

Hardware-assisted Secure Decentralized Cloud Storage via Self-audit and Self-repair

Josh Dafoe¹, Niusen Chen², and Bo Chen^{1*}

¹ Department of Computer Science, Michigan Technological University, MI, USA

² Department of Computer Science and Engineering, University of Wisconsin-La Crosse, WI, USA
`bchen@mtu.edu`

Abstract. Outsourcing data to the decentralized cloud has faced significant security concerns. The storage peers might be infected by malware, or motivated by other means to arbitrarily manipulate data. Most current decentralized cloud storage system designs rely on a blockchain to implement auditing via smart contracts and rely purely on storing redundant data across peers to repair corruptions. This results in significant overhead for executing the smart contracts, as well as substantial bandwidth consumption for performing repairs over time.

This work has introduced HiDCS, a Hardware-assisted integrity-assured Decentralized Cloud Storage system. HiDCS relies on two trusted components: 1) trusted execution environment which is a secure area of the processor, and 2) flash translation layer which is the firmware encapsulated in the flash storage hardware. Using these two components, HiDCS enables self-audit, while also explicitly preventing both outsourcing and generation attacks. Additionally, HiDCS enables self-repair, which attempts to fix corrupted data using only local resources, thereby eliminating the significant communication overhead typically associated with repairs over time. Security analysis and experimental evaluation justify the fact that HiDCS can ensure the reliability of outsourced data in the decentralized cloud with minimal performance impact.

Keywords: Decentralized Cloud Storage, Trusted Execution Environment, Flash Translation Layer, Self-audit, Self-repair

1 Introduction

Cloud storage allows data owners to outsource their data to cloud providers, releasing their burden on both data storage and management. There are two distinct cloud storage models that have been widely used today, namely a centralized and a decentralized model. The centralized model was the first to be implemented on a large scale. Under this model, the cloud service provider is a centralized entity (e.g. Dropbox, Google Drive, and Amazon S3) that maintains

* Corresponding author.

a dedicated storage infrastructure, which typically consists of multiple data centers in different geographic locations. This model has some inherent limitations. For example, the low number of data centers may result in speed limitations for clients distant from the servers and significant data inaccessibility during outages; the storage cost could be higher due to the required maintenance of expensive dedicated resources at scale. In contrast, the decentralized model features a collective infrastructure formed by a distributed network of autonomous storage servers, commonly known as “peers”. Anyone can join as a storage peer, providing storage resources to the network for a reward. Using spare resources in storage peers can reduce the need to maintain an expensive dedicated storage infrastructure, thus reducing the cost of using cloud storage [2]. In addition, a client has the ability to distribute data redundantly among multiple peers, selected based on the reliability metrics such as uptime. This enhances the resilience of the decentralized model, as data loss would require the simultaneous failure of multiple peers. Mainstream decentralized cloud providers include Filecoin [51], Storj [61], Sia [63], etc.

When outsourcing data to a cloud storage system, the client typically requires the assurance that the data are always intact and available. The need to provide a mechanism for this assurance has been extensively demonstrated [46, 45, 58, 62], which has resulted in many different data integrity schemes. However, in conventional centralized cloud systems, data centers are run by reputable entities such as Amazon, Microsoft, Google, who strictly follow their service level agreements (SLAs [8]) and are very unlikely to perform malicious behaviors. In the decentralized model, this data integrity problem is even more essential, as data are stored in heterogeneous storage peers which are less reputable and reliable, and may behave arbitrarily. For example, an economically motivated storage peer may discard data that are rarely accessed so that storage space can be reclaimed and resold [11]. Additionally, the operating system (OS) of a storage peer can be compromised by a remote hacker, who may arbitrarily manipulate the stored data to affect business processes, organizational understanding, and decision-making. Therefore, such a *data integrity* problem is of critical importance for the wide deployment of decentralized cloud storage systems.

The data integrity issue has been extensively investigated in the centralized cloud, assuming the existence of a trusted verifier (e.g., the client or a trusted third party) that can periodically check the integrity of data outsourced to a data center [11, 43, 10, 59]. Once the verifier finds any corrupted data, it will repair them [31, 16, 20]. In a decentralized environment, this trusted verifier isn’t inherently available. Consequently, existing designs [34, 51, 44, 63, 67] use blockchain to create the trusted verifier. A typical approach is to deploy a smart contract which is responsible for checking data integrity. The trustiness of the smart contract is ensured by its immutability, and the robust consensus mechanism at the core of the blockchain. Unfortunately, this blockchain-based approach suffers from a few drawbacks. First, it is highly inefficient. Transactions and smart contracts are stored, run, and verified by each miner on the blockchain [17]. This introduces a very large overhead in both computation and storage, and

requires frequent coordination between peers. Second, it is highly inflexible. A smart contract, once committed to the blockchain, cannot be updated due to its immutability. This implies that the checking and repairing functionality cannot be updated once it is built and committed to the chain.

This work investigates a new design that can maintain the integrity and reliability of data outsourced to a decentralized cloud, without depending on blockchain and smart contracts. Our design overcomes the problem of distrust by leveraging two trusted components: 1) *Trusted Execution Environment (TEE)*: Many modern processors are equipped with TEEs such as Intel SGX and AMD SEV. The TEE enables a critical application to run within an isolated memory area, so that critical data and execution can remain protected even if the operating system is compromised. 2) *Flash Translation Layer (FTL)*: The FTL is a firmware layer built into mainstream solid state drives, an increasingly popular flash memory medium that has occupied more than half of the external storage market since 2021 [6]. The FTL is isolated from the OS by the storage hardware, so that even if the OS is compromised, the FTL can remain intact.

Our resultant design is a **H**ardware-assisted integrity-assured **D**ecentralized **C**loud **S**torage system (**HiDCS**). This is the first design of a decentralized cloud storage system capable of continuously ensuring the integrity and reliability of outsourced data, with almost no communication between storage peers. At its core, HiDCS leverages TEE and FTL, and develops two innovative functions: self-audit and self-repair.

Self-audit. Towards designing a secure decentralized cloud system, we first need an auditor who can periodically check the integrity of the outsourced data. We establish this trusted auditor using the TEE equipped within each peer. Such a self-auditing design releases the burden on the data owner and eliminates unnecessary external communication during the audit. In addition, we use a *spot checking* technique [11, 43] to keep the auditing process light-weight, which randomly checks a small subset of the entire data during each audit. Although efficient, its probabilistic nature makes it susceptible to *outsourcing and generation attacks* [14], in which the adversary can retrieve or regenerate challenged data on the fly to pass the integrity check without possessing actual data. For the first time, we effectively address these attacks by collaborating the TEE and FTL. Our key insight is that the trusted FTL directly manages the low-layer data storage. By involving the FTL during the integrity check, the auditor can easily detect whether the challenged data are retrieved from local storage or not, essentially mitigating the outsourcing / generation attacks.

Self-repair. Once the corruptions are detected, the next step would be to quickly repair them. Existing works [31, 16, 20] rely on redundant data distributed to other storage servers to repair. This will lead to substantial consumption of communication bandwidth over time. In contrast, we propose a novel self-repair mechanism, aiming to repair corruptions while relying on the resources available in the local peer. Although spot-checking is efficient, it only detects large corruptions due to its probabilistic nature [11]. We therefore propose two repair

strategies: one for large corruptions that spot-checking can detect, and another for small corruptions that spot-checking cannot detect reliably.

1) Upon detecting a large corruption using spot checking, we will repair it by exploiting the hardware nature of flash memory. Our key observation is that, to accommodate the special hardware nature of flash memory, the FTL typically performs out-of-place updates. Therefore, a large corruption attack conducted by the adversary can only invalidate rather than delete the original flash memory data. Therefore, by manipulating the garbage collection policy implemented by the FTL, these data can be preserved until corruption is detected. Repairing detected corruptions can be performed efficiently by rolling back the original data before they are permanently deleted.

2) For small corruptions that remain undetected by spot checks, we can mitigate them by encoding outsourced data using error correction codes (ECC). This enables the restoration of small corruptions upon retrieving data from the peer. However, applying the ECC is not straightforward in a cloud peer with an untrusted operating system. Applying a single ECC to all data will work, but it is highly inefficient for both encoding and decoding [30]. An efficient alternative would be to divide all the data into smaller “stripes” and apply an ECC to each “stripe”, generating a separate code word for each one. However, this is vulnerable to a small corruption attack in which the adversary can simply corrupt a few symbols in a code word, rendering the code word undecodable. Such a small corruption is hard to detect via spot checking and will likely go unrepaired. Fortunately, this small-corruption attack is only possible when the adversary can discover the association between the symbols and the corresponding code word. Therefore, to prevent this attack, we can randomize the association using a secret key only known to TEE and FTL. Additionally, we must prevent the untrusted OS from learning the association by monitoring disk access during ECC encoding and decoding. We protect access efficiently by collaborating TEE with FTL residing inside the disk, eliminating the need for expensive cryptographic primitives such as oblivious RAM [39]. Lastly, to prevent the untrusted OS from corrupting parity symbols, they are integrity verified by the FTL before being stored and made invisible to the OS after storage.

3) In the unlikely event that corruption cannot be restored locally, redundantly distributed data from other peers can be retrieved as a last resort.

Contributions. Our contributions are summarized below.

– We have proposed the first self-audit and self-repair design for decentralized cloud storage. Compared to existing decentralized cloud storage systems, HiDCS is the first secure decentralized cloud storage system design that relies first on the existing trusted components within storage peers. A key advantage of our design is the ability to audit and restore data locally, which significantly reduces the computation / communication needed to maintain the reliability of outsourced data over time.

– We are the first to utilize the low-layer flash translation layer to effectively and efficiently address the outsourcing / generation attacks, a hard problem faced in the literature of cloud outsourcing.

– We have implemented HiDCS in real-world devices using an Intel SGX and open source flash controller, and experimentally validated the efficiency of HiDCS.

2 Background

Trusted execution environment (TEE). Many modern processors are equipped with TEE features such as Intel Software Guard Extensions (SGX [29]) and Trusted Domain Extensions (TDX [26]), AMD Secure Encrypted Virtualization (SEV [1]) and its enhancements SEV-ES and SEV-SNP, and Apple Secure Enclave [5], etc. TEE allows a critical application to run inside a secure memory area (e.g. an SGX enclave), in which critical data and execution are isolated at the hardware level. This allows critical execution to be protected even if the OS is compromised. Hence, the confidentiality and integrity of content and code are protected from any process outside the TEE. A TEE-enabled application typically consists of two parts, a trusted application (TApp) running in the secure memory area and an untrusted application (UApp) running on the main OS. Additionally, there is an interface for UApp to call predefined functions in TApp (e.g. ECALL in SGX) and, conversely, for TApp to call predefined functions in UApp (e.g. OCALL in SGX). Essentially, TEE also provides remote attestation to convince a third party that the correct code is actively running in the TEE. Attestation allows the establishment of a shared key between the TEE and the third party [29] through a secure key exchange protocol such as Diffie-Hellman [32]. Using this key, an encrypted communication channel is established between the TEE and an attesting third party, by which sensitive data can be securely transmitted to the enclave. TEE also supports sealing, which allows encrypting and binding of data (such as keys) to a specific TEE instance.

Flash translation layer (FTL). Unlike traditional hard disk drives, solid-state drives (SSDs) offer significantly faster disk access. SSDs now account for more than half [6] of external storage in modern computers, and their usage continues to grow rapidly. The FTL is a firmware layer built into SSDs to manage the unique hardware characteristics of flash memory. The storage space on a typical SSD is segmented into blocks, each of which consists of pages. The read/write operation of flash memory is performed on pages, while the erase operation is performed on blocks. Erase operations wear down the associated block, which becomes unusable when some threshold is met. Therefore, flash memory typically uses out-of-place updates, instead of in-place ones, to reduce the number of required erasures. FTL manages this out-of-place update policy by maintaining mappings between logical addresses and their evolving physical locations. FTL also implements other functions such as garbage collection and wear leveling. Garbage collection periodically reclaims the space occupied by obsolete data, while wear leveling evenly distributes data across the flash to avoid premature wear. Essentially, we use the FTL in our design because the FTL is *isolated* from the OS by the storage hardware, so that even if the OS is compromised, the FTL can remain intact. This hardware-level isolation ensures the security of the computation performed in the FTL even if the OS is compromised. Additionally, the FTL presents only a simple read/write interface to the OS and thus has a

very limited attack surface. Due to its isolated nature, FTL has been explored for plausibly deniable storage [42, 22], malware defense [41, 13, 66], and secure deletion [24]. We are the first to incorporate FTL for cloud storage auditing.

Remote data integrity checking. Remote data integrity checking provides an integrity guarantee of the data outsourced to an untrusted third party. Provable data possession (PDP) [11, 10, 35, 19] or proofs of retrievability (PoR) [43, 57] can be used to efficiently verify the integrity of outsourced data. The core idea is to check a subset of data blocks *randomly* instead of checking the entire large outsourced data. It has been shown [11, 10] that by checking a small number of data blocks, the “*spot checking*” technique can detect data corruptions with an arbitrarily high probability if the adversary corrupts a certain proportion of the data. To support integrity checking, outsourced data is viewed as a collection of blocks and an integrity verification *tag* will be calculated for each block. Note that the tag can be constructed in a publicly verifiable [11] manner, so that any third party can act as an integrity auditor. The tags and data are then outsourced to the cloud server. This allows an auditor to issue challenges to the server, requesting proof that a random subset of the data are stored correctly. In response to this challenge, the server computes a proof based on the challenge, the data, and the associated tags. The auditor can then check the proof.

Outsourcing attacks and generation attacks. Traditional remote data integrity checking schemes [11, 43] are vulnerable to outsourcing and generation attacks [14, 51]. In both attacks, the adversary attempts to pass an integrity check without actually possessing the data. In the *outsourcing attack*, the challenged storage peer quickly retrieves the data from another peer to produce the proof, pretending to store the data all along. In the *generation attack*, the storage peer claims to store the data, but actually generates the data on demand using a small program. Both attacks are concerning for data owners, as they convince them that the data is reliable even when it is not. In particular, the outsourcing attack moves the data further away from the expected peer, which may take longer to access and will slow down the system. This may also violate retention regulations like GDPR which require the data to be stored in certain geographical locations. Finally, this could increase centralization, as many storage peers could outsource data to the same server.

Error-correcting code (ECC). An error-correcting code allows repairing errors in a message by encoding it redundantly. An (n, k) ECC, with $n \geq k$, takes a message of k symbols and encodes it into a code word of n symbols. The $n - k$ symbols that are added to the original message are known as *parity* symbols. This encoding is done so that if any k out of the n symbols are intact, the entire original data can be restored. Thus, it can correct up to any $n - k$ corrupted symbols. Performing this correction involves running a decoder function over all the data in the compromised code word. However, a code word is irreparable if $n - k + 1$ or more symbols are corrupted. Note that we use a systematic code, in which the first k symbols in any code word are the original k symbols.

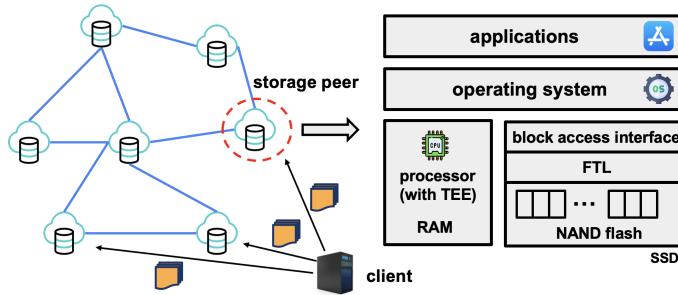


Fig. 1: System model.

3 Models and Assumptions

System model. We consider a decentralized cloud storage system that is made up of numerous storage peers (see Figure 1). The client (data owner) outsources the data to multiple storage peers, a replica in each peer [31]. Each storage peer is a computing device (e.g., a laptop, a desktop, a server computer) owned by an individual or an organization who wants to join the storage network and provide storage services. The computing device is equipped with a processor, RAM, as well as an SSD as external storage. The SSD is managed by an FTL which provides a read/write interface to the OS. The FTL is run on the hardware of the SSD, which is isolated from the OS. Additionally, the processor is equipped with a TEE. This TEE can be any of the options available (as mentioned in Section 2, including SGX, TDX, SEV, Apple Secure Enclave, etc.), enabling a wide range of various hardware support. We consider a trusted application (**TApp**) running in the secure world of TEE which can communicate directly with an untrusted application (**UApp**) running within the untrusted OS. The **UApp**, in turn, can directly access the read/write interface provided by the SSD.

Adversarial model. The storage peers may misbehave. Entities that manage peers may be economically motivated [11], discarding a portion of stored data to save space without the data owner’s awareness (Adversary I). In addition, the operating system of each storage peer may be compromised and behave incorrectly (Adversary II). This is likely to happen in the context of a decentralized cloud, given that storage peers are often managed by individuals who may not adequately secure their devices. For example, the storage peer may be compromised and controlled by a remote hacker or infected by malware that can obtain root privileges. The adversary has controlled the read/write interface, *and* can arbitrarily manipulate the communications between **TApp** and FTL. The main adversarial behaviors include: 1) corrupting a large portion of the data stored in the peer (i.e. *a large corruption attack* [11, 43]); and 2) corrupting a small portion of the data, for example, $n - k + 1$ symbols in an ECC code word, rendering the data irrecoverable (i.e. *a small corruption attack* [10]); and 3) performing outsourcing attacks or generation attacks to pass the integrity check without actually possessing the data [14, 51]. Note that: 1) Both Adversary I and Adversary II are computationally bounded. 2) We do not consider DoS attacks.

Therefore, **UApp** will not refuse to act as a communication proxy between **TApp** and FTL or between **TApp** and remote entities.

Assumptions. We make some assumptions. 1) TEE is secure (i.e., **TApp** is secure). This is a reasonable assumption for secure applications based on TEE [50, 54]. Although various attacks have been found against TEE, for example, the Prime + Probe attack [56], the microarchitectural replay attacks [60], the improvement of TEE security has been actively taken care of in the literature [36] and is outside the scope of this work. 2) The FTL is secure. This assumption has been used broadly in secure FTL applications [41, 13, 64]. The security of FTL is mainly based on the isolation between the flash memory controller and the host operating system, as well as the minimal interface provided by the FTL to the OS. To ensure that each peer runs the trusted FTL, existing SSDs are equipped with certificates that can authenticate the FTL sources. Upon each boot, a secure boot can be performed in the bootloader so that only the trusted FTL code can be loaded into memory [27]. 3) The TEE remote attestation mechanism can always be used by the **TApp** to initiate secure communication (Sec. 2) with an external third party, even when malware is present in the host OS.

In addition, we assume the existence of a shared master secret key K_s between **TApp** and FTL (Appendix A discusses the potential key sharing techniques).

4 HiDCS

To ensure the long-term integrity and reliability of the data outsourced to the decentralized cloud, each storage peer should correctly maintain its own replica. In HiDCS, this goal is mainly achieved by “local repair”, in which each peer periodically audits the stored data and, if corruption is found, HiDCS will repair the corrupted data using the available local resources. To ensure the security of local repair, we need to mitigate a few attacks, and the corresponding mitigation strategies are introduced in Sec. 4.1. In the worst case, if the corrupted data cannot be restored by local repair, the storage peer will collaborate with other peers, retrieving the redundant data for repair (we call this “distributed repair”, as described in Sec. 4.2). In this paper, we focus mainly on the local repair and its design details are further elaborated in Sec. 4.3.

4.1 Secure Local Repair

In HiDCS, **TApp** will run a periodic audit over the data stored on the local SSD (i.e. self-audit) and once any corruption is found, it will collaborate with the FTL for repairing (i.e. self-repair). Any small corruptions that are undetectable by the auditor will be handled by the ECC computed over the data. Essentially, the outsourcing / generation attacks, the large corruption attack, as well as the small corruption attack should be mitigated towards a secure local repair design.

I: Mitigating outsourcing / generation attacks.

Outsourcing and generation attacks are hard to mitigate in typical cloud outsources [14]. Conventional remote data integrity checking (RDIC) schemes [11,

43, 57, 35] only verify data integrity, but cannot bind the data to a certain peer. Some enhanced RDIC designs take advantage of slow cryptographic operations [20, 14] or verifiable delay functions [37] to impose significant resource restrictions on the generation of data on the fly. Although this can discourage the adversary from performing the outsourcing / generation attacks, it will add a large overhead to the data owner when preprocessing the outsourced data. Our key observation is that previous works [20, 14, 52, 9] address these attacks in the upper layer of a storage system, making defenses cumbersome. In contrast, we combat both attacks from a new perspective, by leveraging the FTL, which is essentially staying at the lower storage layer. As the FTL is integrated with the SSD, it is also bound together with the storage peer. Therefore, by collaborating with FTL, TApp can verify that the challenged data genuinely originate from the local SSD, rather than from a remote peer (i.e., outsourcing attack) or are generated on the fly (i.e., the generation attack).

Our self-auditing function works as follows: Periodically, TApp sends a challenge (with a fresh nonce) to the untrusted OS, asking to check a random subset of outsourced data stored on SSD. UApp will then issue read requests, retrieving the challenged data from the SSD, together with the associated verification tags³. Upon sending the requested data to UApp, FTL will embed the challenged data using some secret, which can be used to verify that the data actually come from SSD. Specifically, the FTL will encrypt the challenged data using a secret key shared with TApp (it is derived from the master secret key K_s and the nonce). TApp will perform a decryption prior to verifying the correctness of the challenged data. Therefore, if the challenged data are not from SSD, the integrity check always fails as the adversary does not have the secret key. Note that we only check a small amount of data each time, and the computation for encryption is small.

II: Mitigating the large corruption attack.

If a large portion of the outsourced data is corrupted in the local peer, our self-auditing function will efficiently detect this corruption with high probability [11]. Our next step will be collaborating TApp and FTL to repair this corruption. To corrupt a portion of the outsourced data, the adversary needs to overwrite the data at the OS level. Different from the OS, the FTL performs out-of-place updates (Sec. 2). Therefore, an overwrite performed by the OS will result in new data being written to a different flash memory location, while the old data are preserved in their original location. This original location is marked as invalid by the FTL but will not be reclaimed by garbage collection immediately. Thus, one may repair large corruptions by extracting the “old data” from the flash memory. This requires tuning the garbage collection strategy in the FTL so that “old data” are always available before repairs are made.

HiDCS targets static data; that is, there are infrequent updates on outsourced data. Therefore, an active garbage collection (GC) strategy becomes unnecessary.

³ To release the client’s burden on generating the integrity verification tags (Sec. 2), TApp is responsible for generating them. The client can establish a secure communication channel with the TApp via the attestation, and send the data to TApp.

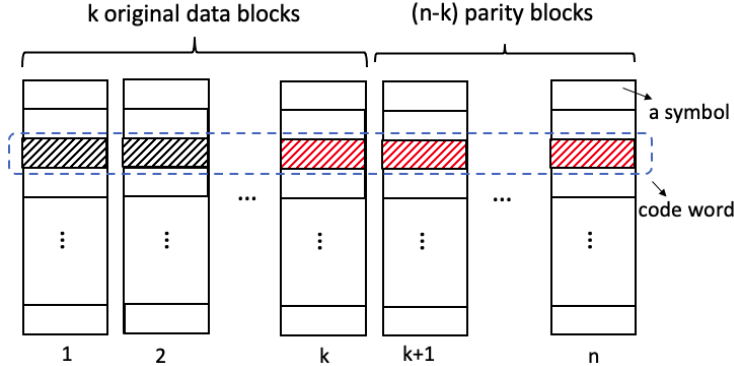


Fig. 2: Compute the parity for a chunk. k original data blocks are encoded via ECC, generating $n - k$ parity blocks. For efficiency [49], the (n, k) ECC is computed over each collection of k symbols, which are located at the same row of the k original data blocks in the chunk.

We instead use a lazy GC strategy guided by self-repair. Specifically, the garbage collection remains frozen until: 1) a self-repair of large corruptions is completed, or 2) the disk is filled. For Case 1, a large-corruption repair has already taken place, ensuring that none of the remaining invalid data is needed. Therefore, GC can be performed immediately and then frozen again. For Case 2, corruptions may be present across the entire data and, we can perform a one-time integrity check. If there are no corrupted data, the GC is performed immediately and frozen again; otherwise, the corrupted data should be repaired by extracting the “old data” before the GC can be performed.

III: Mitigating the small corruption attack.

Due to its probabilistic nature (i.e., checking a random subset of data blocks), the spot checking technique adapted by TApp in the self-audit can only detect large corruptions reliably. Consequently, if the size of corrupted data is sufficiently small, spot checking will not detect it, and the corrupted data may eventually be reclaimed by garbage collection and become irrecoverable locally. To remediate this, each storage peer can generate parity data locally, enabling recovery of undetected small corruptions upon retrieval. This is done by applying an (n, k) ECC to the data [30].

The data symbols for an ECC are typically from a small finite field (e.g., a few dozen bits), in order to support efficient arithmetic operations. To be consistent with the spot-checking technique that performs integrity checking over data blocks (each is usually a few kilobytes in size), we view each data block as a collection of symbols. Thus, we partition the original data D , consisting of $|D|$ blocks, into $r = \lfloor \frac{|D|}{k} \rfloor$ chunks, each of which is a collection of k blocks. We then apply a (n, k) ECC on each block, generating $n - k$ parity blocks (see Figure 2). Each resulted ECC-coded chunk (a code word) therefore consists of the k data blocks and the $n - k$ parity blocks. Importantly, if the adversary can learn the “association” between the data symbols and the respective code word, this design is vulnerable to a small corruption attack. In this attack, the

adversary simply corrupts $(n-k+1)$ small symbols from the same code word (see the red-patterned symbols in Figure 2). Such corruptions are tiny, not repairable via ECC decoding, but hard to detect via spot checking. To mitigate the small corruption attack, we should hide the “association” between the data symbols and their corresponding code words, by randomizing [30] the assignment of the k original data blocks to each ECC-coded chunk via a pseudo-random permutation. Specifically, we first associate the i -th group of k contiguous *indices* with the i -th chunk. Then, we perform a pseudo-random permutation of the indices in a given chunk based on the secret key K_D , and assign the blocks associated with the new indices to that chunk. For example, if $k = 2$, the first chunk is assigned two indices 1 and 2; the permutation provides the mapping $1 \mapsto 7, 2 \mapsto 4$, and we assign blocks 7 and 4 to this chunk instead of blocks 1 and 2. In addition, HiDCS introduces two unique strategies to mitigate the small corruption attack:

1) To avoid adding large system components along the storage path to the small TEE, the **TApp** needs to rely on **UApp** to access the SSD. In addition, the TEE usually processes a few ECC-coded chunks instead of the entire data at a time, as a result of the limited amount of available memory. Therefore, there is an access pattern leakage attack in which, upon observing disk accesses during encoding or decoding, the adversary can determine which blocks belong to the same code words, compromising the “association”. Our solution is to obfuscate the disk access during encoding/decoding. Specifically, an obfuscation mode is introduced and **TApp** can inform the FTL to temporarily activate this upon encoding/decoding. In this mode, before requesting a block at the disk location i , it will be obfuscated (i.e. permuted) as a disguised disk location j . Thus, **UApp** can only observe the request at disk location j , so it performs a read request at this location. However, the FTL can derive i from j , based on the shared secret key (by computing the inverse permutation). The FTL will read and return the data from the actual location i , even though **UApp** requested the data at the location j . In addition, the adversary might observe the content of the returned data and identify the actual location i . To prevent this, FTL will encrypt the returned data using a shared key and send the encrypted data back to the **TApp**. A concrete example is shown in Figure 3. In addition, to prevent the **UApp** from corrupting data blocks in transit, integrity verification tags should be accompanied and used to verify data blocks before the **TApp** uses them for encoding/decoding.

2) The attacker may compromise the “association” if the parity blocks are plainly accessible [30]. Therefore, we encrypt the parity blocks before committing them to the SSD via the **UApp**. In addition, once the parity blocks are committed to the SSD, we disable the writes over them, leveraging the low-level control of FTL. This can prevent the adversary from corrupting any parity symbols towards a successful small corruption attack. Additionally, since no integrity tags are generated for the parity, we must ensure that the parity blocks are correctly transferred to the SSD by the **UApp** (and vice versa). The **TApp** can compute a MAC code over the parity blocks. However, this could be expensive for the FTL to verify. For better efficiency, we rely on checking some “sentinel bits” [43] among the encrypted parity blocks. If any of the sentinel bits are corrupted, it indicates

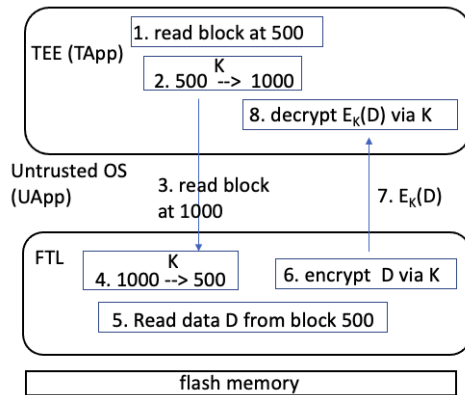


Fig. 3: Securely read a data block in the obfuscation mode.

that the parity blocks are corrupted. Specifically, **TApp** will select a random set of sentinel bits, determined using a fresh secret key (generated from the master secret key K_s and a nonce), and sign the selected sentinel bits. The resulting signature will be sent to the **FTL** together with the encrypted parity blocks and nonce. After having regenerated the secret key (using K_s and nonce), **FTL** can extract the sentinel bits from the encrypted parity blocks and check if they are correct or not using the signature. If the check is passed, it is very likely that there are no corruptions among the encrypted parity blocks. The same procedure is done when sending parity to **TEE** for decoding.

4.2 Secure Distributed Repair

Our auditor relies on efficient spot checks, which only provide a probabilistic guarantee. Therefore, there is a very low likelihood that corrupted data is not detected and therefore cannot be repaired timely using the “old data” temporarily preserved in the flash memory. In addition, this corruption may be large enough that it exceeds the recovery capability of ECC. For such corner cases (which are of low probability as analyzed in Appendix C), **TApp** can initiate communication with other peers, which store replicas. Note that a replication-based distributed storage system usually requires an implied assumption that at least one replica is intact in the system. To avoid retrieving an entire replica, the peer can audit the entire data locally, identifying corrupted blocks, and retrieving them remotely. A potential security concern is how the peer ensures that the data blocks that are being retrieved are correct. To address this concern, **TApp** can communicate with the **TApp** (We call it **TApp-1**) running in another peer through a secure channel, established through attestation. The **TApp-1** reads the blocks from its local SSD, verifies them, and sends them to **TApp** through the secure channel.

Optionally, if the failed peer turns really unreliable (e.g., the audit fails frequently), the client can be involved such that a new storage peer can be found to store the recovered data.

4.3 Design Details of Local Repair

Definitions. We define the following variables:

- **Block (b_j):** Each block b_j is an ordered collection of pages. A block can be represented as $b_j = \{p_0, p_1, \dots, p_{m-1}\}$, where m is the number of pages in block b_j . The access of bit l in a given block is addressed as $b_j[l]$ ($p[l]$ to access bit l of a given page).
- **Data (D):** Data D are an ordered collection of blocks, $D = \{b_0, b_1, \dots, b_{|D|-1}\}$, where $|D|$ is the number of blocks in D . Let $D = P_0 \cup P_1 \cup \dots \cup P_{r-1}$ be a partitioning of D , where r is the number of partitions of D . The partitions correspond to the ECC coded chunks introduced in Sec. 4.1.
- **Tags (T):** A set of verification tags for performing audits, where $T = \{t_0, t_1, \dots, t_{|D|-1}\}$ is an ordered collection of tags. For $0 \leq j < |D|$, the tag t_j is associated with the block b_j in D .
- **Extended Data (\tilde{D}):** $\tilde{D} = D \cup T$. Additionally, $\tilde{D} = \tilde{P}_0 \cup \tilde{P}_1 \cup \dots \cup \tilde{P}_{r-1}$, where $\tilde{P}_a = P_a \cup T_a$, and T_a is the set of verification tags for the blocks in P_a , where $a \in \mathbb{Z}_r$ is the partition number.
- **Parity Data (λ_a):** This is the parity resulting from encoding a partition P_a through an ECC. Each λ_a is a set of parity blocks.
- **Key Length (κ):** The length of all keys.
- **Shared key (K_s):** The master secret key shared between TApp and FTL, which is used to generate ephemeral keys.
- **Data key (K_D):** A key associated with data D . This is stored only in TApp.
- **Nonce (N):** Typically a randomly generated value, used to generate ephemeral keys shared between TApp and FTL.
- **Counter Variable (γ):** A counter local to TApp that is incremented each time it is used. It is for random number generation.
- **Arbitrary Constant (d):** Represents an arbitrary constant.

We also define 10 functions used in the design: 1) $f_{(l,K)}$ is a keyed pseudo-random function (using the key K), which takes in arbitrary data and produces a random l -bit output. 2) $\pi_{(l,K)}$ is a keyed pseudo-random permutation. It is a bijective function that takes an input of l bits and produces a random output of l bits. 3) $h_{(l,K)}$ is a hash function that takes arbitrary data and produces an l -bit output. This is used to produce signatures and ephemeral key generation. 4) Enc_K is an encryption function with key K . 5) Dec_K is a decryption function with key K . 6) **Audit** is an auditing function, taking in some $A \subseteq \tilde{D}$ so that each $b_j \in A$ has its corresponding $t_j \in A$. **Audit** returns 0 or 1 to indicate the integrity of A . 7) **Encode** is an ECC encoding function that takes a set of blocks and encodes it (Figure 2). 8) **Decode** is an ECC decoding function. 9) g is a partitioning function that takes a block number i and returns its partition number $a \in \mathbb{Z}_r$ such that $i \mapsto \pi_{(\log_2(|D|), K_D)}(i) \bmod r$. 10) **Convert** is a function that converts an ordered set of blocks into a vector ordered by the block number.

Initialization. After the storage peer has received a replica from the client (through a secure channel established after attestation), TApp runs Algorithm 1

Algorithm 1 Initialization

Require: $D, n, r, d, l, \gamma, \kappa, K_s, K_D$ **Ensure:** s

- 1: $s \leftarrow 1$
- 2: $\tilde{D} \leftarrow \text{GENTAGS}(D)$
- 3: $\text{STOREDATA}(\tilde{D})$
- 4: $\{N, K_p, \mathcal{I}^*\} \leftarrow \text{PARTITIONDATA}(n, r, d, \gamma, K_s, K_D)$
- 5: **for** $a = 0$ to $r - 1$ **do**
- 6: $\tilde{P}_a^* \leftarrow \text{GETPARTITION}(N, I_a^* \in \mathcal{I}^*, K_s)$
- 7: $\{s, \lambda_a^*\} \leftarrow \text{ENCODEPARTITION}(\tilde{P}_a^*)$
- 8: $\{N_a, K_a, M_a\} \leftarrow \text{GENPARITYPROOF}(\tilde{\lambda}_a^*, l)$
- 9: $(v, v_s) \leftarrow \text{STOREPARITY}(N_a, l, \lambda_a^*, h_{(\kappa, K_p)}(M_a))$
- 10: $s \leftarrow \text{VERIFYSTORAGE}(v, v_s)$
- 11: **return** s

Algorithm 2 SelfAudit

Require: $d, \gamma, K_s, k, n_{seg}, n, c$ **Ensure:** s

- 1: $\{N, K_c, S\} \leftarrow \text{GENCHALLENGE}(d, \gamma, K_s, k, n_{seg}, n, c)$
- 2: $A^* \leftarrow \text{INITIATEAUDIT}(N, S, K_s, k)$
- 3: $s \leftarrow \text{VERIFYAUDIT}(S, A^*, K_c)$
- 4: **return** s

as follows: **TApp** first computes the verification tags T using **GENTAGS** (Appendix B.1). T will be concatenated with the replica, and both are stored using **STOREDATA** (Appendix B.1). To generate parity data which can facilitate small corruption repair, **TApp** partitions the data by randomly selecting the blocks in each of the r partitions (**PARTITIONDATA**, as elaborated in the Appendix B.1), then iterates through each partition, retrieving the data while hiding its content and location from **UApp** (**GETPARTITION**, as elaborated in the Appendix B.1). The parity for each partition will be calculated by **TApp** using **ENCODEPARTITION** (Appendix B.1) and the corresponding sentinels are calculated using **GENPARITYPROOF** (Appendix B.1). The parity of each partition will be stored by the **FTL** using **STOREPARITY** (Appendix B.1) and **TApp** can verify whether each parity is correctly stored or not using **VERIFYSTORAGE** (Appendix B.1).

Self-audit. The self-audit uses Algorithm 2. **TApp** first issues a challenge through **GENCHALLENGE** (Appendix B.2) that selects a random subset of blocks among the replica (together with the corresponding verification tags), verifying their integrity. The challenge will be passed on to **UApp**, who will retrieve the corresponding blocks from **FTL**. The **FTL** will encrypt the challenged blocks via a shared ephemeral key K_c , ensuring that the data come from the **FTL** (**INITIATEAUDIT**, elaborated in Appendix B.2). The **TApp** then decrypts the challenged blocks and verifies their correctness by running **VERIFYAUDIT** (Appendix B.2).

Self-repair. To repair a small corruption, Algorithm 3 is run as follows: **TApp** reads the data and parity of that partition and decodes over it. To read the

Algorithm 3 SmallCorruptionRepair

Require: $n, r, d, l, \gamma, \kappa, K_s, K_D, \text{blockNum}, |\lambda_a^*|$ **Ensure:** \tilde{P}_a

- 1: $a \leftarrow g(\text{blockNum})$
 - 2: $\{N, K_p, \mathcal{I}^*\} \leftarrow \text{PARTITIONDATA}(n, r, d, \gamma, K_s, K_D)$
 - 3: $\tilde{P}_a^* \leftarrow \text{GETPARTITION}(N, I_a^*, K_s)$
 - 4: $\lambda_a^* \leftarrow \text{READPARITY}(a, |\lambda_a^*|)$
 - 5: $\{N_a, K_a, M_a\} \leftarrow \text{GENPARITYPROOF}(\lambda_a^*, l)$
 - 6: $\tilde{P}_a^* \leftarrow \text{DECODEPARTITION}(N_a, r, l, K_p, \lambda_a^*, \tilde{P}_a^*, h_{(\kappa, K_a)}(M_a))$
 - 7: $\tilde{P}_a \leftarrow \text{Dec}_{K_p}(\tilde{P}_a^*)$
 - 8: **return** \tilde{P}_a
-

data, GETPARTITION (Appendix B.1) is invoked. Then FTL will read this parity into memory by invoking READPARITY (Appendix B.3), and when reading the parity, FTL must assure TApp that it is correct, extracting sentinels by invoking GENPARITYPROOF (Appendix B.1). The parity and integrity proof is sent to TApp, which will decode it via DECODEPARTITION (Appendix B.3). After decoding, it writes the restored data back. To repair a large corruption, FTL runs RESTOREMAPPINGS (Appendix B.3) to re-associate the corrupted locations with the good overwritten data.

5 Security Analysis and Discussion

5.1 Security Analysis

In the following, we show that HiDCS can resist outsourcing / generation attacks, large corruption attacks, and small corruption attacks.

HiDCS can resist outsourcing / generation attacks. The TApp checks a random subset of c blocks during each audit. Each block will be encrypted by the FTL via a unique secret key (K_c) of length κ . Thus, if UApp can acquire K_c , it can successfully pass the audit by performing outsourcing / generation attacks. However, the adversary does not have access to K_c . Therefore, its best strategy is to randomly guess K_c with probability $\frac{1}{2^\kappa}$ and encrypt the challenged data (fetching from another peer or being generated on the fly) via this random key. Upon receiving the ciphertext, TApp will use K_c to decrypt it. The probability that the ciphertext is correctly decrypted will be similar to that of generating K_c , which is $\frac{1}{2^\kappa}$. Therefore, the probability of successfully performing outsourcing/generation attacks via key guessing is no better than $\frac{1}{2^\kappa}$, which is negligibly small when κ is sufficiently large.

HiDCS can restore large corruptions. TApp checks a random subset of blocks periodically. For each audit, the probability of detecting data corruption is at least $P_{\text{detect}}^1 = 1 - \left(\frac{|D|-t}{|D|}\right)^c$ [11], where $|D|$ is the total number of blocks in the data D , t is the number of blocks corrupted by the peer, and c is the number of blocks challenged. $P_{\text{detect}}^1 = 1 - \left(\frac{|D|-t}{|D|}\right)^c = 1 - (1 - \beta)^c$, where $\beta = \frac{t}{|D|}$ is

the proportion of corrupted data. If $t \ll |D|$, $\beta \rightarrow 0$, $1 - (1 - \beta)^c \rightarrow 0$. This explains why spot-checking is hard to detect small corruptions. However, if β is sufficiently large, the probability would be high. For example, if $\beta = 0.01$, when checking 1000 blocks, the detection probability will be 0.999957; when checking 2000 blocks, the detection probability is 0.999999998. To incorporate the number of audits α , we can introduce the more general $P_{\text{detect}}^\alpha = 1 - (1 - P_{\text{detect}}^1)^\alpha$.

Once the large corruption is detected (with very high probability as analyzed before), it can be repaired by extracting the invalidated data preserved in the raw flash memory. Therefore, HiDCS can guarantee that the large corruption, once detected, is always recoverable. Very unlikely, the large corruption was not detected and cannot be repaired locally. HiDCS can manage this rare failure of local repair by resorting to distributed repair upon data retrieval (more details are analyzed in Appendix C).

HiDCS can restore small corruptions. The small corruption is repaired by the (n, k) ECC code computed on the outsourced data, in which the n and k should be picked so that the small corruption that is undetectable by spot check can be repaired. The only attack is that the adversary may undermine the correctability of any ECC code word by corrupting as few as $n - k + 1$ symbols in the code word, making the data in the code word irrecoverable. This small corruption is hard to detect by spot checking and hence cannot be repaired by extracting the invalid data from the raw flash. HiDCS has made the parity symbols invisible to the untrusted OS after correctly storing them, and therefore the adversary can only choose to corrupt $n - k + 1$ data symbols. This requires the adversary to associate at least $n - k + 1$ data symbols with the respective code word, which is infeasible as analyzed in the following two cases:

Case 1: the adversary tries to associate the $n - k + 1$ data symbols with its code word by reading the disk. However, the adversary has access to all the $r \cdot k$ data blocks on the disk, but can only guess a correct association without having access to the secret key K_D . To achieve correct guessing, the adversary must find $n - k + 1$ blocks in a given chunk (see Figure 2). The probability of a correct

guess, given the η trials, will be $P_{\text{guess}} = 1 - \prod_{j=1}^{\eta} \left(1 - \frac{r \cdot \binom{k}{n-k+1}}{\binom{r \cdot k}{n-k+1} - (j-1)} \right)$. For

example, when $\eta = 1$, $n = 140$, $k = 128$, $r = 5$, the probability is $3.85 \cdot 10^{-12}$. A larger r will decrease the probability. Changing η to a much larger number, for example, 1,000,000, the probability is still as small as $3.85 \cdot 10^{-6}$. Keeping η as 1,000,000, when $r = 20$, which is slightly larger, P_{guess} will be significantly decreased to $5.24 \cdot 10^{-14}$. Clearly, this probability is negligible when r and k are sufficiently large, even with a large η .

Case 2: the adversary tries to associate $n - k + 1$ data symbols with a code word by observing the TApp disk access requests upon encoding/decoding. As noted above, HiDCS obfuscates the disk addresses when reading the data blocks, and the data blocks are encrypted in transit by FTL. This essentially means that the adversary does not gain any additional advantage in figuring out the association compared to observing random disk access over random content.

5.2 Discussion

About the threshold that separates large and small corruptions. The idea behind this separation is that small corruption has a low probability of being detected by spot checking. Naturally, this threshold will determine the ECC parameters and depend on the self-auditing parameters. For example, when $P_{detect}^1 \geq 0.999999998$, solving for β when $c = 2000$ gives approximately 1% corruption. The threshold separating small and large corruption for this case can be 1% corruption. Thus, k and n should be selected to accommodate at least 1% redundancy, although a higher redundancy provides a safer margin.

Deploying HiDCS in real world. HiDCS requires modifying the FTL firmware. This implies that SSD manufacturers need to incorporate HiDCS during manufacturing or include provisions for a user to easily modify the FTL. For legacy SSDs that do not allow re-flashing of FTL firmware, this could be an obstacle to deploy HiDCS. However, this would not be a problem for open-channel SSDs [15].

Extending HiDCS to other coding methods. HiDCS has been uniquely designed for a replication-based storage system. In the cloud setting, data can be stored redundantly using other coding methods such as erasure coding [16, 18] and network coding [21, 23]. The self-audit and self-repair design for decentralized cloud storage based on other coding methods is currently unavailable and will be explored in our future work.

6 Implementation and Evaluation

A comparison of the theoretical performance of HiDCS with the other decentralized cloud storage systems is shown in Appendix D, which shows that HiDCS can achieve the best performance in both the audit and repair processes. The expense of HiDCS arises mainly from the additional computational overhead required for each peer to perform self-audit and self-repair. We thus implement [4] both functions in real-world computing devices and experimentally assess the respective computational overhead imposed on the local peer.

Experimental setup. We used a Lenovo Yoga C940 laptop (with Intel Core i7-1065G7 1.3 GHz and 12GB LPDDR4 3733 MHz RAM) with SGX enabled as host computer. Additionally, we used an LPC-H3131 [47] development board (with 180 MHz ARM microcontroller, 32 MB SDRAM, and 512MB SLC NAND flash) as a flash storage device. We ported an open-source NAND flash manager OpenNFM [25] to LPC-H3131. The testbed requires the LPC-H3131 to be attached to a virtual machine (VM) running on the host computer via a USB2.0 interface to run correctly. We allocated 3 CPU cores and 2GB memory to a Ubuntu VM hosted on VirtualBox [7]. We implemented TApp which runs in an SGX enclave. FTL was implemented by modifying OpenNFM. Communications between the TApp and the FTL need to go through the UApp running in the untrusted OS. The cryptographic primitives were instantiated as follows: $f_{l,K}$ was instantiated using HMAC SHA-256; $\pi_{l,K}$ was instantiated using a Feistel cipher with AES-128 as its round function; $h_{l,K}$ was instantiated using HMAC SHA-1;

Procedure	GENTAGS	GENCHALLENGE	VERIFYAUDIT
Throughput (MiB/s)	14.84	163.07	4.92

Table 1: Throughput of TApp procedures (audit).

Procedure	STOREDATA	INITIATEAUDIT
Throughput (KiB/s)	1611.79	125.99

Table 2: Throughput of FTL procedures (audit).

Enc and Dec relied on AES-128 with ECB mode. For Audit, we used the construction for private verification in Compact PoR [57]. For Encode and Decode, we used Reed-Solomon coding, with a 2^{16} element finite field (16 bit symbols).

Audit. In both FTL and TApp, we measured the computation required in the procedures relevant to preparing and performing the self-audit. We observed that the computational time is mostly linear with the data size. Therefore, we provide the throughput results that are summarized in Tables 1 and 2. For the FTL, we have the following observations: 1) There is little performance degradation in STOREDATA compared to regular write operations over LPC-H3131. This is because this procedure does not involve unique extra operations for HiDCS. 2) The throughput of the INITIATEAUDIT procedure is a bit low due to the involvement of encryption over the challenged data. Such operations are particularly expensive for low-power embedded systems equipped with limited computational resources. For TApp, we observe that the most intensive procedure, VERIFYAUDIT is the slowest. This is because Audit consists of multiple expensive crypto operations, including decryption. This also implies the need to perform spot checks, as checking the entire data would be very expensive. Conversely, both the GENTAGS and GENCHALLENGE procedures have high throughput.

Repair. We also measured the throughput of the procedures relevant to repair in both FTL and TApp. Similarly, the computational time is mostly linear with the data size. Thus, we provide the throughput results that are summarized in Tables 3 and 4. For the FTL, we have the following observations: 1) As the “old data” are still preserved in the flash memory and, by restoring the small mappings, the large corruption can be repaired. The throughput of RESTOREMAPPINGS is 1722 KiB/s, which means that the repair rate of large corruptions can be as fast as 834.7MiB/s. 2) GETPARTITION is the slowest FTL procedure, due to the expense of both permuting the address and encrypting the data in the low-power embedded system. For TApp, both PARTITIONDATA and GENPARITYPROOF procedures can be performed efficiently.

Performance impact on the SSD. As HiDCS needs to modify the FTL firmware, we also evaluated how our modifications would affect the throughput of regular SSDs. We used the fio benchmarking tool [3] to run multiple access patterns, including sequential read (SR), random read (RR), sequential

Procedure	PARTITIONDATA	GENPARITYPROOF
Throughput (MiB/s)	23.03	881.23

Table 3: Throughput for TApp procedures (repair).

Procedure	GETPARTITION	STOREPARITY	RESTOREMAPPINGS
Throughput (KiB/s)	86.94	1087.61	1724.24

Table 4: Throughput of FTL procedures (repair).

	RW	RR	SW	SR
OpenNFM (KiB/s)	1648.13	2022.28	2099.41	2807.98
HiDCS (KiB/s)	1649.51	2012.46	2096.00	2785.08

Table 5: HiDCS vs. original OpenNFM in throughput.

write (SW), and random write (RW), respectively, for the LPC-H3131 using FTL in HiDCS and the original OpenNFM. The benchmark results are summarized in Table 5. The results indicate that HiDCS has negligible impact on SSD performance. This is because: 1) The self-auditing function is activated periodically, which will not affect the regular I/Os most of the time. 2) The self-repair is a rarely happening event, e.g., encoding is only activated upon pre-processing the data, repairing the large corruptions is performed only when the integrity auditing finds out corruptions, and the decoding happens upon data retrieval. Additionally, after the data are marked to be reclaimed by garbage collection, data reclamation is mostly asynchronous, so it will not significantly impact the throughput rates measured.

Impact on the SSD’s wear leveling effectiveness. The durability of flash memory is influenced by the effectiveness of its wear leveling mechanism. Therefore, we also evaluated how HiDCS can affect the effectiveness of wear leveling, using the Hoover economic wealth inequality indicator. The indicator calculates a normalized total of the differences between each measurement and the mean. For N erase blocks with erase counts e_1, e_2, \dots, e_N , the wear leveling inequality (WLI) [53] can be computed as $\frac{1}{2} \sum_{i=1}^N \left| \frac{e_i}{E} - \frac{1}{N} \right|$, where $E = \sum_{i=1}^N e_i$. A lower WLI indicates better wear leveling effectiveness. To measure this value, we continuously wrote data to LPC-H3131 and erased the device once filled. We repeated such a process until the total amount of data being written reached 20 GB. Under different wear leveling thresholds 10 and 20, we obtained a WLI value of 3.627% and 3.654%, respectively. Such low WLI values indicate that HiDCS has effective wear leveling. Furthermore, the WLI value decreases when the wear leveling threshold decreases, as a lower wear leveling threshold triggers wear leveling more frequently, resulting in more evenly distributed erasures.

7 Related Work

TEE-based cloud auditing. There are some cloud auditing designs [65, 40] that use trusted hardware to build the auditor. However, they do not address the outsourcing/generation attacks, and hence are not secure. In addition, they do not support local repair, which is a key contribution of HiDCS. Some other designs [68, 28] incorporate TEE for cloud integrity check. However, in their designs, the prover, not the auditor, uses TEE. Consequently, these are not self-auditing designs. In addition, they do not support local repair. All these schemes cannot ensure data reliability as data repair is not involved.

Decentralized cloud storage systems. Several existing decentralized cloud storage systems can ensure the reliability of outsourced data. Filecoin [51] is a blockchain-based decentralized storage network that integrates with the Inter-Planetary File System to create an incentivization layer. They introduce outsourcing/generation attacks [14] and a method to prevent them by using expensive cryptographic operations to make the dishonest outsourcing peer noticeably (i.e. 10-100x) slower than an honest one upon responding to an audit challenge. Storj [61] aims to provide a general purpose, inexpensive and reliable decentralized cloud storage system. They provide auditing and relocation mechanisms by relying on a centralized third party, known as a satellite, to periodically check data integrity and to perform relocation using distributed redundancy to relocate any corrupted data. Sia [63] is a general purpose blockchain-based decentralized storage network which is heavily based on smart contracts where storage proofs are required periodically to fulfill the contract terms. There are no required repair mechanisms, but the client can enable relocation by distributing redundancy among multiple peers. Koppercoin [44] integrated the blockchain into a peer-to-peer file storage system, replacing the Bitcoin proof of work with a new proof of retrievability. Blockstore [55] again uses blockchain to establish storage agreements between clients and storage peers. Blockstore is unique in that it does not provide explicit mechanisms for automated auditing. Audita [38] relies on a dealer node to break up the data stored by the client and send it to many different peers. Additionally, periodic storage proofs are submitted to the blockchain, and they aim for a PoW free solution which is fair, in that each data portion is equally likely to be audited regularly. Züs [12] is another blockchain-based decentralized cloud storage network that allows smart contracts that are tied to periodic conditional audits. In addition, they select a subset of the nodes that are paid to verify the audits. However, Züs ultimately relies on consensus to validate the verification results, and so still relies on the entire network for each audit. Du et al. [34] have designed a zero-knowledge auditing scheme with an on-chain check mechanism, with the aim of addressing the free-riding issue.

8 Conclusion

We have proposed HiDCS, an integrity-assured decentralized cloud storage system. Using the trusted components TEE and FTL, HiDCS has successfully developed two novel functions, namely self-audit and self-repair. HiDCS conforms to the notion of storage outsourcing by releasing the data owner’s burden on both data storage and management. In addition, it requires minimal computation/communication in both data auditing and repairing, by fully utilizing the available resources locally in the storage peers. Security analysis and performance evaluation have demonstrated its security and effectiveness.

Acknowledgments

This work was supported by US National Science Foundation under grant number 2225424-CNS and 2043022-DGE.

References

1. AMD Secure Encrypted Virtualization (SEV). Retrieved on February 26, 2025, from <https://www.amd.com/en/developer/sev.html>.
2. Centralized vs decentralized storage cost (2023). Retrieved on February 25, 2025, from <https://www.coingecko.com/research/publications/centralized-decentralized-storage-cost>.
3. fio. Retrieved on February 28, 2025, from <https://www.glennklockwood.com/benchmarks/fio.html>.
4. HiDCS. Retrieved on May 22, 2025, from <https://snp.cs.mtu.edu/HiDCS.html>.
5. Secure Enclave - Apple Support. Retrieved on February 28, 2025, from <https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web>.
6. Ssd market share. Retrieved on February 28, 2025 from <https://www.t4.ai/industry/ssd-market-share>.
7. VirtualBox. Retrieved on February 28, 2025, from <https://www.virtualbox.org/>.
8. What is an sla? best practices for service-level agreements. Retrieved on February 28, 2025, from <https://www.cio.com/article/274740/outourcing-sla-definitions-and-solutions.html>.
9. Giuseppe Ateniese, Foteini Baldimtsi, Matteo Campanelli, Danilo Francati, and Ioanna Karantaidou. Advancing scalability in decentralized storage: A novel approach to proof-of-replication via polynomial evaluation. *Cryptology ePrint Archive*, 2023.
10. Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Osama Khan, Lea Kissner, Zachary Peterson, and Dawn Song. Remote data checking using provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):12, 2011.
11. Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. Acm, 2007.
12. Thomas H. Austin, Saswata Basu, and Züs Team. Züs network architecture. *Züs Network Official Documentation*, October 2022.
13. S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang. Ssd-insider: internal defense of solid-state drive against ransomware with perfect data recovery. In *ICDCS*. IEEE, 2018.
14. Juan Benet, David Dalrymple, and Nicola Greco. Proof of replication. *Protocol Labs, July*, 27:20, 2017.
15. Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. {LightNVM}: The linux {Open-Channel}{SSD} subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, 2017.
16. Kevin D Bowers, Ari Juels, and Alina Oprea. Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM, 2009.

17. Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
18. Bo Chen, Anil Kumar Ammula, and Reza Curtmola. Towards server-side repair for erasure coding-based distributed storage systems. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 281–288. ACM, 2015.
19. Bo Chen and Reza Curtmola. Robust dynamic provable data possession. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 515–525. IEEE, 2012.
20. Bo Chen and Reza Curtmola. Remote data integrity checking with server-side repair. *Journal of Computer Security*, 25(6):537–584, 2017.
21. Bo Chen, Reza Curtmola, Giuseppe Ateniese, and Randal Burns. Remote data checking for network coding-based distributed storage systems. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pages 31–42. ACM, 2010.
22. Chen Chen, Anrin Chakraborti, and Radu Sion. {PEARL}: Plausibly deniable flash translation layer using {WOM} coding. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.
23. Henry CH Chen, Yuchong Hu, Patrick PC Lee, and Yang Tang. Nccloud: A network-coding-based storage system in a cloud-of-clouds. *IEEE Transactions on computers*, 63(1):31–44, 2013.
24. Niusen Chen and Bo Chen. Duplicates also matter! towards secure deletion on flash-based storage media by removing duplicates. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security*, 2022.
25. Google Code. Opennfm. <https://code.google.com/p/opennfm/>.
26. Intel Corporation. Intel trust domain extensions (intel tdx). Retrieved on February 28, 2025 from <https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html>, 2023.
27. Western Digital Corporation. Drive protection. Retrieved on February 28, 2025 from <https://www.westerndigital.com/solutions/data-security/drive-protection>, 2023.
28. Cláudio Correia, Rita Prates, Miguel Correia, and Luís Rodrigues. Potr: Accurate and efficient proof of timely-retrievability for storage systems. In *2023 IEEE 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 111–122. IEEE, 2023.
29. Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.
30. Reza Curtmola, Osama Khan, and Randal Burns. Robust remote data checking. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 63–68. ACM, 2008.
31. Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. Mr-pdp: Multiple-replica provable data possession. In *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, pages 411–420. IEEE, 2008.
32. Whitfield Diffie and Martin E Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 1976.
33. Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

34. Yuefeng Du, Huayi Duan, Anxin Zhou, Cong Wang, Man Ho Au, and Qian Wang. Enabling secure and efficient decentralized storage auditing with blockchain. *IEEE Transactions on Dependable and Secure Computing*, 2021.
35. C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 17(4):15, 2015.
36. Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of sgx and countermeasures: A survey. *ACM Computing Surveys*, 54(6):1–36, July 2021.
37. Ben Fisch. Tight proofs of space and replication. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part II 38*, pages 324–348. Springer, 2019.
38. Danilo Francati, Giuseppe Ateniese, Abdoulaye Faye, Andrea Maria Milazzo, Angelo Massimo Perillo, Luca Schiatti, and Giuseppe Giordano. Audita: A blockchain-based auditing framework for off-chain storage, 2020.
39. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.
40. Yun He, Yihua Xu, Xiaoqi Jia, Shengzhi Zhang, Peng Liu, and Shuai Chang. {EnclavePDP}: A general framework to verify data integrity in cloud using intel {SGX}. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 195–208, 2020.
41. Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K Qureshi. Flash-guard: Leveraging intrinsic flash properties to defend against encryption ransomware. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2231–2244, 2017.
42. Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deft!: Implementing plausibly deniable encryption in flash translation layer. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2217–2229, 2017.
43. Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597. Acm, 2007.
44. Henning Kopp, Christoph Bösch, and Frank Kargl. Koppercoin—a distributed file storage with financial incentives. In *Information Security Practice and Experience: 12th International Conference, ISPEC 2016, Zhangjiajie, China, November 16-18, 2016, Proceedings 12*, pages 79–93. Springer, 2016.
45. David Lazarus. Precious photos disappear, Feb 2005. Retrieved on February 28, 2025, from <https://www.sfgate.com/business/article/precious-photos-disappear-2734149.php>.
46. Emma Lee. Tencent cloud user claims \$1.6 million compensation for data loss . technode, Aug 2018.
47. Olimex Ltd. Lpc-h3131. Retrieved on February 28, 2025, from <https://www.olimex.com/Products/ARM/NXP/LPC-H3131/>.
48. Phison. Securing SSDs with code signing and digital signatures, 2024.
49. James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.
50. Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

51. Protocol Labs. Filecoin: A decentralized storage network. Technical report, Protocol Labs, July 19 2017. <https://filecoin.io/filecoin.pdf>.
52. Reyhaneh Rabaninejad, Bin Liu, and Antonis Michalas. Port: Non-interactive continuous availability proof of replicated storage. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 270–279, 2023.
53. Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 333–348, 2012.
54. Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick PC Lee, and Xiaosong Zhang. Accelerating encrypted deduplication via *sgx*. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 957–971, 2021.
55. Sushmita Ruj, Mohammad Shahriar Rahman, Anirban Basu, and Shinsaku Kiyomoto. Blockstore: A secure decentralized storage framework on blockchain. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, pages 1096–1103, 2018.
56. Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using *sgx* to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.
57. Hovav Shacham and Brent Waters. Compact proofs of retrievability. volume 26, pages 90–107, 12 2008.
58. Mehul A Shah, Mary Baker, Jeffrey C Mogul, Ram Swaminathan, et al. Auditing to keep online storage services honest. In *HotOS*, 2007.
59. Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, pages 325–336. ACM, 2013.
60. Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 318–331. IEEE, 2019.
61. Storj Labs, Inc. Storj: A decentralized cloud storage network framework. October 30 2018. Version 3.0. <https://github.com/storj/whitepaper>.
62. Choon Beng Tan, Mohd Hijazi, Yuto Lim, and Abdullah Gani. A survey on proof of retrievability for cloud data integrity and availability: Cloud storage state-of-the-art, issues, solutions and future trends. *Journal of Network and Computer Applications*, 110, 03 2018.
63. David Vorick and Luke Champine. Sia: Simple decentralized storage. 2014.
64. Peiyang Wang, Shijie Jia, Bo Chen, Luning Xia, and Peng Liu. Mimosaf1: Adding secure and practical ransomware defense strategy to flash translation layer. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 327–338, 2019.
65. Da Xiao, Lvyin Yang, Chuanyi Liu, Bin Sun, and Shihui Zheng. Efficient data possession auditing for real-world cloud storage environments. *IEICE Transactions on Information and Systems*, E98.D:796–806, 04 2015.
66. Wen Xie, Niusen Chen, and Bo Chen. Enabling accurate data recovery for mobile devices against malware attacks. In *International Conference on Security and Privacy in Communication Systems*, pages 431–449. Springer, 2022.
67. Jingting Xue, Chunxiang Xu, and Lanhua Bai. Dstore: A distributed system for outsourced data storage and retrieval. *Future Generation Computer Systems*, 99:106–114, 2019.

68. Yang Zhang, Weijing You, Shijie Jia, Limin Liu, Ziyi Li, Wenfei Qian, et al. Enclavepost: A practical proof of storage-time in cloud via intel sgx. *Security and Communication Networks*, 2022, 2022.

A Key Sharing between TApp and FTL

A naive solution towards sharing K_s between TApp and FTL would be to run the Diffie-Hellman [32] key exchange protocol between them. However, this suffers from a man-in-the-middle attack where UApp pretends to be TApp or FTL. The station-to-station protocol [33] allows TApp / FTL to authenticate key exchange messages, mitigating the man-in-the-middle attack. Specifically for our scenario, it is sufficient for TApp to authenticate FTL. This is because TApp controls the key exchange process and when detecting a failure, it can restart the process (upon continuous failure, the peer will simply fail to register, as the key exchange is the first step in initializing the peer). To allow the TApp to authenticate the FTL via the station-to-station protocol, the FTL must be equipped with a key pair and associated digital certificate. The certificate is essentially a signature of the public key provided by a trusted entity, typically known as the certificate authority (CA). The most natural choice for the CA is the SSD manufacturer. This choice ensures that the CA has physical access to the FTL. Therefore, after the FTL generates a key pair, the CA can embed the certificate in the SSD. Existing practices of secure boot and code signing in SSDs [27, 48] suggest that this already occurs. When the TApp needs to authenticate a key exchange message from the FTL, the TApp can establish a secure channel with CA (via attestation) to retrieve its public key, which is used to authenticate the certificate and, through station-to-station, the key exchange itself.

B Detailed Procedures for Section 4.3

B.1 Procedures Needed for Algorithm 1

The procedures GENTAGS, PARTITIONDATA, ENCODEPARTITION, GENPARTYPROOF and VERIFYSTORAGE are run by TApp which are elaborated as follows:

1. $\text{GENTAGS}(D) \rightarrow \tilde{D}$
This computes the set of tags T for the given data D , each tag for a data block. Tag generation algorithms such as the privately verifiable compact PoR [57] and PDP [11] can be used here. After generating T , TApp concatenates it to D , forming \tilde{D} .
2. $\text{PARTITIONDATA}(n, r, d, \gamma, K_s, K_D) \rightarrow \{N, K_p, \mathcal{I}^*\}$
 - a) First, TApp generates $\mathcal{I} = \{I_0, I_1, \dots, I_{r-1}\}$ such that $\forall I_a \in \mathcal{I}, I_a = \{i \in \mathbb{Z}_{|D|} \mid g(i) = a\}$. Each I_a describes a $\tilde{P}_a \subseteq \tilde{D}$ where each number in I_a is the index of a data block and its tag in \tilde{D} . Using g randomizes the partitioning with K_D .

Algorithm 4 ExtractSentinels

Require: λ_a^*, l, N_a, K_a
Ensure: M_a

```

1:  $M_a \leftarrow \{\}$ 
2: for each block  $b_j$  in  $\lambda_a^*$  do
3:   for each page  $p_i$  in  $b_j$  do
4:      $R \leftarrow f_{(l * \log_2(\frac{|p_i|}{l}), K_a)}(N_a)$ 
5:     Assume  $R = \{r_0, r_1, \dots, r_{l-1}\}$ 
6:      $m_i \leftarrow \{0, 1\}^l$ 
7:     for  $k = 0$  to  $l - 1$  do
8:        $m_i[k] \leftarrow p_i[k * \frac{|p_i|}{l} + r_k]$ 
9:     Append  $m_i$  to  $M_a$ 
10: return  $M_a$ 

```

- b) Next, **TApp** generates a nonce $N = f_{(\kappa, d)}(\gamma)$ and a permutation key $K_p = h_{(\kappa, K_s)}(N)$, and generates $\mathcal{I}^* = \{I_0^*, I_1^*, \dots, I_{r-1}^*\}$ such that $\forall I_a^* \in \mathcal{I}^*, I_a^* = \{\pi_{(\log_2(n), K_p)}(i) \mid i \in I_a\}$. This permutes the association between indices and partition using K_p , ensuring that **FTL** can discover what blocks belong to the same partitions when given \mathcal{I}^* , while **UApp** cannot.
3. **ENCODEPARTITION** $(\tilde{P}_a^*, K_p) \rightarrow \{v, \lambda_a^*\}$
 Compute $\tilde{P}_a = \text{Dec}_{K_p}(\tilde{P}_a^*)$. Then an integrity check is performed on \tilde{P}_a by setting $v = \text{Audit}(\tilde{P}_a)$. **TApp** then extracts P_a from \tilde{P}_a by discarding the tags. Next, it computes the parity $\lambda_a = \text{Encode}(\text{Convert}(P_a))$, and encrypts it, producing $\lambda_a^* = \text{Enc}_{K_D}(\lambda_a)$. This is encrypted so that **UApp** cannot reverse engineer the association between data and partition.
4. **GENPARITYPROOF** $(\lambda_a^*, l) \rightarrow \{N_a, K_a, M_a\}$
 a) The goal is to extract the sentinels M_a from λ_a^* , where $M_a = \{m_0, m_1, \dots, m_{r-1}\}$. Each $m_j = \{0, 1\}^l$. l is a security parameter that indicates the number of bits in each page of λ_a^* to be extracted.
 b) A nonce $N_a = f_{(\kappa, d)}(\gamma)$ and a partition key $K_a = h_{(\kappa, K_s)}(N_a)$ are generated. The Algorithm 4 is run with the inputs λ_a^*, l, N_a , and K_a to obtain M_a .
5. **VERIFYSTORAGE** $(v, h_{(\kappa, K_a)}(v)) \rightarrow \{0, 1\}$
 Compute $h_{(\kappa, K_a)}(v)$ and compare it with the received signature. If they are identical, return 1. Otherwise, return 0.

The procedures **STOREDATA**, **GETPARTITION** and **STOREPARITY** are run by **Uapp** and **FTL**, which are elaborated as follows:

1. **STOREDATA** (\tilde{D})
 Here, **UApp** simply writes \tilde{D} to **FTL**, storing the data and its associated tags.
2. **GETPARTITION** $(N, I_a^*, K_s) \rightarrow \tilde{P}_a^*$
 First, **FTL** generates the permutation key $K_p = h_{(\kappa, K_s)}(N)$. Next, $I_a = \{\pi_{(\log_2(n), K_p)}^{-1}(i^*) \mid i^* \in I_a^*\}$ is computed. Retrieve $\tilde{P} = \{b_j \in D \mid j \in I_a\} \cup \{t_j \in T \mid j \in I_a\}$, then return, $\tilde{P}_a^* = \text{Enc}_{K_p}(\tilde{P}_a)$. This is encrypted

by FTL so that the association between the partition and its data cannot be known by **UApp**.

3. STOREPARITY($N_a, l, \lambda_a^*, h_{\kappa, k_a}(M_a)$) $\rightarrow (v, v_s)$
 First generate $K_a = h_{(\kappa, K_s)}(N_a)$. Extract M'_a from λ_a^* using Algorithm 4, then calculate $h_{\kappa, k_a}(M'_a)$ and compare it to the received signature. If they are mismatched, set $v = 0$ and return. Otherwise, $v = 1$ and the writes are disabled on λ_a^* . Return $(v, h_{(\kappa, K_a)}(v))$.

B.2 Procedures Needed for Algorithm 2

The procedures GENCHALLENGE and VERIFYAUDIT are run by the **TApp**, which are defined as follows:

1. GENCHALLENGE($d, \gamma, K_s, \kappa, n, q$) $\rightarrow \{N, K_c, S\}$
 Generates a challenge for the audit, consisting of a nonce $N = f_{(\kappa, d)}(\gamma)$, used to generate a challenge key $K_c = h_{(\kappa, K_s)}(N)$, and the indices of the challenged blocks. These indices are $S = \{\pi_{(\log_2(n), K_c)}(x) \mid x \in \mathbb{Z}_c\}$.
2. VERIFYAUDIT(S, A^*, K_c) $\rightarrow \{0, 1\}$
TApp first decrypts A^* , computing $A = \text{Dec}_{K_c}(A^*)$. Next, **TApp** computes $\text{Audit}(A)$ and returns the result.

The procedure INITIATEAUDIT is run by **UApp** and FTL, which is defined as follows:

1. INITIATEAUDIT(N, S, K_s, κ) $\rightarrow A^*$
 First, FTL generates the challenge key $K_c = h_{(\kappa, K_s)}(N)$. Next, FTL collects $A = \{b_j \in D \mid j \in S\} \cup \{t_j \in T \mid j \in S\}$ and returns $A^* = \text{Enc}_{K_c}(A)$.

B.3 Procedures Needed for Algorithm 3

The procedure DECODEPARTITION is run by **TApp** which is defined as follows:

1. DECODEPARTITION($N_a, r, l, K_p, \lambda_a^*, \tilde{P}_a^*, h_{(\kappa, K_a)}(M_a)$) $\rightarrow \tilde{P}_a^*$
 Generate $K_a = h_{(\kappa, K_s)}(N_a)$. Extract M'_a from λ_a^* using Algorithm 4. Compute $h_{(\kappa, K_a)}(M'_a)$ and compare with the received signature. Compute $\lambda_a = \text{Dec}_{K_D}(\lambda_a^*)$ and $\tilde{P}_a = \text{Dec}_{K_p}(\tilde{P}_a^*)$. Then, repair any corruptions by computing

$$\tilde{P}_a = \text{Decode}(\text{Convert}(\tilde{P}_a) \parallel \text{Convert}(\lambda_a)). \text{ Next, return } \tilde{P}_a^* = \text{Enc}_{K_p}(\tilde{P}_a).$$

The procedures READPARITY and RESTOREMAPPINGS are run by FTL and **UApp** during repair, which are defined as follows:

1. READPARITY($a, |\lambda_a^*|$) $\rightarrow \lambda_a^*$
 Read the encrypted parity data associated with the a -th partition. Its location is known by **TApp**, and it has length $|\lambda_a^*|$.
2. RESTOREMAPPINGS(N) $\rightarrow \{0, 1\}$
 Restores the back-up mappings in FTL. This restores corrupted data in the event of a large corruption. FTL writes the backup mappings to the current mapping table.

C Robustness Analysis

By robustness, we mean the ability of HiDCS to restore from the worst case of any sized random corruption attack. Robustness in our context ensures that with appropriate parameters, the probability of irreparable data loss is minimized to an acceptably low threshold in the local peer.

We consider the worst case of a random β -proportion corruption of the data D . The best strategy for attackers to perform a small corruption is that the corruptions occur at the same respective location of each block (see Figure 2). Based on this worst case, we can calculate the probability $P_{\text{irreparable}}$ that at least one code word C has at least $n - k + 1$ corrupted symbols based on β , n , and k . This is given as $P_{\text{irreparable}} = 1 - (1 - P_C)^{\frac{|D|}{k}}$, where the probability that a *particular* code word C has at least $n - k + 1$ corrupted symbols P_C is given by the cumulative hypergeometric probability as $\sum_{j=n-k+1}^k \frac{\binom{k}{j} \binom{|D|-n}{\beta \cdot |D| - j}}{\binom{|D|}{\beta \cdot |D|}}$ [57].

Note that we use k as the summation bounds rather than n , as only the k data blocks can be corrupted in our design. Using this, we define the probability that the corruption is either directly repairable via decoding or detectable via spot checking.

$$P_{\text{repair}} = 1 - (1 - P_{\text{detect}}^\alpha) \cdot P_{\text{irreparable}} \quad (1)$$

For example, for $\alpha = 1$, $\beta = 0.02$, $c = 400$, $|D| = 100,000$, $n = 140$, and $k = 128$, $P_{\text{repair}} = 0.999999992$. The probability of local repair failure is $1 - P_{\text{repair}}$, which is $8 \cdot 10^{-9}$.

D Comparing HiDCS with Other Decentralized Cloud Storage Systems

We compare HiDCS with the other decentralized cloud storage systems in Tables 6 and 7. *CTP-free* means free of Centralized Third Parties, that is, no centralized third party is involved in the operations. *SSC* means Server-Side Computation; that is, all operations (other than setup) take place on the servers. For the audit process (Table 6), only HiDCS and Filecoin can be free of centralized third parties, allowing server-side computation and resistance against outsourcing / generation attacks. However, Filecoin’s computation / communication in the auditing is linear to the number of miners, while HiDCS remains constant in computation and zero in communication. For the repair process (Table 7), only HiDCS can support local repair, which results in zero communication of high probability. All other schemes require a linear communication to the size of repaired data. In summary, HiDCS achieves the best performance among the designs of decentralized storage systems in both audit and repair processes due to the introduction of hardware-assisted self-audit and self-repair.

System	CTP-Free SSC		Outsourcing/Generation Attacks Prevention	Computation	Communication
Filecoin [51]	Yes	Yes	Yes	$O(s)$	$O(s)$
Storj [61]	No	Yes	No	$O(1)$	$O(1)$
Sia [63]	Yes	Yes	No	$O(s)$	$O(s)$
Koppercoin [44]	Yes	Yes	No	$O(s)$	$O(s)$
Blockstore [55]	No	No	No	$O(1)$	$O(1)$
Audita [38]	Yes	Yes	Partial	$O(s)$	$O(s)$
Züs [12]	Yes	Yes	Partial	$O(s)$	$O(s)$
Du et al. [34]	Yes	Yes	Partial	$O(s)$	$O(s)$
HiDCS	Yes	Yes	Yes	$O(1)$	0

Table 6: A comparison of auditing in different decentralized cloud systems. s is the number of miners which need to communicate.

System	CTP-Free SSC		Local Repair	Distributed Repair	Communication
Filecoin [51]	Yes	No	No	Relocation	$O(n)$
Storj [61]	No	Yes	No	Relocation	$O(n)$
Sia [63]	Yes	No	No	Relocation	$O(n)$
Koppercoin [44]	Yes	No	No	Relocation	$O(n)$
Blockstore [55]	Yes	No	No	Relocation	$O(n)$
Audita [38]	Yes	No	No	Relocation	$O(n)$
Züs [12]	Yes	No	No	Relocation	$O(n)$
Du et al. [34]	Yes	No	No	Relocation	$O(n)$
HiDCS	Yes	Yes	Yes	$\left\{ \begin{array}{l} \text{Local Repair} \\ \text{Relocation} \end{array} \right.$	$\left. \begin{array}{l} 0 \text{ (with probability } P_{\text{repair}}) \\ O(n) \text{ (with probability } 1 - P_{\text{repair}}) \end{array} \right.$

Table 7: A comparison of repairing in different decentralized cloud storage systems. P_{repair} is close to 1 as analyzed in Appendix C, and therefore, $1 - P_{\text{repair}}$ is close to 0. n is the size of the data being repaired. All the schemes have a computation linear to the size of the repaired data.