View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by D-Scholarship@Pitt

TOWARDS PRACTICAL ACCESS CONTROL AND USAGE CONTROL ON THE CLOUD USING TRUSTED HARDWARE

by

Judicael Briand Djoko

M.S. Computer Engineering, University of Pittsburgh, USA, 2015

B.S. Computer Engineering, The University of Akron, USA, 2013

Submitted to the Graduate Faculty of

School of Computing and Information Sciences Department of

Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2019

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Judicael Briand Djoko

It was defended on

Nov 15th 2019

and approved by

Adam J. Lee, Associate Professor, University of Pittsburgh

Jack Lange, Associate Professor, University of Pittsburgh

Panos K. Chrysanthis, Professor, University of Pittsburgh

Balaji Palanisamy, Associate Professor, University of Pittsburgh

Dissertation Advisors: Adam J. Lee, Associate Professor, University of Pittsburgh,

Jack Lange, Associate Professor, University of Pittsburgh

TOWARDS PRACTICAL ACCESS CONTROL AND USAGE CONTROL ON THE CLOUD USING TRUSTED HARDWARE

Judicael Briand Djoko, PhD

University of Pittsburgh, 2019

Cloud-based platforms have become the principle way to store, share, and synchronize files online. For individuals and organizations alike, cloud storage not only provides resource scalability and on-demand access at a low cost, but also eliminates the necessity of provisioning and maintaining complex hardware installations.

Unfortunately, because cloud-based platforms are frequent victims of data breaches and unauthorized disclosures, data protection obliges both access control and usage control to manage user authorization and regulate future data use. Encryption can ensure data security against unauthorized parties, but complicates file sharing which now requires distributing keys to authorized users, and a mechanism that prevents revoked users from accessing or modifying sensitive content. Further, as user data is stored and processed on remote machines, usage control in a distributed setting requires incorporating the local environmental context at policy evaluation, as well as tamper-proof and non-bypassable enforcement. Existing cryptographic solutions either require server-side coordination, offer limited flexibility in data sharing, or incur significant re-encryption overheads on user revocation. This combination of issues are ill-suited within large-scale distributed environments where there are a large number of users, dynamic changes in user membership and access privileges, and resources are shared across organizational domains. Thus, developing a robust security and privacy solution for the cloud requires: fine-grained access control to associate the largest set of users and resources with variable granularity, scalable administration costs when managing policies and access rights, and cross-domain policy enforcement.

To address the above challenges, this dissertation proposes a practical security solution that relies solely on commodity trusted hardware to ensure confidentiality and integrity throughout the data lifecycle. The aim is to maintain complete user ownership against external hackers and malicious service providers, without losing the scalability or availability benefits of cloud storage. Furthermore, we develop a principled approach that is: (i) *portable* across storage platforms without requiring any server-side support or modifications, (ii) *flexible* in allowing users to selectively share their data using fine-grained access control, and (iii) *performant* by imposing modest overheads on standard user workloads. Essentially, our system must be client-side, provide end-to-end data protection and secure sharing, without significant degradation in performance or user experience.

We introduce NEXUS, a privacy-preserving filesystem that enables cryptographic protection and secure file sharing on existing network-based storage services. NEXUS protects the confidentiality and integrity of file content, as well as file and directory names, while mitigating against rollback attacks of the filesystem hierarchy. We also introduce Joplin, a secure access control and usage control system that provides practical attribute-based sharing with decentralized policy administration, including efficient revocation, multi-domain policies, secure user delegation, and mandatory audit logging. Both systems leverage trusted hardware to prevent the leakage of sensitive material such as encryption keys and access control policies; they are completely client-side, easy to install and use, and can be readily deployed across remote storage platforms without requiring any server-side changes or trusted intermediary. We developed prototypes for NEXUS and Joplin, and evaluated their respective overheads in isolation and within a real-world environment. Results show that both prototypes introduce modest overheads on interactive workloads, and achieve portability across storage platforms, including Dropbox and AFS. Together, NEXUS and Joplin demonstrate that a client-side solution employing trusted hardware such as Intel SGX can effectively protect remotely stored data on existing file sharing services.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

DEDICATION

To my parents, Thérese and Alphonse.

To those who believed in me.

ACKNOWLEDGEMENTS

My deepest gratitude goes to Adam Lee. I still remember attending his security seminar class as a first year graduate student with little background in Computer Security. Adam exposed me to various aspects of security research, and his dedication inspired me to pursue the work that eventually led to this dissertation. This thesis will not be possible without his constant feedback and great patience. Overall, his love for teaching and work ethic are habits that I hope to replicate in my career and personal life.

I am also very thankful for my co-advisor Jack Lange. His high standard for clearness and simplicity in writing and presentation are amongst my most cherished lessons from graduate school. In our interactions, Jack's relentless feedback significantly improved the quality of this dissertation and my other publications.

I want to thank my committee members Panos Chrysanthis and Balaji Palanisamy in agreeing to an inconvenient time, and great feedback on my proposal and dissertation.

My time at the University of Pittsburgh will not have been possible without my friends at the security and database groups. I want to thank Nick, Daniel, Cory, Anatoli, Pranut, Injung for many engaging discussions and a few beers. I also thank Keena Walker for always being helpful and sorting through administrative hurdles.

I am especially thankful to my dearest friends for being there when it mattered: Jeff, David, Edem, Sheriff, Paterne, Bodie, Carey, Patrick, and Mike. I am also grateful for all my soccer friends — it was essential to my well-being. I am very indebted to Isaac Gamwo for inspiring and fostering my interest in advanced studies.

Above all, I owe everything to my family. My parents always believed me and made great sacrifices in order that I pursue my path in life. I also thank Hermann, Stephanelle, and Cyntiche for being a constant source of encouragement and consolation.

1.0 INTRODUCTION

Today, cloud-based file sharing platforms are amongst the most popular services on the Internet. For instance, mainstream file sharing services already boast hundreds of millions in daily users and host millions of gigabytes in stored files [1, 2, 3, 4]. Through these services, users gain access to large amounts of highly available storage and can collaborate with other users globally. Furthermore, by outsourcing data to the cloud, organizations can eliminate the cost and expertise required for provisioning and maintaining complex hardware installations, while only paying for what they use. Given other benefits like data backup and recovery, and the fact that these come at a low cost, it becomes clear why both individuals and organizations increasingly store private information on the cloud [5].

However, relying on cloud storage poses serious threats to data ownership. File sharing services are often built on top of third-party object storage platforms (e.g., Amazon S3, Google Cloud), which distribute user private information across a global network of servers and authorized client machines. Due to limited operational transparency, users have no clear control on where their data is stored, who has access, or whether certain operations are allowed. Unfortunately, even with the provider's best intentions, cloud-based platforms are frequent victims of external hackers, malicious cloud insiders, and unexpected disclosures [6, 7, 8, 9, 10, 11, 12]. These high profile incidents indicate that the very concentration of sensitive information on these platforms not only creates an attack target, but also engenders a situation whereby a vulnerability affecting a few providers can have a substantial user impact. For example, in 2015, a vulnerability on Dropbox, Box, and Google Drive allowed third-party access to shared files after the URL was indexed on search engines [13]. Furthermore, users have to be wary of unscrupulous service providers, who are legally permitted by their terms of service to mine and distribute private information without requiring user

Figure 1: Encrypted file synchronization setting model

consent [14, 15, 16, 11]. Thus, as individuals and organizations store more personal and private information on cloud-based storage platforms, this dissertation develops a real-world data security solution that is independent of the service provider or any trusted intermediary, while maintaining the ability to store, share, and synchronize files across user machines.

Existing cryptographic solutions to data security provide confidentiality and integrity by distributing encryption keys to authorized users, but do not address the practical issues of user revocation or dynamic access control updates over data stored on the cloud. To prevent a revoked user from accessing and making future updates, a naive revocation mechanism involves: (i) downloading the file from the server, (ii) re-encrypting the file with a new key, (iii) uploading the file to the server, and (iv) distributing the new key to users who are still authorized. When considering a large-scale environment with thousands of users and files that change frequently, revoking users through bulk file re-encryption and key distribution will incur significant computational and network penalties [17]. Thus, scalable access control on the cloud requires sophisticated key management enabling users to manage vast amounts of resources across administrative domains.

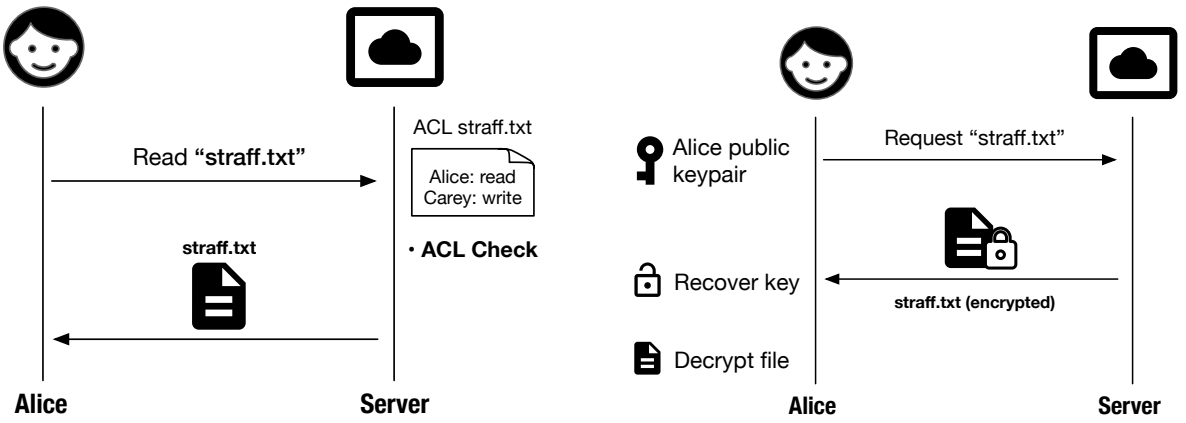
This dissertation focuses on data security in the file synchronization setting, whereby users employ a client application to manage and share files that are stored on a remote server (Figure 1). There is no direct user-to-user communication, users have no control over the server infrastructure, and the server is responsible for synchronizing file changes across individual client machines. Examples include mainstream file sharing services such as Dropbox and Google Drive, as well as traditional network filesystems like AFS and NFS. We

argue that users can still benefit from these services without compromising on data ownership. Our goal is to ensure data protection against other users of the service, external hackers, malicious cloud administrators, and unscrupulous service providers, such that confidentiality and integrity is guaranteed even in the event of a data breach or unauthorized disclosure. Furthermore, for mass user adoption, a practical security solution must fulfill the following high-level requirements: (i) **Portability**, for deployment across storage providers without requiring any server-side modifications or trusted intermediaries; (ii) **Flexibility**, whereby users can selectively dictate the conditions for data access using fine-grained policies; and (iii) **Performance**, in that typical user workloads run with modest overheads. Esseentially, given the key role of user sharing in the growth of cloud services, dynamic policy changes must be efficient, and should not significantly degrade system performance.

1.1 CURRENT APPROACHES

Consider an organization outsourcing internal documents to the cloud for shared access with employees and other external organizations. Notwithstanding the scalability and availability of cloud storage, the organization still wishes to keep their data as private as if it were stored on-premise. Access control is the ability to prevent unauthorized operations over sensitive information, while ensuring data confidentiality, integrity, and availability. In a client-server model, enforcement could be provided by: (i) deploying a server-side monitor that mediates every access request against a centralized policy database, or (ii) applying privacy-preserving techniques over sensitive data without any changes to the existing infrastructure.

Using a server-side reference monitor, Figure 2a depicts a user requesting read access to the "*straff.txt*" file. To prevent unauthorized access, every file is protected using an Access Control List (ACL) that contains the list of authorized users and their corresponding permissions (e.g., read, write, delete). In this example, the ACL shows that Alice can *read* the file. Upon receiving the request, the monitor checks the user's identity and permission within the ACL before returning the requested file to the user. However, this approach requires implicit trust on the server to: (i) enforce the access control policy faithfully, and (ii)



(a) Server-side mechanism

(b) Client-side mechanism

Figure 2: Client requesting *straff.txt* file from the server

protect the file contents at rest and in transit. Unfortunately, the threats surrounding cloud infrastructure makes such trustworthy assumptions non-trivial on the part of the organization.

Alternatively, data protection could be performed before uploading documents unto the server. A natural protection mechanism is to encrypt the file using a symmetric key, and distributing the key to authorized users. By protecting the file encryption key, symmetric encryption provides both data confidentiality and integrity to prevent attackers (who do not possess the key) from recovering or (undetectably) tampering with the file contents. Figure 2b shows the client-side mechanism for requesting the "*straff.txt*" file, which the server simply returns as an encrypted file without performing any extra permission checks. Once on their local machine, an authorized user in possession of the symmetric key can then proceed to decrypt the file and recover its contents. Since possession of the encryption key grants access to the contents, revoking a user essentially consists in re-encrypting the file with a new key, and re-distributing the new symmetric key to users who are still authorized. However, a naive implementation could incur significant computational and network penalties, especially as the number of users and resources explodes.

1.2 CHALLENGES

Authorization represents the set of permitted actions a user can exercise over objects [18]. Although authorization has been extensively researched within static scenarios where the set of users and resources is known, the unique characteristics of the cloud imposes several challenges. (i) First, user private data is stored and processed on remote machines that lie outside the owner's control. Consequently, in addition to protecting data at rest and during use, the security framework must support the inclusion of the user's environmental context (e.g., time) when evaluating authorization requests. (ii) Second, large-scale cloud deployments have thousands of users accessing an unlimited number of files across organizational domains. Thus, policy specification should provide administrators and data owners enough flexibility in associating users and resources with arbitrary granularity, without requiring a priori knowledge of specific entities. (iii) Third, permissions and policies change dynamically as users and resources are added, removed, and modified; therefore, policy administration costs must be scalable when updating access permissions and revoking users.

Addressing the above challenges requires a careful combination of privacy-preserving techniques and distributed usage control. The former transforms data in a manner that preserves confidentiality and integrity, whereas the latter extends access control notions to prevent data misuse even after authorization is granted. Essentially, usage control not only empowers users to selectively share data with arbitrary granularity, but also ensures continuous policy enforcement as the access context changes throughout the data lifecycle. For example, the usage control policy *"delete the file after 10 writes"* could abort an ongoing access request under satisfying conditions.

Privacy-preserving computation can be achieved using cryptography or isolated execution that relies on software and hardware mechanisms to prevent untrusted access. Much research has been done on cryptographic access control to protect sensitive data [19, 20, 21, 22, 23, 24, 25, 26]. However, when employed in a distributed setting, encryption turns user authorization into a key management problem; existing solutions either require server-side coordination, impose burdensome key management on users, or incur severe bulk re-encryption overheads on access revocation. Furthermore, incorporating environmental factors during policy evaluation

is not achievable using cryptography alone, but requires a tamper-proof and non-bypassable mechanism to enforce context-aware policies on remote machines.

Recently, Trusted Execution Environments (TEEs) such as Intel SGX and ARM Trustzone that provide isolated execution, sealed storage, and remote attestation have become standard security features in commodity machines and mobile devices; with a small TCB and minimal performance overheads, they provide a tamper-proof environment to securely manage sensitive applications secrets [27, 28, 29, 30, 31]. Existing work employing TEEs to secure untrusted storage require trusted hardware support on the server, which may not possible for users of typical file sharing platforms [32, 33, 34, 35, 36, 37]. In this dissertation, it is our hypothesis that *the widespread availability of hardware-enabled trusted execution environments on consumer devices can provide data confidentiality and integrity, as well as scalable access and usage control within an untrusted cloud environment, while improving portability, flexibility, and performance over unmodified remote storage platforms.*

1.3 APPROACH AND CONTRIBUTIONS

The high-level approach consists in combining the benefits of hardware-enabled TEEs and cryptographic protection to ensure confidentiality and integrity throughout the data lifecycle. TEEs perform arbitrary computations in a manner that cannot be subverted by any (trusted or untrusted) party, whereas cryptography relies on mathematical theory to protect persistent data. We employ TEEs to encrypt sensitive user information inside metadata objects without leaking key material. This prevents an attacker from recovering any sensitive information as encrypted content is only decryptable within the TEE, which can leverage its isolated runtime to enforce access control. Also, because the TEE controls key material, the user is not subject to the data retention policies of the service provider, since deleting the encryption key essentially makes the file content irrecoverable.

We propose two systems: NeXUS, a stackable filesystem that provides cryptographic protections to shared files, and Joplin, an access and usage control system for dynamic and fine-grained data sharing. NeXUS ensures confidentiality and integrity of file content,

alongside the names of files and directories. On the other hand, Joplin protects access control information, including user privileges and policies. Both systems are completely client-side, do not require any server-side changes, and leverage trusted hardware to enable transparent security protections over unmodified storage. Unlike prior solutions that require server-side hardware support or rely on a service provider enclave, our approach is novel in that a user-controlled TEE protects private information on individual client machines. As such, NEXUS and Joplin provide a solid foundation for developing user-centric and decentralized policy enforcement using trusted hardware. Our contributions are as follows:

Implementation and Evaluation. Building a scalable system requires taking into account TEE runtime limitations, as well as the I/O characteristics of the underlying storage platform. For example, current real-world TEEs have limited memory and restricted network access, whereas cloud storage imposes significant network latency. We developed prototypes for both NEXUS and Joplin using Intel SGX as the TEE. Notably, NEXUS manages a virtual filesystem, whereas Joplin hosts a policy engine that evaluates user-specified policies. We ported NEXUS to run atop FUSE to allow unmodified applications access to protected files on any storage platform. This allows users to maintain their typical workflow, without requiring any modifications to the OS or underlying filesystem. The Joplin prototype builds upon this for expressive access control and usage control over shared files. To demonstrate the effectiveness of our approach, we evaluated both prototypes over unmodified network storage such as Dropbox and AFS. Specifically, we measured the network latency under different scenarios using a variety of microbenchmarks and end-to-end tests. Results show that both prototypes are applicable to a wide range of user workloads, and provide scalable data protection when processing filesystem data and access control information. Therefore, users can manage protected data on existing storage platforms without significant performance degradation in their typical workflow.

Expressiveness and Obligations. Developing a robust access control and usage control solution for a large-scale cloud environment requires a fine-grained model for formulating policies, while also accommodating the local user context. Attribute-Based Access Control (ABAC) has been proposed as an expressive and dynamic model for open collaborative environments; ABAC determines access requests by evaluating a set of policies against user

attributes, object attributes, and relevant environmental conditions. Obligations pertain to usage control, and are mandatory actions or requirements that are fulfilled as part of a request (e.g., *delete the file after 20 reads*). Joplin provides continuous enforcement using cryptographically-protected metadata and a policy interpreter, while addressing other the practical challenges of decentralized policy enforcement, such as efficient revocation, multi-domain policies, secure user delegation, and mandatory audit logging. Access control information is completely managed and protected by the TEE, which enforces authorization requests using an ABAC model and performs obligations. As a result, data owners can manage sensitive content on the cloud using fine-grained authorization and obligation policies independently of the service provider or any trusted intermediary.

Portability and Performance. For maximum user adoption, it is essential that our system does not lock users into any particular service provider in order to achieve scalable access control. Therefore, we minimize our storage assumptions and treat the cloud storage as another generic layer providing a filesystem interface, while enabling efficient user revocation and access right updates. Specifically, both access control information and sensitive key material are stored inside cryptographically-protected metadata *files* that are only decryptable within the TEE. This approach requires no server-side changes, and allows seamless key distribution as the server synchronizes the encrypted metadata files across client machines. Furthermore, because key material remains under TEE control, revocation only requires updating and re-encrypting relatively small metadata, thereby obviating the bulk re-encryption and key redistribution overheads typically associated with purely cryptographic solutions. Overall, our approach allows for arbitrary deployment scenarios as it is seamlessly portable across remote storage platforms, while introducing minimal policy management costs.

Rollback protection. NEXUS prevents rollback attacks against protected files and directories. Because sensitive information is stored as a collection of files on the server, a malicious service provider may perform a rollback attack whereby older versions of the metadata are returned to the user. Although the returned metadata is cryptographically valid, this attack not only violates metadata freshness, but also the consistency of the entire protected state. To provide stronger integrity guarantees, we maintain a hash tree alongside filesystem information within the metadata files. A hash tree is an efficient method for authenticating a

collection of items using a cryptographic digest. When decrypting metadata at access time, the TEE can validate that the metadata is fresh with respect to a known local version and other metadata in the filesystem hierarchy. With these defenses, users can ensure that every filesystem state results from a valid sequence of changes, while limiting the server's ability to move/rename files and hide file updates.

1.4 ROADMAP

The remainder of this thesis follows a chronological organization. In Chapter 2 we cover the necessary background for cryptography, trusted hardware, and access control. Chapter 3 outlines our system and threat model. We discuss the design of NEXUS in chapter 4, and elaborate on how we enable cryptographic file sharing on existing cloud-based platforms. After giving some background and problem description, we proceed with a system design of its overall architecture, and implementation using Intel SGX. This chapter closes with a performance evaluation, related work, and conclusions. Chapter 5 introduces Joplin, a secure access and usage control system. We begin with some background and related work in decentralized policy enforcement. Then, we describe its design and implementation showing how its client-side approach can provide secure and scalable enforcement. We evaluate our SGX-based prototype via use cases and performance benchmarks. Chapter 6 demonstrates how NEXUS and Joplin meet their security guarantees in protecting filesystem information and enforcing access control. We conclude this dissertation in 7 with a summary of our contributions and some discussion of future work.

2.0 BACKGROUND

This chapter covers the necessary background in privacy-preserving computation and decentralized access enforcement. The former can be achieved either with cryptography (Section 2.1) or isolated execution. On the cloud, virtualization provides isolated resource management (i.e., CPU, memory, storage), but requires including the hypervisor and other untrusted software components as part of the trusted computing base. We explore how novel trusted hardware extensions can alleviate these security concerns (Section 2.2), and focus on Intel SGX (Section 2.3) as our target real-world instantiation. We close the chapter with an exploration of access and usage control (Section 2.4).

2.1 CRYPTOGRAPHIC PRELIMINARIES

This dissertation employs encryption and hashing primitives as building blocks in providing confidentiality, integrity, and freshness within a scalable system. *Confidentiality* prevents an attacker from recovering the plaintext, whereas *integrity* detects any improper or unauthorized modification of the ciphertext. Unlike confidentiality, integrity cannot prevent the attacker from tampering the file, but only provides detection. Integrity can be extended with a monotonic counter or random nonce to provide *freshness*. In exchanges involving multiple rounds (e.g., key exchange), freshness ensures that a given message is not a replay from a stale version. Consider that Carey stores tax return documents in an encrypted filesystem hosted on the server. Upon encrypting each file: (i) Confidentiality ensures that an attacker cannot see how much Carey made that year, (ii) Integrity guarantees that the contents are authentic and tamper-evident, and (iii) Freshness detects if the server is returning an older

version of the tax documents by comparing it with a known version. Moreover, we assume these cryptographic primitives are theoretically secure, and treat them as blackboxes.

2.1.1 Encryption and Hashing

At a high level, a encryption algorithm takes a sensitive *plaintext* (P) and an encryption *key* (K) as inputs, and outputs a random stream of bytes as *ciphertext* (C), such that $C = \mathrm{E}(K, P)$. Conversely, a decryption algorithm uses the decryption key (K^{-1}) to recover the original plaintext as follows: $P = \mathrm{D}(K^{-1}, C)$. Encryption mainly provides confidentiality, and the ciphertext length is a function of the encryption algorithm and the plaintext.

On the other hand, a hashing algorithm is a one-way function takes an input message (M) to return cryptographically secure hash of a given length: $H(M) \rightarrow \{0,1\}^l$. Hashing provides integrity, and can be combined with a key to produce Message Authentication Code (MAC) or tag, such that $\mathrm{MAC}(K, M) \rightarrow \{0,1\}^l$. The MAC or tag output is small and fixed (e.g., usually 32 bytes), and cannot be reverted to the original message.

2.1.2 Symmetric and Asymmetric Encryption

Cryptographic protection relies on the secrecy of encryption keys, which must be large, unique, and unpredictable. As such, key generation relies on a good source of randomness (e.g., CPU temperature) to ensure that the cryptographic operation is semantically secure, and the attacker cannot feasibly guess the encryption key.

Symmetric encryption uses the same key for both encryption and decryption. There exists 2 types: block ciphers that split messages into blocks, and stream ciphers that consume messages as a bitstream. The Advanced Encryption Standard (AES) is the state of the art symmetric key algorithm, and enjoys hardware acceleration support on various instruction sets [38, 39]. Further, symmetric primitives can be combined with a hashing function to provide both confidentiality and integrity. AES-GCM is an *Authenticated Encryption with Associated Data* (AEAD) cipher that combines the AES cipher and the GHASH hashing function [40, 41]. Although fast at encrypting large amounts of data, symmetric encryption requires establishing a common secret between the parties.

Asymmetric encryption uses different keys for encryption and decryption. The process relies on a (public, private) keypair, such that any plaintext encrypted with the public key is only decryptable using the corresponding private key, and vice versa. Although there exist a mathematical relation between the keys, the public key that can be released without compromising the private key, which must remain in the possession of the owner. A notable application of public key encryption is unforgeable signatures, which are the created using a *signing algorithm* with the hash of a message and a user's private key as inputs. RSA is an example of a signing algorithm [42]. Signatures are only verifiable using the corresponding public key, and provide origin authenticity (i.e., integrity) as the private key is only accessible to its owner. Although they use larger keys and are significantly slower than symmetric encryption, public key algorithms are mostly used to encrypt small pieces of data (e.g., encryption keys). Another example is hybrid encryption over untrusted storage, whereby a copy of the file encryption key is encrypted with the user's public key. At access time, the authorized user employs their private key to recover the file encryption key, before decrypting the file. RSA is also an encryption algorithm. As such, asymmetric encryption perfectly suitable to establish a secure communication channel in open environments where there is no preexisting shared secret. Elliptic-curve cryptography [43] can be used in a key exchange protocol to generate a common secret over an insecure channel. Elliptic-curve Diffie-Hellman (ECDH) is an example of a key exchange protocol [44]; it is usually coupled with nonces and a signature algorithm to ensure freshness and origin authenticity, respectively.

2.1.3 Cryptographic Notation

Table 1 lists the cryptographic primitives employed in this dissertation. Each primitive is used with their recommended key size, along with necessary additional information (e.g., using initialization vectors). We denote a (public, private) keypair as $\{pk, sk\}$, and use PKGEN() to indicate public keypair generation. SIGN(sk, m) represents a signature over m using sk, and VERIFY(pk, s) indicates the verification of a signature s using pk. Lastly, ENC/DEC(k, m) denotes symmetric encryption/decryption. Note that by using AES-GCM AEAD symmetric encryption, we gain confidentiality alongside integrity.

Algorithm	Key size (bits)	Guarantees
AES-GCM	128	Confidentiality, Integrity (Tag), Freshness
RSA	2048	Integrity (signatures)
ECDH	256	Freshness (Key exchange)

Table 1: Cryptographic primitives and their respective key sizes

2.2 TRUSTED EXECUTION ENVIRONMENTS

A Trusted Execution Environment (TEE) is a secure, integrity-protected environment, consisting of processing, memory, and storage capabilities [45]. TEEs facilitate the instantiation of trusted applications that are isolated within a hardware-protected context. Specifically, by placing the trusted application binary within a hardware-protected memory region, the CPU ensures confidentiality and integrity of both code and data against all untrusted system components, including other system processes, OS or hypervisor, and even external hardware devices. The TEE runs at the highest privilege level within the CPU context, and updates its state dynamically as the trusted application updates registers and memory locations. For cloud-based applications, TEEs can be remotely attested to prove the authenticity of the trusted application and the underlying platform.

Although early instantiations of trusted hardware were provided via secure co-processors like the TPM, they only provided a fixed set functions and a limited interface for processing cryptographic material [46]. However, recent years have witnessed several TEEs providing general purpose secure computation such as Intel SGX on x86 machines, and ARM Trustzone on mobile and IoT devices (e.g., Samsung KNOX) [27, 28, 29, 30, 31]. ARM Trustzone splits the processor into two logical modes: a *secure world* containing the TEE, and a *normal world* containing the normal OS runtime [27]. Each world has separate registers and memory, such that the normal world cannot observe the memory accesses of the secure world. Trustzone ensures that only the secure world can be executing at any time on a given chip, thereby

preventing potential side-channel attacks from untrusted software. This is unlike Intel SGX, which multiplexes hardware resources between trusted and untrusted software [28]. Sanctum and Keystone are open source TEE design that target the RISC-V hardware; their main improvement over SGX is the prevention of a class of side-channel attacks [30].

These features has inspired extensive work on TEE-enabled secure remote computation, including blockchains, machine learning, databases, IoT, and remote data storage [47, 32, 48, 49, 34, 35, 37, 50, 51, 52]. However, extending trusted hardware protections beyond the isolated runtime environment unto persistent storage poses several challenges. Although the main idea behind TEEs is encompassing security-critical functionality within a small container that exposes a minimal interface (e.g., cryptography), a full fledged application requires integration with the rich execution environment provided by the OS and other untrusted applications. Specifically, data protection must persist across TEE restarts and data migrations, while accommodating runtime restrictions such as limited memory (e.g., SGX provides about 96MB) and no direct access to system software functionality (e.g., networking, time). Therefore, the trusted application must perform proper sanity-checks and apply cryptographic protections when copying sensitive data across the TEE boundary.

2.3 INTEL SOFTWARE GUARD EXTENSIONS

Intel SGX [28, 53] is a set of x86-based processor extensions that provide secure execution environments called *enclaves*. These extensions enable clients to both measure and verify the code running within an enclave, while also providing very strong isolation guarantees. The measurement is a hash of the initial enclave code and data to denote the enclave identity, which is verified by the CPU to detect any tampering at launch time. Further, the enclave measurement is also used to generate reports during enclave attestation. When activated, enclaves execute in user space and are protected from inspection or modification by other processes, the OS and hypervisor, BIOS, or even hardware peripherals. At the hardware level, enclaves exist as a special CPU context that ensures data privacy by blocking access from other hardware devices and encrypting the contents of enclave-managed memory as it leaves

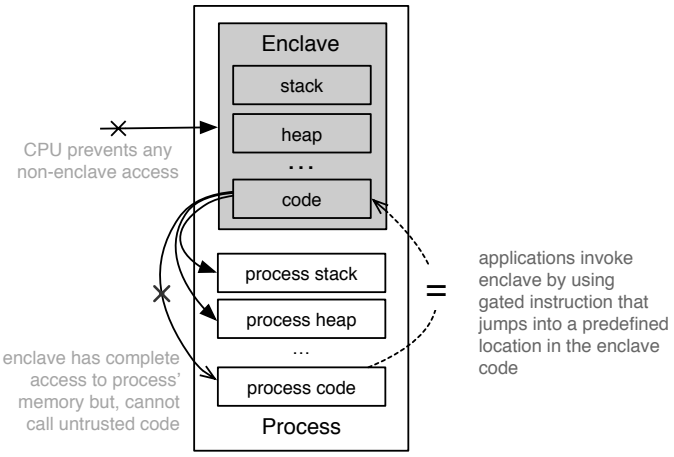


Figure 3: Isolated Execution. The enclave (grey) has a separate stack, heap, and code sections that are independent of the untrusted portion.

the CPU boundary. 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. With such guarantees, the SGX platform effectively limits an application's Trusted Computing Base (TCB) to the CPU package and the enclave code. We now describe the SGX features relevant to this dissertation: *Isolated Execution*, *Sealed Storage*, and *Remote Attestation*. For a more in-depth discussion of SGX, we refer the reader to the following publications [29, 54, 28, 53, 55].

2.3.1 Isolated Execution

An SGX application is comprised of an untrusted portion and an enclave. The untrusted portion coordinates the enclave lifecycle by invoking both system calls (e.g., enclave creation and destruction) and userspace SGX instructions (e.g., entering the enclave). Note that the untrusted portion does not have access to any application secrets (e.g., encryption keys, passwords), but simply serves as a proxy for enclave communication (e.g., network access) and integration (e.g., file interface). On the other hand, the enclave exposes a set of entry functions, which are invoked by the untrusted portion to perform specific operations.

Depicted in Figure 3, 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 encrypted and inaccessible from untrusted code, including OS/hypervisor, and hardware devices. Because SGX assumes a multiprocess environment, it allows privileged software to control the assignment of EPC pages unto multiple enclave instances. The CPU prevents unauthorized cross-enclave access by tagging each EPC page with their assigned enclave instance and checking this at runtime. On termination, the enclave's EPC pages are cleared to prevent any leakage of sensitive information.

The application enters the enclave by invoking an SGX *ecall* that switches the CPU context and jumps to a predefined entrypoint of the enclave code. While executing, the enclave code performs arbitrary computations, accesses enclave memory, and can read and write to untrusted memory. By allowing access to untrusted buffer pointers, an enclave can efficiently exchange data with the host application. However, the enclave code is not allowed to directly call untrusted functions, but must invoke an SGX *ocall* that explicitly switches from enclave mode before calling the untrusted function. The enclave memory is defined as a linear range within the virtual space the host application. Per virtual memory semantics, the OS is responsible for translating enclave virtual addresses into their corresponding EPC page, but the CPU ensures that enclave memory can only be accessed from enclave code. The enclave code exits once execution is completed, and can be re-entered on subsequent invocations. The enclave memory state is preserved across invocations, but is lost once the enclave is terminated.

2.3.2 Sealed Storage

From the description above, the enclave loses all data on termination. To securely persist sensitive data across executions, SGX allows enclaves to derive a *sealing key* from platform keys burned in the CPU fuses. The sealing key is symmetric, and can then be used to encrypt and seal sensitive data before copying it to untrusted memory. On restart, the enclave regenerates the sealing key to decrypt the sealed information. The sealing key can only be

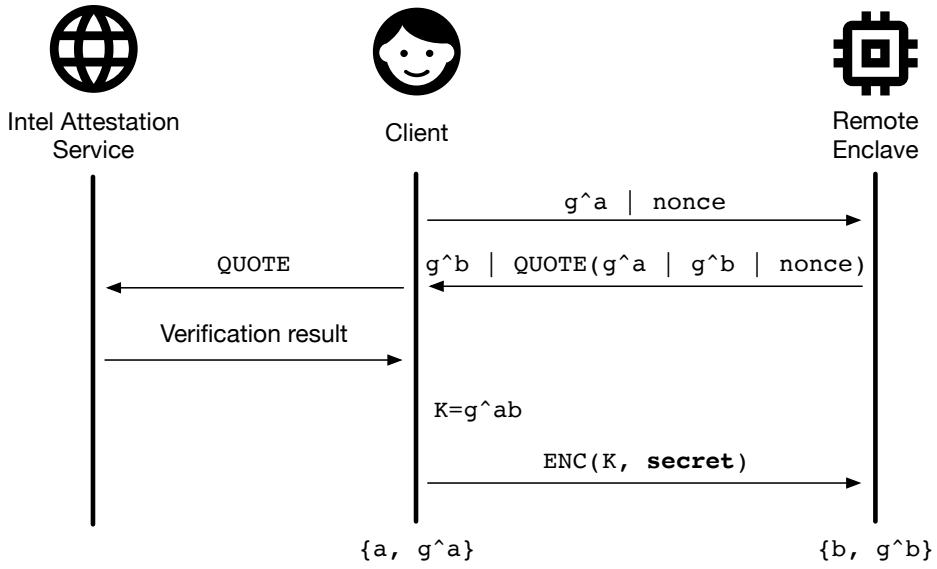


Figure 4: SGX Remote Attestation example for simple ECDH key exchange. The enclave quote contains the client-generated nonce and both public ECDH keys. In the end, both the client and enclave generate the shared key `K`.

generated within enclave memory, and is unique to the enclave identity and the particular CPU within which the enclave is executing. More specifically, this sealing key cannot be derived in any other SGX machine or by a different enclave on this particular SGX machine. As a result, SGX sealing is especially useful to protect long-term secrets on the local machine in a manner that prevents access outside of an enclave context as offline migration to other SGX platforms. For example, the enclave could generate a public-private keypair and seal the private key to local disk. At runtime, the enclave private key is unsealed within trusted space, and then used as cryptographic material to establish secure communication channels.

2.3.3 Remote Attestation

Remote attestation allows a *challenger* to validate the identity of a remote enclave, and its underlying platform. Specifically, attestation confirms that the trusted hardware is still considered valid (e.g., CPU model is not blacklisted), and the secure software runtime

is initialized correctly. In SGX, this process relies on two Intel-provisioned enclaves: a *Provisioning Enclave* that verifies the platform with Intel's attestation service to fetch a unique asymmetric attestation key, and a *Quoting Enclave* that uses the attestation key to generate *quotes*. A quote is a signature of the target enclave's measurement along with enclave-generated data (e.g., nonce, ECD public key), which is used to ensure freshness. This allows the challenger to verify the quote using an Intel-provided public certificate, and ascertain the enclave identity by checking its measurement and the additional data. Figure 4 depicts a simple ECDH key exchange, where the client generates a nonce and the remote enclave generates a quote. A critical step in the exchange is to verify the quote via the Intel attestation service, which ensures that the quote is from a valid enclave running on SGX hardware. Once the quote is validated, the client then uses the contained enclave public ECDH key to derive a common secret that is only derivable within the enclave. As such, remote attestation is essential in bootstrapping secure communication channels with a remote enclave, before provisioning any secrets (e.g., sending a password).

2.3.4 SGX Limitations

The Intel SGX SDK provides a complete toolchain for building and distributing enclaves. Its programming model relies on the programmer to carefully split the application into trusted and untrusted portions, as well as define an interface for exchanging data across the enclave boundary. This is specified as a set of ecalls and ocalls within an *EDL* file. The enclave runtime is managed by system software, which shares hardware resources between other applications and device drivers. Therefore, beyond resource contention, understanding the limitations of SGX hardware is critical to design a practical and secure enclave.

- The EPC memory has a current maximum size of 128MB, which leaves about 96MB for application enclaves (some is used for SGX metadata). For larger enclave workloads or in situations of high memory pressure, the OS can swap EPC pages with main memory using SGX privileged instructions. Before copying EPC pages into main memory, the CPU encrypts and applies freshness protections to prevent replay attacks from privileged software. This paging mechanism incurs significant performance overheads of up to 40K

cycles, especially when the enclave working set exceeds 128MB [56, 57].

- Because the enclave cannot directly invoke untrusted functions, access to system services (e.g., file I/O, time) requires exiting the enclave, executing the system call, copying the results into enclave memory, and re-entering the enclave. These enclave transitions are expensive as they require flushing and restoring the CPU context to prevent data leaks. Although library OSes can facilitate enclave communication with system software, such an approach not only explodes the enclave TCB, but may also conversely affect overall application performance [58, 59, 60]. Conversely, we argue that enclave applications can be tailored to operate within a dynamic cloud environment, without requiring any server-side support for trusted hardware.
- The enclave runtime can be interrupted at any time (e.g., page fault, exception), which triggers an Asynchronous Enclave Exit (AEX). SGX handles this by safeguarding the enclave state for resumption once the fault is handled. Each AEX overhead is minor but repeated exits could seriously slow down the enclave execution.
- SGX sealing does not offer any rollback protection to prevent sensitive state replay in between enclave restarts. Therefore in practice, applications desiring strong freshness guarantees have either resorted to slow and write-limited hardware monotonic counters, or relied on a distributed quorum of enclaves [47, 61, 62].
- Several side-channel attacks have been identified on SGX, including cache attacks, page-fault attacks, branch prediction attacks, and speculative execution. These attacks exploit the fact that the SGX architecture collocates enclave execution on the same hardware as other system resources. *In this dissertation, SGX side-channel attacks are considered out-of-scope.* However, work on defense measures is currently the subject of active research [63, 64, 65, 66, 67, 68, 69], and can be applied orthogonally to our work.

2.4 ACCESS AND USAGE CONTROL

Access control is employed to enforce security requirements such as confidentiality and integrity of data resources (e.g., files, database tables) to prevent unauthorized use of resources (e.g.,

programs, processor time, expensive devices), or to prevent denial of service to legitimate users [18]. The decision to authorize or deny a given operation is guided by a security policy. Examples of practical violations that can be prevented are students seeing the grade files or an employee tampering documents. Thus, an access control system is comprised of: (i) policies that dictate the authorization conditions, (ii) a formal model that defines core access control elements and their relations (e.g., users, attributes), and (iii) an enforcement mechanism that provides low-level functions and implements system policies according to the model. Because policy expressiveness is ultimately dictated by the access control model, selecting one that is suitable for the target use-case is critical. On the other hand, a reference monitor unifies policy enforcement by mediating all object accesses, but its implementation is typically a function of system constraints and environmental factors. We go over existing access control models and reference architectures to explore how they can be practically implemented within a dynamic cloud environment.

2.4.1 Traditional Access Control

Discretionary Access Control (DAC) is a model whereby data owners can grant or revoke access to their objects, such that authorization consists in validating the user's identity within an access structure [70]. DAC is popular within OS and database system as ACLs or user capability lists, where each file/row has an owner who can share access with other users or groups in the system using predefined permissions. For example, Alice might create a shared folder within her home directory on a shared server, and specify that the "work" group has both read and write access. This form of decentralized management alleviates the administrative burden, as users can discretely delegate their assigned privileges to others. However, because the data owner cannot prevent leakage of assigned privileges, DAC is more suitable for environments where the set of users and their possible operations are known.

Mandatory Access Control (MAC) eliminates user discretion and restricts information flow using security classifications and system-wide policies [71]. For example, in the Bell-LaPadula confidentiality model, the administrator assigns a clearance level to each user and a sensitivity level to each object, such that users cannot read objects at a higher classification nor write

to objects at inferior classifications. To inhibit data modifications, the Biba Integrity model allows a users to read objects at a higher classification, but can only write to lower classified objects. However, MAC models are best suited for rigid environments such as military and government agencies, where there is no concept of user ownership, and data confidentiality and integrity are of primary concern. Unfortunately, this cannot be feasibly applied to open environments where user membership and access rights are dynamic.

Role-Based Access Control (RBAC) is an approach whereby administrators define roles; grant permissions to roles according to tasks; and assign users to roles based on their duties [72]. Users can activate a subset of their assigned roles for a given session, and then access the objects permitted by the activated roles. RBAC improves permission management over MAC and DAC, as roles group users by responsibility and users can control how roles access objects they own. For additional flexibility, roles can be organized in a hierarchy to inherit permissions between junior and senior roles, and extended with constraints to provide rich semantics (e.g., separation of duty). However, RBAC is more suited for organizations where the role definitions are static, and user responsibilities do not change frequently. Because permission management only depends on role assignment and requires knowledge of the user or object entity, RBAC does not readily support multi-domain policies where entities are not known in advance, and environmental context is required for multi-modal access control.

2.4.2 Attribute-Based Access Control

Attribute-Based Access Control (ABAC) is an emergent model that determines authorization requests using user attributes, object attributes, environment attributes, and policies expressed as logical sentences over those attributes [73, 74, 75, 76, 77, 73, 78, 79]. Attributes are traits that are either statically assigned by the administrator (e.g., role, clearance), or dynamically set by the runtime (e.g., time, IP). For example, after assigning the necessary atttributes, an organization can specify the given policy to protect internal documents: "*Insiders* can write *Feedback* documents before *1PM*". As a result, attributes can group users and objects with arbitrary granularity and without the explicit identification of pairwise relationships between entities. Policies then combine attributes to provide fine-grained access control,

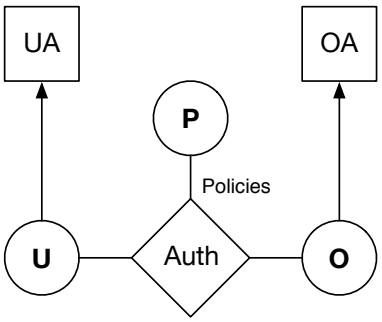


Figure 5: ABAC_α

such that expressiveness is only limited by the policy language and the richness of attributes. With such flexibility, administrators can formulate a concise set of policies to regulate user authorization, without a priori knowledge of individual user or object entities. Attributes and their values are captured as users and objects are provisioned unto the system, while keeping the pairwise relationships encoded within the existing policies. As a result, ABAC provides dynamic access control as changes in attributes may result in a different evaluation result between requests.

Many ABAC variants exist, but in this dissertation we focus on ABAC_α [75], a foundational model with minimal assumptions on system attributes and administrative control. The state of an ABAC system consists of users (U), objects (O), user attributes (UA), object attributes (OA), permissions (P), and authorization policies. Users are authenticated individuals requesting access, and objects are protected resources. Attribute are key-value pairs, and operate as functions that takes a user or object entity to return an atomic value or a set of values. Each user is associated with a set of user attributes that are assigned by the system administrator (e.g., name, occupation). Objects are created by users, and likewise have a corresponding set of object attributes (e.g., type, format). Permissions are simply the possible actions that users can perform on objects (e.g., read, write, delete). Policies govern access to system resources, and are expressed as a boolean combination of attributes. Each authorization policy takes the permission, user, and object as inputs; returns a true or false depending on their attributes. A user is permitted access on an object if there exist a policy

that satisfies a subset of the user and object attributes.

Despite its simplicity, ABAC_α still provides enough expressive power to configure MAC, DAC, and even RBAC. For instance, ACLs could be represented by associating each object with as many attributes as there are permissions; each attribute maps to a set of authorized users. Likewise, capabilities are user attributes that map permissions to a set of objects. MAC security labels can be handled by atomic-valued user (e.g., clearance) and object (e.g., sensitivity) attributes. To support RBAC, each user has an associated user attribute function that returns a set of roles, while objects are assigned attributes that map each permission to a list of roles. This has resulted in several works proposing enhancements of traditional access control models with attributes [80, 81].

Traditionally encompassed within a trusted reference monitor, the enforcement mechanism must be integrated with other security components such as authentication, administration, cryptographic protection, and audit logging, while also distributing access control information via the storage infrastructure. In Figure 6, a minimal reference monitor consists of the Policy Decision Point (PDP), Policy Information Point (PIP), and Policy Enforcement Point (PEP). The PIP is responsible for fetching and updating access control information such as attributes and policies. On each access request from the PEP, the PDP communicates with the PIP to fetch the necessary access control information before updating its internal state and evaluating system policies. The PEP enforces the PDP result, and only releases the object if the user is authorized.

However, access control is restricted to one-time request-response authorization, such that once a user is granted access, there's no further action that regulates future data usage. As usage spans over a longer time period, a policy such as "do not read consumer report after 30 *days*," could invalidate future data access. Unlike traditional closed-world settings in which users are known beforehand and enforcement occurs within a trusted subsystem, this may be problematic in an open setting whereby data storage and processing occurs on remote machines. Thus, in a distributed cloud environment, the security framework should allow data owners to specify what factors to continuously re-evaluate while access is ongoing.

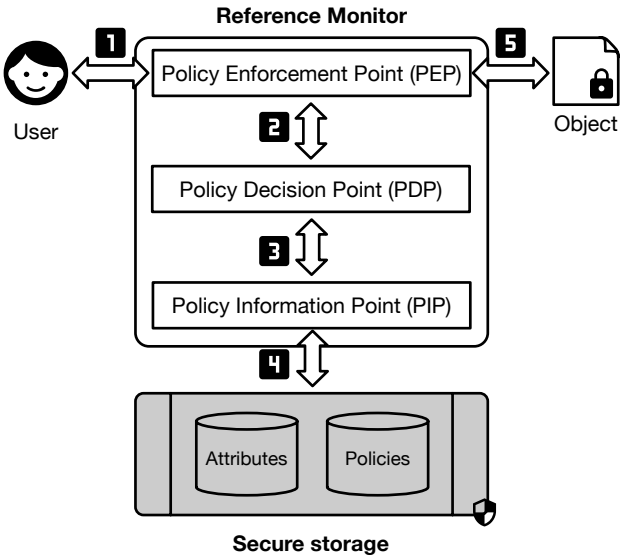


Figure 6: Reference monitor Architecture

2.4.3 Usage Control

Usage control (UCON) extends traditional access control notions with attribute mutability and continuous enforcement [82, 83, 84, 85]. Mutability refers to the fact that the user's access may change while access is in progress. This may be the result of the usage or a static update from the administrator. Continuous enforcement ensures that usage policies are re-evaluated throughout the usage period, and may result in a termination if considered invalid. With attribute-based authorization model, UCON also introduces obligations and conditions as part of a usage policy. Obligations denote mandatory action or requirements that have to be fulfilled before, during, and after usage, whereas conditions define the necessary environment context before and during usage. Usage control can be enforced using the same architecture as access control. However usage control in distributed cloud environments present two principal challenges: (1) integrating environmental factors beyond user and object attributes, and (2) tamper-proof and verifiable enforcement mechanism [86, 87, 88, 89, 90, 91, 92]. In this dissertation, we leverage the isolation and attestation features of commodity trusted hardware to enable distributed usage control.

3.0 SYSTEM AND THREAT MODEL

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 data persistence and availability, 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 remotely-stored data. Beyond regular file system access, many services also provide auxiliary sharing capabilities with other users of the service.

We follow prior work in cryptographic storage by considering an *owner* who sets up a secure *volume* inside a synchronized directory, and shares this with multiple *users* [93]. System actions are a combination of key-management operations and cryptographic protections, which are observed as encrypted reads and writes on the server. Within this context, we aim to provide users with additional security guarantees against unauthorized disclosure or modification of their files without hindering their ability to share these files with other *authorized* users.

Security Objective. *Unless granted explicit access by the owner, file contents along with access control information 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, as well as the integrity of the directory structure. Additionally, we aim to provide: (i) *Forward secrecy* to prevent revoked users from accessing future access control updates even when they

are in possession of older files, and (ii) *Backward secrecy* to prevent newly added users from accessing older data versions.

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. We do not consider availability attacks (e.g., denial-of-service), as our primary concern is protecting the confidentiality and integrity of file system contents. We do not protect against access pattern attacks, which can be addressed using orthogonal techniques [33, 94, 36]. Users can read/write data they are allowed to, and may attempt to read and modify files to which they do not have access. 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. Our enclave is assumed to be correctly implemented and free of any security-relevant vulnerabilities. Also, 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. However, SGX does not explicitly defend against software and hardware side-channels, and existing defenses against are orthogonal to our work. We assume side-channel attacks are considered out-of-scope and would apply solutions from the literature. Overall, these assumptions are in line with the standard SGX threat and widely adopted by prior work [60].

4.0 NEXUS

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 two client-side NeXUS prototypes: AFS and FUSE that allow unmodified user applications to access protected volumes as directories. Although our AFS-based prototype required minimal changes to the AFS client for efficient cache management, porting to FUSE demonstrates the generality of our design as well as facilitate the migration of volumes across arbitrary filesystems. We provide stronger freshness guarantees by maintaining a hash tree alongside the filesystem hierarchy to mitigate against rollback attacks, without requiring direct communication between users. Our evaluation reveals a $\times 2$ overhead for a variety of interactive user workloads and improved portability over AFS and Dropbox filesystems, while offering reasonable performance overheads when comparing our FUSE prototype to the AFS implementation.

4.1 INTRODUCTION

File-sharing services such as Dropbox and Google Drive have received widespread adoption in recent years [95, 4]. Through these services, users gain access to large amounts storage, and can collaborate with sharing capabilities enforced by the service provider. However, the recurrence of data breaches and unplanned disclosures has raised concerns about user privacy on these platforms (e.g., [6, 8, 9]). Another source of concern is the fact that service providers are legally permitted to modify and distribute sensitive data without requiring user discretion (e.g., [14, 15, 16, 11]). Consequently, it is evermore critical to provide a security solution that addresses these risks [96] as more users store sensitive information on these services.

Much research has been done to provide cryptographic access control over untrusted storage. The canonical approach is to encrypt the file end-to-end, and distribute the encryption key to authorized users. However, purely cryptographic access control solutions either require server-side coordination, impose burdensome key management on users, or incur severe bulk re-encryption overheads on access revocation [19, 20, 21, 22, 23, 24, 25, 26]. Recently, other works have used trusted hardware to provide strong security primitives over untrusted storage [32, 33, 34, 35, 36, 37]. However, these typically require trusted hardware on the server or do not consider user sharing. This limits their applicability, especially when users cannot readily modify the server-side components and consider dynamic sharing as an essential part of their workflow.

We propose a practical security solution that: (1) allows for user deployment without requiring any server-side coordination or trusted intermediary, (2) supports user sharing with custom access controls, and (3) performs with comparable overheads on typical user workloads. Our goal is to protect the confidentiality and integrity of a user's files against all untrusted parties, including unauthorized users, external attackers, and even the service provider.

To address this need, we present NEXUS, a privacy-preserving filesystem 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 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 the files' cryptographically protected metadata, which are decrypted by the enclave for enforcement 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.

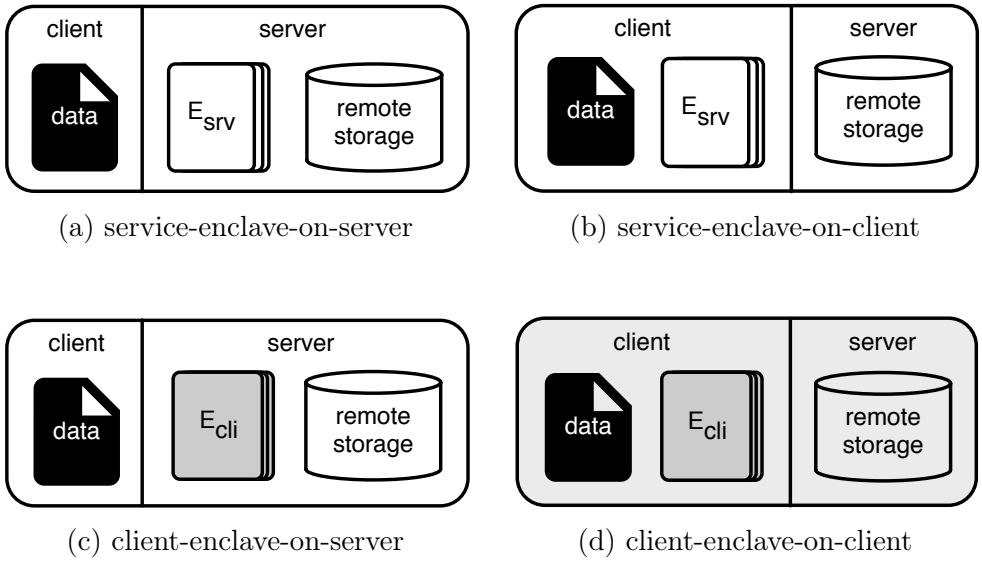
NEXUS is user-centric, transparent, and requires no server-side changes. The aim is to maintain the user workflow, as well as the functionality and benefits of the underlying storage platform. NEXUS is implemented as a protection layer between users/applications and an underlying filesystem, and leverages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

This chapter makes the following contributions:

- We propose a novel client-side architecture that allows 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 of a client-side enclave, NEXUS allows for seamless and secure key distribution, minimal user key management, and efficient user revocation.
- 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.

- We propose a cryptographic protocol that uses SGX remote attestation to enable secure file sharing between users on different machines. Communication is completely in-band and asynchronous, as it uses files on the underlying shared filesystem to exchange data and does not require both users to be simultaneously online.
- We propose an optional rollback protection mechanism that prevents the server from returning stale metadata. Our construction provides stronger freshness guarantees by leveraging the filesystem hierarchy to maintain a hash tree within encrypted metadata objects. Without requiring any direct user-to-user communication, our defenses ensure fork consistency by limiting server equivocation attacks to a single occurrence.
- We implemented a NEXUS prototype that runs as a userspace daemon on top of the AFS, a network filesystem popular with research and educational institutions. Without requiring any server-side coordination or trusted intermediary, We modified the AFS client to allow unmodified applications the ability to access protected volumes as AFS directories, whilst cryptographically enforcing AFS ACLs at the directory level.
- We implemented another NEXUS prototype that runs as a standalone FUSE userspace filesystem, and allows unmodified applications to access protected volumes from a mounted directory. Compared to the AFS-based implementation, our FUSE prototype improves portability by: (1) requiring no changes to the OS or underlying filesystem, and (2) enabling the migration of volumes by simply copying its directory contents across filesystems.
- We evaluate NEXUS over two popular remote data stores, Dropbox and AFS. Using microbenchmarks and other end-to-end latency tests, results show that NEXUS incurs modest overheads on standard user workloads, supports a wide-range of Linux applications, and obviates bulk re-encryption overheads typically associated with cryptographic solutions. When compared to the AFS-based prototype, our FUSE implementation improves portability and offers comparable performance characteristics.

The chapter is organized as follows: Section 4.2 provides an account of our protection model and the limitations of existing cryptographic solutions. In Section 4.3, we describe the design of NEXUS, and Section 4.4 provides a prototype implementation. Respectively, Section 4.5 describes the performance evaluations of the NEXUS prototype. We review related work in Section 4.6, and Section 4.7 concludes the chapter.



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

(c) client-enclave-on-server (d) client-enclave-on-client

Figure 7: Different architectures for enabling SGX security in a client-server environment. Each architecture shows a different combination of enclave location and enclave provenance.

4.2 BACKGROUND AND PROTECTION MODEL

4.2.1 SGX Design Space

With its robust security primitives, SGX presents a wide range of options on deploying enclaves within a cloud setting. Depending upon the security needs of the distributed application, different considerations must 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.

Figure 7 shows all the possibilities within this design space. The *service-enclave-on-server* (e.g., PESOS [34]) and *service-enclave-on-client* (e.g., EndBox [97]) 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 [51]) 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 [98]). Second, a substantial amount of server-side software may need to be retrofitted for SGX support. Depending upon the system's complexity, this may be a challenging task as changes could range from modifying the client – server communication protocol, to including untrusted software components inside the enclave [33, 99, 47, 34, 48]. On the other hand, the *client-enclave-on-client* scenario only requires that the user trusts their local machine, and does not impose any server-side support for its deployment. Moreover, this architecture enables a user-centric approach, such that the enclave protects sensitive user information using a user-specified policy.

4.2.2 Cryptographic Filesystem Design

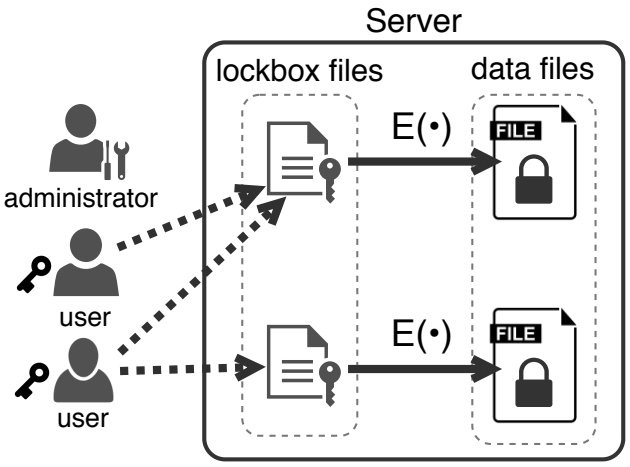


Figure 8: Architecture of a typical cryptographic filesystem. Encrypted file data are protected by a lockbox, which are in turn cryptographically restricted to authorized users.

Cryptographic Filesystems employ encryption techniques to enforce access control on untrusted storage. In this section, we construct a strawman filesystem and then discuss its practical implications within a real-world environment.

4.2.2.1 Strawman Construction

Figure 8 depicts a closed-world model typical of existing cryptographic filesystems in which the system consists of administrators, users, and a storage provider [22, 17]. Administrators are responsible for managing users and group membership operations. Users can read and write files, which are centrally stored by the storage provider. Each user is assigned a public keypair, and must digitally sign every operation. Although encryption provides confidentiality and integrity, it cannot prevent an unauthorized user from overwriting data. Thus, we rely on a minimal server-side reference monitor that serializes system writes and ensures data updates are from authorized users by checking digital signatures. The aim is to ensure that file data and access control information are only accessible to authorized users.

This construction realizes a simplified file sharing scenario in which read access to an encrypted file is protected using lockbox (e.g., [21, 100]). File data is stored in an encrypted form on the server, and the lockbox contains the file's encryption key and the list of authorized users. The lockbox is cryptographically protected, and access to the file encryption key could be protected using Identity-Based Encryption (IBE), or Attribute-Based Encryption (ABE). Thus, file access consists of recovering the file encryption key inside the lockbox with the user's private key, before decrypting the file contents.

4.2.2.2 Practical Implications

Despite the simplicity of our strawman construction, the revocation process requires: (i) re-encrypting the file with a new key, (ii) removing the revoked user's access from the filekey, and (iii) updating the other user entries to grant access to the new file encryption key. This method is unsuitable for large files or updates impacting multiple files. Moreover, given a large number of users in the filekey, the resulting overhead could be prohibitive. These operations result from the fact that file decryption allows for key-scraping attacks [101], whereby a user caches the file encryption key in anticipation of future revocation. Although alternative approaches support re-encryption on the cloud, their deployment model requires placing some trust on the server, have complex key management, and exhibit asymptotic re-encryption costs.

4.2.3 Our Approach

NEXUS combines the client-side encryption model used by existing cryptographic filesystems with SGX security guarantees. Shown in Figure 7d, 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 facilities (Sealed Storage). Then, before sharing keys with authorized users, we ensure the exchange occurs between valid NEXUS enclaves running on genuine SGX processors (Remote Attestation). As a result, encryption keys are never leaked to untrusted memory, and as such, kept under the complete control of the NEXUS enclave.

We explore a deployment model that targets applications generating sensitive data exclusively at the client and rely on a remote server as a storage provider. In the case of distributed filesystems, the user's file contents are opaque to the server, which we assume can access, modify, and disseminate any file that it stores [14, 15, 16]. 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: (i) our solution can be easily deployed without any out-of-band setup, file synchronization service implicitly provides key distribution, and (ii) 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.

4.3 SYSTEM DESIGN

4.3.1 Design Goals

In designing NEXUS, we balanced security and ease of use with the following aims:

1. **Practicality.** After the 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 should allow 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., [24, 26, 21]). Our goal is to offer similar protections with superior key management, efficient revocation, and no server-side 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 dictate file access permissions selectively. 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. Finally, NEXUS must ensure that the unencrypted data contents never leave the TCB unless the access control policy allows it.

4.3.2 High-Level Architecture

In order to meet the objectives outlined in Section 4.3.1, we have designed NEXUS to allow users transparent security protections on existing file storage services. NEXUS presents to the user a regular filesytem directory based on a protected *volume*. 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 (`UUID`s). Each object is stored as a regular data file on the underlying storage service using its `UUID` as the filename. In effect, NEXUS implements a virtual filesystem on top of the underlying target filesystem. Figure 9 shows a high-level NEXUS configuration.

Accessing data from a NEXUS volume consists of the user issuing filesystem 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 filesystem 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 storage services (including object-based storage services).

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 encrypt volume objects individually. Since the enclave creates it, NEXUS can 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 2.3.2) and stored on the local filesystem in an 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

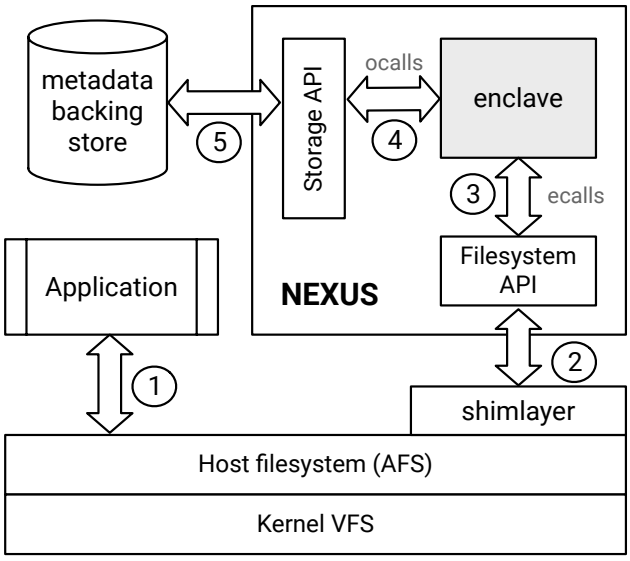


Figure 9: NEXUS architecture.

NEXUS enclave, which is also able to apply the file's access control policy before exposing the data to the user (Section 4.3.5). 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 can to provide sharing capabilities (Section 4.3.5) using SGX remote attestation (Section 2.3.3), 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.

4.3.3 Filesystem Interface

Users access data in NEXUS using standard filesystem interfaces, which are translated into a set of generic API calls implemented by the NEXUS enclave. This API is shown in Table 2, 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

Filesystem Call	Description
Directory Operations	
`nexus_fs_create()`	Creates a new file/directory
`nexus_fs_remove()`	Deletes file/directory
`nexus_fs_lookup()`	Finds a file by name
`nexus_fs_stat()`	Returns stat information
`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 2: NEXUS Filesystem API. The arguments typically include the directory path(s), and file name(s).

decrypting and performing access control checks at each layer. This method 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. Moreover, NEXUS contains several performance optimizations to limit the impact of these overheads (Section 4.4).

4.3.3.1 Metadata Structures Figure 10 gives a high-level overview of the structure of a NEXUS volume. NEXUS stores the file system structure internally using a set of encrypted

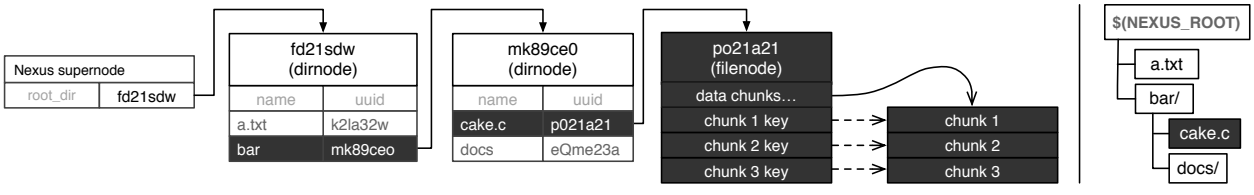


Figure 10: **Authenticated user view.** Directory traversal by NEXUS to present the plain contents of the user's data files.

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 filesystem (seen on the right side of Figure 10) is only accessible by decrypting the metadata inside the NEXUS enclave. The metadata not only store the filesystem layout, but also the cryptographic keys and access control policies needed to ensure that the filesystem data is confidential and tamper-evident.

The metadata structures implement a standard hierarchical filesystem namespace. Each NEXUS filesystem is specified by a *supernode* (corresponding to a *superblock* in a normal filesystem). The filesystem 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 filesystem's root directory along with the identity (public key) of the filesystem's owner. It also contains a list of other user identities that have been granted access to the filesystem 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 filesystem is immutable. However, the owner can add and remove authorized users at any time. Moreover, the supernode also stores a hardlink table that maintains the `UUID` of files with multiple links. The hardlink table consists of (`UUID`, link count) entries.
- **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. Also, each 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

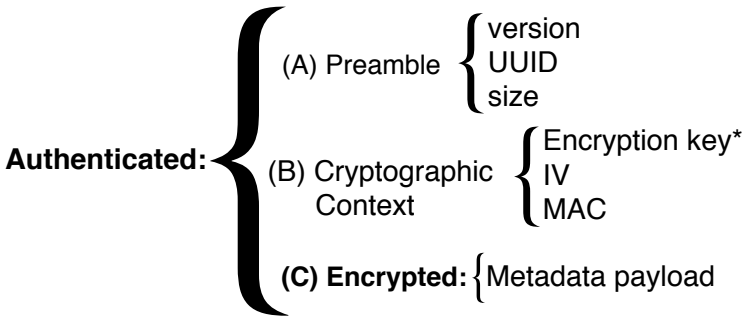


Figure 11: Metadata Layout. The encryption key is protected with the volume rootkey, which is only accessible within the enclave.

context. These contexts are stored as an array in the filenode structure with the UUID corresponding to the actual data file.

4.3.3.2 Metadata Encryption Figure 11 depicts 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 stores non-sensitive information (e.g., `UUID`, size). This section is integrity-protected.

(2) A cryptographic context containing the information used to secure 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 for 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 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 [102] AEAD construction, and refer the reader for a more in-depth discussion on keywrapping.

The metadata is protected using its cryptographic context which, in turn, is protected using the rootkey. This approach simplifies key management,as it embeds every encryption key within its corresponding metadata. Therefore, to access a volume, an user only needs to store the volume's sealed rootkey, which can only be unsealed within the NeXUS enclave running on the particular platform.

4.3.3.3 Metadata Traversal Because a NeXUS 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 10 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`. Initially, 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 `UUID` field of the loaded dirnode matches the value in its parent. By ensuring we load the correct metadata, we guard against *file swapping* attacks [26] that threaten the integrity of the filesystem structure (Section 6). If the verification or lookup operation fails, the metadata traversal terminates. Otherwise, the final metadata object is returned.

4.3.3.4 Virtual Filesystem Operations We now describe how the NeXUS enclave orchestrates the metadata structures to implement the Filesystem API in Table 2. Each API

call takes a target path, which is traversed in order to fetch metadata. The enclave then invokes a handler with the metadata to perform the filesystem operation. On completion, all modified metadata are committed to network storage, and the enclave output is returned to the untrusted caller. We assume that the enclave virtual filesystem is single-threaded, and does not support concurrent requests from multiple threads (left for future work). For simplicity, we omit details on basic filesystem checks (e.g., checking if the directory is non-empty before deleting its metadata).

- **create():** This operation creates a file/directory. For example, consider creating `foo.txt` in the root directory (Figure 9). We first check that file name is not present inside the root dirnode. After generating a `UUID`, the enclave inserts a new (name, `UUID`) pair into the root dirnode, and creates the child (filenode) metadata.
- **remove():** This operation removes a file/directory by name inside the parent dirnode and also deletes the corresponding filenode/dirnode metadata. If the entry is a file, we check for other links using the hardlink table before deleting the filenode metadata. If multiple links exist, we decrement its link count within the hardlink table and keep the filenode metadata.
- **lookup()/stat():** Common filesystem calls used for checking if a file/directory exists.
- **symlink():** A symlink is a shortcut to a target file. This creates a (name, symlink target) entry in the dirnode.
- **hardlink():** Hardlinks associate a name with a file. This increments the file's link count inside the hardlink table.
- **rename():** Renaming takes four arguments: the source directory, the target directory, the old name, and the new name. The old name is removed from the source dirnode, and the new name is added to the target dirnode. Note that the renamed entry still points to the same metadata.
- **encrypt()/decrypt():** Encrypt and decrypt are used to write and read files, respectively. The required arguments include the file path, offset, and a buffer to hold the file data. After fetching the filenode metadata, the offset is used to find the chunk entry that contains cryptographic material. The file contents are then encrypted/decrypted inside the enclave, and copied into the untrusted buffer.

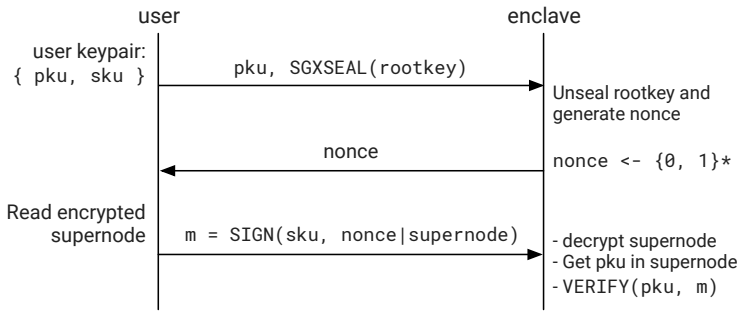


Figure 12: User Authentication with NEXUS enclave

These provide the minimum set of operations for interfacing with a fully functional filesystem API such as FUSE (Section 4.4.2). Note that to operate as a secure filesystem, NEXUS performs necessary access control checks prior to executing any of these functions (Section 4.3.5).

4.3.4 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 filesystem'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.

In NEXUS, identity is established using public-private keypairs, 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. With this, authenticating into a volume involves the following challenge-response protocol (Figure 12):

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.
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 authorized, the volume is mounted and becomes available.

4.3.4.1 User Sharing 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 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 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 13):

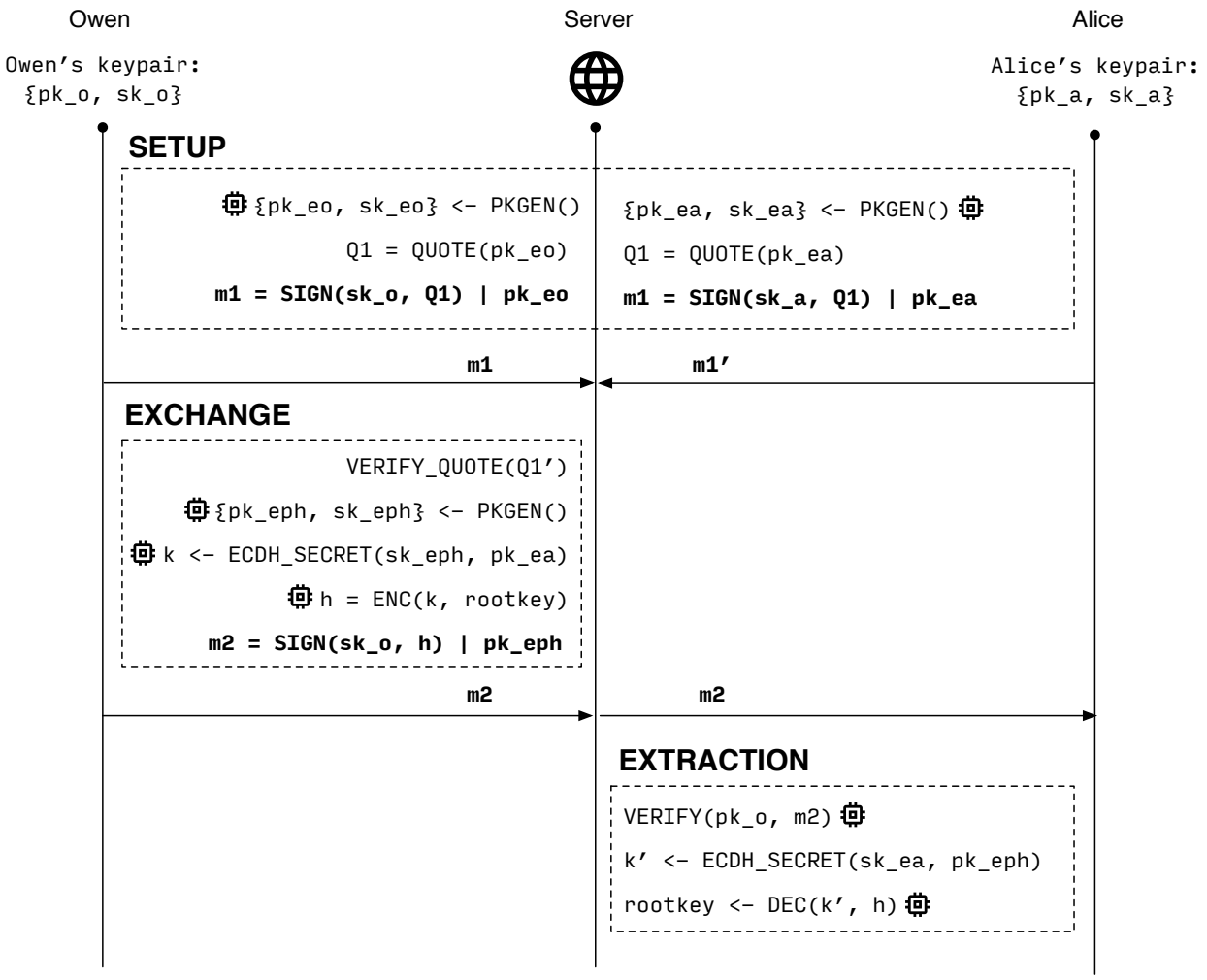


Figure 13: Key Exchange protocol diagram for Owen sharing his NEXUS volume rootkey with Alice.

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 identifies the user's enclave and 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.

4.3.5 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: (i) the user's identity as specified by the private key they authenticated with, (ii) 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.

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, permission) tuples in the encrypted portion of 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. Authorization is granted if the user's ACL matches the requested permission.

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 either case, the process is relatively inexpensive as it only requires updating and re-encrypting the affected metadata.

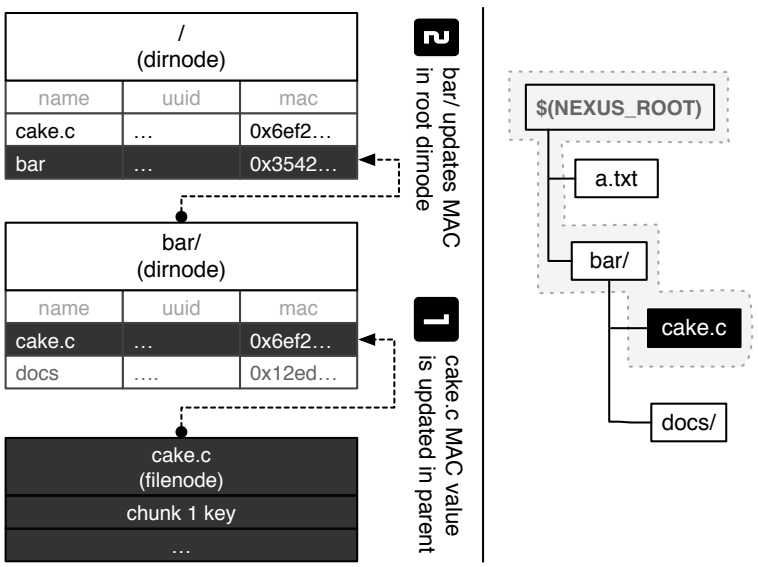


Figure 14: Metadata update after writing to `bar/cake.c` (right). After propagating the MAC values to the root dirnode, the root MAC and version are then stashed locally.

4.3.6 Rollback Protection

NEXUS provides data confidentiality and integrity by enclosing the filesystem state within encrypted metadata. However, this does not prevent *Rollback Attacks* in which the server or attacker returns older metadata versions to the user. Since these metadata are still cryptographically valid, the enclave will successfully decrypt and integrity-check their contents. Thus, without strong freshness guarantees, an attacker could trick the enclave into using a stale or inconsistent filesystem state.

NEXUS prevents rollback attacks by maintaining a hash tree alongside the filesystem hierarchy. A typical hash tree implementation involves hashing the sibling nodes (i.e., contents of a directory) together to create a super hash, which is then stored in the parent node (i.e., dirnode). The digest of the entire tree results in a root hash, which is stored to a secure location (e.g., local client machine). This way, data integrity only requires verifying the root hash and each hash down to a leaf node. We employ a similar strategy by using the NEXUS filesystem hierarchy as a tree. The hierarchy is made of dirnode and filenode metadata, which

store directory and file content respectively. Also, the root dirnode sits at the top of the hierarchy, dirnodes can have other dirnodes and filenodes as children, and filenodes have no children. Our scheme consists in ensuring that the integrity of each metadata is a function of its content and that of its children. Thus, we extend dirnode's directory entries with a MAC field that is managed as follows:

4.3.6.1 Verifying Metadata Every client maintains a local copy of the root hash and its version, which correspond to the MAC and version of the volume's root dirnode. After successfully decrypting any metadata, we verify its MAC as follows:

- For the root dirnode, we first check its MAC against the local root hash value. On failure, we then compare the fetched metadata's version against the local stashed value. If newer, the local root dirnode version and MAC are updated, otherwise the process is aborted.
- For other metadata, their MAC value is checked against their directory entry inside their parent metadata.

4.3.6.2 Updating Metadata We update the MAC value of the directory entry whenever its corresponding metadata is re-encrypted (e.g., file creation/deletion). For example, consider the series of metadata updates when a user writes to `bar/cake.c` (Figure 14). First, the `cake.c` filenode is re-encrypted and its new MAC is updated inside its corresponding directory entry in `bar/`. Then, the process is repeated for `bar/`, which updates its directory entry inside the root dirnode. Finally, after re-encrypting the root dirnode, its MAC and version are stashed into a local file as the root hash.

In addition to ensuring the freshness of individual dirnode or filenode metadata, the above process also protects the hierarchy from root dirnode to leaf dirnode/filenode. In NEXUS, the filesystem structure stores an ordered list of entries in the dirnodes and restricts every file/directory to a single parent metadata. For directories and files with a single link, this is simply the parent dirnode. However, because hardlink files are allowed in different directories, they can be updated through multiple paths and cause a potential mismatch at verification. Thus, we extended the hardlink table with a MAC field for each entry, such that when updating or verifying a file with multiple links, the hardlink table is used as the parent

metadata. As a result, the path to any metadata is unique, thereby allowing the formation of a tree alongside the filesystem hierarchy.

4.4 IMPLEMENTATION

We developed NEXUS as a userspace application that leverages the SGX SDK to securely access protected volumes. We first explain the overall architecture and provide details on the enclave implementation. Then, after describing how NEXUS was ported to run atop FUSE, we complete the section with our data consistency measures and some runtime optimizations.

Our prototypes act as a stackable layer interposed between user applications and the host filesystem. NEXUS is split into an untrusted portion (11200 SLOC) that runs in normal userspace, and a trusted portion that runs inside the enclave (9161 SLOC). The *untrusted portion* mainly implements the NEXUS filesystem API and facilitates enclave access to metadata via the storage API. The *trusted portion* performs secure operations, including key management, metadata encryption, and access enforcement.

The NEXUS enclave is designed to be minimalistic; its small codebase amounts to a 912 KB binary size, which allows for verification by modern model checkers. Additionally, this small size ensures the NEXUS enclave can easily fit into the limited 96 MB enclave-reserved memory and leaving enough memory for runtime allocations. For cryptographic support, we included a subset of MbedTLS Library (216 KB binary), and a C-based implementation of GCM-SIV key-wrapping construction [102]. The enclave interface comprises of 29 enclave calls (ecalls) and 10 outside calls (ocalls). Ecalls invoke specific entry points within the enclave, and are responsible for marshaling data into the enclave. Ocalls manage untrusted memory and access data/metadata objects. To prevent inadvertent data leakage, we sanity-check our inputs, ensure enclave pointer access is within trusted memory, and employ secure data serializers on sensitive outputs. Specifically, every sensitive file or metadata content is automatically encrypted and sealed using enclave-bound keys, before copying the resulting ciphertext to untrusted memory.

4.4.1 AFS Implementation

AFS is a distributed filesystem that relies on a set of trusted servers to provide transparent access to user files over the network. We implemented NEXUS as a userspace Linux service that extends OpenAFS [103] (the de facto open source AFS implementation) 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. To summarize our changes: (i) Because OpenAFS resides in kernel space, we added a device driver that routes the AFS filesystem requests to the NEXUS userspace daemon; (ii) we adapted the OpenAFS file chunking system to use the same chunk size as for seamless interoperability during I/O operations; (iii) we set up a shared memory map region, directly accessible from both kernel and userspace to enable the encryption of large files without any extra copying; and (iv) we modified the AFS "`fs setacl`" utility in setting the access rights of protected directories with AFS ACLs. Excluding third party libraries, our implementation comprises about 22618 SLOC. Integrating with OpenAFS (90K SLOC) required about 3200 SLOC.

4.4.2 FUSE Filesystem

We ported NEXUS to run atop FUSE in about 2353 SLOC. FUSE is a cross-platform userspace library that facilitates the development of filesystems via a simple API [104]. Its high-level architecture consists of: (i) a kernel module that manages the FUSE filesystem, and (ii) the libfuse user space library which communicates with the kernel module on each filesystem request. FUSE allows NEXUS to expose a POSIX-like interface to client applications and provide access to NEXUS volumes on a variety of local and remote filesystems without requiring any server-side changes. As shown in our evaluation, NEXUS supports both AFS and Dropbox with no modifications to their client applications, while allowing volumes to be copied as directories across network filesystems. Our prototype intercepts filesystem requests within a mounted directory and then invokes the corresponding NEXUS filesystem API function for processing. To synchronize the FUSE filesystem view with the enclave, we

maintain a directory structure of the filesystem state in untrusted memory. This structure is updated on every lookup or stat. Moreover, NEXUS follows open-to-close I/O semantics by performing all file writes on the local machine, and only committing changes on `close()`.

Initially, we extended the OpenAFS client to develop a NEXUS prototype that manages protected volumes on an AFS filesystem by simply routed network API calls from kernel space to our NEXUS userspace application. Although relatively less portable, this approach allows our prototype to benefit from the cache management and other kernel-level optimizations provided by the AFS client. Overall, both AFS and FUSE prototypes require kernel to userspace transitions as part of their data flow, but the AFS implementation only performs that action when interacting with the network as it heavily caches data in kernel space. Moreover, it is crucial to note that both implementations are largely unoptimized (maximum local performance was not our principal aim), and were mainly developed to demonstrate the adaptability of the NEXUS filesystem API.

4.4.3 Consistency Considerations

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 multiple users simultaneously access a file, 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. This locking is accomplished by invoking `flock()` on the metadata file. Once the metadata is flushed to network storage, the lock is released, allowing users to access the file. Note that locking is not required on read operations.

Moreover, enabling rollback protection in NEXUS requires maintaining hash-tree consistency encoded across multiple metadata objects. Thus, on stateful filesystem operations, we prevent write-write conflicts by locking metadata from root-to-leaf as part of the metadata traversal. This lock order ensures that accessing and verifying metadata only occurs after its ancestors are locked, but also locks the root dirnode. When the VFS operation is completed, the metadata are flushed and released from leaf-to-root to ensure that parent metadata are

updated after their children. Once the root dirnode is flushed, the lock is released to make it available for other users. This mechanism has a tradeoff between security and sharing; the user acquires strong integrity guarantees at the cost of concurrent volume modifications. We explore the performance implications for maintaining this hash-tree in our evaluation.

4.4.4 Optimizations

For every filesystem request, the NeXUS enclave fetches one or more metadata objects from the backing store to complete the operation. Because of the network latency, this makes metadata-intensive operations cost prohibitive. To address this, we introduced several caches to speedup data access, including 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 on the remote server, locally cached information can be used to fulfill filesystem requests.

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

By default, NeXUS commits all metadata changes to the network-backed storage. Although essential for consistency, the network cost is unnecessary for non-interactive workloads in which several operations are required to achieve the desired state. Examples include extracting compressed archives and cloning a git repository. Thus, we developed a variant of NeXUS that performs metadata I/O operations on a local directory, and not on the underlying network filesystem. This batching mode of operation can be activated at runtime by the user, who can readily commit the resultant filesystem state unto the server. This runtime setting is geared towards metadata-intensive workloads during which user interaction with the NeXUS volume is absent.

4.5 EVALUATION

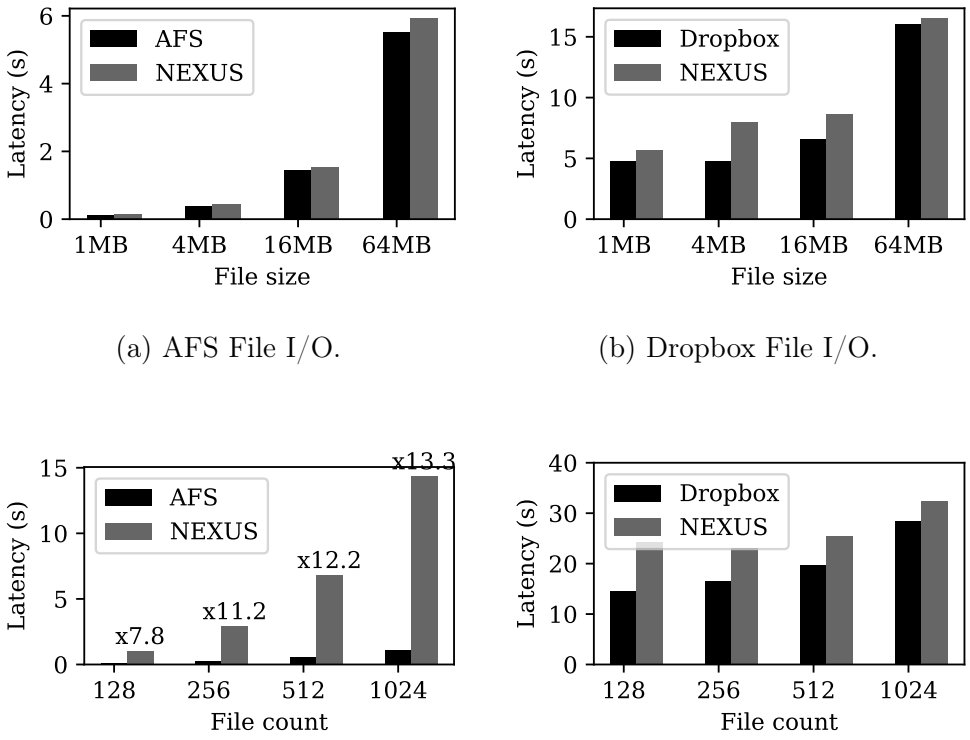
This section presents various performance benchmarks and real-world, end-to-end evaluation of NEXUS. To achieve the design goals stated in Section 4.3.1, we chose the following criteria:

- **Portability.** Does the NEXUS prototype support various network filesystems? What of user applications?
- **Performance.** Are the overheads incurred by NEXUS reasonable? How does it perform on normal user workloads? What of bulk metadata workloads? How is rollback protection affected by directory depth?
- **Efficient Revocation.** How cheap are user revocations when compared to purely cryptographic implementations?

Experimental Setup. We evaluate NEXUS on an i7 @3.4GHz with 8GB RAM and 128MB EPC, running SGX SDK 2.2 and Ubuntu 18.04 LTS. The experiments compare the overhead of NEXUS (and its variants) against unmodified network-based filesystems, namely AFS and Dropbox. Whereas AFS behaves like a full-fledged network filesystem by synchronously committing local changes to the server (on the same LAN), Dropbox monitors changes in a mounted folder and asynchronously uploads them to the cloud. In particular, although Dropbox tries to minimize network latency (e.g., batching, chunking, compression), it is sometimes affected by spurious delays and synchronization time takes longer than needed (e.g., during frequent file updates [105, 106]). For the NEXUS prototype, we used 1MB file chunks and 128-entry dirnode buckets. Moreover, our measurements are averaged over 10 runs.

4.5.1 Microbenchmarks

We ran two microbenchmarks to isolate the overheads imposed by NEXUS. We began with measuring the latency of basic file I/O operations using a python program that reads and writes a file at different sizes. Compared to AFS (Figure 15a) and Dropbox (Figure 15b), results show that NEXUS incurs a negligible overhead across all file sizes. This is because the runtime is dominated by file I/O, which is the same amount on each prototype. Although



(a) AFS File I/O.

(b) Dropbox File I/O.

(c) AFS directory operations.

(d) Dropbox directory operations.

Figure 15: Microbenchmarks comparing file and directory operations.

Workload	Total Size	Native	NEXUS	Overhead
AFS				
140 MP3s	150MB	2.91s	5.35s	×1.84
1 movie	650MB	9.35s	14.79s	×1.58
211 PDFs	315MB	7.61s	13.92s	×1.83
30 videos	1.6GB	24.78s	41.25s	×1.66
Dropbox				
20 MP3	20MB	10.33s	21.59s	×2.09
1 movie	650MB	86.45s	94.82s	×1.09
211 PDFs	315MB	96.92s	71.83s	×0.74
30 videos	1.6GB	180.19s	216.90s	×1.20

Table 3: Latency(s) for copying PDFs and videos.

the metadata I/O by NEXUS increases as the filenode grows to accommodate additional file chunks, this is still small compared to the file size (about 80B of filenode 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. Compared to AFS (Figure 15c), the overhead incurred by NEXUS increases with file count, meanwhile the overhead on Dropbox (Figure 15d) remains less than ×2. This is because every file created increases the size of the directory's dirnode, which becomes considerably bigger than individual directory entries. For large directories, this could result in significant performance overheads as the size discrepancy between the directory entry and the dirnode becomes more pronounced.

4.5.2 Macrobenchmarks

We evaluate the end-to-end impact of NEXUS on copying various workloads, including MP3s, PDFs and videos. To cover a wide range of user storage workloads, we vary the file count and size to generate a unique mix of directory and file operations. The results in Table 3 show that NEXUS incurs about a ×2 overhead on most workloads. In AFS, the 140 MP3s and 211 PDFs workloads incurred the highest overhead due to their larger directories. However, the

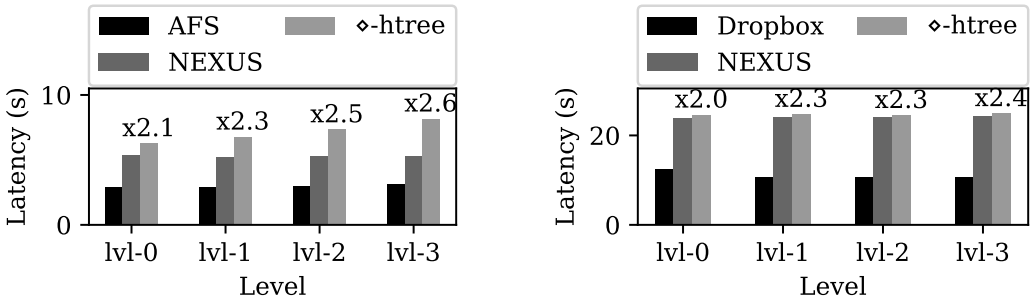


Figure 16: Copying 150 MP3s at different directory depths.

single 650MB movie and the 1.6GB video collection had the lowest overhead on both AFS and Dropbox. Because their runtime is dominated by file I/O, NEXUS processes them efficiently. Surprisingly, NEXUS is 25% faster than Dropbox in synchronizing PDFs. We reran this experiment multiple times, but it seems the workload's I/O patterns benefit NEXUS.

4.5.3 Rollback Protection Overhead

In this test, we measure performance impact of providing rollback protection. First, we extract the source code of the Julia programming language to initialize the directory hierarchy. Then, we measure the latency of copying a music collection of 140 files (155MB) at different directory depths. Figure 16 shows that on both AFS and Dropbox, the overhead incurred by rollback protection (i.e., ⋄-htree) increases slightly with directory depth, whereas that of plain NEXUS remains constant. This additional overhead is because ⋄-htree must lock and update multiple metadata objects on each filesystem operation. Moreover, the results show that this increase is relatively minimal.

4.5.4 Bulk Metadata Operations

We evaluate the performance of NEXUS on bulk metadata operations by cloning the git repositories of Redis (618 files) and Julia (1096 files). Git cloning requires a series of filesystem operations (e.g., `create`, `write`, and even `hardlink`) to download pack objects, and extract

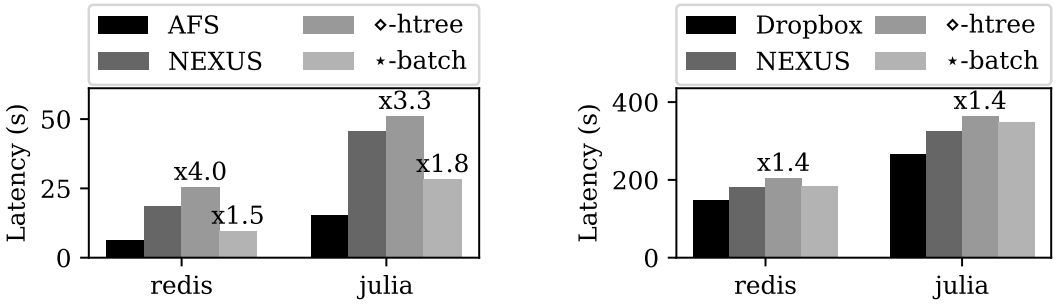


Figure 17: Git cloning of Redis and Julia.

them into constituent files and directories. Figure 17 shows the latency measurements. For both Redis and Julia, the NEXUS overheads on AFS and Dropbox are $\times 2.9$ and $\times 1.2$, respectively. Meanwhile, the ◇-htree prototype incurs slightly higher overheads as it requires additional metadata I/O to provide data freshness. However, when combined with batch mode (i.e., ⋆-batch), the overhead drops on both AFS and Dropbox. The is due to the fact that fewer filesystem operations are propagated to network, thereby reducing overall latency.

4.5.5 Revocation Estimates

NEXUS supports two types of revocation: (1) removing a user from the volume, and (2) removing a user's rights from a directory. In a typical cryptographic filesystem, revoking user access involves the following steps: re-encrypting the affected file(s), uploading the ciphertext to the server, and then distributing the new keys 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 211 PDFs workload mentioned above (Figure 16). For 315MB of file data, the NEXUS will have to re-encrypt and update about 653KB of metadata (recall access control is stored in the dirnode). Whereas, in the 30 videos workload, the metadata payload drops to 210KB for 1.6GB of file data.

Operation	AFS	NeXUS-AFS	NeXUS-FUSE
LevelDB			
Fillseq	10.5 MB/s	8.1 MB/s	8.5 MB/s
Fillsync	2.2 µs/op	4.5 µs/op	10 µs/op
Fillrandom	5.9 MB/s	3.7 MB/s	4.6 MB/s
Overwrite	4.0 MB/s	2.6 MB/s	3.4 MB/s
Readseq	664.6 MB/s	718.1 MB/s	673.9 MB/s
Readreverse	425.0 MB/s	425.7 MB/s	429.8 MB/s
Readrandom	2.27 µs/op	3.7 µs/op	2.3 µs/op
Fill100K	11.0 MB/s	7.2 MB/s	10.7 MB/s
SQLITE			
Fillseq	6.5 MB/s	6.4 MB/s	3.1 MB/s
Fillseqsync	14.4 µs/op	31.4 µs/op	37.3 µs/op
Fillseqbatch	70.2 MB/s	69.7 MB/s	49.5 MB/s
Fillrandom	4.2 MB/s	4.2 MB/s	2.2 MB/s
Fillrandsync	13.4 µs/op	31.2 µs/op	45.3 µs/op
Fillrandbatch	7.6 MB/s	7.7 MB/s	4.1 MB/s
Overwrite	3.4 MB/s	3.4 MB/s	1.8 MB/s

Table 4: Database benchmark results on AFS.

4.5.6 Comparing FUSE Overhead

We compare the performance of our FUSE-based prototype with another NeXUS implementation that required modifying the AFS client [107]. FUSE provides better portability and does not require any changes to the underlying filesystem. However on every filesystem request, FUSE incur overheads from transitioning and copying data between the kernel and userspace. Whereas, the AFS-based prototype *only* transitions from the kernel to our userspace NeXUS daemon on network API calls, but manages caching and other low level operations inside the kernel.

For this test, we ran the database benchmarks of LevelDB and SQLite, 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 4. Results show that both NeXUS variants performed similar to vanilla AFS on most

asynchronous operations. Because the database utility does not wait for the operation to complete, the effect of each filesystem call is amortized. However, synchronous operations incurred significant overhead on both prototypes, with NEXUS-AFS being significantly faster due to less transitioning and copying compared to FUSE. However, as shown by the previous tests, FUSE allows more general filesystem support, and does not require any changes to the OS or any other system components.

4.5.7 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 on writes, which we believe is acceptable in practice for the majority of users. We envision that users of typical file sharing platforms will mostly perform reads, and manage relatively small workloads such as text documents and photos [108]. Alternatively, NEXUS supports a batch mode that allows users to speed up the execution of bulk metadata workloads.

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: (i) The asynchronous rootkey exchange (Section 4.3.4) requires a single file write. (ii) Adding/removing users (Section 4.3.5) is not unlike revocation, which has been shown to require a single metadata update. (iii) Although policy enforcement (Section 4.3.5) scales with the number of ACL entries, its cost is dominated by the initial metadata fetch.

4.6 RELATED WORK

4.6.1 SGX-Enabled Storage

Since its release, SGX has generated considerable research aimed at achieving secure remote storage [34, 33, 36, 48, 109, 35]. LibSEAL [48] detects cloud provider integrity violations by creating a non-repudiable log of the service requests and responses. BesFS [110] provides a formally-verified filesystem API that protects enclave applications against Iago attacks. PESOS [34] enforces custom server-side access control on top of untrusted storage, but its prototype requires a LibOS [58] that severely impacts the TCB. SPEICHER [37] and eLSM [109] provide secure key-value stores. ZeroTrace [36] and OBLIVIATE [33] 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, while taking into account the practicalities of dynamic user sharing and seamless integration with existing storage services. SGX-FS [35] is an enclave-protected userspace filesystem, but does not provide any sharing capabilities.

IBBE-SGX [32] and A-SKY [111] propose a computationally efficient schemes for achieving scalable access control an enclave. However, unlike NEXUS, their access control models restrict all group membership operations to the administrator.

4.6.2 Cryptographic Filesystems

Starting with CFS [112], TCFS [113], NCryptfs [114], a long line of work has been dedicated to employing client-side encryption to secure remote untrusted storage [115, 116]. Most of them either assume a trusted server, or require deploying semi-trusted intermediaries. Latter systems such as [24, 26, 21] took a closer regard to file sharing, and proposed schemes that rely on clients to manage the encryption keys. Unfortunately, pure encryption techniques are plagued by issues of non-trivial key management and bulk file re-encryption on user revocation. These overheads could be considerable, even with modest policy updates [17]. Although mitigating schemes such as lazy encryption (delay file re-encryption until the next

write [117]) and proxy re-encryption (use a trusted server for key distribution [19]) 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.

Another line of research has focused on providing stronger integrity protections to remotely hosted data [116, 118]. In a rollback attack, the server returns stale versions of the data to the user. Fork consistency [116] is the strongest form of consistency achievable without direct client communication; it ensures that the filesystem state observed by the user is derived from a valid sequence of changes. Specifically, if the server equivocated and presented each client with a different view of their data, they can never see each other's further changes. If the clients were to communicate with each other offline, they will detect the equivocation. This requires the use of authenticated data structures (e.g., Merkle trees or hash chains) that provide efficient integrity-protection over a collection of items [100]. However, because of the network and concurrency costs involved in maintaining these structures, existing solutions either rely on server-side participation, use a byzantine fault-tolerance protocol, or restrict themselves to a single-user setting [115, 119, 120]. Verena relies on a trusted server to enable efficient integrity verification of webpages by clients.

4.7 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. As a solution, 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 sensitive data on the client and then attaches cryptographically-protected metadata that ensures the encryption keys are enclave-bound. Furthermore, NEXUS provides confidentiality and integrity of file data, as well as file and directory names. NEXUS also provides optional rollback protection

to detect when the server returns stale metadata while also enforcing access control at each user's local machine enabling file sharing through SGX remote attestation. Finally, we implemented a FUSE-based prototype that runs on top of AFS and Dropbox. Our evaluation shows that NeXUS achieves good performance in file I/O operations and incurs modest overheads on workloads that involved bulk metadata.

However, our current access control implementation only supports simple ACLs, which are evaluated using simple checks, and does not take into account the environmental context. Futhermore, traditional access control schemes do not take into account changes in the access context, especially as usage occurs on remote client machines and extend over large time periods. In the next chapter, we extend the NeXUS design to implement fine-grained access and usage control within a large-scale cloud environment.

5.0 JOPLIN

We present Joplin, a secure access and usage control system that provides confidentiality and integrity on top of existing cloud storage systems. Joplin leverages trusted hardware to address the practical challenges of attribute-based access control, including efficient revocation, multi-domain policies, and secure user delegation. This is enabled by a client-side enclave that hosts a policy interpreter, and applies cryptographic protections without leaking encryption keys. Joplin embeds user-specified policies within encrypted metadata, which are decrypted within the enclave for continuous enforcement at access time. We implemented a prototype by extending our stackable filesystem, NEXUS, to enable fine-grained sharing of protected volumes without requiring any server-side changes. Our prototype imposes minimal key management on users, supports a declarative policy language, mandatory access logging, and ensures both forward and backward secrecy. Using microbenchmarks and example cases, we demonstrate that our prototype can enforce a wide range of user policies, and provides scalable policy management for up to 25,000 policies over Dropbox.

5.1 INTRODUCTION

Today, mainstream file sharing services boast hundreds of millions of daily users, and store billions of gigabytes in files [2]. Built on top of low-cost and globally available cloud storage, these services provide a convenient interface to store, share, and synchronize files online. However, the sale and unauthorized sharing of private data by service providers, as well as frequent data breaches and unplanned disclosures on the cloud, have raised user privacy concerns [9, 10, 11]. As individuals and organizations store larger amounts of personal and

private information on these platforms, it is evermore critical to provide a data security solution that does not rely on the service provider.

Beyond providing data confidentiality and integrity, users may also wish to selectively share information with other users within or across organizations. Furthermore, users typically share information under a given context or towards a particular purpose. Thus, addressing these security and privacy concerns requires: (i) *access control* [121, 74] to determine the conditions for user authorization, and (ii) *usage control* [122, 82, 84] to restrict how data may be handled post authorization. Usage control extends traditional notions of data access control with continuous enforcement, and obligations such as: "delete this file in 15 days". However, developing a robust enforcement mechanism for the cloud poses several challenges: (i) a large number of users accessing an unbounded number of resources with varying granularity; (ii) dynamic access changes as users and files are continuously added, removed and updated; and (iii) information sharing across administrative domains such that user identity is not known in advance.

This chapter explores the subtleties of distributed policy enforcement by focusing on Attribute-Based Access Control (ABAC), a flexible and dynamic access control model that can be extended to also support usage control (UCON) [74, 123, 84]. ABAC generalizes previous identity-based schemes, and evaluates access requests using attributes that are assigned to users, objects, and the environment. For example, an educational institution may have "*Professors* can access *Faculty* directories", as a policy to restrict access to department files. Attributes simplify policy management, especially when user or object entities are not known a priori (e.g., multi-domain scenarios), and contextual information (e.g., time, location) is required for multi-factor access evaluation. This also reduces the administrative burden, as users and objects can be continuously provisioned without requiring any updates to existing policies.

We propose a practical security solution that: (i) allows for user deployment without requiring any server-side coordination or reliance on trusted intermediaries, (ii) supports user sharing with fine-grained and dynamic access control policies, and (iii) incurs modest performance overheads on typical user workloads. In addition, our solution must provide a foundation for rich sharing semantics such as multi-authority attributes, delegation, and

usage control. Our goal is to protect the confidentiality and integrity of a user's data against all untrusted parties, including unauthorized users, external attackers, and even the service provider.

Existing approaches to secure access and usage control, however, fall short of satisfying these requirements. Much work on usage control has focused on obligation languages, formal specifications, and reference architectures, but often make simplified assumptions about the client-server architecture [86, 87, 88]. Several cryptographic schemes such as Attribute-Based Encryption (ABE) have been proposed to achieve fine-grained access control, while protecting sensitive data from untrusted parties [124, 125, 126]. However, pure cryptographic solutions exhibit significant overheads on user revocation, and oftentimes rely on server-side support for dynamic access changes. Alternatively, Trusted Execution Environments (TEEs) such as Intel SGX or ARM Trustzone can be used to enable strong security primitives within an untrusted environment. However, existing approaches either restrict themselves to centralized administration on a single machine or require server-side support for trusted hardware, which in effect hinders adoption by users of cloud-based services [32, 111, 34].

We present Joplin, a privacy-preserving ABAC enforcement system that guarantees confidentiality and integrity over unmodified third-party storage platforms. Joplin enables users to manage attributes and specify declarative access policies separately from the underlying storage. By adopting a declarative syntax, policies can unambiguously express the conditions under which a user has authorized access, without requiring a priori knowledge of particular user/object entities nor details about the enforcement mechanism. To accomplish this, Joplin leverages a client-side SGX enclave that stores access control information within cryptographically-protected metadata for enforcement at decryption time. Because the enclave performs cryptographic operations without leaking the encryption key, updating attributes and policies only requires re-encrypting relatively small metadata objects. This leads to efficient revocation even in large-scale dynamic environments.

Joplin is based on the NEXUS stackable filesystem, is completely client-side, and does not require any server-side changes or trusted intermediaries. Policy enforcement exclusively occurs on each client machine, which collectively realize user-centric and decentralized access control. Further, placing our reference monitor at the client provides direct access to the

client's environment when enforcing usage policies. To demonstrate the practicality of Joplin, we extended NEXUS, a stackable filesystem that allows mutually trusting users to securely share files on top of unmodified storage. Joplin's modular architecture is split into (i) an enclave that applies cryptographic protections and hosts a virtual filesystem alongside a Joplin Controller, and (ii) an untrusted portion that maps filesystem and ABAC operations from the enclave unto the underlying storage platform. This approach enables independent policy enforcement, while transparently supporting a wide range of storage services such as remote filesystems and object-based storage platforms.

Our contributions are summarized as follows:

- We propose a client-side architecture for enforcing fine-grained ABAC and usage control over remote untrusted storage. Joplin provides a unified interface to manage attributes and policies, while ensuring data confidentiality and integrity by performing cryptographic operations and access control enforcement inside the enclave without leaking sensitive key material. With Joplin, policy management on existing storage platforms is independent of the service provider or any trusted intermediary.
- We develop a prototype that addresses the practical challenges of decentralized policy management on the cloud without requiring any server-side modifications or additional infrastructure. This includes attribute sharing across domains, user delegation, and mandatory access logging. Additionally, Joplin provides efficient revocation with forward and backward secrecy, while maintaining complete ownership throughout the data lifecycle.
- We evaluate our prototype using several case studies and filesystem workloads. Results show that our implementation enforces declarative policies efficiently, operates over unmodified Dropbox, and has policy maintenance overheads that scale with the number of users and policies. Joplin allows both individuals and organizations to manage and protect their data without significant degradation on typical user workloads.

The chapter is organized as follows. Section 5.2 covers background and related work on access and usage control. In Section 5.3, we describe the high-level design of our ABAC system, and Section 5.4 realizes a prototype from this design. In Section 5.5, we evaluate the expressiveness and performance of our prototype. Section 5.6 concludes the chapter.

5.2 BACKGROUND AND RELATED WORK

This section presents a brief description of ABAC, usage control, decentralized policy management, and trusted hardware.

5.2.1 Attribute-Based Access Control

In ABAC [74, 123], policies are expressed as boolean combinations of attributes, and access decisions are determined by evaluating policies against user attributes, object attributes, and relevant environment conditions. Attributes denote traits that either statically assigned by an administrator (e.g., user role) or dynamically set by the runtime (e.g., file path). This simplifies policy management as attributes can arbitrarily group entities with flexible granularity, without requiring the apriori identification of individual users or objects. Dynamic access control is provided by simply updating attribute values, which may change the access decision between requests. This has allowed ABAC to supplant prior identity-based access models, including DAC, MAC, and even RBAC [75].

The related work on ABAC can be broadly grouped into: (i) *Formal models* that define the basic ABAC elements (i.e., users, objects, attributes etc.) along with their relations and constraints [75]; (ii) *Policy Languages* to express authorization rules [77]; and (iii) *Attribute-Based Encryption (ABE)* for fine-grained cryptographic access control using a many-to-one public encryption scheme [125, 126]. In ABE, an administrator generates keys in a tree-like structure and assigns them to users, such that decryption is only possible if the user has the necessary attributes (KP-ABE) or satisfies the ciphertext access policy (CP-ABE).

Although ABAC offers a promising solution for protecting cloud applications, practical instantiations either consider static scenarios, or oftentimes rely on a semi-trusted server for dynamic access control [124, 125, 126]. Whereas extensive research has focused on data security with ABE, implementations offer limited expressiveness in supporting environment attributes, and incur severe computational overheads on user revocation. By requiring bulk file re-encryption and key redistribution, the revocation cost is proportional to the number of

affected user files and their degree of sharing. Given that we target a dynamic environment where access policies and user memberships may change frequently and unpredictably, we address these shortcomings without any server-side coordination, while providing efficient user revocation, as well as fine-grained and context-aware access control.

5.2.2 Usage Control

Usage control (UCON) is a generalization of access control that considers not only who can access data, but also how this data may be used or distributed in the future [82, 83, 84]. UCON is an attribute-based model centered on two aspects: mutable attributes and continuous enforcement. Mutability recognizes that user, objects, or environment attributes may change while access is in progress. Continuous enforcement ensures that policies are evaluated before, during, and after the usage period, and may terminate usage in the event of a violation. Besides attribute-based authorization, a UCON policy also includes obligations and conditions. Obligations are actions or requirements (e.g., "delete file after 20 reads") that must be fulfilled throughout the usage, whereas conditions are environment restrictions (e.g., time) validated before and during usage. UCON can encompass traditional access control models, trust management, and even DRM [82].

Prior research on usage control has mostly focused on obligations and reference enforcement architectures [86, 87, 90]. Distributed usage control is the ability of a data owner to restrict remote remote usage by requiring enforcement on every machine that stores, processes, and distributes sensitive data [91]. Depending on the application, enforcement may occur either on the server, the client, or a combination of both. Whichever the case, the monitoring on the client must be tamper-proof, impossible to circumvent, and verifiable in enforcing access and usage policies. However, in addition to privacy issues, centralized scenarios also incur significant client–server communication overhead, and require the server to be always online. Other works make trustworthiness assumptions of client-side components to ensure data confidentiality and integrity [86]. Hardware-based approaches have been proposed, but they either rely on expensive hardware or do not provide isolated execution [92]. We address these data security issues by leveraging commodity trusted hardware to provide scalable access

and usage control within a distributed file sharing environment.

5.2.3 Decentralized Policy Management

In large scale systems with many users and files, requiring centralized administration for every access right may be unbearable and even obstruct user collaboration. Due to the flexible and dynamic nature of ABAC, we identify two important features that could serve as key enablers for upcoming generations of decentralized access control systems: multi-domain policies and delegation.

Multi-domain policies. Typically, administrators in ABAC define the set of attributes within a given domain. Attributes can only be assigned to users/objects within that domain, and presumably, system policies are only expressible in terms of those attributes. However, there exist situations where policy specification could include attributes from other domains. For instance, consider a group of researchers who want to share resources, but their respective organizations have no common trusted root authority. Each organization maintains complete control their attribute definition and policy specification, but want to selectively share a subset of their resources, while ensuring safety from inappropriate access. Trust Management and Trust Negotiation are techniques for establishing trust within an open environment using attributes [127, 128]. We offer similar attribute-based access control with multi-domain support using commodity trusted hardware and distributed reference monitors, as opposed to prior implementations that require special infrastructure to distribute credentials.

Delegation. This is the ability of a *delegator* to selectively their transfer rights unto a *delegatee* [129]. The delegatee is granted temporary access on behalf of the delegator, who can revoke the transferred privileges at anytime. For example, a professor could delegate "maintainer" privileges for their teaching assistant to manage a submission directory, before revoking them at the end of the semester. However, user delegation within a distributed environment poses several challenges, including timely revocation and using fresh attributes at evaluation time. In addressing these issues, secure delegation offers a flexible mechanism for discretionary access control between trusting parties, without leakage of private credentials.

5.2.4 Hardware-Assisted Access Control

Several works have proposed using TEEs to enable secure access control. IBBE-SGX [32] proposes a computationally efficient IBBE scheme and A-SKY [111] addresses the impracticality of anonymous broadcast encryption. However, both restrict all group membership operations to an administrator that hosts the enclave. PESOS allows users to specify per-object access policies, which are enforced by policy-based engine within the enclave [34]. Like Joplin, the policies are written using a declarative syntax. However, it requires server-side support for trusted hardware as well as specialized Kinetic storage drives, which at the time of writing are not readily available on cloud services. In this paper, our approach requires no server-side modifications, while providing a decentralized enforcement and rich user sharing.

5.3 SYSTEM DESIGN

We now present Joplin, a secure access and usage control enforcement system for untrusted storage platforms. After outlining our design goals, we describe the client-side approach, ABAC model, policy language, obligations support, and high-level architecture. Altogether, this provides a solid foundation for developing fine-grained and scalable attribute-based user sharing within existing cloud-based environments.

5.3.1 Design goals

Similar to NeXUS, our design provides a balance between practicality and portability with the following requirements:

- **Practicality:** The number of keys managed by users should not increase with group membership or permitted objects.
- **Portability:** The system must not require any server-side modifications or trusted intermediaries, and should be compatible with commodity cloud storage platforms. We rely on the underlying filesystem interface for metadata storage and distribution.

- **Scalability:** Access control enforcement must scale up to thousands of users and policies. Likewise, revoking users and updating access control information should be efficient without inducing a cascade of metadata updates.
- **Rich semantics.** Beyond providing fine-grained access and usage control, the system must provide a secure foundation for decentralized policy enforcement, including user delegation, usage control, and cross-domain attribute sharing.

We opt for a small TCB and minimal interface so that library applications and secure filesystems (e.g., NEXUS [107]) can easily incorporate our solution for secure ABAC enforcement. Furthermore, the distribution of generated metadata should not require the deployment of additional services; instead our solution should allow users to employ their existing storage platform. Hence, we minimize storage assumptions and adopt a lightweight solution that stores metadata as files, such that integration with various storage systems is possible with moderate effort.

5.3.2 Client-side Approach

Reference monitors are the standard mechanism for continuous enforcement of access and usage control policies in traditional systems [130]. As part of the TCB, a reference monitor observes all accesses to sensitive objects, evaluates requests against security policies, and undertakes corrective action in the event of a violation. A straightforward solution would be to develop a server-side monitor that mediates all user requests against a centralized repository. This could provide fast and granular data access and provide a global scope for enforcing policies. Because our threat model considers the server as untrusted, one could employ trusted hardware to prevent malicious providers from subverting the monitor's operation. However, this requires trusted hardware support on the server, which at the time of writing still is not widely available amongst cloud providers. Alternatively, we can treat the cloud as an opaque storage layer with a client-side approach that intercepts and enforces access requests at the consumer end. To prevent unauthorized users from accessing or tampering sensitive information, this approach requires data distribution in encrypted form. Although purely cryptographic techniques like ABE provide data security, they do not

readily support environment attributes, which are necessary for multi-factor authorization and obligations enforcement.

In this work, our approach is to execute the reference monitor on individual client machines and utilize TEE primitives (Section 5.2.4) for decentralized access control. Isolated execution enables tamper-proof access enforcement and cryptographic protections, whereas Sealed storage and Remote attestation allow the persistence of cryptographic material across valid trusted hardware platforms. We ensure confidentiality and integrity by storing user-provided access policies within encrypted metadata files, and enable seamless key distribution by securely attaching key material that is only accessible inside the enclave. As a result, cryptographic material remains under the control of the enclave, which in turn independently enforces access and usage control. Given the widespread availability of trusted hardware on commodity machines, our user-centric solution can be easily deployed by individuals and organizations of existing remote storage services, without relying on a centralized server or additional infrastructure.

Although trusted hardware may alleviate the computational and network overheads associated with pure cryptographic schemes, building a practical access control system presents several challenges. Because continuous enforcement relies on multiple components, performance depends on their complexity and degree of inter-communication. Therefore, beyond providing an access control model and a policy language, the enforcement architecture must also support efficient user revocation, manageable administrative costs, as well as interoperability across domains.

5.3.3 ABAC Model

This work focuses on ABAC, a flexible and dynamic model which uses attributes to separate privilege assignment from policy specification; user, object, and environment attributes can be provisioned and managed in a decentralized fashion, whereas policies arbitrarily combine attributes in a fine-grained manner. This makes ABAC particularly well-suited for providing access and usage control within an open environment. We build on ABAC_α, a foundational model with minimal assumptions on administrative control or system attributes [75]. Formally,

Command	Meaning
add_user/remove_user	Add or remove user
add_object/remove_object	Add or remove object
add_attribute/remove_attribute	Add or remove user/object attribute
grant_attribute/revoke_attribute	Grant or revoke user/object attribute
add_policy/remove_policy	Add or remove policy rule

Table 5: Joplin Administrative ABAC commands

our ABAC model is described by the tuple ⟨U, O, UA, OA, UAA, OAA, PERM, Po, Auth⟩ as follows:

- The set U contains identities for all users in the system, whereas O refers to all the protected objects.
- User attributes (UA) and object attributes (OA) are the sets of all attributes that can be assigned to users and objects, respectively. The relations UAA (U × UA) and OAA (O × OA) are user-assigned and object-assigned attributes, respectively. Assigned attributes have an optional integer or string value.
- Permissions (PERM) are possible system actions or privileges, which include read, write, create, delete, and audit. At runtime, the audit permission is enforced as an obligation that records the current access request unto a log file.
- Policies (Po) are the set of policy rules representing the conditions under which an access is authorized. The syntax is described by our policy language in Section 5.3.4, whereby each rule maps a given permission to combination of user and object attributes.
- The authorization function (Auth) that takes a request $(perm, u, o)$ and the current access state ⟨UAA, OAA, Po⟩ as inputs, and returns true if the user/object attributes satisfy a system policy with the matching permission.

Administrative model. With regards to our system model, the volume owner is in charge of administering access control information within a given domain, but is not required to mediate access requests. Listed in Table 5, administrative commands include attribute

Predicate	Meaning
Dynamic Predicates	
@uname(U, x)	Check user name
@upubkey(U, x)	Check user public key
@oname(O, x)	Check object name
@opath(O, x)	Check object full path
@oversion(O, x)	Check object version
@ocreator(O, U)	Check for the object creator
Boolean Predicates	
_eq(x, y)	x=y
_ne(x, y)	x≠y
_gt(x, y)	x>y
_ge(x, y)	x≥y
_lt(x, y)	x<y
_le(x, y)	x≤y

x and y are constants (i.e., a string or number)

Table 6: Predicates in Joplin's predicate language

management, user membership, and policy specification. Each command takes the current system state ⟨U, O, UA, OA, UAA, OAA, Po⟩ with arguments, and transition unto a new state ⟨U′, O′, UA′, OA′, UAA′, OAA′, Po′⟩. In our implementation, we extend this model to provide flexible administrative and sharing paradigms.

5.3.4 Policy Language

Joplin provides a declarative policy language for protecting confidentiality and integrity with an ABAC environment. The language is based on Datalog, a logic-based programming language with a simple syntax, concise semantics, and efficient computation over large datasets [131]. A Datalog program consists of facts that assert relevant traits, rules that deduce facts from other facts, and queries that verify the existence of facts. Rules are expressed as Horn clauses with the form 'L_0 :- $L_1, ..., L_n$', where L_0 is the head and each L_i is a literal with the shape 'predicate(term,...)'. Adopting a declarative approach not only

provides fine-grained and context-aware expressiveness, but also ensures that policies have a precise meaning that is independent of the enforcement mechanism or the identities of existing user or object entities [34].

Essentially, Joplin policy rules are authorization functions that grant specific permissions by evaluating user (U) and object (O) attributes with predicates. For example, to authorize "read if the user is a student and the object is the CS449 book":

```
read :- isStudent(U), book(O, "CS449")
```

Our policy language also supports predicates that encode dynamic elements of a system. Shown in Table 6, Joplin provides 3 types of predicates: (i) static predicates that capture assigned attributes; (ii) dynamic predicates that capture runtime information; and (iii) boolean predicates that can compare attribute values. Predicates take two terms, but static predicates can omit the second argument to check for attribute existence. Both static and dynamic predicates are tied to user or object entities, but the latter checks for information that is not explicitly set by the owner (e.g., user location, object version). For example, "write if the user's role attribute is faculty and file name is foo.md" translates to:

```
write :- role(U, "faculty"), @oname(O, "foo.md")
```

Another example, "write if the user is a student and object's version is below 30" becomes:

```
write :- isStudent(U), @oversion(O, X), _lt(X, 30)
```

Lastly, our language also allows policies to capture attributes that belong to other domains. These are similar to static predicates, but their identifier is prefixed with the domain's namespace. Assuming a user wishes to capture attributes from a domain named 'mint' and restrict read access to 'mint' employees:

```
read :- _mint_employee(U), @oname(O, "secret_docs")
```

At runtime, user and object attributes are copied as facts into the Datalog reasoner, which evaluates each authorization request against the system's access and usage policies.

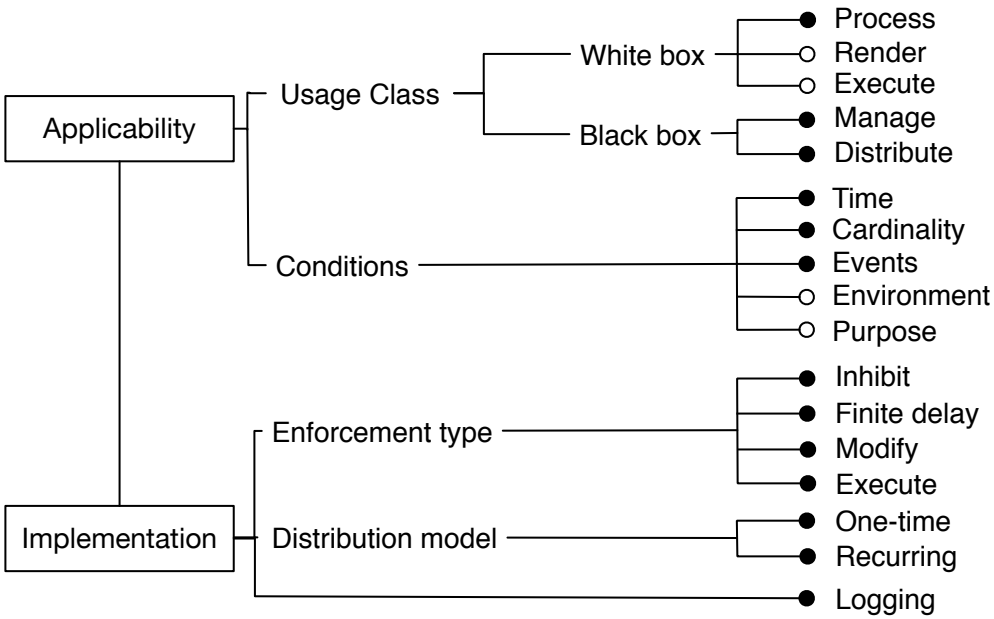


Figure 18: Taxonomy of Obligations from prior work focusing on Applicability and Implementation [132]. Black circles can be readily supported by Joplin.

5.3.5 Support for Obligations

Obligations are mandatory actions or requirements that have to be satisfied as part of an object's usage [82, 133, 134]. Depending on the specific scenario, distinct obligations policies can dictate how data may be used or distributed beyond authorization. However, our client-side architecture and use of trusted hardware restricts the scope of enforceable obligations. For example, SGX does not allow direct access to system facilities such as time, system calls, or library functions. Therefore, we survey how obligations enforcement has been employed in prior work, and explore how different aspects apply to our design [132, 135, 85]. Shown in Figure 18, we focus on Applicability and Implementation, leaving out criteria such as License and other non-functional properties (e.g., cost).

Applicability deals with what is expressible within a policy, including the usage class and conditions. There are two usage classes: black box usage where data is simply managed and distributed as bytes, and white box usage that involves further data processing, rendering, or execution. *Conditions* define the environmental context: (i) Time denotes when to fulfill the obligation; (ii) Cardinality is to how many times an action occurs; (iii) Events result from specific system operations; (iv) Environment refers to organizational, technical, and physical requirements; and (v) Purpose are human-specified rules. Conditions are evaluated before and during usage, and can be combined to form expressive policies; e.g., "delete file by 20 days or after 20 reads" uses both time and cardinality.

On the other hand, *Implementation* is concerned with the types of enforcement, distribution model, and logging capabilities. Enforcement types include: (i) inhibiting the attempted usage, (ii) introducing a finite delay (e.g., waiting for a server response), (iii) modifying usage data (e.g., down-sampling images on mobile), or (iv) executing actions (e.g., deleting an attribute). Lastly, the distribution model is whether the usage can be enforced recurrently or just once, whereas logging is the ability to securely record system operations.

As described in the literature, obligations enforcement can be expressed in terms of controllability or observability [133]. Controllable obligations are executed either as part of internal system actions (e.g., deleting a file), or require communication with an external component that guarantees fulfillment. As such, our architecture supports black box usage

fully, while providing partial support for white box usage via in-enclave processing (e.g., image compression). For conditions: (i) time can be incorporated by validating the responses from a trusted time service with a public key pinned inside the enclave [28, 136], (ii) cardinality such "homework file can only be read *twice*" could be enforced by counting the number of read entries in a log file, and (iii) events could be detected from internal system operations. For different enforcement types: inhibition, finite delay, modifications, and execution of actions can be supported if performed within the enclave, although finite delay will require a fine-grained time source [136].

Observable obligations provide a weaker notion in that fulfillment is only verifiable because they either require human intervention, or interact with an external component that does not provide strong enforcement guarantees. For example, although we could readily support the detection of platform specifications (e.g., CPUID), integrating physical aspects such as location or external technologies (e.g., firewall presence) requires trustworthiness assumptions. This also applies to white-box usages such as rendering to a screen and executing another program. Purpose restrictions (e.g., for private use only) and license agreements cannot be readily enforced by our system.

Non-observable obligations are neither executable nor verifiable. This usually involves obligations whose enforcement occurs in the future (e.g., notify the user in 2 days). However, non-observable obligations can be made observable through logging capabilities that could later be examined to discover any potential violations [133]. To this purpose, we implemented an audit mechanism that leverages the enclave runtime to record access authorizations inside an encrypted log file (Section 5.4.3).

5.3.6 High-Level Architecture

The central component of Joplin is a Controller that runs inside an enclave. Joplin unifies policy enforcement and data security by mapping user/object identities to attributes, while internally managing access control information within metadata. To provide confidentiality and integrity, the enclave encrypts metadata using enclave-resident keys, before sending the ciphertext to the server. This ensures that metadata is only decryptable within a valid Joplin

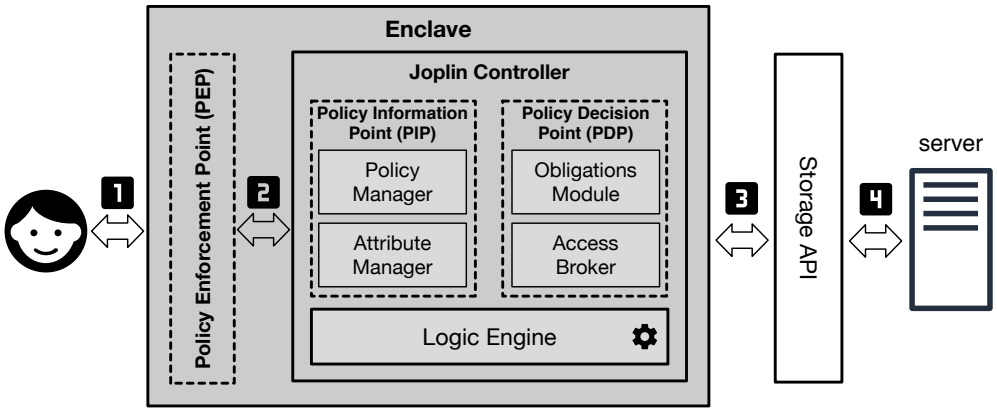


Figure 19: Joplin High-Level Design

enclave, which utilizes trusted hardware protections to independently enforce the embedded access and usage control policies.

Depicted in Figure 19, Joplin exposes an interface for administering access control and providing continuous enforcement. To accomplish this, Joplin coordinates the following subcomponents: (i) an Attribute manager that handles user, object, and environment attributes; (ii) a Policy manager that manages access and usage control policies; (iii) an Access broker that determines user authorization; (iv) an Obligations module that performs usage requirements; (v) a logic engine that evaluates access requests using system policies and relevant attributes. The Attribute and Policy managers collectively serve as the Policy Information Point (PIP), whereas the Access broker and Obligations module refer to the Policy Decision Point (PDP). The PIP provides an interface for retrieving, subscribing, and updating attributes, as well as policies. This includes user attributes from other domains and delegated attributes from other users. The Policy Enforcement Point (PEP) mainly intercepts and forwards access requests from the user, and enforces the PDP's authorization/obligation decision.

On every access request, the Controller first invokes the PIP to fetch the necessary attributes and policies. This involves interacting with the Storage API to fetch the necessary metadata, which are then decrypted and verified inside the enclave before use. After updating

the internal access control state (i.e., attributes, policies, and partial results), the Controller invokes the PDP to evaluate the request. The Access broker communicates with the logic engine to determine if the user has the requested permission over a given object. Likewise, the obligations module evaluates usage policies and performs obligation actions, which may include updating attributes via the PIP or aborting the request. Finally, after re-encrypting and committing every updated metadata object to the storage API, the Controller returns the PDP decision to the PEP for enforcement. Thus, by managing access control information inside encrypted metadata, Joplin essentially acts like a transparent security layer over unmodified storage.

5.4 IMPLEMENTATION

We now describe the implementation of a Joplin prototype that enforces fine-grained access and usage control within a secure filesystem environment. We extend NEXUS filesystem interface with an access broker that uses attributes and policies to evaluate authorization requests. While taking into account the architectural limitations of trusted hardware, Joplin provides, efficient revocation, and decentralized policy administration with multi-domain support and user delegation.

5.4.1 Metadata

ABAC enforcement within a filesystem requires storing user attributes for each volume user, object attributes for each file and directory, as well as policy rules for the volume. In addition, access control data must be confidential, integrity-protected, and consistent with filesystem information. Depicted in Figure 20, Joplin manages a flat namespace comprising of three metadata types: attribute space, policy store, and assignment table.

- **Attribute space:** Defines the set of all user and object attributes in a volume. It contains a list of (name, type, `UUID`) triplets called schemas, where type is either user or object. The volume owner can add or remove attributes at any time. To prevent name aliasing,

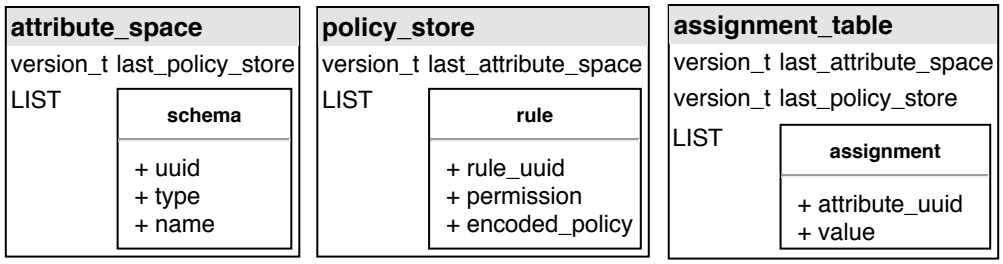


Figure 20: Joplin Metadata Structures.

each attribute is assigned a randomly generated `UUID` at creation time. The attribute space also tracks the last known metadata version of the policy store.

- **Policy store:** Lists all the policy rules in the volume. Each policy rule maps a permission to an encoded string representation. Rules can be added and removed by the volume owner. The policy store also stores the last known version of the attribute space.
- **Assignment table:** Contains the list of assigned attributes for a particular user or object (file/directory). Each entry consists of an (attribute_uuid, value) tuple, where attribute_uuid points to the schema inside the attribute space. The volume owner can grant, revoke, and update assigned attributes at any time. In a similar fashion, the assignment table tracks the last known versions of the attribute space and the policy store metadata.

The attribute space contains the set of user and object attributes (UA $\cup$ OA), the policy store contains the set of all policies (Po), and the assignment table handles individual user-assigned ($\in$ UAA) and object-assigned ($\in$ OAA) attributes. We integrate access control information with the NeXUS VFS by: (i) extending the supernode with the `UUID`s of the attribute space and policy store to load them at system startup, (ii) inlining assignment tables within individual filenodes and dirnodes to store object-assigned attributes, and (iii) creating an assignment table for every user entry in the supernode to store user-assigned attributes. This ensures that file/directory access control information is provided alongside with filesystem information during access requests, and that attribute/policy changes require only a fixed number of metadata updates. For example, attribute revocation only requires

deleting the entry from the corresponding user or object assignment table. Likewise, removing a user from the volume consists in deleting their entry inside the supernode, along with their user assignment table metadata.

5.4.2 System Initialization

In a typical workflow, a user first mounts their volume on a local directory and begins accessing its contents. However, the enclave must first authenticate and load the user's credentials. This includes directly assigned attributes within the local volume, attributes from other domains, as well as delegated attributes. Assuming that the volume rootkey has already been exchanged amongst the parties (Section 4.3.4.1), we now describe how Alice mounts a volume owned by Owen.

User authentication. The supernode stores the identities of the volume owner and other authorized users as (name, public key, `UUID`) entries. For user authentication, the enclave: (i) decrypts/verifies the supernode metadata to ensure it has not been tampered with, (ii) confirms Alice has been granted to the volume by checking her public key inside the supernode, and (iii) establishes that Alice owns the matching private key through a signature verification on a nonce [107]. On success, the enclave uses the metadata `UUID`s inside the supernode to fetch the attribute space, policy store, and Alice's user assignment table. After each metadata is decrypted and verified, the Attribute and Policy managers load the attributes and system policies inside enclave memory.

Cross-volume attributes. To support user attributes from multiple domains, we provide an interface to attach and detach volumes at runtime. We extended the supernode with a list of foreign volumes, such that each (supernode_uuid, namespace) tuple maps a foreign volume unto a unique namespace. Let us assume Owen adds the MintCorp volume with "mint" as the namespace, and Alice is a MintCorp user that wishes to load her foreign attributes. Similar to user authentication, attaching the foreign volume involves: (i) decrypting and verifying the MintCorp supernode, (ii) validating the MintCorp supernode `UUID` is a foreign volume within Owen's supernode, and (iii) confirming the presence of Alice's public key in the MintCorp supernode. To complete the process, the MintCorp attribute

space and Alice's foreign user assignment table are loaded into the enclave. The Attribute Manager maintains a map of volume namespace and the corresponding user attributes.

Delegated attributes. We provide an interface for delegators to select attributes from their user assignment table, and transfer them unto a delegatee. This enabled by a delegate file, a metadata object that contains the public keys of the delegator and delegatee, as well as the `UUID`s of the delegated attributes. To track the delegate files issued by each user, we also extended the user assignment table with a list of delegate file `UUID`s. Assume that another user, Carey has delegated the "maintainer" attribute to Alice. Specifically, Alice is in possession of a delegate file, and Carey's user assignment table contains the delegate file `UUID`. With this, the enclave loads Alice's delegated attributes by: (i) decrypting and verifying the delegate file metadata, (ii) fetching Carey's assignment table, and (iii) checking the presence of the delegate file inside Carey's assignment table. To complete the process, the Attribute Manager adds the delegated attributes and Carey's assignment table to a list.

With initialization complete, the logic engine can begin pre-computing intermediary results using the loaded system policies and user attributes. This pre-caching of results enables efficient query evaluation at access time. However, although user attributes are expected to change rarely, dynamic enforcement requires refreshing the enclave state to ensure correct access and usage decisions at evaluation time. Therefore, we measure the query evaluation latency and the refreshing cost in our evaluation.

5.4.3 Enforcing Access Policies

Once mounted, users can employ their applications to issue filesystem requests within the volume. To ensure that only authorized users can access particular files and directories, this section describes how the Joplin Controller orchestrates metadata and various components on every filesystem request. The core functionality enabling continuous enforcement is a logic engine that reasons over a knowledge base inside trusted memory.

The knowledge base represents an abstract datalog program, containing a subset of the access control state with up-to-date *facts* and *rules*. Each fact is a $\langle \texttt{UUID}, predicate, value \rangle$ triple, which asserts that a user/object entity has a given predicate and an associated

optional value. Facts are of two types: assigned facts that are extracted from the user/object assignment table, and contextual facts that are derived from system runtime information (e.g., file size, user name). More specifically, assigned facts comport local, foreign, and delegated attributes. On the other hand, rules are extracted from the volume's policy store. To detect stale facts and rules, the knowledge base also tracks the metadata version for each cached user/object entity, as well as the policy store.

5.4.3.1 Preprocessing From the previous subsection, consider Alice, who after mounting Owen's volume, has attached the MintCorp volume and loaded the delegate file from Carey. Assume that Alice now requests to read the '`bar/cake.c`' file. This is intercepted by the NEXUS VFS, which traverses the path to retrieve the filenode metadata. The Joplin Controller is then invoked with the read permission and the filenode containing the object assignment table.

- First, the Attribute and Policy managers retrieve metadata to refresh Alice's attributes and polices, including the metadata of every mounted and attached attribute space, Alice's local and foreign assignment table, Carey's assignment table (for the delegated attributes), and the policy store. For each metadata, we check the knowledge base for staleness and retract the corresponding facts/rules if the metadata is considered newer.
- We use the Attribute Manager to generate assigned facts from the user and object assignment tables. For each ⟨attribute_uuid, value⟩ entry of the assignment table, the attribute_uuid is used to lookup its corresponding name inside the attribute space (see Figure 20). The attribute name is then conjoined with the user/object metadata `UUID` and attribute_value to form an assigned fact. For example, Alice's attribute table may return $\langle role, student \rangle$, $\langle city, pittsburgh \rangle$ as assigned facts. The process is repeated for the foreign and delegated user attributes as follows:
  - For each foreign assignment table, we prefix each attribute name with the namespace of the corresponding volume. For example, if Alice has $\langle role, employee \rangle$ attribute assignment in the Mint Corp volume, the corresponding assigned fact will be $\langle \_mint\_role, employee \rangle$. By prefixing attributes with their respective namespace, the

logic engine can differentiate between attribute domains without requiring any changes to the existing reasoning algorithm.

  - For each delegate file, (i) we check its `UUID` is still present in the delegator's assignment table, and (ii) use its delegated attribute `UUID`s as a mask over the delegator's assignment table. This ensures that the delegate file has not been revoked, and the reasoning process uses up-to-date attributes.

- For contextual facts, we enumerate the metadata using a fixed set of runtime functions (see dynamic predicates in Table 6). Examples of Alice's contextual facts are $\langle @uname, alice \rangle$ and $\langle @upubkey, ab29e8 \rangle$, as her name and public key, respectively.
- Finally, we use the Policy manager to load the policy rules into the knowledge base.

5.4.3.2 Evaluation This stage serves as the central point for policy evaluation (i.e., authorization and obligations), before returning execution to the NEXUS VFS, which then enforces the evaluation result. For authorization, the access broker queries the logic engine with the permission, user, and object. The Datalog engine reasons over the knowledge base to generate inferences that match the query. Meanwhile, the Obligations module monitors system events, evaluates the usage policies, and performs the required actions.

We implemented an obligations mechanism that records the read and write operations on a given file or directory. After authorization is granted, the logic engine is queried with the 'audit' permission to determine whether the current operation has to be recorded. As an example, to record filesystem operations on 'bar/cake.c':

```
audit :- @opath(O, "bar/cake.c")
```

To accomplish this, every object (i.e., filenode and dirnode metadata) is associated with a log whose entries contain: the operation (read or write), the user's identity, and the object's version. The log is maintained as a metadata object, and is thus cryptographically-protected by the enclave runtime. Upon successful evaluation of the audit policy, an entry is simply appended to the log before proceeding with the operation. This lightweight approach only indicates that the access request was permitted, and does not imply the operation was successful (e.g., the application may crash).

5.4.4 Implementation Details

We developed a userspace program that uses SGX for transparent and secure access to protected volumes residing on remote untrusted storage. Per our design goals, our prototype does not require any server-side changes. Using the SGX SDK, we split our application into an untrusted portion that runs in normal userspace and a trusted portion that runs inside the enclave. The untrusted portion (14.2K SLOC) mainly (i) implements the filesystem and ABAC interfaces, and (ii) provides metadata access via the storage API. Whereas, the trusted portion encloses sensitive operations, including cryptographic protection and policy enforcement.

The enclave design is small and minimalistic. With a total binary size of 1.8MB, the enclave easily fits within the limited enclave-reserved memory (SGX provides about 96MB), while leaving ample memory for runtime allocations. Excluding external libraries, our TCB primarily comprised of the Joplin Controller (5K SLOC) and the NeXUS VFS (15K SLOC). For cryptographic support, we included a subset of the MbedTLS library and a C-based implementation of GCM-SIV key-wrapping primitive. Our logic engine comprised of a small datalog reasoner (1K SLOC) written in Lua, a lightweight embeddable scripting language with a small virtual machine (<20K SLOC) [137, 138]. The enclave interface comprised of 44 ecalls and 10 ocalls. Ecalls marshal data from the filesystem and ABAC interfaces into the enclave, whereas ocalls facilitate enclave access to data/metadata objects. To prevent inadvertent leakage, we sanity check our inputs and explicitly copy untrusted buffers into trusted memory before passing it to sensitive enclave code.

5.4.4.1 Cache management Joplin employs several caches in order to improve the efficiency of various ABAC operations, including in-enclave caches for recently accessed metadata and the knowledge base, and untrusted memory buffers storing encrypted metadata. On every request, the enclave checks for metadata freshness via the storage API and updates the caches on change. The Lua reasoner represents the knowledge base as a Lua hash table, and also provides an interface for asserting/revoking facts and rules. To prevent the enclave from running out of memory, the user can set a maximum for the number facts and policies

in the knowledge base.

5.5 EVALUATION

5.5.1 Use cases

In this section, we evaluate the expressiveness of the Joplin policy language by demonstrating its applicability across several use cases.

5.5.1.1 Case study 1 For a semester's course, a professor creates a volume to store student homework submissions within a flat directory. There are two user types: students and a Teaching Assistant (TA).

- The professor creates the *isStudent* and *isTA* user attributes, and assigns them to students and the TA, respectively.
- Students can submit a single file, but can overwrite up to 5 times.
- Students can read their own submission.
- The TA can read all submissions.

```
create :- isStudent(U)
write :- @oversion(O, X), _lt(X, 5), @ocreator(O, U)
read :- isStudent(U), @ocreator(O, U)
read :- isTA(U)
```

In this use case, the data producers are the invited users and the administrator just formulates rules to control who can read/write. Note that the enclave automatically increments every object's version during metadata encryption. Although the policies do not prevent the students from creating multiple submissions (creating a file with a different name), the TA could use the oldest created file and ignore the extras.

5.5.1.2 Case study 2 Jessie wants to share her pictures with her friend Mallory, and two groups of people: friends and family. This scenario is one in which a data producer intends to share files with data consumers, who can only access a portion of the data.

- Jessie creates two user attributes: *isFriend* and *isFamily*, and 3 object attributes: *isFavorite*, *isVacation*, and *isSensitive*.
- Friends can access favorite pictures.
- Family can access vacation pictures.
- Mallory can access sensitive pictures.

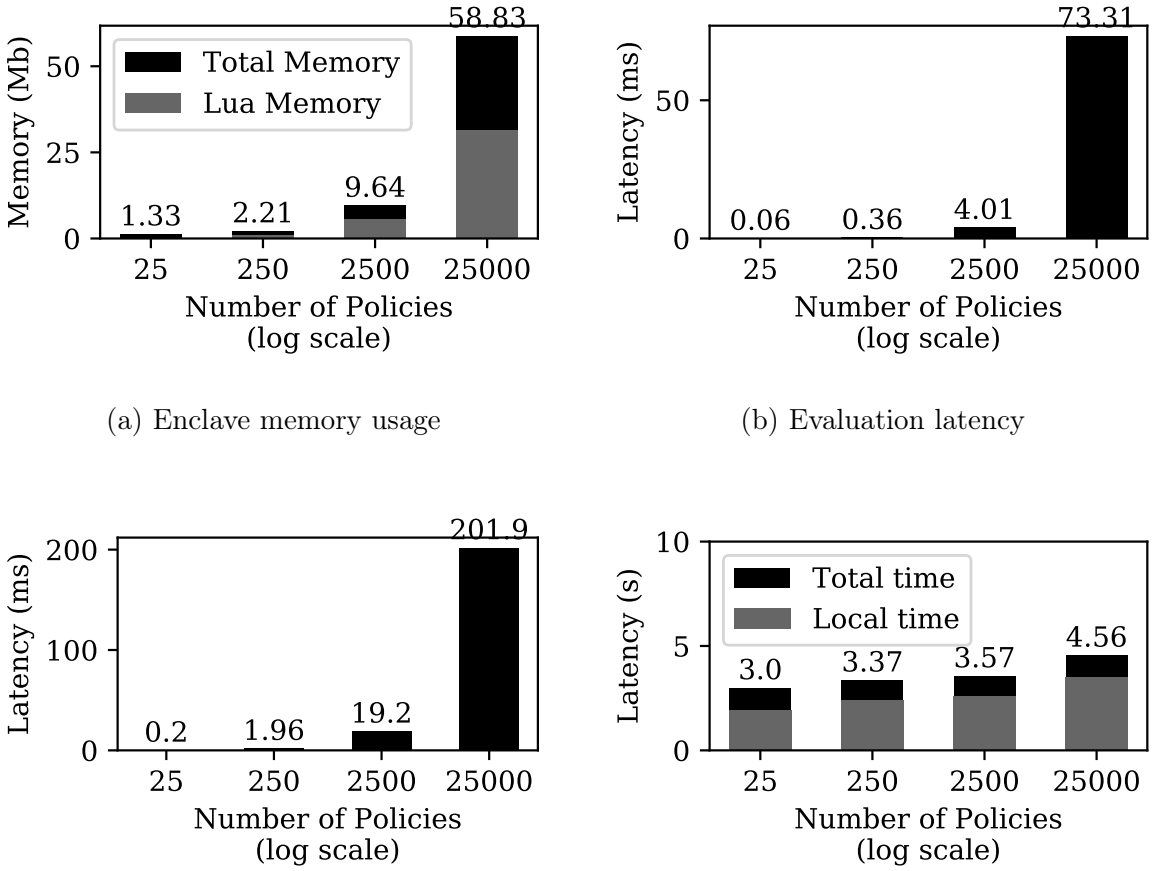
```
read :- isFriend(U), isFavorite(O)
read :- isFamily(U), isVacation(O)
read :- @uname(U, "mallory"), isSensitive(O)
```

5.5.2 Performance

To demonstrate that Joplin can provide scalable access control, we evaluated our prototype through a series of microbenchmarks and end-to-end tests. Based on the design goals in Section 5.3.1, we measure the enforcement and administrative overheads to answer the following:

1. Can the system support a large set of users and policies?
2. Can policies be added/removed efficiently?
3. What are the overheads on standard user workloads?

We performed our experiments on an i7 @3.4GHz with 8GB RAM and 128MB of EPC memory, running SGX SDK 2.2 and Ubuntu 18.04 LTS. The tests consist in measuring the latency of various access control and filesystem operations within Joplin. For timing enclave operations, we use an untrusted script that updates a memory location with a time value, which is then copied into the enclave. We stored the volume in a Dropbox shared folder, such that all changes are asynchronously uploaded by the daemon unto the cloud. To measure I/O latency, we used a python script that pauses execution until the Dropbox daemon completes synchronization. Moreover, our experimental results are averaged over 10 runs.



(a) Enclave memory usage

(b) Evaluation latency

(c) Latency to load policies

(d) Latency for policy deletion

Figure 21: Joplin Microbenchmarks.

5.5.2.1 Microbenchmarks In this test, we isolate the performance overhead incurred by Joplin by evaluating various aspects of the access control system. For our workloads, we developed a policy generator that uses a dictionary of 10000 user and object attribute names to create synthetic policies with 7–10 predicates in length.

Enclave memory usage. We measure the amount of memory required for loading a given set of policies inside the enclave. Specifically, we record the memory usage of the Lua runtime (stores knowledge base and logic engine), as well as the overall enclave runtime. Results in Figure 21a show that although memory use increases proportionally with the number of policies, the enclave can support up to 25,000 policies with less than 60MB of memory. Given that SGX provides about 96MB, this leaves enough space to cache other objects such as file and directory metadata.

Access evaluation. We measure the latency for the logic engine to generate access decisions. First, we load the knowledge base with a number of policies, and then assign attributes that will satisfy the access request. We then query the knowledge base and record the response latency in Figure 21b. Results show that even with 25,000 policies, the evaluation latency is about 73.31ms, which is well below the 100ms threshold for noticeable user delay [139].

Knowledge base refresh. We measure the time for refreshing the knowledge base with a new set of policies. Results in Figure 21c show that for a small number of policies, the evaluation latency is as small as 0.2ms, and go up to 200ms for 25,000 policies. However, this number represents a worse case scenario whereby all the policies are loaded into the enclave, which could be optimized by selecting the policies that match the requested permission.

Policy store update. We measure the latency for deleting a policy from a volume hosted on Dropbox. Recall that every in operation in Joplin requires updating and re-encrypting metadata within the enclave, before committing to the backing store. The results in Figure 21d show that a policy store of 25,000 rules can be updated in 5s. The overall latency is mostly dominated by the network cost to synchronize the 3.7MB policy store file.

5.5.2.2 End-to-end Latency In this test, we measure the latency of copying the following datasets inside a synchronized Dropbox folder: an MP3 collection (155MB), a large movie

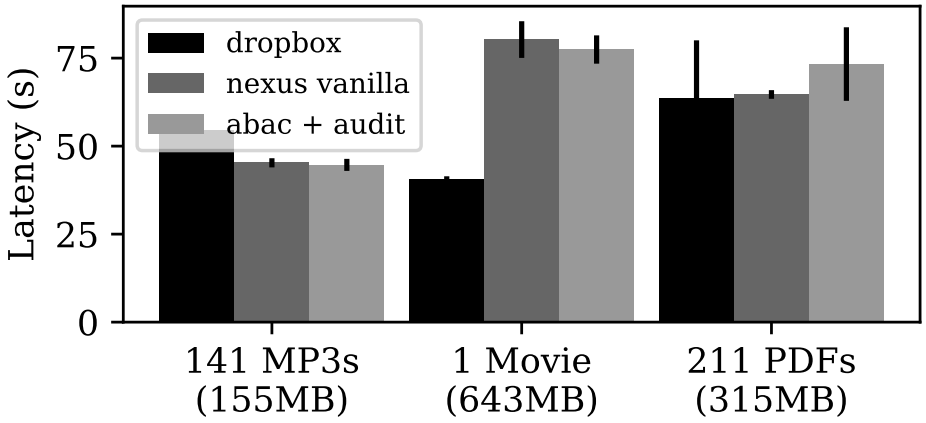


Figure 22: Latency on top of Dropbox

(643MB), and several PDFs documents (315MB). We used 3 prototypes: plain Dropbox, unmodified NEXUS, and Joplin. For our Joplin prototype, we added an audit policy that records all directory operations (e.g., when a file is created). Figure 22 shows that compared to Dropbox, both NEXUS and Joplin have less than a $\times 2$ overhead across workloads. Oftentimes, the computational and network processing (i.e., batching, compressing etc.) by the Dropbox daemon introduced spurious latencies. We recall that the reported value is the total time required to synchronize data to Dropbox, not the latency experienced by the user. In both NEXUS and Joplin, the time for copying files to the local folder was well under 5s.

5.5.3 Takeaway Discussion

Our experimental results show that Joplin effectively provides a scalable ABAC system, with applicability ranging from single user environments to large scale organizations. The microbenchmarks show support for large working sets, while providing efficient query evaluation and low administrative overheads. On standard user workloads, Joplin incurs less than a $\times 2$ overhead when synchronizing files inside a shared Dropbox folder.

Joplin is designed to operate within a multi-user environment. Although much of our

microbenchmarks focused on policies, we extrapolate other costs as follows: granting/revoking attributes requires a single write operation, and adding/removing users involves updating 2 metadata objects. Unlike pure cryptographic solutions such as ABE that require bulk data re-encryption and cascading key updates, Joplin only requires updating small metadata objects to efficiently provide both user and attribute revocation. Additionally, given that attribute assignments are relatively small (compared to policies), loading them into the enclave should be fast and use less memory. Therefore, updating the runtime access control state can be likewise performed efficiently, as it only requires dropping stale attributes from the knowledge base.

5.6 CONCLUSIONS

We proposed Joplin, a secure ABAC system that leverages trusted hardware to enabled fine-grained access and usage control on existing storage, without requiring any server-side coordination. Joplin addresses the technical challenges of decentralized access control by providing efficient revocation, dynamic policy changes, and multi-domain policy enforcement. This is enabled by a client-side enclave that provides a tamper-proof, unavoidable, and verifiable reference monitor. It applies cryptographic protections to user-provided access control policies and stores it within metadata, which is in turn decrypted for continuous enforcement at runtime. We describe how different obligations can be supported by our design. We implemented a prototype by extending NeXUS, a stackable filesystem to enable fine-grained sharing of protected volumes. Our prototype hosts a Lua-based logic engine to evaluate access requests, while enabling efficient revocation with forward and backward secrecy, multi-domain policy enforcement, user delegation, and mandatory access logging. Using example scenarios and microbenchmarks, our evaluation shows that our prototype can express a wide range of policies and impose minimal runtime overheads.

6.0 SECURITY ANALYSIS

By combining encryption and access control within the enclave, NEXUS and Joplin achieve *self-protection* [140]: the ability to protect sensitive data from all entities (trusted or untrusted) using the data's attached policy. Against the backdrop of threats in Chapter 3, we now discuss how NEXUS and Joplin meets their security objectives. We consider an attacker who has complete control over the server, including full access to exchanges with the client, and a history of the user's encrypted files. Since we are principally concerned with protecting user-created content (i.e., file data, file and directory names, access control information), we foresee the following attacks:

(1) Accessing file data
(2) Modifying file data
(3) Listing directory contents
(4) Moving file/directory to a different location
(5) Reverting file/directory to a previous version

The first four violations affect the confidentiality and integrity of individual files and directories, while the rest targets the integrity of the filesystem structure and the access control state. We now demonstrate how our design prevents an attacker from recovering or tampering protected content. For attack (5), we also assume that the attacker was removed from a given volume, i.e., the attacker has a copy of the sealed volume rootkey, but does not have their identity stored inside the supernode.

6.1 CONFIDENTIALITY AND INTEGRITY

NEXUS is mainly concerned with the protection of file content, as well as file and directory names. The user's files are encrypted in fixed-sized chunks, which are re-encrypted using fresh keys on every update. These per-chunk encryption keys are stored in the encrypted portion of the filenode metadata associated with the file. To protect directory entries, we store the correspondence between the file/directory name and its `UUID` in the encrypted portion of the dirnode associated with the directory. Likewise, Joplin stores sensitive volume information within encrypted metadata, and only performs decryption after proper access control checks inside the enclave. The attribute space lists all user and object attributes, the policy store contains all policies, and the assignment table stores both user-assigned and object-assigned attributes. All metadata are in turn re-encrypted on every update, and their metadata encryption key is key-wrapped with the volume rootkey. Moreover, because all enclave cryptography is performed using AEAD symmetric encryption, data integrity is provided alongside confidentiality. Hence, any illegal modifications of the metadata's ciphertext will be detected by the NEXUS enclave. Therefore, to read or modify file contents and file/directory names, one needs to obtain the volume rootkey.

6.2 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. The enclave generates the rootkey at volume creation, before sealing it to local disk to ensure 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 (Section 4.3.4). 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 initiated within the enclave. 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 4.3.4, 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 an attacker reconstructs the matching enclave ECDH private key, they could derive 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 trade-offs of either approach will be left to the volume owner.

6.3 ATTACKING THE FILESYSTEM STRUCTURE

Because NEXUS spreads the filesystem state across all metadata files, an attacker may attempt to modify the filesystem structure in two ways: file-swap attacks and rollback attacks.

- **File-swap attacks:** This consist in renaming or moving metadata files to other volume locations, such that the incorrect metadata file is returned to the enclave. We prevent this potential mismatch by the use of `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 `UUID` pointer validation (Section 4.3.3) to fail.

- **Rollback attacks:** The server returns older versions of the metadata files to the user. NEXUS prevents rollback attacks by maintaining a hash-tree within encrypted metadata (Section 4.3.6). This ensures that: (i) every metadata is fresh with respect to the remote root hash value, and (ii) the remote root hash is not older than the local root hash value. In the event (i) is not satisfied (e.g., on first time access), the server still must return a valid filesystem state. Thus, in effect, our rollback protection raises the attacker's burden by requiring the storage of entire volume snapshots, rather than just individual files.

Although our rollback protections provide fork-consistency by ensuring that users always observe valid volume states, the server could still equivocate about the order of operations and present different filesystem views to each user. Consider Carey and Alice are collaborating on a common file; upon request by Alice, the server may equivocate by hiding the changes performed by Carey and returning an older file to Alice. However, fork consistency limits equivocation to a single occurrence (diverging user views) and does not require users to communicate with one another directly [116]. For future work, we plan to explore mitigations against equivocation attacks with the aid of distributed monotonic counters that provide global state consistency [62].

6.4 FORWARD AND BACKWARD SECRECY

Joplin prevents arbitrary rollback attacks by ensuring forward and backward secrecy against revoked and new users, respectively. Revoking an attribute consists in removing the entry inside the assignment table, updating the last known versions of the volume attribute space and policy store, and re-encrypting the metadata with a new key. This ensures that: (i) the revoked user is no longer assigned the attribute, and (ii) older policies cannot be used with newer assignment tables. Joplin also ensures consistency between filesystem data and access control information by inlining the object assignment table within filenode and dirnode metadata. We provide backward secrecy by preventing new assignment tables from being used with older policies. Please note that Joplin can leverage the rollback protection implemented in NEXUS (Section 4.3.6).

7.0 SUMMARY AND FUTURE WORK

Today, the proliferation of consumer devices has led to an explosion in user-generated data, including pictures, documents, and videos. Cloud-based filesharing platforms provide a convenient and low-cost solution to meet the burgeoning storage needs of individuals and organizations alike. Users can access their files on any device and collaborate with one another on a global scale, while only paying for what they use. However, frequent data breaches and unplanned disclosures on this platforms have raised serious user privacy concerns. This dissertation is concerned with developing a practical security solution that ensures data confidentiality and integrity, without placing any implicit trust in the service provider.

Access control has been extensively studied within traditional closed-world scenarios, where users and resources are known and rarely change. A simple approach involves a reference monitor that validates every authorization request against a centralized database. However, the unique characteristics of the cloud environment pose several challenges that warrant a reconsideration of trust assumptions and usage scenarios. This includes: (i) data storage and processing on machines that lie outside the user's control, (ii) large number of users accessing a limitless amount of resources provisioned across organizational domains, and (iii) dynamic changes in access rights as users are added, removed, and modified. As such, a robust security solution for the cloud must provide scalable policy administration costs, fine-grained access control to associate users and resources with arbitrary granularity, while taking into account the local user context when evaluating authorization requests.

To address the above challenges, this dissertation adopts a two-pronged approach by combining the benefits of cryptography and trusted hardware to protect data both at rest and during use. Cryptographic protection ensures that sensitive information cannot be feasibly recovered by the service provider, or leaked as a result of a data breach. On the other

hand, hardware-enabled Trusted Execution Environments (TEEs) provide a tamper-proof, non-bypassable, and verifiable reference monitor. As a result, access control enforcement cannot be subverted by any external party or even authenticated users. We give a summary of our contributions, as we develop a client-side solution that requires no server-side changes, and solely relies on trusted hardware readily available on commodity user machines. We close the chapter with a survey of possible future work.

7.1 SUMMARY

This dissertation's overall hypothesis is that: *the widespread availability of trusted hardware extensions on consumer devices can provide data confidentiality and integrity, as well as scalable access and usage control within an untrusted cloud environment, while improving portability, flexibility, and performance over unmodified storage platforms.* Usage control extends traditional notions of access control in preventing data misuse even beyond user authorization. Specifically, our aim is to develop a practical solution that is *portable* across file sharing platforms without requiring any server-side coordination or trusted intermediary, *flexible* in allowing users to share files using fine-grained access control policies, and *performant* in imposing modest overheads on typical user workloads and dynamic policy changes. To this end, we developed NEXUS (Chapter 4) and Joplin (Chapter 5), two client-side solutions that leverage the Intel SGX trusted hardware to enable scalable data sharing of protected volumes. We target a deployment model that minimizes the cloud storage interface by storing metadata as files. This leads to the first research question:

RQ1: How can we adapt cryptographic protection unto cloud-based filesharing platforms?

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, and dynamic sharing of user content is required. We introduced NEXUS

(Chapter 4), a privacy-preserving filesystem that provides confidentiality and integrity to user files on untrusted platforms. Specifically, NEXUS protects file content, as well as file and directory names. When developing NEXUS, we made the following contributions:

(A1) We propose a client-side architecture to protect sensitive information within a client-server setting (Section 4.2.1). NEXUS instantiates a distributed access control platform using trusted hardware, such that the SGX enclave serves as a trusted reference monitor that executing independently on each client machine rather than centrally on the (untrusted) server. Compared to alternative architectures, this client-side approach ensures portable data protection without requiring server-side support for trusted hardware, nor relying on an enclave controlled by the service provider.

(A2) We present the design of NEXUS to share files hosted on untrusted cloud infrastructure, while ensuring the confidentiality and integrity of file content as well as file/directory names (Section 4.3.1). NEXUS performs cryptographic protection and access control inside a client-side enclave, while ensuring sensitive key material does not leak to untrusted memory. As a practical system, NEXUS enables seamless and secure key distribution using metadata files (Section 4.3.3), minimal user key management (Section 4.3.4), and efficient user revocation (Section 4.3.5).

(A3) We propose a cryptographic protocol that uses SGX remote attestation to enable secure file sharing across client machines (Section 4.3.4.1). The protocol employs files to exchange messages on the underlying shared filesystem, and does not require the parties to be simultaneously online.

(A4) We propose a rollback protection mechanism that prevents the server from returning stale metadata (Section 4.3.6). Our construction provides stronger freshness guarantees by leveraging the NEXUS virtual filesystem hierarchy to maintain a hash tree within encrypted metadata objects. This ensures the volume's filesystem state results from a valid sequence of changes without requiring any user-to-user communication.

(A5) We implement NEXUS as a userspace filesystem that allows unmodified applications to access protected volumes from a mounted directory (Section 4.4). As a practical system, our minimal enclave takes into account trusted hardware limitations (e.g., limited SGX memory), as well as the network latency of filesystem operations. We also ported to run

atop AFS and FUSE; both prototypes are completely client-side and do not require any server-side modifications. Furthermore, our FUSE prototype improves portability by: (i) requiring no changes to the OS or underlying filesystem, and (ii) enabling the migration of volumes by simply copying its directory contents across filesystems. This allows NEXUS to be readily deployable across storage platforms using any client machine equipped with SGX hardware.

(A6) We evaluate our FUSE-based NEXUS prototype over two popular remote storage platforms: Dropbox and AFS (Section 4.5). Specifically, we measured the latency using microbenchmarks that isolate the overhead of file and directory operations, database benchmarks, and popular Linux applications. We also measure the performance of user revocation relative to cryptographic solutions, as well as the overheads of rollback protection. Results show that NEXUS incurs modest penalties on standard user workloads, supports a diverse set of applications workflows and storage platforms, and offers efficient access revocation.

Our work with NEXUS shows that trusted hardware can address the practical challenges of cryptographic protection over remote untrusted platforms. However, ACLs are coarse-grained and do not incorporate the environmental context (Section 2.4.1). Attribute-Based Access Control (ABAC) uses attributes as a level of indirection to separate privilege assignment from policy specification within an open environment. Thus, we shift our focus to more expressive access control and usage control enforcement; the former is concerned with fine-grained user authorization, whereas the latter uses *Obligations* to prevent future data misuse. Obligations are mandatory actions that must be fulfilled as part of an authorization request (e.g., *record authorized accesses to a log*). This leads to the following research question:

RQ2: How can we provide scalable distributed usage control in a cloud environment?

We present Joplin (Chapter 5), a secure access control and usage control system that ensures continuous policy enforcement within a protected volume. Joplin addresses the practical challenges of ABAC, including efficient revocation with forward and backward secrecy, cross-domain policies, mandatory audit logging, and user delegation. The enclave unifies policy enforcement by mapping users and resources to access control information stored

within encrypted metadata objects. At access time, the enclave hosts a policy interpreter that continuously evaluates system attributes and policies before authorizing access and performing obligations. To that end, we made the following contributions:

(B1) We propose a client-side architecture that ensures access control information (i.e., attributes, policies etc.) is secure and reliable source, while continuously enforcing user-specified policies and obligations on each authorization request (Section 5.3). We describe our ABAC model (Section 5.3.3) and policy language (Section 5.3.4), before performing a taxonomy of obligations that could be readily supported by our client-side approach (Section 5.3.5). With this, our client-side reference monitor (Section 5.3.6) can enforce declarative policies in a tamper-proof environment without requiring any server-side support or trusted intermediary.

(B2) We address the practical challenges of ABAC by instantiating a Joplin prototype that provides decentralized policy enforcement within a secure filesystem (Section 5.4). The Joplin enclave uses encrypted metadata to manage volume attributes and policies, such that access changes require a fixed number of updates. Next, we describe how Joplin initializes (Section 5.4.2) and enforces policies (Section 5.4.3) with cross-volume support and user delegation. Our minimal enclave implementation uses a Lua-based Datalog interpreter to evaluate policies, and employs several caches to speed up authorization requests (Section 5.4.4). The prototype provides efficient revocation, mandatory access logging, and can be readily deployed on any SGX client machine.

(B3) We evaluate Joplin using several case studies and filesystem workloads (Section 5.5). The former explores how a user could share a collection of files using our declarative policy language, whereas the latter measures the latency and memory use under different scenarios, as well as the end-to-end impact on top of Dropbox. Results show that our implementation can efficiently enforce declarative policies, and has policy maintenance overheads that scale with the number of users and policies. For up to 25,000 policies, the Joplin enclave can evaluate and refresh its internal access control state in the order of milliseconds. These results imply that administrators and data owners can protect files without significant performance degradation on typical workloads.

As years go by, there has been an exponential growth in the financial and societal costs of cloud security incidents on service providers and users alike; unfortunately, legal regulations have had limited impact in curtailing these cloud data breaches. By offering a cheap and convenient access to large amounts of highly-available storage, the trend of increased user adoption of cloud-based file sharing services is bound to persist, event at the risk to data ownership and user privacy. In this dissertation, we want to provide a practical security tool that is easy to install and use, such that users can employ across storage platforms. Furthermore, users do not need to place any trust on the service provider, other than providing data storage and availability. Both NEXUS and Joplin demonstrates that individuals and organizations can manage their files on existing storage platforms using rich access control primitives, without requiring any external party or changes on the server. Their enclaves are self-contained, have a small Trusted Computing Base (TCB), and can be setup to manage multiple volumes for cross-domain collaboration. Through our implementation and evaluation, we improve the user privacy on these cloud-based services using only mass market trusted hardware already present on modern user machines. Overall, our client-side design and implementation provides a solid foundation for developing user-centric and decentralized policy enforcement using modern trusted hardware.

7.2 FUTURE WORK

The growing adoption of trusted hardware technology by major vendors can fundamentally transform how we design secure systems. Given that seamless privacy-preserving computation is the foundation of our approach, we see the following avenues for future work:

Applicability to other distributed systems. Although our current work pertains to secure file sharing, our client-side architecture could be adapted to protect other distributed applications. Our deployment model targets situations whereby sensitive data is completely generated on the client and the server mainly provides storage. Specifically, the server returns data to the client after performing very little data processing, such that sensitive user data can be modified without disrupting the server's operation. Example

client-server applications include key-value stores, messaging, and even web services. However, each scenario has a unique request-response pattern when accessing remote data, as well as imposes particular memory and processing requirements. As a result, these factors have to be taken into account when designing a practical and scalable system. For example, the hierarchical organization of filesystems require traversing the parent directories before fetching the target file; this may not the case for key-value stores that has a flat namespace.

Alternative Client-server Architecture. Adopting a client-side approach facilitates user deployment across storage platforms, but this has several drawbacks including, coarse data access, high network latency, and lack of support for concurrent access. As shown by the NEXUS benchmarks, this results in significant performance overheads on metadata-intensive workloads (e.g., writing inside a large directory). Although users of typical file sharing services are largely unaffected in their daily use, these overheads limit the applicability of our approach where performant and highly-granular data access is required. Alternatively, as trusted hardware support on the cloud continues to grow, we consider a different architecture that splits the computation between client–server enclaves. The client will still be responsible for access control and managing the encryption keys, but can temporarily share key material with a server-side counterpart to decrypt large datasets on demand. The server-side enclave never persists any secret key material, but can decrypt sensitive content to fulfill metadata intensive operations. For example, when listing a directory, the server-side enclave could collect all the file information in advance and send it to the client via a secure channel. Due to data proximity, this approach is significantly faster than downloading the individual files on the client. Furthermore, the server-side enclave could provide stronger security guarantees, such as oblivious data access with an ORAM controller and rollback protection using a global monotonic counter.

Richer Authorization and Obligation support. Another direction for future work is to extend our ABAC authorization model to support more complex relationships between attributes, such as hierarachies and constraints. A richer policy language will be required to capture these relations. As for usage control, our taxonomy in Chapter 5 lists all the applicable obligations within our client-side architecture. In addition to mandatory access

logging, we could implement other usage scenarios such as automatically deleting sensitive information from the cloud using data retention policies. Also, given the availability server-side enclave, an alternative architecture could provide access to global user context information (e.g., the number of users online).

BIBLIOGRAPHY

[1] U. Security and E. C. (SEC), "Form s-1: Registration statement - dropbox, inc." https://www.sec.gov/Archives/edgar/data/1467623/000119312518055809/d451946ds1.htm, 02 2018.

[2] S. Wodinsky, "Google drive is about to hit 1 billion users." https://www.theverge.com/2018/7/25/17613442/google-drive-one-billion-users, 07 2018.

[3] F. Lardinois, "Google updates drive with a focus on its business users." https://techcrunch.com/2017/03/09/google-drive-now-has-800m-users-and-gets-a-big-update-for-the-enterprise/, 03 2018.

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

[5] Reuters, "Personal cloud market 2019 global analysis, segments, size, share, industry growth and recent trends by forecast to 2023." https://www.reuters.com/brandfeatures/venture-capital/article?id=72896, 1 2019.

[6] 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.

[7] S. 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.

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

[9] "Dropbox hack leads to leaking of 68m user passwords on the internet." https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach, 08 2016.

[10] A. Charlton, "iCloud accounts at risk of brute force attack as hacker exploits 'painfully obvious' password flaw." https://www.ibtimes.co.uk/icloud-accounts-risk-brute-force-attack-hacker-exploits-painfully-obvious-password-flaw-1481623.

[11] Wired, "Was it ethical for dropbox to share customer data with scientists?." https://www.wired.com/story/dropbox-sharing-data-study-ethics/, 07 2018.

[12] 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.

[13] A. Technica, "Dropbox disables old shared links after tax returns end up on google." https://arstechnica.com/information-technology/2014/05/dropbox-disables-old-shared-links-after-tax-returns-end-up-on-gooogle/, 5 2014.

[14] "Dropbox terms of service." https://www.dropbox.com/terms, 04 2018.

[15] "Google terms of service." https://policies.google.com/terms.

[16] Microsoft, "Microsoft services agreement." https://www.microsoft.com/en-us/servicesagreement, March 2018.

[17] W. C. Garrison, A. Shull, S. Myers, and A. J. Lee, "On the practicality of cryptographically enforcing dynamic access control policies in the cloud," in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 819–838, IEEE, 2016.

[18] G. Brose, *Access Control*, pp. 2–7. Boston, MA: Springer US, 2011.

[19] Z. Qin, H. Xiong, S. Wu, and J. Batamuliza, "A survey of proxy re-encryption for secure data sharing in cloud computing," *IEEE Transactions on Services Computing*, 2016.

[20] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 89–98, Acm, 2006.

[21] E. Geron and A. Wool, "CRUST: Cryptographic Remote Untrusted Storage without Public Keys," *International Journal of Information Security*, vol. 8, no. 5, pp. 357–377, 2009.

[22] A. L. Ferrara, G. Fuchsbauer, and B. Warinschi, "Cryptographically enforced rbac," in *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pp. 115–129, IEEE, 2013.

[23] Y. Tang, P. P. Lee, J. C. Lui, and R. Perlman, "Fade: Secure overlay cloud storage with file assured deletion," in *International Conference on Security and Privacy in Communication Systems*, pp. 380–397, Springer, 2010.

[24] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage.," in *Fast*, vol. 3, 2003.

[25] W. Wang, Z. Li, R. Owens, and B. Bhargava, "Secure and efficient access to outsourced data," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, (New York, NY, USA), pp. 55–66, ACM, 2009.

[26] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "Sirius: Securing remote untrusted storage.," in *NDSS*, vol. 3, pp. 131–145, 2003.

[27] T. Alves, "Trustzone : Integrated hardware and software security," 2004.

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

[29] V. Costan and S. Devadas, "Intel sgx explained.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[30] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pp. 857–874, 2016.

[31] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, "Keystone: A framework for architecting tees," *arXiv preprint arXiv:1907.10119*, 2019.

[32] S. Contiu, R. Pires, S. Vaucher, M. Pasin, P. Felber, and L. Réveillère, "Ibbe-sgx: Cryptographic group access control using trusted execution environments," *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 207–218, 2018.

[33] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious file system for intel sgx," 2018.

[34] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "Pesos: Policy enhanced secure object store," 2018.

[35] D. Burihabwa, P. Felber, H. Mercier, and V. 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)*, pp. 67–72, IEEE, 2018.

[36] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx," 2017.

[37] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani, "{SPEICHER}: Securing lsm-based key-value stores using shielded execution," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pp. 173–190, 2019.

[38] J. Daemen and V. Rijmen, "The design of rijndael: Aes - the advanced encryption standard," 2002.

[39] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar, "Breakthrough aes performance with intel aes new instructions," *White paper, June*, p. 11, 2010.

[40] S. Gueron and Y. Lindell, "Gcm-siv: Full nonce misuse-resistant authenticated encryption at under one cycle per byte," in *ACM Conference on Computer and Communications Security*, 2015.

[41] D. A. McGrew and J. Viega, "The galois/counter mode of operation (gcm)," 2005.

[42] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[43] V. S. Miller, "Use of elliptic curves in cryptography," in *Conference on the theory and application of cryptographic techniques*, pp. 417–426, Springer, 1985.

[44] E. B. Barker, L. Chen, A. Roginsky, A. T. Vassilev, and R. Davis, "Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography," 2007.

[45] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 29–37, 2014.

[46] N. Sumrall and M. Novoa, "Trusted computing group (tcg) and the tpm 1.2 specification," in *Intel Developer Forum*, vol. 32, 2003.

[47] S. M. Kim, J. Han, J. Ha, T. Kim, and D. Han, "Enhancing security and privacy of tor's ecosystem by using trusted execution environments.," in *NSDI*, pp. 145–161, 2017.

[48] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. M. Eyers, and P. R. Pietzuch, "Libseal: revealing service integrity violations using trusted execution," in *EuroSys*, 2018.

[49] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves," *National University of Singapore, Tech. Rep*, 2016.

[50] J. Lind, O. Naor, I. Eyal, F. Kelbert, P. Pietzuch, and E. G. Sirer, "Teechain: Reducing Storage Costs on the Blockchain With Offline Payment Channels," pp. 125–125, 2018.

[51] B. Li, N. Weichbrodt, J. Behl, P.-L. Aublin, T. Distler, and R. Kapitza, "Troxy: Transparent access to byzantine fault-tolerant systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 59–70, IEEE, 2018.

[52] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 38–54, IEEE, 2015.

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

[54] I. Anati, F. McKeen, S. Gueron, H. Haitao, S. Johnson, R. Leslie-Hurd, H. Patil, C. Rozas, and H. Shafi, "Intel software guard extensions (intel sgx)," in *Tutorial at International Symposium on Computer Architecture (ISCA)*, 2015.

[55] Intel, "Intel®SGX Commercial Use Licence." https://software.intel.com/en-us/sgx/commercial-use-license-request.

[56] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel sgx enclaves," in *Proceedings of the 19th International Middleware Conference*, pp. 201–213, ACM, 2018.

[57] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *ACM SIGPLAN Notices*, vol. 53, pp. 665–678, ACM, 2018.

[58] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," in *2017 USENIX ATC*, 2017.

[59] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, *et al.*, "Scone: Secure linux containers with intel sgx," in *12th USENIX Symp. Operating Systems Design and Implementation*, 2016.

[60] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[61] C. Soriente, G. Karame, W. Li, and S. Fedorov, "Replicatee: Enabling seamless replication of sgx enclaves in the cloud," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 158–171, IEEE, 2019.

[62] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "{ROTE}: Rollback protection for trusted execution," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 1289–1306, 2017.

[63] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A.-R. Sadeghi, "Dr. sgx: hardening sgx enclaves against cache attacks with data location randomization," *arXiv preprint arXiv:1709.09917*, 2017.

[64] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Jitguard: hardening just-in-time compilers with sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2405–2419, ACM, 2017.

[65] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2421–2434, ACM, 2017.

[66] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 178–194, IEEE, 2018.

[67] S. Chandra, V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Securing data analytics on sgx with randomization," in *European Symposium on Research in Computer Security*, pp. 352–369, Springer, 2017.

[68] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs.," in *NDSS*, 2017.

[69] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting {SGX} enclaves from practical side-channel attacks," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 227–240, 2018.

[70] G.-J. Ahn, "Discretionary access control," in *Encyclopedia of Database Systems*, 2009.

[71] R. S. Sandhu, "Lattice-based access control models," *Computer*, vol. 26, no. 11, pp. 9–19, 1993.

[72] D. Ferraiolo, J. Cugini, and D. R. Kuhn, "Role-based access control (rbac): Features and motivations," in *Proceedings of 11th annual computer security application conference*, pp. 241–48, 1995.

[73] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, "First experiences using xacml for access control in distributed systems," in *Proceedings of the 2003 ACM workshop on XML security*, pp. 25–37, ACM, 2003.

[74] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (abac) definition and considerations," *NIST Special Publication*, vol. 800, p. 162, 2014.

[75] X. Jin, R. Krishnan, and R. S. Sandhu, "A unified attribute-based access control model covering dac, mac and rbac," in *DBSec*, 2012.

[76] X. Zhang, Y. Li, and D. Nalla, "An attribute-based access matrix model," in *SAC*, 2005.

[77] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter, "Enterprise privacy authorization language (epal)," *IBM Research*, 2003.

[78] D. Servos and S. L. Osborn, "Hgabac: Towards a formal model of hierarchical attribute-based access control," in *FPS*, 2014.

[79] M. Y. Becker, C. Fournet, and A. D. Gordon, "Secpal: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, vol. 18, no. 4, pp. 619–665, 2010.

[80] D. R. Kuhn, E. J. Coyne, and T. R. Weil, "Adding attributes to role-based access control," *Computer*, vol. 43, no. 6, pp. 79–81, 2010.

[81] X. Jin, R. Sandhu, and R. Krishnan, "Rabac: role-centric attribute-based access control," in *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pp. 84–96, Springer, 2012.

[82] J. Park and R. Sandhu, "The ucon abc usage control model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 1, pp. 128–174, 2004.

[83] J. Park and R. Sandhu, "Towards usage control models: Beyond traditional access control," in *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, SACMAT '02, (New York, NY, USA), pp. 57–64, ACM, 2002.

[84] R. Sandhu and J. Park, "Usage control: A vision for next generation access control," in *International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security*, pp. 17–31, Springer, 2003.

[85] A. Lazouski, F. Martinelli, and P. Mori, "Usage control in computer security: A survey," *Computer Science Review*, vol. 4, no. 2, pp. 81–99, 2010.

[86] F. Kelbert and A. Pretschner, "A fully decentralized data usage control enforcement infrastructure," in *International Conference on Applied Cryptography and Network Security*, pp. 409–430, Springer, 2015.

[87] A. Lazouski, G. Mancini, F. Martinelli, and P. Mori, "Architecture, workflows, and prototype for stateful data usage control in cloud," in *2014 IEEE Security and Privacy Workshops*, pp. 23–30, IEEE, 2014.

[88] F. Kelbert and A. Pretschner, "Data usage control for distributed systems," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, p. 12, 2018.

[89] F. Kelbert, "Data usage control for the cloud," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 156–159, May 2013.

[90] F. Kelbert and A. Pretschner, "Decentralized distributed data usage control," in *Cryptology and Network Security* (D. Gritzalis, A. Kiayias, and I. Askoxylakis, eds.), (Cham), pp. 353–369, Springer International Publishing, 2014.

[91] A. Pretschner, M. Hilty, and D. Basin, "Distributed usage control," in *Communications of the ACM*, p. 39, 2006.

[92] R. B. Lee, "Hardware-enhanced access control for cloud computing," in *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pp. 1–2, ACM, 2012.

[93] J. Crampton, "Cryptographic enforcement of role-based access control," in *FAST 2010*, 2010.

[94] A. J. Aviv, S. G. Choi, T. Mayberry, and D. S. Roche, "Oblivisync: Practical oblivious file backup and synchronization," *arXiv preprint arXiv:1605.09779*, 2016.

[95] B. 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.

[96] Cisco, "Cisco global cloud index: Forecast and methodology, 2016–2021 white paper cisco global cloud index: Forecast and methodology, 2016–2021 white paper." www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html, 11 2018.

[97] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P.-L. Aublin, P. Cosa, C. Fetzer, P. 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)*, pp. 386–397, IEEE, 2018.

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

[99] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: a distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[100] K. E. Fu, *Group sharing and random access in cryptographic storage file systems*. PhD thesis, Massachusetts Institute of Technology, 1999.

[101] S. Myers and A. Shull, "Practical revocation and key rotation," in *Cryptographers' Track at the RSA Conference*, pp. 157–178, Springer, 2018.

[102] S. Gueron and Y. 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.

[103] The OpenAFS Foundation, Inc. https://www.openafs.org/, 2018.

[104] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To {FUSE} or not to {FUSE}: Performance of user-space file systems," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pp. 59–72, 2017.

[105] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai, "Efficient batched synchronization in dropbox-like cloud storage services," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 307–327, Springer, 2013.

[106] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang, "Towards network-level efficiency for cloud storage services," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, pp. 115–128, ACM, 2014.

[107] J. B. Djoko, J. Lange, and A. J. Lee, "Nexus: Practical and secure access control on untrusted storage platforms using client-side sgx," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 401–413, June 2019.

[108] P. G. Lopez, M. Sanchez-Artigas, S. Toda, C. Cotes, and J. Lenton, "Stacksync: Bringing elasticity to dropbox-like file synchronization," in *Proceedings of the 15th International Middleware Conference*, pp. 49–60, ACM, 2014.

[109] J. Chen, K. Li, J. Xu, Q. Zhang, *et al.*, "Authenticated key-value stores with hardware enclaves," *arXiv preprint arXiv:1904.12068*, 2019.

[110] S. Shinde, S. Wang, P. Yuan, A. Hobor, A. Roychoudhury, and P. Saxena, "Besfs: Mechanized proof of an iago-safe filesystem for enclaves," *arXiv preprint arXiv:1807.00477*, 2018.

[111] S. Contiu, S. Vaucher, R. Pires, M. Pasin, P. Felber, and L. Réveillère, "Anonymous and confidential file sharing over untrusted clouds," *arXiv preprint arXiv:1907.06466*, 2019.

[112] M. Blaze, "A cryptographic file system for unix," in *Proceedings of the 1st ACM conference on Computer and communications security*, pp. 9–16, ACM, 1993.

[113] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano, "The design and implementation of a transparent cryptographic file system for unix.," in *USENIX Annual Technical Conference, FREENIX Track*, no. 1-880446, pp. 10–3, 2001.

[114] C. P. Wright, M. C. Martino, and E. Zadok, "Ncryptfs: A secure and convenient cryptographic file system.," in *USENIX Annual Technical Conference, General Track*, pp. 197–210, 2003.

[115] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," *ACM SIGOPS Operating Systems Review*, vol. 36, 2002.

[116] J. Li, M. N. Krohn, D. Mazieres, and D. E. Shasha, "Secure untrusted data repository (sundr).," in *OSDI*, vol. 4, pp. 9–9, 2004.

[117] M. Backes, C. Cachin, and A. Oprea, "Lazy revocation in cryptographic file systems," in *Security in Storage Workshop, 2005. SISW'05. Third IEEE International*, pp. 11–pp, IEEE, 2005.

[118] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea, "Iris: A scalable cloud file system with efficient integrity checks," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 229–238, ACM, 2012.

[119] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Athos: Efficient authentication of outsourced file systems," in *International Conference on Information Security*, pp. 80–96, Springer, 2008.

[120] H. Jin, H. Jiang, K. Zhou, R. Wei, D. Lei, and P. Huang, "Full integrity and freshness for outsourced storage," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 362–371, IEEE, 2015.

[121] D. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-based access control*. Artech House, 2003.

[122] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park, "Formal model and policy specification of usage control," *ACM Trans. Inf. Syst. Secur.*, vol. 8, pp. 351–387, Nov. 2005.

[123] D. Servos and S. L. Osborn, "Current research and open problems in attribute-based access control," *ACM Comput. Surv.*, vol. 49, pp. 65:1–65:45, 2017.

[124] M. S. Artigas, C. Cotes, M. R. Rodríguez, and P. G. López, "Stacksync: Attribute-based data sharing in file synchronization services," *Concurrency and Computation: Practice and Experience*, vol. 30, 2018.

[125] K. Yang, X. Jia, and K. Ren, "Attribute-based fine-grained access control with efficient revocation in cloud storage systems," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, (New York, NY, USA), pp. 523–528, ACM, 2013.

[126] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Infocom, 2010 proceedings IEEE*, pp. 1–9, Ieee, 2010.

[127] A. J. Lee, M. Winslett, and K. J. Perano, "Trustbuilder2: A reconfigurable framework for trust negotiation," in *IFIP International Conference on Trust Management*, pp. 176–195, Springer, 2009.

[128] T. H. Noor, Q. Z. Sheng, S. Zeadally, and J. Yu, "Trust management of services in cloud environments: Obstacles and solutions," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 12, 2013.

[129] Y. Chunxiao, W. Zhongfu, F. Yunqing, *et al.*, "An attribute-based delegation model and its extension," *Journal of Research and Practice in Information Technology*, vol. 38, no. 1, p. 3, 2006.

[130] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[131] R. Ramakrishnan and J. D. Ullman, "A survey of deductive database systems," *The journal of logic programming*, vol. 23, no. 2, pp. 125–149, 1995.

[132] A. Pretschner, M. Hilty, F. Schütz, C. Schaefer, and T. Walter, "Usage control enforcement: Present and future," *IEEE Security & Privacy*, vol. 6, no. 4, pp. 44–53, 2008.

[133] M. Hilty, D. Basin, and A. Pretschner, "On obligations," in *European Symposium on Research in Computer Security*, pp. 98–117, Springer, 2005.

[134] F. Martinelli, P. Mori, A. Saracino, and F. Di Cerbo, "Obligation management in usage control systems," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 356–364, IEEE, 2019.

[135] N. Li, H. Chen, and E. Bertino, "On practical specification and enforcement of obligations," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pp. 71–82, ACM, 2012.

[136] H. Liang and M. Li, "Bring the missing jigsaw back: Trustedclock for sgx enclaves," in *Proceedings of the 11th European Workshop on Systems Security*, p. 8, ACM, 2018.

[137] The Programming Language Lua. https://www.lua.org/, 2019.

[138] John D. Ramsdell. https://sourceforge.net/projects/datalog/, 2019.

[139] J. Nielsen, *Usability engineering*. Elsevier, 1994.

[140] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, "A software-hardware architecture for self-protecting data," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 14–27, ACM, 2012.