SemFT: A Lightweight Semantic Fault Tolerance Framework for Encrypted File Systems in Cloud Storage

Hui An
Chongqing University
Chongqing, China
anhui@stu.cqu.edu.cn

Chunhua Xiao*
Chongqing University
Chongqing, China
xiaochunhua@cqu.edu.cn

Xiaoyu Wang
Chongqing University
Chongqing, China
wangxiaoyu@stu.cqu.edu.cn

Jiaxuan Long
Chongqing University
Chongqing, China
longjiaxuan@cqu.edu.cn

Yajie Li
Chongqing University
Chongqing, China
liyajie@stu.cqu.edu.cn

Lin Wu
Chongqing University
Chongqing, China
wulin_cqu@alu.cqu.edu.cn

***Abstract*—Silent data corruption (SDC) in the encryption path poses a serious yet underexplored threat to cloud storage systems. Unlike traditional fail-stop errors, SDC remains hidden from the system, making it difficult to detect and allowing corruption to silently propagate across layers. However, existing hardware- and software-based fault-tolerance techniques either incur high overhead or fail to address the semantic vulnerability of encryption. To fill this gap, we first conduct a system-level analysis and characterization of SDCs in the encryption path, revealing that such errors are not only silent but also highly propagative. Based on these insights, we propose SemFT, a lightweight semantic-level fault-tolerance framework built on encrypted file systems. SemFT combines semantic validation with decision-driven redundant execution to enable runtime detection and immediate recovery without modifying applications or hardware. To further address performance bottlenecks under high concurrency and limited CPU resources, SemFT introduces a heterogeneous offloading mechanism that accelerates redundant execution by offloading it to a dedicated accelerator, while keeping semantic decision logic on the CPU. We evaluate SemFT using end-to-end fault injection under both normal and extreme SDC scenarios. Experiments on real encryption workloads show that SemFT reliably detects and corrects all injected faults, completely suppresses fault propagation, and prevents system-level corruption. The framework introduces $< 3\%$ runtime overhead and maintains stable tail latency under fault injection. Compared to a CPU-only baseline, offloading redundant execution to the heterogeneous accelerator improves performance by up to 2.73× and significantly reduces system latency. Overall, SemFT provides an effective and deployable defense against SDC in the encryption path.**

***Index Terms*—Cloud storage, Encrypted File System, Silent data corruption, Fault tolerance, Heterogeneous Acceleration**

I. Introduction

With the development of cloud computing, cloud storage has become a critical infrastructure for large-scale data management. To protect data integrity and security, files are encrypted before persisting to backend storage devices [1], [2]. However, major cloud providers have increasingly reported the severe impact of silent data corruption (SDC) along the encryption path in production systems. SDC refers to erroneous results caused by transient hardware or software faults that do not immediately trigger crashes, allowing corruption to propagate undetected. In cloud storage, SDC can silently traverse the encryption path and be written into persistent storage, ultimately causing backend data corruption, significantly reduced availability, and even irreversible user data loss [3]–[21].

Current cloud storage deployments rely on error-correction code (ECC) memory, redundant array of independent disks (RAID), cyclic redundancy check (CRC) [8], [16], [22], [23], and replica validation. These mechanisms are effective against bit errors in storage media or transmission, but share a fundamental limitation: they primarily ensure correctness of storage and transport, not the semantic correctness of the encryption computation itself. Consequently, corrupted ciphertext may be durably persisted, replicated, and amplified across replicas without being recognized. In contrast, processor-level approaches such as lockstep execution, dual modular redundancy, and hardware instruction-level detection can improve computational reliability [24]–[27], but they usually increase hardware cost, reduce deployment flexibility, and cannot be applied easily to already deployed infrastructure [5], [11], [28]–[36]. The above limitations are not addressed by state-of-the-art fault tolerance mechanisms: hardware approaches are difficult to deploy in practice, and conventional fault tolerance in cloud storage often fails to meet accuracy demands.

Two opportunities arise for addressing this gap. First, utilizing file system level encryption to provide transparent protection near the data path [37]. This allows encryption outputs to be validated for consistency and semantics, without modifying application interfaces or hardware. Second, leveraging heterogeneous architectures and accelerators to offload

*Corresponding author.

partial tasks from the main CPU [38]–[42]. In general, such offloading mainly helps reduce the overhead of redundancy and other extra computations.

However, these approaches are still not fully exploited in practice. Most existing system-level fault tolerance methods still lack a mechanism that can leverage the semantics of encryption operations oriented to results [43]. Meanwhile, the boundary of heterogeneous offloading is often difficult to define precisely. Thus, a key challenge is how to organically integrate semantic-aware fault tolerance with heterogeneous execution while preserving accurate result validation.

Fortunately, although existing studies have not fully resolved these issues, they have already provided important insights for this direction. Traditional redundancy mechanisms show that re-execution and result comparison can improve computational reliability [8], [16], [22], [23]. System-oriented methods suggest that embedding detection and recovery logic into the execution path can stop error propagation at an earlier stage. Heterogeneous offloading schemes further indicate that using auxiliary computing resources can improve overall efficiency [38]–[42]. Together, these efforts suggest that it is feasible to build a fault tolerance framework that balances result correctness, online detection capability, and low overhead.

Based on these observations, we propose **SemFT**, a lightweight semantic-level fault-tolerance framework for encrypted file systems, which operates in-path and without hardware changes to provide accurate online detection and correctness of SDC, while maintaining low overhead and strong robustness.

In summary, this paper makes the following contributions:

(1) We propose SemFT, a semantic-level fault-tolerance framework for encrypted file systems. SemFT is grounded in a system-level characterization of encryption-path SDC, which reveals a unique failure mode: corruptions that are structurally valid but semantically incorrect and can propagate across layers. SemFT enables in-path detection and prompt recovery without modifying applications or hardware.

(2) To overcome CPU resource constraints, we design a heterogeneous redundancy scheme that keeps correctness-critical semantic decisions on the CPU while offloading redundant computation to heterogeneous accelerators. This design preserves full fault-tolerance guarantees and improves throughput without compromising detection accuracy.

(3) We implement SemFT in an enterprise-level encrypted file system and evaluate it with end-to-end fault injection. Results show that SemFT suppresses SDC propagation, introduces 3% runtime overhead, maintains stable tail latency across diverse workloads, and achieves up to 2.73× throughput improvement with heterogeneous offloading compared to a CPU-only baseline.

II. Motivation

The proposed SemFT framework aims to solve silent data corruption in the encryption path of cloud storage systems. In this section, we first analyze the root causes and propagation characteristics of SDC in the encryption path with experimental results, and then present an industrial case to highlight its practical risk in real cloud storage systems.

A. SDC Root Causes and Propagation in the Encryption Path

SDC originates from diverse sources across hardware and software layers, posing significant challenges to system reliability. At the processor level, SDC may result from transient faults such as bit-flips in registers, arithmetic units, or control logic, typically induced by radiation or power fluctuations [9], [11], [13], [44]–[46]. In memory and storage subsystems, faults in I/O paths, such as during DMA operations or zero-copy data transfers, may propagate undetected due to insufficient end-to-end validation [8], [18], [19]. At the system software level, SDC can stem from kernel bugs, driver faults, or file system inconsistencies, which may manifest during data transformation operations [14]. These multi-layered and often transient fault sources make SDC difficult to detect and correct, motivating the need for cross-layer, lightweight fault-tolerant mechanisms tailored to modern high-reliability environments such as cloud storage systems [10], [12], [15], [16].

TABLE I: Per-round normalized Hamming distance (NHD) under a single-bit fault injected at the FEK stage

Round r	$NHD(K^{(r)})$	$NHD(S^{(r)})$
R0 (Init)	0.504	0.500
R1	0.500	0.505
R2	0.507	0.499
R3	0.501	0.504
R4	0.500	0.507
R5	0.502	0.500
R6	0.501	0.503
R7	0.500	0.505
R8	0.501	0.503
R9	0.502	0.505
R10	0.501	0.500
Mean±Std	0.502±0.002	0.503±0.003
Min–Max	0.500–0.507	0.499–0.507
CV (%)	0.42	0.55

To systematically analyze the sources of silent data corruption within the encryption path, we select a representative set of injection points, spanning both the file system write path itself and the various stages of encryption operations. Then we tailor our fault modes according to the computational characteristics of each specific injection point [47]. The fault injection methodologies, selected interfaces, and experimental results are summarized in Table II. The results demonstrate that SDCs invariably result in system-level impacts regardless of whether the perturbation occurs during any state of the encryption computation phase or during standard file system operations.

Having identified the types of fault injection points within the encryption path that can trigger SDC, we proceeded to

TABLE II: Fault injection processes and effects in the encryption I/O path

Process	Injected Process	Fault Injection Description	Effect	Type
Initialization	`ext4_setup_super()`	During decode, the faulted function address is substituted with a different target address.	File system mount failure	Fatal
	`sync_filesystem()`	Same as above.	System startup failure	Fatal
Read	`vfs_read()`	A NEW VALUE fault bitmasks the function pointer during execution, redirecting control to another function.	System startup failure	Fatal
Write	`generic_perform_write()`	A fault is injected at the memory address of the write-path injection switch, causing a read-after-write mismatch.	Read/write data mismatch	SDC
	`ecryptfs_write_lower()`	A user-defined global variable controls fault injection and specifies the target file size via its injected value.	File read/write failure	SDC
Encryption	`FEK`	A single-bit flip is injected into the AES round key during encryption.	Encrypted data corruption	SDC
	`Round key`	Same as above.	Encrypted data corruption	SDC
	`Intermediate state`	Same as above.	Encrypted data corruption	SDC

analyze the propagation characteristics of transient perturbations within the write path. In cryptographic algorithms, even a single-bit perturbation can propagate through nonlinear transformations, rapidly affecting multiple output bits. To characterize this diffusion effect, we employed the bit-level normalized Hamming distance (NHD) as a quantitative metric to conduct a statistical analysis of error propagation characteristics across different fault injection locations.

Let the reference ciphertext obtained under fault-free execution be

$$C^{\text{ref}} = \{c_1^{\text{ref}}, c_2^{\text{ref}}, \ldots, c_L^{\text{ref}}\},$$

and let the ciphertext obtained under fault injection be

$$C^{\text{fault}} = \{c_1^{\text{fault}}, c_2^{\text{fault}}, \ldots, c_L^{\text{fault}}\},$$

where L denotes the length of the ciphertext. The Hamming distance between the two ciphertexts is defined as

$$HD(C^{\text{fault}}, C^{\text{ref}}) = \sum_{i=1}^{L} \mathbb{I}\left[c_i^{\text{fault}} \neq c_i^{\text{ref}}\right],$$

where $\mathbb{I}[\cdot]$ denotes the indicator function, which equals 1 when the enclosed condition holds and 0 otherwise. The normalized Hamming distance is then defined as

$$NHD = \frac{HD(C^{\text{fault}}, C^{\text{ref}})}{L}.$$

By measuring the bit-level differences between the fault-injected output and the fault-free reference, followed by normalization, we can quantify the diffusion of a single-bit perturbation through the encryption computation. The NHD value ranges from $[0, 1]$. When NHD is close to 0, the perturbation affects only a few bits; when NHD approaches 0.5, the output is approximately randomized, indicating that the fault has fully diffused during encryption. Theoretically, under ideal diffusion conditions, a single-bit perturbation propagates through all rounds of the cipher, yielding a bit-flip probability close to 0.5. Therefore, NHD near 0.5 can be regarded as a sign of sufficient fault diffusion.

As shown in Table I, the per-round NHD measured at both the round keys $K^{(r)}$ and the intermediate states $S^{(r)}$ remains consistently close to 0.5 across all rounds. This indicates that even a single-bit perturbation in cryptographic operations diffuses rapidly. Consequently, subsequent downstream checks for byte-level consistency or structural integrity alone cannot identify the source of the computational error.

The preliminary experiment results indicate:

Observation 1: encryption-path faults can remain structurally valid yet semantically wrong. We observe that encryption-path SDCs induce read/write mismatches and corrupt critical metadata. Such errors often escape normal execution and are only revealed by explicit read-back checks or user reports, yet they can accumulate to the point of compromising file system consistency and even causing unavailability.

Observation 2: once committed, SDC can propagate across layers and replicas. Multilayered failures originating from computation and file system operations can rapidly propagate through the encryption process. This dispersion makes corruption difficult to localize and easy to commit silently, after which it may propagate across operations and eventually manifest as persistent SDCs.

B. Industrial Case: Encryption-Induced SDC Fault in Huawei Cloud EVS Cluster

In large-scale cloud storage systems, user data reliability may be affected by SDCs. In an incident disclosed by Huawei Cloud in Fig. 1, a transient fault on the primary node perturbed the encryption computation, causing the produced ciphertext to deviate from the correct result. Importantly, such faults do not necessarily break the ciphertext length or format. The faulty ciphertext still appears as a structurally valid byte sequence. The EVS cluster primarily relies on CRC to validate block integrity before persistence and replica propagation. However, traditional CRC can only confirm that the ciphertext has not

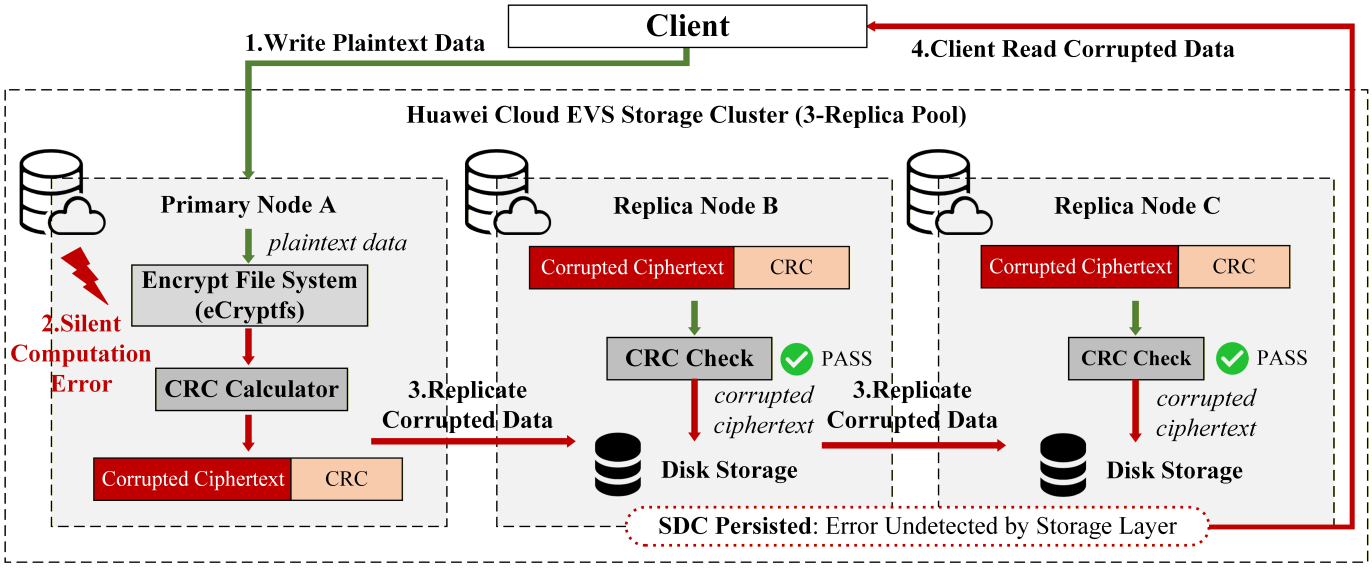


Fig. 1: Propagation of encryption-induced SDC in EVS cluster

been additionally corrupted in transit, but it cannot verify whether the ciphertext was computed from the correct plaintext under the correct encryption context. Consequently, the faulty ciphertext passed CRC verification and was propagated to all replicas, and the corruption was only discovered when the client later read the affected data. The data reconstruction process depended on remote backups stored in an independent backup pool and took several days to recover.

This case illustrates a representative and commercially significant failure mode: semantically incorrect ciphertext can remain structurally valid, bypass conventional integrity checks, and propagate across replicas before detection. Such incidents lead to service unavailability, expensive recovery operations, and erosion of user trust, turning a low-level encryption-path perturbation into a system-wide reliability and business risk. This motivates introducing result-oriented fault-tolerance mechanisms within the encryption path, so that erroneous ciphertext can be identified and blocked before committing.

Therefore, this work is motivated by a critical yet insufficiently explored challenge in cloud storage: SDCs arising in the encryption path are not isolated low-level faults, but semantically significant errors that can propagate through metadata updates and successive operations to trigger system-level failures. This reveals the need for a system-level fault-tolerance perspective that explicitly captures the semantic characteristics of encryption, beyond what conventional low-level mechanisms can effectively provide.

III. SemFT Design

Based on the above analysis, we argue that SDC in the encryption path is a system-level reliability problem rather than an isolated computation error. Faults may originate from different layers, rapidly diffuse through encryption computation and propagate across the storage path before being exposed. Therefore, a practical solution must detect and recover errors in-path, exploit the semantic properties of encryption operations for result validation and at the same time control the overhead of redundant execution. In this section, we first present the design overview of SemFT, and then describe its semantic validation mechanism, redundancy strategy, and heterogeneous execution design in detail.

SemFT does not attempt to verify intermediate micro-architectural states. Instead, it validates the correctness of completed transformation results at operator boundaries. Therefore, SDC in the encryption path must be modeled at the level of data transformation semantics, rather than low-level execution states.

A. Overview and Design Goals

SemFT is designed from a system perspective for the encrypted storage path, rather than as a local protection mechanism for a single encryption operator. Accordingly, SemFT is designed around the following three goals:

G1: Orient to results at semantic boundaries. Our design prioritizes the correctness of the final output of the encryption operation, rather than enforcing consistency at the instruction level. We establish verifiable semantic criteria at transformation boundaries and constrain error propagation by validating the produced output, thereby achieving end-to-end correctness with substantially lower system overhead.

G2: Keep the logic above instruction granularity. The system confines detection and fault-tolerance logic strictly to the semantic level, avoiding fine-grained instruction-level instrumentation of the kernel execution path and eliminating the need for hardware-dependent modifications. This design improves portability and scalability, while remaining friendly to high-throughput workloads.

G3: Prevent erroneous outputs from reaching commit. Upon detecting semantic inconsistencies or anomalous output, the system triggers a recovery mechanism within the current

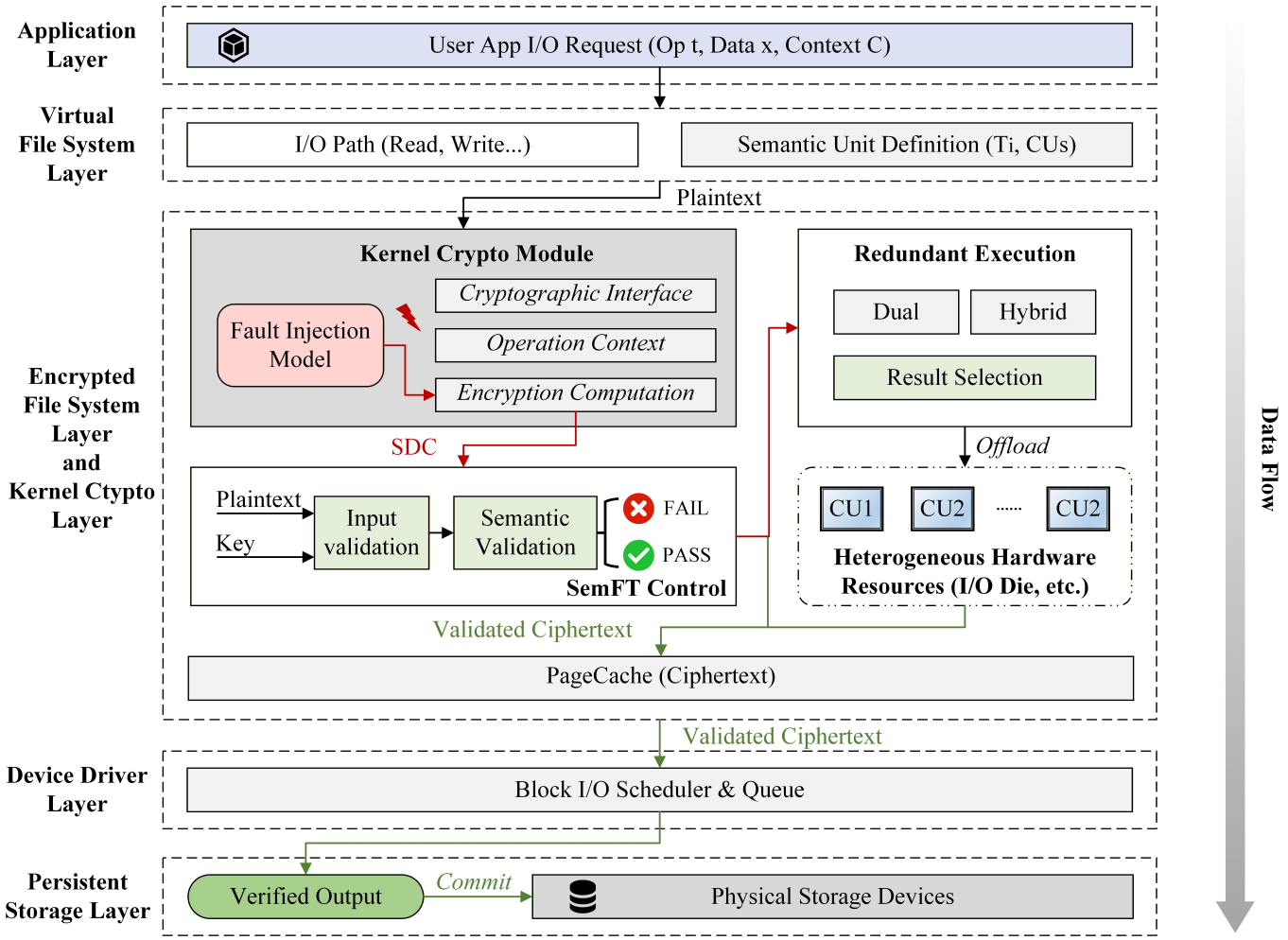


Fig. 2: Overview of the SemFT framework and workflow

I/O lifecycle. Through real-time detection and correction, the system completes the necessary rectifications prior to data commit, without introducing additional persistence semantics or modifying the file system interface. This prevents SDC from propagating to subsequent writes and persistent media, while maximizing system availability and service continuity.

As illustrated in Fig. 2, SemFT is organized as a cross-layer framework spanning the application, virtual file system, encrypted file system, kernel crypto layer, device driver, and persistent storage. This design reflects a key observation: SDC in the encryption path cannot be reliably contained by protecting isolated components alone, but instead requires semantic validation and recovery to be integrated into the path before corrupted results reach persistence.

B. Semantic Abstraction of the Encryption Path

In the encryption path, SDC does not primarily corrupt individual instructions or transient hardware states, but rather compromises the semantic correctness of transformation results under given input data and execution context. Therefore, SemFT does not rely on costly and poorly portable instruction-level modeling. Instead, it abstracts, validates, and controls the encryption process at semantic boundaries.

In Fig. 2, this semantic perspective is reflected across the hierarchical layers of the encryption path in three aspects. First, user I/O requests inherently include contextual information that defines the semantic conditions of an encryption operation and constrains the expected correct output. Second, prior to encryption computation, the semantic unit definition abstracts key steps in the encryption path into a set of verifiable compute units (CUs), providing a clear granularity for semantic-level detection and recovery. Finally, within the encrypted file system layer, encryption computation, input validation, and semantic validation together form the execution and enforcement of semantic correctness: the former performs context-constrained data transformation, while the latter evaluates the result at transformation boundaries and determines whether it can safely proceed to the block layer and persistent storage.

In the following, we elaborate on these design elements in detail. Specifically, we first introduce the semantic abstraction of system execution units, then describe the semantic correctness and validation criteria, followed by the placement of

semantic checkpoints, the decision-driven redundancy mechanism for recovery, and its reliability analysis.

C. Core Mechanisms of SemFT

Algorithm 1 summarizes the high-level workflow of SemFT on the protected encryption path. Each request is partitioned into verifiable compute units, and each unit is processed with semantic validation and policy-driven recovery. Only verified outputs are assembled and committed, reflecting SemFT's unified in-path design for semantic checking, error correction, and commit-time containment.

Algorithm 1 SemFT-protected encryption path

Require: Request req, policy $p \in \{\text{DUAL}, \text{HYBRID}\}$
Ensure: Commit only verified ciphertext

```
x ← GETINPUT(req), C ← GETCTX(req)
{CU_i}_{i=1}^n ← PARTITION(x)
for each CU_i do
    (y_i, ok) ← RECOVER(CU_i, C, p)
    if ¬ok
        ABORT(req);
        return EIO
    end if
end for
Y ← ASSEMBLE({y_i}_{i=1}^n)
COMMIT(Y);
return SUCCESS
```

1. Semantic abstraction of system execution units.

We treat each complete invocation of an encryption operator as a semantic unit. To clarify the granularity of redundancy, we introduce the concept of a compute unit (CU): a CU corresponds to a semantic unit of computation that can be independently completed within a kernel crypto module. The context of a CU includes the algorithm type, key, IV, logical offset, and block size. Under a fixed context, the input-to-output mapping for each CU is deterministic:

$$f : (\textit{Input\_Block}, \textit{Context}) \rightarrow \textit{Output\_Block}.$$

By performing redundant computations at the CU granularity, we can ensure that the semantic invariants match the output of the original operation. Given that the CU's configuration, execution context, and encryption parameters remain unchanged throughout the execution lifecycle of a single I/O operation, re-executing this operation preserves semantic properties consistent with the previous execution.

2. Semantic correctness and validation criteria.

A data-transformation invocation is semantically correct if and only if its output satisfies both the operator specification and a set of semantic invariants that must hold for all fault-free executions. For an operator T, we define a set of semantic invariants:

$$\mathcal{I}_T = \{I_1, I_2, \ldots, I_m\}.$$

For encryption, example invariants include: (i) *Length invariant:* $|out| = |in|$, i.e., ciphertext length equals plaintext length; (ii) *Block-structure invariant:* outputs are aligned to a fixed block size; (iii) *Context invariant:* for a deterministic implementation under $(key, IV, mode, offset)$, the same input under the same context produces identical ciphertext.

By partially redundantly executing the CU, the above invariants provide an operator-level validation criterion to hold for every computation, eliminating the need for lockstep verification of every instruction across the entire system.

These invariants strengthen the validation criterion from byte-level equality to result consistency under high-level semantic constraints. They also scope correctness to the operation level, rather than the instruction or bit level. Since semantic invariants are typically verifiable only at computing boundaries, they guide the placement of checkpoints.

3. Placement of semantic checkpoints.

We abstract the encryption data path as a sequence of atomic transformations. From a system perspective, the write path starts from application data, traverses the file system interface, enters the encryption routine and undergoes a series of internal transformation steps before committing to disk. Each T_i denotes a sub-step of the encryption procedure and may consist of one or more CUs, each modeled as a black-box function. If a fault occurs inside a CU, SemFT ensures that it is detected and recovered before it can escape beyond the CU output boundary.

Accordingly, we place a semantic checkpoint at the output of each encryption sub-step, rather than at intermediate internal states, ensuring that faults cannot escape beyond the sub-step output boundary.

4. Decision-driven redundancy for recovery.

To prevent inconsistencies detected during encryption from directly causing write failures, we introduce a decision-based redundancy strategy in the layered fault-tolerant design. When SemFT's semantic check determines that the output of an encryption computation T_i is anomalous, the system does not commit any ciphertext. Instead, the computation is re-executed once at the same operator granularity, and the new output participates in the consistency decision again, forming a *detection–re-execution* control.

We support two concrete redundancy policies. DUAL and HYBRID are two alternative redundancy policies used to validate and recover each compute unit before it is accepted by the global execution flow illustrated in Algorithm 1:

1) **Dual redundancy with re-execution for correction (Dual):** At the CU level, T_i is executed twice to perform a semantic consistency check. If the two outputs are inconsistent, a redundant execution is triggered, and the final output is determined by selecting the result that matches the majority, ensuring that any detected discrepancy is corrected at the CU granularity.
2) **Hybrid redundancy with input and compute for correction (Hybrid):** The system first validates the consistency of the input plaintext and master key before execution. If the input validation succeeds, computation proceeds at the CU granularity and the output of T_i is further checked by semantic validation. Once an

anomaly is detected, redundant re-execution is triggered to obtain a semantically consistent output.

Unlike traditional redundancy methods, which rely on a fixed number of parallel replicas and determine the output through majority voting, SemFT's redundancy mechanisms are configurable and decision-driven, operating at the boundaries of semantic operations. The system selects the final output based on semantic verification results rather than a fixed execution count, allowing redundancy to be triggered and resolved dynamically according to detected anomalies. This approach allows flexible selection of redundancy mechanisms according to resource availability and reliability requirements. In the SemFT heterogeneous hardware collaboration strategy detailed later, redundant computations can be further offloaded to heterogeneous computing resources such as I/O Die to alleviate the computational pressure and resource contention experienced by the main CPU during redundant execution.

5. Reliability analysis for decision-driven redundancy. This design is motivated by the statistical characteristics of transient faults in the encryption path. Let faults during a single encryption execution be modeled as a Poisson process with rate λ, and let T_{enc} denote the vulnerability window of the encryption routine. Then, the probability that one execution experiences at least one fault is

$$p_{\text{enc}} = 1 - e^{-\lambda T_{\text{enc}}} \approx \lambda T_{\text{enc}}.$$

Since CPU soft error rates under normal operating conditions are extremely low and the vulnerability window of a file system encryption operation is short, p_{enc} is typically very small in practice [7]. This observation justifies SemFT's decision-driven design: instead of always performing a fixed number of redundant executions, the system executes the primary computation first and invokes additional redundancy only when semantic validation reports an anomaly. In other words, redundancy is activated conditionally according to the validation outcome, rather than being imposed unconditionally on every operation.

Once such a decision is triggered, the correctness of the final result can still be characterized by redundancy theory. With n executions and threshold $t = \lfloor n/2 \rfloor + 1$, the probability of obtaining the correct output is

$$P_{\text{correct}} = \sum_{k=t}^{n} \binom{n}{k} (1 - p_{\text{enc}})^k p_{\text{enc}}^{n-k},$$

which rapidly approaches 1 as additional executions are introduced. Therefore, under the practical regime where faults are rare, SemFT avoids unnecessary replication in the common case, while still providing strong reliability once an abnormality is detected.

D. Resource-Optimized Redundant Execution Scheme Based on Heterogeneous Hardware Acceleration

To reduce the load on the main CPU and alleviate contention for kernel scheduling and memory bandwidth, we offload redundant encryption and decryption operations from the host CPU to heterogeneous hardware accelerators. In this architecture, the CPU still functions as the control core and manages the full lifecycle of all tasks. This design ensures that the decision logic for fault-tolerance remains within the trusted execution domain, preventing critical consistency decisions from being migrated into the heterogeneous device. As illustrated in Fig. 3, SemFT adopts a heterogeneous execution model in which the CPU remains responsible for semantic decisions and control, while redundant encryption tasks are offloaded through a control interface to heterogeneous hardware accelerators for execution and result return. In other words, the heterogeneous device only performs computation and does not participate in the final semantic decisions along the write path, which is a fundamental principle of the collaborative design.

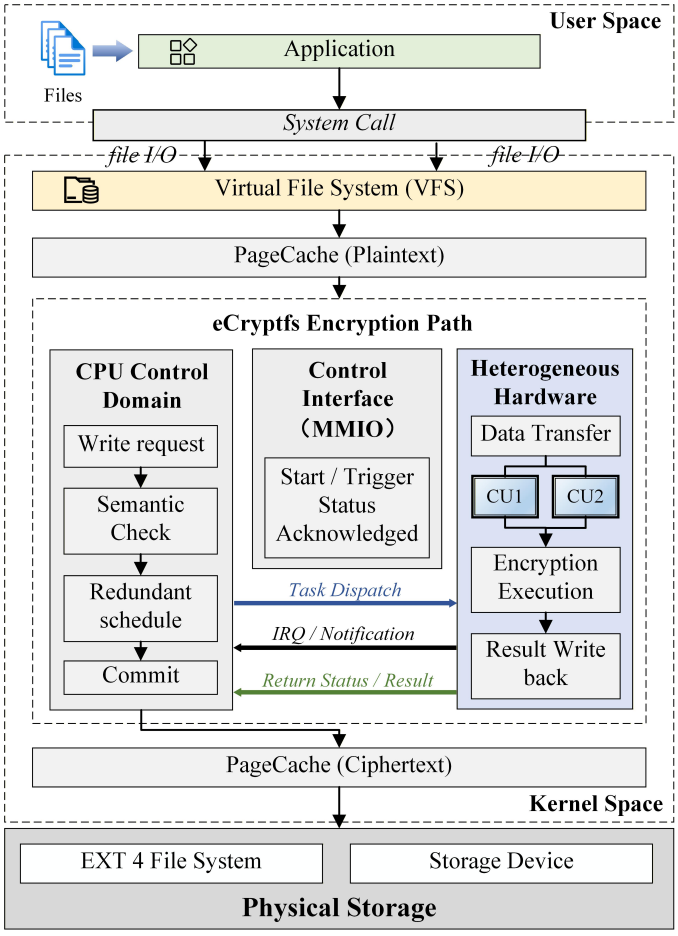


Fig. 3: Overview of SemFT's heterogeneous model

The key significance of the aforementioned heterogeneous co-design lies in the fact that it not only preserves the reliability of the SemFT decision logic but also provides additional parallel resources for redundant execution. Consequently, the actual performance overhead of the redundancy mechanism is no longer determined solely by the degree of redundancy; rather, it is jointly influenced by the number of available computing units and the associated coordination overhead. In the following section, we employ an overhead model to

analyze the end-to-end execution time under various resource conditions.

a) Overhead Model: We analyze the relationship between redundant execution overhead and the number of available computation units. Let the following denote:

- T: time required for one encryption operation on a single CU;
- N: redundancy factor (number of executions, including the primary one);
- T_r: coordination overhead (scheduling, comparison and result selection);
- C: number of available CUs;
- T_{total}: end-to-end execution time under different resource scenarios.

We consider two typical scenarios.

b) Scenario 1: Sufficient Execution Capacity ($C \geq N$): All N executions can be scheduled concurrently on distinct CUs. The total time is:

$$T_{\text{total}} = T + T_r.$$

In this case, the overhead is marginal, and the delay is dominated by the normal operation plus a small coordination cost.

c) Scenario 2: Constrained Execution Capacity ($C < N$): If redundant executions cannot be placed concurrently and must be time-multiplexed on fewer CUs, the total time increases due to serialization. In the extreme case where only one CU (one execution slot) is available, the total time becomes:

$$T_{\text{total}} = N \cdot T + T_r.$$

More generally, when only k CUs are available ($k < N$), the time can be approximated as:

$$T_{\text{total}} = \left\lceil \frac{N}{k} \right\rceil \cdot T + T_r.$$

When sufficient computation units are available ($C \geq N$), the overhead of redundant execution is minimal, dominated only by a small coordination cost. When execution capacity is constrained ($C < N$), the total execution time increases due to serialization, and can grow significantly if only a few units are available. By offloading redundant operations to the heterogeneous hardware, the effective number of available computation units increases, allowing parallel execution of redundant tasks, reducing end-to-end latency, and fully leveraging heterogeneous hardware resources.

E. Fault Injection Model Construction and Experimental Workflow

To systematically evaluate SemFT under a realistic cloud storage write path, we select AES as the target encryption algorithm. AES is one of the most representative symmetric encryption algorithms in cloud storage. It provides strong security protection while maintaining relatively low computational overhead, which makes it an ideal baseline for evaluating fault tolerance mechanisms [25]–[27]. For fault modeling, we adopt the single-bit flip transient fault model, which is the most widely used model in soft error research. This model is easy to instantiate, highly controllable, and closely related to real transient faults. For this reason, it has been widely used in reliability evaluations at both the processor level and the operating system level. As shown in Fig. 4, the fault injection model is built around the encryption computation path, covering both plaintext, master key and intermediate AES state corruptions, so as to emulate how SDC propagates to corrupted ciphertext in the experimental workflow.

To avoid biasing the experimental results with non-statistical cases in which an injected fault would inevitably lead to an error, we trigger random single-bit transient faults during each write operation with a predefined probability. Write operations in which no fault is injected are treated as internal control samples within the same experimental group. This probabilistic injection method allows both normal and abnormal execution paths to be included in the same batch of experiments, thereby more closely reflecting the way transient disturbances occur in real systems.

We randomly select fault injection points across multiple stages of the AES execution context so that the entire write path is covered. The experiments follow a single file sequential write model. In this setting, each write operation goes through the complete encryption process, including the plaintext buffer, the master key, the internal AES state, and the encryption computation itself. The effects of the injected faults are introduced at the level of CPU arithmetic instructions so that the direct impact of transient hardware faults on encryption results during computation can be accurately characterized. With this setup, we are able to quantitatively evaluate how transient computational faults arising at different stages of encryption propagate along the encrypted write path, and to examine the effectiveness of the proposed layered fault tolerance mechanism.

The fault injection points are categorized as follows:

Plaintext injection (P): Single-bit flips in the plaintext buffer to simulate transient disturbances during the input data acquisition stage;

Master key injection (K): Single-bit flips during key loading or usage to simulate errors in parameter state;

Computation state injection (S): Single-bit flips in the AES internal states during encryption to simulate hardware faults at the computation level.

The experimental framework also supports mixed injection across P, K, and S points to emulate complex disturbance scenarios closer to real runtime conditions. With this design, faults are naturally incorporated into the I/O workflow from the file system through the crypto module to block device access. This model allows evaluation of SemFT's detection and recovery capabilities along real encryption and decryption paths without modifying the system architecture.

IV. Results and Performance Evaluation

In this section, we evaluate the effectiveness of the proposed SemFT in the following dimensions:

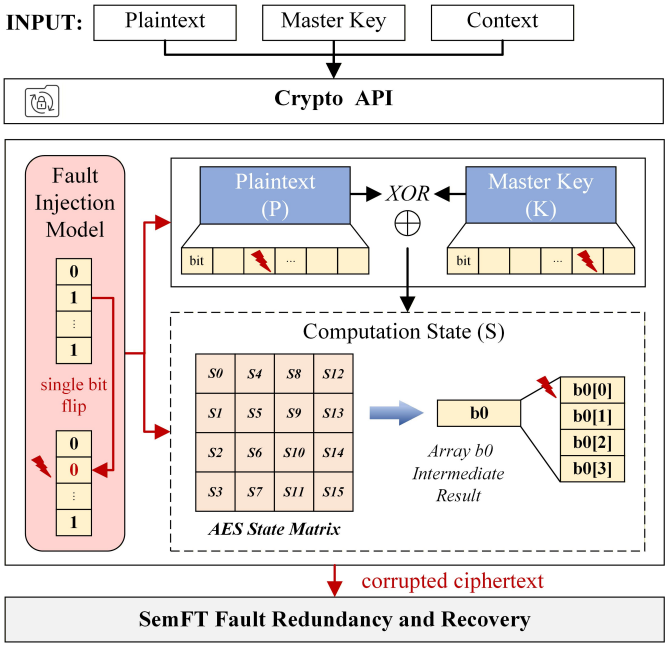


Fig. 4: SemFT's fault injection model and workflow

- Reliability under end-to-end fault injection.
- Cost under normal throughput-oriented workloads.
- Robustness under overload workloads.
- Tail behavior under latency-sensitive workloads.
- Benefit of heterogeneous redundancy offloading.

We first describe the experimental setup, including the system prototype and workload configurations, and then present results for each of the above dimensions, following the order listed.

A. Experimental Setup and Metrics

We implement SemFT on the enterprise-level encrypted file system eCryptfs in the Linux kernel, enabling end-to-end validation without modifying application interfaces. EXT4 serves as the underlying file system for encrypted data storage and persistence verification. To support heterogeneous execution, we deploy the redundant execution unit on I/O Dies with driver-level support for task dispatch, DMA transfer, and interrupt handling, allowing offloaded encryption execution and result return to the CPU.

We define key metrics to evaluate the reliability and performance of SemFT.

- Error Detection Rate (EDR): Measures the system's ability to detect injected computation errors. Here, I is an indicator function for the presence of conditions for error and E denotes the set of errors:

$$EDR = \frac{\sum_{i=1}^{n} \mathrm{I}\left(E_{detect}, i\right)}{\sum_{i=1}^{n} \mathrm{I}\left(E_{inject}, i\right)}$$

- Data Integrity (DI): Reflects the correctness of data after applying fault-tolerance mechanisms. Given a set of test files D, and D_{error} as the subset remaining incorrect post-processing:

$$DI = 1 - \frac{\sum_{i=1}^{n} \mathrm{I}\left(D_{error}, i\right)}{\sum_{i=1}^{n} \mathrm{I}\left(D_{total}, i\right)}$$

- P99: Reports the 99th-percentile I/O latency, capturing tail performance.
- Latency: Denotes the average per-request I/O completion time observed during the benchmark.
- Bandwidth: Measures the sustained data transfer rate achieved by the system during the workload.
- IOPS: Represents the number of I/O operations completed per second.

Since the fault arrival rate λ and the triggering conditions are typically unobservable in production, we parameterize this probability in a controllable way via *per-operation random single-bit flip injection* with probability r, so that the random fault injection rate r directly corresponds to the Poisson-model probability p_{enc}. Therefore, for a given injection rate $r \in \{0, 1\%, 10\%, 50\%\}$, the equivalent fault arrival rate can be derived as

$$\lambda = -\frac{\ln(1-r)}{T_{\mathrm{enc}}} \approx \frac{r}{T_{\mathrm{enc}}} \quad \text{(for small } r\text{)}.$$

By adjusting r, we covered fault severity levels ranging from sporadic transient errors to extreme risk scenarios, thereby probing the reliability boundaries of SemFT under extreme conditions. The injection rates to amplify rare events and expose propagation, detection, and recovery behavior within feasible experiment time, while the 0% case characterizes normal-case overhead.

B. RQ1: Can SemFT reliably detect and correct SDCs?

Across all experimental settings, the number of injected faults matches exactly the number of faults detected by SemFT, indicating that all injected errors were captured in a timely manner. Simultaneously, no files remained corrupted after recovery, demonstrating that SemFT successfully prevents anomalous results from propagating to the write-back stage. Both the EDR and DI reach 100% under every evaluated configuration in approximately 200,000 single-bit flip injections and no errors were ever silently accepted. In fault-free runs, we did not observe spurious recovery triggers. Moreover, the accuracy for CPU-based fault tolerance and heterogeneous collaborative execution is identical, showing that relocating execution from the CPU to the heterogeneous unit does not affect semantic decision logic or consistency.

Beyond correctness, the results show that SemFT also changes the distribution of error propagation. As shown in Fig. 5, the Hybrid scheme suppresses more fault-induced outputs than the other schemes, indicating stronger containment of SDCs. Meanwhile, the different empirical cumulative distribution function curves under different schemes suggest that SemFT not only reduces propagated errors but also changes their distribution characteristics.

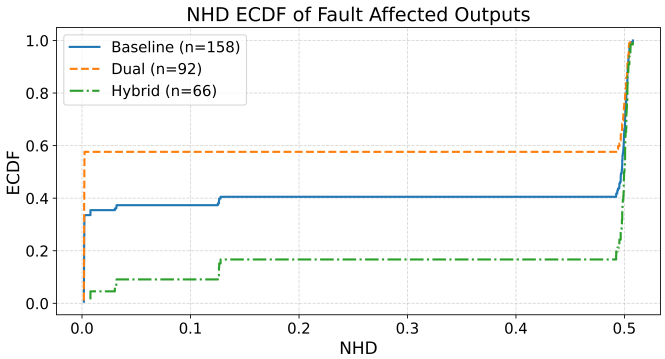


Fig. 5: Empirical cumulative distribution function of NHD under different schemes

Note that the curve does not describe residual SDCs after recovery. Instead, it summarizes the NHD distribution of fault-affected outputs observed before the final semantic decision and recovery stage. Since these outputs are later detected and corrected by SemFT, the figure reflects propagation patterns rather than escaped corruptions.

Conclusion: SemFT consistently achieves 100% SDC detection and recovery throughout the file-system encryption path.

C. RQ2: How does SemFT behave under normal throughput-oriented workloads?

RQ2 evaluates SemFT on the real encryption path under normal operating conditions. The workload is characterized by large-block sequential I/O and a sustained, non-overloaded request stream.

1) Performance breakdown: We break down the execution time of the encryption and decryption paths into several stages as illustrated in Fig. 6. The results show that the overhead is mainly concentrated in the redundant computation phase. The latency of the initial computation is low, whereas the second execution and the associated correction logic account for most of the additional cost. This indicates that the overhead of SemFT mainly comes from re-computation rather than from the normal execution itself. To further reduce this cost, we offload redundant computations to heterogeneous hardware execution units. Even so, the latency introduced by redundancy remains at the nanosecond level and has little impact on the overall service time or system performance.

2) Ablation Study: We conducted a series of ablation experiments by selectively enabling different redundancy strategies and comparing them against the baseline configuration. Across different levels of parallelism and fault injection rates, the performance remains largely flat as illustrated in Table III. Even when increasing the injection rate from 1% to 50%, the additional overhead is consistently bounded within about 3%. After enabling fault tolerance, the performance metrics

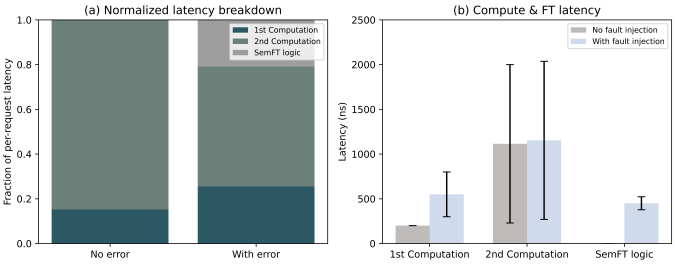


Fig. 6: Performance breakdown of SemFT under fault injection

exhibit only small deviations from the benchmark, indicating that SemFT incurs stable overhead.

Conclusion: SemFT achieves full error detection and recovery at $< 3\%$ bandwidth and latency overhead, with performance largely unaffected by the fault rate.

D. RQ3: How does the system scale under overload and high throughput workloads?

For RQ3, we construct an overload scenario with high concurrency, deep queue depth, and large data volume, pushing a single storage node to near or beyond its nominal capacity. With CPU, memory bandwidth, and I/O scheduling close to saturation, we evaluate whether enabling SemFT further degrades performance or harms scalability under contention. Latency and bandwidth results are shown in Table IV.

The elevated P99 latency in RQ3 primarily reflects the inherent tail amplification of high throughput overload: once concurrency and queue depth push the node into a queueing-dominated regime, small fluctuations in service time are magnified into long-tail delays. In this setting, the P99 increase should be attributed to system-level contention and scheduling bottlenecks rather than to deficiencies of SemFT, which does not introduce disproportionate degradation relative to the baseline trend.

Conclusion: SemFT remains resilient across varying fault probabilities under overload with high concurrency and high throughput.

E. RQ4: How does the mechanism behave under latency-sensitive workloads?

Latency-sensitive requests are highly affected by even small CPU or synchronization overheads. They typically involve small files, short I/O paths, and frequent invocations. Across all parameter settings in this workload, enabling SemFT does not cause observable long-tail latency inflation. We use the P99/average latency ratio, as shown in Fig. 8 to evaluate SemFT's stability. In latency-sensitive workloads, the P99/average latency ratio remains below 2, showing no pronounced long-tail latency and stable performance under SemFT.

TABLE III: Ablation study: comparing Baseline, Dual and Hybrid.

numjobs	Fault rate	Baseline		Dual				Hybrid			
		BW (MB/s)	Lat (μs)	BW (MB/s)	Lat (μs)	ΔBW (%)	ΔLat (%)	BW (MB/s)	Lat (μs)	ΔBW (%)	ΔLat (%)
1	0	852	4.43	856	4.41	+0.47	-0.45	873	4.32	+2.46	-2.48
	1%	849	4.44	844	4.47	-0.59	+0.68	871	4.33	+2.59	-2.48
	10%	840	4.49	842	4.48	+0.24	-0.22	873	4.32	+3.93	-3.79
	50%	852	4.43	846	4.46	-0.70	+0.68	860	4.39	+0.94	-0.90
2	0	1554	4.86	1551	4.87	-0.19	+0.21	1573	4.80	+1.22	-1.23
	1%	1542	4.90	1537	4.92	-0.32	+0.41	1561	4.84	+1.23	-1.22
	10%	1539	4.91	1535	4.92	-0.26	+0.20	1549	4.88	+0.65	-0.61
	50%	1538	4.91	1529	4.94	-0.59	+0.61	1550	4.87	+0.78	-0.81
4	0	2833	5.33	2755	5.48	-2.75	+2.81	2856	5.29	+0.81	-0.75
	1%	2794	5.40	2702	5.59	-3.29	+3.52	2825	5.34	+1.11	-1.11
	10%	2773	5.44	2724	5.55	-1.77	+2.02	2809	5.38	+1.30	-1.10
	50%	2777	5.44	2705	5.58	-2.59	+2.57	2814	5.37	+1.33	-1.29
8	0	3127	9.71	3080	9.86	-1.50	+1.54	3125	9.71	-0.06	+0.00
	1%	3122	9.72	3073	9.87	-1.57	+1.54	3121	9.72	-0.03	+0.00
	10%	3113	9.76	3050	9.94	-2.02	+1.84	3099	9.78	-0.45	+0.20
	50%	3109	9.77	3075	9.88	-1.09	+1.13	3093	9.82	-0.51	+0.51

This aligns with SemFT's semantic-level design: checks and corrections are performed after each data encryption, without inserting extra queueing or waiting into the I/O path, so overhead is only weakly dependent on file size or path length. With the page cache enabled, latency is dominated by modest redundant computation and comparison rather than storage-device bottlenecks, and thus is not amplified for smaller requests.

Fig. 9 shows that IOPS varies only slightly as the fault injection rate increases from 0% to 50%, with minor fluctuations rather than consistent drops. This suggests SemFT does not noticeably reduce file-level throughput under latency-sensitive workloads.

Conclusion: SemFT introduces small and stable latency overhead in latency-sensitive small-request workloads, while keeping tail behavior well-controlled: the **P99/AVG latency ratio remains below 2** across fault rates and access patterns, indicating no pronounced long-tail inflation and supporting safe deployment on latency-critical paths.

F. RQ5: How much does the heterogeneous mechanism improve system performance?

We significantly reduce the load on the main CPU through the offloading mechanism. We measured both the per-request response performance and the overall overhead of the entire processing workflow for the two methods. Table V reports the detailed breakdown of processing stages, comparing execution times for CPU-only and I/O Die configurations. It highlights that stages such as AES encryption benefit from offloading, resulting in a reduction of total end-to-end time from 602.40 ms on the CPU to 585.28 ms on the I/O Die. Figure 7 presents the overall performance comparison, showing throughput and per-request latency. The system fully utilizes heterogeneous hardware resources, improving performance without compromising fault-tolerance guarantees.

TABLE IV: High-throughput performance under fault injection.

Fault (%)	BW (MB/s)		Avg Lat (μs)		P99 (μs)	
	Dual	Hybrid	Dual	Hybrid	Dual	Hybrid
Config: numjobs=64, iodepth=32						
0	5490	5491 (+0.02%)	2342	2368 (+1.09%)	23462	23725 (+1.12%)
1	5415	5509 (+1.74%)	2366	2362 (-0.15%)	23725	23725 (0.00%)
10	5414	5469 (+1.02%)	2395	2401 (+0.23%)	23987	23987 (0.00%)
50	5472	5495 (+0.42%)	2363	2336 (-1.14%)	23725	23725 (0.00%)
Config: numjobs=128, iodepth=32						
0	5218	5226 (+0.15%)	4784	4781 (-0.06%)	47973	47973 (0.00%)
1	5211	5255 (+0.84%)	4840	4781 (-1.21%)	48497	47973 (-1.08%)
10	5227	5043 (-3.52%)	4847	4793 (-1.11%)	48497	47973 (-1.08%)
50	5286	5132 (-2.91%)	4732	4831 (+2.09%)	47449	48497 (+2.21%)
Config: numjobs=128, iodepth=64						
0	5104	5073 (-0.61%)	6967	6955 (-0.17%)	69731	69731 (0.00%)
1	5155	4982 (-3.36%)	6949	6938 (-0.16%)	69731	70779 (+1.50%)
10	5106	5004 (-2.00%)	7055	6951 (-1.47%)	70779	69731 (-1.48%)
50	5126	5015 (-2.17%)	7052	6946 (-1.51%)	70779	69731 (-1.48%)

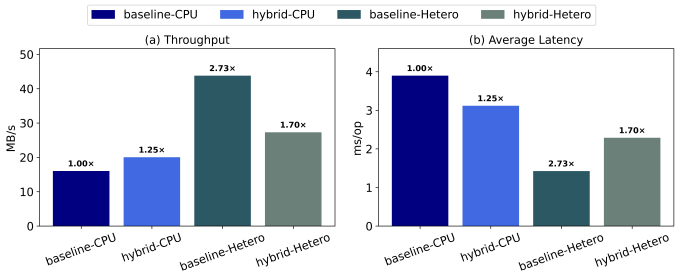


Fig. 7: Performance comparison between CPU-only and I/O Die solution

TABLE V: Performance comparison between CPU-only and I/O Die solution

Stage	CPU (ms)	I/O Die (ms)
Memcpy copy time	27.59	-
AES encryption time	574.81	-
Write time	-	365.57
Read time	-	160.56
Trigger + wait time	-	59.15
Total time	**602.40**	**585.28**

Conclusion: Heterogeneous offloading of redundancy achieves up to 2.73× performance improvement compared to the baseline CPU-only configuration, while preserving the CPU-side semantic decision logic.

G. Conclusion and Analysis of Experimental Results

The low overhead and stable tail latency of SemFT stem from multiple design optimizations. Semantic-level checking and correction occur after each encryption operation rather than continuously monitoring the full path, so most computations incur no extra work; overhead arises only during recomputation or verification, ensuring low average latency. The majority of the additional cost is concentrated in redundant computation, while single encryption latency remains minimal. Offloading redundancy to heterogeneous hardware allows multiple redundant computations to execute in parallel without blocking the CPU, reducing load and improving parallelism, which stabilizes throughput and latency.

With page cache enabled, most I/O requests hit memory rather than storage, mitigating storage latency fluctuations. Combined with lightweight semantic checks, tail latency remains stable, keeping the P99-to-average ratio low. Experiments under multi-threaded, high-concurrency, and deep I/O queue scenarios show good scalability. Across both throughput-oriented and latency-sensitive workloads, SemFT maintains low overhead and stable tail behavior, demonstrating its practicality for real-world cloud storage deployment.

V. Related Work

To solve SDC in cloud environments, researchers have proposed a variety of solutions from both hardware and application perspectives [28]. Various models have also been developed for fault injection and error propagation simulation [25]–[27]. At the software level, many cloud service providers have also introduced approaches to mitigate the impact of hardware faults on data integrity, either from a spatial or temporal perspective [15], [16]. These strategies encompass multi-level optimizations ranging from low-level hardware design to high-level application algorithms.

(1) Hardware-level Fault Tolerance. Several studies focus on the hardware-based fault tolerance mechanisms [28]. Kasap et al. developed a rollback/roll-forward mechanism for SoCs, ensuring faster error recovery and minimal downtime [32]. Similarly, Sim et al. proposed a dual-lockstep SoC design that enables seamless error recovery by switching between processor cores [31]. Khalil's NoC method enables dynamic fault-tolerance policies, providing redundant paths and real-time fault recovery in networked systems [29]. Marcinek et al. introduced the VDCLS architecture, which dynamically adjusts synchronization latency between cores to balance performance and reliability [35].

(2) Software-level Fault Tolerance. At the software level, cloud platforms such as AWS and Azure use multi-copy redundancy and failover strategies to ensure data integrity. While these methods effectively mitigate data loss due to hardware failures, they incur significant storage overhead and slow recovery times due to their reliance on redundancy.

(3) Large-scale Cluster-level Fault Tolerance. Bacon et al. proposed a combined hardware-software fault tolerance mechanism for SDC detection in large-scale systems, using redundancy coding, memory protection, and error detection algorithms to minimize data loss [4]. Zhang et al. introduced Parallaft, a runtime-based method leveraging heterogeneous parallelism for rapid recovery, making it well-suited for large-scale cluster environments [36]. Dutta et al. proposed Hardware Sentinel, a large-scale SDC detection framework that monitors software-level symptoms to identify faulty CPUs in production, achieving higher detection coverage than traditional testing tools [11].

(4) Fault Injection Tools. Existing tools, such as LLVM-Based Fault Injection Tool (LLFI) and FAIL*, support error simulation at various levels but are limited in system-level applications or performance due to virtualized environments [48]. F-SEFI can simulate instruction-level errors but is not open-source, limiting reproducibility [49].

VI. Discussion and Future Work

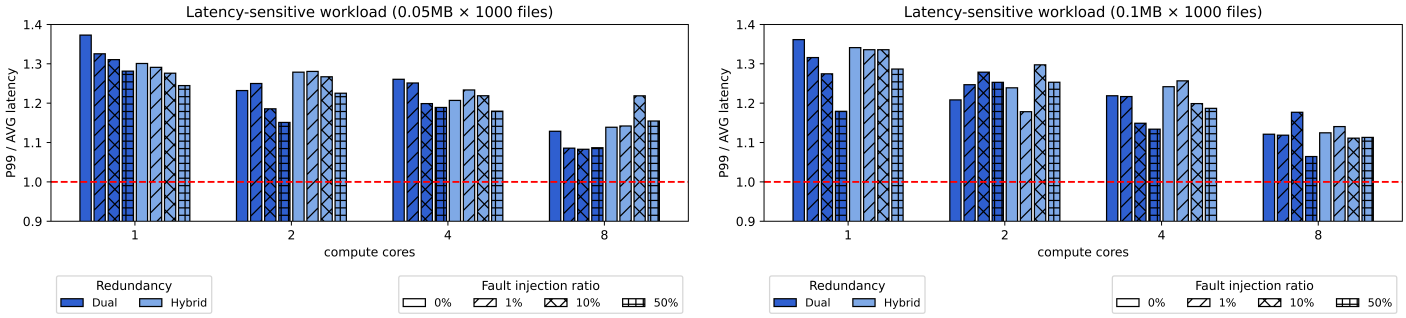
A. Scope and Limitations

SemFT demonstrates how semantic-level fault tolerance can be applied along the file-system encryption path to prevent SDC propagation. While the current evaluation focuses on single-node encryption under controlled fault injection, the design principles such as semantic units, output boundary checkpoints, and decision-driven redundancy are broadly applicable to other deterministic transformation operators and storage systems. The heterogeneous offloading mechanism provides additional computation capacity under resource contention, but the CPU retains control over semantic decisions, ensuring correctness. Consequently, the framework is conceptually extensible to larger-scale and distributed storage deployments, although detailed performance and coordination behavior may vary in such environments.

B. Future Work

There are two important directions for extending SemFT.

First, although the current prototype demonstrates strong reliability and low overhead on the AES-based eCryptfs path under controlled fault injection, future work should broaden both the fault model and the system scope. In particular, it is important to evaluate SemFT under more diverse fault



(a) Latency-sensitive (0.05MB×1000)

(b) Latency-sensitive (0.1MB×1000)

Fig. 8: P99 / AVG latency ratio in latency-sensitive workloads

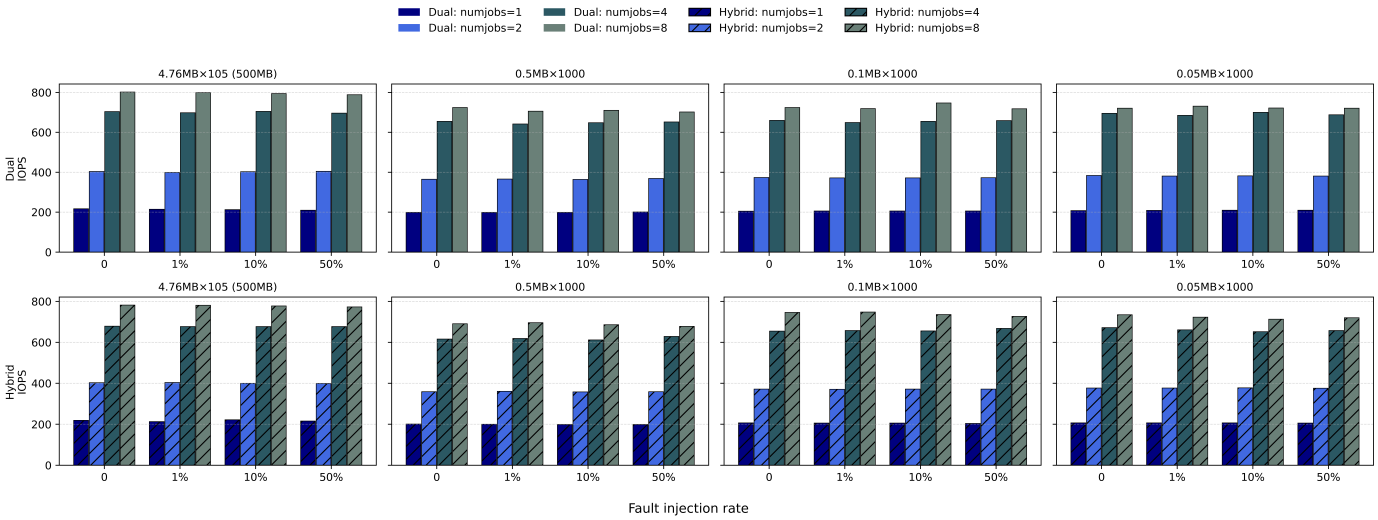


Fig. 9: IOPS variation in latency-sensitive workloads

conditions, such as correlated transient faults, burst errors, and multi-point disturbances across computation and data movement paths. At the same time, the framework should be extended to additional encryption modes, storage configurations, and deterministic data-transformation operators, in order to examine how semantic validation generalizes across broader system settings.

Second, while this work focuses on preventing SDC propagation within a single-node encrypted file-system path, an important next step is to extend semantic fault tolerance to distributed storage workflows. Future work will investigate how semantic validation can be incorporated into replica synchronization, cross-node verification, and distributed commit paths, so that semantically incorrect results can be detected and blocked before they propagate across replicas. This would further strengthen end-to-end data integrity guarantees in large-scale cloud storage systems.

VII. Conclusion

This paper targets SDC on the encryption path in cloud storage. We propose and implement SemFT, a semantic-level fault-tolerance framework deployed at the file-system layer. SemFT combines configurable redundant execution with semantic invariants to perform online detection and immediate recovery within the lifetime of a single I/O request, without requiring changes to application interfaces or hardware. End-to-end fault injection experiments show that SemFT achieves 100% detection and data integrity restoration across fault rates and workload types. In throughput-oriented and overload settings, its performance remains stable and largely insensitive to the fault rate. Under latency-sensitive small-request workloads, SemFT does not introduce noticeable tail-latency amplification, while keeping the runtime overhead consistently below 3%.

Future work includes extending the framework to a broader set of transformation operators and incorporating resource-aware scheduling to adapt redundancy policies under constrained compute capacity, enabling a better cost–reliability balance in large-scale heterogeneous clusters.

References

[1] P. Shah and W. So, "Lamassu: storage-efficient host-side encryption," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15. USENIX Association, 2015, p. 333–345.

[2] Y. Zhou, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhang, and C. Li, "Secdep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '15, 2015, pp. 1–14.

[3] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, 2021, pp. 9–16.

[4] D. F. Bacon, "Detection and prevention of silent data corruption in an exabyte-scale database system," in *The 18th IEEE Workshop on Silicon Errors in Logic–System Effects (SELSE)*, 2022.

[5] A. Mahmoud, R. Venkatagiri, K. Ahmed, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve, "Minotaur: Adapting software testing techniques for hardware errors," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 1087–1103.

[6] M. A. Shahid, N. Islam, M. M. Alam, M. S. Mazliham, and S. Musa, "Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment," *Computer Science Review*, vol. 40, p. 100398, 2021.

[7] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[8] M. Kishani, M. Tahoori, and H. Asadi, "Dependability analysis of data storage systems in presence of soft errors," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 201–215, 2019.

[9] G. Papadimitriou and D. Gizopoulos, "Silent data corruptions: Microarchitectural perspectives," *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3072–3085, 2023.

[10] D. Gizopoulos, G. Papadimitriou, O. Chatzopoulos, N. Karystinos, H. D. Dixit, and S. Sankar, "Silent data corruptions in computing systems: Early predictions and large-scale measurements," in *2024 IEEE European Test Symposium (ETS)*, 2024, pp. 1–10.

[11] R. Dutta, H. D. Dixit, R. V. Riel, G. Vunnam, and S. Sankar, "Hardware sentinel: Protecting software applications from hardware silent data corruptions," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS)*, 2025, pp. 482–497.

[12] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, "Understanding silent data corruptions in a large production CPU population," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023, pp. 216–230.

[13] D. Gizopoulos, G. Papadimitriou, and O. Chatzopoulos, "Estimating the failures and silent errors rates of CPUs across ISAs and microarchitectures," in *2023 IEEE International Test Conference (ITC)*, 2023, pp. 377–382.

[14] T. Macieira, S. Gurumurthy, S. Gurumurthi, A. Haggag, G. Papadimitriou, and D. Gizopoulos, "Silent data corruptions in computing: Understand and quantify," in *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2024, pp. 1–7.

[15] M. Brandenburger, C. Cachin, and N. Knežević, "Don't trust the cloud, verify: Integrity and consistency for cloud object stores," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, pp. 1–30, 2017, available at https://doi.org/10.48550/arXiv.1502.04496.

[16] C.-T. Huang, L. Huang, Z. Qin, H. Yuan, L. Zhou, V. Varadharajan, and C.-C. J. Kuo, "Survey on securing data storage in the cloud," *APSIPA Transactions on Signal and Information Processing*, vol. 3, p. e7, 2014, available at https://doi.org/10.1017/ATSIP.2014.6.

[17] S. Di and F. Cappello, "Adaptive impact-driven detection of silent data corruption for HPC applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2809–2823, 2016.

[18] O. Mutlu, A. Olgun, and A. G. Yağlıkcı, "Fundamentally understanding and solving rowhammer," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2023, pp. 461–468.

[19] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and G. R. Goodson, "An analysis of data corruption in the storage stack," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, pp. 1–28, 2008.

[20] H. Dixit, "Keytone: Silent data corruptions at scale," in *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2023, pp. 1–2.

[21] C. Liu, Z. Zhu, Q. Li, Y. Xia, Y. Qiao, X. Deng, Y. Lu, T. Xie, H. Cui, Z. Du, H. Xu, and C. Wang, "Orthrus: Efficient and timely detection of silent user data corruption in the cloud with resource-adaptive computation validation," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, ser. SOSP '25, 2025, p. 286–304.

[22] S. Ahmed, M. Nahiduzzaman, T. Islam, F. H. Bappy, T. S. Zaman, and R. Hasan, "FASTEN: Towards a FAult-tolerant and STorage EfficieNt cloud: Balancing between replication and deduplication," in *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*, 2024, pp. 44–50.

[23] X. Xie, C. Wu, J. Gu, H. Qiu, J. Li, M. Guo, X. He, Y. Dong, and Y. Zhao, "Az-code: An efficient availability zone level erasure code to provide high fault tolerance in cloud storage systems," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '15, 2019, pp. 230–243.

[24] C. Chang, G. Li, and M. Erez, "Evaluating compiler IR-level selective instruction duplication with realistic hardware errors," in *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, 2019, pp. 41–49.

[25] Z. Li, H. Menon, K. Mohror, P. Bremer, Y. Livant, and V. Pascucci, "Understanding a program's resiliency through error propagation," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2021, pp. 362–373.

[26] G. Li and K. Pattabiraman, "Modeling input-dependent error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 279–290.

[27] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 27–38.

[28] S. Venkatesha and R. Parthasarathi, "Survey on redundancy based-fault tolerance methods for processors and hardware accelerators – trends in quantum computing, heterogeneous systems and reliability," *ACM Computing Surveys*, vol. 56, pp. 1–76, 2024.

[29] K. Khalil, A. Kumar, and M. Bayoumi, "Dynamic fault tolerance approach for network-on-chip architecture," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, pp. 384–394, 2024.

[30] Z. Guz, H. H. Li, A. Shayesteh, and V. Balakrishnan, "Performance characterization of NVMe-over-fabrics storage disaggregation," *ACM Transactions on Storage (TOS)*, vol. 14, no. 4, pp. 1–18, 2018.

[31] M. T. Sim and Y. Zhuang, "A dual lockstep processor system-on-a-chip for fast error recovery in safety-critical applications," in *IECON 2020 – The 46th Annual Conference of the IEEE Industrial Electronics Society*, Singapore, 2020, pp. 2231–2238.

[32] S. Kasap, E. W. Wächter, X. Zhai, S. Ehsan, and K. D. McDonald-Maier, "Novel lockstep-based fault mitigation approach for socs with roll-back and roll-forward recovery," *Microelectronics Reliability*, vol. 124, p. 114297, 2021.

[33] Z. Zeng, C. Zhu, and S. M. Goetz, "Fault-tolerant multiparallel three-phase two-level converters with adaptive hardware reconfiguration," *IEEE Transactions on Power Electronics*, vol. 39, no. 4, pp. 3925–3930, 2024.

[34] Y. Deng, X. Shi, Z. Jiang, X. Zhang, L. Zhang, Z. Zhang, B. Li, Z. Song, H. Zhu, G. Liu, F. Li, S. Wang, H. Lin, J. Ye, and M. Yu, "Minder: Faulty machine detection for large-scale distributed model training," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 505–521.

[35] K. Marcinek and W. A. Pleskacz, "Variable delayed dual-core lockstep (VDCLS) processor for safety and security applications," *Electronics*, vol. 12, no. 2, p. 464, 2023.

[36] B. Zhang, S. Ainsworth, L. Mukhanov, and T. M. Jones, "Parallaft: Runtime-based CPU fault tolerance via heterogeneous parallelism," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 2025, pp. 584–599.

[37] C. Xiao, L. Zhang, W. Liu, L. Cheng, P. Li, Y. Pan, and N. Bergmann, "Nv-ecryptfs: Accelerating enterprise-level cryptographic file system with non-volatile memory," *IEEE Transactions on Computers*, vol. 68, no. 9, pp. 1338–1352, 2019.

[38] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, New York, NY, USA, 2021, p. 756–771.

[39] Z. Duan, H. Feng, H. Liu, X. Liao, H. Jin, and B. Li, "Aegonkv: a high bandwidth, low tail latency, and low storage cost kv-separated lsm store with smartssd-based gc offloading," in *Proceedings of the 23rd USENIX*

Conference on File and Storage Technologies, ser. FAST '25. USENIX Association, 2025.

[40] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. R. K. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam, "Leapio: Efficient and portable virtual nvme storage on arm socs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, New York, NY, USA, 2020, p. 591–605.

[41] Z. Ruan, T. He, and J. Cong, "Insider: designing in-storage computing system for emerging high-performance drive," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19, 2019, p. 379–394.

[42] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, "Flash storage disaggregation," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, 2016.

[43] W. Zhu, G. Hernandez, W. Garcia, D. J. Tian, S. Rampazzi, and K. R. B. Butler, "Srftl: Leveraging storage semantics for effective ransomware defense in flash-based ssds," *ACM Trans. Storage*, vol. 21, no. 4, 2025.

[44] Z. Li, H. Menon, D. Maljovec, Y. Livnat, S. Liu, K. Mohror, P.-T. Bremer, and V. Pascucci, "Spotsdc: Revealing the silent data corruption propagation in high-performance computing systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 10, pp. 3938–3952, 2021.

[45] D. Bittman, M. Gray, J. Raizes, S. Mukhopadhyay, M. Bryson, P. Alvaro, D. D. Long, and E. L. Miller, "Designing data structures to minimize bit flips on nvm," in *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2018, pp. 85–90.

[46] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, D. Srinivasan, B. Panda, A. Baptist, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li, "Fail-slow at scale: Evidence of hardware performance faults in large production systems," *ACM Trans. Storage*, vol. 14, no. 3, 2018.

[47] A. Höller, G. Schönfelder, N. Kajtazovic, T. Rauter, and C. Kreiner, "FIES: a fault injection framework for the evaluation of self-tests for COTS-based safety-critical systems," in *2014 15th International Microprocessor Test and Verification Workshop (MTV)*, 2014, pp. 105–110.

[48] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2015, pp. 11–16.

[49] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 1245–1254.