How to make Secure Storage fast for DBMSs in Intel SGXv2

Adrian Lutsch*
Technical University of Darmstadt
Darmstadt, Germany

Christian Franck*
Technical University of Darmstadt
Darmstadt, Germany

Muhammad El-Hindi
Technical University of Munich
Munich, Germany

Norman May
SAP SE
Walldorf, Germany

Zsolt István
Technical University of Darmstadt
Darmstadt, Germany

Carsten Binnig
Technical University of Darmstadt &
DFKI & hessian.AI
Darmstadt, Germany

Abstract

Recent Trusted Execution Environments based on Intel SGXv2 enable fast and confidential in-memory processing for DBMSs. However, secure persistence remains a major performance and security challenge. While native SGX-based storage mechanisms provide confidentiality and integrity out of the box, they incur high overheads for DBMSs. In this paper, we analyze the overheads and introduce novel techniques for secure and fast DBMS storage. As we show, these techniques reduce the high storage overheads of native SGX storage mechanisms to negligible overheads, enabling practical, high-performance, secure storage for cloud databases.

CCS Concepts

• **Information systems** → **Transaction logging**; • **Security and privacy** → **Database and storage security**.

Keywords

Database Storage, Write-Ahead Logging, TEE, Intel SGX

ACM Reference Format:
Adrian Lutsch, Christian Franck, Muhammad El-Hindi, Norman May, Zsolt István, and Carsten Binnig. 2026. How to make Secure Storage fast for DBMSs in Intel SGXv2. In *22nd International Workshop on Data Management on New Hardware (DaMoN '26), May 31–June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3789237.3809121

1 Introduction

Securing database systems in the cloud. Cloud database systems increase scalability, elasticity, and cost-efficiency [4, 11, 13]. However, they raise concerns about data security and privacy [41, 45, 54]. For example, several risks arise from untrusted infrastructure, malicious insiders, or regulatory non-compliance [26]. These challenges limit the adoption of cloud database systems for use cases that require strong confidentiality and integrity guarantees, such as healthcare, finance, and government workloads.

SGXv2 is fast but only protects data in memory, not on disk. Hardware-based Trusted Execution Environments (TEEs), such as Intel Software Guard Extensions (SGX) [37], have been proposed to mitigate these risks. Intel SGX provides hardware-enforced isolation: code and data execute inside enclaves that remain protected even from the cloud operator [17]. Recent work shows that server-grade SGX (SGXv2) can provide near-native performance for DBMSs and can support OLTP and OLAP processing with small overheads [19, 33–35]. However, this only holds if data can be cached in memory and durability is ignored.

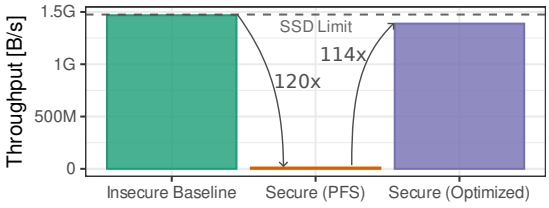

Figure 1: Insecure baseline vs. secure storage. The standard library in SGX (PFS) has extremely high overheads. Our optimized, secure storage achieves baseline performance.

DBMSs require additional action to secure storage. To provide durability, DBMSs must persist state to storage outside the enclave. However, while TEEs such as Intel SGX effectively protect the privacy and integrity of in-memory computations, data outside the enclave boundaries, such as data on disk, remains unprotected. DBMSs, therefore, must use cryptographic mechanisms to ensure the confidentiality and integrity of durable data on disk [24]. For this, Intel's SGX Software Development Kit (SDK) provides the Protected File System (SGX PFS) library as a general-purpose solution for secure file I/O [21].

The SGX I/O library has high overheads. Using SGX PFS for implementing secure DBMS storage comes at a high cost. To illustrate the overhead, we implemented write-ahead-log-based durability for an in-memory key-value store. Figure 1 shows the throughput of two variants: one outside an SGX enclave (left) and a naive in-enclave implementation (middle) that uses SGX PFS. As we can see, PFS reduces throughput by a factor of 120×. However, as we show in this paper, by designing a write-ahead-logging-optimized secure storage path, we can remove nearly all of the overhead and achieve very similar performance inside the enclave (right).

Scope and contributions. In this paper, our main contribution is to study techniques for achieving fast yet secure storage access for DBMSs using Intel SGX. In particular, we explain why generic solutions, such as SGX PFS, underperform (Section 4.1) and present three complementary techniques for fast, secure storage on Intel SGX: (1) DBMS-aware co-design of security guarantees (Section 4.2),

*Both authors contributed equally to this research.

DaMoN '26, Bengaluru, India

ACM ISBN 979-8-4007-2455-8/26/05
https://doi.org/10.1145/3789237.3809121

(2) low-latency I/O with durability guarantees using kernel-bypass mechanisms (Section 4.3), (3) asynchronous message passing between the enclave and the untrusted environment (Section 4.4). Our results, based on a secure write-ahead log prototype, show that with these techniques, the performance overhead of secure storage can be reduced to negligible levels (Section 4.5), enabling practical trusted storage for cloud databases. Additionally, we discuss how the results translate to other database use cases and other Trusted Execution Environments, such as AMD SEV-SNP [29], Intel TDX [16, 22]), and AWS Nitro Enclaves [7, 35] (Section 5). We focus on Intel SGX as a widely deployed and well-studied hardware-based TEE platform.

2 Background on Intel SGX and Storage

This section summarizes basic Intel SGX concepts and the SGX PFS design to understand the sources of its overheads.

Main Intel SGX guarantees. Intel SGX implements *enclaves* as isolated execution environments whose memory is protected by the CPU [28, 37]. Two key features are *isolation* and *attestation*. Isolation ensures that enclave code and data remain confidential and are only accessible as plaintext within the CPU caches during execution. It is achieved by CPU-based memory encryption and a special CPU enclave mode. Attestation allows a local or remote party to verify the code within an enclave (i.e., its identity) and detect unauthorized modifications. It is achieved by hashing the enclave's initial state and using a special CPU instruction only available in enclave mode.

Threat Model. We work within the typical threat model for server-grade SGX enclaves: we aim to protect against adversaries with full administrative control over the operating system, the hypervisor, and all hardware except the CPU and memory subsystem [17, 28]. This includes control over the disk. We do not assume trusted disks, transparently encrypted drives, or similar technologies. Adversaries with these capabilities can read and manipulate all inputs and outputs of SGX enclaves at runtime and offline. They can, however, not access the enclave's memory, forge enclave attestation reports to create fake enclaves, or break state-of-the-art encryption, such as AES-GCM. We assume that the adversaries are not authenticated clients of the enclave-protected DBMS. In the face of these adversaries, we aim to ensure confidentiality and integrity for the write-ahead log at enclave runtime. During enclave restarts, we aim to protect against all unauthorized changes to the write-ahead log except for the removal of blocks at the end of the log. We note that the methods we propose cannot prevent an adversary from deleting or altering data written to disk; they only protect confidentiality and enable the enclave to detect changes and respond accordingly. We believe that detecting data manipulation is sufficient to disincentivize attacks among database and infrastructure service providers because the detection of such data manipulation would cause a severe loss of reputation for a service provider. It also allows tenants to restore their databases from backups when they detect changes. Like all SGX-based systems, we do not protect against denial-of-service attacks, which are trivial for service providers to implement and cannot be prevented with SGX.

Interaction with storage devices. In SGX, the trusted computing base (TCB) is kept small by design and is thus limited to the CPU, its caches, and enclave memory [17, 37]. The operating system, hypervisor, and peripherals (including storage) are untrusted. Consequently, enclaves cannot directly access hardware devices and must rely on the OS to, e.g., execute I/O operations. Interactions with storage devices thus require *enclave transitions*, which switch execution between enclave and non-enclave mode. We refer to transitions into an enclave as `ECall` and transitions out as `OCall`.

Overhead of enclave transitions. Enclave transitions are expensive [53] due to context switching and cache/TLB effects. Since I/O requires system calls executed outside the enclave, frequent reads and writes can substantially increase latency and reduce throughput for database workloads [12, 51]. To reduce transition overheads, prior work proposed *switchless* designs [12, 40, 49, 50, 53], which delegate requests to worker threads outside the enclave, thereby avoiding the cost of enclave transitions. However, most existing switchless approaches block the requesting enclave thread until completion [40, 49, 50], limiting concurrency even if the untrusted side uses asynchronous I/O interfaces such as io_uring [48] or SPDK [18] for fast I/O.

Overheads of standard library PFS. SGX's standard approach to secure storage, the PFS library, introduces significant overhead, as shown in Figure 1. To better understand its overheads, we review the anatomy of the write path: SGX PFS uses memory-mapped I/O to write an encrypted file to disk. It splits and handles the file on the granularity of 4 KiB blocks and executes the following steps to implement a secure write: First, when `sgx_fwrite` (equivalent of `fwrite`) is called, the input data is copied into a plain-text in-enclave cache managed by SGX PFS. Second, once the in-enclave cache is full, or when `sgx_fflush` is called explicitly, the following substeps are executed: (a) The library writes a recovery file containing the old content of all disk blocks to be overwritten. This ensures the integrity of the file in case the enclave is interrupted during the following steps. (b) The library sets an update-in-progress flag in the file's metadata, which is necessary for file recovery. (c) The library encrypts all updated plaintext data blocks in the cache and writes them to the memory-mapped file region. The encryption ensures the confidentiality and internal integrity of the data blocks. (d) The library updates the Merkle tree used for overall file integrity protection, encrypts the updated nodes, and also writes them to the memory-mapped file region. The Merkle tree ensures that no adversary can reorder, replace, add, or remove blocks in the file without being detected. (e) Finally, the library updates the metadata block with the new root hash and resets the update-in-progress flag. This atomically updates the file version and indicates the update is complete. After all these steps, however, the updated file contents are still only written to the memory-mapped I/O region and are not forced to disk. Thus, SGX PFS provides neither ordering guarantees nor guarantees that data reaches stable storage, both of which are needed for DBMSs, particularly for durable WAL writes. To address this issue, we added a new function `sgx_fsync` to SGX PFS. This function uses an additional `OCall` to invoke `msync` to force a write of all changes to disk.

Summary. The main sources of overhead in secure I/O are: the costs associated with encryption and integrity protection; I/O access itself, which requires enclave transitions; and the lack of asynchronous execution in the I/O subsystem. In the following, we quantify the I/O overheads of PFS when used in a DBMS.

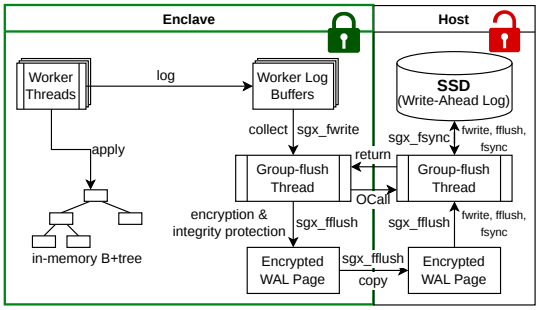

Figure 2: Secure WAL prototype. Steps starting with `sgx_` are used in SGX PFS. Other steps are equal between implementations (apply, log) or used in the optimized encryption and integrity protection scheme.

3 Experimental Setup for Analysis

To quantify the overheads of PFS in DBMSs, we focus on write-ahead logging (WAL). Write-ahead logging is a challenging workload because it is on the critical path in OLTP DBMSs during commit, thereby imposing strict latency and durability requirements.

Secure WAL prototype. To analyze a database-style WAL access pattern, we implement an in-memory key-value store with WAL-based durability. The prototype design is depicted in Figure 2. Our prototype executes single-statement transactions that read and update records stored in an in-memory B^+-tree. Our B^+-tree uses optimistic lock coupling [31] to allow for concurrent transactions. After applying the changes in the B^+-tree, each worker thread buffers log records locally. A dedicated group-flush thread periodically drains these buffers into a shared buffer [39]. When the shared buffer is full, the group-flush thread appends its contents to the write-ahead log using SGX PFS, using the three functions introduced above. We call the contents of one buffer written to the end of the WAL a *WAL page*. After the I/O completes, workers are notified, and the corresponding transactions are committed. In our experiments, we use one fixed core for the background group-flush thread and all other cores of the same CPU for worker threads. To isolate storage costs, we generate client requests within the enclave (not shown in Figure 2). This avoids network overhead and additional `OCall`s.

Hardware. We run all experiments on a server with two Intel Xeon Gold 6326 CPUs (16 cores each) with hyperthreading disabled. We use only one of the two CPUs to prevent NUMA and additional I/O latency. Each socket has 256 GB DDR4-3200 ECC RAM across 8 memory channels. We configure 64 GB as SGX Enclave Page Cache (EPC). Our experiments write to a Micron 7450 PRO 960GB NVMe SSD with up to 1.4 GB/s sequential write speed and up to 82000 random write IOPS. The server runs Ubuntu 24.04.3 with kernel 6.14, SGX SDK 2.27, SPDK v24.05, and Boost 1.89. We use GCC 14 for compiling.

4 How to make Secure Storage Fast?

In this section, we analyze the overheads introduced by the standard SGX Protected File System library (SGX PFS) and then introduce our main optimizations step by step: optimizing the encryption and integrity protection for the WAL (Section 4.2), optimizing the storage access method (Section 4.3), and making the whole I/O subsystem asynchronous (Section 4.4). Finally, we compare the performance of the baseline, all intermediate optimizations, and our final prototype to a non-SGX baseline (Section 4.5).

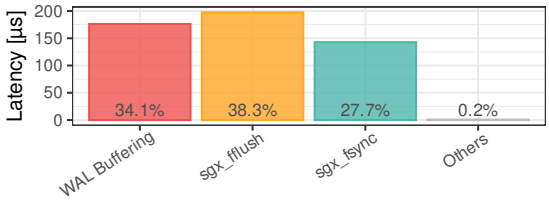

Figure 3: Initial latency distribution of WAL prototype. Encryption, integrity protection, durability (`sgx_fflush`), and I/O (`sgx_fsync`) of SGX PFS dominate the latency. Steps contributing less than 3% summarized as "Others"

4.1 Analysis of PFS-based WAL

As described in Sections 2 and 3, writing data using SGX PFS is a multi-step process. We start by identifying which steps in the WAL write process introduce the highest latency in our WAL prototype.

Figure 3 shows the median latencies of the steps in the WAL write process. We measured those at a write rate close to saturation to minimize the batching overhead visible in WAL buffering. When looking at the absolute throughput, however, the disk is experiencing only very low load, and the latency is dominated by the steps implemented in SGX PFS. The two main bottlenecks in SGX PFS are `sgx_fflush` and `sgx_fsync`: `sgx_fflush` is slow because it contains the integrity and durability protection process, including the write of the recovery file and the update of multiple Merkle tree nodes in addition to the written WAL page. `sgx_fsync` is slow because it is based on `msync`, which requires the OS first to identify all changed memory pages and then write them to disk. Next, we investigate how the integrity protection can be made cheaper by leveraging domain knowledge of write-ahead logging.

4.2 Optimizing Encryption and Integrity

The previous experiment raised the question of whether the Merkle tree, the recovery file, and the write amplification they cause are necessary for the WAL workload.

Merkle trees are not required for WAL. Merkle trees are a general-purpose mechanism for file integrity protection. In addition to protecting the overall file integrity, they have three upsides: They support integrity verification of random reads and writes with logarithmic complexity, protect the order of all file contents, and ensure that all file contents belong to the same file version. However, none of these three guarantees is required for a write-ahead log. Fast integrity verification for random reads and writes is not required because the WAL has an append-only write pattern and a sequential read pattern during recovery. Enforcing a strict ordering of WAL pages is not necessary because WAL entries contain timestamps or log sequence numbers that enable the DBMS to restore their order even if they got shuffled in storage, as long as all WAL pages are available and unchanged. Finally, file versioning is not required because each encrypted WAL page has only one version, and the overall WAL version can be determined from the last page written. Overall, for a secure WAL, we only need to ensure confidentiality and that the DBMS can detect that it is reading the exact set of WAL pages it wrote, without removed, changed, or added

pages. Consequently, our optimization of encryption and integrity protection for WAL aims at removing the Merkle tree.

Optimized encryption and integrity protection. In our scheme, the group flush thread encrypts WAL pages using AES-GCM, an authenticated encryption algorithm. Additionally, it adds a header to each encrypted WAL page. This header contains the encryption nonce, a WAL page ID, and the AES-GCM message authentication code (MAC). WAL page IDs are unique, start at 0, and are incremented by one for each encrypted page, so that the series of IDs has no gaps. We include the WAL page ID in the encryption as authenticated metadata, ensuring it cannot be changed without invalidating the MAC. All steps for encryption and integrity protection happen inside the enclave. No plain-text data leaves the enclave. We do not use any integrity protection between encrypted pages, such as a Merkle tree or a hash chain. This provides all the required confidentiality and integrity guarantees that SGX PFS provides, but in a simpler solution: Authenticated encryption of WAL pages ensures confidentiality, protects against changes to individual encrypted WAL pages on disk, and prevents inclusion of fake WAL pages. Since each WAL page is written only once, it is impossible to rollback individual encrypted pages to outdated versions because there are no outdated versions. Checking the WAL page IDs for continuity and uniqueness ensures that no pages in the WAL are omitted or duplicated. A similar scheme has been used in other works [14, 42, 43, 52]. A more detailed discussion of possible attacks and how the scheme prevents them is available in Appendix A. Since our design removes the Merkle tree, each write of a 4 KiB WAL page requires encrypting only that page and atomically writing it to the end of the WAL; writing a recovery file for durability purposes is no longer required.

Thereby, we address the two main issues of SGX PFS: (1) Updating and encrypting Merkle tree nodes, and writing a recovery file, as done in `sgx_fflush`, are no longer required. (2) Since the file is now append-only, we can easily switch from memory-mapped files and `msync` to writing the WAL with `fwrite` and persisting it with `fflush` and `fsync`, removing the durability issues of memory-mapped files and the need to find changed pages in memory.

Optimizing the secure WAL. The optimized encryption and integrity protection scheme simplifies the steps required for making a change durable in our secure WAL prototype. It processes writes in the following steps: (1) apply the change to the in-memory B^+-tree, (2) write a log entry into the thread-local WAL buffer, (3) collect log entries from thread-local buffers, (4) encrypt the collected WAL page using authenticated encryption, (5) add the page header, including the MAC generated by AES-GCM, to the encrypted page and copy both to shared memory outside the enclave, (6) transition out of the enclave (`OCall`) to write the new encrypted page using `fwrite`, `fflush`, and `fsync`, (7) transition back into the enclave to confirm the write. The first three and the final step are identical to the baseline using SGX PFS. The steps are visualized in Figure 2.

Encryption APIs influence performance. Implementing the optimized WAL encryption and integrity protection raises the question of which API to use for page encryption with AES-GCM. The SGX SDK offers three APIs for authenticated encryption at varying abstraction levels, which build on each other and provide different performance and security guarantees: The high-level `sgx_seal` API [24] builds on the intermediate Trusted Cryptography library (TCrypto) [24], which in turn utilizes the low-level Intel Cryptography Primitives library (ICP) [25]. To identify performance differences and overheads, we compared the three APIs in a microbenchmark. The microbenchmark encrypts 4 KiB of random data from an input buffer into an intermediate buffer and then copies it to one of 16 possible output buffers. This microbenchmark mimics the behavior of the encryption and enclave transition steps in the WAL write process. Figure 4 depicts the average latency of one encryption and copy step using the different APIs. It shows that encryption using `sgx_seal` is over 10× slower than using the AES-GCM function in the Intel Cryptography Primitives library.

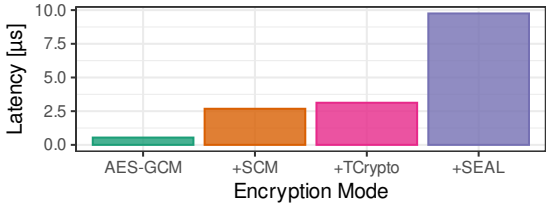

Figure 4: Latency of encrypting a 4 KiB buffer and copying the encrypted data out of the enclave with various SGX encryption APIs that build on each other. Pure AES-GCM is over 10× faster than the `sgx_seal` data functionality.

Investigation of the SGX SDK source code reveals three main sources of overhead: (1) The TCrypto library activates a side-channel protection for a power side channel [32] inside the ICP library. This increases latency by 5× (+SCM). (2) The TCrypto library randomizes stack pointers, excessively cleans buffers, and re-initializes the AES-GCM state for every encryption (+TCrypto) (3) The `sgx_seal` library derives a new random key from the CPU-provided enclave-specific key for every encryption. This increases the latency by another 3× (+SEAL). Furthermore, the sealing library must store the metadata required to recover the associated encryption parameters for each message. This deducts 560 bytes, i.e., 14% of a 4 KiB page, without accounting for WAL-specific integrity logic.

Our recommended choice: low-level API. We believe that the threat of the power side channel can be mitigated by regular key rotations, as described by Liu et al. [32]. Thus, it is unlikely in the WAL scenario. We also think that, in practice, DBMSs will not use the sealing API because it binds the encrypted data to the CPU it was encrypted on [23]. This is unwanted in a cloud setting where migration of workloads between different servers is common. Thus, we think that DBMSs will use user-provided master keys and derive all required sub-keys from them in a custom manner. As a result, an optimized, stripped-down encryption implementation based on the pure ICP library is well-suited to the WAL use case and is 15× faster than the SGX sealing library. Therefore, in our custom WAL prototype, we directly use ICP's AES-GCM functionality without power side channel mitigation.

Performance improvements. Finally, we compare the performance of the baseline WAL prototype using SGX PFS with our prototype using the optimized encryption and integrity protection scheme with optimized encryption functions. We first compare the overall median latency and throughput of both approaches. They are depicted in a latency-throughput plot in Figure 5. We can see that latency decreases with increasing throughput, as long as the I/O path can keep up with the demand. This is caused by buffering

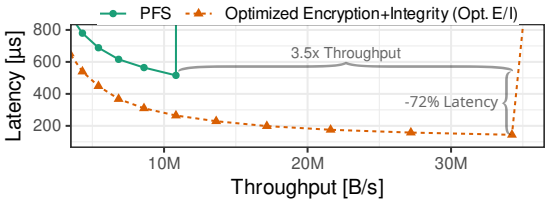

Figure 5: Latency and throughput comparison between SGX PFS and custom integrity protection. The custom solution achieves 3.3× better throughput and 67% lower latency.

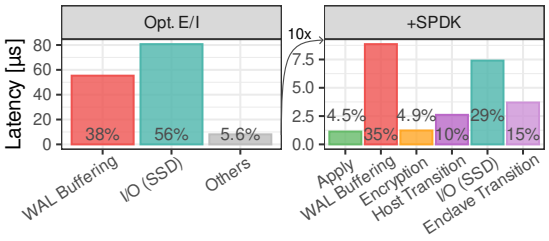

Figure 6: Left: Latency with optimized encryption and integrity protection. Slow file I/O dominates the latency. Right: Latency when using SPDK and polling the response queue. Notice the 10× difference in y-axis scaling.

latency, which decreases as buffers fill faster. As soon as the I/O path is throughput-limited, latency spikes due to queuing. Comparing the maximum throughput and minimum latency between SGX PFS and our optimized integrity protection scheme with fast encryption shows an improvement in minimum latency from 500 µs to below 150 µs. The maximum throughput improves by 3.5×, respectively.

Next, we break down the latency into the individual steps of the write pipeline. Comparing the PFS latency distribution in Figure 3 with the new latency distribution on the left side of Figure 6 reveals that encryption and integrity protection, which previously contributed 38.3% of the overall latency (`sgx_fflush`) are now less than 1% of overall latency (subsumed under "Others"). As we expected, I/O latency is also reduced by approximately 45% because fewer pages need to be written, and because we replaced `msync` with `fsync`. However, I/O latency still dominates overall latency and limits throughput.

4.3 Optimizing Storage Access

After replacing SGX PFS with our own, optimized encryption and integrity protection scheme, I/O remains the dominant source of latency (second bar in Figure 6 left). This section details our approach to reducing this overhead by enabling low-latency writes from SGX enclaves to NVMe SSDs using the Storage Performance Development Kit (SPDK).

I/O method comparison. We conducted a microbenchmark to evaluate the I/O costs of different disk write methods (Figure 7). Our findings align with prior work [27]: Blocking POSIX APIs introduce significant overhead compared to kernel-bypass techniques like SPDK. Consequently, we adopt SPDK's user-space NVMe driver, the lowest-latency and lowest-overhead method for modern NVMe storage, into our WAL prototype. SPDK makes SSD I/O operation queues available in process memory. Instead of dispatching a write command and then waiting for the SSD to confirm the write, this

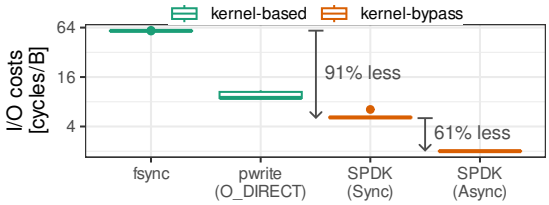

Figure 7: Cost of writing a 4 KiB buffer to disk using various I/O methods. Leveraging SPDK delivers costs orders of magnitude lower, especially when used asynchronously.

enables the process to queue multiple write requests and asynchronously read the SSD's responses from the response queue at an opportune time. This has two main upsides: (1) Having multiple queued writes allows the SSD to parallelize writes, thereby increasing I/O throughput and further reducing I/O costs per byte, as shown in the SPDK (Async) setting in Figure 7. (2) The asynchronous nature of queuing writes and dequeuing responses allows the CPU threads involved to perform other useful work while the SSD is performing the writes.

WAL prototype with SPDK backend. To assess the impact of SPDK on WAL write performance in SGX enclaves, we replaced the `fwrite`-based I/O, that we introduced with the optimized encryption and integrity protection scheme in Section 4.2, with SPDK. Now, the enclave's group-flush thread writes encrypted, integrity-protected data to an SPDK-configured DMA buffer and issues an `OCall` to submit write commands to SPDK's queues, bypassing `fwrite`, `fflush`, and `fsync`. We developed two modes of operating the SPDK queues: synchronous writes, in which the group-flush thread polls the response queue until it receives the write confirmation, and asynchronous writes, where the group-flush thread checks the response queue once and then transitions back into the enclave without waiting for specific responses.

An issue with asynchronous SPDK usage in SGX enclaves is that, in our prototype, the SPDK implementation resides outside the enclave. This avoids the need to manually extend and partially rewrite the SPDK library to support SGX, but requires an `OCall` to check the response queue. Compared to checking the SPDK queue, which requires only a negligible number of CPU cycles in the absence of new responses, enclave transitions incur a high overhead of nearly 9000 CPU cycles per direction. Thus, to avoid additional enclave transitions, we decided to check the response queue only immediately after the group-flush thread dispatched a write command. This, however, increases latency between write dispatch and commit because the response can be received only after the next page is written, at the earliest.

Performance improvements. Figure 8 compares the previous file-based WAL prototype with SPDK-optimized versions in terms of latency and throughput. With synchronous SPDK, maximum throughput improves by 6× and minimum latency decreases 83%. The throughput improvement stems directly from reduced I/O latency. With asynchronous SPDK, maximum throughput improves by another 18% because multiple writes can be queued simultaneously. However, latency is worse than in both the baseline and the synchronous SPDK setting. Overall, the asynchronous version performs far worse than the microbenchmark in Figure 7 promised,

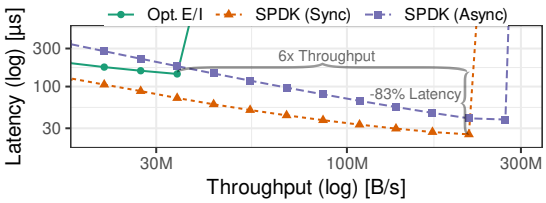

Figure 8: Latency and throughput comparison between WAL writing with `fsync` for durability and WAL writing with SPDK. Optimal latency and throughput improve by 6× (synchronous). Asynchronous SPDK achieves 18% higher throughput at the cost of increased latency.

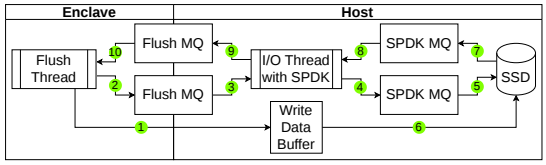

Figure 9: Asynchronous message passing design.

and there is still a large gap between the achieved 290 MB/s and the potential 1.4 GB/s throughput of the SSD.

Figure 6 compares the latency distribution before and after introducing SPDK on the NVMe SSD using the polling approach. It shows that using SPDK reduces I/O operation latency by 91%. Using SPDK, the I/O latency is within the same order of magnitude as the other steps in the pipeline.

4.4 Enabling End-to-end Asynchronous I/O

Given the optimized integrity protection and I/O, the enclave's synchronous I/O interface now incurs significant performance overhead. Figure 6 (right) shows that of the total 25 µs of latency, 6.25 µs are spent transitioning out of and back into the enclave. Since enclave transitions do not advance execution, the CPU cycles spent on them are essentially wasted. Furthermore, the previous experiment showed that the latency of the enclave transitions limits the performance of dispatching asynchronous writes using SPDK. The solution to this issue is to eliminate the need for enclave transitions and make the entire I/O path asynchronous.

Asynchronous message passing. Our architecture for a fully asynchronous storage access path in SGX enclaves leverages message passing via shared queues in host memory. It is depicted in Figure 9. In our design, a write transitions through the following steps: After the flush thread inside the enclave encrypts a WAL page and copies it into an intermediate buffer outside the enclave (Write Data Buffer, (1)), it writes a message to its flush queue (2), notifying an I/O worker thread outside the enclave that the encrypted WAL page has been written to the intermediate buffer. Then, it continues processing new requests and regularly polls the corresponding confirmation queue asynchronously (10). The I/O thread(s) outside the enclave are responsible for dispatching messages between queues: they fetch requests from the enclave threads (3), convert them into write commands for the SSD via SPDK (4), receive completions from the SSD (8), and convert them into responses for the enclave threads (9). The design is flexible and supports an arbitrary number of flush threads within the enclave and I/O worker threads outside the enclave. In practice, we use one group-flush thread inside the enclave and one I/O worker thread outside the enclave. This suffices to fully utilize the NVMe SSD in our system. Introducing an I/O thread outside the enclave reduces the number of CPU cores available for worker threads inside the enclave by one. This could, in theory, reduce throughput. In practice, however, repurposing a worker thread is advantageous for performance because it reduces latency and improves throughput of the current system bottleneck.

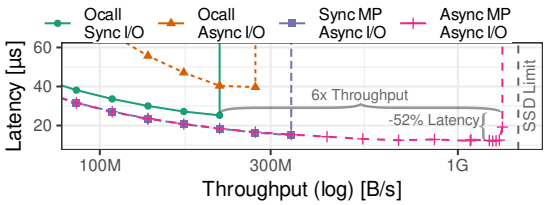

Figure 10: Performance improvements due to asynchronous message passing and I/O. Message passing outperforms both `OCall` settings. When combined with asynchronous I/O, SSD bandwidth is saturated.

To the best of our knowledge, we are the first to build such an asynchronous solution natively into an SGX-based DBMS prototype. Previous approaches either used synchronous message passing [30, 43, 49] or implemented message passing in a libOS while exposing only a synchronous interface to the DBMS [6, 12, 14, 40, 46, 52].

Performance improvements. To show the improvements of our asynchronous design in terms of latency and throughput, we compare the following four settings, each using all CPU cores of one CPU in our system: (1) `OCall` with synchronous I/O (previous): The group flush thread transitions out of the enclave, sends an SPDK write command, polls the completion queue, and returns into the enclave when it received the response. (2) `OCall` with asynchronous I/O (previous): The group flush thread transitions out of the enclave, sends an SPDK write command, reads the contents of the completion queue once, and returns into the enclave. (3) Synchronous message passing with asynchronous I/O (similar to related work, esp. [6]): The group-flush thread sends a write message to a write queue and polls the response queue inside the enclave. The I/O thread outside the enclave manages the enclaves' and SSDs' queues asynchronously. (4) Asynchronous message passing with asynchronous I/O (ours): The group flush thread sends a write message, continues other work, and regularly checks the response queue for write confirmations. The I/O thread outside the enclave manages the enclaves' and SSDs' queues asynchronously. The results are depicted in Figure 10.

With synchronous message passing, minimum latency improves by 40% over the previously best latency achieved by `OCall` with synchronous I/O. Maximum throughput improves by 46% over the previously best throughput achieved by `OCall` with asynchronous I/O. With asynchronous message passing, minimum latency improves by 18% further. But more importantly, and in line with the asynchronous SPDK speedup from Figure 7, maximum throughput increases by 6×. This is because, without the overhead of enclave transitions, the asynchronous processing pipeline can fully leverage SSD parallelism, and neither the I/O thread nor the in-enclave flush thread wastes cycles polling for responses. The remaining 6% gap to the maximum SSD throughput is caused by the space overhead of the encrypted WAL page header.

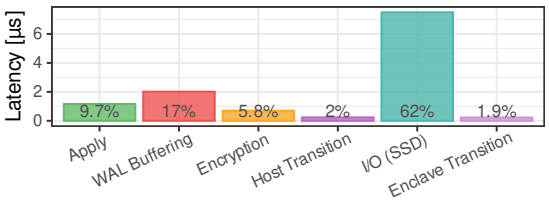

Figure 11: Final median latencies of write steps.

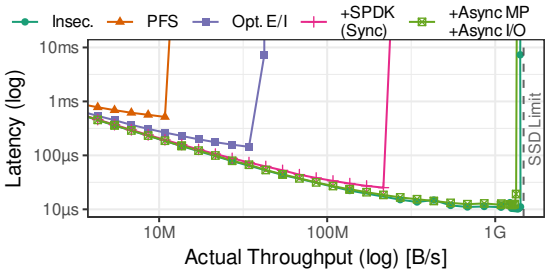

Figure 12: Overall latency-throughput plot comparing SGX PFS baseline, insecure baseline, and important (intermediate) optimized solutions presented in this paper.

Figure 11 shows the median latencies of the steps in a write, including our asynchronous message passing approach. Due to message passing, the latency of crossing the enclave boundary (Host- and Enclave Transition) is reduced from 6.3 µs to less than 0.5 µs. Due to the throughput improvement gained with asynchronous writes, the median buffer latency was reduced from 9 to 2 µs. With the latest optimizations, the main contributor to write latency is again the SSD write latency.

4.5 Overall Performance Improvements

Finally, we compare the performance of the WAL prototype, including all optimizations introduced in this section, with our baseline with SGX PFS and important intermediate optimization steps. The results are depicted in Figure 12. Overall, minimum latency is reduced by 97.7%, and maximum throughput improved by 114×. When compared to a baseline outside the enclave using SPDK as the I/O interface without encryption and integrity protection, minimum latency is only 18% higher, and maximum throughput is only 6% lower due to the encrypted WAL page header.

5 Discussion and Next Steps

In this section, we briefly discuss how our findings generalize beyond the WAL-based access pattern, how our findings generalize to other TEE technologies beyond SGX, and the issue of rollback protection in our optimized integrity protection scheme.

Building a secure storage engine. Besides the write-ahead log (WAL), another durability-critical component is the database system's storage engine – typically implemented as a buffer-managed page store. Many of our WAL optimizations also apply to page I/O: using a low-latency I/O stack such as SPDK, optimizing integrity mechanisms on the I/O path, employing optimized cryptographic primitives, and using an asynchronous enclave interface for I/O.

Compared to WAL I/O, buffer-pool page I/O differs in two ways. First, it largely uses random access. Hence, the buffer manager must provide integrity verification for arbitrary page reads and writes, for example, by maintaining a Merkle tree over pages. As discussed above, using a Merkle tree adds logarithmic metadata overhead compared to encryption alone. Second, unlike the WAL, page updates do not require immediate durability. This characteristic enables asynchronous, batched updates of the Merkle-tree metadata [10, 52]. It also enables the use of a second-level page cache in untrusted host memory that stores encrypted and integrity-protected pages, thereby reducing storage access latency [44]. We leave the full implementation and evaluation of a secure buffer manager using these optimizations for future work.

LSM-Trees. Many modern DBMS, especially those optimized for maximum write throughput, use a Log-structured Merge Tree (LSM-Tree) as their underlying storage data structure. Examples include LevelDB [20] and RocksDB [38]. To achieve durability, these systems also require a WAL, to which our design can be applied directly. Additionally, as with the page I/O of a buffer manager, our general optimizations also apply to LSM-Trees, namely the use of low-latency, asynchronous I/O with SPDK, an asynchronous enclave interface, and developing an optimized encryption and integrity protection scheme.

With their linear write access patterns, LSM-Trees already optimize for high-throughput disk access with low write amplification. This also helps when designing the encryption and integrity protection scheme. It seems to be most practical to extend the existing metadata stored in memory and in the SSTable file headers with cryptographic metadata. This naturally extends the LSM-Tree to an integrity-protected LSM-Tree, where in-memory metadata stores the integrity protection information needed to verify the SSTable headers, and the SSTable headers store the integrity protection information for the data pages they contain. Like the WAL pages in our prototype, the LSM-Tree's data pages can be encrypted with authenticated encryption to detect changes to these files. This approach was also followed by the SGX-based Key-Value Stores SPEICHER [14] and Pldb [43].

Portability to VM-based TEEs. Although our prototype targets Intel SGX, similar I/O challenges arise in VM-based TEEs such as Intel TDX [22], AMD SEV [29], and AWS Nitro Enclaves [7]. These TEEs protect an entire confidential virtual machine (CVM) against malicious infrastructure or service providers. However, storage devices (e.g., SSDs) remain outside the trust boundary. Consequently, CVMs still incur (1) the need to encrypt and integrity-protect data before writing it to persistent storage, and (2) overhead to traverse the protected I/O path. Accordingly, related work identified I/O as a core CVM performance issue [35].

We expect our techniques to translate to CVMs. Since the encryption and integrity protection mechanism is generic, it also provides secure WAL storage in the CVM setting. While current CVM deployments generally do not allow direct sharing of protected guest memory with storage devices, an SPDK-based datapath can still reduce overhead by running SPDK on the host (e.g., via vhost) [2] and using a corresponding virtio driver inside the CVM [3]. With file-system-backed storage, Linux block-layer mechanisms such as dm-crypt [1] and dm-verity [47] provide encryption and integrity, but this OS-based approach prevents DBMS-specific optimizations.

Rollback Protection. In this paper, the main contribution is to provide secure I/O for WAL purposes while removing as much

overhead as possible. While our prototype provides the most important security properties for the WAL, we deliberately left out particular guarantees, such as rollback protection of the enclave running the workload. When building a full transactional engine, our prototype could be extended with rollback protection based on prior work [8, 15, 36].

6 Related Work

Previous research on secure databases using Intel SGX has explored several directions.

SGX-native DBMS. Early works focused on developing specialized DBMS for the first, memory-restricted version of Intel SGX [51, 55], including studies on key-value stores [5, 14, 44] or on securing only parts of the DBMS within the SGX enclave [9, 42, 44].

Fast I/O for SGX enclaves. Another line of research addresses I/O performance in SGX enclaves, particularly for storage [12, 14, 46] and network access [6, 12, 46]. Key contributions include synchronous switchless enclave calls [12, 40, 49, 50, 53], and the integration of fast I/O primitives outside the enclave with library OSs [6, 14, 46]. However, these works primarily focus on generic applications and do not optimize DBMS performance by leveraging asynchronous I/O within enclaves.

DBMS integrity protection in enclaves. Other studies explore integrity protection for enclave-based DBMS, including for write-ahead logging. These approaches only simulated SGX [42], used the memory-restricted first version of SGX [14], or used a libOS [52], limiting their ability to leverage fast I/O primitives like SPDK to optimize performance.

7 Conclusion

Hardware-based Trusted Execution Environments like Intel SGX offer performance comparable to execution outside the enclave when data is in memory. However, storage access suffers performance degradation due to trust boundaries and the need for encryption and integrity protection. In this paper, we showed how to build an SGX-optimized write-ahead logging subsystem, targeting the most latency-sensitive storage operation in OLTP DBMSs. By co-designing security guarantees, leveraging fast user-space I/O, and implementing an asynchronous message-passing enclave interface, our prototype achieves a 114× throughput improvement and 97.7% reduction in latency compared to a naive implementation using the Protected File System library from the SGX SDK.

Acknowledgments

This work was supported by SAP SE, the grant "High-Performance Analytical & Secure Query Processing for Cloud Databases (HiPeAC)" by the National Research Center for Applied Cybersecurity ATHENE, the LOEWE initiative (Hesse, Germany) within the emergenCITY center, and the LOEWE Spitzenprofessur of the state of Hesse. We also thank hessian.AI and DFKI for their support.

References

[1] 2024. Dm-Crypt. *Wikipedia* (Dec. 2024). https://en.wikipedia.org/w/index.php?title=Dm-crypt&oldid=1261009941
[2] 2026. SPDK: Vhost Target. https://spdk.io/doc/vhost.html
[3] 2026. SPDK: Virtio Driver. https://spdk.io/doc/virtio.html
[4] Daniel J. Abadi. 2009. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Eng. Bull.* 32, 1 (2009), 3–12. https://www.academia.edu/download/48857081/DataEngBullMarch09CloudComputing.pdf#page=5
[5] Aghiles Ait Messaoud, Sonia Ben Mokhtar, and Anthony Simonet-Boulogne. 2024. Tee-Based Key-Value Stores: A Survey. *The VLDB Journal* 34, 1 (Dec. 2024), 10. doi:10.1007/s00778-024-00877-6
[6] Mansour Alharthi, Fan Sang, Dmitrii Kuvaiskii, Mona Vij, and Taesoo Kim. 2025. Rakis: Secure Fast I/O Primitives Across Trust Boundaries on Intel SGX. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1177–1193. doi:10.1145/3689031.3696090
[7] Amazon Web Services. 2025. What Is Nitro Enclaves? - AWS. https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html
[8] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Berkeley, CA, USA, 193–208. https://www.usenix.org/conference/osdi23/presentation/angel
[9] 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. doi:10.1145/3318464.3386141
[10] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, Nikhil Swamy, Alexander van Renen, and Min Xu. 2021. FastVer: Making Data Integrity a Commodity. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 89–101. doi:10.1145/3448016.3457312
[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58. doi:10.1145/1721654.1721672
[12] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Berkeley, CA, USA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov
[13] 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. doi:10.14778/3625054.3625058
[14] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores Using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Berkeley, CA, USA, 173–190. https://www.usenix.org/conference/fast19/presentation/bailleu
[15] Shanwei Cen and Bo Zhang. 2019. Trusted Time and Monotonic Counters with Intel® Software Guard Extensions Platform Services. https://www.intel.com/content/www/us/en/content-details/671564/trusted-time-and-monotonic-counters-with-intel-software-guard-extensions-platform-services.html
[16] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2024. Intel TDX Demystified: A Top-Down Approach. *ACM Comput. Surv.* 56, 9 (April 2024), 238:1–238:33. doi:10.1145/3652597
[17] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf
[18] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and Io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*. ACM, Haifa Israel, 120–127. doi:10.1145/3534056.3534945
[19] 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. doi:10.1145/3533737.3535098
[20] Google. 2026. Google/Leveldb. Google. https://github.com/google/leveldb
[21] Intel Corporation. 2016. Protected File System with Intel® Software Guard Extensions (Intel® SGX) on Microsoft Windows. https://cdrdv2-public.intel.com/671174/sgx-protected-file-system.pdf

[22] Intel Corporation. 2022. Intel® Trust Domain Extensions Whitepaper. https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf

[23] Intel Corporation. 2026. Intel(R) Software Guard Extensions Developer Guide. https://download.01.org/intel-sgx/sgx-linux/2.27/docs/Intel_SGX_Developer_Guide_for_Linux.pdf

[24] Intel Corporation. 2026. Intel(R) Software Guard Extensions Developer Reference for Linux* OS. https://download.01.org/intel-sgx/sgx-linux/2.27/docs/Intel_SGX_Developer_Reference_for_Linux_OS.pdf

[25] Intel Corporation. 2026. Intel® Cryptography Primitives Library. Intel® Corporation. https://github.com/intel/cryptography-primitives

[26] Wayne Jansen and Timothy Grance. 2011. *Guidelines on Security and Privacy in Public Cloud Computing* (0 ed.). Technical Report NIST SP 800-144. National Institute of Standards and Technology, Gaithersburg, MD. NIST SP 800–144 pages. doi:10.6028/NIST.SP.800-144

[27] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, and Carsten Binnig. 2025. A Wake-Up Call for Kernel-Bypass on Modern Hardware. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, 1–5. doi:10.1145/3736227.3736235

[28] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. 2025. Supporting Intel(r) SGX on Multi-Package Platforms. doi:10.48550/arXiv.2507.08190 arXiv:2507.08190 [cs]

[29] David Kaplan. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/solution-briefs/amd-secure-encrypted-virtualization-solution-brief.pdf

[30] 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. doi:10.1145/3302424.3303951

[31] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/2933349.2933352

[32] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. 2022. Frequency Throttling Side-Channel Attack. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1977–1991. doi:10.1145/3548606.3560682

[33] Adrian Lutsch, Muhammad El-Hindi, Matthias Heinrich, Daniel Ritter, Zsolt István, and Carsten Binnig. 2025. Benchmarking Analytical Query Processing in Intel SGXv2. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*, Alkis Simitsis, Bettina Kemme, Anna Queralt, Oscar Romero, and Petar Jovanovic (Eds.). OpenProceedings.org, Konstanz, Germany, 516–528. doi:10.48786/EDBT.2025.41

[34] Adrian Lutsch, Muhammad El-Hindi, Zsolt István, and Carsten Binnig. 2025. Towards High-performance and Trusted Cloud DBMSs. *Datenbank-Spektrum* 25, 1 (March 2025), 39–50. doi:10.1007/s13222-025-00495-8

[35] Adrian Lutsch, Christian Franck, Muhammad El-Hindi, Zsolt István, and Carsten Binnig. 2025. An Analysis of AWS Nitro Enclaves for Database Workloads. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3736227.3736234

[36] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. https://eprint.iacr.org/2017/048

[37] 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. doi:10.1145/2487726.2488368

[38] Meta Platforms, Inc. [n. d.]. RocksDB | A Persistent Key-Value Store. http://rocksdb.org/

[39] Lam-Duy Nguyen, Adnan Alhomssi, Tobias Ziegler, and Viktor Leis. 2025. Moving on From Group Commit: Autonomous Commit Enables High Throughput and Low Latency on NVMe SSDs. *Proc. ACM Manag. Data* 3, 3 (June 2025), 191:1–191:24. doi:10.1145/3725328

[40] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 238–253. doi:10.1145/3064176.3064219

[41] 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, San Francisco, CA, USA, 693–702. doi:10.1109/CloudCom.2010.66

[42] 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. doi:10.1109/SP.2018.00025

[43] Chenkai Shen and Lei Fan. 2023. Pldb: Protecting LSM-based Key-Value Store Using Trusted Execution Environment. In *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, San Francisco, CA, USA, 762–771. doi:10.1109/TrustCom60117.2023.00111

[44] 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. doi:10.14778/3447689.3447705

[45] Hassan Takabi, James B.D. Joshi, and Gail-Joon Ahn. 2010. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security & Privacy* 8, 6 (Nov. 2010), 24–31. doi:10.1109/MSP.2010.186

[46] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. 2021. Rkt-Io: A Direct I/O Stack for Shielded Execution. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 490–506. doi:10.1145/3447786.3456255

[47] The kernel development community. [n. d.]. Dm-Verity — The Linux Kernel Documentation. https://docs.kernel.org/admin-guide/device-mapper/verity.html

[48] The kernel development community. 2020. Io_uring(7) - Linux Manual Page. https://man7.org/linux/man-pages/man7/io_uring.7.html

[49] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*. Association for Computing Machinery, New York, NY, USA, 22–27. doi:10.1145/3268935.3268942

[50] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. 2017. SGXKernel: A Library Operating System Optimized for Intel SGX. In *Proceedings of the Computing Frontiers Conference (CF'17)*. Association for Computing Machinery, New York, NY, USA, 35–44. doi:10.1145/3075564.3075572

[51] 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. doi:10.2478/popets-2019-0052

[52] Haitao Wang, Bingsheng Zhang, and Kui Ren. 2025. EncDB-FR^{3}3: An Encrypted Database With Fault Recovery and Rollback Resistance. *IEEE Transactions on Dependable and Secure Computing* 22, 6 (Nov. 2025), 6147–6162. doi:10.1109/TDSC.2025.3580232

[53] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 81–93. doi:10.1145/3079856.3080208

[54] Pan Yang, Naixue Xiong, and Jingli Ren. 2020. Data Security and Privacy Protection for Cloud Storage: A Survey. *IEEE Access* 8 (2020), 131723–131740. doi:10.1109/ACCESS.2020.3009876

[55] Xinying Yang, Cong Yue, Wenhui Zhang, Yang Liu, Beng Chin Ooi, and Jianjun Chen. 2024. SecuDB: An In-Enclave Privacy-Preserving and Tamper-Resistant Relational Database. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3906–3919. doi:10.14778/3685800.3685815

A Security Discussion

Assumptions. In line with our threat model, we make the following assumptions about the adversary: (1) The adversary controls the system running the enclave that writes the WAL, but it cannot compromise SGX security. Thus, the SGX enclave's memory contents are not accessible to the adversary and cannot be manipulated by it. (2) The adversary is not a client communicating with the DBMS running inside the enclave. (3) The encryption key is only available inside the enclave. The adversary has no access to the encryption key. (4) Like described in our prototype, only encrypted data leaves the enclave. (5) The adversary cannot break AES-GCM. This means it cannot decrypt encrypted data, nor can it forge encrypted data that passes the integrity check using the AES-GCM Message Authentication Code (MAC). (6) The goal of the adversary is to remain undetected. It will refrain from detectable attacks and attacks that will be detected with high probability. The goal of the integrity protection techniques used in our WAL prototype is to detect changes made by an adversary, not to prevent them. (7) We are not interested in denial-of-service attacks, such as an adversary

halting the DBMS's execution or deleting the entire WAL. These are trivial to execute for every adversary with control over the OS or hardware and cannot be prevented with TEEs.

Potential Attacks and Protections. Next, we go over attacks that an adversary could try and describe how our encryption and integrity protection scheme prevents them, or at least enables the DBMS inside the enclave to detect them. **Adversary tries to roll back parts of the WAL.** The adversary cannot roll back random pages inside the WAL because each page with its page ID is only written exactly once. Thus, there is no old version of the page to roll back to. The adversary can rollback the entire WAL by truncating the log at the end. Like SGX PFS, we do not protect against this global rollback attack. A Trusted Monotonic Counter (TMC) to protect against this could be added in the future [8, 14, 36, 42, 43]. **Adversary tries to replace a WAL page with a page created by a fork.** We see the prevention of forking attacks as an orthogonal issue. For example, forking attacks can be prevented using a centralized trusted key management service [42]. **Adversary tries to read WAL data.** According to our design, only encrypted WAL pages leave the enclave. The adversary has no access to the decryption key and cannot break AES-GCM. Thus, the adversary cannot decrypt and read the WAL data. **Adversary tries to add entries to WAL.** Since the adversary cannot forge WAL pages that pass the AES-GCM integrity check, all the adversary's attempts to add entries to the WAL will be detected and thereby fail.

Adversary tries to change entries in WAL. As introduced above, the adversary cannot read the WAL to attempt controlled changes. If it randomly changes parts of the WAL, the message authentication code of the authenticated encryption will no longer match, revealing the attack. **Adversary tries to reorder WAL entries.** The adversary cannot reorder WAL entries within an encrypted WAL page, as the AES-GCM integrity check would fail thereafter. The adversary can reorder encrypted WAL pages on disk, and this will not be detected by checking the MAC, because all pages were correctly encrypted with the key inside the enclave. However, the enclave-based DBMS can restore the order using WAL page IDs and timestamps within the encrypted WAL entries. The adversary cannot change the WAL page IDs because it cannot forge WAL pages that pass the AES-GCM integrity check. **Adversary tries to remove entries from inside the WAL.** The adversary can delete encrypted WAL pages on disk. However, the enclave-based DBMS can detect missing pages by the absence of WAL page IDs. Thus, the attack is revealed. The adversary cannot remove individual entries within WAL pages, just as it cannot change WAL entries, as described above. **Adversary tries to duplicate WAL entries.** The adversary can duplicate whole WAL pages on disk. However, the enclave-based DBMS can deduplicate WAL pages by their page IDs, thereby revealing the attack and preventing it. The adversary cannot duplicate individual WAL entries, just as it cannot change WAL entries, as described above.