MOOSEGUARD: SECURE FILE SHARING AT SCALE IN UNTRUSTED ENVIRONMENTS

by

Joseph C. Baker

B.S. in Computer Science, University of Pittsburgh, 2017

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2020

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This thesis was presented

by

Joseph C. Baker

It was defended on

July 16, 2020

and approved by

Jack Lange, Associate Professor, Department of Computer Science

Adam J. Lee, Associate Professor, Department of Computer Science

Xulong Tang, Assistant Professor, Department of Computer Science

Thesis Advisor: Jack Lange, Associate Professor, Department of Computer Science

Copyright © by Joseph C. Baker
2020

MOOSEGUARD: SECURE FILE SHARING AT SCALE IN UNTRUSTED ENVIRONMENTS

Joseph C. Baker, M.S.

University of Pittsburgh, 2020

Shared storage systems provide cheap, scalable, and reliable storage, but secure sharing in these systems requires users to encrypt their data and limit efficient sharing or trust a service provider to faithfully keep their data private. Current research has explored the use of trusted execution environments (TEEs) to operate on sensitive data and sharing policies in isolated execution. That work enables the utilization of untrusted shared resources to store and share sensitive data while maintaining stronger security guarantees. However, current research has limitations in scaling these solutions, as it bottlenecks both metadata and data operations within the same physical TEE, whereas a scaled file system distributes metadata and data operations to separate devices.

This paper explores the use of two TEEs specialized for metadata and data operations to provide file sharing at scale with less overhead in addition to strong security guarantees. This approach achieves scaled metadata and concurrent use by utilizing a server-side TEE for isolated execution on a master server and provides data privacy and efficient access revocation through a client-side TEE. MooseGuard is the prototype implementation of this design, utilizing Intel SGX as a TEE and extending the MooseFS distributed file system. MooseGuard's implementation details the modifications needed to provide security and shows how this approach can be applied to a typical distributed file system. An evaluation of MooseGuard demonstrates that TEEs specialized for metadata and data operations allow a secured distributed file system to maintain its scale with only constant overheads. As TEEs and secure hardware become more widely available in public clouds, enterprise, and personal devices, MooseGuard presents a way for users to get the best of both worlds in data privacy and efficient sharing when using scaled, shared storage systems.

Keywords: Security; Distributed File Systems; Intel SGX; MooseFS; Cloud.

Table of Contents

1.0 INTRODUCTION . . . 1
2.0 BACKGROUND . . . 4
2.1 THREAT MODEL . . . 5
2.2 TRUSTED EXECUTION ENVIRONMENTS (TEEs) . . . 7
2.2.1 Intel SGX . . . 7
2.2.1.1 Isolated Execution . . . 7
2.2.1.2 Attestation . . . 8
2.2.1.3 Sealed Storage . . . 8
2.2.1.4 Limitations . . . 8
2.3 DISTRIBUTED FILE SYSTEMS . . . 9
2.3.1 Distributed File System Properties . . . 9
2.3.2 MooseFS . . . 9
2.3.2.1 Master Server . . . 10
2.3.2.2 Chunk Servers . . . 10
2.3.2.3 Clients . . . 11
2.4 RELATED WORK . . . 11
2.4.1 TEEs . . . 11
2.4.2 File Systems . . . 12
3.0 MOOSEGUARD . . . 13
3.1 MASTER SERVER . . . 13
3.1.1 Master Enclave . . . 15
3.1.2 Metadata Cache . . . 16
3.1.3 Persistent Metadata . . . 17
3.2 SECURE COMMUNICATION . . . 18
3.2.1 Attestation . . . 19
3.2.2 User Authentication and Permissions . . . 20

3.3 CLIENTS 21
3.3.1 Client Enclave 22
3.3.2 File Encryption Scheme 22
3.3.2.1 Chunk Server Modifications 23
3.3.3 File Access Control and Revocation 24
4.0 SECURITY ANALYSIS 27
4.1 CONFIDENTIALITY AND INTEGRITY 27
4.2 AUTHORIZED ACCESS 27
4.2.1 Authorizing User Access 28
4.2.2 Securing the Enclave Boundary 28
4.2.3 File Access and Revocation 29
4.3 CONSISTENCY 29
5.0 EVALUATION 31
5.1 FILE I/O BENCHMARKS 31
5.1.1 File Sizes 32
5.1.2 Access Patterns 33
5.1.3 Client Scaling 34
5.2 METADATA BENCHMARKS 35
5.2.1 Directory Tree Sizes 35
5.2.2 Master Metadata Cache 37
5.2.3 Client Scaling 38
5.3 APPLICATION BENCHMARKS 39
5.3.1 UNIX Applications 39
5.3.2 Spark 41
5.4 EVALUATION TAKEAWAYS 42
6.0 CONCLUSION 43
Bibliography 44

List of Tables

1 MooseGuard Encryption Metadata 14

2 UNIX Directory Trees 40

3 Spark Performance 42

List of Figures

1 MooseGuard Deployment 5

2 MooseGuard Architecture 14

3 MooseGuard File Extensions 23

4 MooseGuard Read Access Control 25

5 MooseGuard Write Access Control 26

6 File I/O Overheads over Varying File Sizes 32

7 File I/O Performance over Access Patterns 33

8 File I/O Performance over Client Scale 34

9 Metadata Performance over Directory Tree Sizes 36

10 Metadata Cache Impact on Performance 37

11 Metadata Performance over Client Scale 39

12 UNIX Application Performance 41

1.0 INTRODUCTION

Shared storage solutions such as cloud storage or distributed file systems in private data centers allow users to easily store and share large amounts of data. However, using cloud storage requires users to accept terms of service that do not hold the cloud service provider accountable for lapses in privacy [4, 8], and private data centers need to enforce segmented access to systems to achieve security requirements. Unintentionally or otherwise, there are cases where a cloud service provider or a private data center can leak user data [14, 3, 15]. These terms of service and restricted utilization of resources combined with the likelihood of a data breach present an ultimatum. Users must sacrifice the privacy of their data or the efficiency of sharing that these scaled services provide. This leaves existing users' sensitive data vulnerable and prohibits users with stricter privacy requirements from deploying cloud-based storage solutions or utilizing shared infrastructure.

Past efforts to protect sensitive data have required users to either place additional trust in service providers and their platforms or to use a solution that performs client-side encryption, but suffers from expensive or complex access revocation [22, 29, 31, 33, 40]. More recent research has explored the use of trusted execution environments (TEEs) to implement file sharing mechanisms with stronger security guarantees while requiring less trust in the service provider [27, 35]. These approaches have shown the feasibility of using TEEs to perform file sharing, but have limitations in scale due to hardware restrictions or no server-side support.

Typical distributed file systems (DFSes) are organized so that data-bound operations can be distributed to clients and storage servers and metadata-bound operations can be distributed to a master server. Recent works to enable secure file sharing utilize a single TEE on each device to perform sensitive data and metadata operations. NeXUS [27] uses only client-side enclaves, which limits the scale of metadata and concurrent use with the file system as clients must synchronize and share metadata updates. Pesos [35] uses server-side enclaves for policy enforcement but requires specialized hard drives to keep data secure. This paper considers the use of two types of enclaves, allowing each enclave to specialize in metadata- and data- bound operations respectively. This approach allows a DFS to maintain

its typical architecture of separating metadata and data operations, providing greater scaling of metadata and users and smaller overheads while maintaining the security of users' data.

This paper presents MooseGuard, an approach to extend a distributed file system deployed in an untrusted environment that can provide secure scalable storage, and efficient file sharing and access revocation. This paper first abstracts the required behavior for this goal, then presents an approach for implementing a solution in a way that is compatible with a broad class of DFSes. The basis for additional security in a DFS stems from the observation that a DFS may utilize modern approaches to authorize users and encrypt file data, but this is insufficient if users cannot trust the platform the file system runs on or if the file system cannot trust users to honestly execute file encryption. A service provider can arbitrarily abuse a master server to change the file system, network traffic can be manipulated, and clients can ignore access revocation to file data by directly accessing storage servers or reusing old file encryption keys. MooseGuard identifies that sensitive operations must be executed in isolation from an untrusted host, including metadata and user login operations on a master server and file encryption and access enforcement on a client.

MooseGuard's generalized approach leverages TEEs to perform isolated execution for both clients and servers within the file system. This enables the use of a master server in an untrusted environment for greater scaling of the file system, and a client enclave allows key sharing between users without requiring re-encryption of data when access is revoked. To address the required improvements in a DFS, MooseGuard uses common features of TEEs to strengthen the security of the file system in the face of untrusted client and server behavior. Specifically, this is accomplished by protecting the confidentiality and integrity of user data and metadata, ensuring consistent and integrity-protected metadata operations, and implementing stricter user logins that operate within a TEE and enforce correct file permissions.

The MooseGuard prototype is implemented using the distributed file system MooseFS [9] and Intel SGX [39] as a TEE. In addition to generalizing the approach to securing a DFS, this paper presents the details of this prototype implementation and an evaluation of the prototype. The findings in the evaluation show that using both client and server enclaves in a DFS allows for both file sharing and privacy with much greater scale than

in previous efforts. Previous efforts like NeXUS showed greater than linear overheads in scaled metadata operations and limited focus of concurrent use to personal-sized workloads. MooseGuard improves upon these results by enabling scaled sizes of data storage, metadata structure sizes, and concurrent client usage with constant overheads. With TEEs becoming increasingly available, especially in cloud environments [11], MooseGuard can enable efficient and secure file sharing in the cloud for more users and more data.

This paper presents the following contributions:

1. The requirements of a secure DFS on an untrusted platform and a novel approach to securing a DFS for deployment on an untrusted environment using client- and server-side TEEs.
2. MooseGuard, a prototype implementation of this approach using Intel SGX enclaves and MooseFS.
3. An evaluation of the overheads that this approach introduces with focuses on file and metadata benchmarks and different applied uses of a DFS.

The remainder of this paper is organized as follows: Chapter 2 provides the necessary background including MooseGuard's goals, deployment and threat models, overviews of the technologies used, and a summary of related works. Chapter 3 presents MooseGuard, detailing it's design considerations and implementation. Chapter 4 analyzes the security guarantees of MooseGuard given the threat model and the design of MooseGuard. Chapter 5 describes the evaluation of MooseGuard and a discussion of the results of this evaluation. The paper concludes with a summary and future work in Chapter 6.

2.0 BACKGROUND

Shared storage services often provide cheaper, more resilient storage while enabling sharing of data with ease. However, current shared storage models on untrusted platforms require users to choose between efficient file sharing and data privacy. This trade-off stems from the analysis that key management and access revocation while sharing encrypted data is often prohibitively expensive [32]. This limitation hinders many common use cases of scaled storage, such as sensitive government or medical data, from utilizing the cloud. Service providers and cloud users have even worked around this problem by enumerating special end-user license agreements or other contracts to ensure that service providers do more than act in good faith to protect users' sensitive data [2, 13].

MooseGuard's use of TEEs proposes an alternate approach to store and share private data in an insecure environment. MooseGuard's efforts are guided by two goals. The first goal is to provide a secure and scalable file system by protecting the confidentiality and integrity of user data while enabling sharing and efficient access control and preventing unauthorized modifications of the file system state. The second goal is to provide an approach to protecting data that can be implemented on a typical distributed file system with minimal operational overhead. These goals make MooseGuard worthwhile by providing an attainable solution to the problem at hand.

Private data centers and cloud services frequently virtualize applications so the management of services can be abstracted from the maintenance of hardware and software infrastructure. Segmented access to hardware has been used to ensure stricter security guarantees, but requiring this prevents providers from further applying this abstraction and achieving greater scale. This paper envisions the deployment of a DFS with MooseGuard enhancements in an untrusted environment. By utilizing TEEs instead of segmenting whole systems, this deployment model allows a DFS to be run as-a-service, restoring the virtualization of services. Figure 1 illustrates this intended deployment model for MooseGuard.

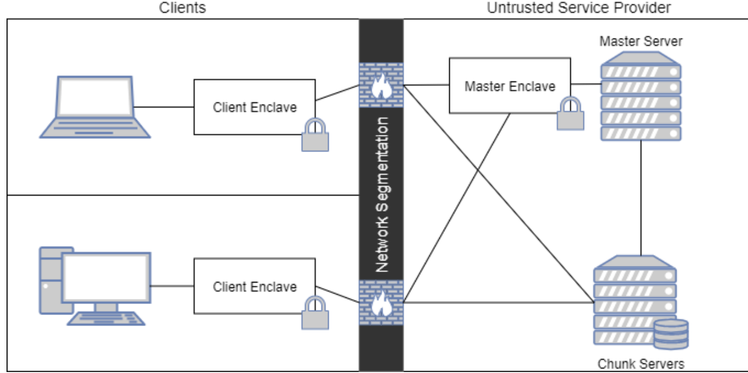

Figure 1: MooseGuard Deployment

2.1 THREAT MODEL

MooseGuard assumes an adversary with abilities similar to adversaries discussed in related TEE-based research [27]. An attacker can manipulate the execution of all software outside of a TEE and manipulate the operation of all of a platform's hardware except the SGX-enabled CPU. This includes access to the service provider's platform OS and hypervisor. The attacker can also observe, modify, drop, and reorder any network transmissions between clients and the service provider's platform. MooseGuard trusts that the CPU package is not physically tampered with. Further, MooseGuard assumes that SGX's isolated execution, attestation, and sealing mechanisms operate correctly. Finally, MooseGuard assumes that MooseFS, MooseGuard, and SGX SDK [5] code implemented within the TEE's trusted codebase (TCB) is free from bugs and security vulnerabilities.

The master server and storage servers of this DFS are expected to be deployed in a possibly untrustworthy platform where an adversary can apply all of its abilities listed above. Though client devices may be partitioned on a separate network, as shown in Figure 1, MooseGuard assumes that a client device is susceptible to the same attacks as a master or storage server and is therefore hosted on an untrusted platform. Because clients are not initially trusted, MooseGuard validates the integrity, authenticity, and authorization of clients before sharing sensitive information. MooseGuard does not protect plaintext sensitive

data outside of the client enclave on client systems and assumes that clients will take proper steps to keep this data private from such an adversary. This assumption considers the common case where client's trust in the device that they use to access data is proportional to the sensitivity of the data being accessed.

Given an adversary with these capabilities, an attacker may attempt to interfere with the operation of the file system by adding, modifying, or erasing file metadata or data on the master or storage servers. The attacker may further interfere by attempting to mislead clients or pose as an additional client to make unauthorized changes to the state of the file system. On a client system, data is already available in plaintext, so the role of the adversary is reduced. A client adversary focuses its attack on gaining unauthorized access to data or obtaining unauthorized access to the master. MooseGuard assumes that operations received from an authenticated and authorized client are intended, so a compromised client system may alter the file system state as much as that client's access allows.

MooseGuard's security objectve considers several factors in light of such an adversary. First, MooseGuard seeks to prevent any unauthorized access or manipulation of file data and metadata. The integrity of this information is also protected by detecting unauthorized modification both at rest and in transit. MooseGuard aims to further secure the file system by restricting access to only authenticated and authorized clients and enforce file permissions for each user. The next objective of MooseGuard is to enforce fork-consistency [37] of file data so that users cannot view or modify file data more recent than the data available at the time access was revoked. MooseGuard maintains the integrity of the file system's directory structure, but the structure may be visible to the adversary by observing access patterns. Confidentiality of the directory structure could be provided by implementing an ORAM [16] technique on directory structure accesses, but this approach is left for future work. MooseGuard protects the consistency of the file system's metadata state by ensuring that the order of operations cannot be altered, but does not consider rollback attacks that perform a wholesale restoration of file system metadata state to an earlier version. Finally, MooseGuard assumes the service provider will attempt to provide general availability of the master and storage service. Denial of service attacks, enclave side-channel attacks [24, 57, 25], and hardware-based attacks on an SGX-based CPU are not considered.

2.2 TRUSTED EXECUTION ENVIRONMENTS (TEEs)

Trusted execution environments (TEEs) are secure regions of hardware that provide a platform to execute code with strengthened confidentiality and integrity. By securing an area of a CPU to perform isolated execution, TEEs can reduce the attack surface of an application down to the physical CPU package and reduce the level of trust required to run a sensitive application on an otherwise insecure platform. TEEs such as Intel SGX [39] and ARM TrustZone [1] have become increasingly available for both consumer and enterprise markets [11]. TEEs have several models in which they can be deployed on servers and clients, making them useful for a wide range of applications. Service providers can provision TEEs on remote servers or on client devices to accomplish Digital Rights Management. Alternatively, clients can utilize TEEs present on a service provider's platform to perform secure computation in a multitenancy environment. MooseGuard's design includes both client and server model deployments of TEEs.

2.2.1 Intel SGX

Intel SGX is a feature in modern Intel CPUs that provides three principal functions of a TEE: isolated execution, attestation, and sealed storage. Together these features provide a way to develop applications that are both secure and powerful. The following sections provide a more detailed background on the functionality of these SGX features.

2.2.1.1 Isolated Execution SGX provides isolated execution through enclaves. Enclaves are shared libraries implemented with fixed entry and exit points, ECALLs and OCALLs. Enclaves achieve isolated execution by securing the CPU instructions and memory used while executing the code within an enclave. The Enclave Page Cache (EPC) is a physical segment of memory claimed by the CPU and dedicated to enclave use. Data bound for the EPC is encrypted by the CPU's memory controller. Isolated execution of instructions is accomplished by a new SGX CPU instruction, which switches the CPU into a secure mode and jumps to one of the enclave's well-defined entry points. While in this secure mode, the

CPU can only execute instructions loaded into the EPC. As a result, the enclave code must exit the enclave before the application can execute typical system calls. Similarly, interrupts sent while in the secure mode must save enclave context and exit the enclave before handling the interrupt.

2.2.1.2 Attestation Attestation allows users to establish trust with enclaves created in local or remote environments. SGX embeds unique keys in each SGX-enabled CPU and uses a new SGX instruction, which computes a quote of the enclave. SGX defines a quote as a secure signature of the enclave's EPC with this identifying key. Attestation is the process that users follow to challenge an enclave to prove that it is genuine and that its integrity is intact. In local attestation, users can confirm the authenticity of a local enclave and establish a shared key for secure communication to the enclave. Remote attestation additionally enables a user to confirm the authenticity of an enclave on a remote system by leveraging the Intel Attestation Service (IAS). The IAS receives enclave quotes and confirms that the quote was created by a genuine SGX enclave.

2.2.1.3 Sealed Storage Enclaves can only be useful if sensitive data can be passed in or out. The enclave interface and secure channels established during attestation provide a mechanism to exchange sensitive data, but these channels are ephemeral. In addition to these mechanisms, SGX provides the ability to seal and unseal enclave data. Sealing generates a key that is unique to the enclave implementation and specific CPU where sealing was invoked. This key allows an enclave to persistently store data on a local untrusted device.

2.2.1.4 Limitations Developing with SGX presents limitations that can impact the design of an application. Enclaves are limited in the number of system resources they can use and by how much they can trust the resources they are provided. Specifically, SGX only offers a non-configurable 128 MB of memory for the EPC shared on all enclaves on a system. Any additional memory used by an enclave must encrypt pages and evict them into unprotected memory. Additionally, enclaves are limited to a static number of threads and must encrypt the enclave's state during context switches, increasing the cost of transitions

around ECALLs and OCALLs. It is also up to the application developer to protect their application from side-channel attacks against SGX applications.

2.3 DISTRIBUTED FILE SYSTEMS

This section provides a brief background on distributed file systems. Specifically, Section 2.3.1 reviews the properties of DFSes that are impacted by the design of MooseGuard. Section 2.3.2 provides a detailed background on MooseFS, through which MooseGuard's prototype is implemented.

2.3.1 Distributed File System Properties

Fundamental properties of a distributed file system include access transparency, location transparency, and concurrency control [46]. Access transparency in a DFS provides consistent file data access for a user whether the data is local or remote. Location transparency enables the decoupling of file metadata and data so that data can be stored remotely at scale. Concurrency control allows multiple users to operate on data in the file system in parallel while all clients maintain a consistent view of the file system state. Location transparency and concurrency control are typically accomplished with a master server (or name server). Clients will consult the master server to locate and serialize access to files, and provide access transparency by presenting a uniform interface to clients and resolving remote requests through that interface.

2.3.2 MooseFS

MooseGuard's design is intended to be implementable on a DFS that employs a master server to handle location transparency and concurrency control and where clients access distributed file data. This paper implements MooseGuard enclaves on top of MooseFS [9], a general-purpose and open-source DFS. MooseFS is a practical, established, and stable DFS that supports many use cases and can be deployed on shared infrastructure in cloud or data

center environments. MooseFS's design is spread across a master metadata server, chunk servers for storage, and clients that interact with both server types. This approach is similar to approaches taken by the Google File System [30], Lustre [7], and Ceph [58]. MooseFS demonstrates the typical properties of a DFS, so implementations of MooseGuard on other DFSes should be able to follow patterns established by the prototype.

2.3.2.1 Master Server MooseFS uses a master server to store file metadata, synchronize client file access, map file names to chunk servers, and manage deployed chunk servers. The master server is implemented as a single-threaded userspace daemon process. All file metadata in the file system is stored in memory. This approach allows for a simple master server implementation while keeping the master fast enough to serve a large number of clients and chunk servers. The open-source implementation of MooseFS persists file metadata to local storage on the master server and replicates that persisted file to designated metadata logger daemons on remote servers for redundant backups. MooseFS offers a paid "pro" version of their software, which supports hot-standby backup master servers. Additionally, Yu et al. implement a distributed metadata server architecture [59]. Our implementation of MooseFS uses the open-source, single master server implementation of MooseFS.

2.3.2.2 Chunk Servers MooseFS chunk servers store and replicate file data in the file system. Chunk servers are implemented as a multi-threaded userspace daemon process. Chunk servers are intended to be deployed on heterogeneous devices for simple and cost-efficient scaling of the file system's raw storage capacity. Each chunk server can locally optimize checksumming and storing chunks of data on local storage. The master server manages chunk servers and coordinates replication of chunks, while chunk servers perform the I/O-bound replication operation between each other. On chunk writes, chunk servers are provided a list of chunk replica locations from the client, and chunk servers handle the I/O-bound replication of a write before acknowledging the write to clients. MooseFS supports at least 100 chunk servers and up to 16,384 petabytes of data.

2.3.2.3 Clients

MooseFS uses FUSE [56] to present a uniform POSIX file system interface to clients. Client interactions between the master and chunk servers are implemented by a multi-threaded userspace daemon process. The MooseFS client leverages several buffers: the system page buffers, FUSE synchronous and asynchronous buffers, and MooseFS client caches for directory entries, file attributes, and chunk locations. Clients communicate with the master server for all metadata operations and chunk lookup operations, and communicate directly with chunk servers for chunk read and writes.

2.4 RELATED WORK

2.4.1 TEEs

NeXUS [27] achieves practical secure file sharing for users by implementing a file system within a client-side enclave and uses the cloud to store and share data. NeXUS maintains efficient access revocation and wide compatibility with distributed storage by only using client-side enclaves, with limited scaling of file system metadata. Pesos [35] implements a secure object store with rich policies to control and audit access. Their approach utilizes server-side enclaves and specialized Kinetic disks for deployment in untrusted environments. MooseGuard's approach examines the trade-off between convenience and scale, utilizing both client and server enclaves. This approach enables greater scale in data and metadata storage and in concurrent use. Other SGX research in data storage considers stricter privacy by preventing Iago attacks [51], using ORAM techniques in file access [16], and other forms of data management such as securing a DBMS in an untrusted environment [44].

TEEs have also been used in a wide range of use cases. Many efforts have explored providing a secure virtual system to deploy applications on insecure platforms [21, 55, 19, 54, 50]. Applications for secure data processing in the cloud have been built, including machine learning and analytics [48, 47] and secure messaging [18, 34]. Research has found vulnerabilities, particularly through side-channel attacks, in the implementation of SGX [24, 57, 25]; there have also been efforts to mitigate these flaws [26, 49, 41]. Several papers have

proposed ways to optimize or provide enhanced features to SGX, specifically through better memory management [38, 53] or by avoiding enclave exits [42]. Finally, various techniques have been examined to enable more secure application development within TEEs with stricter integrity checking [20, 36, 23, 43].

2.4.2 File Systems

Distributed file systems research has seen growth from original principals of a DFS [45, 10] into highly scaled and specialized deployments that focus on high throughput, parallelism, or size in specific applications [30, 58, 7, 52, 28]. MooseGuard considers the fundamental features of a DFS so that its design applies to many of these systems. Cryptographic file systems research has previously explored locally encrypting files, which typically aims to secure the file system against other non-privileged users and assumes trust in the host OS [22, 29]. Research focused on providing stricter security in DFSes for untrusted environments has focused on encryption at the client-side [31, 33, 40]. These efforts are limited by expensive access revocation or complex key management. MooseGuard's use of TEEs enables both improved security and efficiency.

3.0 MOOSEGUARD

MooseGuard is an approach towards securing a distributed file system so that the file system is capable of being deployed in an untrusted environment. The modifications that MooseGuard propose leverage SGX enclaves on both clients and on a master server within a DFS. Enclaves specialized to handle metadata and data operations allow a DFS to operate at scale while protecting the confidentiality and integrity of a user's data and metadata. MooseGuard identifies three general components of a DFS that need to be extended to support its goal of a secure file system on an untrusted platform. These components are isolated execution of sensitive metadata operations on a master server, enhanced authentication and authorization of users, and isolated execution of file encryption and decryption on clients.

The following sections of this chapter describe the design and implementation of MooseGuard used to extend MooseFS. Figure 2 depicts the components surrounding MooseGuard and how they interact. Isolated execution on the master is accomplished by the Server Enclave and described in Section 3.1. Enhanced authentication and authorization of users between clients and the master server utilizes SGX's attestation properties and is detailed in Section 3.2. Trusted file encryption and access control enforcement is performed on clients via the Client Encalve and is described in Section 3.3. Table 1 lists all of the keys and other encryption metadata introduced by MooseGuard, grouped by each component in which they are used. The utilization of each value is described in the following sections.

3.1 MASTER SERVER

Many distributed file systems employ the use of a master server to centralize and synchronize client operations and to act as a name server to locate chunks of data on other servers. Deploying a master server on an untrusted platform can threaten the security of a client's sensitive information stored on the system. An untrusted or vulnerable service provider can read, modify, or erase file metadata and present false locations as a way to pass

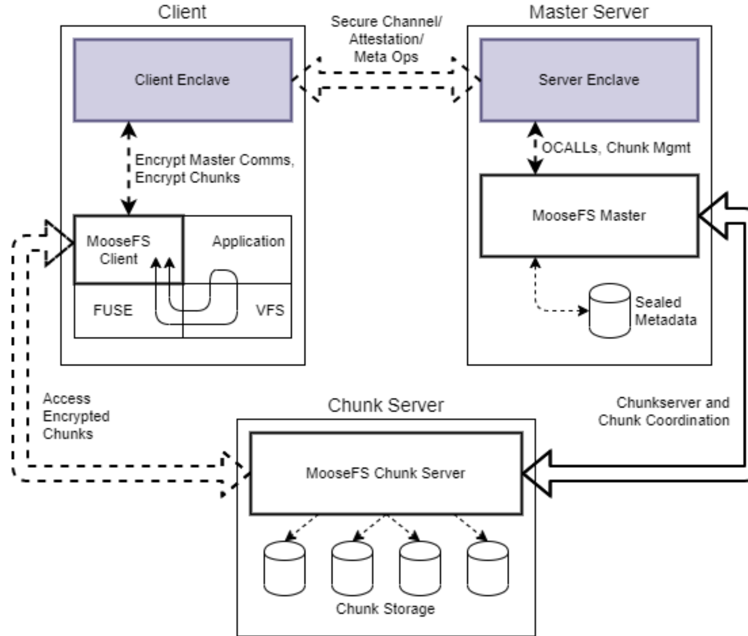

Figure 2: MooseGuard Architecture. Greyed boxes represent MooseGuard extensions to MooseFS, bold boundaries highlight components of MooseFS and MooseGuard, and dashed lines indicate encrypted data in transit.

Table 1: MooseGuard Encryption Metadata

Symbol	**Type**	**Description**	**Key Usage**		
			Master Enclave	**Client**	**Client Enclave**
K_r	AES-GCM 128 bit	Metadata Cache Sealing	Y	N	N
K_c	AES-GCM 128 bit	Metadata Changelog Sealing	Y	N	N
K_m	AES-CTR 128 bit	Metadata Backup Sealing	Y	N	N
K_f	AES-GCM 128 bit	File Encryption Key	Y	N	Y
H^i_{int}	SHA-256	File Chunk Hash Versioning	Y	N	Y
H^i_{ext}			N	Y	Y
T^i_j	AES-GCM 128 bit	File Encryption Autenticated Data	N	Y	Y
K_u	ECDSA-256	User Access Permission	Y	N	N
K_u^{-1}			N	Y	Y
K_r	AES-GCM 128 bit	Client ↔ Master Secure Channel	Y	N	Y

false file data to clients. However, the use of a master server also provides greater scalability of both a DFS's total storage and metadata sizes.

MooseGuard enables a DFS to utilize a master server while addressing the security concerns of an untrusted platform by performing sensitive operations in a server enclave. Additionally, MooseGuard supports scaled metadata sizes regardless of a TEE's memory limitations by implementing a caching layer for metadata. Finally, the master server can persistently store metadata backups by utilizing the sealing functionality of a TEE. Together, these functionalities secure the master server functionalities enabling the deployment of a DFS for secure use in an untrusted environment.

3.1.1 Master Enclave

MooseFS's master server coordinates changes to the state of a file system, manages chunk servers, chunks, and stores all metadata in RAM for quick access. MooseGuard partitions the work of the master server into trusted and untrusted operations. This approach separates the secure policy decisions of the file system from the underlying server mechanisms that the service provider is responsible for maintaining. Partitioning the master server functionality reduces the TCB of file system policy decisions that must be kept secure. By removing the untrusted operations from the TCB, the scalability of the file system increases as it allows the service provider to manage raw storage and networking at large scale.

Trusted operations include attesting and establishing secure communication with clients, coordinating file system state changes (operations on metadata), and sealing and unsealing metadata stored in untrusted spaces on the master server. Untrusted operations include managing chunks and chunk servers, persisting reliably to disk, and all network communications. The master enclave defines the boundary between trusted and untrusted operations as SGX ECALLs and OCALLs. Untrusted code may invoke an ECALL to respond to a user- or system- driven event that needs to change the file system state. Trusted code may invoke an OCALL to perform a system call in response to an ECALL and trusted processing. Implementing MooseGuard with a different file system, or creating master server logic from scratch with SGX enclaves in mind would yield a different number of ECALLs and OCALLs.

However, any implementation will have similar categories of operations.

Although the interface of the master server's trusted operations is reduced to well-defined entry points, an attacker could still modify the integrity of the file system's state by invoking ECALLs in an unexpected order, or with unexpected inputs. Unexpected inputs are considered during ECALL transitions with parameter checking implemented by SGX's SDK. MooseGuard protects against the issue of unexpected ECALL ordering by grouping ECALLs into phases. During each phase, ECALLs from other phases are rejected. The first phase is an initialization phase where the enclave is created and persisted metadata is unsealed into the enclave. The second phase allows new users to connect, attest, and issue file system operations through a secure channel. During this phase, only the ECALLs to receive client attestation and file operation messages are enabled. The final phase halts the file system, sealing new versions of metadata and terminating the enclave.

3.1.2 Metadata Cache

DFSes may need to utilize much more memory than a TEE can provide. Specifically, MooseFS stores all of the file system's metadata in memory for quick access. This can be several gigabytes in size, but the SGX EPC is much smaller. The default SGX implementation provides a workaround where enclave memory can be paged out by encrypting and evicting pages to untrusted memory, but this can be up to 35x slower than typical random memory accesses as the eviction process must access a special memory encryption engine [53]. MooseGuard takes an alternative approach by using untrusted memory as the backing store of metadata objects and implementing a cache for these objects within the server enclave. The cache encrypts all metadata objects before evicting objects to untrusted memory. This approach allows the master enclave to define when encryption-based paging occurs and optimize the cache based on access patterns of metadata instead of memory access patterns. MooseGuard's implementation leverages the SGX SDK sealing APIs to derive a key to seal evicted metadata objects, K_r from Table 1. Each object is encrypted using AES-GCM 128-bit encryption, and encrypted data is authenticated using AEAD. Since K_r is only used to encrypt non-persistent data, this key is ephemeral and only used for the lifetime of the

enclave.

A DFS implementing this approach with MooseGuard should cache enough metadata to prevent the thrashing of metadata objects for a single client operation. The cache implementation for MooseGuard on MooseFS uses as much free enclave memory as possible to improve the performance of repeated and predictable file system operations. This enables MooseGuard's server enclave to traverse an entire directory path for operations such as the POSIX extension readdirplus. The implementation of this cache is a simple, 60 MB LRU write-through cache. This cache is large enough to hold approximately 200,000 MooseFS metadata objects which may include file names, inodes, and symlink information. This implementation holds a working set large enough to satisfy large workloads, and its performance is measured in Section 5.2.2; further optimization of this cache is left for future improvement.

3.1.3 Persistent Metadata

Saving metadata to disk is an important requirement for the master server of any DFS. MooseGuard considers this functionality and utilizes the sealing capability of a TEE to accomplish this. In MooseFS's implementation of saving metadata, the master server logs each file system operation in a changelog on-demand and compacts changelogs into binary checkpoints of the file system metadata on an interval (by default every hour for MooseFS). The master server uses SGX sealing capabilities to derive enclave-specific keys, K_c, K_m shown on Table 1, to encrypt data to be written to disk. Each changelog message is encrypted using K_c and authenticated individually, and each stores the metadata epoch counter inside the message. Each changelog message is additionally chained to the previous message by using the associated data from AES-GCM encryption as additional data for the next encryption call. The binary checkpoint file contains all of the metadata in a single encrypted and authenticated file along with the metadata epoch. Each changelog message is encrypted with 128-bit AES-GCM encryption, and the checkpoint file is encrypted with 128-bit AES-CTR encryption and authenticated with HMAC-256.

With this approach, the file system's state can be restored into an enclave with the same signature. Given the same enclave signature and sealed data, the server enclave can derive

K_c and K_m. This allows the service to close down and be restored in the event of a failure. Because all file and directory metadata is sealed as one checkpoint file, this approach does not leave the file system vulnerable to rollback attacks at a file or directory granularity, but only at the epoch counter granularity. A secure hash of file chunks, H^i_{int}, is stored alongside file metadata in the checkpoint file. An attacker cannot restore the file system to an older metadata epoch to gain access to a newer version of file data. This is because checks on an older value H^i_{int} would fail during decryption. The details of this check are explained in detail in Section 3.3.3. Attempting to rollback a specific change in a changelog would be detected with integrity checks since each changelog message is chained together when encrypted. Similarly, modifying the binary, which stores metadata, is detected through authenticated decryption. Overall, an adversary may leverage rollback attacks to restore the file system to an older snapshot, but the integrity of the file system and the access set during that snapshot do not allow a user to access new data with an older metadata epoch.

3.2 SECURE COMMUNICATION

Authorized and authenticated access is common in modern DFSes, however typical DFS usage assumes that clients and servers faithfully provide correct access control when credentials are provided and protocols are followed. In an untrusted setting, the service provider can collude to circumvent authorized access to the master server by manipulating network traffic or altering the system that hosts the master. Additonally, a client's system can attempt to use invalid credentials or exploit an authentication process on an untrusted platform to avoid access control.

MooseGuard fortifies existing authorized and authenticated access by establishing secure channels of communication between clients and the master server to protect the privacy and integrity of operations on the file system. Given the stricter threat model, clients and the master server must verify that the other will perform file system operations securely and as intended. In addition to ensuring that operations that require permission are evaluated in a TEE, as discussed in Sections 3.1 and 3.3, MooseGuard leverages the attestation feature

of Intel SGX between client enclaves and the master enclave to ensure that enclaves will faithfully execute permission checks. Furthermore, after establishing a connection to the master server, clients must only be able to access what they are granted permission to, as opposed to any arbitrary client issuing arbitrary operations on the master. To accomplish this, MooseGuard requires a public/private key-pair used by a user to authorize an action.

3.2.1 Attestation

The SGX Remote Attestation Protocol [17] establishes a secure communication channel between master and client enclaves. This process is an extended Sigma protocol to conduct a Diffie-Hellman Key Exchange. Remote Attestation establishes several key facts: the identity of the client and master enclaves, the integrity of those enclaves, the authenticity of the platform each enclave is running on, and finally a shared key that can be used to derive keys for secure end-to-end encryption between enclaves. Both endpoints for this secure channel are established within enclaves, which makes this channel suitable for the master and client to share keys for file encryption and for the master to associate this channel instance with a set of UNIX file permissions. Thus, establishing a secure channel is a key requirement for MooseGuard to provide authenticated and authorized access to a DFS in untrusted environments.

Both client enclaves and the master enclave must prove their authenticity to each other. As a result, MooseGuard performs the attestation process twice. The process begins with a typical ECDH key exchange which establishes the shared key, K_r (Table 1), between the master and client enclaves. From this step onward, all communication between enclaves is encrypted with K_r using AES-GCM 128-bit encryption. Next, the master enclave challenges the client enclave to provide a quote. This quote is received by the master enclave and sent to the Intel Attestation Service (IAS). After the quote is validated, the master server has established the identity and integrity of the client enclave and its SGX-enabled CPU. The client has now established a trustworthy place to perform isolated execution, but must still prove to the master that it is authorized to share file encryption keys (FEKs) and accept file operation requests. Finally, the client enclave issues the same challenge and the master

replies with a quote. The quote is validated, and the client enclave can trust the identity and authenticity of the master server it is connected to.

3.2.2 User Authentication and Permissions

After the attestation process, both enclaves have established a trusted platform to operate on, but have not established that the client who initiated the connection has permissions to operate on the file system. In a typical DFS, after a client connection has been established, the client will provide credentials to authorize itself on the file system. MooseGuard extends this step by requiring clients to provide a credential that can be verified within the master enclave. This process restricts access to only client enclaves that are granted permission to use the file system, restricts each client connection to act as only a single user, and allows the master server to identify what permissions each connected client has within its isolated execution environment.

MooseGuard ensures that only authorized users access the file system by requiring each user to authenticate their access with a public/private key pair, $K_u|K_u^{-1}$ from Table 1. The master server has a whitelist of all authorized users' public keys, and a mapping of UNIX user IDs to public keys $f : u \rightarrow K_u$. These keys ensure that only authorized users can access the file system, and allows the master server to validate the authorization of a user within its enclave. Each client knows their private key K_u^{-1}, and the ID of the user they wish to authenticate as. The client enclave signs the user ID with their private key and sends this signature to the master enclave. The master enclave validates the signature and associates the validated UNIX ID with the current secure channel. The master enclave maps all file system operations on this secure channel to the established UNIX ID. This process further restricts each client's connection to operate as a single, authenticated user. If a client attempts to authenticate as a different user with their private key by signing a different UNIX ID, the master will detect this violation during the verification of the signature.

The public/private key pair is the mechanism that allows the master enclave to authenticate users. Associating each key pair with a UNIX ID provides convenience for clients to interact natively with the FUSE mount. Further, it allows the MooseFS master server's

implementation of access control, performed in the master enclave, to remain unmodified. Finally, after the client and client enclave authorize themselves with the master server, the secure channel can be used to issue file system operations, and the client enclave can be used to access encrypted file data from chunk servers.

3.3 CLIENTS

DFSes implement functionality on clients to combine the task of locating data with the master and accessing data on chunk servers into a coherent client-side interface for users of the file system. Whether the implementation of the client runs in privileged or unprivileged modes on the CPU, a client can evade access control for file data by directly accessing chunk servers instead of using the client interface. A DFS can address this vulnerability by encrypting chunks and only sharing keys with clients after an access check on the master server. However, even with a master server running in a TEE this incurs expensive re-encryption costs when access is revoked. The DFS could instead perform all encryption directly on the master and send decrypted file data to clients through a secure channel, but this approach would severely bottleneck the scalability of read and write throughput at the master server. Finally, a DFS could enforce access control on chunk servers or through a third party, but the management of keys and other metadata would limit the scalability and expand the attack surface for the DFS.

MooseGuard's solution to this issue utilizes a TEE on each client, where file encryption keys can be shared, utilized, and revoked in a trusted, isolated environment on the client. This approach enables the master to distribute work to the client for both enforcing access to files and encrypting files for I/O. Distributing the work of encryption and decryption to the client allows users to operate on data in parallel and simplifies the chunk server requirements. This allows the service provider to easily scale-out storage with no major changes to the chunk server implementation. This section describes the organization of the client enclave, additional file encryption metadata MooseGuard introduces, and the distributed access control procedure that MooseGuard proposes.

3.3.1 Client Enclave

The client functionality of a DFS can combine intricate caching and optimization, but at its core, it must communicate with the master to retrieve chunk metadata and communicate with chunk servers for file I/O. MooseGuard proposes a simple client enclave inserted in between client functionality and network interactions with remote servers in the DFS. This allows the client enclave to intercept master messages to send and receive additional encryption metadata, and intercept file I/O and perform encryption in-line between the client and chunk servers.

With its limited responsibilities, MooseGuard's implementation of a client enclave on MooseFS is much simpler than the master enclave. The client enclave interface includes ECALLs to connect and attest to the master, send messages to the master over the secure channel, and encrypt and decrypt blocks of data from a chunk server. Like the master enclave, the client enclave utilizes the SGX SDK's generated boundary checks but does not implement any further prevention of Iago attacks.

3.3.2 File Encryption Scheme

Data servers in DFSes distribute files or chunks over multiple storage servers for redundancy, concurrency, and heterogeneous support. Specificially, MooseFS divides file data into 64 MB chunks as a unit of replication between chunk servers, and further divides each chunk into a collection of 64 KB blocks as a common unit for accessing data and computing checksums for data integrity. Clients can obtain read and write locks at a chunk granularity for each file. MooseGuard's implementation on MooseFS chooses to encrypt data and compute secure checksums on the same block-alignments to balance the trade-off of storing additional associated encryption data with block-aligned data access that MooseFS generally optimizes for. MooseGuard's general approach of assigning a file encryption key (FEK) to each file and storing a hashed version of the file generalizes its approach, making it adaptable to different divisions of data for each DFS implementation.

MooseGuard implemented on MooseFS extends the file inode structure on the master to store K_f (Table 1), a 128-bit FEK, and H^i_{int}, an array of 256-bit chunk hashes for each chunk

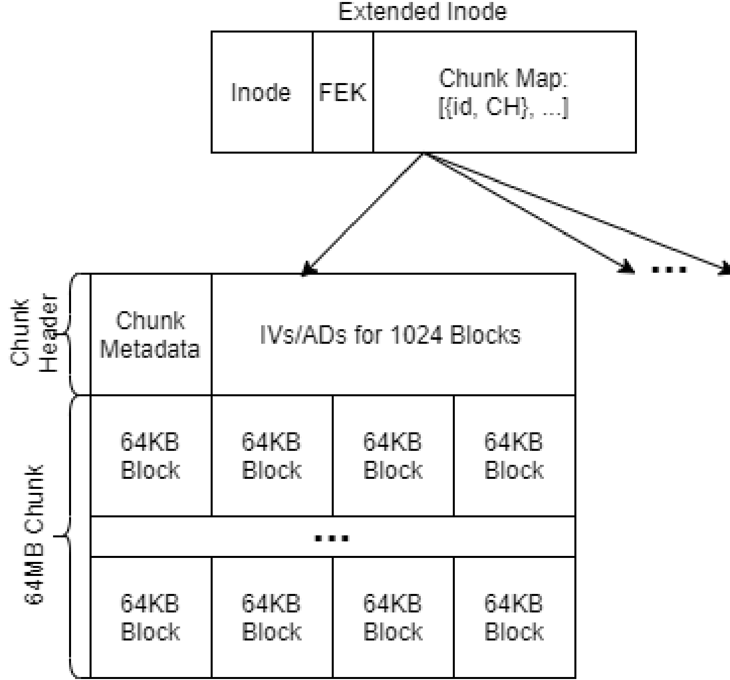

Figure 3: MooseGuard File Extensions

i, stored alongside the chunk map for each file. Each block within a chunk is encrypted with K_f and a random IV using AES-GCM 128-bit encryption producing authenticated data AD. For each block j in chunk i, the pair of associated encryption data IV and AD is stored together as the value $T^i_j := IV|AD$ in the extended header for a 64 MB chunk. On updates, the chunk header is hashed using SHA-256 hashing to determine the new version of chunk i. Formally, $H^i := SHA256(T^i_1|...|T^i_{1024})$. During access control, as described in Section 3.3.3, the external and internal versions of the H^i are compared. Figure 3 depicts how encryption metadata is organized in the inodes and chunk metadata.

3.3.2.1 Chunk Server Modifications The chunk server does not require an enclave; however, the implementation of MooseGuard on MooseFS required slight modifications to the MooseFS chunk server daemon to support the storage of additional encryption metadata for each chunk. The first change was to store encryption metadata alongside other chunk

metadata. The second change was to modify the messages between clients and chunk servers to read and write encryption metadata, and between chunk servers to replicate the encryption metadata. The messages were extended so clients could store each chunk's T_j^i values on the chunk server. These changes were trivial for MooseFS and should not affect a service provider's ability to deploy the chunk server daemon.

3.3.3 File Access Control and Revocation

MooseGuard clients access file data in two phases. The first phase is a common behavior for both read and write access. The client first requests access to chunk i on file f from the master. After the master validates the client's access, it returns all necessary metadata required to access the chunk. This metadata includes K_f, H_{int}^i, and the location of the chunk. The client enclave strips K_f and H_{int}^i and caches them inside the enclave for the pending chunk server access, while the location is returned to the client.

Figure 4 depicts how a client reads a chunk in both phases. During phase two of a chunk read, the client requests block j and all block headers $T_1^i, ..., T_{1024}^i$ from the chunk server. Then the client enclave computes $H_{ext}^i := SHA256(T_1^i|...|T_{1024}^i)$ and checks that $H_{ext}^i = H_{int}^i$ to confirm that the version of the chunk the client is decrypting matches the version of the chunk they have access to decrypt. Finally, block j is decrypted with K_f from phase one and T_j^i and returned to the client.

Similarly, Figure 5 depicts how a client writes a chunk in both phases. Write access checks and encryption use the same concept as reads of checking the internal and external hash versions, but need to commit a new hash to finalize the write. During phase two of a chunk write, the client requests all block headers $T_1^i, ..., T_{1024}^i$ from the chunk server (this step is optimized through caching). Then the client enclave computes $H_{ext}^i := SHA256(T_1^i|...|T_{1024}^i)$ and checks that $H_{ext}^i = H_{int}^i$ to confirm that the version of the chunk the client is about to encrypt matches the version of the chunk they have access to encrypt. Next, the client enclave encrypts block j using K_f from phase one and T_j^i and updates the internal hash $H_{int}^i := SHA256(T_1^i|...|T_j^i|...|T_{1024}^i)$. The client enclave writes the new encrypted block and header to the chunk server and finally commits the newly written version of the chunk to

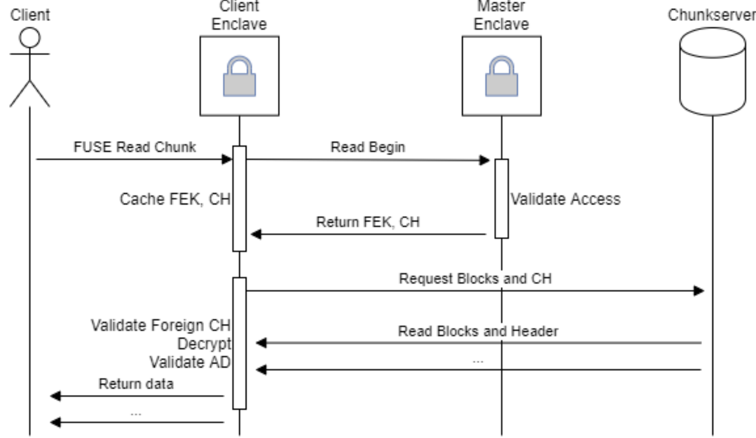

Figure 4: MooseGuard Read Access Control

the master enclave by returning the new value of H^i_{int}. When the master commits the new version of the chunk, the write access is confirmed in the master enclave one more time.

In the most basic implementation, the overhead MooseGuard imposes over MooseFS is reading the entire chunk header, validating H^i through a SHA-256 hash, and encrypting or decrypting with AES-GCM 128-bit. The IVs and ADs for each chunk require 28 KB of space and each block is 64 KB. Thus, the worst case overhead for any read or write is the cost an additional 28 KB read from the chunk server, combined with the costs of hashing 28 KB and encrypting 64 KB. This cost can be mitigated by combining encryption or decryption of multiple blocks in one block access. Additionally, the MooseGuard implementation on MooseFS utilizes extra enclave memory to cache as many encryption headers $T^i_1, ..., T^i_{1024}$ as possible, preventing multiple 28 KB header reads from the chunk server on each chunk operation. When a client frequently works on the latest version of this file, the cost of reading the whole header is heavily mitigated. The impact of this overhead and the benefits of caching are shown in the file I/O access pattern benchmark in the evaluation at Section 5.1.2.

MooseGuard can revoke access to file metadata immediately as the master server handles a request. However, when access to a file's data is changed through UNIX file permissions, access to file data from each client is revoked lazily as the client enclave detects a change in

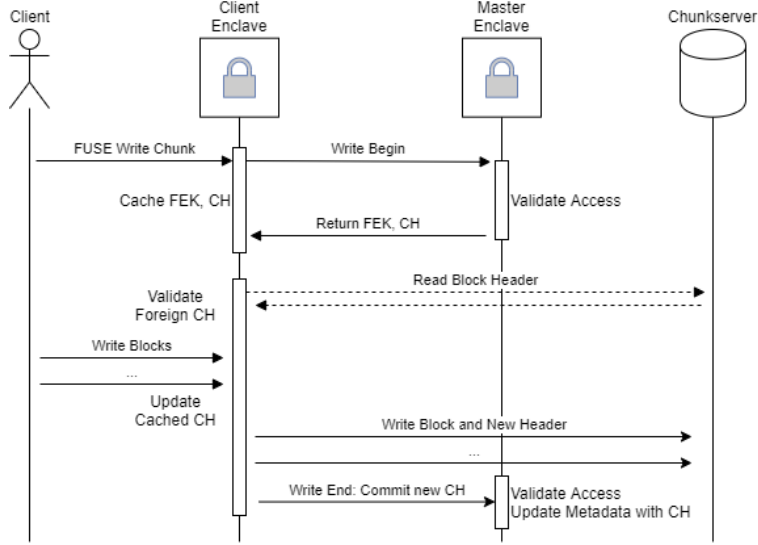

Figure 5: MooseGuard Write Access Control

the chunk hash $H^i_{int}! = H^i_{ext}$. This approach could allow a service provider to collude with a client that had access revoked, allowing the client to continue accessing an older version of a file. MooseGuard views this as an acceptable vulnerability as users typically anticipate that data shared with another user can be copied to a separate location.

4.0 SECURITY ANALYSIS

MooseGuard's goal is to enable a secure distributed file system that can be deployed in untrusted environments. MooseGuard considers a powerful adversary amongst a server architecture deployed on an untrusted platform with client devices that attempt to gain unauthorized access to data. In the context of the deployment and threat models defined in Chapter 2, this chapter discusses MooseGuard's design and how it accomplishes its security goals. This chapter considers the confidentiality and integrity guarantees on user data, how MooseGuard prevents unauthorized access, and finally how the state of the file system is kept consistent in the presence of a strong adversary.

4.1 CONFIDENTIALITY AND INTEGRITY

MooseGuard identifies several categories of information in a DFS that are sensitive and must be protected. The sensitive information that MooseGuard protects is file data, file metadata, and messages between the master and client enclaves. Table 1 summarizes the three categories of data that MooseGuard protects, and the keys and encryption methods used for each data type. All information that MooseGuard protects is encrypted using AEAD encryption algorithms to ensure confidentiality and integrity. Except for a user's key (K_u), all keys originate from isolated execution within a TEE. All encryption operations are executed only within a validated enclave, and key sharing between enclaves (K_f and H^i_{int}) is done between authenticated and authorized enclaves over a secure channel.

4.2 AUTHORIZED ACCESS

MooseGuard ensures that only authorized users of the file system can read or change the state of the file system. MooseGuard identifies three areas where authorization must

be confirmed before operating on the file system. The remainder of this section details the steps MooseGuard takes to ensure unauthorized access to the file system is prevented.

4.2.1 Authorizing User Access

Users of MooseGuard must follow several steps to gain authorized access to the file system. Client enclaves first use a Diffie-Hellman key exchange to establish K_r and a secure communication channel. The master and client enclaves then prove the authenticity and integrity of the other's enclave using remote attestation with the IAS. The client enclave then binds it's UNIX ID and groups to the secure channel by signing its ID with K_u^{-1}, provided by the user, and sends this to the master. The master enclave verifies this access with K_u, which is hardcoded into the enclave. At this point, the user has established a secure channel to communicate, proven the authenticity and integrity of its enclave, and proven they are authorized to access the system.

4.2.2 Securing the Enclave Boundary

The master server enclave mitigates unauthorized manipulation of the file system by only making ECALLs available during phases of operation. Because the master enclave executes in a single-threaded environment, incorrect ordering of OCALLs can simply be checked with the SGX SDKs generated boundary code. During the initialization phase, ECALLs are expected to be called in an exact sequence that initializes the enclave, restores the sealed metadata snapshot (with K_m), and replays changelogs after the metadata snapshot version (with K_c). During the main phase of operation, the only entrances to the enclave are through ECALLs for new or existing client communication. New clients follow an exact sequence of attestation ECALLs, and all existing client messages are sent through the same ECALL. These well-defined steps allow the enclave to decrypt the message in each ECALL and verify that the ECALL was called in the correct sequence. OCALLs to commit changelog entries, access the metadata cache, and make chunk changes are all executed serially through OCALLs during the initial ECALL. Finally, the shutdown phase is restricted to the single ECALL to initiate sealing metadata. All other operations to store metadata are completed through

serial OCALLs until the process completes. Additional hardening of enclave boundaries to prevent further attacks, such as Iago attacks, could be pursued by implementing a solution similar to the work by Shinde et al. [50, 51]; this is left for future work.

4.2.3 File Access and Revocation

The client enclave interface is considerably simpler. The client enclave interface contains sequenced ECALLs to attest and authorize with the master and contains ECALLs used to send metadata operations or encrypt or decrypt file data. ECALLs for metadata messages and file access may be called in any order, but MooseGuard will only return decrypted data or commit newly encrypted data when the client enclave follows the protocol detailed in Section 3.3.3 to gain authorized access to files. Access revocation is checked at the master enclave when a user requests access to a file and is enforced at the client enclave by checking that the version of the chunk requested matches the version they have access to ($H^i_{int} = H^i_{ext}$).

4.3 CONSISTENCY

A DFS provides basic consistency guarantees by ensuring that state changes in the file system are committed to disk and that concurrent client operations are properly serialized. In light of the threat model in Section 2.1, MooseGuard further protects the consistency of the file system from an adversary that may attempt to create an inconsistent state in the file system. Specifically, MooseGuard ensures fork consistency on access revocation. Formally, when a user's access of a file is revoked at version v_f, the user may be able to read copies of the file at versions v for all versions $v < v_f$, but the user will not be able to read copies of the file at versions v' for all versions $v' \geq v_f$.

Another area of MooseGuard that is vulnerable to attacks on consistency is metadata stored outside of the master enclave. Confidentiality and integrity for metadata is addressed in Section 4.1. However, attackers may try to reorder or remove pieces of persistent metadata. An attacker may try to swap encrypted objects in the metadata cache. MooseGuard detects

such an attack by embedding the ID of the cache entry in the metadata object before sealing it. When unsealing the entry, MooseGuard asserts that the expected ID matches the unsealed ID. MooseGuard also considers attacks on the ordering of persistent metadata. When MooseGuard unseals and replays the changelog to initialize the file system, an attacker may reorder encrypted lines in the changelog. MooseGuard detects this attack during unsealing, as each changelog entry is chained together. As an example, given changelog entry C_i and associated data AD_{i-1}, MooseGuard will perform the encryption $ENC(C_i|AD_{i-1}) \rightarrow E_i, AD_i$, which produces encrypted data E_i and associated data AD_i. During unsealing, swapped changelogs will break this chaining and authenticated decryption will detect the violation.

Finally, an attacker could choose to omit changelog entries at the end of the chain or perform a wholesale swap of a compacted metadata backup. These rollback attacks put MooseGuard into an old state, but not an inconsistent state. Work to detect rollback attacks on the persistent metadata files could be accomplished by embedding counters using a hardware counter with the SGX SDK or by using an approach similar to ROTE [36]; this is currently unexplored in this paper and left for future improvement.

5.0 EVALUATION

MooseGuard's evaluation considers the performance impact that it imposes on a distributed file system. The performance impact is measured by comparing the latency overhead of file system operations on a stock MooseFS file system versus a MooseGuard file system. The costs of encryption, additional verification in chunk access, and master performance in a TEE are examined in the context of file I/O benchmarks, metadata benchmarks, and various applications.

The prototype implementation of MooseGuard was built with Intel's SGX SDK v2.7.1, SGX driver v2.6 and MooseFS v3.0.99. The TCB of the compiled server enclave was 2.9 MB, and the client enclave was 1.4 MB. Using *SLOCCount* [12] to count lines of code, MooseGuard's implementation added 20,907 new lines, of which 10,002 were lines generated by the SGX SDK enclave interfaces.

MooseGuard was evaluated on a private gigabit network on machines running Ubuntu 18.04. MooseGuard was configured with one master server (Intel Core i7-8700 CPU @ 3.20GHz, 16GB RAM), 1-4 clients, and 4 chunk servers with approximately 1 TB of storage on each server. Chunk replication was set to a factor of 2 and 4 chunk servers were used in each test. All results shown were averaged over 10 samples.

5.1 FILE I/O BENCHMARKS

The intent of the file I/O benchmarks are to analyze MooseGuard's impact on file I/O throughput. MooseGuard implements file encryption on a 64 KB block size and accesses the master to check file permissions and manage keys on every chunk access. Because of these requirements of MooseGuard, the evaluation of file I/O considers several elements. The evaluation looks at performance over varying file sizes, access patterns, and the effect of multiple clients concurrently operating on files. File I/O performance was measured by using *IOR* [6]. IOR uses POSIX C APIs to access files and uses MPI to coordinate reading and writing to files via clients hosted on multiple remote computers.

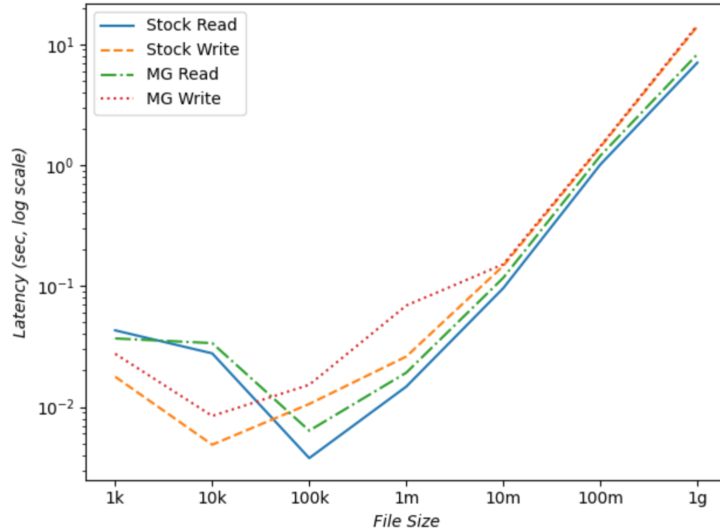

Figure 6: File I/O Overheads over Varying File Sizes

5.1.1 File Sizes

MooseGuard's client enclave must encrypt all file data and synchronize encryption metadata with the master enclave. The file size evaluation isolates the cost of encrypting file data in MooseGuard. This test is run through a single client that reads and writes files sizes of varying orders of magnitude. All files were read and written sequentially. The Linux page cache was cleared at the beginning of each read iteration to force all reads through MooseFS to ensure all data is read and written through our file system. MooseFS was configured to never cache file data locally on clients. IOR was configured to read and write blocks of data 1 KB at a time, flushing writes before closing each file.

Figure 6 shows that as file size varies, the overhead that MooseGuard introduces remains constant. Over all file sizes, the overhead for both read and write operations is constant. MooseFS optimizes for block accesses of 64 KB, so small file accesses can be completed in one operation for MooseFS. Operating on larger file sizes took more operations and reflected an average latency with less variance than smaller file operations. These results indicate that MooseGuard's client-side encryption introduces a constant overhead per read or write.

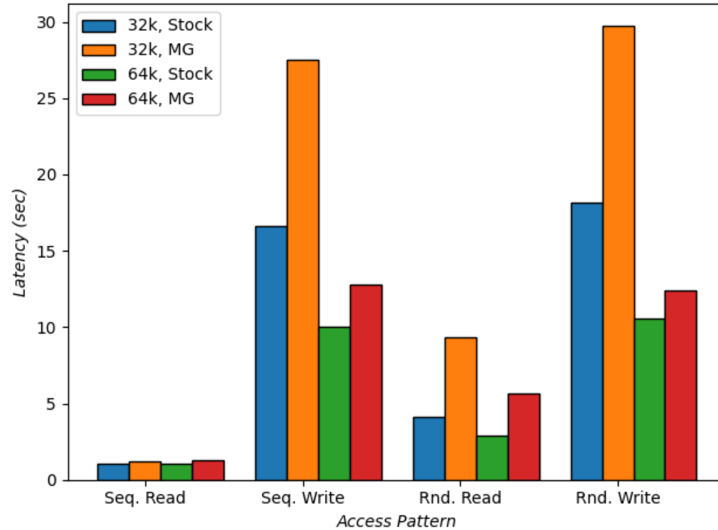

Figure 7: File I/O Performance over Access Patterns

5.1.2 Access Patterns

MooseGuard's implementation chooses to encrypt data in 64 KB blocks. By encrypting blocks of data at this size, MooseGuard can balance between performance in accessing smaller units of data and the storage overhead of encryption metadata per-block. The access pattern evaluation compares the file I/O performance overhead of MooseGuard in workloads that vary in how much data is accessed and in the order that data is accessed. This test accesses a 128 MB file through a single client, flushing writes after every block access and clearing the Linux page cache before each read.

Figure 7 shows the runtime of each operation in each access mode. Both file systems improve in performance with 64 KB block size, the default unit of storage for MooseFS. In sequential reads, both file systems on both block sizes can leverage MooseFS's read-ahead and write-coalescing capabilities, and the overheads are minimal as a result. All other access patterns show that MooseGuard incurs nearly a 2x overhead in 32 KB operations. Each 32 KB operation requires MooseGuard to fetch 64 KB of data for encryption, while stock MooseFS can access smaller block slices. In 64 KB blocks, this is not a factor, so the overhead

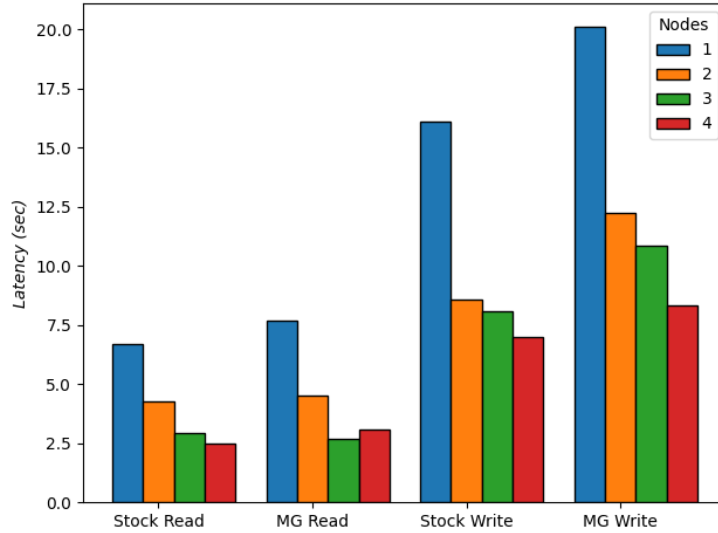

Figure 8: File I/O Performance over Client Scale

is reduced substantially to approximately the cost of encryption alone. Overall, the results of this test show that this approach suits most access patterns with reasonable overhead, and the performance for specific access patterns can be enhanced by properly adjusting the block size for encryption.

5.1.3 Client Scaling

The client scaling benchmark evaluates the impact of MooseGuard on file I/O operations during concurrent use of the file system by many clients. MooseGuard's approach of managing FEKs with the master enclave but distributing keys to client enclaves for encryption aims to enable concurrent use of the file system by limiting where the master enclave can be a bottleneck. This test distributes the work of writing and reading 1 GB of data over multiple clients. The work is split into 1024 tasks of reads and writes of 1 MB of data each. IOR was configured to ensure that each client never read data that it also wrote. Writes were flushed at the close of each file, and the Linux page cache was flushed before reads in the one node case.

Figure 8 shows the overall latency of the operation decreases as the number of clients increases for both file systems. Reads are faster than writes in both cases because writes require replication of data while reads can access any chunk server. The performance does not decrease linearly, which may be a result of our chunk servers becoming overwhelmed. MooseFS typically uses up to 250 threads on clients and on chunk servers to handle file I/O. During testing, we found that 2 clients can saturate 4 chunk servers with our hardware setup. To analyze the impact of client scale, the number of worker threads for clients was reduced to 125 threads for this test.

5.2 METADATA BENCHMARKS

This section of the evaluation focuses on the overhead of metadata operations, which are impacted by MooseGuard's enclaves differently than file I/O. All metadata operations occur only between clients and the master; the client enclave's file encryption methods are not used. While there are many types of metadata operations, this section of the evaluation focuses on three core metadata operations at scale: creating, querying, and deleting a node. Other typical metadata operations are evaluated in Section 5.3.

This section of the evaluation uses *mdtest* [6], another HPC file system benchmarking tool from the LLNL. Like IOR, mdtest uses POSIX C APIs to perform file system operations and utilizes MPI to coordinate running these operations across many clients on remote computers. Unlike the file I/O, metadata operations for MooseFS are not as heavily parallelized on individual clients. To operate on large directory structures, the metadata benchmarks always distribute the work across four clients unless otherwise noted. Additionally, MooseGuard caches recently used directories and attributes for up to 1 second. For these tests, those caches were disabled.

5.2.1 Directory Tree Sizes

MooseGuard's master enclave protects the integrity of metadata operations but aims to keep the implementation of metadata management the same where possible. The first

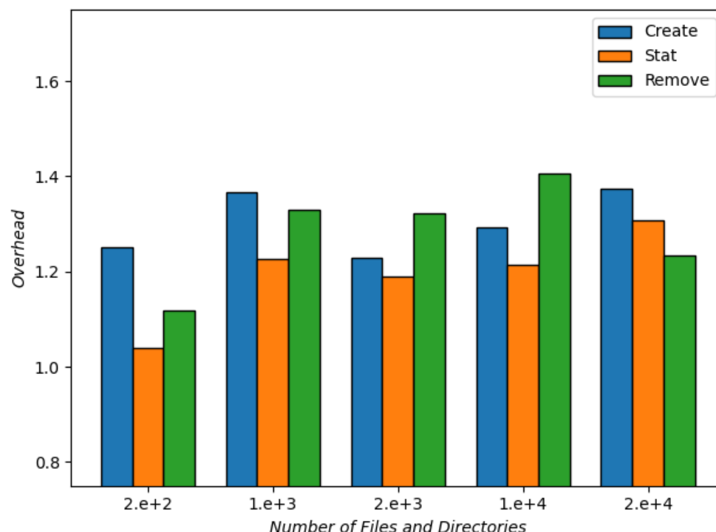

Figure 9: Metadata Performance over Directory Tree Sizes

metadata benchmark examines the overhead in latency that MooseGuard introduces while operating on directory structures of various sizes. This test creates a deep directory structure, then creates, stats, and deletes an equal number of files and directories at each level of the structure. The total number of files and directories in an iteration of the test ranges from 200 to 20,000 files and directories. All operations were run sequentially using mdtest's default behavior.

Figure 9 shows that the overhead of performing these operations ranges between 1x to 1.5x, varying between directory tree sizes. Though the runtime required to perform metadata operations increases for larger directory trees, all operations on deep trees are performed in-memory on the master, which prevents linear growth in runtime of deep tree accesses. As a result, the overhead does not show an increase with various directory tree sizes. Stat operations generally had less overhead than create or remove operations, suggesting that the extra memory allocation operations required by the MooseGuard enclave can impact performance. Overall, this benchmark suggests that MooseGuard does not prohibit large directory trees in a secure file system.

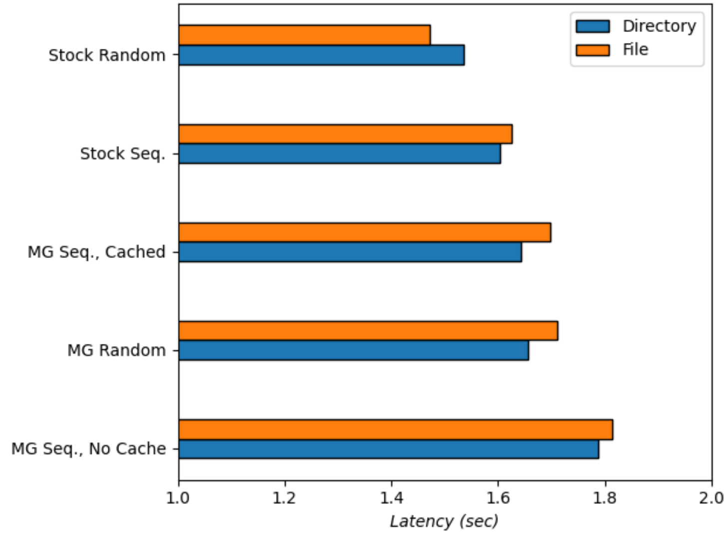

Figure 10: Metadata Cache Impact on Performance

5.2.2 Master Metadata Cache

MooseGuard's implementation of the client enclave has limited memory, so it uses a cache to store elements of the directory tree in enclave memory. This imposes encryption costs for nodes evicted from the cache and has an extra cost in general to manage memory for the cached items. This benchmark compares the overhead of a stat operation from delays caused by the cache. The test also shows delays caused by entering and leaving enclaves, which is required for any operation but occurs more frequently with uncached data. This test creates 5000 files and directories in one flat root directory, then runs the stat operation on all entries. In the MooseGuard variants of the test, the size of the cache is modified to hold all entries (MG Seq., Cached), no entries (MG Seq., No Cache), or half of the entries (MG Random).

Figure 10 shows the runtime of the stat operations in each environment. The cost of calling stat on a directory node or a file node is approximately the same in any environment. The overhead of sequentially calling stat for MooseGuard with all cached entries versus stock MooseFS is 1.034, which is the approximate overhead for encryption costs in the client to

master secure channel and the cost to enter and exit the client and master enclaves. The overhead of always missing cached data vs. always hitting cached data in MooseGuard is 1.078, which is the approximate overhead the cache itself imposes. The worst-case overhead between always missing the cache in MooseGuard and stock is an overhead of 1.115, roughly the combination of both previous overheads. Randomly hitting the cache shows an overhead between both scenarios. These results suggest that a TEE with limited memory can support a secure file system with MooseGuard's approach by leveraging caching cold metadata outside the enclave.

5.2.3 Client Scaling

MooseGuard avoids bottlenecks in file I/O by distributing work to clients, but metadata operations are coordinated by a master server. The client scale benchmark for metadata examines the impact of MooseGuard on the file system's ability to scale concurrent metadata operations to multiple clients. This test distributes the work of creating, stating, and deleting 20,000 files and directories over 1 to 4 clients. All 20,000 nodes are organized in one flat root directory.

Figure 11 shows the trend of latency to run the operations as the number of concurrent clients increases. In general, the latency to perform all operations decreases in both file systems as more clients work together. The performance starts to flatten between 3 and 4 clients when the single-threaded master server starts to near its capacity to handle concurrent requests. The overhead of MooseGuard decreases slightly in all types of operations as the number of clients scales up since the amount of overhead in enclave context switches and secure channel encryption cost becomes parallelized over multiple clients. Overall, MooseGuard imposes only a constant overhead per-client and does not impact the scaling of concurrent metadata operations by client.

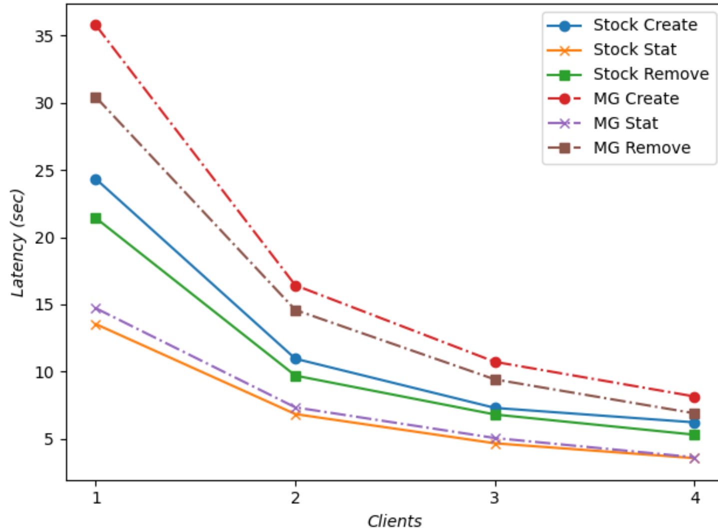

Figure 11: Metadata Performance over Client Scale

5.3 APPLICATION BENCHMARKS

Analyzing the performance of applications utilizing a distributed file system shows the holistic cost that MooseGuard incurs. This section of the evaluation considers two common use cases for a DFS: typical UNIX operations on a shared file system and Spark [60], a distributed scientific application. These applications utilize file I/O-bound operations such as sequential, random, and parallel access. They also incorporate metadata-bound operations during the same runtime, showing how MooseGuard comprehensively affects performance for end-users.

5.3.1 UNIX Applications

To analyze UNIX application performance, we use the same benchmark used by NeXUS [27], another paper that utilizes TEEs in a secure file system. This benchmark considers common UNIX applications that end-users of a shared file system may run. The applications used are:

Table 2: UNIX Directory Trees

Workload	**Files**	**Total Size**
large-file-small-dir	32	3.2 GB
medium-file-medium-dir	256	2.5 GB
small-file-large-dir	1024	10 MB

- *tar_x* - Extract a gzip-compressed tar file
- *grep* - Recursively search for a word in all files
- *tar_c* - Create a tar file
- *cp* - Recursively copy a directory
- *ls* - Recursively list and stat a directory
- *rm* - Recursively remove a directory

Each iteration of the test runs all applications in the order shown against 3 different directory trees described in Table 2. Between each application, the Linux page cache is flushed. To observe MooseGuard's impact on metadata performance, both client metadata caches for MooseFS are also disabled.

Figure 12 shows the results of this benchmark for stock MooseFS and MooseGuard. These results show that the holistic overhead of MooseGuard is smaller than the overhead found for similar microbenchmarks in Sections 5.1 and 5.2. In the small-file-large- dir workload, the file I/O-bound applications saw a higher overhead than other workloads. In this workload, the size of each file was 10 KB, where file I/O operations for MooseGuard were performed on 64 KB of encrypted data. These results are consistent with the results of Section 5.1.2 and could be improved by optimizing the block size of the file system for the use case. Overall, as the overheads for file I/O and metadata operations mix with applications using the data, MooseGuard's performance overhead decreases and suggests that MooseGuard is well suited for these applications.

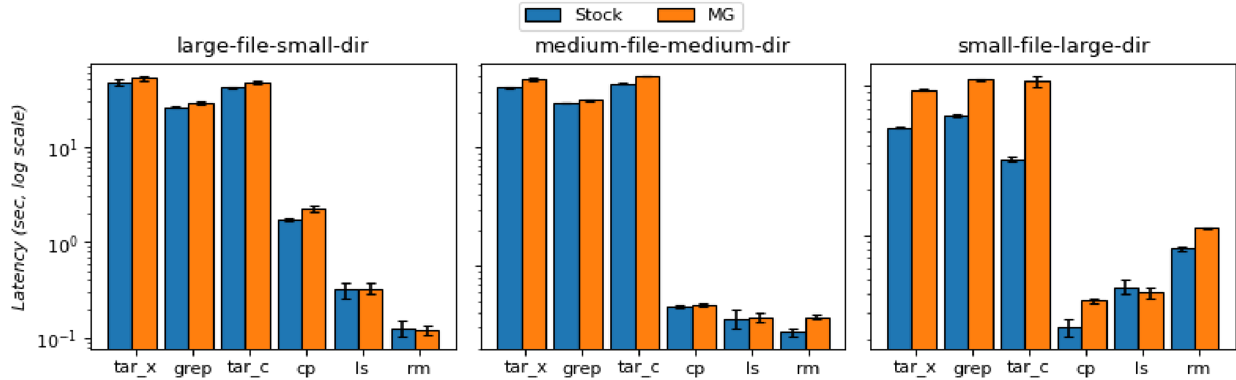

Figure 12: UNIX Application Performance

5.3.2 Spark

Scientific distributed computing is another common use case for distributed file systems. HDFS, a distributed file system commonly used in conjunction with Spark, and shares a similar architecture to MooseFS as they both have a name server (master server) and data servers. To analyze the performance impact of MooseGuard for this use case, we ran a distributed application with the Spark framework and MooseGuard as the storage service. Spark and MooseFS were deployed on the same systems, with the MooseFS master and Spark Cluster Manager on the master server and an instance of a MooseFS client, a MooseFS chunk server, and Spark Worker on each of the four other servers. The application for Spark was the KMeans data generator and clustering applications. The amount of data generated was 10,000,000 rows and 24 columns for a total of 2 GB of overall data. The data was divided into 4 RDD partitions, approximately 458 MB per file, so the work could be parallelized to all worker nodes.

Table 3 summarizes the results of running KMeans data generation and clustering with stock MooseFS or MooseGuard as the backing file system. During data generation, the Spark workload is I/O bound. The file I/O operations during this phase are mainly sequential writes of large chunks of data, for which MooseGuard performs optimally with a 1.022 overhead. For the data generation task as a whole, MooseGuard's overhead is minimal. In the clustering

Table 3: Spark Performance

Operation	Stock (sec)	MooseGuard (sec)	Overhead
Generate I/O	16.567	16.933	1.022
Generate Total	19.668	19.989	1.016
Clustering I/O	0.626	0.873	1.396
Clustering Total	31.328	33.036	1.055

phase, the Spark workload is CPU-bound and can work around the delays of file I/O to retrieve data. Despite spending less overall time reading data during the clustering phase, MooseGuard's impact is seen with a higher overhead of 1.396 during I/O operations. The performance in this case is similar to the random read overheads in Section 5.1.2 and suggests that the overhead here may be attributable to Spark accessing RDD files in a non-sequential manner. The overhead for the clustering phase in total is smaller at 1.055, as the CPU-bound portion of this phase dominates the file I/O overhead. Overall, MooseGuard shows that it is capable of protecting a distributed file system in distributed computing applications with minimal overheads to the application.

5.4 EVALUATION TAKEAWAYS

MooseGuard showed that it only imposes a constant overhead for file I/O and metadata options when scaling upward with size or outward with clients. Though performance can vary on specific block sizes, that attribute can be tuned for specific uses. MooseGuard improves performance in scaled metadata and concurrent use cases compared to NeXUS by reducing the overhead from a greater than a linear factor to a constant factor. In applications with the presence of caching and CPU-bound operations, MooseGuard's impact can be further minimized. Overall, the results of this evaluation show that the trade-off of using separate TEEs to secure metadata and data can greatly reduce typical overheads seen in secure file systems, allowing a DFS to return to its normal scaled performance.

6.0 CONCLUSION

Shared storage services provide great utility, but have traditionally required users to sacrifice the privacy of their data to share it effectively. By employing the use of TEEs, research has taken steps forward in empowering users so they can keep their data private and share it without concern that a service provider may leak or abuse their data.

MooseGuard provides another step in this direction. The design of MooseGuard is adaptable to many DFSes and uses functionality that is common in TEEs that are becoming more widely available and adding enhanced features. MooseGuard's approach provides a blueprint for securing a distributed file system for shared infrastructure deployments by isolating file system policy decisions on the master server, performing stricter verification of users, and encrypting user data. The prototype for MooseGuard demonstrates the feasibility of building such a system and the results of the evaluation show that with only a constant overhead, MooseGuard can further enable users to use shared infrastructure to share their data at scale with confidence in the privacy of their data.

This paper additionally identifies areas of work for future exploration. Within MooseGuard, enhancements can be made to further protect the integrity of persistent metadata by detecting rollback attacks. MooseGuard also currently only supports fork-consistency of revoked access to files. This level of consistency can be made stricter by adapting MooseGuard to consult the master server on finished reads or to optimize the client enclave to complete reads within a single ECALL. Finally, MooseGuard's encryption process can be optimized to reduce the overhead of random access on smaller blocks by leveraging client- side caching or tuning the block and chunk sizes.

Broadly, MooseGuard's approach leverages client- and server- side TEEs to perform both DRM-style key sharing and access revocation, and secured remote computation within the same application. Distributed applications which require a centralized service for synchronization, while delegating work to client devices can follow this approach to achieve stronger security. These improvements to MooseGuard and broader concepts are left for future work to provide even stronger scaling and sharing and future enhancements in secure computation.

Bibliography

[1] ARM TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone. Accessed: 06/28/2020.

[2] AWS HIPPA Usage. https://d1.awsstatic.com/whitepapers/compliance/AWS_HIPAA_Compliance_Whitepaper.pdf. Accessed: 06/25/2020.

[3] Equifax data breach. https://www.csoonline.com/article/3444488/equifax-data-breach-faq-what-happened-who-was-affected-what-was-the-impact.html. Accessed: 06/29/2020.

[4] Google terms of service. https://policies.google.com/terms. Accessed: 06/29/2020.

[5] Intel sgx sdk. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html. Accessed: 06/28/2020.

[6] LANL HPC IO Benchmark. https://github.com/hpc/ior. Accessed: 06/16/2020.

[7] Lustre. http://lustre.org/. Accessed: 06/28/2020.

[8] Microsoft terms of service. https://www.microsoft.com/en-us/servicesagreement. Accessed: 06/29/2020.

[9] MooseFS. https://moosefs.com/. Accessed: 06/27/2020.

[10] Openafs. http://www.openafs.org/. Accessed: 06/28/20.

[11] SGX on Microsoft Azure. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Accessed: 06/28/2020.

[12] Sloccount. https://dwheeler.com/sloccount/. Accessed: 06/28/2020.

[13] US Government JEDI Contract. `https://www.geekwire.com/2019/jedi-explaining-10b-military-cloud-contract-microsoft-just-won-amazon/`. Accessed: 06/25/2020.

[14] Verizon's data breach report. `https://enterprise.verizon.com/resources/executivebriefs/2019-dbir-executive-brief.pdf`. Accessed: 06/29/2020.

[15] Voter cloud data exposed. `https://www.wired.com/story/voter-records-exposed-database/`. Accessed: 06/29/2020.

[16] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[17] Ittai Anati, Shay Gueron, Simon Paul Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. 2013.

[18] Sergei Arnautov, Andrey Brito, Pascal Felber, Christof Fetzer, Franz Gregor, Robert Krahn, Wojciech Ozga, André Martin, Valerio Schiavoni, Fábio Silva, Marcus Tenorio, and Nikolaus Thummel. Pubsub-sgx: Exploiting trusted execution environments for privacy-preserving publish/subscribe systems. *CoRR*, abs/1902.09848, 2019.

[19] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: secure linux containers with intel SGX. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 689–703. USENIX Association, 2016.

[20] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David M. Eyers, and Peter R. Pietzuch. Libseal: revealing service integrity violations using trusted execution. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 24:1–24:15. ACM, 2018.

[21] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, 2015.

[22] Matt Blaze. A cryptographic file system for UNIX. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993*, pages 9–16. ACM, 1993.

[23] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. *CoRR*, abs/1701.00981, 2017.

[24] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 991–1008. USENIX Association, 2018.

[25] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, jun 2019.

[26] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 178–194. IEEE Computer Society, 2018.

[27] Judicael Briand Djoko, Jack Lange, and Adam J. Lee. Nexus: Practical and secure access control on untrusted storage platforms using client-side SGX. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 401–413. IEEE, 2019.

[28] Dennis Fetterly, Maya Haridasan, Michael Isard, and Swaminathan Sundararaman. Tidyfs: A simple and small distributed file system. In Jason Nieh and Carl A. Waldspurger, editors, *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*. USENIX Association, 2011.

[29] Kevin Fu. Group sharing and random access in cryptographic storage file systems. 1999.

[30] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43. ACM, 2003.

[31] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[32] William C. Garrison III, Adam Shull, Steven Myers, and Adam J. Lee. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 819–838. IEEE Computer Society, 2016.

[33] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In Jeff Chase, editor, *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. USENIX, 2003.

[34] Seong Min Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. Enhancing security and privacy of tor's ecosystem by using trusted execution environments. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 145–161. USENIX Association, 2017.

[35] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–17, 2018.

[36] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David M. Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: rollback protection for trusted execution. *IACR Cryptol. ePrint Arch.*, 2017:48, 2017.

[37] David Mazières and Dennis E. Shasha. Building secure file systems out of byzantine storage. In Aleta Ricciardi, editor, *Proceedings of the Twenty-First Annual ACM*

Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002, pages 108–117. ACM, 2002.

[38] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos V. Rozas. Intel® software guard extensions (intel® SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*, pages 10:1–10:9. ACM, 2016.

[39] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP 13, New York, NY, USA, 2013. Association for Computing Machinery.

[40] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (Cat. No.01CH37210)*. IEEE.

[41] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 227–240. USENIX Association, 2018.

[42] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless OS services for SGX enclaves. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 238–253. ACM, 2017.

[43] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W. Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, 2020.

[44] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 264–278. IEEE Computer Society, 2018.

[45] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. *Design and Implementation of the Sun Network Filesystem*, page 379390. Artech House, Inc., USA, 1988.

[46] Mahadev Satyanarayanan. A survey of distributed file systems. *Annual Review of Computer Science*, 4(1):73–104, 1990.

[47] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54. IEEE Computer Society, 2015.

[48] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1211–1228. ACM, 2017.

[49] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: eradicating controlled-channel attacks against enclave programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[50] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with SGX enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[51] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. Besfs: Mechanized proof of an iago-safe filesystem for enclaves. *CoRR*, abs/1807.00477, 2018.

[52] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[53] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. VAULT: reducing paging overheads in SGX with efficient integrity verification structures. In Xipeng

Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 665–678. ACM, 2018.

[54] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. Sgxkernel: A library operating system optimized for intel SGX. In *Proceedings of the Computing Frontiers Conference, CF'17, Siena, Italy, May 15-17, 2017*, pages 35–44. ACM, 2017.

[55] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 645–658. USENIX Association, 2017.

[56] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: performance of user-space file systems. In Geoff Kuenning and Carl A. Waldspurger, editors, *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 59–72. USENIX Association, 2017.

[57] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. *CoRR*, abs/1705.07289, 2017.

[58] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 307–320. USENIX Association, 2006.

[59] Jiongyu Yu, Weigang Wu, and Huaguan Li. DMooseFS: Design and implementation of distributed files system with distributed metadata server. In *2012 IEEE Asia Pacific Cloud Computing Congress (APCloudCC)*. IEEE, nov 2012.

[60] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):5665, October 2016.