RIVA: Robust Integrity Verification Algorithm for High-Speed File Transfers

Batyr Charyyev and Engin Arslan, *Member, IEEE*

Abstract—End-to-end integrity verification is designed to protect file transfers against silent data corruption by comparing checksum of files at source and destination end points using cryptographic hash functions such as MD5 and SHA1. However, existing implementations of end-to-end integrity verification for file transfers fall short to detect undetected disk errors that causes inconsistency between disk and cache memory. In this paper, we propose Robust Integrity Verification Algorithm (RIVA) to strengthen the integrity of file transfers by forcing checksum computation tasks to read files directly from disk. RIVA achieves this by invalidating memory mappings of file pages after their transfer such that when the file is read again for checksum calculation, it will be fetched from disk and silent disk errors will be captured. We design and conduct extensive fault resilience experiments to evaluate the robustness of integrity verification algorithms against undetected disk write errors. The results indicate that while the state-of-the-art integrity verification algorithms fail to detect the injected errors for almost all file sizes, RIVA captures all of them with the help of cache invalidation. We further run statistical analysis to assess the probability of missing silent disk errors and find that RIVA reduces the likelihood by 10 to 15 orders of magnitude compared to the existing approaches. Finally, enforcing disk read in integrity verification introduces an inevitable overhead in exchange of increased robustness against silent disk errors, but RIVA keeps its overhead below 15% in most cases by running transfer, cache invalidation, and checksum computation processes concurrently for different portions of the same file.

Index Terms—End-to-end integrity verification, file transfers, high performance networks, undetected disk errors, silent data corruption.

1 Introduction

Large scientific experiments such as environmental and coastal hazard prediction [1], climate modeling [2], genome mapping [3], and high-energy physics simulations [4], [5] generate data volumes reaching petabytes per year. This massive amount of data often needs to be moved to geographically distributed sites for various purposes including processing, collaboration, and archival. The integrity of transfers is crucial for many applications whose computations are extremely sensitive to the content of the data such as Dark Energy Survey [6] and Sky Survey [7] projects. Although some components of file transfers have built-in integrity verification mechanisms, they are either weak or available only in a subset of network and end systems. For example, TCP uses 16-bit checksum to capture data corruption but it fails to detect errors once in 16 million to 10 billion packets [8], which is not rare in today's high-speed networks that operate at the speed of hundreds of gigabits per second.

Moreover, data corruption can also happen at storage systems during file read and write operations as disk drives are prone to silent data corruption referred as undetected disk error (UDE) [9]–[11]. UDEs occur mainly due to firmware or hardware malfunctions in disk drives and corrupt file contents silently. UDEs are categorized into two groups as undetected read error (URE) and undetected write error (UWE). UREs manifest as mostly transient errors and are unlikely to affect system state beyond causing transfer repetitions. UWEs, on the other hand, are persistent errors which are only detectable during a read operation subsequent to the faulty write, and thus posing a significant threat to data reliability [9].

Storage systems implement several techniques to detect and recover from UDEs such as file system scrubbing and RAID reconstruction, studies show that errors can still go undetected [10]. As an example, parity-enabled RAID architectures are designed to be more resilient against disk failures and latent sector errors, however previous studies found that they are also susceptible to silent data corruption [9]–[11]. Although several techniques have been proposed to prevent UDEs in RAID systems, the empirical results show that they incur up-to 43% performance overhead [12]. Moreover, even if disks are error-proof, data corruption can still happen during data transmission from memory to disk due to faulty cables and firmware bugs, thus a comprehensive solution is required to prevent silent errors to go undetected.

Application-layer end-to-end integrity check is proved to be a robust solution to detect and recover from silent data corruption as it relies on secure cryptographic hash functions and covers all operations in between [13]–[17]. A typical implementation of end-to-end integrity verification for file transfers works as follows: Sender first reads the file from disk and sends it to receiver to save it. Once data transfer is completed, the sender reads the file again to compute its checksum using a hash algorithm such as MD5 and SHA1. Receiver also reads the file to compute the checksum and then sends it to the sender to compare. If the checksum values match, then the transfer is marked as successful. Otherwise, the file at the receiver side is assumed

- *B. Charyyev is with Stevens Institute of Technology. E-mail: bcharyye@stevens.edu*
- *E. Arslan is with University of Nevada, Reno*

Mr. Charyyev was with the University of Nevada, Reno when this work was performed.

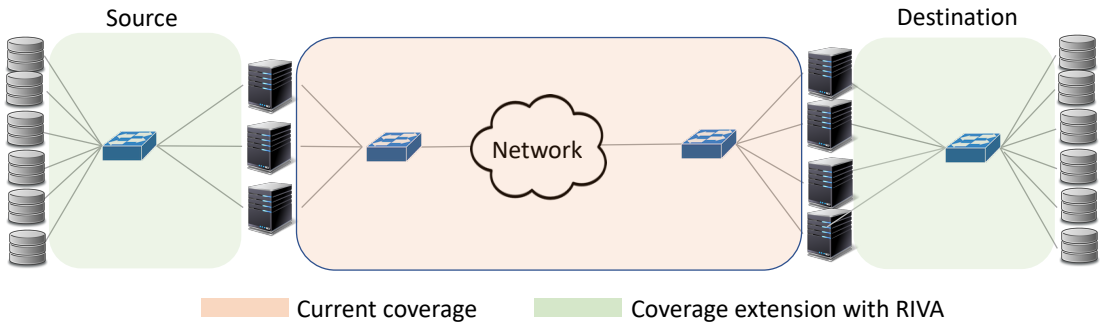


Fig. 1: RIVA extends the coverage of end-to-end integrity verification by checking for errors that might happen between memory and storage subsystems.

to be corrupted and the transfer is restarted. However, we discover that existing implementations of end-to-end integrity verification for data transfers are vulnerable to UDEs due to calculating checksum on cached data which might be different than the disk copy in the presence of UDEs.

In this paper, we propose Robust Integrity Verification Algorithm (RIVA) to detect and mitigate silent data corruptions for file transfers by enforcing checksum calculations to read files directly from disk. RIVA works as follows: When the transfer of a file is completed, RIVA first finds the virtual address space of the file. Then, it deletes the mappings for the specified address space such that further references to the address space will generate invalid memory references (i.e., page faults). As a result, when the file is read to calculate its checksum, operating systems (OS) will not be able to locate the file pages in page cache and will fetch them directly from the disk, allowing the detection of UDEs that might have happened while transferring the file data from memory to disk. It is important to note that by invalidating cached copy of files before comparing checksums, *RIVA extends the coverage of end-to-end integrity verification without losing existing error-detection capabilities (e.g., capturing network errors)* as showed in Figure 1. We conducted extensive experiments using different network, storage subsystem, and dataset configurations and observed that unlike state-of-the-art solutions, RIVA always captures injected disk errors even under extreme conditions. RIVA can also capture errors that may happen while transmitting data from memory to disk, however this work only evaluates its capability in detecting silent disk errors. On the other hand, invalidating memory copy of the files and forcing OS kernels to read data from storage increases the execution time of transfers. But, RIVA keeps its overhead at minimum by splitting large files into blocks and concurrently executing transfer, cache eviction, and checksum computation operations for different blocks. Contributions of this paper are as follows:

- We propose RIVA to enhance resilience of end-to-end integrity verification for data transfers against UDEs by enforcing checksum calculations to read files directly from disk.
- We introduce an extreme fault injection technique to reproduce undetected disk write errors and evaluate RIVA and the state-of-the-art integrity verification algorithms in terms of the reliability against such errors.
- We conduct extensive experiments using variety of network, dataset, and fault injection scenarios to evaluate robustness and performance of RIVA.
- We calculate the likelihood of silent data corruption with and without RIVA and confirm that RIVA reduces the probability of UDEs to go undetected by a significant margin.

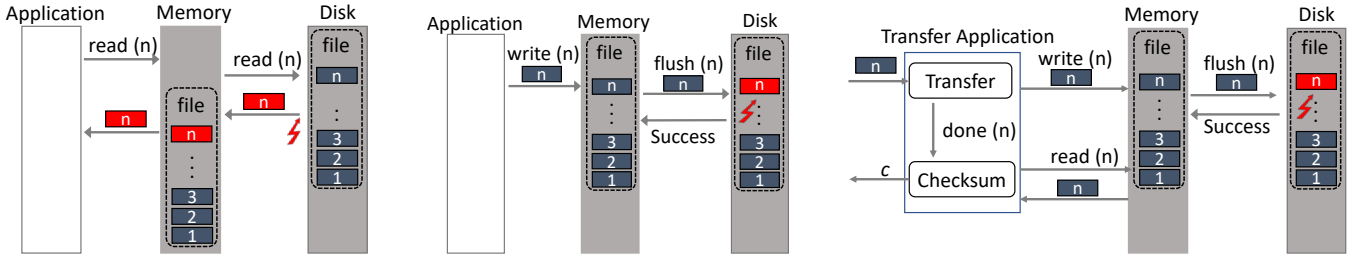
The rest of the paper is organized as follows: Section 2 defines undetected disk errors and describes the typical approach to run end-to-end integrity verification for file transfers. Section 3 presents related work and Section 4 details design principles of the proposed solution. Section 5 discusses experimental results and Section 6 concludes the paper with the summary and future directions.

2 Background and Motivation

In this section, we provide background on the end-to-end integrity verification and undetected disk errors.

2.1 Undetected Disk Errors

UDEs can be caused by several reasons such as Latent Sector Errors (LSEs), lost, torn and off-track writes [10], [11], [18]. LSEs stem from physical problems within the disk drive such as media scratches and mostly detected by drive's internal error-correcting codes (ECC) [10]. Lost writes also referred as dropped writes that occur when the write head fails to overwrite existing data on a track, causing the disk to remain in its previous state as if the write never occurred. In torn writes disk drives end up writing only a portion of the sectors in a given write request and the rest of the sector contains stale data. This often occurs when the drive is power-cycled in the middle of processing the write request. Off-track writes also referred as misdirected writes which happen when write head is not properly aligned with target track and data is partially or completely written to a adjacent tracks. Near off-track writes happen when the data is written to the gap between tracks adjacent to the intended track. As a result, the original disk location does not receive the write it is supposed to receive (lost write) and data in a adjacent track is overwritten. LSEs occur more frequent compared to other errors affecting about 19% of nearline and about 2% of enterprise class disks within 2 years of use [11]. Luckily, they can mostly be detected by existing fault tolerance techniques of storage systems. On the other hand, lost, torn, and off-track writes can go undetected.



(a) Undetected Read Error (URE) (b) Undetected Write Error (UWE) (c) UWE-exposed transfer

Fig. 2: Undetected disk errors can happen during file read (a) and write (b) operations. When undetected read errors happens, page cache keeps corrupt data while disk holds genuine version. In case of undetected write error, corrupted data is written to disk while genuine version is cached in memory. If checksum computation is performed on cached data, UWEs can cause permanent data loss for file transfers (c).

Krioukov et al. reported that lost and off-track writes occur in about 0.04% of nearline and 0.007% of enterprise class disks within the first 17 months of operation whereas these values for torn writes are 0.6% and 0.06% respectively [11]. In a different study, Bairavasundaram et al. estimated UDE probability to be between 10^{-12} and 10^{-14} [10], [19].

Undetected disk errors can be categorized into two groups as undetected read errors (URE) and undetected write errors (UWE). UREs cause applications to see a different version of data than the one stored on the disk. As shown in Figure 2(a), while disk hosts the genuine data, URE leads to corrupted file page n to be served to the data transfer application. On the other hand, UWEs corrupt data while it is being written to disk. In Figure 2(b), file page n is exposed to a UWE during disk write operation despite the *success* response. UWEs can corrupt one or more bits of a file page and if not recovered would lead to permanent data loss.

2.2 End-to-end Integrity Verification

A simple implementation of integrity verification for file transfers (aka sequential) involves three steps. In the first step, the file is read from the disk of source server and transferred to destination. Once the transfer is finished and the file is written to the disk at the destination, the checksum of original file at the source and the transferred copy at destination are computed using a hash function such as MD5 or SHA1 as a second step. In the third and final step, checksum values of the original file and the transferred copy are exchanged between the source and the destination servers to compare. If they match, then the file transfer is assumed to be successful. Otherwise, the transferred copy of the file at destination is considered corrupt and file is transferred again.

The main objective of running end-to-end integrity check is to detect possible data corruption by comparing the checksum of a file at the source and the destination servers. On the other hand, operating systems are designed to minimize cache misses, so if a file is recently read or written, it will be kept in the memory to optimize successive accesses to the file content. In turn, this causes checksum computation to access the cached copy of the file when file size is small, preventing the detection of silent data corruption that may occur during disk read and write operations at the transfer phase. For example, if a URE happens at the source server during the transfer of a file, then receiver will receive and store corrupted data on the destination server. After file transfer completes, both sender and receiver read the file again to compute and compare checksum values to verify the integrity of the transfer. If the file size is small, then the OS kernel on the sender side will keep the corrupt data on page cache, causing the checksum calculation to be executed based on the corrupted data. This will then trigger checksum match, so transfer of the file will be deemed as successful despite the URE. Similarly, if file write operation on the receiver side is exposed to UWE and checksum is computed based on cached copy, then UWE will be missed since the page cache would still hold the correct version of the file. Figure 2(c) illustrates how integrity verification can fail to capture UWEs. The transfer application receives the page n and writes it to disk during which UWE corrupts the page. However, OS still keeps the genuine copy on memory uses it when checksum thread attempts to read the file. As a result, the checksum c is calculated based on genuine data and the UWE goes undetected.

Transfer of a file with integrity verification involves three file read and one file write operations. The first file read happens when sender reads the file from the disk for the first time to initiate the transfer. The other two file read operations take place when both sender and receiver read their copy of the file to calculate the checksum. The file write is done at the receiver server to save the transferred file to storage system. Hence, three file read and one file write operations can be exposed to UREs and UWEs. Although UWEs pose more serious threat to the integrity of transfers, UREs can also lead permanent data loss when integrity verification operates on cache data as described in Section 5.2.3.

2.3 Motivating Example

As running checksum computation on cache makes file transfers vulnerable to UDEs, we evaluated the sequential end-to-end integrity verification approach in terms of its receiver-side cache behaviour during the checksum computation process in HPCLab-WS network where servers are equipped with 16 GB of RAM and direct-attached hard drives. We transferred a mixed dataset that contains 273 files with total size of 274 GB. Out of 273 files in the dataset, only three files are larger than the memory size. We monitored page miss value on the receiver server to infer disk access behavior. Increased page miss value can be used to deduce

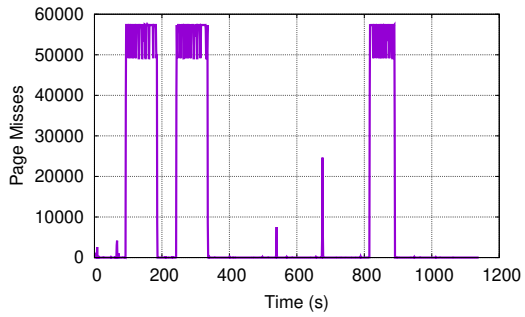


Fig. 3: Transferring a mixed dataset with integrity verification reveals that only files that are larger than memory size contribute to page misses during the checksum computation. Thus, majority of files are vulnerable to undetected disk errors.

disk read for checksum computation since receiver does not perform read I/O during the transfer phase. The Figure 3 demonstrates that page misses occur in three intervals during which checksum process attempts to read the three large files. For the remaining 270 files, operating system returns file access requests from cache memory (i.e. page hit) as they are cached during the transfer. The total number of page misses is around 145M that corresponds to 56GB, the total size of three large files. We also confirmed that other implementations of integrity verification (i.e. block-level [20] and FIVER [21]) exhibit similar behavior of reading small files from page cache during checksum calculation. Hence, existing integrity verification approaches for file transfers are susceptible to UDEs for files that are smaller than the memory size. A recent analysis on data transfers in high performance computing facilities revealed that average file size of file transfers is in the order of megabytes [22]. When combined with the fact that most production systems use 64 GB or larger memory units in their transfer nodes, it raises significant concerns on the reliability data transfers despite enabling end-to-end integrity verification. Moreover, even if a storage system is resilient to UDEs, data path between memory and storage can cause silent data corruptions due to faulty cables and driver firmware bugs. As a result, despite the availability of integrity verification solutions, file transfers are still vulnerable to silent data corruption since existing implementations of integrity verification fall short to offer an "true" end-to-end coverage.

3 Related Work

In this section, we discuss related work on high-performance data transfers, end-to-end integrity verification, and data corruption in storage systems.

High-performance data transfers: High-speed data transfer studies mostly focus on scheduling [23], [24], throughput optimization [25]–[27], and power consumption optimization [28]. In a recent study, Yun et al. proposed ProbData to tune the number of parallel streams and buffer size for TCP transfers using stochastic approximation [27]. ProbData uses Simultaneous Perturbation Stochastic Approximation algorithm to identify optimal transfer configurations for TCP and UDP based transport methods. Rao et al. presented stochastic gradient descent based algorithm to discover the number of parallel TCP streams that yields the maximum network throughput [29]. HARP runs regression analysis based on historical data and active probing, and then uses the derived model to explore the application layer transfer parameters that maximize transfer throughput [25], [30]. Alan et al. [28] proposed energy-efficient data transfer algorithms to tune application layer parameters, and find a balance between transfer throughout and energy consumption at the end hosts. They monitor CPU usage of end hosts and estimate energy consumption with the help of models that relate CPU usage to energy consumption. A cost function is then used to measure the energy efficiency of different configurations based on throughput and energy consumption results.

Integrity verification: Researchers studied integrity verification in the context of storage outsourcing [15], [31], [32], long term archiving [16], [33], file systems [34]–[36], databases [17], provenance [37], and data transfer [20], [21], [38]. Zhang et al. [36] evaluated Zetabyte Files System (ZFS) in terms of robustness to disk and memory fault injections. It has been found that while ZFS is able to detect and mostly recover from disk corruptions with the help of end-to-end integrity verification of data blocks, it is vulnerable to memory corruptions since it does not check the integrity of data blocks when they reside in the memory.

Globus [39] supports end-to-end integrity verification for data transfers. It pipelines transfer and checksum computation processes for different files to minimize the overhead of integrity verification. However, its pipelining technique does not work as expected when a dataset consist of files with mixed sizes. Liu et al. propose block-level pipelining to optimize integrity verification for mixed size datasets by dividing large files into blocks [20]. It reduces execution time considerably especially when dataset is composed of files with mixed sizes, however it requires careful tuning of block size to perform well. In a previous work, we proposed Fast Integrity Verification Algorithm (FIVER) to run transfer and checksum operations of each file simultaneously and enable I/O sharing between these the two [21]. FIVER outperforms file-level and block-level pipelining solutions by reducing the overhead of integrity verification from up-to 60% to less than 10%. However, we discovered that none of the existing file transfer end-to-end integrity verification algorithms can detect injected disk errors, thus they all fail to offer "true" end-to-end coverage.

Data corruption in storage systems: Studies on disk fault analysis investigate drive failures [40]–[42], latent sector errors [19], and data corruption [10], [11], [36]. Shah et al. investigated the underlying reasons for disk failures and identified several factors including media errors, scratch in disk, high-fly writes, rotational vibration, hard particles, and head slap [41]. Schroeder et al. [42] analyzed data from 100,000 disks over a five- year period and found that disk failures have positive correlation with disk ages. Hence, most modern storage systems store checksum of file blocks next to the block in the disk to detect data corruption, however it can develop checksum mismatch which is defined as the discrepancy between the stored checksum and calculated checksum of a block. It can happen because of several reasons such as (i) a misdirected write in which the data is written to an incorrect disk location, thus

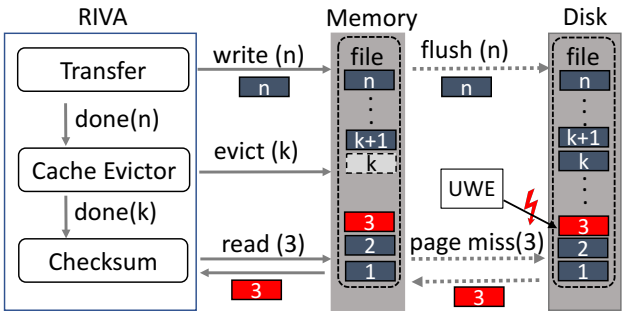


Fig. 4: System architecture of RIVA. When *Checksum* thread attempts to read UWE-exposed page-3, it will trigger a page miss since *Cache Evictor* has evicted it. Consequently, page-3 will be read from the disk and the UWE will be detected.

overwriting and corrupting data, (ii) write error in which only a portion of the data block is written successfully, and (iii) data corruption caused by components within the data path [13], [43].

Bairavasundaram et al. monitored 1.53 million disk drives over 41 months and observed more than 400,000 checksum mismatches [10]. They also found that nearline disks have an order of magnitude higher probability of developing checksum mismatches than enterprise-class disks. Yet, corrupt enterprise-class disks tend to develop more checksum mismatches. Although data scrubbing and RAID reconstruction can detect and possibly recover checksum mismatches, they take a long time to finish during which data becomes inaccessible. In another work, Bairavasundaram et al. monitored 1.5 million hard drives over 32 months and found that 8.5% of all disks developed at least one latent sector error during observation period [19]. Moreover, Krioukov et al. showed that even tough silent data corruption is detected, the system may not recover the block, causing data to be lost permanently [11].

4 System Design

RIVA enforces checksum calculations to read files from disk rather than page cache of memory to detect silent disk errors that might happen while transmitting data between memory and disk or reading/writing from/to disk. Hence, it clears page cache before reading files to compute their checksum. Figure 4 depicts RIVA receiver which consists of three threads. The *Transfer* thread receives file pages from network and flushes them to the disk, the *Cache Evictor* thread evicts file pages from the memory, and the *Checksum* thread reads the evicted pages back from the disk to calculate the checksum. The *Cache Evictor* uses `mmap` and `munmap` system calls to locate and evict file pages. When a file is recently transferred and written to the disk, its pages are kept in the page cache by operating system (OS) to optimize future accesses, however we found that this paves the way for UDEs to go undetected. Thus, *Cache Evictor* runs `mmap` to locate file pages in the virtual address space which is used to keep track of the memory mapping of file pages that are stored on the disk. If the mapping points to a valid memory address for a file page, it indicates that the file page is currently cached in the page cache, thus subsequent accesses will be served from the cache. Since RIVA aims to avoid cached data, it executes `munmap` system call to delete virtual address space mapping of file pages such that when the *Checksum* thread attempts to read the file, OS would not be able to locate valid mappings (i.e. page miss) and fetch it from disk. Moreover, `munmap` ensures coherence between page cache and Translation Lookaside Buffer (TLB) by flushing related entries in TLB to avoid using stale data in CPU caches (i.e., L1, L2, and L3) as well [44]. Moreover, RIVA's cache invalidation works in the file-level – it only invalidates only the pages of transferred file from cache –, thus it neither requires root privileges nor interferes with the operation of the rest of the system.

Although single execution of `munmap` is generally sufficient to clear the pages from cache, it is possible that OS brings some of the pages back to the cache immediately. Hence, *Cache Evictor* checks whether the file is completely evicted from cache using `mincore` syscall. If the file pages are not fully removed from the page cache, then the *Cache Evictor* will keep calling `munmap` until `mincore` confirms that none of the pages reside in the memory. Once the pages are successfully removed from page cache, they are passed to the *Checksum* thread to calculate checksum and send it to RIVA sender to verify the integrity of the received file. It is worth to note that cache eviction may not be necessary for large files since OS automatically evicts old file pages when page cache is full. However, since most transfers in scientific facilities are dominated by small files [22], clearing file pages from page cache is necessary to avoid UDEs. Besides, relying on OS to evict file pages for large files is not reliable choice since cache eviction policy can be manipulated to remove the recently inserted pages and cause the *Checksum* thread to locate some pages in page cache.

As an example, assume that page 3 is exposed to a UWE as shown in Figure 4. Instead of reading page 3 back again immediately for checksum calculation, RIVA first lets *Cache Evictor* to remove its memory mapping from virtual address space. This way, when *Checksum* thread attempts to read the page 3 to calculate checksum, it will trigger a page miss and the corrupted page will be fetched from the disk. As a result, this trigger a checksum mismatch between source and destination and the file will be transferred, allowing RIVA to detect and recover from the UWE. RIVA sender will operate in a similar way to locate and clear page cache after the file is read for the transfer such that checksum process can detect and avoid UREs that might happen during the transfer of a file. When checksum mismatch is identified, RIVA assumes that the sender copy of the file is the genuine one and retransfers the file from sender to receiver again. While this may cause redundant retransfers in cases where successful transfer is followed by miscalculated checksum due to undetected read errors, this is a trade-off to avoid undetected write errors.

A simple approach to implement integrity verification with RIVA would involve running *Transfer*, *Cache Evictor*, and *Checksum* threads sequentially for each file in dataset. However, it would significantly increase execution time of transfers especially for large files. For example, transferring a 100GB file first and then running cache eviction and checksum computation would lead to total execution time to be sum of all three operations. RIVA thus take advantage of multithreading by runs its threads concurrently on different

Specs	Storage	CPU	Memory (GB)	Bandwidth (Gbps)	RTT (ms)
HPCLab-WS	SATA HDD	(8) Intel Core i5-7600 @3.50GHz	16	1	0.2
Chameleon Cloud	SATA HDD	(12) Intel Xeon E5-2670 @2.30GHz	128	10	0.2
Pronghorn	GPFS	(16) Intel Xeon E5-2683 @2.10GHz	192	10	0.1
HPCLab-DTN	NVMe SSD	(16) Intel Xeon E5-2623 @2.60GHz	64	40	30
ESnet	RAID-0	(12) Intel Xeon E5-2643 @3.40GHZ	128	100	89

TABLE 1: System specification of test networks.

portions of files to lower the total execution time. In Figure 4, *Transfer* thread is transferring page n, *Cache Evictor* thread is evicting page k from memory, and *Checksum* thread is reading page 3 to calculate checksum. *Transfer* thread sends periodic signals to *Cache Evictor* to inform about received pages after writing to disk so that those pages could be removed from page cache. Similarly, *Cache Evictor* thread sends messages to *Checksum* to notify it about evicted pages.

Most operating systems define page size as 4 KB, so running thread communication for each file page could be prohibitively expensive for large files. For example, it would require 262,144 messages to be exchanged between *Transfer* and *Cache Evictor* threads for a 1 GB file. Thus, RIVA threads send messages for a collection of pages (aka block) to reduce communication overhead. Block size is configurable and is by default set to 256MB, so each block contains 65,536 file pages for a page size of 4KB. As an example, when the transfer of a 1 GB file is requested with block size 256 MB, the *Transfer* thread will start by moving the first block (0-256MB) of the file. Once the first block is received and written to disk, it will move to the transfer of the second block (256M-512M). At the same time, *Cache Evictor* will remove the first block from memory and let *Checksum* thread to start reading the first block to calculate its hash. So, while the *Transfer* thread is sending the second block, *Cache Evictor* and *Checksum* can start processing the first block. As a result, all three operations (i.e., file transfer, cache eviction, and checksum computation) can execute in parallel, leading to shorter execution time compared to sequential execution of them. It is also worth to note that these operations can take different execution times, so overall execution time will be determined by the speed of slowest operation. Another benefit of processing file in blocks is the cost of recovery in the case of checksum mismatch as RIVA only needs to resend the failed blocks.

5 Evaluations

We run the experiments using two types of dataset; uniform and mixed. Uniform datasets contain of one or more files in same size and mixed datasets consist of files with various sizes. We evaluated RIVA in four different networks: HPCLab, ESnet, Pronghorn, and Chameleon Cloud whose specifications are given in Table 1. In HPCLab, we have two sets of servers; workstations (HPClab-WS) and data transfer nodes (HPCLab-DTN). The workstations are connected with a 1G link whereas data transfer nodes are connected with a 40G link. Although data transfer nodes are located in the same local area network, we injected artificial delay between them to emulate wide-area network condition using traffic controller (`tc`) of Linux. Pronghorn is a campus cluster and its nodes are connected with 10G links. Chameleon Cloud is an OpenStack based cloud service and its nodes are connected with 10G links. ESnet testbed offers a dedicated 100G bandwidth and 89 ms delay between end points. Finally, we used one Pronghorn server as a sender and one Chameleon Cloud server as a receiver to run Pronghorn-Chameleon experiments. We repeated the experiments at least five times and present average results unless stated otherwise.

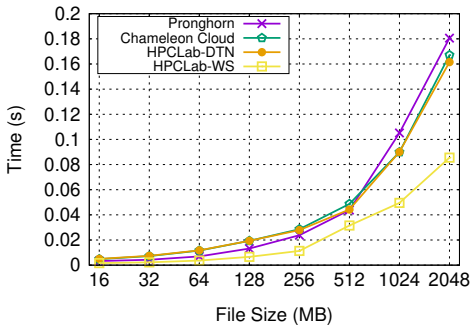


Fig. 5: Cache eviction times for different file sizes. It takes less than 0.03 seconds for 256MB files.

5.1 Execution Time Analysis

RIVA executes transfer, cache eviction, and checksum computation tasks to transfer a file with integrity verification. The transfer task involves disk read at source, network transfer, and disk write at destination. As these three operations are executed in parallel, transfer time approximately equals to runtime of the the slowest operation. On the other hand, checksum computation runs disk read and hash calculation operations sequentially (i.e, read a set of file pages and digest them for checksum before moving to next set), so its execution time becomes sum of runtime of these two routines. Since both source and destination end points calculate checksum, the overall checksum speed is dictated by the slower end point. Finally, both source and destination end points performs cache eviction for files before the checksum computation.

To shed a light on the total execution time of RIVA, we investigate the runtime of transfer, checksum computation and cache eviction procedures. Figure 5 presents the cost of cache eviction for different file sizes. It is clear that it is a lightweight procedure and takes less than 0.2s for a 2 GB file. Since RIVA splits files into blocks with default size of 256 MB, cache eviction would take less then 0.03s for a file block. On the other hand, the transfer time of a 256 MB file block would generally take between 0.5s-2s depending on I/O and network speeds (0.5s with HPCLab-DTN, and 2s in HPCLab-WS). Checksum computation operates at a rate of 360 MB/s with Intel Xeon E5 @2.60GHz CPU and 450MB/s with Intel i5 @3.50GHz CPU and takes between 0.5s-0.7s.

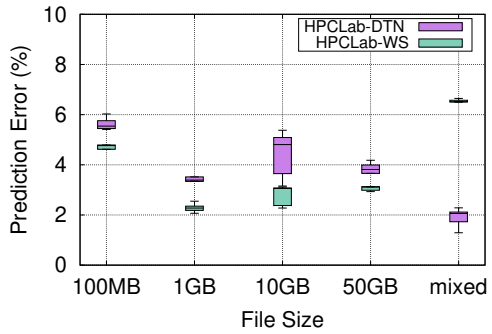


Fig. 6: Error rate of RIVA execution time estimation in two networks. Bottleneck operation is file transfer in HPCLab-WS and checksum computation in HPCLab-DTN.

As a result, transfer and checksum calculation speeds are in the order of a magnitude slower than cache eviction speed. When combined with RIVA's concurrent execution of transfer, cache eviction, and checksum calculation, this would result in either transfer or checksum computation to be the bottleneck for RIVA. Thus, we can formulate the runtime of RIVA as given in Equation 1 where t_t, t_c, and t_e are transfer, checksum computation and cache eviction times for a single block and n_{blocks} is the number of file blocks which is calculated by dividing file size to block size.

$$t_{RIVA} = \begin{cases} t_t \times n_{blocks} + t_c + t_e, & \text{if } t_t \geq t_c \\ t_c \times n_{blocks} + t_t + t_e, & \text{otherwise} \end{cases} \quad (1)$$

$$\epsilon = 100 * \frac{|t_{RIVA} - t_{actual}|}{t_{actual}} \quad (2)$$

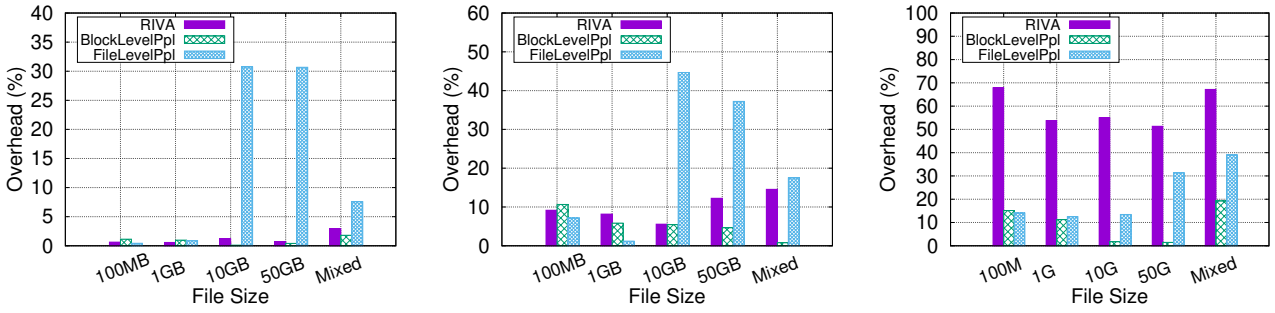
If the transfer speed is slower than the checksum speed, the execution time will be determined by transfer time of all blocks ($t_t \times n_{blocks}$ plus cache eviction and checksum calculation times for the last block. Similarly, if the checksum computation is slower, then total time can be calculated by the transfer and cache eviction time for the first block plus checksum computation time for all the blocks. Figure 6 evaluates the accuracy of the model in two networks; in one transfer is the bottleneck (i.e., HPCLab-WS) and in another checksum computation is the bottleneck. Error is calculated as percentage of difference between predicted, t_{RIVA} and actual measurement t_{actual} time as shown in Equation 2. The results show that the model is able to predict the execution time with less than 7% error rate in both networks for all data types. We observed that concurrent execution of checksum and transfer operations leads up-to 10% degradation of I/O speed of both operations both in HPCLab-WS and HPCLab-DTN networks. This in turn increases execution time a bit compared to isolated execution times of these operations as used in Equation 1.

Compared to "transfer only" (i.e., without integrity verification) scenario, RIVA's overhead depends on the bottleneck operation. If the transfer is the bottleneck, then the overhead will be the slowdown ratio in transfer speed due to concurrent checksum computation, nearly 10% in most networks. On the other hand, if checksum computation is the bottleneck, then RIVA will extend the runtime by the ratio of checksum and transfer times for a single block, t_c/t_t. Yet, we showed in our previous work that one can take advantage of multicore architectures to run multiple checksum instances simultaneously to mitigate it from being the bottleneck [38]. Moreover, because RIVA does not introduce the integrity verification, rather aims to increase its robustness, it would be fair to evaluate its performance against other integrity verification approaches. Thus, in the rest of the paper we compare RIVA against existing integrity verification algorithms; file-level pipelining (FileLevelPpl), block-level pipelining (BlockLevelPpl), and FIVER.

FileLevelPpl overlaps the transfer of a file with the checksum calculation of another file. BlockLevelPpl splits large files into blocks similar to RIVA and overlaps the transfer of a block with the checksum of another block of the same file. Finally, FIVER overlaps transfer of a file with the checksum of the same file to share I/O between the two. Experimental results indicate that FIVER always yields the shortest execution time [21]. Hence, we calculate the performance of different approaches relative to FIVER and define overhead as shown in Equation 3. t_{FIVER} and $t_{algorithm}$ refer to the times it takes to transfer a dataset with integrity verification using FIVER and a different algorithm. For example, if a transfer takes 120 seconds with FIVER, and 130 seconds with RIVA, then the overhead becomes 8.3% $(100 * \frac{130-120}{120})$. The main difference between RIVA and FIVER is that the former enforces disk read for checksum computation whereas the latter uses cached pages for the same task. Since cache eviction system call has negligible impact on the runtime of RIVA, the potential cost of RIVA can be explained by the I/O slowdown triggered by concurrent disk accesses of transfer and checksum operations. Theoretically, the slowdown can be up-to 100% (checksum I/O can be blocked completely while the transfer is running) but in practice we observed that it is around 10% in most networks.

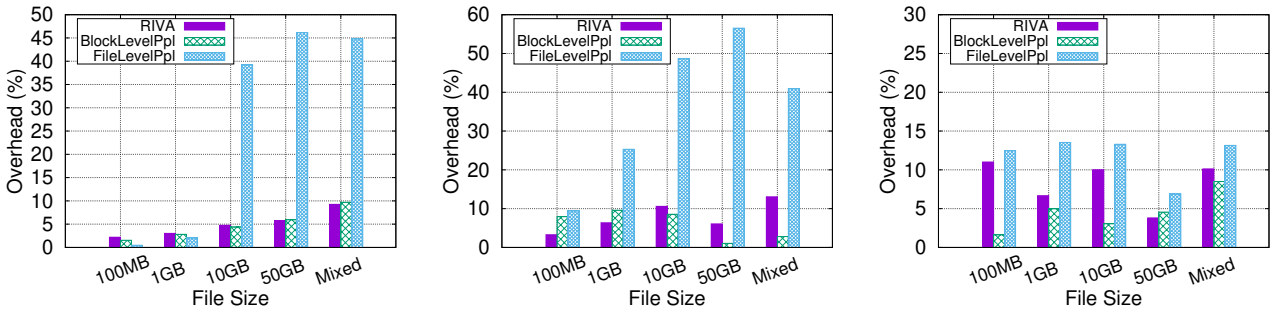
$$Overhead = 100 * \frac{t_{algorithm} - t_{FIVER}}{t_{FIVER}} \quad (3)$$

We conducted extensive experiments in six different networks which are grouped as local-area network (Figure 7) and wide-area network (Figure 8) results. The overhead of RIVA is always less than 5% in HPCLab-WS transfers (Figure 7(a)) which can be attributed to slow transfer speed. Since RIVA pipelines cache eviction and checksum computation operations with file transfers, it incurs negligible overhead when the transfer speed is the bottleneck. The overhead of FileLevelPpl reaches up-to 30% for 10GB and 50GB files since there is only one file in those datasets, causing FileLevelPpl to perform transfer and checksum operations sequentially. On the other hand, the overhead of RIVA increase to 15% in Pronghorn as its disk read speed is worse than disk write and transfer speeds. Finally, the overhead of RIVA exceeds that of FileLevelPpl in Chameleon Cloud as given in Figure 7(c). Chameleon Cloud nodes are customized for high-performance computing with large memory size and multi-core CPUs, and exhibit poor disk read/write performance. However, the OS is able to cache file writes in the memory and flush them to the disk at a slower rate, effectively increasing disk write speed as long as file size is smaller than free memory space. On the other hand, slow disk read speed cannot be improved in similar way as OS kernels cannot predict which file to cache in advance. Disk read speed is even further



(a) HPCLab-WS (b) Pronghorn (c) Chameleon Cloud

Fig. 7: Performance comparison of algorithms in LAN experiments. While RIVA is able to keep its overhead below 17% in HPCLab and Pronghorn networks, slow disk speed and large memory size cause its performance to deteriorate significantly in Chameleon Cloud.



(a) HPCLab-DTN (b) ESnet (c) Pronghorn-Chameleon Cloud

Fig. 8: RIVA is able to keep its overhead below 12% in WAN experiments.

degraded when it is overlapped with disk write operation as RIVA does. However, BlockLevelPpl and FileLevelPpl do not experience slow disk speed as they mostly read files from page cache for checksum computation because OS kernel keeps them in memory after they are transferred. As a result, FIVER, BlockLevelPpl, and FileLevelPpl perform transfer and checksum computations faster than disk speeds whereas RIVA enforces disk read to be able to detect UDEs, imposing significant performance penalty. On the other hand, we noticed drastic performance degradation for FileLevelPpl (nearly 70% overhead) when a file whose size it larger than memory size (128 GB) as it cannot take advantage of write caching anymore.

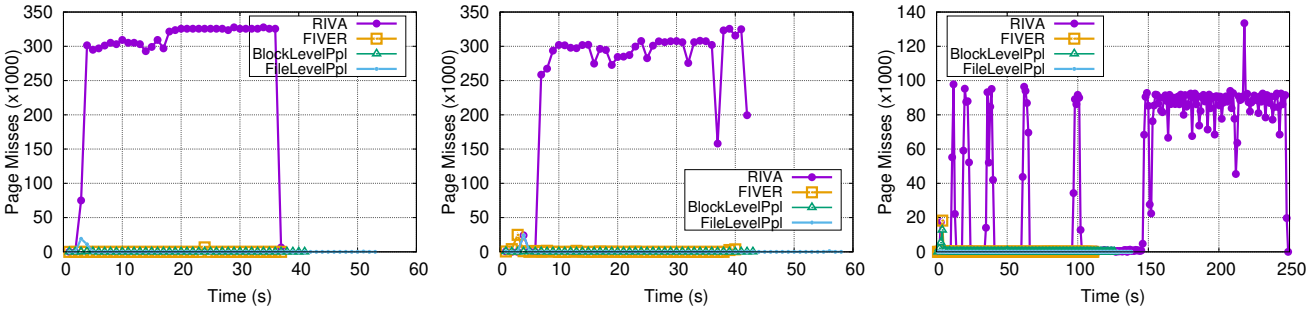
RIVA keeps its overhead less than 12% in all WAN experiments as shown in Figure 8. Checksum computation is the bottleneck operation in HPCLAb-DTN and ESnet networks, thus FileLevelPpl suffers significantly for the single file transfers (i.e., 10GB and 50GB) and takes up-to 50% more time than FIVER. FileLevelPpl also performs 40% worse than FIVER for mixed dataset since it fails to benefit from overlapping of transfer and checksum processes when dataset contains both small and large files. BlockLevelPpl, however, achieves the lowest overhead in almost all cases since its pipelining method overcomes the overlapping issue that FileLevelPpl experiences.

5.2 Fault Resilience

5.2.1 *Cache Hit Analysis*

We investigate the disk access behavior of different solutions by measuring page miss values during checksum computation phase. Page miss value defines the number of file pages that the OS fetches from disk since they are not located in the page cache of the server. Figure 9 shows the receiver-side page miss behavior for a 10 GB file transfer in three testbeds. The transfer speed in HPCLab-DTN and ESnet networks is higher than that of checksum computation, letting *Checksum* thread to run uninterrupted. As a result, RIVA sustains consistent disk I/O rate (around 2.4 Gbps) as shown in Figure 9(a) and 9(b). On the other hand, RIVA returns different behaviour in Chameleon Cloud where disk read speed is significantly worse than transfer and disk write speeds for files that are smaller than 100GB. Moreover, when the disk read and write operations overlap, the read performance degrades even further. Consequently, RIVA has short periods of disk reads until disk write completely finishes. The transfer of the file completes at around 150s after which disk read performance recovers and RIVA's *Checksum* thread obtains high and consistent read I/O rate. On the other hand, all the other approaches reads the file completely from page cache in all networks, leading to negligible page misses throughout the checksum calculation phase.

We further calculated total page misses for RIVA and verified that it is close to 10GB. While page miss behavior can be used to infer I/O access pattern of an algorithm, it alone cannot be used to guarantee cache avoidance. This is because the measured pages miss value may include misses caused by other operations such as network transfer and disk write. Thus, we introduce *extreme-injection* test in the next section to confidently claim that none of the pages of a file is read from the page cache of main memory.



(a) HPCLab-DTN (b) ESnet (c) Chameleon Cloud

Fig. 9: RIVA causes high page misses as it evicts file pages from memory and reads directly from disk to capture UWEs.

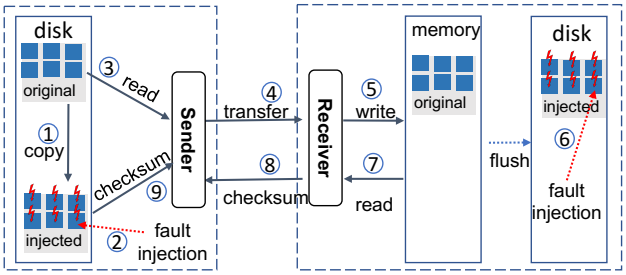


Fig. 10: Extreme-injection test involves injecting UWEs to all pages of files to determine if transfer applications read any pages from page cache during checksum calculation.

5.2.2 Error Injection

Given that the rate of UWE occurrence is low, testing RIVA in the real world would be costly, probably requiring a prohibitively large number of disks to observe within a reasonable period of time. Hence, we reproduced UWEs by injecting faults to files pages on hard disks. We first identify the disk sectors that file pages reside, then update the content of pages by writing directly to disk partition which is not reflected to file pages in page cache. For example, when we inject fault to the page $k+1$ in Figure 4, the cached copy will not be updated, creating a similar impact of UWEs as shown in Figure 2(b).

In *extreme-injection* test, all pages of a file are exposed to a UWE to verify that RIVA does not read even a single file page from page cache during the checksum computation. We first create a copy of the file on sender side and flip the first bit of all file pages (step 1 in in Figure 10). This copy is called *injected* and used to validate the output of the receiver side checksum. The sender then transfers the *original* file to the receiver which is written to disk (step 5). As file is being written to disk, we flip the first bit of all file pages that are flushed to disk just before checksum computation starts. We flip same bit of pages at the sender and the receiver to be able to compare injected copies. Upon completion of fault injection to all file pages, we let checksum thread to read the file and compute its checksum (step 7). The computed checksum value is then sent to the sender to be compared against the *injected* copy (step 8 and 9).

If checksum values match, we can then confidently claim that the checksum thread of the receiver must have read all file pages from the disk. If the receiver reads even one page of the file from page cache, then its checksum value will be different than that of the sender's *injected* version. We conducted the *extreme-injection* test in HPCLab-WS using several files that are smaller and larger than the memory size, 16GB. Table 2 presents the results of integrity verification algorithms in terms of being able to detect all injected errors. FIVER and BlockLevelPpl failed to pass the test for all file sizes as their checksum threads always read the files from page cache. FileLevelPpl is able to catch the faults only for 50GB file which is three times larger than memory size. Surprisingly, FileLevelPpl fails to pass the test for the 20GB file despite yielding high page misses, implying that at least one file page is read from page cache on the receiver side. Finally, RIVA is able to capture all fault injections regardless of file size by means of invalidating cache copies of file pages before running the checksum computation.

	1MB	100MB	1GB	10GB	20GB	50GB
FIVER	–	–	-	–	–	–
BlockLevelPpl	–	–	–	–	–	–
FileLevelPpl	–	–	–	–	–	✓
RIVA	✓	✓	✓	✓	✓	✓

TABLE 2: The results of extreme-injection experiment in HPCLab-WS network with 16GB main memory. RIVA is able to detect all injected faults whereas FileLevelPpl can only detect them for 50GB files.

5.2.3 Statistical Analysis On The Likelihood of UDEs

In overall, there are four I/O operations that take place during the transfer of a file with integrity verification. They are (i) file read on the source server to transfer file content (*Read(S)* in Table 3), (ii) file write on the destination server to save the transferred data to disk (*Write(D)*), (iii) second file read on the source to compute the checksum of the transferred file (*Read(S)*), and (iv) file read on the destination to compute the checksum of the received file (*Read(D)*). Although RIVA is resilient against UDEs that affect one of these I/O operations, it may fail if multiple I/O operations are exposed to similar UDEs. For instance, if a UWE on receiver is followed by a URE again on receiver (while reading the file from disk to compute the checksum) in a way that it is exactly reverse of the UWE, then RIVA will miss the UWE and corrupted data will be stored in the disk.

Table 3 lists few examples of single and multi-bit errors that affect one or more I/O operations during the transfer

Row ID	# of I/Os Impacted	Transfer		Checksum		Detected by RIVA?
		Read (S)	Write (D)	Read (S)	Read (D)	
1	1	01	01	00	01	Yes
2	1	00	11	00	11	Yes
3	1	00	00	01	00	Yes
4	2	01	01	00	01	Yes
5	2	01	01	01	01	No
6	2	00	11	00	00	No
7	3	10	00	01	00	Yes
8	3	01	01	10	10	No
9	3	00	10	11	11	No
10	4	01	10	11	01	Yes
11	4	11	10	11	11	No
12	4	01	00	10	10	No

TABLE 3: Examples of UDEs that affect disk read/write I/O during the transfer of a two bit data with original value of *"00"* at the source. Out of four I/O operations (two for transfer and two for integrity verification), RIVA is able to capture all errors which affect only one operation. However, RIVA may miss errors if they appear in multiple operations in a way that source (S) and destination (D) checksum values match.

of a two-bit data which is originally stored on the disk of source server as *"00"*. Note that when any one I/O operation (i.e. the number of I/Os impacted is 1) is exposed to single or multi-bit errors, RIVA can guarantee its detection it since it is impossible for single UDE to result in checksum match (RIVA will only miss UDEs if checksum comparison returns positive result) while the stored data is different than the original one. On the other hand, when multiple I/O operations are affected by UDEs, then RIVA may miss them if they happen in a way that source and destination checksum values match. As an example, assume that a URE happened at the source server during file transfer and caused data to be read as *01* (row 5 on Table 3). The receiver then will receive the corrupted data and saves it to disk as it receives, *01*. Once the transfer operation is complete, both source and destination servers read the data from disk to compute checksum. Assume also that the second UDE took place when reading the file on source server and causes data to be read as *01*, again. Then, since destination server saved the data as *01* as well and checksum I/O on destination is not exposed to a UDE, it will cause checksum values of source and destination servers to match, preventing the detection of silent data corruption. In the rest of this section, we calculate the probability for RIVA to miss UDEs caused by uncorrectable single or multiple bits flip.

Undetected bit error rate (UBER) is defined as the rate of errors that have escaped from *ECC* [35]. The probability of an undetected bit flip is equal to UBER assuming each bit error in a data block is independent and the number of bit errors follows a binomial distribution. Equation 4 shows the probability of i bit flips in a b-bit block with uncorrectable bit error rate E assuming that there exist at most one flip for each bit [35], [45]. Then, Equation 5 defines the probability of undetected bit error for a block, which is the sum of the probabilities of all possible combinations of bit flips (from total of 1 bit flip to b bit flips).

$$P_c(block, i) = \binom{b}{i} \times E^i(1-E)^{b-i} \tag{4}$$

$$\begin{aligned} P_c(block) &= \sum_{i=1}^{b} \binom{b}{i} \times E^i(1-E)^{b-i} \\ &= \sum_{i=0}^{b} \binom{b}{i} \times E^i(1-E)^{b-i} - (1-E)^b \\ &= (E+(1-E))^b - (1-E)^b \text{ (Binomial Theorem [1])} \\ &= 1-(1-E)^b \\ &\approx bE \end{aligned} \tag{5}$$

RIVA will miss UDEs if they appear in more than one I/O operation as described above. Hence, two or more I/O operations must be exposed to UDEs out of total four disk read/write operations. Among those, we first explore the possibility UDEs that leads to data corruption in two I/O operations. Equation 4 defines the probability of i uncorrectable bit flips in an operation with block size of b. Since UDEs in two I/O operations must be similar (see rows 4 and 5 in Table3) or complementary (row 6), for RIVA to miss them, the probability becomes the multiplication of i bit UDE combinations and the probability of a UDE to happen twice, as shown in Equation 6. Then, the Equation 7 shows the cumulative probability of all bit errors to appear in two I/O operations.

$$P(block, i) = \binom{b}{i} \times (E^i \times (1-E)^{b-i})^2 \tag{6}$$

$$P_{RIVA}(block) = \sum_{i=1}^{b} \binom{b}{i} \times (E^i \times (1-E)^{b-i})^2 \tag{7}$$

$$= \sum_{i=0}^{b} \binom{b}{i} \times E^i(1-E)^{b-i} - (1-E)^b$$

$$= -(1-E)^{2b} + \sum_{i=0}^{b} \binom{b}{i} E^{2i}(1-E)^{2(b-i)}$$

$$= -(1-E)^{2b} + (E^2+(1-E)^2)^b \text{ (Binomial Theorem)}$$

$$= (E^2+(1-E)^2)^b - (1-E)^{2b}$$

$$= (1+2E^2-2E)^b - (1-E)^{2b}$$

expanding both terms

$$= (\binom{b}{0}1^b(2E^2-2E)^0 + \binom{b}{1}1^{b-1}(2E^2-2E)^1$$

$$+ \cdots + \binom{b}{b}1^0(2E^2-2E)^b) - (\binom{2b}{0}1^{2b}(-E)^0$$

$$+ \binom{2b}{1}1^{2b-1}(-E)^1 + \cdots + \binom{2b}{2b}1^0(-E)^{2b})$$

$E^b, E^{b-1}, \cdots, E^4, E^3 << E^2 \ \forall E < 10^{-10}$, thus ignoring terms with power of E is greater than 2

$$\approx (\binom{b}{0}1^b(2E^2-2E)^0 + \binom{b}{1}1^{b-1}(2E^2-2E)^1)$$

$$- (\binom{2b}{0}1^{2b}(-E)^0 + \binom{2b}{1}1^{2b-1}(-E)^1)$$

$$= (1+b(2E^2-2E)) - (1-2bE)$$

$$= 1+2bE^2-2bE-1+2bE$$

$$= 2bE^2$$

1. Binomial Theorem: $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k}b^k$

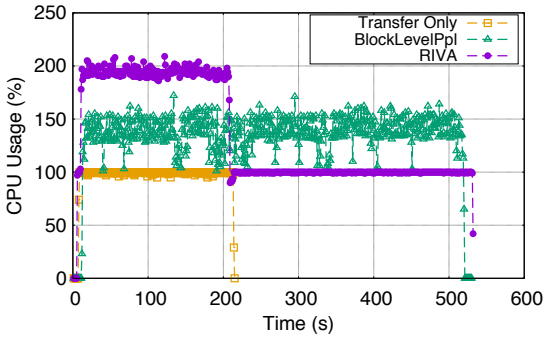


Fig. 11: CPU utilization analysis for mixed dataset transfer in HPCLab-DTN network. While RIVA has different CPU usage pattern than BlockLevelPpl, average utilization is same for both algorithms.

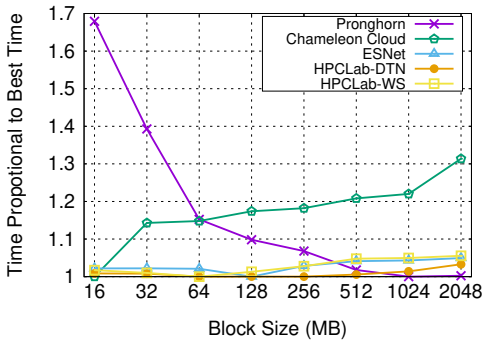


Fig. 12: While block size has negligible impact in most networks, it affects execution time by around 70% in Pronghorn and up-to 30% in Chameleon Cloud.

Since there are six possible combinations for two of four I/O operations to be affected by UDEs, the maximum probability of RIVA to miss UDEs becomes $6 \times 2bE^2 = 12bE^2$. Please note that this is an upper bound as it counts all possible UDEs that happen twice whereas many of such error will be captured as shown in row 4 in Table 3. In addition, the likelihood of three or four I/O operations to be exposed to UDEs in a way that it will mislead RIVA is much lower than that of two I/Os to be impacted since $E^2 >> E^3 >> E^4$ when E is lower than 10^{-10}. Thus, we can conclude that while the probability of UDEs to go undetected is bE with existing integrity verification algorithms, RIVA reduces it to bE^2. To better evaluate the impact of such decrease in probability on real-world scenarios, let's take $E = 10^{-10}$ and $b = 256$ MB which leads error rate ($P(error) = bE$) to be 0.2 for traditional integrity verification algorithms whereas it is reduced to 10^{-11} with RIVA. Even for more conservative E value of 10^{-15}, the probability of error is reduced from 10^{-6} to 10^{-21}. As a result, although RIVA cannot completely rule out undetected disk errors to happen in file transfers, it significantly reduces their likelihood compared to existing solutions.

5.3 Processing Overhead

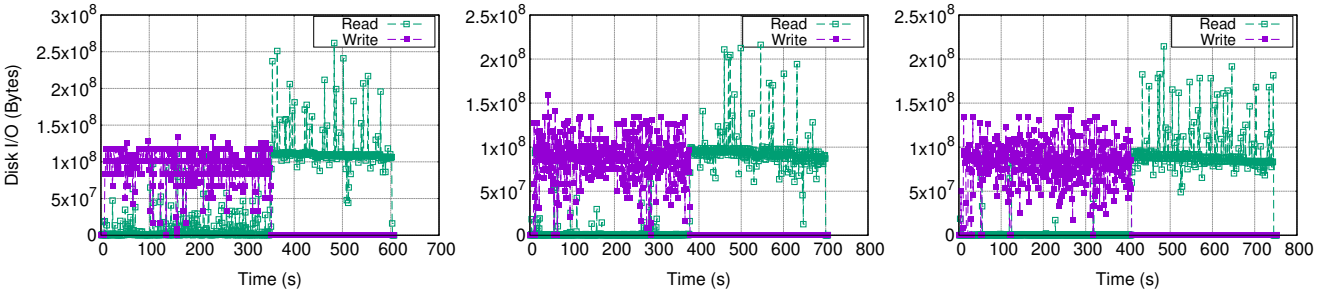
In addition to execution time, integrity verification also incurs processing overhead as checksum computation is a CPU intensive operation. Figure 11 illustrates CPU usage for transfer-only, BlockLevelPpl, and RIVA for the transfer of a mixed dataset in HPCLab-DTN network. Since checksum computation is the bottleneck, transfer-only method finishes earlier than the others. Moreover, since it only create one transfer thread, its utilization is capped at 100%. BlockLevelPpl concurrently executes checksum and transfer tasks on separate threads, so its CPU usage reaches to 150%. Checksum computation contributes to 100% of total usage and remaining comes from transfer thread. Since BlockLevelPpl enforces transfer and checksum threads to synchronize after each block processing, transfer thread completes its operation and sits idle while checksum is still working on, causing CPU usage of transfer thread to be between 20% and 50%.

On the other hand, despite running transfer and checksum computation concurrently, RIVA does not impose thread synchronization between threads other than consumer-producer relationship. For instance, the *Cache Evictor* can only evict file blocks that *Transfer* thread processed, so it cannot run faster than *Transfer* thread. On the other hand, if file transfer is faster than checksum computation, then the *Transfer* thread does not have to wait for cache eviction and checksum computation threads to complete. Consequently, RIVA's CPU usage reaches to 200% while both checksum and transfer threads execute, which falls to 100% upon the completion of the transfer at around 220s. In addition, the *Cache Evictor* of RIVA has 5% peak CPU utilization but only runs for few seconds, thus has negligible impact on overall CPU utilization considering the total runtime. Thus, RIVA and BlockLevelPpl have similar (133%) CPU utilization on average.

5.4 Impact of Block Size

As discussed in Section 4, RIVA executes *Transfer*, *Cache Evictor*, and *Checksum* threads concurrently for different portion of the same file (aka block) to keep its overhead at minimum. By default, we define the block size to be 256 MB, however, this might not be an optimal value in all testbeds due to differences in storage subsystems. Thus, we evaluated the impact of block size on execution time by transferring a 30GB file with integrity verification as shown in Figure 12. It is clear that block size has negligible impact on the performance for HPCLab-DTN, HPCLab-WS, and ESNet networks. On the other hand, it leads to 30% and 70% change in execution time for Chameleon Cloud and Pronghorn testbeds, respectively. Interestingly, while small block size yields the best performance in Chameleon Cloud, it returns the worst performance in Pronghorn. As a result, the default value of 256MB yields up-to 20% and 8% lower performance compared to the best performing block size for Chameleon Cloud and Pronghorn networks.

We further analyzed receiver-side disk read and write I/O throughput for various block size values for Chameleon Cloud network in Figure 13. Receiver side write I/O is a result of writing the file to disk after receiving it from the network. Read I/O, on the other hand, is part of integrity verification process for which the file is read back from the disk to compute its checksum. Overlapping read and write I/O operations in Chameleon Cloud slows down both operations which is further exacerbated by increased block



(a) 16MB (b) 256MB (c) 2048MB

Fig. 13: Disk I/O throughput for different block size values for the transfer of a 30GB file in Chameleon Cloud network. Small block size yields higher throughput when file read and write operations are overlapped.

size. For example, while file write operation finishes at around 350s for 16MB block size, it takes more than 400s for block size of 2048MB. Moreover, when block size is 16MB, considerable amount of disk read activity can still takes place, whereas it becomes less frequent as the block size increases and almost disappears for 2048MB case. Consequently, remaining read operation (after write operation is complete,) takes longer for large block size values. For instance, while it takes around 250s (from 350s to 600s) when block size is 16MB, it increases to 340s (from 410s to 750s) when block size is set to 2048MB.

In summary, while block size can have significant impact on execution time, no single predefined value yields the close-to-optimal performance in all networks. Thus, a careful tuning is necessary to efficiently utilize underlying storage subsystem and shorten execution time, however we leave it as a future work.

6 Conclusion

End-to-end integrity verification is vital for many scientific applications which cannot tolerate silent data corruptions. However, its current implementations for file transfers fail to capture undetected disk write errors, creating possibility of permanent data loss. In this paper, we propose RIVA to improve robustness of the end-to-end integrity verification by enforcing checksum calculation to read files directly from disk. RIVA invalidates the cached copies of file pages such that checksum calculation can read disk copies and detect undetected disk errors. Our extensive experiments show that RIVA offers a robust solution to capture and recover from all undetected disk write errors. In exchange, RIVA increases transfer execution time, however it can keep its overhead less than 15% in most cases by concurrently executing transfer, cache eviction, and checksum operations.

As a future work, we aim to develop dynamic optimization methods to tune parameters of RIVA to maximize system utilization and minimize overhead. For example, RIVA defines block size to logically split large files into partitions, however we observed that optimal setting for the value of block size can improve execution time by up-to 20% in some networks compared to the default value. We also plan to extend experiments to include more silent data corruption scenarios beyond undetected disk errors to demonstrate that RIVA's error detection capability is not limited to storage-related errors. Moreover, while RIVA strengthens end-to-end integrity verification against UWEs and UREs that happen while reading and writing to disk, files can still be corrupted while sitting on disk due to other errors such as sector errors and misdirected writes. Hence, we will investigate solutions to store the checksum of files in global and immutable databases (e.g., blockchain) such that users can periodically verify the authenticity of files by comparing their checksum against the ones in the database. Finally, we will also explore options to forecast UDE occurrences using low-level disk statistics (i.e. SMART features) such that integrity verification can be enforced only for error-prone disks to alleviate the I/O and CPU overhead of integrity verification.

Acknowledgments

This project is in part sponsored by the National Science Foundation (NSF) under award number OAC-1850353. Some of the results presented in this paper were obtained using the Chameleon testbed supported by the NSF.

References

[1] R. J. T. Klein, R. J. Nicholls, and F. Thomalla, "Resilience to natural hazards: How useful is this concept?" *Global Environmental Change Part B: Environmental Hazards*, vol. 5, no. 1-2, pp. 35 – 45, 2003.

[2] J. Kiehl, J. J. Hack, G. B. Bonan, B. A. Boville, D. L. Williamson, and P. J. Rasch, "The national center for atmospheric research community climate model: CCM3," *Journal of Climate*, vol. 11:6, pp. 1131–1149, 1998.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," *Journal of Molecular Biology*, vol. 3, no. 215, pp. 403–410, October 1990.

[4] CMS, "The US Compact Muon Solenoid Project," http://uscms.fnal.gov/.

[5] "A Toroidal LHC ApparatuS Project (ATLAS)," http://atlas.web.cern.ch/.

[6] "Dark Energy Survey," https://www.darkenergysurvey.org/.

[7] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.

[8] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," in *ACM SIGCOMM computer communication review*, vol. 30, no. 4. ACM, 2000, pp. 309–319.

[9] J. L. Hafner, V. Deenadhayalan, W. Belluomini, and K. Rao, "Undetected disk errors in raid arrays," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 413–425, 2008.

[10] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, p. 8, 2008.

[11] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Parity lost and parity regained." in *FAST*, vol. 2008, 2008, p. 127.

[12] M. Li and P. P. C. Lee, "Toward i/o-efficient protection against silent data corruptions in raid arrays," *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, 2014.

[13] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Transactions on dependable and secure computing*, vol. 1, no. 1, pp. 87–96, 2004.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google file system*. ACM, 2003, vol. 37, no. 5.

[15] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *Proceedings of the 14th ACM conference on Computer and communications security*. Acm, 2007, pp. 598–609.

[16] M. Vigil, J. Buchmann, D. Cabarcas, C. Weinert, and A. Wiesmaier, "Integrity, authenticity, non-repudiation, and proof of existence for long-term archiving: a survey," *Computers & Security*, vol. 50, pp. 16–32, 2015.

[17] M. U. Arshad, A. Kundu, E. Bertino, A. Ghafoor, and C. Kundu, "Efficient and scalable integrity verification of data and query results for graph databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 5, pp. 866–879, 2018.

[18] E. W. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. Rao, and P. Zhou, "Evaluating the impact of undetected disk errors in raid systems," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 83–92.

[19] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1. ACM, 2007, pp. 289–300.

[20] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, and M. Papka, "Towards optimizing large-scale data transfers with end-to-end integrity verification," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 3002–3007.

[21] E. Arslan and A. Alhussen, "A low-overhead integrity verification for big data transfers," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 4227–4236.

[22] Z. Liu, R. Kettimuthu, I. Foster, and N. Rao, "Cross-geography scientific data transfer trends and user behavior patterns," in *27th ACM Symposium on High-Performance Parallel and Distributed Computing, HPDC*, vol. 18, 2018, p. 12.

[23] T. Kosar, E. Arslan, B. Ross, and B. Zhang, "Storkcloud: Data transfer scheduling and optimization as a service," in *Proceedings of the 4th ACM workshop on Scientific cloud computing*. ACM, 2013, pp. 29–36.

[24] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, "Software as a service for data scientists," *Communications of the ACM*, vol. 55:2, pp. 81–88, 2012.

[25] E. Arslan and T. Kosar, "High Speed Transfer Optimization Based on Historical Analysis and Real-Time Tuning," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[26] E. Yildirim, E. Arslan, J. Kim, and T. Kosar, "Application-level optimization of big data transfers through pipelining, parallelism and concurrency," *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 63–75, 2015.

[27] D. Yun, C. Q. Wu, N. S. Rao, and R. Kettimuthu, "Advising big data transfer over dedicated connections based on profiling optimization," *IEEE/ACM Transactions on Networking*, 2019.

[28] I. Alan, E. Arslan, and T. Kosar, "Energy-aware data transfer algorithms," in *Proceedings of SC'15*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 44:1–44:12.

[29] N. S. Rao, Q. Liu, S. Sen, G. Hinkel, N. Imam, I. Foster, R. Kettimuthu, B. W. Settlemyer, C. Q. Wu, and D. Yun, "Experimental analysis of file transfer rates over wide-area dedicated connections," in *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*. IEEE, 2016, pp. 198–205.

[30] E. Arslan, K. Guner, and T. Kosar, "HARP: Predictive Transfer Optimization Based on Historical Analysis and Real-Time Probing," in *Proceedings of SC'16*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 25:1–25:12.

[31] Y. Zhu, H. Hu, G.-J. Ahn, and M. Yu, "Cooperative provable data possession for integrity verification in multicloud storage," *IEEE transactions on parallel and distributed systems*, vol. 23, no. 12, pp. 2231–2244, 2012.

[32] C. Liu, C. Yang, X. Zhang, and J. Chen, "External integrity verification for outsourced big data in cloud and IoT: A big picture," *Future generation computer systems*, vol. 49, pp. 58–67, 2015.

[33] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. Rosenthal, and M. Baker, "The LOCKSS peer-to-peer digital preservation system," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 1, pp. 2–50, 2005.

[34] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. K. Mckusick, "Ffsck: The fast file-system checker," *ACM Transactions on Storage (TOS)*, vol. 10, no. 1, p. 2, 2014.

[35] Y. Zhang, D. S. Myers, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Zettabyte reliability with flexible end-to-end data integrity," in *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*. IEEE, 2013, pp. 1–14.

[36] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: A ZFS case study." in *FAST*, 2010, pp. 29–42.

[37] R. Hasan, R. Sion, and M. Winslett, "The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance." in *FAST*, vol. 9, 2009, pp. 1–14.

[38] B. Charyyev, A. Alhussen, H. Sapkota, E. Pouyoul, M. H. Gunes, and E. Arslan, "Towards securing data transfers against silent data corruption," in *IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing, IEEE/ACM*, 2019.

[39] "Globus," https://www.globus.org/.

[40] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population." in *FAST*, vol. 7, no. 1, 2007, pp. 17–23.

[41] S. Shah and J. G. Elerath, "Reliability analysis of disk drive failure mechanisms," in *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*. IEEE, 2005, pp. 226–231.

[42] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you?" in *FAST*, vol. 7, no. 1, 2007, pp. 1–16.

[43] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, *IRON file systems*. ACM, 2005, vol. 39, no. 5.

[44] "Cache and TLB Flushing Under Linux," 2019, "https://www.kernel.org/doc/html/latest/core-api/cachetlb.html".

[45] S. O. K. J. Amy Tai, Andrew Kryczka and A. C. Michael J. Freedman, "Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy," in *2019 USENIX Annual Technical Conference*. IEEE, 2019.

Batyr Charyyev is PhD student in Systems Engineering at the Stevens Institute of Technology. He received his Masters degree in Computer Science and Engineering from University of Nevada Reno. His research interests are in the areas of fault tolerance in data transfer between HPC platforms, trusted computing with Intel SGX and Trusted Platform Module and edge computing for IoT.

Engin Arslan is an Assistant Professor in the Department of Computer Science and Engineering, University of Nevada, Reno. He received his BS degree of Computer Engineering from Bogazici University, MS degree from University at Nevada, Reno and PhD degree from Computer Science and Engineering at University at Buffalo, SUNY. Prior to joining to UNR, he spent a year in National Center for Supercomputing Applications as a Postdoctoral Research Associate. His research interests include high performance networks, data intensive distributed computing, distributed systems, computer networks, and cloud computing.