NEXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-side SGX

Judicael B. Djoko
University of Pittsburgh
jbriand@cs.pitt.edu

Jack Lange
University of Pittsburgh
jacklange@cs.pitt.edu

Adam J. Lee
University of Pittsburgh
adamlee@cs.pitt.edu

***Abstract*—With the rising popularity of file-sharing services such as Google Drive and Dropbox in the workflows of individuals and corporations alike, the protection of client-outsourced data from unauthorized access or tampering remains a major security concern. Existing cryptographic solutions to this problem typically require server-side support, involve non-trivial key management on the part of users, and suffer from severe re-encryption penalties upon access revocations. This combination of performance overheads and management burdens makes this class of solutions undesirable in situations where performant, platform-agnostic, dynamic sharing of user content is required.**

We present NEXUS, a stackable filesystem that leverages trusted hardware to provide confidentiality and integrity for user files stored on untrusted platforms. NEXUS is explicitly designed to balance security, portability, and performance: it supports dynamic sharing of protected volumes on any platform exposing a file access API *without* requiring server-side support, enables the use of fine-grained access control policies to allow for selective sharing, and avoids the key revocation and file re-encryption overheads associated with other cryptographic approaches to access control. This combination of features is made possible by the use of a client-side Intel SGX enclave that is used to protect and share NEXUS volumes, ensuring that cryptographic keys never leave enclave memory and obviating the need to re-encrypt files upon revocation of access rights. We implemented a NEXUS prototype that runs on top of the AFS filesystem and show that it incurs $\times 2$ overhead for a variety of common file and database operations.

I. INTRODUCTION

Cloud-based data storage and sharing services are among the most widely used platforms on the Internet [1, 2]. By relying on centralized, cloud-based infrastructures, users gain access to vast storage capacities, seamless multi-device access to files, and point-and-click data sharing at very low cost. However, this flexibility brings with it risks to the confidentiality and integrity of users' data. These services suffer from frequent data breaches [3, 4, 5], and oftentimes their Terms of Service grant providers full licensing rights, allowing them to store, modify, and distribute user data as they choose [6, 7, 8]. As more users leverage these types of services to manage sensitive information, addressing these types of security issues is crucial [9].

To this end, our objective is to provide a *practical* solution for securing user files on unmodified, distributed file-sharing services that: (1) allows users to maintain complete control over how their data can be accessed, modified, and disseminated by others users and the storage platform itself; (2) does not alter the user's typical file access workflow; and (3) is performant enough to satisfy the demands of the user's typical file access workloads. Our goal is to ensure the confidentiality and integrity of user data in the face of untrusted administrators, data breaches, and other unauthorized disclosures *without* requiring server-side support.

Several schemes have been proposed to provide rich access control semantics for untrusted storage platforms using cryptography. Unfortunately, when implemented in a distributed setting [10, 11, 12, 13, 14], purely cryptographic approaches incur very high overheads on user revocation. This results from the observation that when decrypting files on a client machine, the encryption key is inevitably exposed to the client application and can be cached by the user. Therefore, revoking a user's access to a file requires re-encrypting the file under a new key. As shown by Garrison et al. [15], even under modest policy updates, the resulting overhead can be significant as the incurred cryptographic and network costs are proportional to both the total size of the affected files *and* the degree to which they are shared. Alternative approaches make use of trusted hardware to provide strong security features like access pattern obliviousness (via ORAM) and policy-based access control [16, 17, 18]. However, these approaches require server-side hardware support, which limits their availability for users of personal cloud storage services.

To address this need, we present NEXUS, a privacy preserving file system that provides cryptographically secure data storage and sharing on top of existing network-based storage services. NEXUS is novel in that it leverages the Intel SGX [19] extensions to provide efficient access control and policy management, in a manner that is not possible using a software-based cryptographic approach. NEXUS allows users to add strong access controls to existing unmodified and untrusted distributed data storage services to protect the confidentiality and integrity of their data from both unauthorized users and the storage service itself, while enabling sharing with authorized users. Data is protected through client-side cryptographic operations implemented inside an SGX enclave. NEXUS embeds user-specified access control policies into files' cryptographically protected metadata, which are enforced by the enclave at access time. Therefore, unlike existing purely cryptographic approaches to access control, revocations are efficient and do not require the bulk re-encryption of file contents. Instead, the policies embedded in the smaller attached metadata are simply updated and re-uploaded to the server.

NEXUS is user-centric, transparent and requires no server-side changes. It is implemented as a protection layer between users/applications and an underlying file system, and leverages hardware security features (SGX) in order to securely intercept and transform file system operations. Its two primary components are (1) a secure enclave that provides cryptographic and policy protections, and (2) a file system interface layer that maps the generic file system API exported by the enclave to the actual underlying storage platform. This approach allows NEXUS to present a standard hierarchical file system view while supporting a broad range of underlying storage services such as remote file systems and distributed object stores.

In this paper, we make the following contributions:

(1) We propose a novel client-side architecture that allows mutually-trusting users to securely share files hosted on untrusted cloud infrastructure. This architecture allows for efficient volume sharing and access control policy changes. By performing all access controls and cryptographic operations inside the enclave, NEXUS allows for seamless and secure key distribution, minimal user key management, and efficient user revocation.
(2) NEXUS instantiates a distributed access control platform using trusted hardware. An SGX enclave serves as a trusted reference monitor that executes independently on each client machine rather than centrally on the (untrusted) server. This enables efficient cryptographic access control without requiring server-side support for deployment.
(3) We propose a cryptographic protocol that uses SGX remote attestation to enable secure file sharing between users. Communication is completely in-band as it uses files on the underlying shared filesystem to exchange data, and does not require both users to be online simultaneously.
(4) We implemented a NEXUS prototype that runs on top of OpenAFS [20]. The prototype runs as a userspace daemon, and allows unmodified user applications to access files in a protected folder whilst cryptographically enforcing user-specified Access Control Lists (ACLs). Our performance evaluation shows that, compared to OpenAFS, NEXUS incurs modest overheads on metadata-intensive operations.

The paper is organized as follows: Section II provides an account of our protection model. Section III describes the assumptions and threats of our system. In Section IV, we describe the NEXUS system and Section V provides a prototype implementation. Respectively, Sections VI and VII describe the security and performance evaluations of the NEXUS AFS prototype. We review related work in Section VIII, and Section IX concludes the paper.

II. Background and Protection Model

A. Intel Software Guard Extensions (SGX)

Intel SGX is a set of processor extensions that provide secure execution environments, called enclaves, on modern x86 based platforms. These extensions enable clients to both measure and verify the code running within an enclave, while also providing very strong isolation guarantees. When activated, enclaves execute in user space and are protected from inspection or modification by other processes, as well as the underlying OS. At the system level, enclaves exist as a special CPU hardware context that ensures data privacy by encrypting the contents of enclave-managed memory as it leaves the CPU. Secure execution is achieved by placing both the code and data contents needed for a given computation inside the protected memory region, thus ensuring both confidentiality as well as integrity of the execution.

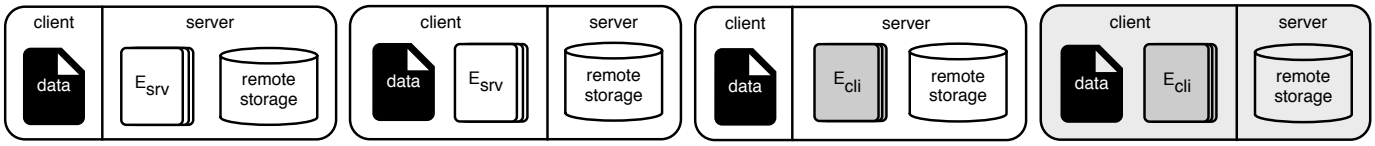
1) Isolated Execution: An enclave is set to be an isolated region within a userspace application. When creating the enclave, the CPU performs a secure hash *measurement* of its contents as they are copied into a protected region of physical memory called the *Enclave Page Cache* (EPC). The EPC is inaccessible from untrusted code, including privileged software and hardware devices. To run the enclave, the application invokes a special SGX instruction (`EENTER`) to jump inside a predefined entrypoint of the enclave code. While executing, the enclave code performs arbitrary computations, and can read and write to untrusted memory. This allows an enclave to efficiently exchange data with the host application. To prevent potential leakage of sensitive data, the enclave code is not allowed to call untrusted functions. Enclave memory is only accessible from the enclave code, and is defined as a linear range in the host application's address space. The OS is responsible for translating enclave virtual addresses into their corresponding EPC page. On enclave destruction, the CPU clears its EPC pages and in so, prevents the recovery of sensitive information.

2) Sealed Storage: To persist sensitive data across executions, SGX allows enclaves to derive a *sealing key* that can be used to encrypt and seal data before copying it to untrusted memory. The sealing key is only accessible within enclave memory, and is unique to the enclave and the particular CPU within which the enclave is executing. In NEXUS, we leverage SGX sealing facilities to protect long-term encryption keys.

3) Remote Attestation: Remote attestation allows a *challenger* to validate the identity of a remote enclave, and its underlying platform. In SGX, this process relies on an Intel-provisioned Quoting Enclave, which uses a unique asymmetric key embedded in the CPU to generate *quotes* [21]. A quote is a signature of the target enclave's identity along with additional data generated inside the enclave. This allows the challenger to verify the quote using an Intel-provided public certificate. To support file sharing, we use remote attestation for securely transferring encryption keys across valid NEXUS enclaves.

B. SGX Design Space

With its strong security primitives, SGX presents a wide range of options on how to deploy enclaves within a cloud setting. Depending upon the security needs of the distributed application, different considerations have to be taken into account. Thus, we define the design space of enclave deployment along the following dimensions: (1) Enclave provenance — whether the enclave is owned by the client or the service provider and; (2) Enclave location — whether the enclave is running on the client or the server.



(a) service-enclave-on-server (b) service-enclave-on-client (c) client-enclave-on-server (d) client-enclave-on-client

Fig. 1: Different architectures for enabling SGX security in a client-server environment. Each architecture shows a different combination of enclave location and enclave provenance.

Figure 1 shows all the combinations in this design space. The *service-enclave-on-server* (e.g., PESOS [18]) and *service-enclave-on-client* (e.g., EndBox [22]) collectively describe Digital Rights Management (DRM) scenarios: access to data is controlled by the service provider's enclave. On the other hand, the *client-enclave-on-server* (e.g., Troxy [23]) denotes a scenario in which the client provisions enclaves on the server to achieve secure remote computation. However, running the client enclave on the server has drawbacks. First, the server must be equipped with SGX hardware which, at the time of this writing was only offered by one major cloud provider (Microsoft Azure [24]). Second, a substantial amount of server-side software may need to be retrofitted for SGX support. Depending upon the system's complexity, this may not be an easy task as changes could range from modifying the client – server communication protocol, to including untrusted software components inside the enclave [16, 25, 26, 18, 27].

C. Our Approach

NEXUS combines the client-side encryption model used by existing cryptographic filesystems with SGX security guarantees. Shown in Figure 1d, NEXUS adopts the *client-enclave-on-client* architecture to encrypt data on the local machine before uploading the resulting ciphertext onto the server. The idea is to have every client run NEXUS locally and then leverage the aforementioned SGX features to form a secure key distribution system. On the local machine, all cryptographic data protection is performed within an enclave (Isolated Execution), and keys are persisted to disk using SGX sealing facilities. Then, before sharing keys with authorized users, we use the remote attestation feature to ensure the exchange occurs between valid NEXUS enclaves running on genuine SGX processors. As a result, encryption keys are never leaked to untrusted memory, and as such, kept under the complete control of the NEXUS enclave.

In this paper, we explore a deployment model that targets applications generating sensitive data exclusively at the client, but rely on a remote server as a storage provider. In the case of distributed filesystems, the user's file contents are opaque to server, which we assume can access, modify, and disseminate any file that it stores [6, 7, 8]. To protect each file, we encrypt its contents, and attach cryptographically-protected metadata containing access control policy along with key material that can only be accessed using a valid NEXUS enclave. The benefits are two-fold: (1) our solution can be easily deployed without any out-of-band setup, as key distribution is implicitly provided by the file synchronization service, and (2) users maintain control over their data and decide on who is authorized to access its contents. As SGX-enabled machines come to reach more end-users, we expect this client-side approach to user-centric access control to become increasingly mainstream.

III. Problem Description

We consider a typical cloud storage service, in which the service's *users* download and run a *client-side program* to access the remote storage platform. Data is stored on remote cloud based systems that are under the control of the *service provider*. In addition to ensuring the persistence and availability of data, the cloud service typically provides authentication and access control, but in a way that requires the user to trust the service implicitly. Users interact with their data via their local file system API, thus allowing arbitrary applications on their systems to access and operate on the remotely stored data. Beyond normal file system access, many services also provide auxiliary sharing capabilities with other users of the service. Within this context, we aim to provide users with additional protections against unauthorized disclosure or modification of their files without hindering their ability to share these files with other *authorized* users.

A. Scope, Assumptions, and Threat Model

Security objective. *Unless granted explicit access by the owner, a file's contents must be inaccessible to unauthorized entities and tamper-evident*. In this case, unauthorized entities may include other users of the storage service, entities monitoring communication between the user and the storage service, and the storage service provider itself. We are concerned solely with the protection of user-created content: i.e., the confidentiality and integrity of the contents of files, file names, and directory names; and the integrity of the directory structure itself. The protection of other file attributes (e.g, file size or access patterns) is considered an orthogonal problem that can be addressed using other techniques.

Threat model. We consider an attacker who has complete control of the server (including the OS or hypervisor), and can thus access or alter any files stored on the server. The attacker may also tamper with, delete, reorder, or replay all network packets exchanged between the server and the client. Since our primary concern is protecting the confidentiality and integrity of file content, we do not consider availability attacks (e.g., denial-of-service). Since authorized users ultimately gain

access to decrypted file contents, we do not consider client-side malware that may maliciously leak files that have been decrypted by authorized users.

We assume that each user has access to an SGX-enabled CPU running a commodity OS. The NEXUS enclave is assumed to be correctly implemented, and free of any security-relevant vulnerabilities. In addition, we assume the enclave attestation and memory protection features of the SGX hardware function properly: i.e., once the enclave's identity is established, enclave-provisioned secrets are not accessible from untrusted code.

B. Design Goals

In designing NEXUS, we chose to strike a balance between security and ease of use with the following aims:

1) **Practicality.** After an initial setup, the user should be able to access their data using their typical workflow. NEXUS should be simple and impose minimal key management on the user. Also, throughout its execution, the overheads imposed by NEXUS should not significantly degrade the system's performance.
2) **Portability.** All changes required to run NEXUS must occur on the client. NEXUS's design should be flexible in a way that allows users to either store data locally, or on a remote storage platform. This implies no server-side coordination, and the use of the underlying filesystem as the NEXUS metadata store.

This approach closely follows the direction taken by existing cryptographic filesystems (e.g., [28, 10, 12]). Our goal is to offer similar protections with superior key management, efficient revocation, and no server participation. It is important to note that NEXUS is not a full-blown standalone filesystem, but is designed as a security layer over an existing host filesystem. To minimize our TCB, it is essential for the trusted portion of NEXUS to be small, and its interface minimal. Our solution must be transparent and adaptable, such that users can access their protected files without having to update their applications, and integrating with various filesystems should be possible with moderate effort. Moreover, the distribution of generated metadata should not require the deployment of additional services, instead our solution should allow the user to use their available storage for both file data and metadata.

Access Control. NEXUS should adopt a standard discretionary approach to access control in which object owners can specify custom access control policies to selectively dictate file access permissions. NEXUS must support standard file access rights such as read and write. Administrative control over a file's access permission should remain with the owner, and enforcement must occur *without* the cooperation of the (untrusted) storage service provider. To achieve this, NEXUS must internalize access control information as part of the filesystem state, and enforce access control policies inside the NEXUS TCB. In addition, NEXUS must ensure that the unencrypted data contents never leave the TCB unless the access control policy allows it.

IV. NEXUS

In order to meet the objectives outlined in Section III, we have designed NEXUS to allow users transparent security protections on existing file storage services. The design of NEXUS is based on the concept of a protected *volume*, which is presented to the user as a normal file system directory. In order to ensure that the structure and contents of each volume are only visible to authorized users, NEXUS internally manages the volume layout in addition to the user's data. The entirety of the volume state is stored as a collection of *data* and *metadata* objects that are managed by NEXUS, and tracked using universally unique identifiers (UUIDs). Each object is stored as a normal data file on the underlying storage service using its UUID as the filename. In effect, NEXUS implements a virtual file system on top of the underlying target file system. Figure 2 shows a high level NEXUS configuration.

Accessing data from a NEXUS volume consists of the user issuing file system requests that are intercepted by NEXUS and translated into a series of metadata and data operations that are dispatched to the underlying storage service as file operations from the NEXUS enclave. The data retrieved from the underlying storage service is then routed to the enclave where it is decrypted and either returned as part of the original request (data) or used to drive further enclave operations (metadata). Because NEXUS internally implements a standard hierarchical file system in its metadata structures, this allows NEXUS to be portable across a wide range of storage service architectures. Both data and metadata are stored as self-contained objects in NEXUS, thus allowing them to be stored on a wide variety of potential storage services (including object-based storage services).

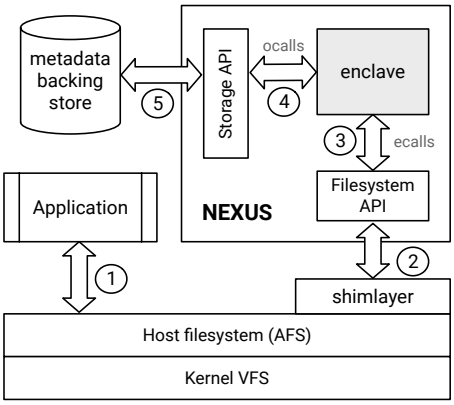


Fig. 2: NEXUS architecture.

The linchpin of data confidentiality and integrity in NEXUS is an enclave-generated symmetric encryption key called the volume *rootkey*. This rootkey allows a NEXUS enclave to decrypt the volume state and all other encryption keys used to individually encrypt volume objects. Since it is created by the enclave, NEXUS is able to access the rootkey only when running inside a restricted enclave environment. When the NEXUS enclave is not running, the rootkey is sealed using SGX (Section II-A2) and stored on the local filesystem in an

Filesystem Call	Description
Directory Operations	
`nexus_fs_touch()`	Creates a new file/directory
`nexus_fs_remove()`	Deletes file/directory
`nexus_fs_lookup()`	Finds a file by name
`nexus_fs_filldir()`	Lists directory contents
`nexus_fs_symlink()`	Creates a symlink
`nexus_fs_hardlink()`	Creates a hardlink
`nexus_fs_rename()`	Moves a file
File Operations	
`nexus_fs_encrypt()`	Encrypts a file contents
`nexus_fs_decrypt()`	Decrypts a file contents

TABLE I: NEXUS Filesystem API.

encrypted state that can only be decrypted from inside the NEXUS enclave running on the same machine that sealed it. This approach requires that all decryption operations be performed within the NEXUS enclave (Section II-A1), which is also able to apply the file's access control policy before exposing the data to the user (Section IV-C). In this way, even should a user obtain a copy of the enclave and a valid rootkey for a volume, they would still be unable to access the protected data unless they also possessed a valid identity that had been granted access permissions. With this approach, NEXUS is able to provide sharing capabilities (Section IV-C) using SGX remote attestation (Section II-A3), where the *rootkey* may be accessible to multiple users while still maintaining per-file access controls that limit access to a subset of those users.

A. Filesystem Interface

Users access data in NEXUS using standard file system interfaces, which are translated into a set of generic API calls implemented by the NEXUS enclave. This API is shown in Table I, and consists of 9 operations — 7 directory operations and 2 file operations. Each operation takes as a target a file or directory stored inside the NEXUS volume. Each target is represented as a metadata object stored by NEXUS, as well as a potential data object in the case of file operations. As part of each operation, NEXUS traverses the volume's directory hierarchy decrypting and performing access control checks at each layer. This has the side effect of turning single operations in multiple potential operations on the underlying storage service. While this does introduce additional overheads, we show that these are acceptable for most use cases. In addition, NEXUS contains a number of performance optimizations to limit the impact of these overheads (Section V).

Next, we describe how the enclave manages and protects metadata in order to provide a virtual hierarchical filesystem.

1) Metadata Structures: Figure 3 gives a high level overview of the structure of a NEXUS volume. NEXUS stores the file system structure internally using a set of encrypted metadata files alongside the encrypted data files using obfuscated names. These obfuscated names consist of a globally unique 16-byte ID (`UUID`), that is tracked by the metadata structures. The UUIDs are randomly generated within the enclave at metadata creation, and are universally unique across all machines. The unencrypted view of the file system (seen on the right side of Figure 3) is only accessible by decrypting the metadata inside the NEXUS enclave. The metadata files not only store the file system layout, but also contain the cryptographic keys and access control policies needed to ensure that the file system data and metadata are confidential and tamper-evident.

The metadata structures implement a standard hierarchical file system approach. Each NEXUS file system is specified by a *supernode* (corresponding to a *superblock* in a normal file system). The file system hierarchy is then implemented using a set of *dirnodes* (corresponding to *dentries*) and *filenodes* (corresponding to *inodes*).

- **Supernode:** A supernode defines the context of a single NEXUS volume. The supernode structure stores the `UUID` of the file system's root directory along with the identity (public key) of the file system's owner. It also contains a list of other user identities that have been granted access to the file system by the owner. These identities consist of a user name along with an associated public key that is used for authentication. The owner of a file system is immutable, however, the owner has the ability to add and remove authorized users at any time.
- **Dirnode:** Dirnodes represent directories in a NEXUS file system. Each dirnode contains a list of directory contents consisting of a mapping between file/directory names and their `UUID`s. It is important to note that each `UUID` in a dirnode only references other metadata files, and never directly references an actual data file. In NEXUS, because access control is maintained at the directory level, the dirnode also stores the directory's access control policy.
- **Filenode:** Filenodes store the metadata that is necessary to access the data files stored in NEXUS. Specifically, the filenode stores the cryptographic keys needed to encrypt/decrypt the file contents. To support efficient random file access, NEXUS divides each data file into a set of fixed-sized chunks, each of which is encrypted with an independent cryptographic context. These contexts are stored as an array in the filenode structure, along with the UUID corresponding to the actual data file.

2) Metadata Encryption: The general layout of a metadata structure consists of three components, each of which has a different degree of cryptographic protection.

(1) A preamble that is used to store non-sensitive information (e.g., `UUID`, size). This section is integrity-protected.
(2) A cryptographic context containing the information used to secure the metadata contents. It has a 128-bit encryption key, an initialization vector and an authentication tag. This section is integrity-protected, and the encryption key is stored in keywrapped form to protect its confidentiality.
(3) A section where the metadata's sensitive information is stored. This section is encrypted and integrity-protected using the unique metadata key stored in (2).

Encryption of the metadata file occurs on every update, and is performed within the enclave in two stages. After generating a fresh encryption key and IV inside the cryptographic context from (2), the first stage of encryption is performed using the AES-GCM cipher with metadata section (3) as input, and the

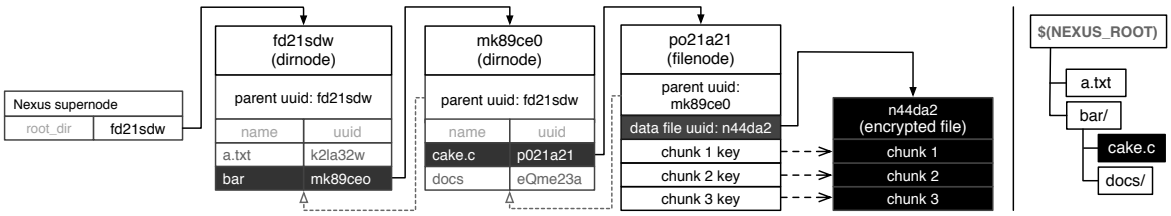


Fig. 3: **Authenticated user view.** Directory traversal by NEXUS to present the plain contents of the user's data files.

other two sections as additional authenticated material. This operation outputs an authentication tag, which is copied into (2). The second stage involves a keywrapping scheme that uses the volume's rootkey to encrypt the freshly generated key. We use the GCM-SIV [29] AEAD construction, and refer the reader for a deeper discussion on keywrappring.

Essentially, the metadata is protected using its cryptographic context which, in turn, is protected using the rootkey. This simplifies key management, as every encryption key is embedded within its corresponding metadata. Therefore, to access a volume, a user only needs to store the volume's sealed rootkey, which can only be unsealed within the NEXUS enclave.

3) Metadata Traversal: Because a volume is just a normal directory, if directly accessed by the user, the files will be encrypted and bear obfuscated names. Therefore, to expose this protected state, — i.e., plain content and human-readable filenames — NEXUS has to translate each local filesystem request into the corresponding metadata. Figure 3 shows the metadata traversal to access **`bar/cake.c`**. We abstracted all metadata operations into a simple primary-key only interface that provides access to metadata using a `UUID`. To begin, the root dirnode is loaded using the root directory's `UUID` stored in the supernode. Then, for each path component, the current dirnode's directory list is used to lookup the `UUID` of the next dirnode. As each metadata object is read into trusted memory, the enclave uses the volume rootkey to decrypt and verify its contents. Before performing the lookup, the enclave also checks the `parent_uuid` field of the loaded dirnode matches the `UUID` of its parent. This guards against *file swapping* attacks [10], and helps provide integrity protection for the filesystem structure. If the verification or lookup operation fails, the metadata traversal terminates. Otherwise, the final metadata object is returned.

B. Authentication and User Sharing

To access a NEXUS volume, a user must first be authenticated to a NEXUS enclave in order to be granted access to the file system's rootkey. While the rootkey allows a user to launch a NEXUS instance for a particular volume, it does not automatically grant access to the data stored in that volume. For that, the NEXUS enclave performs a second step ensuring that the identity used to authenticate into the volume is authorized by the access control policies stored in the file's metadata.

Cryptographic Notation. We denote a (public, private) keypair as $\{pk, sk\}$, and use PKGEN() to indicate public keypair generation. $\text{SIGN}(sk, m)$ represents a signature over m using sk, and $\text{VERIFY}(pk, s)$ indicates the verification of a signature s using pk. $\text{ENC}/\text{DEC}(k, m)$ denotes symmetric key encryption/decryption.

In NEXUS, identity is established using public-private key pairs, where each authorized user's public key is stored inside the supernode metadata file. Each identity has an associated user ID that is used in the access control policies maintained by the dirnodes. To authenticate into a NEXUS volume the user performs the following challenge-response protocol:

1) The user requests to authenticate by making a call into the NEXUS enclave with their public key (pk_u) and the sealed volume rootkey as arguments.
2) Inside the enclave, the rootkey is unsealed. Then, a random 128-bit nonce is generated, and returned to the calling user.
3) The user then uses their private key to create a signature over the encrypted supernode structure of the volume and the enclave nonce. This signature and the encrypted supernode are then passed to the enclave.

$$m = \text{SIGN}(sk_u, nonce \,|\, \text{ENC}(rootkey,\ supernode))$$

4) Inside the enclave, the volume rootkey is used to decrypt and verify the supernode. After finding the user's entry inside the supernode, the enclave then validates the signature with the user's public key.
5) On success, the user's ID is cached inside the enclave.

This protocol establishes that (i) the user as the owner of the public key stored (via signature verification), (ii) the user has been granted access to the volume (via the presence of their public key in the supernode), and (iii) the supernode itself has not been modified (via metadata protection). Once access is granted, the volume is mounted, and becomes available.

1) Granting access to other users: Sharing data with NEXUS is complicated by the fact that SGX generates a unique sealing key on each machine. This means that a sealed rootkey cannot simply be passed between enclaves when a new user is granted permission to access a volume, or when an authorized user accesses a volume using a new machine. At the same time, the rootkey cannot be encrypted with a key available outside of the enclave context (e.g., a user's public key) without compromising the volume's security. To overcome this challenge, we incorporated a key exchange protocol that allows a volume's rootkey to be distributed to remote NEXUS instances while ensuring that it will only be accessible from within a NEXUS enclave. This protocol relies on an Elliptical Curve Diffie Hellmann (ECDH) key exchange combined with

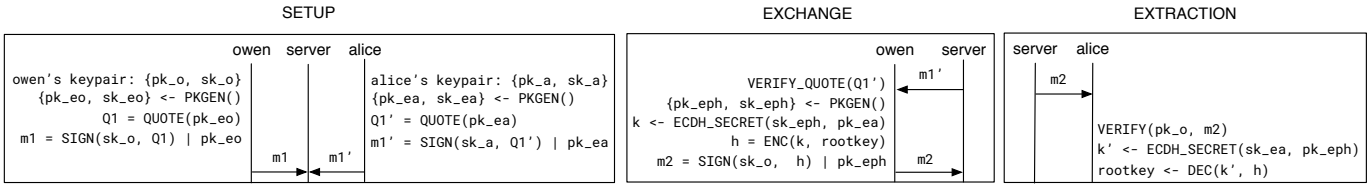


Fig. 4: Key Exchange protocol diagram for Owen sharing his NEXUS volume rootkey with Alice.

enclave attestation features available in SGX. All messages are communicated in-band using the underlying storage service to exchange data between endpoints.

Consider the case where a NEXUS volume owner, Owen, wishes to grant access to his volume to another user, Alice. The end result of the protocol will be that Alice has a locally sealed version of the rootkey for Owen's NEXUS volume, and Alice's public key will be present in list of users stored inside the volume's supernode. We assume that Alice's public key is available to Owen via some external mechanism (e.g., as in SSH). The endpoints of the protocol are actual NEXUS enclaves, and the execution is as follows (Figure 4):

1) *Setup:* As part of the initialization process of a NEXUS volume, an ECDH keypair (pk_e, sk_e) is generated inside the NEXUS enclave. The private key is only ever accessible inside the enclave, and is encrypted with the enclave sealing key before being stored persistently. To export the public key, the user generates an enclave quote supplying the public key as authenticated data. This quote a) identifies the user's enclave and b) cryptographically binds the ECDH public to the enclave. The quote is signed with the owner's private key, and then stored on the underlying storage service in a location that is accessible to the other users in the system.

$$Q = \text{QUOTE}(pk_e)$$
$$m1 = \text{SIGN}(sk_u,\, Q) \,|\, pk_e$$

   Where $\{pk_u, sk_u\}$ is the volume owner's public keypair and Q is the enclave quote with the enclave ECDH public key, pk_e, as authentication data.
2) *Exchange:* Whenever Owen wishes to grant Alice access to his file system, he must transfer a copy of his volume rootkey to Alice. To do this, Owen first validates the quote generated from Alice's enclave (by checking that the signature matches Alice's public key and verifying the quote with Intel) before extracting the enclosed enclave public key, pk_{ea}. Then, within the enclave, Owen generates an ephemeral ECDH keypair (pk_{eph}, sk_{eph}), and combines it with pk_{ea} to derive a common secret that encrypts his volume rootkey. The encrypted rootkey and the ephemeral ECDH public key (the private portion is discarded) are signed using Owen's private key and stored on the underlying storage service in a location that is accessible to Alice.

$$k \leftarrow \text{ECDH\_SECRET}(sk_{eph},\, pk_{ea})$$
$$h = \text{ENC}(k, rootkey)$$
$$m2 = \text{SIGN}(sk_o,\, h) \,|\, pk_{eph}$$

3) *Extraction:* Alice first validates Owen's signature and then, using the enclave private key, she derives the ECDH secret and decrypts the rootkey.

$$k \leftarrow \text{ECDH\_SECRET}(sk_{ea}, pk_{eph})$$
$$rootkey = \text{DEC}(k, h)$$

Since the ECDH secret can only be derived within the enclave, our protocol ensures the rootkey is only accessible within valid NEXUS enclaves. The rootkey can then be sealed and stored to Alice's local disk. Later, once Alice authenticates, she can decide to mount Owen's volume using the corresponding rootkey.

C. Access Control

Even after a user has been granted access to a volume's rootkey, access to files within the volume is further restricted via access control policies enforced by the NEXUS enclave. Access control is based on: 1) the user's identity as specified by the private key they authenticated with, 2) the permissions stored in the respective metadata. With this, access control enforcement is independent of the server, and because the metadata is encrypted and sealed, the access policies cannot be viewed nor undetectably tampered by any attacker.

We implemented a typical Access Control List (ACL) scheme in which users have unique IDs mapped to (username, public key) pairs, and permissions apply to all files (and subdirectories) within a directory. We leveraged the user list in the supernode to bind every user to a unique ID, and store the directory ACLs comprising of (user ID, access right) in the dirnode. Hence, to enforce access control within a given directory:

- The dirnode metadata is decrypted inside the enclave.
- If the current user is the owner of the volume, permission is granted to the user and the enclave exits.
- Otherwise, the user's ID is used to find the corresponding ACL entry inside the dirnode's ACL. Permission is granted if the user's ACL entry satisfies the required access rights.

NEXUS denies access by default and automatically grants administrative rights to the volume owner, who maintains complete control over their volume. Revoking a user is performed either by removing them from the user list, or removing their ACL entry from the dirnode. In both cases, the process is relatively inexpensive as it only requires re-encrypting the affected metadata.

V. IMPLEMENTATION

We implemented NEXUS as a Linux service that provides secure access to protected volumes. We extended OpenAFS [20]

— a widely used opensource distributed filesystem — to manage protected volumes on the network, without any modifications on the server-side or changes in the user's typical file management workflow. Our interface does not make any internal modifications to OpenAFS, it simply calls the NEXUS filesystem API via a shimlayer. Excluding third party libraries, our implementation comprises about 22618 SLOC. Integrating with OpenAFS (90K SLOC) required about 3200 SLOC.

Our prototype acts as a stackable layer interposed between user applications and the host filesystem. We split the prototype into an untrusted portion and a trusted portion. The untrusted portion (9005 SLOC) mainly (1) forwards requests into the enclave via the filesystem API, and (2) facilitates enclave access to data and metadata on the underlying storage service.

The NEXUS enclave is designed to be minimalistic; with a codebase size amounting to 9900 SLOC and a 512 KB binary, its verification is well within the reach of modern model-checking tools. Additionally, this small size ensures that NEXUS easily fits in enclave-reserved memory (SGX provides about 96 MB [30]). We included a subset of the MbedTLS Cryptographic Library [31], which added about 212KB. For GCM-SIV key wrapping, we used the C-based implementation provided by Gueron et al. [29, 32]. Our enclave interface comprises 13 enclave calls (ecalls), and 10 outside calls (ocalls). Ecalls invoke specific entrypoints within the enclave, and are mostly concerned with marshalling I/O requests from the filesystem API. Ocalls help with managing untrusted memory and accessing data/metadata objects. To prevent inadvertent data leakage, we sanity-check our inputs and employ secure data serializers on sensitive outputs.

A. Data Consistency

Because NEXUS manages metadata internally, every filesystem request triggers several I/O requests to the underlying storage service. As a result, in the situation whereby a file is simultaneously accessed by multiple users, a user's NEXUS enclave might fetch an older version of the metadata. To prevent this possible mismatch, on every filesystem request that updates metadata (e.g., create, delete, rename), NEXUS locks metadata structures via the facilities provided by the storage service. In our OpenAFS-based implementation, this locking is accomplished by invoking `flock()` on the metadata file. Once both data and metadata are flushed to storage, the lock is released, allowing users to access the file. Note that the lock is not required when accessing metadata files on read operations.

B. Optimizations

For every filesystem request to NEXUS, the enclave fetches one or more metadata objects from the backing store to complete the request. Because of the network cost, this makes metadata-intensive operations cost prohibitive. To address this, we introduced several caches to speedup data access. This includes a VFS-like directory cache structure (dentry tree) inside the enclave, and caching the metadata locally (unencrypted in enclave memory, or encrypted in untrusted memory). This way, unless a file is modified remotely, subsequent access need not download the file contents from the server.

To improve performance on larger directories, we split dirnodes into independently-encrypted *buckets*. Each bucket contains a user-configurable number of directory entries, and are stored as separate metadata objects. The main bucket stores the directory's access control as well as the MAC of each bucket to prevent rollback attacks at the bucket level. When writing the dirnode to the underlying storage service, the enclave only flushes the main bucket, and any *dirty* buckets.

VI. Security Analysis

Our goal is to provide a secure and scalable filesystem in which users maintain complete control over their data. Against the backdrop of threats outlined in Section III-A, we now discuss how NEXUS meets its security objectives. By combining encryption and access control within the enclave, NEXUS achieves *self-protection* [33]: the ability to protect sensitive data from all entities — trusted or untrusted — using the data's attached policy. Thus, security guarantees are:

- File and (protected) metadata contents, as well as file/directory names are confidential, and only accessible to authorized users.
- All data and metadata are tamper-evident, and can only be updated by individuals with the necessary write permissions.

Recall that we consider an attacker who has complete control of the server, including full access to all packets exchanged with the client, and a history of the user's encrypted files. Further, to encompass the abilities of a revoked user, we assume that the attacker once had access to the owner's volume i.e., the attacker has a copy of the volume owner's sealed rootkey, but their public key is no longer stored in the volume's supernode.

A. Confidentiality and Integrity

Confidentiality is enforced by encrypting all sensitive data within the enclave, allowing decryption only after performing adequate permission checks. The user's files are encrypted in fixed-sized chunks, and are re-encrypted using fresh keys on every file content update. These per-file chunk keys are stored in the encrypted portion of the filenode associated with the file. To protect directory entries, we replace the original human-readable filename (or directory name) with a random name, and store the correspondence in the encrypted portion of the dirnode. The metadata are re-encrypted on every update, and their encryption keys are key-wrapped with the volume rootkey. Therefore, to read the data, one must obtain access to the volume rootkey. Moreover, because all encryption is performed using AEAD cryptographic primitives, data integrity is provided alongside confidentiality. Hence, any illegal modifications of the ciphertext will be detected by the NEXUS enclave.

B. Authorization: Access to keys

Our security guarantees hinge on the secrecy of the rootkey, which must only be accessible within the enclave and require validation of the user's identity before use. At volume creation, the volume rootkey is generated within the enclave and is

persisted to the local disk using SGX sealed storage. This ensures that it cannot be accessed outside of a valid NEXUS enclave running on this particular processor. Before permitting the use of a volume rootkey, the NEXUS enclave validates the user's identity. To accomplish this, the user must demonstrate proof of knowledge of the private key associated with a public key stored in the volume's supernode via a challenge/response protocol. Therefore, even with a sealed copy of the rootkey, unless the attacker's public key is stored within the volume's supernode, they will be denied by the enclave.

As shown in Section IV-B, we enable secure file sharing by leveraging SGX Remote Attestation to exchange rootkeys between valid NEXUS enclaves running on genuine SGX processors. Our construction involves an asynchronous ECDH key exchange in which the recipient's NEXUS enclave is remotely attested before securely transmitting the rootkey encrypted with the ECDH secret. The ECDH keypairs are generated within the enclave, and their public keys are used to create SGX quotes. Since the ECDH private keys never leave enclave memory, the ECDH secret can only be derived within the enclave, thereby ensuring that the rootkey is not leaked unto untrusted storage. However, because we keep long-term ECDH keypairs fixed and exposed on the remote server, our key exchange protocol fails to provide perfect forward secrecy. In the event the attacker reconstructs the matching enclave ECDH private key, he will be able to extract every rootkey exchanged with the user. To mitigate this, we propose an alternative synchronous solution where, both parties generate ephemeral ECDH keys on every exchange and mutually attest their enclaves. This approach introduces an additional delay as it involves multiple rounds to attest the enclaves. Please note that in practice, the security and convenience tradeoffs of either approach will be left to the volume owner.

C. Attacking File System Structure

A malicious server might wish to modify the structure of the filesystem by, e.g., moving files or directories to other locations within the volume. This is prevented by the use of parent UUID pointers within our metadata structures, and the authenticated encryption used to protect these structures: the content of metadata cannot be altered without detection, and swapping of equivalently named objects will cause the parent UUID pointer validation to fail.

A more subtle attack is the *rollback attack*, in which the server exposes a previous version of a user's files. In this case, the NEXUS enclave will cryptographically validate the metadata, but cannot tell if it is the most recent version. To address this freshness issue, we have extended our metadata structures with a version number. On every metadata update, the version number is incremented and stored locally before uploading the metadata file. The metadata is considered stale if the downloaded version is lower than the local value. However, this approach is limited as it only offers per-file protection, and not protect the entire file hierarchy. As a result, a malicious server server could mount a *forking attack* [34], whereby file updates are hidden from users resulting in each user perceiving a different state of the volume. As a mitigating strategy, one could maintain a hash tree of the metadata content as part of the filesystem state [35, 10]. However, this naive design approach requires root-to-leaf locking along write paths to ensure metadata consistency. This not only impacts overall latency, but also raises synchronization concerns in ensuring the availability of the root hash. We leave further exploration of this protection and performance tradeoff to future work.

VII. EVALUATION

To show how NEXUS achieves the design goals outlined in Section III-B, we organized our performance evaluation around the following criteria:

1) **Utility.** Does our prototype support a wide range of user applications and workloads?
2) **Performance.** Are the overheads imposed by our prototype reasonable? How does it perform on workloads representative of normal users?
3) **Efficient Revocation.** How cheap are user revocations when compared to pure cryptographic techniques?

Experimental Setup. Our experimental hardware consisted of Intel i7 3.4 GHz CPUs with 8 GB RAM and 128 MB SGX Enclave memory. For SGX support, we installed v1.7 of the Linux SGX SDK [36]. On the server-side, we used the OpenAFS server distribution from the Ubuntu software channels. In our experiments, we compare our approach against an unmodified version of OpenAFS. For both setups, the file chunk size was 1MB. As for NEXUS, we set dirnode bucket size to 128 entries (See V-B), and used a normal AFS directory as the metadata backing store. Unless otherwise noted, all of our experiments are averaged over 10 runs.

A. Microbenchmarks

We ran several microbenchmarks to isolate the overhead incurred by NEXUS. Recall from Section IV-A that NEXUS requires repeated interaction with the underlying storage service to find the correct metadata. This generates additional network traffic on the server, and impacts the overall latency. Thus, we break down the overhead as follows:

1) *Enclave Runtime* — The total time spent within the enclave, including enclave transitions (ecalls and ocalls), access control enforcement and metadata encryption.
2) *Metadata I/O Latency* — The time spent performing I/O on the metadata objects, including reading, locking, and writing. This is influenced by two main factors: the directory depth of the path (each path component requires a metadata access) and the size of the metadata being accessed.

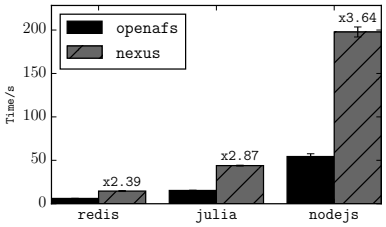
We start by measuring the overhead on basic file I/O operations using a python utility that reads and writes a file. Before each run, we flush the AFS file cache to ensure initial data access requires a network trip to the server. AFS follows open-to-close semantics, which implies that all file writes are local until the file is closed (at which point NEXUS encrypts the file chunks). Table 5a shows that overheads increase proportionally with the file size. The enclave cost

Prototype	File Size (MB)			
	1	4	16	64
OpenAFS	0.61	1.52	5.55	22.24
NEXUS	0.51	1.46	6.81	28.56
Metadata I/O	0.09	0.12	0.14	0.80
Enclave	0.02	0.09	0.58	2.07

(a) Latency (seconds) of File I/O operations.

Prototype	Number of files			
	1024	2048	4096	8192
OpenAFS	1.27	2.63	5.26	11.93
NEXUS	19.38	38.62	81.98	172.29
Metadata I/O	17.44	34.63	73.66	154.34
Enclave	0.38	0.79	1.67	3.55

(b) Latency (seconds) of directory operations.



(c) Latency for cloning Git repositories.

Fig. 5: Latency measurements.

Operation	OpenAFS	NEXUS	Overhead
LevelDB			
Fillseq	10.5 MB/s	8.1 MB/s	1.29
fillsync	2.2 ms/op	4.5 ms/op	2.04
fillrandom	5.9 MB/s	3.7 MB/s	1.59
overwrite	4.0 MB/s	2.6 MB/s	1.53
readseq	664.6 MB/s	718.1 MB/s	0.94
readreverse	425.0 MB/s	425.7 MB/s	0.99
readrandom	2.27 μs/op	3.7 μs/op	1.62
fill100K	11.0 MB/s	7.2 MB/s	1.52
SQLITE			
fillseq	6.5 MB/s	6.4 MB/s	1.01
fillseqsync	14.4 ms/op	31.4 ms/op	2.18
fillseqbatch	70.2 MB/s	69.7 MB/s	1.00
fillrandom	4.2 MB/s	4.2 MB/s	1.00
fillrandsync	13.4 ms/op	31.2 ms/op	2.34
fillrandbatch	7.6 MB/s	7.7 MB/s	0.98
overwrite	3.4 MB/s	3.4 MB/s	1.00

TABLE II: Database benchmark results.

per MB is almost constant (between $\times 0.01$ — $\times 0.02$), and enclave execution is a small contribution to the overall runtime. Metadata I/O overheads increase as the size of the filenode grows to accommodate more file chunks. However, this is still small compared to the file size (about 80B of encryption data for every 1MB file chunk).

Next, we analyzed the performance of directory operations using another python program that creates and deletes files within a flat directory. Table 5b shows that enclave execution scales proportionally while remaining a small component of the overall system runtime. However, the metadata I/O latency is a major constituent of the overall runtime because every file created increases the size of the directory dirnode, which becomes much larger than the corresponding directory entry. For large directories, this could result in significant performance overheads as the size discrepancy between the directory entry and the dirnode becomes more pronounced.

B. Database Benchmarks

We ran the database benchmarks of LevelDB [37] and Sqlite [38], two embeddable database engines commonly used to provide a data layer. Using 4 MB of cache memory, each benchmark generates several database files to emulate a key-value store of 16-byte keys and 100-byte values. The latency of various database operations was measured and displayed in Table II. NEXUS' performance closely matches OpenAFS in asynchronous operations. Because the benchmark tool does not wait for the data to propagate to disk, the overhead incurred by NEXUS is amortized and does not noticeably affect the overall latency. On the other hand, NEXUS incurs a $\times 2$ performance overhead on synchronous operations.

Workload		#files	Total Size
LFSD	Large Files and Small Directory	32	3.2 GB
MFMD	Medium Files and Medium Directory	256	2.5 GB
SFLD	Small Files and Large Directory	1024	10 MB

TABLE III: Workloads for benchmarking Linux Applications.

C. Cloning Git Repositories

We evaluated the performance of NEXUS in accessing volumes with arbitrary directory hierarchies by cloning 3 repositories: Redis (618 files), Julia (1096 files) and NodeJS (19912 files). Figure 5c shows a $\times 2.39$ and $\times 2.87$ overhead on cloning Redis and Julia respectively, while incurring a $\times 3.64$ overhead on NodeJS. This is because NodeJS has significantly more files/directories, a deeper directory hierarchy (up to 13 levels), larger directories (top 3 directories: 1458, 762, 783), and bigger files. This increases the server-side load as each filesystem access on NodeJS requires more metadata operations.

D. Linux Applications

In this test, we generated 3 characteristic workloads (Table III) to evaluate the performance of common Linux utilities:

- `tar -x`: Extract an archive.
- `du`: Traverse and list the file sizes.
- `grep`: Recursively search the term "javascript".
- `tar -c`: Create an archive.
- `cp`: Duplicate a file.
- `mv`: Rename a file.

With the exception of `du` and `mv`, all the applications perform both file and directory operations. To prevent cache effects, we flush the system cache before running each application.

Figure 6 shows the plot of the test over 25 runs. The `tar` extraction reaffirms the result of the directory operations microbenchmark: the relative overhead of NEXUS with respect to OpenAFS is proportional to the number of files in the directory. This is further confirmed by the single file write operations in the `tar` archive creation and `cp` tests; they impose a constant overhead across all workloads. The same applies to the single directory operation performed by `mv`. In the `du` test, NEXUS is indistinguishable from OpenAFS. Since

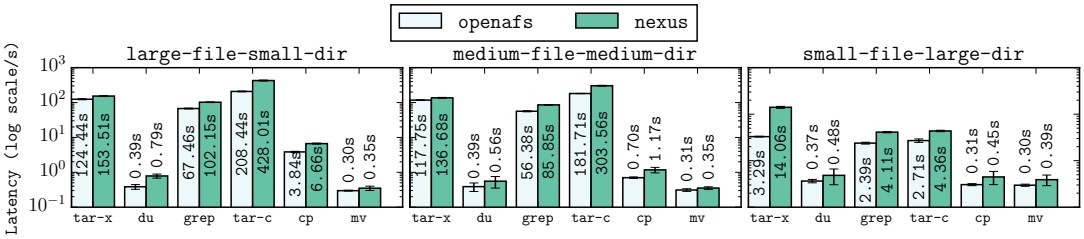


Fig. 6: Latency (over 25 runs) of common Linux applications under 3 generated worklords.

the directory structure is flat, once the corresponding dirnode gets cached in memory, lookup operations can be performed locally. The same applies to `grep`, which has an overhead between $1.5\times$—$1.7\times$ on all workloads.

E. Revocation Estimates

In a typical cryptographic fileystem, revoking user access involves the following steps: re-encrypting the affected file, uploading the file on the server, and then distributing the key to authorized users. Because NEXUS ensures encryption keys never escape the enclave boundary, revocation becomes as simple as re-encrypting the metadata with a new key. For instance, consider the scenario in which a user is revoked from the directory containing the `SFLD` workload mentioned above. For 10MB of file data, NEXUS will have to re-encrypt and update about 95KB of metadata (recall access control is stored in the dirnode). Whereas, in the `LFSD` workload, the metadata payload drops to 3.2KB for 3.2GB of file data.

F. Takeaway Discussion

The results of our evaluation demonstrate the ability of NEXUS to meet the demands of standard user workloads. While our approach does necessarily introduce additional overheads, these are predominately encountered during metadata modifying operations that generally do not fall on the critical path for most personal data workloads. In general, interactive programs exhibit less than $\times 2$ performance degradation, which we believe is acceptable in practice for the majority of users.

Moreover, NEXUS is designed to operate within a multi-user environment that offers standard file sharing capabilities. Although our evaluation occurs within a single machine, we document the costs of providing sharing as follows: (1) The asynchronous rootkey exchange (Section IV-B) requires a single file write. (2) Adding/removing users (Section IV-C) is not unlike revocation, which has been shown to require a singe metadata update. (3) Although policy enforcement (Section IV-C) scales with the number of ACL entries, its cost is dominated by the initial metadata fetch.

VIII. Related Work

SGX-Enabled Storage: Since its release, SGX has generated considerable research aimed at achieving secure remote storage [18, 16, 17, 27, 39, 40, 41]. PESOS [18] enforces custom server-side access control on top of untrusted storage, but its prototype requires a LibOS [42] that severely impacts the TCB. ZeroTrace [17] and OBLIVIATE [16] use an ORAM protocol to protect file contents and access patterns from the server, but do not consider file sharing. Moreover, because these solutions require server-side SGX support, they have limited applicability in the personal cloud storage setting. We circumvent this by running the NEXUS enclave on the client. SGX-FS [41] is an enclave-protected userspace filesystem, but does not provide any sharing capabilities.

IBBE-SGX [43] proposes a computationally efficient IBBE scheme [44] for achieving scalable access control. However, unlike NEXUS, its access control model restricts all group membership operations to an administrator.

Cryptographic Filesystems: By encrypting user files before uploading the ciphertext to the server, cryptographic filesystems [10, 11, 12, 14] have been proposed as a flexible solution for secure data sharing. Unfortunately, pure encryption techniques are plagued by issues of bulk file re-encryption on user revocation. This incurs a significant performance overhead which, according to Garrison et al. [15] is considerable even with modest access policy updates. Although mitigating schemes such as lazy encryption [45] and proxy re-encryption [46] have been proposed, concerns remain on how practical they perform under real world environments. By having the NEXUS enclave mediate access to all encryption keys, we offer superior user key management and obviate the necessity of bulk file re-encryption on policy updates.

IX. Conclusions

The protection of user data on cloud storage remains an active research area, however existing works either require substantial changes to server/client, or impose severe data management burdens on the user. We presented NEXUS, a stackable filesystem that protects files on untrusted storage, while providing secure file sharing under fine-grained access control. NEXUS is a performant and practical solution: it requires no server-side changes, and imposes minimal key management on users. NEXUS uses an SGX enclave to encrypt file contents on the client, and then attaches metadata that ensures the encryption keys are enclave-bound. Access control is enforced at each user's local machine, and file sharing is enabled using SGX remote attestation. We implemented a prototype that runs on top of AFS, which achieves good performance on file I/O operations and incurs modest overheads on workloads that involved bulk metadata.

Acknowledgements: This work was supported in part of the National Science Foundation under awards CNS–1704139 and CNS–1253204.

References

[1] Business Insider. Google Drive now hosts more than 2 trillion files. http://www.businessinsider.com/2-trillion-files-google-drive-exec-prabhakar-raghavan-2017-5, 2017.

[2] Dropbox. Celebrating half a billion users. https://blogs.dropbox.com/dropbox/2016/03/500-million/, 2016.

[3] SC Media. Data breach exposes about 4 million Time Warner Cable customer records. https://www.scmagazine.com/data-breach-exposes-about-4-million-time-warner-cable-customer-records/article/686592/, 2017.

[4] CNBC. Credit reporting firm Equifax says data breach could potentially affect 143 million US consumers. https://www.cnbc.com/2017/09/07/credit-reporting-firm-equifax-says-cybersecurity-incident-could-potentially-affect-143-million-us-consumers.html, 2017.

[5] ZDNet. Yet another trove of sensitive US voter records has leaked. http://www.zdnet.com/article/yet-another-trove-of-sensitive-of-us-voter-records-has-leaked/, 2017.

[6] Dropbox Terms of Service. https://www.dropbox.com/terms, 04 2018.

[7] Google Terms of Service. https://policies.google.com/terms, 2017 10.

[8] Microsoft Services Agreement. https://www.microsoft.com/en-us/servicesagreement, 8 2018.

[9] Privacy Rights ClearingHouse. Data Breaches. https://www.privacyrights.org/data-breaches, 2017.

[10] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *NDSS*, volume 3, pages 131–145, 2003.

[11] Aniello Castiglione, Luigi Catuogno, Aniello Del Sorbo, Ugo Fiore, and Francesco Palmieri. A secure file sharing service for distributed computing environments. *The Journal of Supercomputing*, 67, 2014.

[12] Erel Geron and Avishai Wool. CRUST: Cryptographic Remote Untrusted Storage without Public Keys. *International Journal of Information Security*, 8(5):357–377, 2009.

[13] Ethan Miller, Darrell Long, William Freeman, and Benjamin Reed. Strong security for distributed file systems. In *IEEE International Conference on Performance, Computing, and Communications.*, pages 34–40. IEEE, 2001.

[14] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36, 2002.

[15] William C Garrison, Adam Shull, Steven Myers, and Adam J Lee. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 819–838. IEEE, 2016.

[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*, 2018.

[17] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[18] 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*, EuroSys '18, pages 25:1–25:17, New York, NY, USA, 2018. ACM.

[19] Intel Software Guard Extensions Programming Reference, 2017. https://software.intel.com/en-us/sgx-sdk.

[20] The OpenAFS Foundation, Inc. http://www.openafs.org/, 2018.

[21] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[22] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. Endbox: Scalable middlebox functions using client-side trusted execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 386–397. IEEE, 2018.

[23] Bijun Li, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza. Troxy: Transparent access to byzantine fault-tolerant systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 59–70. IEEE, 2018.

[24] Mark Russinovich. Azure confidential computing. https://azure.microsoft.com/en-us/blog/azure-confidential-computing/, 5 2018.

[25] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: a distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[26] 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 *NSDI*, pages 145–161, 2017.

[27] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. LibSEAL: Revealing Service Integrity Violations Using

Trusted Execution. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 24:1–24:15, New York, NY, USA, 2018. ACM.

[28] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Fast*, volume 3, 2003.

[29] Shay Gueron and Yehuda Lindell. GCM-SIV: Full nonce misuse-resistant authenticated encryption at under one cycle per byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[30] Ittai Anati, Frank McKeen, Shay Gueron, H Haitao, Simon Johnson, Rebekah Leslie-Hurd, Harish Patil, Carlos Rozas, and Hisham Shafi. Intel software guard extensions (Intel SGX). In *Tutorial at International Symposium on Computer Architecture (ISCA)*, 2015.

[31] SSL Library mbed TLS / PolarSSL. https://tls.mbed.org/, mar 2019.

[32] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV implementations. https://github.com/Shay-Gueron/AES-GCM-SIV/, 2018.

[33] Yu-Yuan Chen, Pramod A Jamkhedkar, and Ruby B Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 14–27. ACM, 2012.

[34] David Mazieres and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117. ACM, 2002.

[35] Jinyuan Li, Maxwell N Krohn, David Mazieres, and Dennis E Shasha. Secure Untrusted Data Repository (SUNDR). In *OSDI*, volume 4, pages 9–9, 2004.

[36] Intel. Intel(R) Software Guard Extensions for Linux* OS. https://github.com/intel/linux-sgx, 2018.

[37] LevelDB. https://www.leveldb.org, 2018.

[38] SQLite. SQLite Home Page. https://www.sqlite.org, 2018.

[39] S. Shinde, S. Wang, P. Yuan, A. Hobor, A. Roychoudhury, and P. Saxena. BesFS: Mechanized Proof of an Iago-Safe Filesystem for Enclaves. *ArXiv e-prints*, July 2018.

[40] Ju Chen Yuzhe (Richard) Tang. LPAD: Building Secure Enclave Storage using Authenticated Log-Structured Merge Trees. Cryptology ePrint Archive, Report 2016/1063, 2016. https://eprint.iacr.org/2016/1063.

[41] Dorian Burihabwa, Pascal Felber, Hugues Mercier, and Valerio Schiavoni. SGX-FS: Hardening a File System in User-Space with Intel SGX. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 67–72. IEEE, 2018.

[42] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX ATC*, 2017.

[43] Stefan Contiu, Rafael Pires, Sébastien Vaucher, Marcelo Pasin, Pascal Felber, and Laurent Réveillère. IBBE-SGX: Cryptographic Group Access Control Using Trusted Execution Environments. *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 207–218, 2018.

[44] Cécile Delerablée. Identity-based broadcast encryption with constant size ciphertexts and private keys. In *ASIACRYPT*, 2007.

[45] Michael Backes, Christian Cachin, and Alina Oprea. Lazy revocation in cryptographic file systems. In *Security in Storage Workshop, 2005. SISW'05. Third IEEE International*, pages 11–pp. IEEE, 2005.

[46] Zhiguang Qin, Hu Xiong, Shikun Wu, and Jennifer Batamuliza. A survey of proxy re-encryption for secure data sharing in cloud computing. *IEEE Transactions on Services Computing*, 2016.