SHIELD: Encrypting Persistent Data of LSM-KVS from Monolithic to Disaggregated Storage

Viraj Thakkar[1]*, Dongha Kim[1]*, Yingchun Lai[2], Hokeun Kim[1], Zhichao Cao[1]
[1]Arizona State University, [2]Apache/Pegasus

Abstract

Log-Structured Merge-tree-based Key-Value Stores (LSM-KVS) are widely used to support modern, high-performance, data-intensive applications. In recent years, with the trend of deploying and optimizing LSM-KVS from monolith to disaggregated storage (DS) setups, the confidentiality of LSM-KVS persistent data (e.g., WAL and SST files) is vulnerable to unauthorized access from insiders and external attackers and must be protected using encryption. Existing solutions lack a high-performance design for encryption in LSM-KVS, often focusing on in-memory data protection with overheads of 3.4-15× and lack the scalability and flexibility considerations required in DS deployments.

This paper proposes two novel designs to address the challenges of providing robust security for persistent components of LSM-KVS while maintaining high performance in both monolith and DS deployments - a simple and effective instance-level design suitable for monolithic LSM-KVS deployments, and **SHIELD**, a design that embeds encryption into LSM-KVS components for minimal overhead in both monolithic and DS deployment. We achieve our objective through three contributions: (1) A fine-grained integration of encryption into LSM-KVS write path to minimize performance overhead from exposure-limiting practices like using unique encryption keys per file and regularly re-encrypting using new encryption keys during compaction, (2) Mitigating performance degradation caused by recurring encryption of Write-Ahead Log (WAL) writes by using a buffering solution and (3) Extending confidentiality guarantees to DS by designing a metadata-enabled encryption-key-sharing mechanism and a secure local cache for high scalability and flexibility. We implement both designs on RocksDB, evaluating them in monolithic and DS setups while showcasing an overhead of 0-32% for the instance-level design and 0-36% for SHIELD.

1 Introduction

Log-Structured Merge-tree-based Key-Value Stores (LSM-KVS) have become a foundational component in modern applications, such as machine learning, stream processing, and artificial intelligence for supporting high-throughput workloads [26, 68]. LSM-KVS enhances write performance by appending new data to storage rather than applying in-place updates to the on-storage files. Incoming writes are logged sequentially in a Write-Ahead Log (WAL) to ensure crash consistency and subsequently buffered in an in-memory self-sorting structure called memtable. When full, the memtable is persisted into an immutable Sorted String Table (SST) file. This out-of-place update model, combined with periodic background compactions that merge and reorganize SST files, provides a high write throughput and moderate read throughput [34].

Traditionally, LSM-KVS systems were designed for monolithic, shared-nothing infrastructures [34, 35]. Scaling these systems involved deploying multiple LSM-KVS instances across servers, with sharding used to distribute data [26, 38]. Recently, disaggregated data centers (DDCs), which decouple resources into compute, memory, and storage heavy servers [39, 92] have introduced new opportunities for LSM-KVS. Recent research including NovaLSM [44], HailStorm [22], CaaS-LSM [93], Disaggregated RocksDB and HBase [25, 36] has proposed optimizations such as offloading compaction tasks, leveraging tiered storage, and light-weight read-only instance mechanisms for higher write and read performance to leverage the decoupled LSM-KVS architecture in a disaggregated storage (DS) deployment. Such work relies upon sharing client data stored in the WAL and SST files at disaggregated storage across multiple servers at different server clusters, which in DS may have other users and applications.

The attack surface has expanded as deployments have moved from monolith to disaggregated deployments. To safeguard client data in these files from unauthorized access, data confidentiality must be ensured using strong encryption algorithms [65] like AES [31] or ChaCha [21]. Specifically, this paper focuses on three threats: (1) unauthorized users with legitimate access to the server, (2) external attackers gaining filesystem access to disaggregated storage cluster, and (3) untrusted storage providers who may have physical or backdoor access to the storage media.

Encryption [31, 49] involves different transformative processes of performing operations on data using a secret Data Encryption Key (DEK) that transforms data to be written to file. Only holders of this secret DEK can access this data, which is a robust method to ensure protection from the aforementioned threats. However, the process also introduces overhead in repetitive, expensive memory allocation calls, secure DEK management, and optional integrity checks. Additionally, to minimize data exposure risk in situations of DEK leaks, it is recommended [65] to implement DEK-handling practices such as the use of unique DEKs for every file and to perform frequent re-encryption of data using DEK-rotation. All these processes introduce additional overhead and latency, which is highly undesirable in high-performance systems like LSM-KVS.

Existing LSM-KVS Encryption solutions, such as SPEICHER [16] and PLDB [84], are designed primarily for providing strong data-in-use protection in monolithic, shared-nothing infrastructures. However, they fall short in addressing the specific needs of data-at-rest protection, which requires mechanisms like DEK rotation and unique DEKs per file to enhance security. Furthermore, these solutions do not propose DEK-sharing methods to adapt to different flexible deployments of disaggregated storage, as shown in Table 1. To ensure data in use is protected, these systems perform encryption using high-overhead [10] hardware-based environments called

Table 1: Comparison of Our Designs with Existing Work.

	Support for Disaggregated Storage	Focus On Data-at-Rest Protection	Focus on Data-in-Use Protection	DEK Handling Practices	Throughput Degradation
No-Encryption	✓	N/A	N/A	N/A	N/A
Existing LSM-KVS Encryption Soln.	×	×	✓	×	340–1,500%
Instance-level Encryption	×	✓	×	×	0–32%
SHIELD	✓	✓	×	✓	0–36%

enclaves (like Intel SGX [29]), which gives them additional data confidentiality guarantees. However, this approach incurs significant performance overheads, 6.7-13× (SPEICHER) and 3.4-9.4× (PLDB) - with the high-overhead enclaves as the primary bottleneck, limiting existing research from finding other potential bottlenecks that may arise from integrating encryption in LSM-KVS. Moreover, existing solutions use a single DEK without supporting DEK rotation, increasing the risk associated with DEK compromise. Addressing these gaps is critical for balancing the distinct requirements of data at rest and data in use while ensuring robust performance and adaptability across deployment architectures.

Our research focuses on providing data confidentiality for persistent files of LSM-KVS while maintaining high performance for flexible environments from monolith to DS. To achieve this, careful design considerations are required. DEK-handling practices must especially be implemented for DS deployments where other users utilize the server, and taking it offline to protect compromised data is not an option. Removing high-overhead in-memory protection also exposes a significant overhead when performing atomic encryption in the WAL, which must be managed. Further, a flexible and highly available approach must be presented to support deployments and optimizations in DS, including offloaded compaction or read-only instances.

To achieve our objective, three key issues must be resolved. **First,** while simply encrypting data before persistence is possible, it does not solve challenges with integrating DEK practices. How do we utilize the LSM-KVS design and embed encryption while minimizing the overhead from unique DEKs per file and DEK rotation? The solution must be hardware-independent and provide secure generation and distribution of DEKs in DS. **Second,** encrypting each WAL-write (small writes) necessitates repetitive high-overhead encryption initialization processes like memory allocation and introduces considerable performance degradation. How to mitigate this overhead to minimize the impact of performance from encrypting WAL-writes? **Third,** as DS grows in popularity, scalable DEK distribution and management systems must be integrated within LSM-KVS deployments. How to create a highly available DEK-sharing design with minimal latency for DS deployments?

To overcome these challenges, we propose two novel LSM-KVS designs. The first is a simpler instance-level design that is a non-intrusive solution designed for monolithic and distributed environments where more control over the server is present. This design (EncFS) is a unified I/O engine that overloads the I/O function of LSM-KVS with encryption and decryption support. Based on EncFS, we further propose a comprehensive solution **SHIELD**, **SH**ield **I**ntegrates **E**ncryption to **L**SM-KVS for **D**DCs, that can be used for both monolith and distributed deployments. However, it has been specifically designed to tackle the challenges from DS deployments of LSM-KVS using three main novel designs.

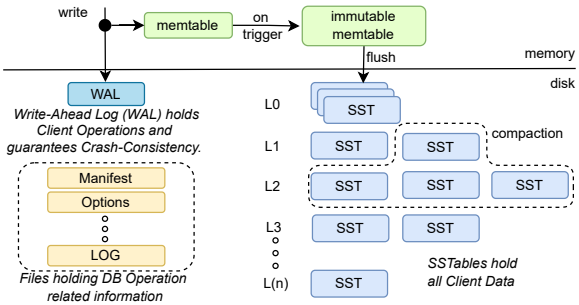


Figure 1: Different Components involved in Writes for KV-Pairs in LSM-KVS.

- **Embedding encryption** within the write path of LSM-KVS. SHIELD uses a new DEK for every new file created. This integration with compaction allows it to simultaneously provide periodic DEK rotation.
- **WAL performance optimization** uses an application-managed buffer and proposes a data-persistency and performance trade-off. This buffer defers encryption and amortizes the cost of encryption over several writes.
- **Flexibility for DS** by designing our solution to be compatible with certain pre-existing encryption key distribution services (KDS) and integrating DEK management into the metadata of LSM-KVS files. Other servers can use the embedded DEK identifier to retrieve DEK from the KDS, which is responsible for authorization authentication.

We implement a prototype for EncFS and SHIELD using RocksDB and open-source it on GitHub[1]. We evaluate both EncFS and SHIELD in various micro and macro benchmarks to cover specific and real-world workloads while also performing a sensitivity analysis of different tunable parameters in LSM-KVS. We also evaluate SHIELD in disaggregated storage and offloaded compaction deployments, and present our results. We evaluate all major modules of SHIELD using db_bench [37] and YCSB [27]. EncFS shows a performance regression of 0-32% and SHIELD of 0-36% compared to an unencrypted implementation of RocksDB. In the DS deployments, SHIELD has 15% throughput regression compared with RocksDB deployed in the same environment.

2 Background

2.1 LSM-KVS

LSM-KVS is widely used in today's infrastructure for storing unstructured data and is optimized for write-heavy workloads [35]. To achieve a high write throughput, LSM-KVS leverages an out-of-place update strategy and append-only data files [68]. The overall architecture of LSM-KVS is shown in Figure 1. Key-Value Pairs

[1]https://github.com/asu-idi/SHIELD

(KV-pairs) are first written to an on-disk Write-Ahead Log (WAL) to guarantee crash recovery and then buffered in a self-sorting in-memory structure (e.g., a skiplist) called the *memtable*. Once the memtable is full, *flush* operation persists the memtable to the storage system as an immutable Sorted-String Table (SST) file. SST files are maintained at multiple levels, and they do not have key-range overlaps in the same level (excluding level-0). Once a level triggers a pre-set *compaction* condition, one SST file at the level is selected to be merged with SST files in the next level with key-range overlaps to eliminate duplicate or obsolete KV-pairs for quicker read lookups and reduced storage overhead [26, 34].

The data persistency of LSM-KVS relies on three types of on-storage files: metadata files (e.g., Manifest file in RocksDB), WAL files, and SST files. The WAL and SST files are the only files holding any user data. **Write-Ahead Logs.** In popular LSM-KVS like RocksDB [80] and LevelDB [5], each KV-pair is appended to the WAL via a buffered I/O interface. This interface leverages an OS-managed in-memory buffer that aggregates multiple write operations before flushing them to disk. This reduces the frequency of costly disk I/O operations, improving performance at the cost of data persistence [28, 58, 71]. Buffered I/O ensures recovery from process crashes; however, KV-pairs in the OS buffer (not yet flushed to disk) are lost during a system crash. **SST Files.** KV-pairs are written to SST files using either direct I/O or buffered I/O during flush or compaction, depending on the implementation. These KV-pairs are sorted and stored as size-capped blocks (e.g., 4KB in RocksDB), enabling efficient lookups and range queries [79, 80]. When KV-pairs are requested, the block is the basic I/O unit to be read from the storage system to the LSM-KVS in-memory *block cache* for further binary search. SST files are organized across levels, with each successive level containing files sized capped by approximately N times (e.g., configured to 10 in RocksDB and LevelDB) larger than those in the previous level (called *fanout*).

Current LSM-KVS implementations often store unencrypted (plaintext) data within WAL and SST files, exposing sensitive client data to server compromises. Data confidentiality should be ensured using encryption to ensure sensitive information remains protected from unauthorized access. Encryption involves using a secret Data Encryption Key (DEK) to transform plaintext into ciphertext, ensuring that unauthorized users cannot access the data [31, 49]. This process can be applied at various levels - the disk, filesystem, and application - each with its own trade-offs [9, 67, 86] in terms of access control and performance. Disk or filesystem-level encryption generally provides broad access control, allowing all system users to access data in plaintext. Meanwhile, application-level encryption offers more granular control over who can access plaintext from specific data files, making them more appropriate solutions for multi-user systems. However, while the granularity of application-level encryption is desirable, it introduces additional overhead [13, 88] for the management, generation, storage, retrieval, and the associated processing latency for DEKs. The encryption and decryption processes also carry their own overheads, including memory allocation, computational expenses, secure key management, and optional integrity checks.

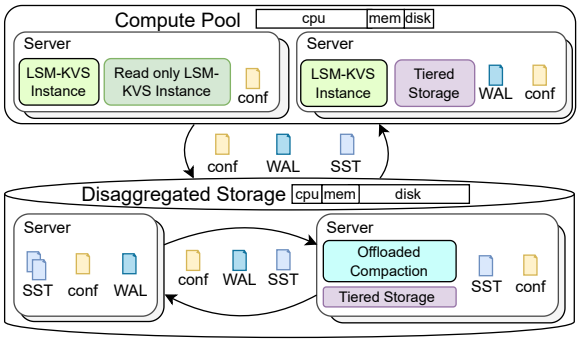


Figure 2: Different LSM-KVS Deployments in DS.

2.2 LSM-KVS in Disaggregated Storage

Disaggregated Data Centers (DDCs) enable flexible resource management and dynamic resource scaling by decoupling resources into compute, memory, and storage heavy server pools. These servers are interconnected through high-speed networks [39, 92], and resource provision can be independently scaled to application demand [14, 57, 83, 96]. Recently, disaggregated setups have gained prominence; in particular, Disaggregated Storage (DS) has been deployed by companies in large-scale systems like Google's GFS [40] and Meta's Tectonic [69].

The shift towards DDCs has changed the LSM-KVS initially designed for monolithic and shared-nothing infrastructures [36]. In the past, hundreds of LSM-KVS instances would be deployed on tens of monolithic servers for distributed data management using data sharding [35, 38, 81]. Each LSM-KVS instance uses the same compute, memory, and storage resources and may encounter resource contention with other instances in the same server. To address the limitations of monolithic deployments and be adapted to DDCs, LSM-KVS architecture has since evolved to leverage disaggregated resources better. Different components such as persistent storage, indexing, and compaction are decoupled and re-distributed to specialized resource pools [22, 44, 72, 91, 93] to improve the scalability, resource utilization, and reduce resource wasting [25, 36].

Figure 2 shows various deployment strategies and optimizations for LSM-KVS in a DS setup. SST files and I/O-heavy tasks like compaction are offloaded to the disaggregated storage cluster to reduce I/O traffic [22, 36, 44, 93]. Tiered storage optimizes throughput by initially storing WALs locally before moving them to disaggregated storage for long-term data persistence [98]. The data-sharing capabilities of disaggregated storage enable optimizations like compaction offloading and on-demand read-only instance launching via WAL and SST sharing [93, 98]. During intensive read workloads, multiple read-only instances can be launched in the compute pool to serve queries, while offloaded compaction can run on any available node by accessing shared SST files and storing results back in disaggregated storage [93].

In such setups, if WAL and SST files are not encrypted, the attack surface expands to multiple servers since multiple servers can access data files belonging to a single LSM-KVS instance. This creates vulnerabilities from unauthorized users and applications accessing shared disaggregated storage, necessitating data confidentiality throughout the data lifecycle—in transit, in memory, and at rest. Solutions employing Transport Layer Security [78, 90] and Trusted

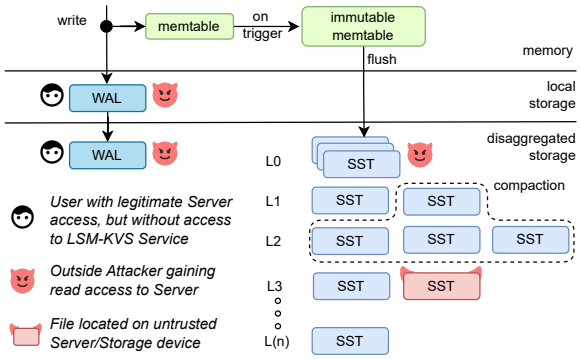


Figure 3: Security Threats Considered in the Threat Model.

Execution Environments [29, 30, 56] provide data confidentiality in transit and in memory respectively [63, 77]. For data at rest, confidentiality is provided by encrypting data before persisting it.

For at-rest data (e.g., SST files and WAL), a DS setup presents a larger attack surface with other potentially compromised users and applications using the same physical resources of the server. In such settings, DEK compromises are more likely, and to limit and mitigate such compromises, the re-encryption of data with a new DEK, DEK-rotation, is required. Such action necessitates large I/Os for re-encrypting all data and persisting it again, deleting older data, which can reduce the performance and availability of LSM-KVS. While this is acceptable in controlled environments, such an approach will affect the availability of other applications in a disaggregated setup. To avoid such situations, two DEK-handling practices [65] are used: (1) unique DEK for every file to limit data exposure while having to re-encrypt only a single file in DEK compromises and (2) periodic DEK rotations to limit the time frame in which a compromised DEK is effective. Further, these practices facilitate fine-grained access control, allowing implementations where the compaction server is only given access to specific files.

3 Motivation and Challenges

LSM-KVS often do not encrypt stored files holding client data (WAL and SST), making them susceptible to compromise in DS. On-disk confidentiality of these files is essential using encryption with careful consideration of flexibility for different deployments of LSM-KVS from monolith to disaggregated infrastructures and effective methods to implement DEK-management practices to have minimal impact on performance. This section covers the possible threats of leaving data insecure, followed by our motivation and challenges for this research.

3.1 Threat Model

We utilize threat models from prior research on LSM-KVS security (e.g., PLDB [84], SPEICHER [16]) and adapt these models to account for the demands of DS. Specifically, we consider the following threats at different system components.

- ***Unauthorized access*** by the users with legitimate access to the server but no read or write access to any LSM-KVS component.
- ***External attackers*** exploiting system vulnerabilities to gain unauthorized access to the local file system or any DS system.
- ***Untrusted storage medium*** where any storage devices within DS are untrusted, thus, may leak sensitive information and be tampered with.
- ***Data Encryption Key (DEK) compromise*** where a strong adversary gains access to the DEK and potential data exposure risk must be minimized.

This paper focuses on confidentiality for the persistent data components (i.e., SST files, WAL, and Manifest) of LSM-KVS, as shown in Figure 3. Memory and network-based attacks, while critical, introduce tangential challenges (e.g., buffer overflows and replay attacks) [19, 97] and are left for future work. We assume that the security primitives and the security infrastructure remain secure, including the KDS [52, 66] which is the DEK [42, 43] issuing and management service. Compromised security primitives could bypass authentication, which is beyond our model's scope.

3.2 Motivation

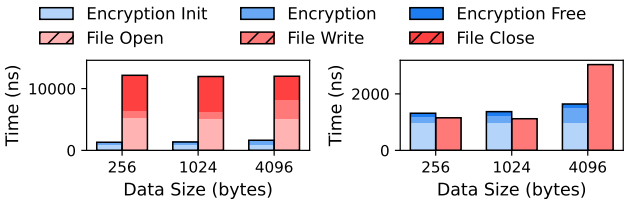
A secure LSM-KVS must ensure the confidentiality of client data in SST, WAL, and metadata (Manifest) files across both monolithic and disaggregated storage (DS) deployments, protecting against threats outlined in our model. This section highlights the limitations of existing solutions and our motivation for this research.

Limitations of Current Solutions. Current encryption implementations for LSM-KVS, like SPEICHER [16] and PLDB [84], focus on in-memory data protection using hardware-based trusted execution environments called enclaves, which incur significant performance degradation (6.7–13× for SPEICHER and 3.4–9.4× for PLDB). These solutions extend encryption to data at rest but use a single DEK for all files, centralizing cryptographic risk and increasing vulnerability to DEK compromise. Moreover, they do not implement DEK-handling practices of having unique DEKs per file or regular DEK-rotation that enhance security by limiting compromised keys' scope and temporal exposure. The enclave-based approach also shifts focus away from exploring performance bottlenecks introduced by encryption. Neither solution supports DS deployments, as they lack integration with KDS for securely distributing DEKs across multiple servers in DS.

Benefits of Embedding Encryption in LSM-KVS. Focusing on the persistent component of LSM-KVS, a layer of encryption implemented between the file I/O engine and LSM-KVS can ensure the confidentiality of files (i.e., SST, WAL, and Manifest files). Further security enhancements can be achieved with additional DEK-handling practices of unique DEKs per file, which isolate the impact of a DEK compromise on individual files, and regular DEK rotation, which minimizes the duration any single key is in use and reduces long-term exposure risks as discussed in Section 2.2. Additionally, integration with a KDS for secure DEK distribution across servers enables a highly available and scalable deployment of LSM-KVS in different environments. These practices must be integrated to provide stronger data confidentiality while supporting more robust response mechanisms to DEK compromises, making the system more resilient and secure in monolithic and DS settings.

Minimizing Encryption Overhead for WAL-Writes. Data encryption introduces additional overhead [46] from the setup of

[2]Averaged 100 iterations on Intel i9-13900 CPU, with an NVMe SSD, using OpenSSL v3.0.2 (Encryption) and C++17 stdio (File I/O) libraries.



(a) Encryption vs File Operation. (b) Encryption vs File Write.

Figure 4: Analysis of Encryption Costs and Writes.[2]

Table 2: Impact of Encryption for WAL-Writes.

	Throughput	Difference (%)
No Encryption	291966	
Encrypted SST	280353	-3.9%
Encrypted All (SST & WAL)	196021	-32.8%

necessary components like memory allocation and initialization vectors, the process of encrypting the data using algorithms such as AES [31] or ChaCha [21], and the cleanup where resources like memory are released. As shown in Figure 4a, the cost of encryption is significantly lower (approx. 9×) than writing the same amount of data to a file. However, unlike file initialization—where a pointer can be held in memory and reused, encryption initialization must be performed repeatedly for separate encryption. The encryption overhead is amortized for large-sized write operations; however, for smaller writes, the effect is more pronounced.

In LSM-KVS, SST files are created via background operations, and Manifest is infrequently appended. Conversely, foreground WAL writes are highly frequent and synchronous, and if KV-pairs are smaller in size, the encryption overhead of a single KV-pair is very significant compared to the duration of writes, as shown in Figure 4b. The same effect is much less significant for larger background writes from flush and compactions, where data is asynchronously persisted in large batches [32], allowing the encryption cost to be amortized over the batch. We demonstrate the effect of encryption using Table 2 where adding WAL encryption reveals an approx. 32% performance degradation. However, leaving the WAL in plaintext exposes data to potential compromise, and disabling the WAL will sacrifice crash consistency, making neither approach viable.

Flexibility for Different Deployments. Section 2.2 discusses different deployments for LSM-KVS. In a distributed system, sharding is performed to coordinate hundreds of LSM-KVS instances; and in DS, components of LSM-KVS (like offloaded compaction, read-only instances) can be present on different servers. With the implementation of security measures like unique DEKs per file and regular DEK rotation, implementing a secure KDS is necessary for all servers to request new and existing DEKs. However, if multiple instances on the same server need access to the same DEK, it is impractical to introduce network latency costs for repetitive DEKs requests from the KDS that may be on a different server.

3.3 Research Objectives and Challenges

In this paper, we focus on encrypting the persistent component of LSM-KVS to ensure data confidentiality. We explore the integration of encryption and how DEK-handling practices become more important and suitable as we move from monolith to DS deployments of LSM-KVS.

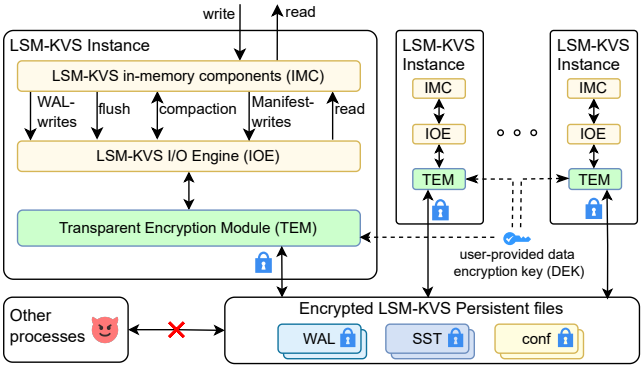


Figure 5: Design Overview for Instance-level Encryption.

However, there are three main challenges to achieving the objectives. *First*, we need to **integrate encryption into LSM-KVS** for both monolith and disaggregated storage (DS) deployments. How should the ideal encryption integrations be chosen in both monolith and DS systems as an additional layer or by embedding it within the LSM-KVS for implementation with minimal overhead while implementing DEK-handling practices? *Second*, implementing DEK-handling practices requires additional processes, including distributing new DEKs per file, re-encrypting data when DEKs are rotated periodically, communicating with a KDS, and providing a granular access control mechanism. Each process introduces additional latency, undesirable for high-performance systems like LSM-KVS. Careful considerations must be made while performing this integration. *Third*, encryption initialization is an expensive process that needs to be done over and over for every write operation - doing it for every WAL-Write is an expensive process that needs to be managed carefully. How do we design a solution that **manages the WAL encryption bottleneck** to minimize performance impact while still maintaining strong encryption guarantees and addressing the outlined threat model? *Finally*, with **flexible deployments** for LSM-KVS in distributed systems with sharding or DS with offloaded compaction and read-only instances, client data holding files (SST and WAL) are required on multiple instances and servers. How to effectively coordinate DEKs that are used to encrypt these files while avoiding repetitive DEK requests for LSM-KVS instances on the same server?

We note the distinct differences in server control for monolith designs with complete control, and how it has evolved to having lesser control over server access control in DS. To address the above challenges, we propose two designs. Our designs propose solutions apt for each setting. Section 4 proposes a design for Monolith deployments of LSM-KVS, and Section 5 proposes SHIELD, a solution designed for DS deployments of LSM-KVS with backward compatibility to support monolith.

4 Instance-level Encryption for Monolithic LSM-KVS

As discussed in Section 2.1, all LSM-KVS data is stored in different file types. By intercepting I/O operations at the engine layer and applying encryption or decryption before persistence, we can achieve transparent data protection (i.e., the core LSM-KVS codebase remains unchanged and unaware of encryption/decryption). This

design is particularly effective in controlled environments, such as monolithic or distributed deployments [35, 38, 81], where the LSM-KVS instances are the only applications on the servers.

4.1 Design Overview

To enable transparent encryption, all file I/O, such as WAL, SST, Manifest, and configuration file writes/reads, is intercepted. Data is encrypted before being stored via the underlying file system (e.g., EXT4 or XFS), and decrypted on demand during access. A user-provided DEK, supplied at LSM-KVS startup, is kept solely in memory to ensure low-latency DEK access for encryption/decryption and restrict DEK access to the active LSM-KVS session. In distributed environments, KDS like Kerberos [66] or Macaroons [23] can securely supply the DEK to all LSM-KVS instances.

The LSM-KVS core is agnostic to the encryption layer, requiring modifications only for systems using direct I/O (e.g., RocksDB) to ensure block alignment. Existing LSM-KVS interfaces supporting block alignment can address this need, maintaining compatibility with direct I/O. As a result, LSM-KVS operations like WAL writes, compactions, and flushes remain unchanged, with encryption uniformly applied before persisting data. This design ensures data confidentiality and safegaurds sensitive information from unauthorized access with minimal impact to LSM-KVS performance.

4.2 Tradeoff Discussion

The design that encrypts all persisted data using a single DEK is simple and transparent as it avoids complexities and added overheads of managing a unique DEK for every file. However, it comes with limitations and trade-offs for employing DEK-handling practices such as distinct DEKs per file, strong access control, and regular DEK rotation (detailed in Section 2.2).

The encryption for every write I/O relies on the I/O calls rather than the write path of LSM-KVS. The indiscriminate encryption disallows using unique DEKs for different files and forces using a single DEK. With a single DEK, any user with DEK access can retrieve data from any file. Additionally, if a DEK compromise occurs, the entire LSM-KVS data becomes vulnerable until a new secure DEK is circulated and all files are re-encrypted. This is a time-intensive process, increasing the system's exposure to potential attacks during the re-encryption window. Furthermore, re-encrypting all files is a large-scale operation that is I/O-intensive and impacts system availability. These trade-offs between simplicity and limitations in security and availability necessitate careful deliberation.

4.3 Considerations for DS Implementation

This instance-level encryption design is effective in controlled environments where the LSM-KVS is the only application (such as monolithic or distributed setups) and can be extended to a disaggregated storage (DS) setup by leveraging a KDS to securely distribute the DEK to different servers. However, if the same physical servers are shared among multiple applications and users, there is a significantly higher risk of unauthorized access to data or external attackers who might gain system access through other users. In such scenarios, implementation of DEK-handling practices, such as unique DEKs per file and DEK rotation, is important to limit data exposure in case of DEK compromise or a quick response.

The instance-level design is effective in controlled environments. However, it does not scale well for modern DS deployments where availability and security must not be compromised. A more robust solution that addresses the challenges of LSM-KVS in DS environments, such as more granular key management, non-disruptive DEK rotation, and effective access control, is essential.

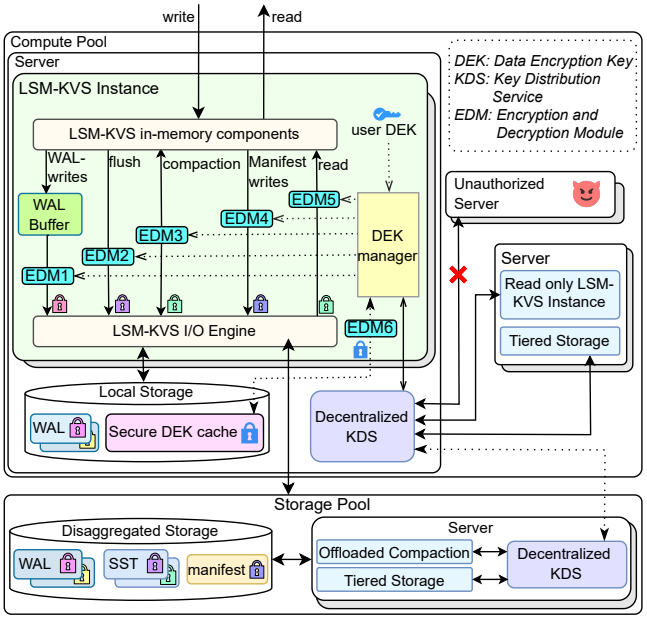


Figure 6: Design Overview for SHIELD.

5 SHIELD: Embedding Encryption in Write-Path of LSM-KVS in DS

We discussed how DEK handling practices of unique DEKs per file and regular DEK rotation enhance confidentiality guarantees in Section 2.2. While the instance-level encryption design can implement these practices, they are not necessary for controlled environments like in monolith or distributed setups. This section details the design of **SHIELD**, our solution that embeds DEK-handling practices into the LSM-KVS for disaggregated storage (DS) deployments while being compatible with monolithic deployments and sidestepping the encryption overhead of WAL-writes.

5.1 Overview of SHIELD Design

Integrating DEK-handling practices involves additional tasks and processes to request new DEKs and regularly rotate them. To minimize the overhead, SHIELD embeds these practices into the LSM-KVS. As shown in Figure 6, every time a critical file (SST, WAL, or Manifest) is created, a new DEK is requested from the KDS. For WAL-writes and flushes, encryption happens at the last stage to not interfere with the minimal processing and the self-sorting memtable structures, respectively. For compaction, encryption is done right after creating the SST file block to enable multi-threaded encryption. Additionally, the use of a new DEK on every file creation facilitates DEK rotation by compaction. For every data compaction, KV-pairs are re-encrypted using a new DEK. To reduce network latency, SHIELD implements a secure local cache for DEKs, protected by a user-provided passkey, which enables fast access to keys on database restarts (Section 5.2). Further, to sidestep the performance overhead of encrypting every WAL write, we design a

buffering solution that amortizes the cost of encrypting small KV-pairs (Section 5.3). Finally, SHIELD embeds a DEK-Identifier in each file metadata. Using this DEK-ID, a server can request the DEK from the KDS without relying on centralized control (Section 5.4).

To achieve on-disk data confidentiality for DS setups, SHIELD emphasizes flexibility by leveraging metadata-based DEK sharing and its secure cache that allows efficient key management across instances, allowing setups like offloaded compaction to work. Further, the use of a decentralized KDS allows SHIELD to be used with multiple instances in a read-only instance mechanism. SHIELD is tightly integrated with LSM-KVS, leveraging compaction for efficient DEK rotation and batching WAL writes to minimize encryption overhead. While per-file DEK usage could apply to other storage systems, SHIELD's reliance on LSM-KVS-specific mechanisms—like compaction-driven re-encryption and WAL buffering—makes direct adoption difficult. Systems requiring in-place file updates may face bottlenecks from repeated decryption and re-encryption, with frequent small writes further exacerbating overhead due to encryption initialization costs.

5.2 Embedding Encryption into LSM-KVS

SHIELD embeds encryption in the write path by leveraging LSM-KVS design knowledge to minimize the performance overheads from DEK-handling practices. SHIELD implements unique DEKs for each file with DEK rotations and designs a solution for highly available and decentralized key distribution with secure caching.

Encryption in the Write Path. Three processes are responsible for persisting client data to the WAL and SST files in LSM-KVS: WAL-write, Flush, and Compaction. SHIELD embeds the encryption in the write path of these processes.

WAL-writes are foreground operations that persist every client write operation immediately to ensure crash consistency. Data is persisted with minimal processing to ensure a high write throughput, leading to SHIELD implementing encryption right before persistence. SHIELD employs a buffer that collects and encrypts multiple WAL-writes to get around the bottleneck discussed in Section 3.2 for encrypting individual WAL-writes. We discuss this optimization in more detail in Section 5.3.

Flush is a background process that persists memtables to SST files. Memtables are in-memory self-sorting data structures (e.g., SkipLists) that keep updating until the last write when they reach the threshold. Preemptive data encryption could cause data to be re-encrypted if a write causes the entire data structure to update. To avoid this, SHIELD performs encryption just before data persistence for every flush operation.

Compaction is a background process responsible for up to 90% of I/Os for LSM-KVS [93]. It merges and sorts key-value pairs into ordered blocks of a pre-configured size (e.g., 4096 bytes by default in RocksDB), which are then persisted as SST files. Unlike instance-level encryption that encrypts all data at once, SHIELD performs encryption in user-configurable-sized chunks for finer-grained control. SHIELD further optimizes the chunk-based scheme with multi-threaded encryption during compaction, increasing resource usage but minimizing encryption overhead when handling large chunks.

The compaction style in an LSM-KVS significantly impacts encryption efficiency. Leveled compaction results in frequent writes, increasing encryption overhead, while tiered compaction involves fewer, larger I/O operations. SHIELD's use of metadata-driven DEK handling and chunk-based encryption allows it to adapt to different compaction styles with minimal performance degradation, balancing encryption overhead and I/O efficiency.

Embedding DEK-Handling Practices. SHIELD uses a ***unique DEK*** for each file with client data (WAL and SST) in LSM-KVS, by requesting a new DEK from a KDS whenever any write process creates a new file. This design introduces a new overhead of requesting a new DEK for every file creation, compared to the single DEK request for the instance level design from Section 4. However, the design is effective in limiting the confidentiality breach by allowing faster recovery from DEK compromises. If an attacker compromises an active DEK, the exposure risk is limited to a single file, not the entire KVS. Additionally, DEK rotation is much faster as it is only required for a single file, contrary to re-encrypting all files as for a single DEK implementation.

SHEILD enables ***DEK rotations*** by assigning new DEKs to files during compaction. As files of LSM-KVS are append-only and immutable, copies of the same KV-pair are collected over multiple files. Eventually, compaction merges and sorts these files, creating a new set of files containing only the most recent values for duplicate keys. A KV-pair using an older DEK (DEK-1) will switch to a newer DEK (DEK-2) during compaction. Integrating DEK assignments into compaction allows DEK rotation to become a built-in advantage that enhances security without incurring additional overhead. The integration of DEK-rotation with compaction allows no additional DEK-rotation overhead, however, it also limits SHIELD implementation to storage systems with a compaction-like implementation.

Decentralized Key Distribution Service. A Key Distribution Service (KDS) is used for issuing, managing, and revoking cryptographic keys [61, 62]. Various services, including PKI [24], Kerberos [66], Macaroons [23], and the Secure Swarm Toolkit [52], cater to different needs. We design SHIELD to integrate with a KDS that meets two criteria: (1) Decentralized implementation for high availability, and (2) Provisioning and requiring a DEK with a unique Identifier (DEK-ID) to allow SHIELD to embed DEK-IDs in LSM-KVS file metadata (discussed in Section 5.4).

On-Demand Key Retrieval with Secure Caching. The DEK is stored in memory as part of the LSM-KVS metadata while the LSM-KVS instance is running. However, upon restarting the database or recovering from a crash, making repeated requests for all relevant DEKs from the KDS introduces unnecessary latency. To avoid this, SHIELD uses a secure local disk cache to store previously used DEKs. These DEKs are encrypted with a local server password, which can be user-defined or controlled by the KDS. This server password is never persisted to disk and is required only when starting the LSM-KVS service. Multiple LSM-KVS instances (similarly to ZippyDB deployment [6]) on the same server can share this cache as long as the server password is provided to the other instances, thus eliminating additional network requests to the KDS and reducing network latency for DEK retrieval if the KDS is on a different server.

During DEK rotation, the secure cache is updated with the new DEK, storing it after securing it with the password. The DEK in the cache corresponding to the older files is deleted in conjunction with the SST files by SHIELD. This process ensures that only the current DEK is available for access.

5.3 Performance Optimizations for WAL

Section 3.2 discusses the encryption initialization overhead is critical on multiple small-sized writes. For a foreground process like the WAL that performs write synchronously, for smaller KV-pair sizes, encrypting each small WAL write can introduce significant performance slowdowns in LSM-KVS. To mitigate this, SHIELD leverages the understanding of the buffered I/O interface used by WAL-writes, and proposes a design that shifts data persistency guarantees from the OS to the LSM-KVS using an LSM-KVS managed buffer.

Buffered I/O Implementation in WAL. Disabling the WAL improves performance but compromises crash consistency, which is unacceptable. Popular LSM-KVS like LevelDB and RocksDB implement the WAL using the buffered I/O interface, allowing the OS to manage an in-memory buffer that aggregates multiple write operations before flushing them to disk, enhancing performance. This interface guarantees crash consistency [4] if the LSM-KVS process crashes. However, this crash consistency guarantee does not extend to data persistency, as any data present in the OS buffer will be lost in the event of a system crash.

Performance and Data Persistency Trade-Off with Buffered WAL. A naive approach to balance the performance and data persistency in LSM-KVS is to implement a secondary WAL alongside the primary WAL. As writes persist in the primary WAL, the secondary WAL asynchronously receives encrypted writes. Once both WALs are full, the primary WAL is deleted, and the secondary WAL replaces it. In a crash, some writes may not have persisted to the secondary WAL, and the plaintext WAL is used for recovery of the active log. In contrast, the encrypted secondary WAL handles recovery for immutable logs. Although this design upholds data persistency, it exposes unencrypted data in the primary WAL, compromising security and introducing overhead from CPU, memory, and storage due to the dual WAL management and encryption.

To achieve robust security with high performance, SHIELD uses an application-controlled buffer for WAL writes. Instead of relying on the OS, SHIELD shifts persistency control to the application, temporarily storing writes in memory and encrypting them before persisting once a user-set threshold is reached. By amortizing encryption and I/O costs over a larger batch of data, this approach significantly reduces the overhead of write operations compared to the naive design. SHIELD ensures that KV-pairs are encrypted within the same buffer, preventing partial data loss during crashes and supporting proper WAL replay for memtable recovery.

SHIELD introduces a trade-off in data persistency: any unpersisted data in the buffer is lost if the LSM-KVS crashes, shifting the failure risk from OS crashes to application-level failures. However, SHIELD guarantees that only encrypted data is written to storage, fully addressing the threat model's security requirements. While this approach incurs slight memory overhead (i.e., 512-byte buffer) for the buffer and adds computational costs for encryption, it provides a more efficient and secure alternative to the naive design.

5.4 Secure DEK Sharing in DS

In LSM-KVS that are deployed in DS, tasks like compaction can be offloaded to separate servers [22, 36, 44, 93], and network I/O overhead can mask the performance impact of encryption. For example, when new SST files are created on remote storage servers during offloaded compaction, the network I/O latency can absorb much of the encryption overhead, making it less noticeable. However, in such systems, remote servers also need access to the DEKs to perform operations like compaction or SST file migration.

A naive approach to managing encryption in such setups is to have the KDS handle the mapping between files and DEKs. However, this introduces latency for DEK provisioning and adds the complexity of the KDS's responsibilities. In cases like offloaded compaction (e.g., implemented in Disaggregated-RocksDB [36]), where temporary filenames are assigned during new SST file creation and compaction results updated in the LSM-KVS metadata, this approach can lead to mismatches between the DEK and the final file name. Fixing these mismatches would require extra operations, increasing both complexity and latency. Moreover, it introduces availability risks, as the KDS could become a single point of failure, exacerbating potential performance bottlenecks.

Metadata-Enabled DEK Sharing. SHIELD's design addresses the complexities of mapping files with DEKs by incorporating a DEK identifier (DEK-ID) into the metadata of WAL and SST files. In an LSM-KVS, metadata is read before data blocks to identify any operations that need inversion (e.g., compression). SHIELD leverages this process, allowing the LSM-KVS to preemptively request the necessary DEK from the KDS using the embedded DEK-ID. This approach simplifies the design by enabling decentralized DEK management, making the system more resilient to failures. Such DEK management allows SHIELD to support different KDS policies, such as per-server sharing, per-file isolation, or hierarchical derivation, by relying only on the DEK-ID for retrieval, making it agnostic to how the KDS manages and distributes keys as long as it can resolve the DEK-ID to a valid key.

However, since the metadata is stored in plaintext, the DEK-ID is exposed to anyone with server access, including potential attackers. If an attacker gains access, they could read the DEK-ID, request a new DEK from the KDS, and access client data. To mitigate this, SHIELD relies on the KDS to employ two safeguards: server authorization and one-time DEK provisioning. First, the KDS only allows authorized servers to request DEKs. If a server is breached, the KDS can revoke its authorization to block further DEK requests. Second, the KDS enforces one-time DEK provisioning, denying any attempt to request a DEK that has already been issued, preventing retrieval even if the DEK-ID is exposed.

5.5 Security Guarantees

With the design as mentioned above, SHIELD provides embedding for encryption with DEK-handling practices in LSM-KVS for different flexible deployments. This section briefly discusses some scenarios and how SHIELD handles them.

Scenario 1 - Storage Media Compromise: If the storage media (e.g., HDD, SSD) is physically stolen or its data is dumped over the network by an attacker, data is at risk of exposure to the bad actor. SHIELD's integration of robust encryption algorithms for all on-storage files guarantees data confidentiality.

Scenario 2 - Unauthorized Access: Data should not be compromised if an attacker with filesystem access or other server user tries to access on-disk data. Performing encryption on the LSM-KVS level safeguards data from such threats.

Scenario 3 - DEK compromise: In the even more challenging scenario of a DEK compromise, detected or undetected, a strong attacker with access to the DEK and the filesystem can decrypt the associated data. The compromised DEK can be replaced if detected, and the file will be re-encrypted with a new DEK. In the case of undetected compromise, SHIELD's design minimizes data exposure by restricting access to only the file associated with the compromised DEK. Moreover, if the file has already been compacted, the DEK becomes ineffective, preventing any further data leakage.

5.6 Case Study: Offloaded Compaction in DS

In DS, network I/O latency can hide the encryption overhead. However, in setups like offloaded compaction [22, 36, 93], where SST files are read and written across different servers, remote servers need access to DEKs for encryption and decryption. SHIELD handles this efficiently using a metadata-driven approach. When creating new SST files, remote servers read the metadata to identify the required DEK, request it from the decentralized KDS, and securely cache it on disk, allowing the compaction process to proceed without repeated requests. SHIELD's decentralized DEK management and caching system reduces latency and enhances resilience by avoiding a single point of failure.

6 Implementation and Evaluations

We implement our two designs, Instance-level Encryption (EncFS) and SHIELD, based on RocksDB (v8.8.1) [80], a widely used LSM-KVS. The source code is available on GitHub [85]. We conduct comprehensive evaluations to answer the following questions:

- What are the performance overhead of EncFS and SHIELD compared with RocksDB under Micro (RandomWrite, RandomRead) and Macro (YCSB, Mixgraph) workloads in a monolithic setup?
- What are the impacts of different design components of SHIELD evaluated in the detailed breakdown analysis?
- How does SHIELD perform in DS deployments?

6.1 Experimental Setup

For our evaluation, we refer to an unmodified, out-of-box RocksDB setup as *unencrypted RocksDB*. When using the version of RocksDB that integrates EncFS, we address it as *EncFS*; and when using SHIELD, we address it as *SHIELD*. Finally, the *WAL Optimization (WAL-Buf* referred to in the evaluation is the WAL optimization proposed in Section 5.3.

Hardware Setup. Our experimental evaluation utilizes two servers: Server 1 with Intel Xeon Gold 6330 CPU @ 2.00GHz, 256 GiB of RAM, a 3.84TB SAS SSD running Ubuntu 22.04.3 LTS, and Server 2 with Intel Xeon Silver 4310 CPU @ 2.10GHz, 64 GiB of RAM, an 8TB SATA HDD storage running Ubuntu 20.04.6 LTS. To ensure minimal latency in DEK provisioning, we deploy SSToolkit on Server 1, which always handles client communication. Server 1 and Server 2 are connected via a network switch with a 1Gbps link that facilitates communication.

In the *Monolithic* setup, we utilize server 1 to handle both computing and storage. This server's more powerful CPU and high-performance SAS SSD allow us to minimize I/O and computational bottlenecks, reflecting a tightly coupled architecture where compute and storage reside on the same node. In the *Disaggregated Storage* setup, server 1 is repurposed as the compute servers, while server 2 acts as the disaggregated storage utilizing HDFS [1]. Both servers are on the same rack and connected to a gigabit connection switch. We further implement the offloaded compaction (the same as Disaggregated-RocksDB [36] and CaaS-LSM [93]) at the storage servers to demonstrate the compatibility of SHIELD for LSM-KVS optimizations for DS.

Key Distribution Service. To be compatible with SHIELD, a KDS has to be (1) decentralized for DS implementations and (2) Provide access control and unique DEK identifiers. We considered four options, PKI [24], Kerberos [66], Secure Swarm Toolkit [52], and Macaroons [23]. We choose the open-source Secure Swarm Toolkit (SSToolkit) for its relatively lightweight and highly customizable C API [50, 51] with a decentralized implementation.

Encryption Algorithm Selection. SHIELD is flexible and allows the implementation of different encryption algorithms that the KDS allows. We use AES, the industry-standard and fastest symmetric encryption algorithm, widely used for its high performance and strong security [31]. We run 128-bit AES in CTR mode, which provides robust encryption for data confidentiality while maintaining a lightweight implementation to minimize performance overhead [2, 76].

Workloads. We use different test cases from popular benchmarking tools, including db_bench [37] (fillrandom, readrandom, readrandomwriterandom, mixgraph) and YCSB [27] (A, B, C, D, E, F). All pure write tests are conducted over 50 Million (M) KV-pairs unless specified and all tests involving read over 10M KV-pairs. We utilize direct I/O for compaction, reads, and flushes across all tests. Unless specified, all parameters of LSM-KVS are the default as found in db_bench (e.g. Key Size: 16 bytes, Value Size: 100 bytes, etc).

6.2 Evaluation for Monolith LSM-KVS

To effectively evaluate the overall performance implication in different scenarios, we utilize multiple tests from db_bench and YCSB. We divide our tests into micro and macro benchmarks to evaluate focused and real-world scenarios.

Micro Benchmarks. We use fillrandom, readrandom, and a set of mixed read-and-write ratios tests on writerandomreadrandom benchmark in db_bench to simulate situations of random writes, random reads, and a mixed test. The random write tests are run with 50M operations, and the other tests with 10M operations. Figure 7 and Figure 8 illustrate these results. For the random write tests, when compared to unencrypted RocksDB, we observe a performance degradation of 32.9% and 36.2% for EncFS and SHIELD, respectively. This degradation is reduced to 16.6% and 19.4% when comparing the solutions with the WAL buffer optimization. This is the worst-case scenario for our designs, as in situations of reads, the internal latency of LSM-KVS is high enough to hide any added latency of decrypting data, as shown in the random read tests where the performance degradation is within 1% of unencrypted RocksDB. We also show different scenarios of a mixed read and write workload in Figure 8 where we observe a reduction in performance overhead with the increasing ratio of reads until, eventually, there is less than a 1% performance difference between our solutions and unencrypted RocksDB.

Macro Benchmarks. We use two tests to evaluate more real-world-like situations: Mixgraph (Figure 7) and YCSB (Figure 9).

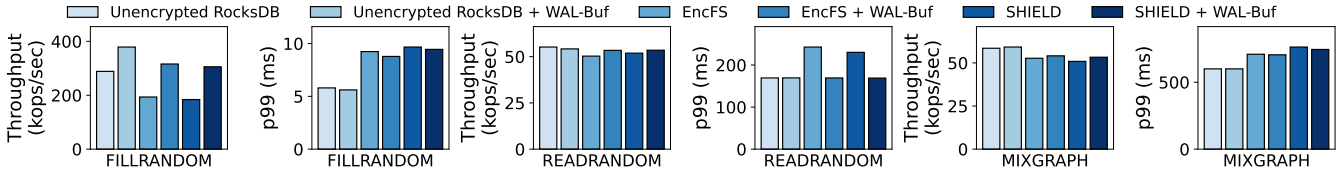


Figure 7: Baseline Results in a Monolith Server.

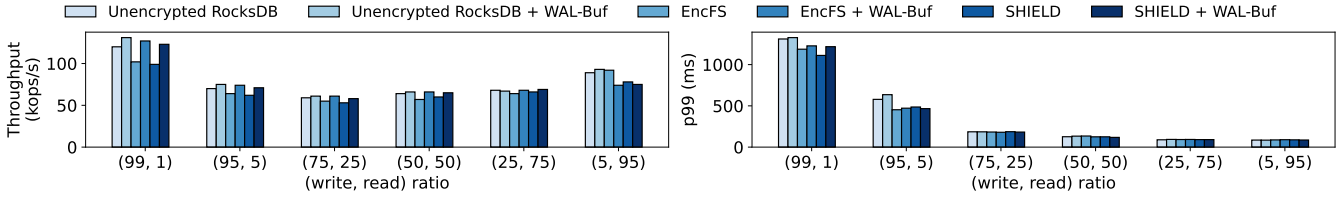


Figure 8: Throughput and P99 Latency for Different Read and Write Ratios in a Monolith Server.

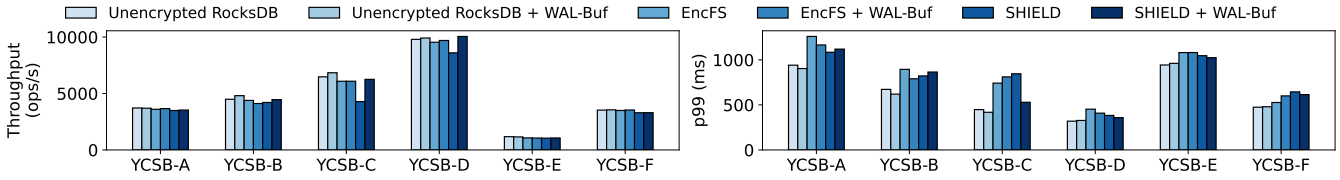


Figure 9: Throughput and P99 Latency for YCSB Tests in a Monolith Server.

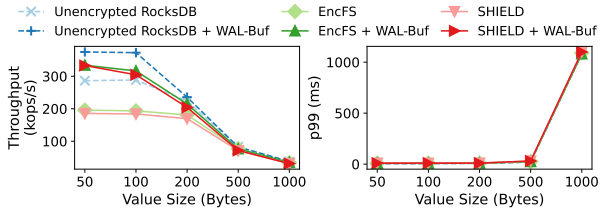


Figure 10: Sensitivity to Value Sizes.

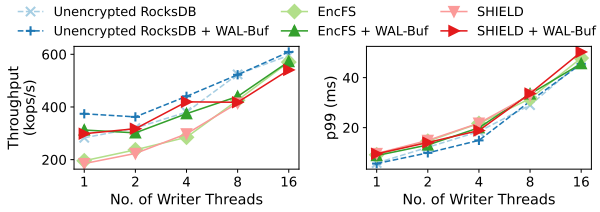


Figure 11: Sensitivity to Writer Threads.

Mixgraph [26], is run with a preloaded database of 50M KV-pairs followed by 10M operations run on it. YCSB [27], with much larger value sizes (1KB compared to approx. 37 bytes for Mixgraph), is run on a preloaded database with 10M operations followed by 1M operations. With real-world tests being a mixture of read and write operations, we expect a smaller performance degradation and observe the same for EncFS and SHIELD. Mixgraph shows a performance overhead of 10% and 12.9% compared to an unencrypted RocksDB, and YCSB has an overhead ranging from 2-15% and 1-23% for EncFS and SHIELD. The least overhead (0% for SHIELD, <2% for EncF) is observed in YCSB-D, a 95% read and 5% insert workload.

6.3 Sensitivity Analysis

Key-Value Pair Sizes. In Section 3.2, we discuss the cost of encryption and how it is amortized for larger chunks of data. This property means that for larger KV-pair sizes, the differences between the baseline solutions would slowly decrease. Figure 10 showcases this as with value size increasing, all different test variations eventually converge. For the unbuffered solutions, small value sizes of 50 bytes, EncFS and SHIELD have an overhead of 31% and 35% against unencrypted RocksDB, which decreases to 9% and 16% at value sizes of 1000 bytes.

Writer Threads. RocksDB implements a pipelined writer [3], where a single queue is maintained for all writers. In such a case, if the write queue is already overwhelmed with a larger number of operations, we would expect the WAL optimizations to prove to be insufficient. To demonstrate this, we use 16 background jobs, ensuring a sufficient amount is present to ensure the bottleneck lies on the writer. Figure 11 demonstrates this as the impact of WAL optimization (WAL-Buf) drops from 21.9% to 1.1% on average for all implementations when using 8 writer threads. With increasing writer threads, the performance of the WAL-Buf solutions converges towards the non-optimized solutions, with the bottleneck shifted from WAL writes to the write ingestion rate.

Background Threads. Flush and compaction are background processes in LSM-KVS and can be responsible for over 90% of I/Os, which by extension makes them responsible for over 90% of the encryption. To isolate the impact of background threads, we fix the number of writer threads to 4. Figure 12 shows results from our test where we find that under limited resources (2 background jobs), SHIELD with a WAL buffer suffers with performance degradation of 6% compared to unencrypted RocksDB without a WAL buffer. However, as soon as enough resources are provided with 4 background

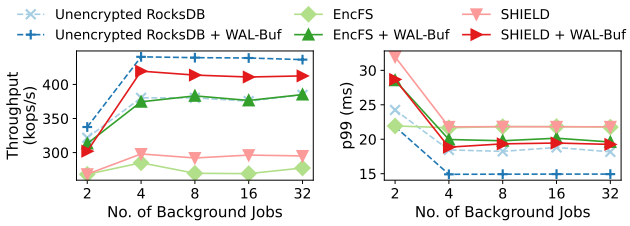


Figure 12: Sensitivity to Background Threads.

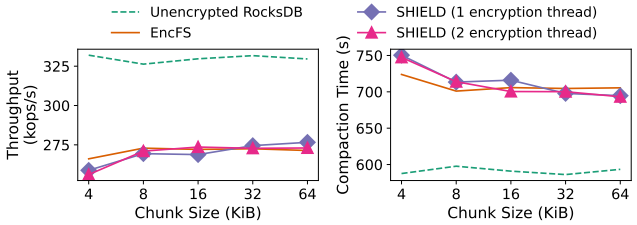


Figure 13: Sensitivity to Chunk Sizes and Encryption Threads.

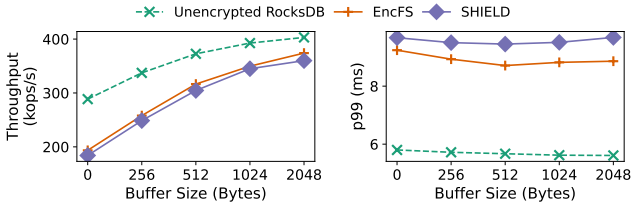


Figure 14: Sensitivity to Buffer Sizes.

threads, SHIELD with the WAL buffer has a 10% performance uplift compared to the unencrypted RocksDB variant.

Chunk Sizes and Encryption Threads. For compaction, SHIELD encrypts data at the block level, enabling parallel encryption across multiple blocks. Figure 13 illustrates the compaction time with threaded encryption, comparing SHIELD against EncFS and unencrypted RocksDB. We observe that while initially, the cost of compression is higher, with an increasing chunk of data being encrypted at the same time, the threading implementation of SHIELD starts to steadily improve. We continue this test up to 2MB chunks, where we observe faster compaction times when treating unencrypted RocksDB as our starting point.

Buffer Sizes. With larger buffer sizes, the encryption initialization overhead will be amortized over more write operations. This is similar to increasing value sizes, where the implementations converge over time. Figure 14 demonstrates that with increasing buffer sizes from 0 (no buffer) to 2048 bytes, the performance overhead of encryption in EncFS (from 32% to 7%) and SHIELD (from 36% to 10%) decreases when compared to unencrypted RocksDB.

Compaction Policies. In LSM-KVS, such as RocksDB, changing compaction policies can impact how the system performs for different workload scenarios. Considering the reliance of SHIELD on compaction to trigger core tasks such as DEK-rotation, we test SHIELD performance with offloaded compaction for different compaction policies (RocksDB's leveled, universal, and FIFO policies) under a 100% random write and random read workload with offloaded compaction turned on. Figure 15 demonstrates SHIELD can perform consistently with a performance overhead varying from

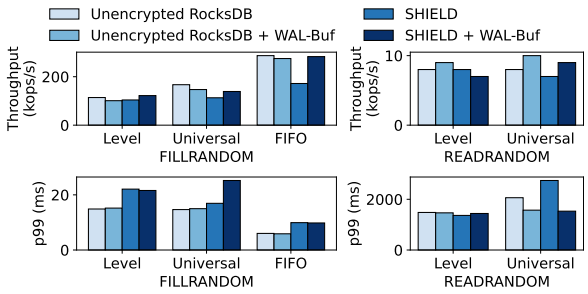


Figure 15: Different Compaction Policies.

Table 3: Read (R) and Write (W) I/O distribution (in GiB) for Different Compaction Styles divided by Server, Operation, and target Storage Media.

		Server 1			Server 2
		WAL-Write (local storage)	Flush (HDFS)	Compaction (HDFS)	Compaction (HDFS)
Level	**Read**	N/A	N/A	0.09 GiB	15.24 GiB
	Write	6.1 GiB	5.61 GiB	0	14.05 GiB
Universal	**Read**	N/A	N/A	0.09 GiB	13.75 GiB
	Write	6.1 GiB	5.66 GiB	0	13.75 GiB
FIFO	**Read**	N/A	N/A	0	0
	Write	6.1 GiB	5.60 GiB	0	0

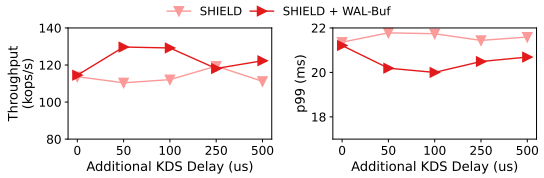


Figure 16: Impact of KDS Latency.

0-40% in random write (fillrandom) tests and 0-11% with random read (readrandom) tests compared to unencrypted RocksDB across different compaction policies. We do not include the readrandom results of FIFO compaction owing to KV-pairs written earlier deleted by FIFO, hence making the reads fail and give a skewed value (approx. 2.5M ops/sec). We also provide supplementary information about SHIELD's average I/O distribution for different compaction styles in Table 3, finding that the compute and compaction servers generate I/O operations to HDFS at a ratio of approximately 1:5.

KDS Latency. The DEK distribution policy and latency both impact SHIELD performance. In this paper, we use SSToolkit, which supports and utilizes the policy of sending one key per request. To evaluate SHIELD under different KDS latency conditions, we synthetically place delays in the KDS and test SHIELD in an offloaded compaction setting. We find that SSToolkit, on average, takes 2750μs to generate and send a DEK to SHIELD. Meanwhile, the latency for an intra-datacenter round-trip is 500μs [7]. The difference in latencies indicates a minimal impact from placing KDS on different servers and, instead, a more profound impact from changing the KDS policies. As illustrated in Figure 16, we find SHIELD to have similar performance as the KDS latency increases with a maximum difference of 10% in throughput to 6% in p99 latency.

Increasing Dataset Sizes. We stress-test the proposed solution by running SHIELD in a disaggregated storage setup under larger and longer running tests. We set the key and value size of 16 and 240 bytes, respectively, and ran evaluations for datasets ranging from 50M KV-Pairs (approx. 10 GB in size) to 1000M KV-Pairs (approx.

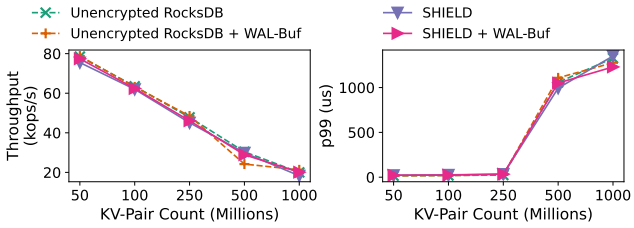
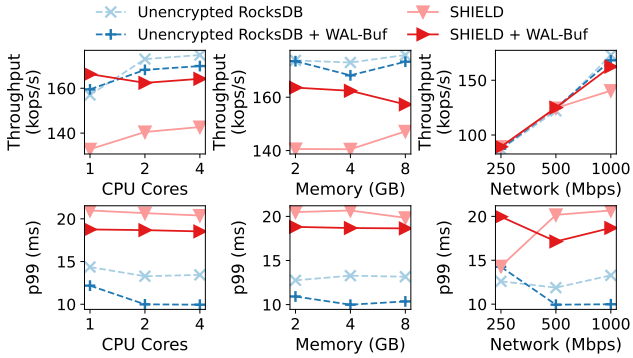


Figure 17: Increasing Dataset sizes.



(a) Variation of CPU cores with 4GB RAM and 1Gbps B/W. **(b) Varying RAM with 2 CPU Cores and 1Gbps B/W.** **(c) Varying B/W with 2 CPU Cores and 4 GB RAM.**

Figure 18: Sensitivity to different CPU Cores, Memory (RAM), and Network Bandwidth (B/W) configurations.

200 GB in size). We find SHIELD consistently has a performance overhead of less than 10%.

CPU, Memory and Network Bandwidth. The amount of resources provided to SHIELD can impact the performance profoundly. To test this, SHIELD is used in offloaded compaction setup; we utilize Linux cgroups to control system CPU and memory resources and TC for network bandwidth limitations that are provided to the setup. As illustrated in Figure 18, we find SHIELD to be least impacted by CPU and Memory variations and the system to be most impacted (throughput improvement of approx. 77%) by increasing network bandwidth, indicating our system bottleneck to be with the bandwidth, and SHIELD able to perform consistently with a maximum 20% overhead with limited system resources.

6.4 Evaluation for LSM-KVS in DS

We use two different setups to evaluate our solution: 1. Disaggregated Storage where our solution is deployed on one server and connected using the RocksDB-HDFS plugin to an HDFS deployment (pseudo-distributed mode) running on a second server, and 2. Offloaded Compaction, where we build on top of the disaggregated storage solution by also deploying an offloaded compaction implementation [36, 93] on the second server where HDFS has been deployed. We notably exclude our EncFS solution, which is designed for monolithic setups and is not compatible with the HDFS plugin implementation.

Disaggregated Storage. Figures 19, 20, and 21 show the same set of micro and macro benchmarks as the monolithic tests. The increased network latency significantly narrows the performance

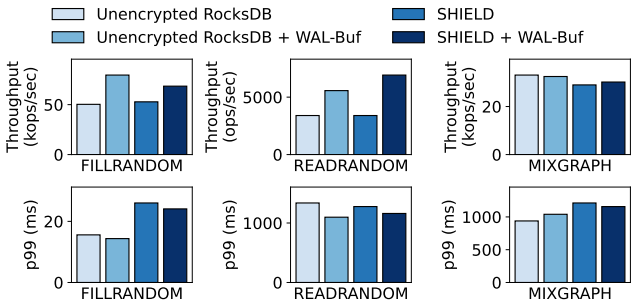


Figure 19: Baseline Results with Disaggregated Storage.

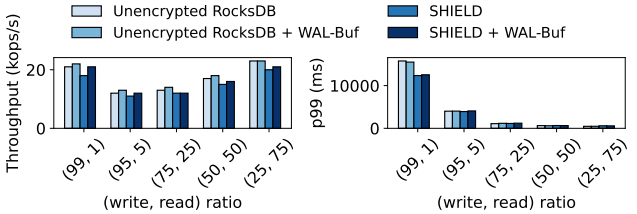


Figure 20: Throughput and P99 Latency for Different Read and Write Ratios in Disaggregated Storage.

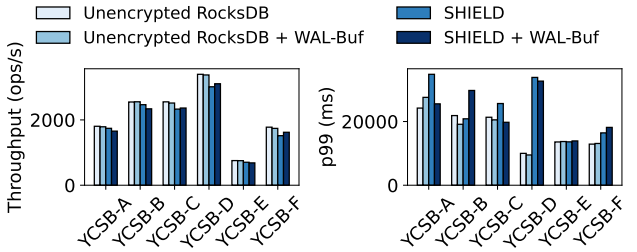


Figure 21: YCSB Results with Disaggregated Storage

gap in the random writes test (fillrandom) to 5% between SHIELD and unencrypted RocksDB even without the consideration of the WAL buffer. In the other micro tests with different read and write ratios (Figure 20), the performance disparity ranges from 6-14%, which is an improvement over the monolith structure. For the macro benchmarks, the performance variances are 8% (YCSB average) and 10% (Mixgraph), which are consistent with expectations due to the additional network latency.

Offloaded Compaction. Figures 22, 23, and 24 show the same suite of micro and macro benchmarks as we do for the monolith tests. The offloaded compaction test suite aims to showcase the system's capability in an environment where DEKs are retrieved over the network to demonstrate the efficacy of our metadata-embedded DEK-ID solution. The performance disparity for the fillrandom tests is 17% between SHIELD and the unencrypted RocksDB solutions. Regarding the macro benchmarks, the performance variances are 4% (YCSB average) and 8.3% (Mixgraph), aligning them with our expectations.

7 Related Work

Secure Datastores. Encrypted databases like CryptDB [74, 75], Monomi [89], Seabed [70], Arx [73], TrustedDB [17], and DJoin [64] protect data confidentiality by encrypting data and processing

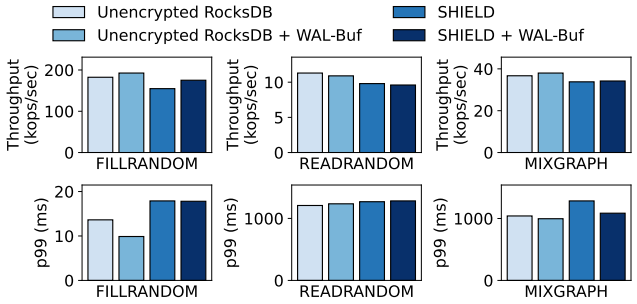


Figure 22: Baseline Results with Offloaded Compaction.

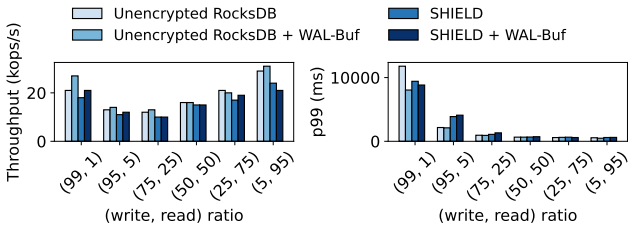


Figure 23: Throughput and P99 Latency for Different Read and Write Ratios with Offloaded Compaction.

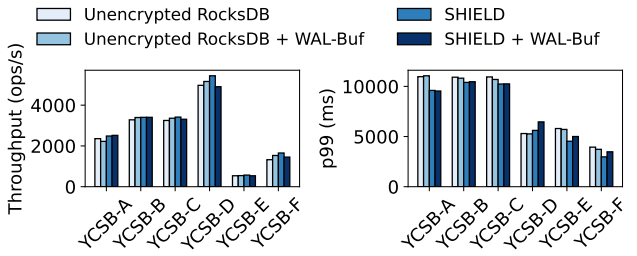


Figure 24: YCSB Results with Offloaded Compaction.

queries over ciphertext using techniques such as homomorphic encryption, secure multi-party computation, or trusted hardware. However, unlike these systems that focus on monolithic databases, SHIELD offers a scalable and flexible design that is suitable for LSM-KVS in disaggregated storage settings. MariaDB [60] supports encryption for data at rest employing a key management plugin, also supporting key rotation. However, MariaDB is a relational database and does not support disaggregated setups.

Secure Distributed Datastores. Several distributed file systems and databases, including HDFS, GlusterFS [41], and Cassandra [82], provide high throughput and support for authentication and transport encryption (via TLS) between nodes. However, they lack built-in encryption for data at rest. HDFS [11, 12, 87] and JuiceFS [48] support encryption for data both in transit and at rest. However, JuiceFS does not support key rotation and per-file encryption. HDFS supports an encryption zone, a special directory that is encrypted, and employs a Key Management Server to manage encryption keys. However, SHIELD provides more granular access control for per-file encryption and employs an integrated encryption and key management solution for LSM-KVS. Tectonic [69] also supports access control but still lacks encryption for data at rest. InterPlanetary File System (IPFS) [20], a peer-to-peer highly decentralized storage, only supports transport encryption, not encryption for the data by default [45] although there have been attempts [18, 47] to provide encryption and access control for the data stored in IPFS.

Secure Key-Value Stores. EncKV [94], Yuan *et al.* [95], and Agarwal and Kamara [8] propose an encrypted key-value store (KVS) that supports secure, efficient query processing leveraging searchable symmetric encryption. However, their approaches target SQL databases and do not address performance overhead associated with frequent small encryption calls. SHIELD specifically targets LSM-KVS and addresses the performance challenges of small atomic writes performed by WAL employing buffered WAL write to minimize overhead while maintaining data confidentiality.

Avocado [15] and ShieldStore [54] both leverage trusted execution environments (TEEs) to enhance in-memory security in untrusted environments. Avocado focuses on distributed storage and consistency across a network, and ShieldStore optimizes memory management and throughput for in-memory KVS. These approaches rely on specific hardware for TEE implementation and are not designed for disaggregated data or persistent data in LSM-KVS. SHIELD provides a scalable and decentralized solution that does not rely on specialized hardware, enabling LSM-KVS deployments in monolithic and disaggregated storage.

Secure LSM-KVS. Kim and Vetter [53] integrate data compression and encryption into high-performance computing LSM-KVS to enhance storage efficiency, performance, and security. However, their work is concentrated on data compression and does not address challenges for embedding DEK handling practices, flexible deployments for disaggregated deployments, and encryption overhead from encrypting every WAL write. SHIELD addresses these challenges, achieving data confidentiality for data at rest in LSM-KVS.

Other studies focus on in-memory data protection by leveraging TEEs. SPEICHER [16] enforces strong security and data freshness, PLDB [84] reduces encryption overhead by reducing the interface calls through the TEE, directly storing encrypted data on the disk. Li *et al.* [55] proposes an authenticated data structure by digesting individual LSM tree levels. All three solutions heavily rely on using a single DEK on TEEs (specifically leveraging Intel SGX [29, 30]). In contrast, SHIELD focuses on persistent data protection in LSM-KVS, embeds encryption with DEK-handling practices into the LSM-KVS architecture, and does not necessitate specialized hardware. The non-reliance on hardware for SHIELD allows other databases and KV-stores that share a similar design for file persistence (e.g., log-structured hash-based key-value stores like BloomStore [59] or FlashStore [33]) to leverage the SHIELD design.

8 Conclusion and Future Work

In this paper, we presented SHIELD, a novel design that secures data confidentiality for persistent data in LSM-KVS while minimizing the encryption overhead. By embedding DEK-handling practices into both monolithic and disaggregated storage, SHIELD ensures strong encryption practices with minimal performance impact, incurring a maximum of 36% overhead in monolithic and 15% in disaggregated setups compared to unencrypted RocksDB. This work secures the persistent storage layer of LSM-KVS, opening several avenues for future research on extending encryption to volatile components like

in-memory caches and exploring advanced techniques such as homomorphic encryption for secure, direct computation on encrypted data.

Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. We thank all the members of ASU-IDI Lab for providing useful comments. This work was partially funded by National Science Foundation under Grant Number 2412436, 24432196, NSF I/UCRC for Intelligent, Distributed, Embedded Applications and Systems (IDEAS), and from NSF grant 2231620.

References

[1] [n. d.]. Apache Hadoop 3.3.1 – HDFS Architecture. https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[2] [n. d.]. File Key Management Encryption Plugin. https://mariadb.com/kb/en/file-key-management-encryption-plugin/

[3] [n. d.]. Pipelined Write · facebook/rocksdb Wiki. https://github.com/facebook/rocksdb/wiki/Pipelined-Write

[4] [n. d.]. Write Ahead Log (WAL). https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-(WAL)

[5] 2024. google/leveldb. https://github.com/google/leveldb original-date: 2014-08-27T21:17:52Z.

[6] Accessed: June, 2023. ZippyDB: a modern, distributed key-value data store. https://www.youtube.com/watch?v=DfiN7pG0D0k.

[7] 262588213843476. [n. d.]. Latency Numbers Every Programmer Should Know. https://gist.github.com/jboner/2841832

[8] Archita Agarwal and Seny Kamara. 2020. Encrypted key-value stores. In *Progress in Cryptology–INDOCRYPT 2020: 21st International Conference on Cryptology in India, Bangalore, India, December 13–16, 2020, Proceedings 21*. Springer, 62–85.

[9] Ibrar Ahmed. 2024. What is Transparent Data Encryption (TDE)? The Ultimate Guide. https://www.percona.com/blog/transparent-data-encryption-tde/

[10] Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, and Sean Peisert. 2021. Performance Analysis of Scientific Computing Workloads on General Purpose TEEs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1066–1076. https://doi.org/10.1109/IPDPS49936.2021.00115

[11] Apache Hadoop. 2020. Hadoop Transparent Encryption. https://hadoop.apache.org/docs/r3.3.0/hadoop-project-dist/hadoop-hdfs/TransparentEncryption.html Accessed: 2024-10-16.

[12] Apache Hadoop. 2024. Hadoop Secure Mode - Data Confidentiality. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SecureMode.html#Data_confidentiality Accessed: 2024-10-15.

[13] Apache Hadoop 3.3.1 – Transparent Encryption in HDFS [n. d.]. Apache Hadoop 3.3.1 – Transparent Encryption in HDFS. https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/TransparentEncryption.html

[14] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. https://www.usenix.org/conference/fast14/technical-sessions/presentation/keynote

[15] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Vijay Nagarajan, Pramod Bhatotia, et al. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 65–79.

[16] 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)*. 173–190.

[17] Sumeet Bajaj and Radu Sion. 2011. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 205–216.

[18] Ammar Ayman Battah, Mohammad Moussa Madine, Hamad Alzaabi, Ibrar Yaqoob, Khaled Salah, and Raja Jayaraman. 2020. Blockchain-based multi-party authorization for accessing IPFS encrypted data. *IEEE Access* 8 (2020), 196813–196825.

[19] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (aug 2015), 26 pages. https://doi.org/10.1145/2799647

[20] Juan Benet. 2014. IPFS-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561* (2014).

[21] Daniel J Bernstein et al. 2008. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, Vol. 8. Citeseer, 3–5.

[22] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 301–316. https://doi.org/10.1145/3373376.3378504

[23] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. 2014. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium*.

[24] Johannes Buchmann, Evangelos Karatsiolis, Alexander Wiesmaier, and Evangelos Karatsiolis. 2013. *Introduction to public key infrastructures*. Vol. 36. Springer.

[25] Zhichao Cao, Huibing Dong, Yixun Wei, Shiyong Liu, and David H. C. Du. 2022. ISHBase: An In-Storage Computing Optimized HBase with I/O Offloading and Self-Adaptive Caching in Compute-Storage Disaggregated Infrastructure. *ACM Trans. Storage* 18, 2, Article 15 (apr 2022), 42 pages. https://doi.org/10.1145/3488368

[26] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[27] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[28] Jonathan Corbet. 2019. Buffered I/O without page-cache thrashing [LWN.net]. https://lwn.net/Articles/806980/

[29] Intel Corporation. 2024. Intel® Software Guard Extensions (Intel® SGX). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html Accessed: 2024-08-28.

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

[31] Joan Daemen and Vincent Rijmen. 2002. *The design of Rijndael*. Vol. 2. Springer.

[32] Haoran Dang, Chongnan Ye, Yanpeng Hu, and Chundong Wang. 2022. NobLSM: an LSM-tree with non-blocking writes for SSDs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 403–408.

[33] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1414–1425.

[34] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[35] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage* 17, 4 (Oct. 2021), 26:1–26:32. https://doi.org/10.1145/3483840

[36] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 192:1–192:24. https://doi.org/10.1145/3589772

[37] Facebook. [n. d.]. RocksDB Benchmarking Tools. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Accessed: 2024-09-16.

[38] Facebook:ZippyDBArchitecture 2021. How we built a general purpose key value store for Facebook with ZippyDB. https://engineering.fb.com/2021/08/06/core-infra/zippydb/

[39] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 249–264. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

[41] Gluster Community. 2024. GlusterFS - Scale-out Network Attached Storage. https://www.gluster.org/ Accessed: 2024-10-15.

[42] Google. 2024. Data encryption with a customer-managed key in Azure Database for PostgreSQL - Flexible Server. https://cloud.google.com/kms/docs/envelope-encryption#data_encryption_keys

[43] Google. 2024. Envelope encryption - Data encryption keys. https://cloud.google.com/kms/docs/envelope-encryption#data_encryption_keys

[44] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 749–763. https://doi.org/10.1145/3448016.3457297

[45] InterPlanetary File System. 2024. IPFS Concepts: Privacy and Encryption. https://docs.ipfs.tech/concepts/privacy-and-encryption/#node-identifiability Accessed: 2024-10-15.

[46] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2008. Cryptography with constant computational overhead. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 433–442.

[47] Yeongbin Jo, Yunsang Cho, and Hokeun Kim. 2023. Secure and Lightweight Access Control for Highly Decentralized and Distributed File Systems. In *Proceedings of the 1st International Workshop on Middleware for the Computing Continuum*. 1–6.

[48] JuiceFS. 2023. JuiceFS Encryption Documentation. https://juicefs.com/docs/community/security/encryption/ Accessed: 2024-10-16.

[49] Gary Kessler. [n. d.]. An Overview of Cryptography. https://www.garykessler.net/library/crypto.html

[50] Dongha Kim, Yeongbin Jo, Taekyung Kim, and Hokeun Kim. 2023. SST v1. 0.0 with C API: Pluggable security solution for the Internet of Things. *SoftwareX* 22 (2023), 101390.

[51] Dongha Kim and Hokeun Kim. 2023. Securing Edge-Based Real-Time IoT Systems. In *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*. 544–545.

[52] Hokeun Kim, Eunsuk Kang, Edward A Lee, and David Broman. 2017. A toolkit for construction of authorization service infrastructure for the internet of things. In *Proceedings of the second international conference on Internet-of-Things design and implementation*. 147–158.

[53] Jungwon Kim and Jeffrey S Vetter. 2019. Implementing efficient data compression and encryption in a persistent key-value store for HPC. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1098–1112.

[54] 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*. 1–15.

[55] Kai Li, Yuzhe Tang, Qi Zhang, Jianliang Xu, and Ju Chen. 2021. Authenticated key-value stores with hardware enclaves. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*. 1–8.

[56] ARM Limited. 2024. ARM Security Technology: Building a Secure System using TrustZone Technology. https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2057-00-00-00-00-53-99/PRD29_2D00_GENC_2D00_009492C_5F00_trustzone_5F00_security_5F00_whitepaper.pdf Accessed: 2024-08-28.

[57] Rui Lin, Yuxin Cheng, Marilet De Andrade, Lena Wosinska, and Jiajia Chen. 2020. Disaggregated Data Centers: Challenges and Trade-offs. *IEEE Communications Magazine* 58, 2 (2020), 20–26. https://doi.org/10.1109/MCOM.001.1900612

[58] Robert Love. [n. d.]. 3. Buffered I/O - Linux System Programming [Book]. https://www.oreilly.com/library/view/linux-system-programming/0596009585/ch03.html ISBN: 9780596009588.

[59] Guanlin Lu, Young Jin Nam, and David HC Du. 2012. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST 12)*. IEEE, 1–11.

[60] MariaDB Corporation. 2024. Data at Rest Encryption Overview. https://mariadb.com/kb/en/data-at-rest-encryption-overview/ Accessed: 2024-10-11.

[61] Microsoft. 2023. Create the Key Distribution Services (KDS) Root Key. https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/manage/group-managed-service-accounts/group-managed-service-accounts/create-the-key-distribution-services-kds-root-key. Accessed: 2024-10-01.

[62] Microsoft. 2023. Key Distribution Center. https://learn.microsoft.com/en-us/windows/win32/secauthn/key-distribution-center. Accessed: 2024-10-01.

[63] Steffen Müller, David Bermbach, Stefan Tai, and Frank Pallas. 2014. Benchmarking the performance impact of transport layer security in cloud database systems. In *2014 IEEE International Conference on Cloud Engineering*. IEEE, 27–36.

[64] Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially private join queries over distributed databases. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 149–162.

[65] National Institute of Standards and Technology. 2016. *Recommendation for Key Management - Part 2: Best Practices for Key Management Organizations*. Technical Report SP 800-57, Rev. 1. NIST. https://csrc.nist.gov/pubs/sp/800/57/pt2/r1/final Accessed: 2024-10-04.

[66] B Clifford Neuman and Theodore Ts'o. 1994. Kerberos: An authentication service for computer networks. *IEEE Communications magazine* 32, 9 (1994), 33–38.

[67] OWASP:CryptographicStorageCheatSheet 2019. Cryptographic Storage - OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

[68] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[69] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[70] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. 2016. Big data analytics over encrypted datasets with seabed. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 587–602.

[71] paxdiablo. 2009. Answer to "Buffered vs unbuffered IO". https://stackoverflow.com/a/1450563/11215536

[72] Hieu Pham. [n. d.]. Remote Compactions in RocksDB-Cloud. https://rockset.com/blog/remote-compactions-in-rocksdb-cloud/

[73] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2016. Arx: A Strongly Encrypted Database System. *IACR Cryptol. ePrint Arch.* 2016 (2016), 591.

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

[75] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2012. CryptDB: processing queries on an encrypted database. *Commun. ACM* 55, 9 (2012), 103–111.

[76] Thomas Pornin. 2013. Answer to "Block chaining modes to avoid". https://security.stackexchange.com/a/27780/282524

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

[78] Eric Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. Request for Comments RFC 8446. Internet Engineering Task Force. https://doi.org/10.17487/RFC8446 Num Pages: 160.

[79] Rocksdb:LookupPerformance 2018. Improving Point-Lookup Using Data Block Hash Index. https://rocksdb.org/blog/2018/08/23/data-block-hash-index.html

[80] Rocksdb:TuningGuide 2023. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[81] @Scale. 2015. Data @Scale Seattle- Muthu Annamalai. https://www.youtube.com/watch?v=DfiN7pG0D0k

[82] Jindal K Shah, Eliseo Marin-Rimoldi, Ryan Gotchy Mullen, Brian P Keene, Sandip Khan, Andrew S Paluch, Neeraj Rai, Lucienne L Romanielo, Brian Rosch, Thomas W Yoo, and Edward J Maginn. 2017. Cassandra: An open source Monte Carlo package for molecular simulation. *Journal of Computational Chemistry* 38, 19 (2017), 1727–1739.

[83] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 69–87. https://www.usenix.org/conference/osdi18/presentation/shan

[84] 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, 762–771.

[85] SHIELD [n. d.]. SHIELD: Integrating Encryption to LSM-KVS in Disaggregated Data Centers with Minimal Performance Impact. https://github.com/asu-idi/SHIELD

[86] Erez Shmueli, Ronen Vaisenberg, Yuval Elovici, and Chanan Glezer. 2010. Database encryption: an overview of contemporary challenges and design considerations. *ACM SIGMOD Record* 38, 3 (2010), 29–34.

[87] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[88] ThalesGroup:SelectingRightEncryptionApproach 2020. Selecting the Right Encryption Approach. https://cpl.thalesgroup.com/encryption/selecting-right-encryption-approach

[89] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. *Proc. VLDB Endow.* 6, 5 (March 2013), 289–300. https://doi.org/10.14778/2535573.2488336

[90] Sean Turner. 2014. Transport Layer Security. *IEEE Internet Computing* 18, 6 (2014), 60–63. https://doi.org/10.1109/MIC.2014.126

[91] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2835–2849. https://doi.org/10.1109/ICDE55515.2023.00217

[92] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. 2018. JoiNS: Meeting Latency SLO with Integrated Control for Networked Storage. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Milwaukee, WI, 194–200. https://doi.org/10.1109/MASCOTS.2018.00027

[93] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: Compaction-as-a-Service for LSM-based Key-Value Stores in Storage Disaggregated Infrastructure. *Proc. ACM Manag. Data* 2, 3, Article 124 (may 2024), 28 pages. https://doi.org/10.1145/3654927

[94] Xingliang Yuan, Yu Guo, Xinyu Wang, Cong Wang, Baochun Li, and Xiaohua Jia. 2017. EncKV: An encrypted key-value store with rich queries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 423–435.

[95] Xingliang Yuan, Xinyu Wang, Cong Wang, Chen Qian, and Jianxiong Lin. 2016. Building an encrypted, distributed, and searchable key-value store. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 547–558.

[96] Qizhen Zhang, Yifan Cai, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. *Conference on Innovative Data Systems Research* (Jan. 2020). https://par.nsf.gov/biblio/10157860

[97] Yahui Zhang, Min Zhao, Tingquan Li, and Huan Han. 2020. Survey of Attacks and Defenses against SGX. In *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*. 1492–1496. https://doi.org/10.1109/ITOEC49072.2020.9141835

[98] Jay Zhuang. [n. d.]. Time-Aware Tiered Storage in RocksDB. https://rocksdb.org/blog/2022/11/09/time-aware-tiered-storage.html. Accessed 10 Jan, 2023.