics in operating systems*. 162–168.

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

[31] Manu Gulati, Michael J Smith, and Shu-Yi Yu. 2014. Security enclave processor for a system on a chip. US Patent 8,832,465.

[32] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*, Vol. 7. 68–78.

[33] Stratos Idreos, Martin L Kersten, and Stefan Manegold. 2009. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 297–308.

[34] Intel. 2022. *Intel Software Guard Extensions Developer Guide 2.18*. Technical Report.

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

[36] Jonas S Karlsson, Amrish Lal, Cliff Leung, and Thanh Pham. 2001. IBM DB2 everyplace: A small footprint relational database system. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 230–232.

[37] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[38] 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.

[39] Masaru Kitsuregawa, Masaya Nakayama, and Mikio Takagi. 1989. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method.. In *VLDB*. 257–266.

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

[41] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[42] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

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

[44] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.

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

[46] Kajetan Maliszewski, Tilman Dietzel, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. TeeBench: Seamless Benchmarking in Trusted Execution Environments. In *Proceedings of the 2023 International Conference on Management of Data*.

[47] 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* 15, 3 (2021), 659–672.

[48] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE transactions on knowledge and data engineering* 14, 4 (2002), 709–730.

[49] Stefan Manegold, Peter A Boncz, and Martin L Kersten. 2000. Optimizing database architecture for the new bottleneck: memory access. *The VLDB journal* 9, 3 (2000), 231–246.

[50] 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*. 1–9.

[51] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *Hasp@ isca* 10, 1 (2013).

[52] 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.

[53] myFICO. 2022. *What's in my FICO Scores?* Retrieved September 26, 2022 from https://www.myfico.com/credit-education/whats-in-your-credit-score

[54] 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.

[55] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. 1988. Hash-partitioned join method using dynamic destaging strategy. In *Proceedings of the 14th International Conference on Very Large Data Bases*. 468–478.

[56] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[57] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)* 51, 6 (2019), 1–36.

[58] Orestis Polychroniou and Kenneth A Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 755–766.

[59] 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.

[60] 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.

[61] 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.

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

[63] 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.

[64] 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.

[65] Snowflake. 2022. *Documentation*. Retrieved November 9, 2022 from https://docs.snowflake.com/en/sql-reference/parameters.html#max-concurrency-level

[66] 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.

[67] 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.

[68] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. 2022. Operon: an encrypted database for ownership-preserving data management. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3332–3345.

[69] 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.

[70] 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.

[71] Zheguang Zhao, Seny Kamara, Tarik Moataz, and Stan Zdonik. 2021. Encrypted Databases: From Theory to Systems. In *11th Conference on Innovative Data Systems Research (CIDR)*.

[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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.