Benchmarking Analytical Query Processing in Intel SGXv2

Adrian Lutsch
adrian.lutsch@cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Muhammad El-Hindi
muhammad.el-hindi@cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Matthias Heinrich
Technical University of Darmstadt
Darmstadt, Germany

Daniel Ritter
daniel.ritter@sap.de
SAP SE
Waldorf, Germany

Zsolt István
zsolt.istvan@tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Carsten Binnig
carsten.binnig@cs.tu-darmstadt.de
Technical University of Darmstadt & DFKI
Darmstadt, Germany

ABSTRACT

Trusted Execution Environments (TEEs), such as Intel's Software Guard Extensions (SGX), are increasingly being adopted to address trust and compliance issues in the public cloud. Intel SGX's second generation (SGXv2) addresses many limitations of its predecessor (SGXv1), offering the potential for secure and efficient analytical cloud DBMSs. We assess this potential and conduct the first in-depth evaluation study of analytical query processing algorithms inside SGXv2. Our study reveals that, unlike SGXv1, state-of-the-art algorithms like radix joins and SIMD-based scans are a good starting point for achieving high-performance query processing inside SGXv2. However, subtle hardware and software differences still influence code execution inside SGX enclaves and cause substantial overheads. We investigate these differences and propose new optimizations to bring the performance inside enclaves on par with native code execution outside enclaves.

1 INTRODUCTION

The need for secure cloud DBMSs. The last decade has seen a fundamental shift in where Database Management Systems (DBMSs) run: public clouds have become the primary location where data is stored and processed. While there are many benefits in running DBMSs in the cloud, such as scaling on demand, the cloud model puts a high stake on the cloud provider regarding the security of the data [33]. Today, customers have to fully trust the cloud providers to keep the data safe and avoid any attacks that can result in data breaches or data corruption. Sadly, there are well-publicized examples of cloud providers failing to provide these guarantees [6, 41].

TEEs to the rescue? Thus, all major cloud providers are moving to provide new offerings to prevent such problems. A prominent technology deployed widely in the cloud is so-called Trusted Execution Environments (TEEs). A TEE is a hardware-based solution that shields a process from a potential attacker and has been successfully used to build secure DBMSs in the cloud [1]. On a high level, TEEs provide two primary protection guarantees. First, they provide integrity, i.e., ensuring that software or hardware attacks cannot manipulate code and data without being detected. Second, they guarantee confidentiality, i.e., code and data are encrypted inside a TEE and can not be accessed outside an enclave.

© 2025 Copyright held by the owner/author(s). Published in Proceedings of the 28th International Conference on Extending Database Technology (EDBT), 25th March-28th March, 2025, ISBN 978-3-89318-098-1 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

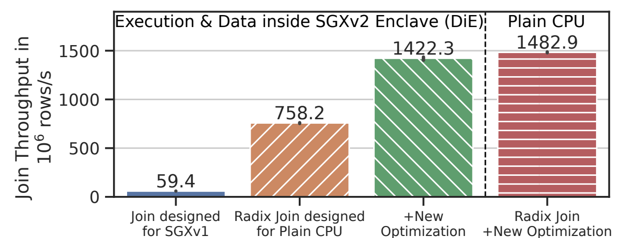

Figure 1: Performance of joining a 100 MB (hash) and a 400 MB (probe) table inside an SGXv2 enclave. The join designed for SGXv1 does not achieve competitive performance (blue). A state-of-the-art radix join is a better starting point (orange), and with our optimization (green), its performance is similar to outside the enclave (red).

Security does not come for free. One of the first broadly available TEE technologies was Intel's Software Guard Extensions (SGX). SGX extends Intel CPUs with instructions and hardware components that enable "secure enclaves", protecting processes against malicious administrators, operating systems, and hypervisors. However, being targeted for mobile and consumer devices, the first generation of SGX (SGXv1) had severe hardware limitations when used for DBMSs. In particular, memory access had a high overhead due to encryption and integrity checks, and the protected memory region that enclaves could access efficiently was only 256 MB, leading to high overheads when the data sizes exceeded that limit. As a result, DBMSs deployed on SGXv1 typically faced orders of magnitude slowdowns [27], making the first generation of SGX unpractical for data-intensive systems [10, 27, 35].

Recent advances of SGX lift limitations. With the Intel Ice Lake architecture, Intel SGX became available on multi-socket server hardware [18]. This second generation of Intel SGX (SGXv2) uses redesigned hardware to achieve isolation and confidentiality guarantees. Most importantly, the new generation relieves the memory limitation issue by allowing enclaves to access up to 512 GB encrypted memory per socket [10, 18]. Additionally, integrity checks have been streamlined, and enclave processes can leverage the newly added multi-socket support. After releasing the second generation of SGX, Intel discontinued the first generation.

The need for a performance study of SGXv2. While SGXv2 promises many benefits over SGXv1, the impact of integrating SGXv2 in the design of secure DBMSs is not yet well understood. For example, it has not yet been explored whether SGXv2 can

Series ISSN: 2367-2005
10.48786/edbt.2025.41

meet the demands of high-throughput analytical query execution operators. Hence, this paper provides the first in-depth study of running query execution operators in SGXv2. Our study is motivated by the observation that previous design principles to improve performance in SGXv1 by optimizing for the limited enclave memory as the main bottleneck are not adequate anymore. Instead, state-of-the-art data processing algorithms that target server-grade hardware and include optimizations like cache consciousness are a better starting point, as shown in Figure 1.

In this initial experiment, we compare a join designed for SGXv1 [26] to a cache-optimized radix join (both executed on SGXv2 hardware). The results in Figure 1 illustrate that the join designed for SGXv1 (blue bar) achieves a much lower performance compared to a state-of-the-art join implementation (orange bar). This performance difference arises because the SGXv1 optimized join prioritizes avoiding enclave paging – a concern irrelevant in SGXv2 – at the expense of not fully utilizing all available cores on the server-grade SGXv2 hardware. However, it also becomes clear that the radix join inside the enclave does not match the performance when executed outside of the enclave (red bar). As we uncover in this study, this performance gap results from different micro-architectural behaviors of running code inside and outside an SGXv2 enclave. To address these micro-architectural differences, we discuss new optimizations allowing DBMSs to achieve almost native performance as exemplified by the SGXv2-optimized join in Figure 1 (green bar).

Focus on analytical query processing. Rich related work has underscored that OLAP DBMSs can only achieve high performance and efficiency if the underlying CPU micro-architecture is taken into account [3, 19, 34]. Given the importance of micro-architectural effects in the context of OLAP and the under-explored performance characteristics of SGXv2, our study focuses on in-memory OLAP workloads. We implemented state-of-the-art (micro-architecture-aware) joins [3] and column scans [42], query execution operators that are at the core of all OLAP databases. This allows us to study their performance characteristics, uncover performance pitfalls, and provide suggestions for designing efficient algorithms for SGXv2.

Contribution and main findings. To summarize, in this paper, we present the results of the first in-depth performance study of key OLAP query execution operators in SGXv2 enclaves. Our study makes the following core contributions:

1. We show that state-of-the-art main memory and cache-optimized algorithms are a better starting point for SGXv2 than algorithms optimized for SGXv1: I.e., previously suggested SGX-optimized joins are no longer required, and throughput-optimized scan algorithms work at nearly equal performance out of the box.
2. We study the hardware and software overheads of state-of-the-art cache-optimized database algorithms in SGXv2 and uncover unknown overheads that are caused by a side channel mitigation enabled inside the enclave but not outside. Based on a radix join, we show how existing algorithms can be optimized to circumvent this previously unknown slowdown and match the throughput of join processing outside of SGXv2 enclaves.
3. Finally, we show that these results can be generalized to the execution of query plans, resulting in performance almost on par with native execution outside the enclave. I.e., the additional security of SGXv2 enclaves comes with only minor performance costs for query execution.

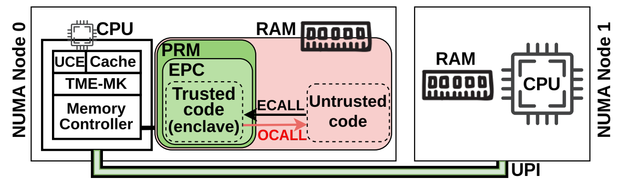

Figure 2: Intel SGX implements enclaves via a protected memory region in RAM, called Processor Reserved Memory (PRM). Data and code of enclaves are stored in encrypted memory pages inside the Enclave Page Cache (EPC). They are decrypted when loaded into the cache. The UPI Crypto Engine (UCE) encrypts enclave UPI traffic.

Being a performance study, this work is not concerned with the security properties of Intel SGXv2. Thus, we do not investigate specialized data structures and algorithms meant to prevent information extraction via side channels, such as algorithms with oblivious memory access patterns. Instead, we focus on the performance costs of the security technology and regard a detailed analysis of its security guarantees and weaknesses as future work.

Outline. The rest of this paper is structured as follows. First, Sections 2 and 3 give the necessary background about Intel SGXv2 and our benchmark setup. Afterward, in Sections 4 to 6, we evaluate the performance of key OLAP algorithms, join and scan, in-depth and then study their composition in query plans. Finally, Section 7 discusses a potential performance model, Section 8 presents related work, and 9 concludes this performance study.

2 INTEL SGXV2 BACKGROUND

The new server-grade generation of Intel SGX introduced with the Intel Ice Lake architecture [18] lifts several limitations of the first generation that led to high overheads in terms of performance. In this section, we review the basics of Intel's SGX technology and discuss the most important changes of SGXv2.

Integrity and confidentiality in SGX. Intel SGX protects the integrity of user code by shielding it even from privileged entities like the Operating System (OS) or the hypervisor. On a high level, this guarantee is achieved by creating a protected memory region in RAM, called Processor Reserved Memory (PRM), which can only be accessed via special CPU instructions [8, 31]. As shown in Figure 2, inside this protected memory region, SGX maintains the Enclave Page Cache (EPC) (light green area) to enforce enclave isolation. The EPC stores the trusted code and data of enclaves within encrypted 4 kB memory pages. These pages are only decrypted when loaded into the CPU cache for processing [8, 31]. Intel SGX guarantees that only trusted code from within the same enclave has access to the EPC pages of that enclave by adding security checks to the address translation. Importantly, code running in the untrusted memory region outside the PRM (including the OS) is prevented from reading and modifying these pages.

Major differences in SGXv2. While the capacity limitations of the PRM made Intel's SGXv1 impractical for data-intensive applications such as DBMSs [10, 27, 35], the new SGXv2 design supports up to 512 GB of PRM per socket, which allows DBMSs to hold large data sets fully in the EPC and avoids expensive enclave paging. This was achieved by replacing the previously used SGX Memory Encryption Engine with the new Total Memory Encryption – Multi-Key (TME-MK) [18]. In addition to changing the encryption hardware, SGXv2 replaced the integrity protection and freshness tree and the associated checks when loading

encrypted enclave data into the cache with specialized bits in ECC memory [18]. Finally, enclaves can now scale across multiple CPU sockets, increasing the number of CPU cores and the amount of memory available even further [18]. To access EPC pages on a remote socket, SGXv2 introduces an additional UPI Crypto Engine (UCE) that encrypts data before transferring it over Ultra Path Interconnect (UPI) [18] (cf. Figure 2).

Implications of SGXv2 for DBMSs. Although our previous work [10] indicates that with the second generation, Intel SGX has become a viable option for OLTP workloads, many important SGXv2 characteristics in the context of OLAP remain unexplored. For example, it is unclear if the new memory encryption hardware can keep up with the high throughput demands of optimized column scan algorithms. Furthermore, while we studied the latency of random cross-NUMA memory accesses in the context of OLTP [10], we did not analyze the effects on throughput and query execution operators like joins, which is essential for analytical query processing. Throughput-optimized OLAP algorithms using multiple threads have only been studied in the context of SGXv1 [26, 27]. However, with the hardware changes of SGXv2 mentioned above, it remains unclear if these findings of OLAP processing on SGXv1 generalize to SGXv2 and how the hardware characteristics of SGXv2 affect query execution performance. We address these questions in this paper.

3 BENCHMARK OVERVIEW

In the following, we give an overview of the benchmark settings, the framework, the used hardware, and the scope of the study. The input data for the algorithms is described at the beginning of the corresponding evaluations in Sections 4 to 6.

Benchmarking settings. The main idea of our evaluation study is to analyze the characteristics of SGXv2 by comparing the performance of join and scan algorithms, both when executed natively on the CPU without security extensions and in an enclave. Therefore, we compare two settings:

(1) *Plain CPU.* Traditional query processing baseline where the code is natively deployed on the CPU. This mode provides no security protections but also does not come with any additional overheads for computation and memory accesses. Data is always stored in untrusted memory in this setting.
(2) *SGX Data in Enclave (DiE).* Code and data are deployed within an enclave for processing. Since data resides in the EPC, it undergoes (transparent) decryption when loaded into CPU caches and encryption when writing data back to memory.

Additionally, we leverage the fact that Intel SGX enclaves can access the unprotected memory of their host process. This mechanism is necessary for communication with the enclave and can be used to trade security for performance, e.g., by storing non-sensitive data outside the enclave. It results in a third setting:

(3) *SGX Data outside Enclave (DoE).* Data is stored in untrusted (non-protected) memory, while code is processed within the enclave. This setting eliminates memory encryption/decryption overheads and allows us to distinguish between slowdowns caused by memory encryption and slowdowns caused by code execution within an enclave.

By comparing the behavior of joins and scans in these settings, we seek to identify computation and memory access patterns that exhibit different throughput or latency behaviors, enabling us to understand and optimize for the characteristics of SGXv2.

Table 1: Hardware used for our benchmarks.

Processor Name	Intel Xeon Gold 6326
Sockets	2
Cores per socket	16
Threads per socket	16 (HT disabled)
Base Frequency	2.9 GHz
L1d Cache (per core)	48 KB
L1i Cache (per core)	32 KB
L2 Cache (per core)	1.25 MB
L3 Cache (per socket)	24 MB
Microcode version	20240312
Memory Channels (per socket)	8
Memory	16 * 32 GB
Memory Speed and Latency	DDR4 3200 22-22-22
Memory Type	RDIMMs with ECC
EPC size (per socket)	64 GB

Benchmarking framework. We implement all our query processing operators either based on published best practices in the OLAP literature (e.g., [34, 42] for column scans) or based on existing benchmarks such as TEEBench [27]. Moreover, to reveal the root causes of performance bottlenecks, we use self-implemented micro-benchmarks. All benchmarking code is written in C/C++ and compiled with GCC version 12.3 using the optimization flags `-O3 -march=native` to ensure the highest optimization for our target architecture. To implement code running inside the SGXv2 enclave, we use the (default) SGX SDK provided by Intel in version 2.24. For measuring execution times, we rely on the RDTSCP instruction[1] since it is the only available method to measure execution times (as CPU cycles) with high precision in both CPU modes. If not otherwise stated, measurements are started after all required data for an operation has been allocated and initialized. This approach allows us to minimize the impact of, e.g., context switches and measure only the execution performance of the actual query processing algorithms. Similarly, our benchmarks only use data sizes that fit completely into the EPC to prevent the paging costs from dominating the measurements. We execute all experiments ten times and report the arithmetic mean and standard deviation.

Benchmarking hardware. For all experiments, we use a dual-socket server featuring 3rd Generation Intel Xeon Scalable, SGXv2-capable processors with 16 cores. The system is equipped with 512 GB (256 per socket) main memory distributed over 16 DIMMs that populate all memory channels of both sockets (see Table 1 for more detailed hardware characteristics). Our server runs Ubuntu 22.04.4 with kernel version 6.5 and uses the latest processor microcode (`20240312/0d0003d1`). Following security guidelines for SGX, we disabled Hyper Threading on the CPUs. To prevent noise caused by CPU frequency changes, we disabled Turbo Boost, changed the maximum CPU frequency to the base frequency (2.9 GHz), and enabled the performance governor to keep the CPU cores consistently on this fixed frequency. To prevent NUMA effects from influencing experiments, we pin execution threads to one NUMA node. On our trusted operating system, this is possible by pinning threads outside of the enclave with `numactl` or `pthreads` since the threads stay pinned to their core upon entering the enclave.

Study overview. Our study is split into three main parts. First, we analyze the performance effects of SGXv2 for joins. Secondly, we examine the throughput of multi-threaded column scans employing SIMD instructions. Finally, we study the performance of both operators in query plans.

[1]Stands for *Read Time-Stamp Counter and Processor ID [16]*

4 JOIN ALGORITHMS IN SGXV2

Joins are performance-critical operators in analytical databases because they involve processing large amounts of data. Hence, they have recently been studied also in the context of the first generation Intel SGX hardware [27].

Importance of the analysis. As shown in the introduction, simply adopting algorithms optimized for SGXv1 does not result in high performance. Further, despite offering enough enclave memory, the new SGXv2 hardware and security mechanisms still influence the performance of state-of-the-art join algorithms like the radix join. In this section, we analyze the root cause of these slowdowns in-depth by studying different classes of join algorithms with varying memory access patterns.

Join algorithms. For the investigation, we have built our own benchmark suite based on TEEBench [27], a collection of parallel join algorithm implementations for benchmarking SGXv1, and optimized the joins for SGXv2. To gain an overview of how the SGXv2 hardware affects the performance of standard join algorithms, we use the following implementations:

(1) *Hash join (PHT).* The *Parallel Hash Table Join* [5] uses multiple threads to create a shared hash table from the smaller join input table. Afterward, the threads iterate over partitions of the larger input table, probing the hash table. It uses a classical bucket chaining hash table and enables parallel writes to the hash table by latching the buckets.

(2) *Radix join (RHO).* The *Radix Hash Optimized* [28] join first partitions both input tables into cache-sized partitions by the least significant bits of their join key. To join the partitions, it employs an optimized hash table design, which achieved the best performance in previous evaluations [3, 27] (implementation from [3]). The implementation studied here uses a two-phase parallel hash partitioning method similar to the method described in [21].

(3) *Sort merge join (MWAY).* Sort merge joins first sort both input tables and then scan the sorted tables for matching rows in one pass. We added the implementation of the Multi-Way Sort Merge Join (MWAY) [21] from TEEBench to our benchmark suite.

(4) *Index nested loop join (INL).* The *Index Nested Loop Join* [27] (INL) in our evaluation uses an existing B-Tree index to find matching tuples for every tuple in the outer table.

In addition to these join algorithms, which are not optimized for SGX, we also investigate CrkJoin [26]. CrkJoin is a partitioned hash join designed with the main bottlenecks of SGXv1 in mind: EPC paging and random main memory accesses. It performs in-place radix partitioning without random memory accesses by iteratively sorting input tables into partitions. The sort happens one bit at a time. Two pointers are moved from the start and end of the table towards the middle until they meet. Tuples with keys in the wrong order are swapped. The sort starts single-threaded and the number of sorting threads is doubled after each bit until the number of hardware threads is reached. After partitioning, CrkJoin uses the same in-cache join method as RHO [26]. We optimized CrkJoin for our experiment hardware by configuring the available L2 cache size, as mentioned by the authors.

We do not study specialized join implementations that hide memory access patterns like the oblivious hash join of ObliDB [11] or the oblivious sort-merge join from Opaque [43]. We excluded them because they include algorithmic overheads that make them much slower than traditional join algorithms [27] and might hide performance overheads caused by architectural effects of SGXv2.

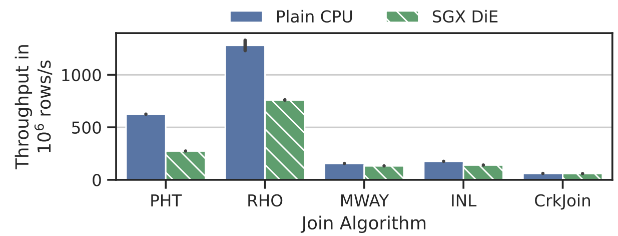

Figure 3: Overview of join algorithm throughput for 5 different joins executed using 16 threads to join a 100 MB and a 400 MB table on SGXv2 hardware. The SGXv1-optimized CrkJoin is the slowest join in this comparison. The hash joins have the highest slowdowns.

Join data. The input tables consist of rows with a 32-bit key (as join columns) and a 32-bit value (as tuple payload). All joins are foreign key joins, and keys follow a random uniform distribution. Similar to previous studies [5, 21, 26, 27, 36], we do not materialize join results in most of our join benchmarks to prevent potential side-effects caused by expensive memory allocations. We look at the effect of result materialization and memory allocation separately in Section 4.4 and in full queries in Section 6. The experiments in this section join a 100 MB and a 400 MB table if not mentioned otherwise. This equals the cache-exceed setting in the TEEBench paper [27] on join performance in SGXv1 and is similar to average join sizes in TPC-H at scale factor 100. Since our goal is not a comprehensive comparison of join algorithms among each other but understanding the performance effect of SGXv2 enclaves on join execution, we do not investigate varying payload sizes, data distributions, or other data characteristics. In particular, we do not investigate data skew since it causes non-SGX-specific effects, such as stragglers during parallel execution. Instead, we focus on those parameters that uncover interesting performance and hardware effects of SGXv2.

Initial results. Figure 3 gives an overview of the throughput of the join implementations in our benchmark. Throughput is expressed as the sum of input cardinalities (numbers of rows) divided by the join execution time. All 16 hardware threads on one socket are used for execution. We compare the performance of the same join implementation running inside an SGX enclave with all inputs, intermediate data structures, and outputs stored inside the enclave (SGX Data in Enclave/DiE) with a plain CPU baseline that runs the join without an enclave.

This experiment shows several interesting insights: Firstly, the performance of all join algorithms is lower when executed inside an SGXv2 enclave. Secondly, the reduction in throughput of all these joins when executed inside the enclave varies considerably between join types. The hash joins PHT and RHO have a much higher performance overhead than MWAY, INL, and CrkJoin. Finally, CrkJoin is the slowest join in our overview, reaching only 60 M rows/s. When executed inside the enclave, all other join algorithms perform better than CrkJoin, with speedups between 3× for INL and 12× for RHO. The measured performance of CrkJoin aligns with what the authors reported for SGXv1 hardware [26], and in our experiments, performance was similar independently of table sizes and skew. Thus, we derive that CrkJoin does not profit from the less restricting SGXv2 hardware. This can be attributed to its sort-based partitioning method, which starts with a single thread sorting on the first bit and then creates exponentially more threads for the following bits until all cores are

used. In contrast, the other join implementations always use all available cores in parallel. CrkJoin's partitioning method is only competitive with other partitioning methods if memory access is severely bottlenecked by the EPC paging, as is the case in SGXv1. However, on the new SGXv2 hardware, EPC paging is no longer a major performance limitation, and parallelism is important to achieve competitive throughput. Thus, CrkJoin is slower than the other joins in SGXv2.

Lessons learned. The main memory and cache-optimized join algorithms perform better inside SGXv2 than the SGXv1-optimized CrkJoin due to changed hardware characteristics, but they exhibit significant overheads.

Root causes of overheads. As we will show in the rest of this section, the slowdowns visible in the overview can be attributed to factors originating from the SGX security mechanisms on a hardware level. Additionally, there are other important performance factors that are rooted not purely in hardware but also in the software (e.g., the SGX SDK). We first summarize these factors below and present more detail in Sections 4.1 to 4.4:

(1) *Hardware-only effects.* Two hardware factors cause the slowdown of the hash joins in the overview. The first (cf. Section 4.1) is the more expensive random main memory access inside the enclave. Optimizing to mitigate this known effect is more important in SGXv2 since EPC paging is no longer the limiting factor. Additionally, we uncover a previously unknown overhead that does not result from SGX-specific memory encryption and security checks but from a side channel mitigation that is always enabled in SGX enclaves. This issue is investigated in Section 4.2, where we also demonstrate how manual loop unrolling and instruction reordering can alleviate it.

(2) *Mixed effects.* Other important performance effects result from an interplay of SGX software (i.e., the SGX SDK and the OS) and the SGX hardware. Firstly, while the support for Non-Uniform Memory Access (NUMA) in SGXv2 enables the usage of more cores in joins, Section 4.3 reveals that the unavailability of NUMA-awareness in SGX enclaves causes slowdowns because cross-NUMA traffic for joins can not be avoided. Secondly, Section 4.4 demonstrates how thread synchronization and memory management for SGXv2 can cause significant slowdowns if not handled carefully.

4.1 Overhead of Random Accesses

As mentioned before, random main memory access is a performance problem known from previous studies on SGXv1 [26, 27] and our own evaluation on OLTP workloads in SGXv2 [10]. In the following, we investigate the performance effects of slower random access on join algorithms. We use the Parallel Hash Table (PHT) join as an example because it is not optimized to reduce cache misses. In our investigation, we vary the size of the smaller input table (build side) from 1 MB to 100 MB and measure the join throughput in enclave relative to the throughput outside of the enclave. The probe table size is fixed at 1 GB and only a single join thread is used to prevent parallelization effects from influencing the measurements. To investigate the correlation between cache misses and performance, we measure the number of Last Level Cache (LLC) misses during the join and report this number divided by the sum of the input table cardinalities as *LLC misses per row*.

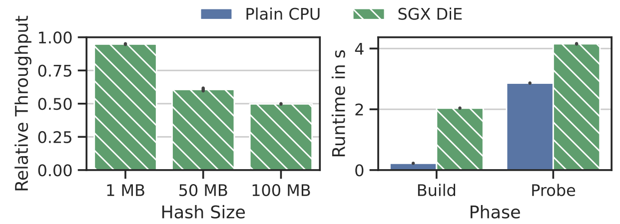

Figure 4: Left: Throughput of a single-threaded hash join with data and execution inside an SGXv2 enclave (DiE) relative to plain CPU. Join performance with large hash tables suffers from random access overhead. Right: Comparison of join phase runtimes at 100 MB hash size. The slowdown of the build phase inside the enclave is significant.

Figure 5: Performance of random memory reads and writes in an SGX enclave relative to plain CPU. In the cache, random access performance is equal. Random accesses to main memory are significantly slower in SGXv2.

The experiment results are depicted on the left side of Figure 4. The first bar shows that for a small table size of 1 MB, which fits into the cache of the tested CPU and causes fewer than 0.1 LLC misses per row, the join throughput inside the enclave is 95 % of the throughput outside the enclave. When increasing the size of the smaller table to 50 MB and 100 MB, which is 4 times larger than L3 cache, the number of cache misses increases to 1.75 and 1.9 LLC misses per row and the relative performance decreases to 62 % and 51 % respectively. Thus, the relative performance of the join correlates with the frequency of cache misses.

The next interesting question is which of the two join phases (building the hash table and probing it) affects performance most. Thus, we break down the hash join runtime into phases in Figure 4, right part. It reveals that the random-write-heavy build phase suffers a considerably higher performance reduction than the join phase. This raises the question if memory writes in SGXv2 have higher overheads than reads.

Random main memory access micro-benchmark. To answer if slower writes cause the higher overhead measured for the hash table build phase, the following micro-benchmark compares reading and writing 8-byte integers at random positions inside an array. The positions are determined by a linear congruential generator. We measure the throughput and vary the array size.

The results are depicted in Figure 5. We derive three main insights: (1) If the data is cache-resident, random memory reads and writes have no performance penalty inside SGX (as expected). (2) When increasing array sizes to larger than the cache sizes, the relative performance of random reads and writes decreases to similar degrees. Therefore, the bigger slowdown of the join's build phase cannot be explained by a performance difference between random reads and writes. (3) The performance decreases are significant. We see nearly 3x higher in-enclave latencies for the 8 GB array size and a doubling in latencies at 256 MB, which is the size of the hash table created in the join benchmark above.

We attribute the observed slowdown to two features of the SGXv2 hardware: (1) Since enclave memory is transparently encrypted, data must be decrypted when read from memory into the cache and encrypted when written to memory. According to measurements by Intel for TME-MK, this adds 11 ns of latency to last level cache misses [7]. (2) Most of the security guarantees of Intel SGX are enforced by adding checks to address translation [8]. This increases the cost of TLB misses. Thus, the constant TLB misses caused by random accesses over large memory areas are extra costly to resolve in SGX enclaves, leading to significantly higher random access latencies and reducing throughput.

Lessons learned. Random main memory access in SGXv2 enclaves causes high performance overheads that lead to a significant slowdown of algorithms and data structures dependent on them, such as the PHT and INL join, grouping, hash tables, and trees. Our micro-benchmarks show up to three times worse random main memory access performance in SGXv2. When data fits in the cache, there is no overhead caused by slower random memory access. Thus, there is a strong incentive to employ techniques that either keep data cache-resident for processing or hide the latency of cache misses.

One open question is why the hash table creation (build phase) is 9x slower inside the enclave, although the random memory access micro-benchmark only explains a 3x slowdown. As discussed in the next section, the underlying reason for this discrepancy and the slowdown of the RHO join is the same.

4.2 Overhead of Side Channel Mitigation

The RHO join does not suffer from random main memory access overheads because of its partitioning. However, the overview in Figure 3 still reveals performance reductions of more than 30 %.

Finding the root cause. To investigate the reason for this slowdown, we again break down the join into its main phases. The upper part of Figure 6 compares the runtimes of the stages in a single-threaded RHO join between the plain CPU (blue bar) and the enclave (green). It reveals that the overhead largely originates from creating histograms (*Hist. 1/2*) for radix partitioning and the partitioning itself (*Copy 1/2*). Especially the histogram creation is up to 2 times slower inside the SGX enclave.

This is curious because the base operations required to create a histogram for radix partitioning do not have high overheads. As shown in Section 5, linear main memory reads only have 3 % overhead in SGXv2 enclaves. Additionally, the previous micro-benchmark revealed that random cache accesses have no overhead inside enclaves (Section 4.1). Hence, it is unlikely that this overhead is caused by memory encryption.

Indeed, further investigation isolated the issue source to read-dependent write positions. That means all algorithms that alternately read values, determine a write position from the read, and then write to the determined position are affected. Their performance in SGX enclaves is reduced even if they are not bottlenecked by memory accesses. The histogram is such an algorithm because the histogram bin that must be written (incremented) depends on the input key. Other affected algorithms in our experiments are the copy phase of radix partitioning, the hash table build phase of the RHO join, and the hash table build phase of the PHT join discussed in the previous section.

Explaining the slowdown. After identifying the slowdown and consultations with Intel, we find that the microcode mitigation against Spectre Version 4, also known as Speculative Store

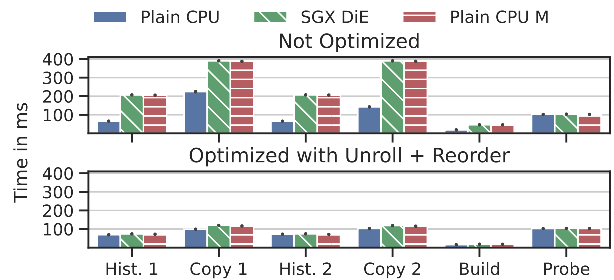

Figure 6: Runtime breakdown for the phases in a single-threaded RHO join with table sizes 100 (build) and 400 MB (probe). Comparison between Plain CPU, SGX Data in Enclave and Plain CPU with SSB M(itigation) enabled. Applying our unrolling and reordering optimization improves the performance of the slower phases significantly.

Bypass (SSB) [14], matches the observed behavior and is a possible explanation. The SSB vulnerability arises from the fact that the CPU speculates if a load position overlaps with the currently unknown position of an unfinished preceding store. If the speculation correctly assumes that the store does not invalidate the load, the load can be done in parallel to the store. However, wrong speculations cause reads of stale data. Although the CPU will detect this and discard values calculated from the stale read, this behavior can be exploited to create a side channel, potentially exposing the secrets of an application [14]. As one option to mitigate this side channel, Intel introduced a microcode patch that exposes a switch to disable the speculative behavior. With this switch activated, loads will not start before the addresses of all preceding stores are known. On a plain CPU, the mitigation is disabled by default. In SGX enclaves, however, the mitigation is permanently enabled and cannot be disabled [14].

To verify if this mitigation causes the performance difference between plain CPU and enclave computation, we enabled the mitigation outside of the enclave with the `prctl` [38] function. As the setting Plain CPU M(itigation) in Figure 6 shows, enabling the mitigation outside the enclave increases the runtime to exactly the same time as inside the enclave. Thus, the performance difference can be explained fully with the mitigation. We verified that this issue still exists on Emerald Rapids processors. This raises the question if the performance overhead of this side channel mitigation can be counteracted with specific optimizations.

Addressing the slowdown. To the best of our knowledge, we are the first to identify these significant performance issues caused by this side channel mitigation and propose a solution. As explained in the next section, since the side channel mitigation essentially deactivates the speculative execution in enclave mode, we suggest compensating this effect with loop unrolling and instruction reordering, as exemplified in Listing 1 for the histogram computation. Our experimental results in Figure 6 (lower part) confirm that our suggested optimization effectively counteracts the slowdowns caused by the SSB mitigation. Histogram and hash table build performance with the optimization applied are nearly equal in the enclave. For the copy step, the results are more nuanced. First, the optimization improves the performance of the copy step in all three settings. The performance of the plain CPU baseline also increases because the unrolling reduces the frequency of expensive miss-speculations. Second, since the copy operation inherently has dependent loads and stores that cannot easily be split up, the optimization cannot remove all performance differences for this algorithm. Third, there

```
// Original histogram loop
for (uint32_t i = 0; i < data_size; ++i) {
    size_t idx = (data[i].key & mask) >> shift;
    ++hist[idx];
}

// Histogram loop with unroll + reorder optimization
uint32_t i = 0;
for (; i + 8 <= data_size; i += 8) {
    size_t idx0 = (data[i].key & mask) >> shift;
    size_t idx1 = (data[i+1].key & mask) >> shift;
    ...
    size_t idx7 = (data[i+7].key & mask) >> shift;
    ++hist[idx0];
    ++hist[idx1];
    ...
    ++hist[idx7];
}
```

Listing 1: First loop: Histogram creation code used in radix partitioning. The table data is scanned, a simple hash function is applied to the join keys, and corresponding histogram bins are incremented. Second loop: Histogram creation for radix partitioning unrolled 8 times (shortened).

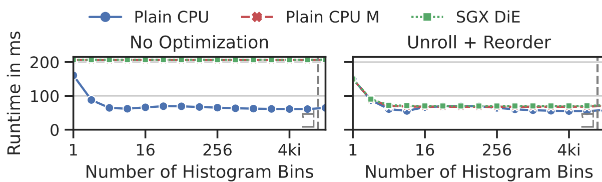

Figure 7: Histogram micro-benchmark for typical numbers of histogram bins. Using the scalar code, histogram creation is 225 % slower when executed inside the enclave. Manual loop unrolling and instruction reordering decreases the slowdown to less than 20 %.

is a remaining performance difference between the enclave and mitigation settings. This can be explained by the random access pattern of copying tuples to their partitions. All in all, the unroll and reorder optimization decreases the runtime of the single-threaded radix join by 62 % and increases the relative throughput from 46 % to 91 % of the baseline without mitigation and to 97 % of the baseline with mitigation enabled.

Optimization in depth. Since the slowdown is triggered by stores to data-dependent positions followed by loads, it can be reduced by increasing the time between data-dependent stores and following loads. One strategy to achieve this time separation is grouping loads and stores, i.e., first loading multiple values and then dispatching multiple stores in sequence. Thereby, most store positions are determined in parallel to other stores and do not block loads. This increases the number of concurrent memory operations and decreases average latency. This effect can be achieved by unrolling the inner loop of the algorithm and reordering the instructions. An example of the histogram creation loop is shown in Listing 1. In every iteration, the algorithm first reads multiple keys from the input and calculates indexes, and then issues multiple increments to the determined indexes in sequence. This optimization decreases the runtime of histogram creation in SGX enclaves to within 20 % of the same code running in normal CPU mode (cf. micro-benchmark in Figure 7).

Tuning of the optimization. Theoretically, the further the code is unrolled, the better the performance should become, as the average latency per load/store caused by the mitigation decreases. In practice, this improvement is limited by the number of registers available to store write positions. The number of registers available for this purpose depends on the algorithm and the

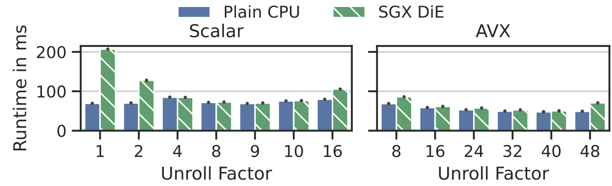

Figure 8: Runtime of the histogram algorithm in an enclave for a 500 MB input array and 32 bins for varying unrolling depth. Left: scalar code. Right: Vectorized code using AVX.

number of required registers for its calculation. As depicted on the left side of Figure 8, on the Ice Lake CPU architecture, runtime for the histogram algorithm decreases until 9x unrolling, where 9 indexes are determined and incremented per iteration. Starting from 10x unrolling, at least one index must be stored on the stack and loaded again. This interleaves loads and stores, thereby introducing additional latency and decreasing performance. Similarly to the histogram algorithm, the optimal unrolling factor can be determined empirically for other algorithms.

A prominent option to increase the number of indexes that can be stored in registers further is the usage of AVX for index calculation. Thus, we implemented index calculation for histograms with AVX intrinsics and unrolled the loop to store more indexes in registers. As depicted on the right side of Figure 8, this can improve performance further. Unrolling the loop 5 times and calculating 40 indexes (since each load consumes 8 indexes) before the increments start achieved the best performance in our experiments and a performance difference of 5 % between the enclave and plain CPU execution. However, when unrolling 6 times or more, the compiler generates instructions that store intermediate registers on the stack, decreasing performance.

Putting it all together. Finally, we investigate the effect of manual loop unrolling and instruction reordering on RHO and PHT using multi-threaded execution with all 16 cores on one socket. Again, we compare the join throughput inside the enclave to the same join code executed without SGX protection. We use the optimal unrolling factors for all algorithms marked with O(ptimized). Additionally, we show the effect of changing input table sizes with three example sizes. The results are depicted in Figure 9. With the optimization applied, the RHO join performance inside the enclave improves by 114 % to 33 % (SGX DiE compared with SGX DiE O). Thereby, it achieves 75 % to 95 % throughput of the fastest RHO plain CPU baseline. RHO performs best in the 100 MB/400 MB setting because of the relatively equal input table sizes that still fit the L1 TLB during partitioning. The PHT join throughput improves by 17 % to 118 %. For the small build size, it achieves 92 % performance of the baseline. For the larger build sizes, it reaches only 67 %/33 % performance of Plain CPU O since it is still limited by slower random main memory access. Thus, while the optimal join algorithm depends on the data characteristics, with our optimizations, the performance degradation inside the enclave is less than 10 %, as visualized by the dashed horizontal lines in Figure 9.

Lessons learned. The side channel mitigation for Spectre V4, which is disabled by default outside of enclaves but forcibly enabled inside enclaves, increases the runtime of the investigated algorithms by up to 225 %. All algorithms that determine write positions from input values, such as histogram creation, hash table creation, and partitioning, are affected. However, we show that loop unrolling and instruction reordering, as well as vectorization, can improve the performance of affected algorithms.

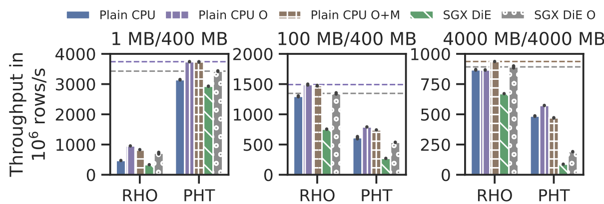

Figure 9: Comparison of RHO and PHT throughput joining three different table sizes with 16 threads with and without optimization (O) and SSB mitigation (M). Both joins profit from the optimization. With the optimization, the optimal algorithm for a specific table size reaches more than 90 % of the fastest baseline throughput (horizontal lines).

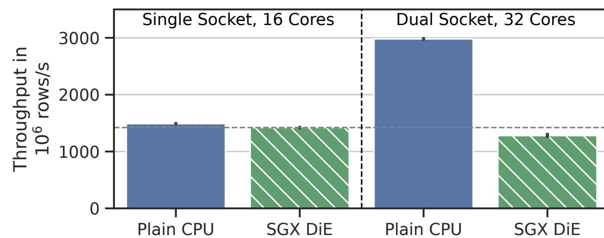

Figure 10: Throughput of an RHO join on a NUMA system in worst and best cases. A partitioned join with double the cores can achieve double throughput on a plain CPU. Inside the SGXv2 enclave, adding a second socket with 16 cores reduced the join performance because of the remote data access.

While previous work noticed the potentially negative effects of microcode updates on performance [39], we are the first to pinpoint a concrete side channel mitigation as the root cause, show the effect on databases, and suggest solutions.

More generally, the results show that, especially with other overheads like EPC paging removed, side channel mitigations can play a large role in software performance for SGX enclaves. Thus, developers should be aware of them and consider optimizations to circumvent performance regressions. Since the slowdown caused by this mitigation can be large, we suggest mentioning it and other mitigations applied in enclaves in the Intel SGX Developer Guide, which already contains other advice for performance in SGX enclaves [17].

4.3 Analyzing NUMA Effects for Joins

As introduced in Section 2, a new feature of SGXv2 is the support for servers with multiple sockets, and enclaves leveraging the secure memory on multiple NUMA nodes. Communication between NUMA nodes is known as an important performance factor for in-memory database operations and, in particular, joins [12, 20]. Moreover, while several NUMA optimizations exist to increase NUMA locality, cross-NUMA traffic cannot be prevented, particularly for complex queries (e.g., those including multiple joins). Hence, in this section, we aim to analyze the effects of cross-NUMA traffic. Since enclave communication via the UPI is encrypted [18] and previous work measured an increase in latency when accessing memory across NUMA boundaries in SGX compared to accessing cross-NUMA without SGX [10], we investigate how these costs influence the performance of join algorithms. An investigation of encrypted UPI throughput is contained in Section 5.4.

The main issue of NUMA in the current SGXv2 is the fact that the main tools for NUMA optimization – memory allocation and thread pinning in specific NUMA regions – are features of the untrusted OS and not available in enclaves. Thus, it is impossible to ensure local processing in the general case. For the following experiment, we make use of the fact that the Linux kernel on our trusted benchmark machine allocates EPC pages of an enclave in the local NUMA region if possible.

Benchmarking extreme NUMA cases. Since optimizations for NUMA in joins is a wide research field on its own, we concentrate on extreme cases in our experiments and expect the performance of real-world cases to fall in between. Our optimal baselines are a NUMA-local join with 16 threads (Single Socket Plain CPU) and a join where both input tables are pre-partitioned on the join key to both NUMA nodes (Dual Socket Plain CPU). The second setup avoids cross-NUMA traffic completely and reaches double throughput of only using one NUMA region. Additionally, we analyze two cases in SGX. The first is a NUMA-local join, where the enclave and all its memory are located on NUMA node 0 and the join is executed by all threads on the same node (Single Socket SGX DiE). In the second setting, all 32 cores of both sockets in the system execute the join, but the enclave and all its memory are allocated exclusively on one of the nodes (Dual Socket SGX DiE).

The results in Figure 10 show that without manual intervention, the use of cores from multiple CPU sockets can decrease the performance of a join executed inside SGX enclaves instead of increasing it as intended. By comparing SGX Data in Enclave (DiE) on a single socket with 16 cores to SGX DiE running on both sockets with 32 cores, it is clear that adding another 16 threads to the join while data is not distributed over both nodes decreases the join throughput inside the enclave instead of increasing it. This wastes the CPU cycles of 16 cores. Thus, the SGX join with 32 cores achieves less than half of the optimal case performance for a join that leverages all cores (Dual Socket Plain CPU).

Lessons learned. Cross-NUMA memory access significantly reduces the performance of joins in SGXv2 enclaves. To improve this situation, NUMA-aware memory allocations and thread placement are required. However, since the OS manages these hardware features, such manual control could currently only be implemented when trusting the OS to do thread pinning and memory allocations on specific CPUs correctly. As such, depending on the setting, NUMA-awareness can not be achieved in SGXv2.

4.4 Synchronization & Memory Allocation

To conclude the investigation of join performance, we discuss the last remaining factors in our *Mixed effects* category of SGXv2 overheads: Slowdowns caused by SGX SDK mutexes and dynamic enclave memory allocation.

Effects of mutexes. Many multi-threaded join implementations require synchronization of threads during execution. The authors of TEEBench [27] showed that the SGX SDK mutex limited the join performance in SGXv1 because it causes costly context switches outside the enclave. We revisit this issue in the context of SGXv2 since efficient synchronization becomes more important due to the new hardware: The increased number of hardware threads that can create more contention and other bottlenecks like EPC size have been removed.

To investigate the overhead in SGXv2, we designed the following experiment: We switched out the lock-free task queue of our radix join, distributing partition and join tasks between cores,

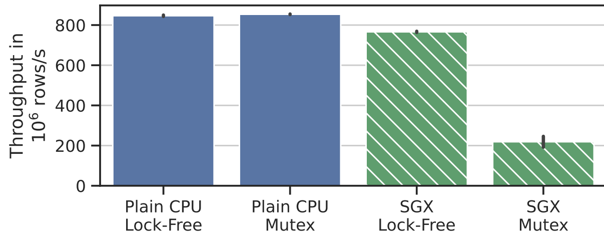

Figure 11: Throughput of an RHO join with contention on the task queue. Outside of the SGX enclave, the queue choice does not make a significant difference. Inside the enclave, protecting the queue with a mutex instead of a lock-free design reduces the throughput by 75 %.

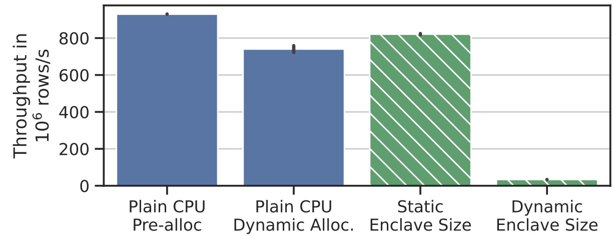

Figure 12: Throughput of the RHO join materializing output tuples inside a statically sized pre-allocated enclave compared with the throughput of the same join in a dynamically sized enclave. Dynamically increasing the enclave size during the join reduces its performance by 95 %.

with the mutex-guarded queue used in the original TEEBench. Since the issue only occurs in case of contention, we forced contention on the mutex by using small join partitions and 16 threads. In Figure 11, we compare the performance of both implementations inside an SGX enclave and on a plain CPU.

The experiment results exemplify that replacing high-overhead SDK functions with more optimized solutions can dramatically change the performance characteristics of an algorithm in SGXv2. Outside of the enclave, the choice of queue implementation does not cause significant throughput differences (blue bars). However, inside the SGX enclave, join throughput drops by 75 % when comparing the lock-free queue that avoids OS interactions with the mutex-guarded queue (green bars).

Root cause of mutex slowdowns. The observed performance difference is caused by the SGX SDK mutex' design. In this design, threads transition out of the enclave to sleep when they encounter a locked mutex. When the owning thread unlocks the mutex, it exits the enclave to wake the first waiting thread up. Then, both re-enter the enclave. During these enclave transitions, the mutex stays locked, extending the critical section and increasing the probability of threads arriving at a locked mutex. This is sensible if the critical section protected by the mutex is significantly longer than an enclave transition. However, critical sections of in-memory join algorithms are orders of magnitude shorter than enclave transitions. Thus, a mutex-based design has a negative performance impact. For the joins in this paper, we considered this effect and replaced mutexes found in their implementations with spin locks or lock-free data structures.

Effects of memory allocation. Memory management is another critical performance factor for DBMSs [9]. Therefore, many real-world databases use buffer managers that pre-allocate memory before it is needed. In cloud settings, however, it is desirable not to pre-allocate all memory available to a server at start time [2]. Additionally, before a query is started, it is not always clear how much memory the execution and result materialization will require. Therefore, DBMSs can be forced to allocate additional memory dynamically during query execution.

Originally, SGX enclaves had a fixed size that could not be changed after enclave creation. With the SGX 2 instruction set, Intel introduced additional CPU instructions that enable securely adding and removing enclave pages at runtime [30]. This feature is called Enclave Dynamic Memory Management (EDMM) and available since Linux version 6.0/SGX SDK version 2.18 [15]. Depending on the enclave settings, EDMM is either transparent to the developer or must be managed explicitly. The following experiment investigates the performance implications of automatic EDMM. In this experiment, we run our SGXv2-optimized RHO join and additionally materialize the result table. By reducing the amount of pre-allocated memory during enclave start to a minimum, we force a situation where all memory required to write the join result tuples must be allocated by dynamically increasing the enclave size. We compare this to a setting where the enclave is large enough to fit all result tuples without adding memory (static enclave size). As additional baselines, we also execute the join in native mode with memory pre-allocated (pre-alloc) and with dynamic memory allocation (dynamic alloc.).

The results in Figure 12 show that dynamically increasing the enclave size for memory allocations is an order of magnitude more expensive than dynamic memory management outside enclaves. In the experiment, the join inside the enclave achieves only 4.6 % throughput compared to the plain CPU join that allocates dynamically and only 4.1 % throughput compared to the static enclave size setup. The difference can be attributed to the security protocols necessary for resizing enclaves [30].

Lessons learned. The SGX SDK mutex and EDMM can introduce large overheads for query execution in enclaves. Thus, lock-free data structures should be preferred in enclaves, and EDMM should either not be used or actively managed to prevent its overheads.

5 SCANS IN SGXV2

In addition to joins, table scans are essential for the performance of OLAP systems since they require scanning large amounts of data with very high throughput. In this section, we use a columnar SIMD scan as a typical scan algorithm in OLAP databases which causes high demands on the memory subsystem.

Importance of the analysis. Interestingly, previous work incorrectly reported the lack of SIMD instructions inside SGX enclaves [26, 27] and hence state-of-the-art vectorized scan algorithms [34, 42] have not been studied yet. Given the high core counts available in recent server processors and the new memory encryption technology used in SGXv2 [18], it is unclear if the memory decryption engine is fast enough to allow for high throughput scans with multiple cores. Similarly, the impact of the additional encryption on the scan throughput when crossing NUMA boundaries has not been explored yet. Thus, studying throughput-optimized column scans is essential for understanding the performance characteristics of SGXv2 for DBMSs.

Scan algorithm and data. For our benchmarks, we implemented state-of-the-art scan algorithms [34, 42] using AVX 512 instructions. Our implementations load 64 byte-sized values at once from a column, compare them to a lower and upper bound (i.e., incorporating a filter condition), and store the comparison result either in a bit vector or, as we show in a later experiment, materialize row identifiers. We use the byte-aligned input value

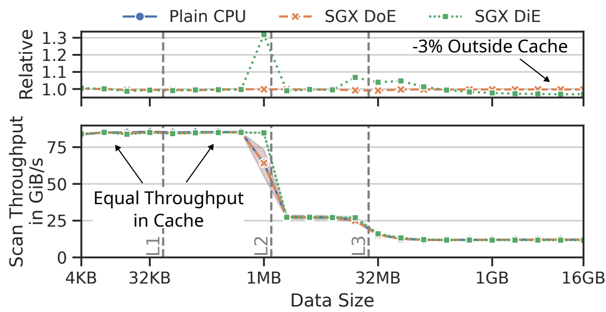

Figure 13: Read throughput of a scan using AVX 512 instructions, scanning over the same data 1,000 times. Comparison between enclave code reading enclave data (DiE), enclave code reading unencrypted data, and non-enclave code reading unencrypted data (plain). Inside the cache, scan throughput is equal, outside the cache we observe a slowdown of 3 %.

size for this experiment to minimize the required processing effort and maximize the throughput requirements on the memory system. As in our join benchmarks, we assume that the memory for the scan result is pre-allocated to see the pure overhead of memory encryption and decryption. We determine the scan throughput by dividing the size of the compressed input array by the scan runtime.

5.1 Single-Threaded Column Scans

Before stressing the limits of the memory encryption engine using multiple threads, we start by analyzing the encryption/ decryption overhead for a single-threaded scan. To this end, we compare the read throughput of a column scan on a single CPU core between enclave code reading enclave data (SGX Data in Enclave), enclave code reading plain data (SGX Data outside Enclave), and our baseline, non-enclave code reading plain data (Plain CPU). Additionally, we vary the size of the scanned column from 4 kB to 16 GB. To show the effect of CPU caches, we first execute 10 warm-up scans and afterward start the time measurement for another 1000 scans.

As we see on the left side of Figure 13, again, there is no SGX-inherent overhead if data is cache resident. This is expected because data in caches is in plain text and does not require any decryption. When the data does not fit into the L3 cache, the column scan over encrypted enclave data (stored in the EPC) is only minimally (i.e., ≈3 %) slower than the scan over unencrypted data. This is a clear improvement over SGXv1, which showed a much larger performance loss of up to 75 % even for simple non-vectorized scans [26].

5.2 Multi-threaded Execution

Next, we explore if the memory encryption engine inside SGXv2 becomes a bottleneck when increasing the scan throughput by using multi-threading, as it is done in many modern DBMS.

To do this, we execute the same scan algorithm as in the previous experiment over a 16 GB column while scaling the number of used cores from 1 to 16. As shown in Figure 14, the enclave memory protection mechanisms do not become a bottleneck on our processor. The scaling behavior is equal between SGX and plain CPU. Further, in both settings, our algorithm is able to reach the memory bandwidth limit with 16 cores. We verified this with Intel VTune for the plain CPU scan.

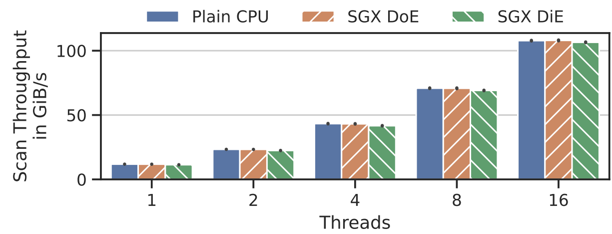

Figure 14: Column scan throughput scales with more threads. Scaling behavior is equal between running inside the enclave and outside. There seems to be no bottleneck caused by memory encryption or decryption in SGXv2.

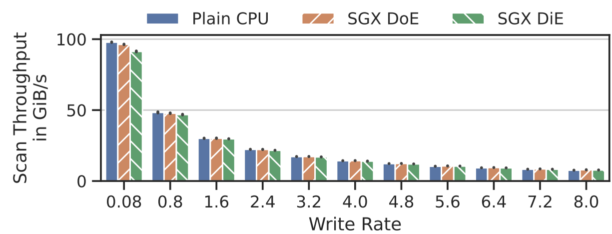

Figure 15: Varying selectivity to increase the write rate of the scan benchmark. Uses a scan that returns matching indexes. Size of input: 4 GB. 16 Threads. An increased write rate does not cause an increased overhead in SGXv2.

5.3 Scans with Varying Read/Write Ratio

The previous experiments are both read-heavy and only have to write a small output as tightly packed bit vectors. As a consequence, the memory encryption engine mainly performs decryption when loading data from the EPC and only a limited amount of encryption. This leaves open the question if increased amounts of writes stress the memory encryption to a degree where it cannot keep up with the column scan. To check if the ratio of reads and writes to memory influences the performance of scans inside the enclave, we evaluate a second scan with a variable write ratio (i.e., by using different selectivities). Instead of a bit vector, the second scan implementation returns 64-bit integers (i.e., row indexes) for the values that match the range criterion. Since a 64-bit index is 8 times larger than an 8-bit value, the write rate of this scan is 8 times the selectivity.

As can be seen in Figure 15, an increased write rate does not lead to a higher reduction of the read throughput inside the enclave compared to outside. The read throughput of the column scan decreases to the same degree inside and outside the enclave.

Lessons learned. The SGXv2 memory encryption mechanism causes minor overheads for column scans optimized for maximum memory throughput. The performance of this operation is, generally speaking, equivalent between normal CPU and enclave mode. This insight is independent of the number of CPUs employed for the scan and the ratio of reads and writes. We expect that other bandwidth-bound algorithms with linear access patterns, such as scalar functions, will behave similarly.

5.4 Scans and NUMA

As introduced in Sections 2 and 4.3, SGXv2 supports enclaves on multi-socket servers. In the context of scans, this theoretically enables the utilization of additional cores available on the second NUMA node, further parallelizing scan algorithms to increase performance. However, as NUMA-local memory allocations and thread pinning are currently not available in SGXv2 enclaves,

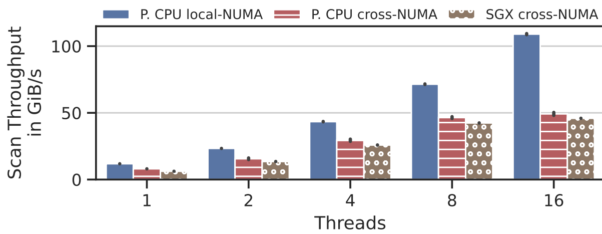

Figure 16: Cross-NUMA column scan throughput in an SGX enclave compared with a cross-NUMA scan without SGX and a local-NUMA scan without SGX. UPI traffic encryption in SGX causes an additional performance decrease.

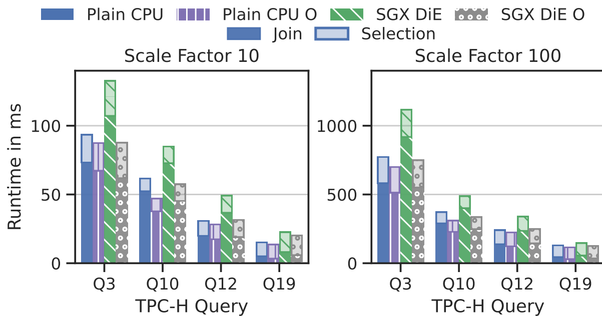

Figure 17: Runtime of four TPC-H queries at scale factor 10 and 100 using the RHO join. Comparison between outside the enclave and inside the enclave, both with and without optimization. Our optimization reduces the performance difference between outside and inside the enclave.

scan threads may be forced to access EPC data on remote nodes over the UPI link, which incurs additional overhead due to encryption. To quantify the overhead of UPI encryption, we analyze the throughput characteristics of cross-NUMA scans.

We again benchmark extreme cases and use the observation that the Linux kernel allocates EPC pages on the local node. To build a cross-NUMA column scan benchmark, we pin the scan execution threads to the node on which the enclave was not allocated. This ensures that all read and write operations cross the UPI link. Using this technique, we compare the read throughput of a NUMA-local plain CPU scan with the performance of a cross-NUMA plain CPU scan and a cross-NUMA scan reading and writing encrypted data inside an SGXv2 enclave. The benchmarked scans use 1 to 16 threads.

Figure 16 shows the results of our benchmark. The measurements show a lower throughput for cross-NUMA scans, especially when using multiple threads. It is important to note here, that the theoretical upper bound for throughput of the 3 UPI links between the sockets in our server is 67.2 GB/s and executing the scan with 8 and 16 threads approaches this upper limit. When comparing the plain CPU cross-NUMA scan performance with its enclave counterpart, we measured 77 % of the baseline throughput with a single thread. This relative performance increases with the number of threads up to 96 % for 16 threads, where the scan is bound by the general speed of the UPI links.

Lessons learned. Cross-NUMA memory access reduces scan throughput in SGX enclaves further than outside enclaves. Thus, optimizations for local access would have a more significant effect if they were possible.

6 COMPOSING OPERATORS IN QUERIES

Finally, we investigate the performance of our optimized join and scan operators when composed in query plans. The goals of this experiment are threefold. First, we examine if the effect of the unrolling and instruction reordering optimization is still relevant in the bigger picture. Second, we investigate the influence of result materialization and TPC-H data characteristics (different table sizes, wider SIMD-scan input values, selective joins, and different payload data) on performance. Third, we assess if the overall query execution performance in an SGXv2 enclave is competitive with the native setting.

For this evaluation, we used TPC-H queries 3, 10, 12, and 19 as workload because these queries mainly consist of scans and joins. We run the queries with the TPC-H data at scale factor 10 and 100 as input. To see the effects of the operators investigated in this paper more clearly, we remove all other operators, replace the final aggregation with count(*), and represent dates and categorical strings as integers, mimicking the evaluation setup for CrkJoin [26]. All operators and queries are implemented in our C++ framework and compiled before execution. The queries are implemented using the optimized RHO join from Section 4. All 16 cores available on one hardware socket are used for parallelism.

The results in Figure 17 show that the optimizations introduced in the previous sections (settings annotated with O) indeed result in performance improvements on the query level and reduce the query runtime by 12 % (Q19, SF10) to 39 % (Q12, SF10) compared to the unoptimized version. Compared to the execution on the native CPU, the overhead of running the queries in SGX enclaves is reduced from 38 % on average to 14 %. As expected, scan & selection performance is very similar across settings. Therefore, the performance difference between the enclave and native setup primarily originates from the join implementation. Result materialization and the data characteristics of TPC-H do not introduce unexpected performance differences between native and SGXv2 execution.

Lessons learned. Using state-of-the-art operator implementations combined with SGXv2-specific optimizations enables query plan execution at near-native performance inside an enclave.

7 DISCUSSION OF A PERFORMANCE MODEL

Given the evaluation results of our paper for in-memory query operators, one remaining question is: How can the findings be transferred to other algorithms and data structures? For such a task, a unified cost model that incorporates memory access costs and optimizations for query execution operators is required.

As a first approximation of such a model, we summarized our findings in Table 2. Given the memory access pattern, as well as the processed data size, the table shows the estimated slowdown to be expected of an algorithm running in an SGXv2 enclave compared to the plain CPU. This table can already guide performance optimizations, such as partitioning or manual unrolling and reordering (as presented in our paper). For example, as we have seen in Section 4.2, histogram computations have a data-dependent write memory access pattern. Thus, even if data fits in the last level cache (LLC), we can expect a high overhead of up to 225 % and thus should consider applying additional optimization in an SGXv2 enclave.

Developing a more sophisticated model that can capture more nuanced performance aspects, such as the order of read and write accesses, and predict exact algorithm performance or slowdown is very involved, as it requires modeling interactions between

Table 2: High-level performance model for the expected slowdown of algorithms in SGXv2 enclaves. As shown in our work, performance is mainly influenced by data size (DS) and the access pattern (AP) of an algorithm.

DS \ AP	linear read/write	independent random read/write	data-dependent write
$< LLC$	None (0 %)	None (0 %)	High (up to 225 %)
$> LLC$	None (3 %)	High (up to 200 %)	High (up to 800 %)

access patterns, concrete implementation, and CPU optimizations like speculative execution. For example, Manegold et al. [29] created a performance model for in-memory query operators for traditional single-core CPUs, and we believe that such a cost model would be a good starting point. Still, it needs to be updated to reflect not only the effects of multi-cores and NUMA but also SGX-related aspects, such as the effect of data-dependent writes, which we have seen to severely impact the performance of query operators. As such, creating such a more detailed model is out of the scope of this paper but represents an important avenue of future work.

8 RELATED WORK

This study has three main areas of related work: Benchmarks and performance evaluations for SGX, specialized enclave database systems and architectures, and recent evaluations of SGXv2.

Benchmarks and performance evaluations for SGX. Multiple papers introduce benchmarks suites for SGXv1 [23, 25, 39] to analyze the performance characteristics of enclaves. Their approach is similar to ours in that they port existing workloads [23, 25] or benchmark suites [39] to Intel SGX and compare the performance to native execution. These efforts do not concentrate on specific application domains like databases, and they were conducted before the introduction of SGXv2.

Specialized DBMS for SGX. There are multiple proposals for data management systems inside SGXv1 enclaves that suggest approaches to circumvent the performance degradations caused by the limited EPC size [1, 22, 35, 37] or investigate the theoretical enclave performance without any memory limit [40]. Most related to our work are the publications by Maliszewski et al. [26, 27] analyzing the performance of join algorithms in SGXv1 enclaves. They observe that radix joins have beneficial properties for enclaves, but all joins greatly suffer from slow random access and EPC paging. To circumvent these problems, the authors develop CrkJoin [26] that reaches superior in-enclave performance in their evaluation. However, our study shows that the CrkJoin optimizations are irrelevant in SGXv2 due to the eliminated EPC bottleneck. To achieve near-native performance for database workloads in the latest SGX generation, new optimizations and a thorough understanding of the performance characteristics of SGXv2 are required.

SGXv2 performance. To the best of our knowledge, there is still minimal research on the performance characteristics of SGXv2 [4, 10, 24, 32]. Aside from our previous studies on OLTP workloads [10] and neural network inference [24], Miwa and Matsuo have examined SGXv2's performance for HPC [32]. Additionally, Battiston et al. are concurrently studying the performance of running DuckDB inside an SGXv2 enclave [4] using the Gramine library operating system [13]. This paper extends our previous work by focusing on modern query execution algorithms, data throughput, and an in-depth investigation of slowdown sources at an architectural level. Through a detailed analysis of SGXv2 performance characteristics in the OLAP context, we identify new optimizations, such as manual loop unrolling and instruction reordering, to enhance the throughput of in-memory algorithms.

9 CONCLUSION

This paper provides a comprehensive analysis of Intel SGXv2, evaluating its advantages and limitations for secure, high-performance analytical databases. Among other insights, our study offers three main contributions: Firstly, we demonstrated that state-of-the-art main memory and cache-optimized join algorithms perform better inside SGXv2 than those optimized for the discontinued SGXv1 due to changed hardware characteristics. Secondly, we uncovered previously unknown overheads caused by a side channel mitigation enabled inside the enclave but not outside, and we showed how existing algorithms can be optimized to circumvent this slowdown. Finally, we verified that SGXv2-optimized operators enable the execution of query plans with performance on par with execution outside the enclave. Overall, our findings highlight the potential of SGXv2 for analytical workloads and show that a deep understanding of its performance characteristics is crucial for designing high-performance DBMSs.

ACKNOWLEDGMENTS

This work was supported by the safeFBDC Research Fund of the German Federal Ministry for Economic Affairs and Climate Action (Grant Agreement No 01MK21002K), the National Research Center for Applied Cybersecurity ATHENE, and by SAP SE. We also thank Nicolae Popovici from Intel for reproducing some experiments on the newest processor generation.

REFERENCES

[1] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1511–1525. https://doi.org/10.1145/3318464.3386141

[2] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, Jiaqi Liu, Lukas M. Maas, Akshay Mata, Ishai Menache, Justin Moeller, Vivek Narasayya, Matthaios Olma, Morgan Oslake, Elnaz Rezai, Yi Shan, Manoj Syamala, Shize Xu, and Vasileios Zois. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proceedings of the VLDB Endowment* 16, 13 (Sept. 2023), 4202–4215. https://doi.org/10.14778/3625054.3625058

[3] 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, Brisbane, QLD, Australia, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[4] Ilaria Battiston, Lotte Felius, Sam Ansmink, Laurens Kuiper, and Peter Boncz. 2024. DuckDB-SGX2: The Good, The Bad and The Ugly within Confidential Analytical Query Processing. In *Proceedings of the 20th International Workshop on Data Management on New Hardware (DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3662010.3663447

[5] 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 (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1989323.1989328

[6] Marion Bonnet. 2023. Cloud Assets the Biggest Targets for Cyberattacks, as Data Breaches Increase. https://www.thalesgroup.com/en/worldwide/security/press_release/cloud-assets-biggest-targets-cyberattacks-data-breaches-increase

[7] David Bronleewe, Hormuzd Khosravi, Shanmathi Rajasekar, Shiny Sebastian, and Raghuram Yeluri. 2022. Runtime Encryption of Memory with Intel® Total Memory Encryption–Multi-Key (Intel® TME-MK). https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html

[8] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf

[9] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*

(DaMoN'19). Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/3329785.3329918
[10] 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 (DaMoN'22)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3533737.3535098
[11] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment* 13, 2 (Oct. 2019), 169–183. https://doi.org/10.14778/3364324.3364331
[12] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Found. Trends Databases* 8, 1-2 (2017), 1–130. https://doi.org/10.1561/1900000058
[13] Gramine Project. 2023. Gramine - a Library OS for Unmodified Applications. https://gramineproject.io/
[14] Intel Corporation. 2018. Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html
[15] Intel Corporation. 2022. Intel® Software Guard Extensions SDK for Linux* OS Release Notes. https://download.01.org/intel-sgx/sgx-linux/2.18/docs/Intel_SGX_SDK_Release_Notes_Linux_2.18_Open_Source.pdf
[16] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, & 2D): Instruction Set Reference, A-Z*. Technical Report 325383-081US. Intel Corporation. 2522 pages. https://cdrdv2.intel.com/v1/dl/getContent/671110
[17] Intel Corporation. 2023. Intel® Software Guard Extensions SDK for Linux OS Developer Reference. https://download.01.org/intel-sgx/sgx-linux/2.21/docs/Intel_SGX_Developer_Reference_Linux_2.21_Open_Source.pdf
[18] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. 2021. Supporting Intel SGX on Multi-Socket Platforms. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf
[19] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proceedings of the VLDB Endowment* 11, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3284966
[20] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. 2013. Experimental Evaluation of NUMA Effects on Database Management Systems. In *Datenbanksysteme Für Business, Technologie Und Web (BTW), 15. Fachtagung Des GI-Fachbereichs "Datenbanken Und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings (LNI, Vol. P-214)*, Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen (Eds.). GI, Magdeburg, Germany, 185–204. https://dl.gi.de/handle/20.500.12116/17321
[21] 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 (Aug. 2009), 1378–1389. https://doi.org/10.14778/1687553.1687564
[22] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3302424.3303951
[23] Sandeep Kumar, Abhisek Panda, and Smruti R. Sarangi. 2022. SGXGauge: A Comprehensive Benchmark Suite for Intel SGX. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 135–137. https://doi.org/10.1109/ISPASS55109.2022.00014
[24] Adrian Lutsch, Gagandeep Singh, Martin Mundt, Ragnar Mogk, and Carsten Binnig. 2023. Benchmarking the Second Generation of Intel SGX for Machine Learning Workloads. In *BTW 2023*. Gesellschaft für Informatik e.V., Bonn, 711–717. https://doi.org/10.18420/BTW2023-44
[25] Mohammad Mahhouk, Nico Weichbrodt, and Rüdiger Kapitza. 2021. SGXoMeter: Open and Modular Benchmarking for Intel SGX. In *Proceedings of the 14th European Workshop on Systems Security (EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 55–61. https://doi.org/10.1145/3447852.3458722
[26] Kajetan Maliszewski, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. Cracking-Like Join for Trusted Execution Environments. *Proceedings of the VLDB Endowment* 16, 9 (May 2023), 2330–2343. https://doi.org/10.14778/3598581.3598602
[27] 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 (Nov. 2021), 659–672. https://doi.org/10.14778/3494124.3494146
[28] S. Manegold, P. Boncz, and M. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (July 2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210
[29] Stefan Manegold, Peter Boncz, and Martin L. Kersten. 2002. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 191–202. https://www.sciencedirect.com/science/article/pii/B9781558608696500251
[30] 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 (HASP '16)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/2948618.2954331
[31] 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. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/2487726.2488368
[32] Shinobu Miwa and Shin'Ichiro Matsuo. 2023. Analyzing the Performance Impact of HPC Workloads with Gramine+SGX on 3rd Generation Xeon Scalable Processors. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 1850–1858. https://doi.org/10.1145/3624062.3624267
[33] Siani Pearson and Azzedine Benameur. 2010. Privacy, Security and Trust Issues Arising from Cloud Computing. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, Indianapolis, IN, USA, 693–702. https://doi.org/10.1109/CloudCom.2010.66
[34] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645
[35] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 264–278. https://doi.org/10.1109/SP.2018.00025
[36] 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 (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1961–1976. https://doi.org/10.1145/2882903.2882917
[37] 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 (April 2021), 1019–1032. https://doi.org/10.14778/3447689.3447705
[38] Various Authors. 2023. Prctl(2) - Linux Manual Page. https://man7.org/linux/man-pages/man2/prctl.2.html
[39] Sébastien Vaucher, Valerio Schiavoni, and Pascal Felber. 2019. Short Paper: Stress-SGX: Load and Stress Your Enclaves for Fun and Profit. In *Networked Systems (Lecture Notes in Computer Science)*, Andreas Podelski and François Taïani (Eds.). Springer International Publishing, Cham, 358–363. https://doi.org/10.1007/978-3-030-05529-5_24
[40] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: A Scalable Encrypted Database with Full SQL Query Support. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (July 2019), 370–388. https://doi.org/10.2478/popets-2019-0052
[41] Zack Whittaker. 2023. Danish Cloud Host Says Customers 'lost All Data' after Ransomware Attack. https://techcrunch.com/2023/08/23/cloudnordic-azero-cloud-host-ransomware/
[42] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units. *Proceedings of the VLDB Endowment* 2, 1 (Aug. 2009), 385–394. https://doi.org/10.14778/1687627.1687671
[43] 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. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng