Looking for Errors TCP Misses

Jack Fitzgerald
Department of Computer Science
Colorado State University
Fort Collins, USA
https://orcid.org/0009-0008-5604-7920

Anju Gopinath
Department of Computer Science
Colorado State University
Fort Collins, USA
https://orcid.org/0009-0003-7365-163X

Logan Cadman
Department of Computer Science
Colorado State University
Fort Collins, USA
https://orcid.org/0009-0005-4376-8225

Sepideh Abdollah
Department of Computer Science
Tennessee Technological University
Cookeville, USA
https://orcid.org/0009-0008-2045-265X

Susmit Shannigrahi
Department of Computer Science
Tennessee Technological University
Cookeville, USA
https://orcid.org/0000-0001-6213-7473

Craig Partridge
Department of Computer Science
Colorado State University
Fort Collins, USA
https://orcid.org/0000-0002-7156-1601

***Abstract*—Inspired by prior work suggesting undetected errors were becoming a problem on the Internet, we set out to create a measurement system to detect errors that the TCP checksum missed. We designed a client-server framework in which the servers sent known files to clients. We then compared the received data with the original file to identify undetected errors introduced by the network. We deployed this measurement framework on various public testbeds. Over the course of 9 months, we transferred a total of 26 petabytes of data. Scaling the measurement framework to capture a large number of errors proved to be a challenge. This paper focuses on the challenges encountered during the deployment of the measurement system.**

We also present the interim results, which suggest that the error problems seen in prior works may be caused by two distinct processes: (1) errors that slip past TCP and (2) file system failures. The interim results also suggest that the measurement system needs to be adjusted to collect exabytes of measurement data, rather than the petabytes that prior studies predicted.

***Index Terms*—TCP checksum errors, undetected network errors, measurement framework deployment, petabyte-scale data, FABRIC, CloudLab, National Research Platform**

I. Introduction

How do you detect and examine errors that the network stack missed?

A few years ago, multiple studies showed that a surprising fraction of large file transfers were delivering files with errors (the received file did not match the original) [1], [2]. Specifically the errors were not being detected by layer 2 and layer 4 error checks (e.g. Ethernet CRCs and the TCP checksum), but rather were detected by file hashes at the receiver, or by the secure socket layer above TCP during the file transfer.

We hypothesized some of these undetected errors were occurring in the network path and set out to discover the source(s) of errors. In work published elsewhere [3], we studied layer 2 and the ability of IEEE CRC-32 to detect modern errors. In this work, we step up to layers 3 and 4, and study errors that slip undetected past the TCP checksum.

We built the client-server measurement system shown in Figure 1, which moves known (read-only) files between servers and clients and detects data that does not match the known values. We discuss the design choices that led to this system, our experience deploying this system in various testbeds on the Internet, and the need to scale the system (or an alternative infrastructure) to capture 1,000 times more data.

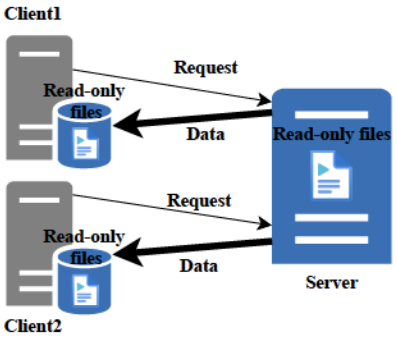

Fig. 1. Client Server Measurement

II. Related Work

A. Checksums

The TCP checksum also known as the Internet checksum was devised in the late 1970s as an easy-to-compute check against errors in the IP and TCP layers. [4] At the time, the community struggled to model errors more complex than single bit errors. (That said, the modeling of single bit errors was outstanding. See Hammond's work to evaluate IEEE CRC-32 [5].)

In the late 1990s, Stone and Partridge captured TCP segments with bad checksums and compared them against the correct retransmitted data to build a portrait of the error processes present in the Internet [6].

More recently, Koopman has revolutionized our understanding of checksums and CRCs. Central to Koopman's work is a characterization of errors by the Hamming distances and the length of the error (distance from first to last errored bit, inclusive), combined with a characterization of the error function (e.g. if it is an independent BER or something more

complex). Koopman has assessed the performance of various checksums and made a strong case that choosing the right error check algorithm depends heavily on the error environment in which it is to be used [7].

A central goal of this project was to collect enough errors that TCP had missed, such that we could do a Koopman-style analysis and determine what error checks *would* have detected the errors.

B. File Transfers

Large data transfers have become the norm in scientific collaborations. Scientific communities such as climate science and high-energy particle physics routinely transfer petabytes of data over wide-area-networks. Several previous works have shown surprisingly high error rates for these data transfers.

The work in [1] analyzed a three-year log from a Department of Energy (DoE) facility that served climate data requests, examining access patterns, request frequency, and failures. The study found 30% of requests failed to deliver a viable file

A study by the Globus community [2], showed that transferring data across the Worldwide Large Hadron Collider Computing Grid (WLCG) often faced 10% failures for more than 2PB of the traffic due to network issues, timeout issues, and checksum errors (due to storage hardware faults).

Finally, [8] showed that as data transfers between geographically distributed sites grow, the risk of undetected data corruption increases. In a 100 Gbps network, researchers observed data corruption of up to 5%.

Beyond network-related issues, undetected errors can also arise from factors like storage corruption in the receiving system [9], [10]. Several studies [8] [10] [11] [12], [13] [14], [15] [16] [9] [17] and [18] have shown that disks are particularly vulnerable to undetected errors during both reading and writing operations. While errors during file reading are generally transient, those that occur during writing are permanent, leading researchers to focus primarily on addressing the latter. These errors occur due to firmware or hardware malfunctions in disk drives or errors in flushing buffer caches.

In summary, hard-to-discover errors can happen in every subsystem involved in the generation and transmission of large files.

III. Creating A Detection System

How do you detect and analyze errors that TCP missed? That is a complex challenge that has not previously been studied.

There turned out to be two central issues: (1) estimating the frequency of undetected errors in the network; and (2) devising an application-level protocol that was robust (kept working well enough to measure undetected errors) even when any byte, data or control, could be corrupted.

A. Design

Based on prior studies [1], [19], our estimate was that we would average a handful of undetected errors approximately every terabyte of data, or roughly every million packets, transferred [20]. We sought to collect thousands of errors, as a large data set would presumably give a more complete picture of the cause of errors. So the system needed to measure petabytes of data.

We did not want to capture raw TCP segments (e.g. snoop the network) due to operator security concerns.

Our initial design called for a multiple replicated servers environment with thousands of lightweight clients that copied known files from multiple servers and used the replication to confirm received copies were correct. The core ideas were that (a) a client would consume no storage and could easily be installed on systems throughout the Internet; and (b) with multiple servers and thousands of clients we would get rich topological coverage. The client would send logging data back to the servers as transfers were completed (again, to be lightweight).

Why use known files rather than, say, have servers auto-generate data? Because, in earlier days of the Internet, there were cases of middleboxes being sensitive to particular data patterns. For instance, there were devices that struggled with packets of all zeros (apparently phase-locked loops in data lines would lose sync). We wanted to be able to try new patterns without having to rewrite software.

The lightweight client proved a disaster. We had to construct both a file transfer protocol and a logging protocol that were robust to errors not detected by TCP (which existing protocols were not). We found ourselves creating a Sorcerer's Apprentice protocol, where each non-data message had to be protected multiple ways, with additional messages, from the chance the original message contained an undetected error.

So we redesigned the protocol to be single servers such that each client had their own reference copy of each file being transferred. This led to a trivial protocol where the client requests a (known) file and the server acks the request followed by sending the entire file. With gigabyte files and small control messages, this approach seemed to promise that if data was corrupted, it would be in the file (not the control message), almost all the time. In the remote event a control message was in error, we would detect it but not necessarily be able to determine the error process that caused it.

B. Server-Client Implementation

We implemented the client and server in C for a Unix/Linux environment. Clients are single-threaded and communicate with a single server. Servers are multi-threaded and designed to handle over 500 clients concurrently. Total code size is approximately 5,000 lines of which about 500 are client-specific, 500 are server-specific, and the rest is shared (including 1,500 lines of a reference implementation of CRC64).

Clients are designed to run as a distinct user (`filedump-client`) whose home account contains *local* copies of reference files. We chose to use local files rather than, for instance, sharing a network attached storage (NAS) to ensure that if an error happened such that the local copy of the reference file was corrupted, we knew it was the

local file system, and not the local network between client and NAS. To create a client container, the client binary and a collection of files are copied into the container.

The client then connects to a server. The clients iterates through its directory of reference files and sequentially requests each file (using a filename, file size, and file CRC to ensure the server and client files match). Upon confirming a file match (name, length and CRC), the server will begin transmitting the file to the client. The file CRC and length are recalculated by the client *every* time the file is requested, to foil file system bit rot. To avoid churning connections, the client requests multiple files over a single TCP connection. When one file is successfully downloaded, the client issues a request for the next file.

As the client receives each buffer of file data, it compares the buffer with the data at the correct offset in the local copy. Any discrepancy means either (1) there was corrupted data that evaded detection by the TCP checksum; or (2) the client's or server's reference copy is corrupted. Discrepancies are logged, including all bytes that are in disagreement. The client also periodically logs the cumulative successful transmissions (so the total data transferred can be tracked).

We supported three permanent servers. These servers sat on high-speed networks near our organizational links to the outside world. Two servers ran production code and one was used to stage last-stage-before-production code. The server configuration is analogous to the client. The server runs as a specific user (`filedump`) and listens on a specific TCP port for connections. The servers log connections and data sent to clients. When servers needed to be stood up within testbeds, a container analogous to the client was created for the server.

Logs were collected nightly via `rsync` and backed up at our home institutions. Log collection proved to be one of the biggest deployment headaches. Every testbed had its own restrictions on our ability to run `rsync` to our clients' containers, so the log pulling scripts had to be customized to each testbed. (We were somewhat surprised there wasn't a common standard for retrieval of test data from testbeds).

C. Code Testing

Because we expected undetected data errors to be rare, we needed to ensure that our software (1) would correctly detect and log the errors and (2) would not generate false errors due to hard-to-test bugs being triggered. To this end, we performed rigorous testing to verify that our code was not only capable of detecting a failed checksum, but free of bugs that could give the illusion of a failed checksum. Our approach included both systematic testing and error injection to ensure that the file transfer process and the client-side detection mechanisms were implemented properly.

A critical part of our testing process involved having our server deliberately introduce bit errors after reading from the disk but before transmitting the files. A variety of error injections were used to assess the client-side code's ability to identify and flag corrupted data. These injections included flipping the first bit, zeroing the last byte, zeroing all bytes, randomizing all bytes, and inverting all bits in the file chunks sent to the clients. This approach allowed us to simulate a wide range of potential data corruptions that could occur during transmission.

In addition to error injection, we utilized several code-based tools to ensure the integrity and stability of our code. We used the `mallopt` function and `valgrind` [21] to search for memory allocation errors. We used `cppcheck` for static analysis of our code. We introduced "canary" variables into the stack with known fixed values, which surrounded buffers and could alert us to any unintentionally overwritten buffers. Furthermore, we performed fuzz testing on the application-level messaging protocol used by the servers and clients to establish the file transfer process, ensuring it could gracefully handle unexpected values without crashing or corrupting data.

TABLE I
MEASURED DATA BY TESTBED (IN TB)

Testbed	Total	Data Internal to Testbed	Data External to Testbed
CloudLab	3,565	2,436	1,129
FABRIC	17,413	11	17,402
Nautilus	4,135	3,168	967
Misc	525	0	525
Totals	25,638	5,615	20,023

IV. MEASUREMENTS

Where possible, we set up clients and servers to encourage the data transfers to flow through varied and heterogeneous network paths. Longer paths through different types of networking equipment are more representative of typical transfer scenarios and have an increased chance of corruption.

We had three servers at our home institutions. Two were stable release servers, and one was a last-stage-before-stable server. There turned out to be no measurement impacting bugs on the last-stage server, so its results are included below. They could handle 500 concurrent clients each.

We placed clients in containers on multiple public testbeds. The typical configuration had three clients, one for each server, in each container.

Some testbeds, notably CloudLab, had limited external connectivity and our tests could overwhelm their connection. In these cases, we placed a server in its own container inside the testbed, and conducted tests purely internal to the testbed, as well as external tests.

Client logs were retrieved daily and archived via `rsync`.

Table I lists the testbeds and volume of data collected from each. Data collected internally (between internal client and server) and externally (between the testbed and one of our three reference servers) is broken out. We discuss the details of each testbed below.

A. CloudLab

CloudLab is a flexible infrastructure for research on the future of cloud computing. It is designed to allow researchers to

work with both virtual and physical machines [22]. Computing resources are defined via profiles in the RSpec format, which allows users to specify the nodes, storage, hardware, software, networks, etc. that they would like to use in their experiment.

Our experiments in CloudLab included both external and internal setups. All nodes were configured to run Ubuntu with 8GB of RAM and 2 CPU cores. The external setup only established clients within CloudLab, and the internal setup established both clients and servers within the CloudLab infrastructure. For the external setup, we utilized two different CloudLab profiles: `OpenStack` and `dijon-vm`. The `OpenStack` profile allowed us to allocate clients on physical machines, while the `dijon-vm` profile enabled us to request physical machines with a specified number of VMs. The external clients were forced to use CloudLab's public facing interface, which was limited to a total of 0.1Gbps for all nodes. For our internal setup, we used the `multisite-lan` profile, which allowed us to allocate multiple physical nodes across two different sites. Server nodes were created on the Apt cluster, while client nodes were setup on the Emulab cluster. The Apt and Emulab clusters had a 40Gbps link between them, and the server and client nodes were connected to a VLAN which facilitated the communication between them.

B. FABRIC

FABRIC's mission is to provide an infrastructure for new ideas that are impossible or impractical with the current Internet. It is a geographically diverse testbed that provides researchers the ability to create VMs across 41 different sites [23]. The majority of sites in FABRIC operate over IPv6-only management interfaces, with only a few accommodating IPv4. This ended up being a bonus because several of our servers were IPv4-only and thus required NAT64 equipment, thus providing an additional middlebox in the network paths.

To establish our clients within the FABRIC infrastructure, we leveraged the FABlib API and FABRIC's JupyterHub server. We created slices consisting of 10 nodes at each FABRIC site, with each being allocated 2 cores, 8GB of RAM, and 100GB of storage, using the `default_ubuntu_22` image that loads Ubuntu 22.04 as the operating system. After deploying the nodes, we used the `get_ssh_command` function from FABlib to retrieve each node's IP address, enabling us to manage them with tools such as Ansible. Once the IP addresses were collected, we executed build scripts on each node and uploaded an executable of our file transfer client program. We then initiated the file transfer processes between the nodes and our three servers. All file transfers were conducted over the management interfaces of each node, which typically operate at 10Gbps. The nodes with an IPv6 interface suffered from a bandwidth limitation of 1Gbps due to the NAT64 translation required when communicating with our two IPv4-only servers. No internal experiments were performed within FABRIC.

C. Nautilus

The National Research Platform's (NRP) Nautilus testbed, formerly knows as the Pacific Research Platform [24], is a HyperCluster designed for running containerized Big Data applications. Nautilus utilizes Kubernetes containers to allocate resources. NRP contains ∼360 nodes from 68 universities and organizations spread across the USA, Europe and Eastern Asia. All participating nodes are expected to utilize a Science DMZ architecture [25], [26] and have 10Gbps - 100Gbps links.

We deployed out clients on Nautilus using K8s pods and spread them over multiple nodes. Our pods used ephemeral storage, but we needed to perform the setup process from scratch whenever a pod would restart. The rate of pod failure was relatively high in our experience, such that experimental data was lost, so we eventually modified our client setup to use permanent storage for logging and configuration. We created two persistent storage volumes on Nautilus using `rook-cephs-central`. One of the persistent storage volumes was initialized with 30GB of space and stored all the files involved in setting up the transfer process. The other persistent storage volume was initialized with 10GB of space and was responsible for storing the logs from the clients. The Nautilus clients were modified to have their logs directory become a symlink pointed at the mounted logging persistent volume.

We performed both external and internal experiments in Nautilus. The external experiment utilized a Kubernetes deployment that would create 100 pods in the Nautilus cluster. Each pod in the deployment was initialized with the official Ubuntu Docker image from Docker Hub without specifying a tag. Each pod's resources were capped at 2 GB of memory and 2000 millicpu. The clients would then mount the persistent storage volumes so that they could pull the necessary files and the client executable, and so that they could store their logs. Once the deployment started, the clients would run indefinitely until the deployment's lifetime expired.

V. Interim Results

There are two sets of results: data collected so far on errors and experience with the collection system.

A. Errors

To date our measurement system has transferred 26 petabytes of known data and captured a handful of errors. Both TCP errors and file system errors were found.

In the first weeks of testing the measurement system, we had ten messages that were flagged as corrupted messages that slipped past the TCP checksum. The messages had two odd characteristics: (1) they were all control messages (unlikely given that only 0.0001% of the data was control messages); and (2) the messages were unintelligible, as if TCP had sent the wrong buffer. The errors were so implausible that our working assumption was that there was a bug in our collection code and we spent weeks instrumenting the code for possible pointer errors, memory allocation errors, and similar bugs that could cause buffers filled with unknown data.

The code passed all the tests and our current view is that the errors were real. Additional control messages have happened since. That leaves us to answer the puzzling question of why all the errors were in rare control messages. The current hypothesis is, because the control messages often occur at the start of a TCP connection, we are seeing race conditions in middleboxes where they are installing state to recognize a new connection and accidentally scramble a buffer.

The file system error was a surprise. A client file system failed and corrupted a substantial fraction of a file, *while* the file was being transferred. The logs make clear the client's copy of the file (which was read from to the validate data from the server but never written to) was valid until suddenly, in the midst of a transfer, the file was permanently corrupted (the file's checksum changed).

The surprise was that none of the prior studies had suggested the potential for this error. They had been interpreted as either the server or the network delivering corrupted data. The notion that the client file system could corrupt data, in the span of a time it took to transfer a file, was not anticipated. This type of error would explain bad file sums at the end of transfers (as in [2]) but does not explain failing network connections (as in [1]).

Observe that, assuming these data points are confirmed with more testing, it is the worst of all possible outcomes as it says we cannot trust either (a) the network; or (b) the disk storage system. That makes developing a solution hard, as there's no reliable component in the file transfer process which we can use as the foundation of a reliable system. Rather we will likely have to craft systems that assume each component can fail, but combines the components such that the likelihood that all components fail is proportional to the product of the individual failure probabilities.

B. The Collection System

Having deployed the collection system for several months, there are a couple of vital lessons learned.

1) Overwhelming Traffic at Initialization: We were surprised to discover that creating 100s of containers often overwhelmed host networks, as each container had to download several gigabytes of data (most notably its copies of the reference files). Distributed systems clusters were particularly vulnerable if we downloaded client containers from outside their cluster. In contrast, running the measurement protocol only occasionally caused issues, which we were largely able to address by throttling clients.

2) More Clients and Data Needed: The original estimate that transmitting petabytes would yield 1,000s of errors was wrong. A paper published after we started measuring suggests exabytes are required [27] and the shortage of errors in our data would support that hypothesis.

Increasing the clients and data sharply increases the challenges of managing the measurement infrastructure and its collected traces and data. Managing 100s of containers across diverse testbeds with distinct interfaces was feasible, just barely, with a small team of students. Managing 1,000,000+ containers would require a new set of tools. Similarly, 100,000+ containers would require tools to estimate bandwidth use. We have estimated and monitored bandwidth informally (and in a few cases got it spectacularly wrong). But when our system is moving 1,000 times more data, informal estimates do not suffice.

VI. Looking Forward

After 26 PB of data collected, we have detected a small set of errors. That has been sufficient to prove there is an issue worthy of further study, but to get a true sense of the scope of undetected error issues we need at least three orders of magnitude more errors, which suggests three orders of magnitude more data needs to be collected. In this section, we try to anticipate the issues in creating a robust measurement at that scale.

A. Infrastructure Limitations

The current approach relies on the substantial testbed infrastructure, especially in the United States, that allowed us to deploy 100s of clients relatively easy. It is not clear that ecosystem is prepared to deal with experiments involving 100,000s of containers. To avoid overloading the testbeds, we may also prefer fewer containers but making them very long-lived (e.g. for months at a time). That's a challenge as many container providers emphasize providing containers for just a few weeks at a time.

B. Leveraging Less Well-Maintained Infrastructure

Most of our experiments were run on recently constructed and well-maintained testbeds. (Nautilus, with its older infrastructure, being the partial-exception). We are cognizant that errors are much more likely in older, less well-monitored, parts of the network, where a failing device may induce a profusion of errors. But deploying to less well-maintained parts of the Internet seems likely to magnify the management challenges noted in section V-B. Furthermore, such infrastructure are less likely be able to sustain the traffic flows required to capture a significant number of errors and record them for further analysis. Testing in these less reliable environments will require running smaller transfers for a long time, possibly months or years.

C. Thinking Exascale for Testing

Testing at exascale levels requires a fundamental shift in approach. For instance, we were willing to transfer tens of gigabytes of replicated data to replicated client containers to simplify testing of known data. If we scale up to collecting exabytes of measurements, that approach suggests transferring 10s of petabytes during the configuration process. Given that we often broke testbeds moving gigabytes, petabytes aren't currently feasible. So as we expand the testing infrastructure, we also have to make it lighter weight. For instance, we have thought about the positives and negatives of having clients and servers transfer random data using synchronized pseudo-random number generators.

D. Better Recognizing Cross-Subsystem Interactions

Errors often result from interactions between multiple subsystems, such as disk storage, network transmission, and other underlying components. It is critical to explore the implications of these compounded errors to fully understand their impact on data integrity. By dissecting how these different subsystems contribute and respond to errors, we can develop more robust error detection and correction mechanisms that address issues across the entire data path.

VII. Conclusions

So far, our error detection system has collected a small set of errors that includes both filesystem errors and errors caused by network errors missed by TCP. That result is enough to show that there are issues. But we need to collect far more errors (1,000+) to be able to properly characterize the error processes present in the network and possible remediations (e.g., doing a Koopman-style analysis and suggesting new TCP checksums, leveraging the TCP alternate checksum option [28]).

Building such a system to detect exponentially more errors will be challenging. We have sketched some of the challenges here and plan to investigate them in future work.

VIII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Numbers 2019163 and 2126148.

References

[1] Susmit Shannigrahi, Chengyu Fan, and Christos Papadopoulos. Request aggregation, caching, and forwarding strategies for improving large climate data distribution with ndn: a case study. In *Proc. of the 4th ACM Conf. on Information-Centric Networking*, pages 54–65, 2017.

[2] Luca Clissa, Mario Lassnig, and Lorenzo Rinaldi. Analyzing wlcg file transfer errors through machine learning: An automatic pipeline to support post-mortem distributed data management. *Computing and Software for Big Science*, 6(1):16, 2022.

[3] Mohit Balany and Craig Partridge. Is it time to upgrade from CRC-32? In *NOMS 2024-2024 IEEE Network Operations and Management Symp.*, pages 1–5, 2024.

[4] W.W. Plummer. TCP checksum, function design. Technical report, Internet Engineering Note 45, 1978.

[5] Joseph L. Jr Hammond, J.E. Brown, and S.S. Liu. Development of a transmission error model and an error control model. Technical report, Georgia Institute of Technology, 1975.

[6] Jonathan Stone and Craig Partridge. When the crc and TCP checksum disagree. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, page 309–319, New York, NY, USA, 2000. Association for Computing Machinery.

[7] Philip Koopman and Theresa C. Maxino. The effectiveness of checksums for embedded control networks. *IEEE Trans. Dependable and Secure Computing*, 6(1):59–72, Jan-Mar 2009.

[8] Ahmed Alhussen. *Fast and Secure Integrity Verification for High-Speed Data Transfers*. PhD thesis, University of Nevada, Reno, 2020.

[9] Hwajung Kim, Inhwi Hwang, Jeongeun Lee, Heon Y Yeom, and Hanul Sung. Concurrent and robust end-to-end data integrity verification scheme for flash-based storage devices. *IEEE Access*, 10:36350–36361, 2022.

[10] Nagmat Nazarov and Engin Arslan. In-network caching assisted error recovery for file transfers. In *2022 IEEE/ACM Intl. Workshop on Innovating the Network for Data-Intensive Science (INDIS)*, pages 20–24. IEEE, 2022.

[11] Moona Yakhchi, Mahdi Fazeli, and Seyyed Amir Asghari. Silent data corruption estimation and mitigation without fault injection. *IEEE Canadian Journal of Electrical and Computer Engineering*, 45(3):318–327, 2022.

[12] Batyr Charyyev, Ahmed Alhussen, Hemanta Sapkota, Eric Pouyoul, Mehmet Hadi Gunes, and Engin Arslan. Towards securing data transfers against silent data corruption. In *2019 19th IEEE/ACM Intl. Symp. on Cluster, Cloud and Grid Computing (CCGRID)*, pages 262–271. IEEE, 2019.

[13] Batyr Charyyev and Engin Arslan. Riva: Robust integrity verification algorithm for high-speed file transfers. *IEEE Trans. Parallel and Distributed Systems*, 31(6):1387–1399, 2020.

[14] Batyr Charyyev. Protecting file transfers against silent data corruption with robust end-to-end integrity verification. Master's thesis, University of Nevada, Reno, 2019.

[15] Md Arifuzzaman, Masudul Bhuiyan, Mehmet Gumus, and Engin Arslan. Be smart, save i/o: A probabilistic approach to avoid uncorrectable errors in storage systems. In *2022 IEEE Intl. Conf. on Cluster Computing (CLUSTER)*, pages 256–266. IEEE, 2022.

[16] Masudul Hasan Masud Bhuiyan. Probabilistic approach to avoid uncorrectable bit errors in storage systems. Master's thesis, University of Nevada, Reno, 2020.

[17] Hwajung Kim, Inhwi Hwang, and Heon Y Yeom. Efficient and robust data integrity verification scheme for high-performance storage devices. In *Proc. of the 36th Annual ACM Symp. on Applied Computing*, pages 1199–1202, 2021.

[18] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC'12: Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[19] Zhengchun Liu, Rajkumar Kettimuthu, Ian Foster, and Nageswara S. V. Rao. Cross-geography scientific data transferring trends and behavior. In *Proc. of the 27th Intl. Symp. on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 267–278, New York, NY, USA, 2018. ACM.

[20] Craig Partridge and Susmit Shannigrahi. Big data, transmission errors, and the internet. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2023.

[21] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with Bit-Precision. In *2005 USENIX Annual Technical Conf. (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[22] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *2019 USENIX annual technical Conf. (USENIX ATC 19)*, pages 1–14, 2019.

[23] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Inder S Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. Fabric: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing*, 23(6):38–47, 2019.

[24] Larry Smarr, Camille Crittenden, Thomas DeFanti, John Graham, Dmitry Mishin, Richard Moore, Philip Papadopoulos, and Frank Würthwein. The pacific research platform: Making high-speed networking a reality for the scientist. In *Proceedings of the Practice and Experience on Advanced Research Computing: Seamless Creativity*, PEARC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Jorge Crichigno, Elias Bou-Harb, and Nasir Ghani. A comprehensive tutorial on science dmz. *IEEE Communications Surveys & Tutorials*, 21(2):2041–2078, 2018.

[26] Emily Mutter and Susmit Shannigrahi. Science dmz networks: How different are they really? In *2024 IEEE 49th Conference on Local Computer Networks (LCN)*, pages 1–9. IEEE, 2024.

[27] Ladislav Pápay, Jan Pustelnik, Krzysztof Rzadca, Beata Strack, Paweł Stradomski, Bartłomiej Wołowiec, and Michal Zasadzinski. An exabyte a day: throughput-oriented, large scale, managed data transfers with effingo. In *Proc. of the ACM SIGCOMM 2024 Conf.*, ACM SIGCOMM '24, page 970–982, New York, NY, USA, 2024. Association for Computing Machinery.

[28] J. Zweig and C. Partridge. TCP alternate checksum options. RFC 1146, RFC Editor, March 1990.