非信頼ストレージを介したプライベートなファイル共有を支援するファイルシステム

非信頼ストレージを介した プライベートなファイル共有を支援するファイルシステム

File system extension to support secure cloud-based sharing

山本憲正
Kensho Yamamoto
法政大学情報科学部コンピュータ科学科
E-mail: kensho.yamamoto.3s@stu.hosei.ac.jp

Abstract

Cloud storage is a common platform for multiple users to share data over the Internet. It offers a huge storage capacity at low cost and an easy-to-access interface for all users. However, data protection is considered to be an issue, especially for highly secret information. Users usually trust cloud storage providers who do not leak information, but the administrator and tools they use can access arbitrary storage data. One approach against these problems is encrypting files stored in cloud storage. Using encryption, files are kept confidential, but this requires a complicated procedure and understanding of some specific level of cryptography. In this paper, we propose a secure cloud filesystem that enables files to be encrypted and decrypted transparently when storing to or extracting from the filesystem. Each file is protected using a key-based encryption mechanism, and the keys are also stored on the cloud filesystem in secure indexed with the pathname of the target files. This enables users to easily use secure cloud storage that maintains confidentiality and share data using multiple cloud storages without the users needing special knowledge on encryption.

1 序論

昨今広く使用されているクラウドストレージサービスは複数のユーザのストレージを集中的に管理し，インターネット経由で提供することによって異なる環境からの利用，同一サービスを利用している他のユーザとの共有を容易にしている．ユーザはサービスから提供された ID を用いて認証を行い，他のユーザの ID を指定して共有が可能である．ファイルの URL を伝達することによる多数への共有が可能である場合もある．一方でデータの機密性についてはストレージサービスを提供するクラウド事業者を信頼することが必要となる．クラウド事業者はユーザが保存したファイルを閲覧し利用することが可能であり，機密性の高いファイルをクラウドストレージへ保存するということはそのファイルを事業者が漏洩や悪用しないという前提が必要となる．またはそのファイルに不正な書き込みが行われ，改ざんされたファイルを利用することになる可能性も考えられる．また，Wani らの研究 [1] や Khashan らの研究 [2] で指摘されているようにクラウドサービスに対する悪意を持った攻撃者や，プライバシーに責任を負う組織によってもデータの機密性は脅かされている．よって，信頼性を確保できないクラウドストレージを利用する場合，機密性が高い情報をそのまま保存する目的には適していないと考えられる．これに対して機密性の保護を確実に行うためには，ユーザがファイルを暗号化してクラウドストレージ上に保存する．暗号化されたファイルは不正な閲覧をされても内容を知られることはなく，可用性を下げることを目的とした改竄を受けたとしても復号時に改竄の検知をすることが可能となる．しかし，この暗号化したデータを他のユーザと共有するには，復号のための鍵情報を共有することが必要となり，これが漏れれば第三者にデータが漏れ得ることになる．公開鍵暗号を用いれば，公開情報を用いて秘密の保持が可能となるが，その操作手順は煩雑であり，またユーザが復号などの処理を明示的に行うことが必要となる．

そこで，本研究では，ファイルの保存時に暗号化を自動的に行うことでクラウドストレージに対する機密性を保持しつつ，クラウドストレージ上の暗号化したファイルを他者と共有可能にする Multi-user Cloud EncFS を提案する．この機構によりクラウドストレージへ保存するデータの暗号化とそれに伴う鍵管理を自動化するため，ユーザは暗号化を意識せず安全にクラウドストレージを利用することが可能となる．

また，暗号化ファイルを複数のユーザに共有する時，ファイル 1 つに対してユーザ毎に，暗号処理に使用する鍵もまた共有する必要がある．この共有情報に関する管理を TEE(Trusted Execution Environment) を用いたサーバ内で行うことによって更なるプライベートなファイル共有を可能とする．

2 EncFS

EncFS[3] は単一のユーザを想定したローカル環境で動く暗号化ファイルシステムである．暗号化したディレクトリ内のファイルやディレクトリを復号，逆に平文のファイルやディレクトリを暗号化して指定した別のディレクトリへマウントする機能がある．EncFS では，暗号化のメタデータはソースディレクトリ直下に作成される，”.encfs6.xml”に保存されている．そのファイルにはそれぞれのエンティティの暗号化鍵をユーザが設定したパスフレーズで暗号化し保存されている．

最上位ディレクトリに存在するメタデータファイルが漏洩した場合にはブルートフォース攻撃などにより容易にパスフレーズが特定されてしまい，すべてのデータが漏洩してしまう．これを防ぐため，クラウドストレージへのバックアップを目的とする場合には最上位のディレクトリのメタデータファイルは除いてクラウドへ保存しなければならない．反対に，メタデータファイルが存在しない場所では暗号化も復号も困難である．

例えば暗号化した環境とは異なるが，クラウドにアクセスできる環境から同じデータの利用を試みたとする．この時，EncFS 導入のためのパスフレーズが適切であったとしても，クラウド上の暗号化されたファイルを復号するための鍵が存在

Supervisor: Prof. Toshio Hirotsu

していないため，データの復号は不可能である．つまりクラウドストレージを利用したファイルの共有には不向きである．

D.Leibenger らの研究 [4] では，単一のユーザのみを想定している EncFS に対し複数のユーザに対応する拡張機能を提案している．まず，ファイルに対してそれぞれ共通鍵暗号方式の鍵が生成されそれを用いてファイル自体のデータと付随するメタデータが暗号化される．一方で Linux のファイルシステム上に存在するユーザやグループに対して ID が振り分けられ，ユーザは自身のパスコードを設定する．ファイルの暗号化に使用された暗号鍵はアクセス権があるユーザそれぞれのパスフレーズで保護され，各ディレクトリに作成される設定ファイルへリスト形式で保存される．設定ファイルが含まれるディレクトリもまた暗号化され，その情報は親ディレクトリの設定ファイルに保存される．メタデータを暗号化することによりアクセス日時や権限情報などからファイルの中身を推測される危険性も解決している．この拡張機能により EncFS と同様の暗号化機構を複数のユーザを対象として実現することができる．

しかしながら，ディレクトリの復号には再帰的に上位のディレクトリの設定ファイルが必要となり，最終的にはルートディレクトリにある設定ファイルが必要となる．逆に言えば全ての暗号化がルートディレクトリの設定ファイルに依存する構造となっているため，この設定ファイルが存在しない環境では利用することができない．つまり，別の環境のユーザへのクラウドストレージ経由のファイル共有にはこの拡張機能でも対応が不可能であると言える．

O. A. Khashan の研究 [5] では，データの爆発的な増加にのために企業が大容量データストレージや柔軟なデータ共有サービスを目的としたクラウドストレージシステムにデータをアウトソースするようになった環境における機密データの機密性と安全性に焦点を当てている．クラウドユーザーは，これらの機密データがクラウドプロバイダーによってどのように扱われ，保護されるのかについて，何の保証もない．さらには，クラウドサービスプロバイダーは，不正な利益のためユーザーデータを無許可の団体に漏えいする可能性があるとすら述べている．この問題に対してデータは信頼できない可能性がある遠隔地のクラウドサーバーに，保存する前に暗号化する必要がある．一方で従来の暗号化システムではデータ所有者にファイルやその暗号化処理等，暗号化操作の管理という大きな負担を課していた．ユーザはクラウドへアップロードする前にデータを手動で暗号化する必要があり，さらに暗号鍵を手動で生成，管理，保管しなければならない．

そこでこの研究では ID ベース暗号化 (IBE) を使用したハイブリット暗号方式でファイルシステムを設計しクラウドストレージ上のデータをファイル単位で共有することを実現している．IBE は信頼できる第三者機関として PKG を利用しており，この PKG は有効期限に応じて定期的にユーザの秘密鍵を更新しており，ユーザはデータの復号の際に ID を用いて PKG から取得した秘密鍵を用いてファイルの復号を行う．

Xiao らの研究 [6] では EncFS とよく比較される透過的な暗号化ファイルシステム eCryptfs の暗号化処理による性能低下に着目している．EncFS との大きな相違点は暗号化データが暗号化ファイル本体に保存されている事である．非暗号化ファイルシステム EXT4 と比較すると eCryptfs は書き込みと読み取りでそれぞれ最大 58.53% と 86.89% の性能低下が発生した．暗号化操作の性能向上のため，新たな命令セットの導入やハードウェアアクセラレートなどが試みられている．しかしながら，ファイルシステムやデバイスドライバは 4KB という小さなブロックサイズによって暗号化処理および I/O 処理を行うため，多くの割込み処理が発生する．これにより命令セットの呼び出しが多くの CPU サイクルを消費したり，ハードウェアアクセラレータを呼び出すオーバーヘッドが利点を相殺する場合もある．

そこでこの研究では新たに高速で安価なストレージシステムとして NVM を利用し暗号化および I/O 処理を完全に並列化している．NVM がバイトアドレス指定可能であることを利用しハードウェアアクセラレータが直接アクセス可能なアドレス空間を提供，ルックアップ処理を高速化するためにインデックス構造を追加構築し I/O リクエストのスケジューリングシステムを導入する事で暗号化処理の性能向上を実現した．NVeCryptfs は eCryptfs に対して読み込みで 23.41 倍，書き込みで 5.82 倍の性能を実現している．

PGP(Pretty Good Privacy) は暗号化されたファイルの共有を支援するソフトウェアである．公開鍵暗号方式を利用した電子署名や，共通鍵暗号方式とのハイブリット方式によりファイルの暗号化を行うことが可能である．

ファイルの暗号化では，ファイル自体を共通鍵で暗号化した後に共有相手の公開鍵で共通鍵の暗号化を行い，それを暗号化されたファイルにバインドして受け渡す．複数のユーザに対する一括共有にも対応しており，共通鍵をそれぞれの公開鍵で暗号化したものを全て暗号化ファイルにバインドし，その中から被共有者に対応する鍵情報を秘密鍵で復号して共通鍵を利用する．PGP の利用には事前に公開鍵を交換する必要があり，その公開鍵の正当性を保証する仕組みとして公開鍵に複数のユーザの署名をするキーチェーンという仕組みがある．取得した他者の公開鍵から署名を計算し，それを公開鍵の持ち主が公開している署名と同一の物かを検証することによって公開鍵を信頼する．信頼した公開鍵には自身の秘密鍵によって署名を行い，以後自身の署名を確認することによって正当性を担保することが可能となる．

3 脅威モデル

本研究で想定する脅威モデルを整理する．ユーザは任意のコンピュータ (以後クライアント) 上でファイルの読み書きを行い，編集したファイルを任意のクラウドストレージサービスを利用してリモート環境へアップロードする．クラウドストレージ上に保存したファイルは同一サービスを利用している他ユーザに対して閲覧権限を付与することを通して公開することが可能となっており，被共有ユーザは任意の時刻に閲覧権限のある共有ファイルにアクセスすることが可能である．攻撃者はクラウドストレージに対して管理者権限のアクセスが可能となっており，クラウドストレージサービスがどのようなセキュリティを行っているかはユーザには不透明とする．

クライアントに対する所有ユーザ以外からのアクセスは本研究の論旨と外れるため考慮しないものとする．クラウドストレージサーバとの通信プロトコルは任意の物とし，暗号化の有無を考慮しない．

4 設計

利用者から透過にアクセスさせるため，クライアント上のファイルシステムとして利用可能とする．ファイルの暗号化を

共通鍵暗号方式のみで行うことは，共通鍵をクラウドストレージに保存することが漏洩と同義である以上，EncFS やその拡張機能と同様に本研究では適していない．では公開鍵暗号方式のみでファイルの暗号化を行うことを考える．クラウドストレージに保存する際には自分の公開鍵で暗号化を行い暗号化ファイルを保存し，利用時には自身のオンプレミスにある秘密鍵で復号する．しかしこの方法では，共有相手や所有者自身が他の端末から利用を試みた際には秘密鍵が無く，利用不可能である．そこで暗号化時に共有相手など他の環境で利用する場合，各秘密鍵に対応する公開鍵それぞれでファイルを暗号化し，利用時には自分用に暗号化されたファイルを自身の秘密鍵で復号することを考える．この方法では共有相手の数だけ暗号化ファイルが作成されることとなり，ファイルの所有者が暗号化処理を行う際に多くアップロードする必要が生まれ，クラウドストレージの容量の圧迫にもつながる．以上の理由から PGP と同様，ファイル自体の暗号化を共通鍵で行いそのファイル鍵を所有者や各共有相手の公開鍵で暗号化することとする．暗号化ファイルと暗号化したファイル鍵をクラウドストレージに保存し，利用者は自分用に暗号化されたファイル鍵と暗号化ファイルをクラウドストレージから取得してファイル鍵の復号という処理を行うこととなる．この時，被共有者は事前に共有者のルートディレクトリの場所に関する情報を得ているものとする．

そして暗号化ファイルと鍵ファイルの対応付けを行う index を導入する．ファイルの暗号処理では，index を参照しファイル鍵のパスを閲覧する．作成したファイル鍵は公開鍵によって暗号化されクラウドストレージに保存されるが，そのファイル鍵の場所と対応するファイル本体の場所，誰の公開鍵で暗号化されたのかということが index によって紐づけられる．

復号時にはファイルを開く処理に応じ，アップロード時と同様に index を閲覧し自身のファイル鍵を秘密鍵で復号し，ファイル自体の復号を行う．

暗号化処理に利用する公開鍵の受け渡し管理のために鍵サーバを用意する．共有を行う際には，鍵サーバから共有相手の公開鍵を取得する．そして index を参照しファイルに対応するファイル鍵を取得，相手の公開鍵でファイル鍵を暗号化しクラウドにアップロードする．共有を取りやめる際には，利用した際に復号した鍵を保存している可能性からファイルの再暗号化の必要がある．index から共有相手とファイル鍵を全て閲覧し，ファイル鍵を再度作成した後，取りやめる相手を除いた共有相手全ての公開鍵でファイル鍵を暗号化する．以前のファイル鍵をそれらで上書きし，取りやめる相手のファイル鍵は削除，index を更新する．

ファイルの作成，取得や更新を行う際にはクラウドストレージに対してリクエストを送る必要があるが，その際操作対象のファイルを指定方法についてサービス側がどのようなリクエストを指定しているかを考慮する必要がある．ルートディレクトリからのパス構造で指定可能な場合とサービス側から振り分けられた ID を使用する場合である．これは主にファイルシステム上のパスルックアップの処理に影響があり，前者であればクライアントのファイルシステム上のパスと対応付ければ良いが，後者の場合は下位のディレクトリのルックアップ処理において手順が多く必要となる．まず，子ディレクトリのファイル ID を得るために，親ディレクトリに含まれるファイルやフォルダの一覧をサービスから取得する．次に取得されたファイル一覧から目的のフォルダと同じパスを探索しそのファイル ID を特定する．そして最後に特定したファイル ID に対して行う予定であったリクエストを行う．ここで親ディレクトリがルートディレクトリはなく，さらなる上位ディレクトリが存在する場合にはルートディレクトリに到達するまで再帰的に処理を行う必要があるため，階層の数だけクラウドへリクエストを行う必要がありオーバーヘッドとなる．そこで，まず各ディレクトリに index としてファイルを 1 つ配置し，そこに暗号化ファイル情報とその鍵ファイルを紐づけたデータを保存する．そしてそれとは別にユーザ毎にパス構造を保持する path_index をルートディレクトリに導入する．この path_index へ，パス構造に対応する index ファイルの情報を保存することによって，利用者はまずパス情報の取得，そこから index ファイルの取得と，どんなにディレクトリ階層が深いファイルであってもクラウドに対して 2 回の要求を行うことでルックアップ処理を行うことが可能となる (図 1)．path_index は共有関係毎に存在するため，ユーザがアクセスする際には自身の path_index を識別する必要がある．その対応付けの仕組みとして，path_index とユーザのマッピングを更に導入するという方法と path_index ファイルの命名規則によってルートディレクトリから探索する方法が考えられる．

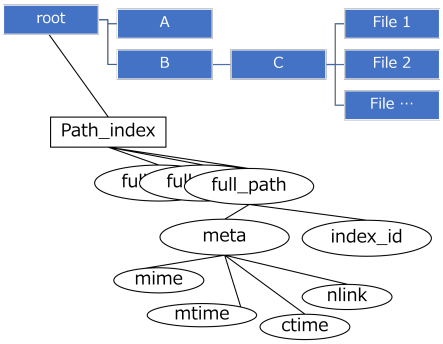

図 1　path_index usage

前者はマッピング情報の取得方法が事前共有情報として必要となり，共有者には他の共有関係を公開することとなる．後者は事前共有情報としてファイル名が必要となり，ユーザ ID が簡易な方法の 1 つだがこれでは共有関係が自明であり安全ではない．そこでユーザ ID をランダムな文字列とする方法が考えられる．本研究では共有者同士の共有関係相互隠匿性を考慮し後者の問題点を解消することによって実現する．ランダムな文字列を導出するにあたり実行環境もまた秘匿される必要がある．これの実現として次章で述べる Intel SGX を使用し，path_index 名の管理を隔離された実行環境で行うことによって path_index 対応付けの機密性を向上させる．

ここで path_index はファイル所有者が共有を行う際に所有者のユーザ ID と被共有者のユーザ ID によって被共有者の path_index 名を得ることとする．共有に関係のないユーザへの path_index 名の漏洩を避けるため，公開鍵を用いて暗号化した文字列によって送信する．

5　実装

実装に使用するクラウドストレージサービスは普及率の高さや個人で API の利用が可能なことから Google Drive APIv3 を用いた．Google Drive はファイルとフォルダの管理を ID

を用いて行っているサービスであり，API を利用する際にはファイル ID の指定が必要である．パスをキーとしたによるファイルへのアクセスは不可能となっている．Ubuntu18.04.2 上で Python3.8 を使用し，FUSE[7] を Python で実装している fusepy3.0.1[8] を用いクライアント上のファイルシステムとして実装した．

FUSE とは，本来ファイルシステムを実装するうえでカーネルプログラムに変更を加える必要があるところ，ユーザースペースからカーネルを操作する橋渡しを行うソフトウェアである．UNIX 系 OS のファイルシステムを抽象化している Virtual File System(VFS) が提供している open や read, write などのインターフェイスを同名の関数として書き込むことで独自の処理を行わせることが可能である．カーネル空間にあるドライバとユーザ空間のデーモンによって構成されており，ドライバは Linux の仮想ファイルシステム VFS に登録される．ファイルシステムに対して操作があると VFS からドライバにシステムコールが受け渡され，デーモンで割込み処理を実行することが可能となる．暗号処理は pycryptodome3.14.1 を使用し, 共通鍵暗号方式には AES, 公開鍵暗号方式には RSA を用いた．また，利用者の識別子として，本実装では Google Drive 利用者が登録時に取得する Email アドレスを利用する．

path_index のファイル名を導出するサーバでは，SGX として linux-sgx-sdk[9] を導入した．5.6 節で述べるように SGX はユーザ権限で実行するため通信に使用する処理が実行不可能である．そこで Enclave 内部から socket の作成や操作，OpenSSL と同様の API の Wrapper の SGX-OpenSSL[10] を使用して実装した．SGX-OpenSSL は OpenSSL1.0.X 系に対応しており OpenSSL1.1.X 系には対応していない．サーバを実行している環境では Ubuntu20.04.5LTS 上で C++ を使用した．CPU は SGX に対応している Intel®Core™i5-10600 @3.30GHz である．

5.1 起動処理

初回にプログラムを起動する際には，ファイルの暗号化に使用する公開鍵と秘密鍵の生成を行い，公開鍵は鍵サーバに登録する．生成する際に鍵を保存するためのディレクトリを隠しファイルとして作成する．鍵生成には cryptography の Random と RSA を使用し，Random.new を実行することで乱数生成器を用意，RSA.generatte 関数の引数に生成器と鍵長を指定して秘密鍵を作成する．秘密鍵から PEM 形式で出力し，同様に公開鍵も出力，それぞれ utf8 形式にデコードしてファイルに書き込む．ここでファイル名は固定としておき，2 回目以降の起動時にはそのファイル名が存在するかどうかを確かめることによって処理が分岐する．暗号処理の都度鍵ファイルから読み取りオーバーヘッドになることを防ぐため，RSA 鍵ファイルが存在した場合や新たな鍵を生成した後に，鍵データはクラス変数として保持することとする．

また，クラウドストレージを API を用いて利用するために認証を行う必要がある．初回起動時にユーザがこのシステムによる自身の Google アカウントへのアクセスをブラウザ上で許可する手順を行うことによってアクセストークンが発行され，API を使用する際にはこのトークンを用いてリクエストを行う．システムで利用する共有スペースとして作成したフォルダの ID は利用者で事前に共有していることとし，ファイルなどの探索はこのフォルダに対して行われることとする．なお，初回に限らずプログラムの実行時には利用者がアクセスするクラウドから取得したファイルを保持する実体ディレクトリの作成を行い，プログラム終了時に削除する．利用者がファイルシステムにアクセスする際にはマウントディレクトリのパスを指定した参照を行うが，FUSE で割り込んだ処理によってパスの変換が行われ実体ディレクトリに存在するファイルやフォルダのデータを読み取ることとなる．

5.2 フォルダ操作

フォルダの作成や削除など操作を行う際には path_index に反映し index ファイルを作成する必要がある．作成時の処理は mkdir 関数で行う．path_index はパスを鍵とした辞書型配列であるため，作成されるパスに対応する値が存在するかによってそのフォルダを確認する．下の階層にフォルダを作成する場合，親フォルダのパスに対応する値がなければ親フォルダに対して FileNotFoundError を発生させる．子フォルダが存在する場合には FileExistsError を発生させ，存在しない場合は次の処理を行う．まずストレージサービスにフォルダの作成要求を行い，フォルダの ID を取得する．ここでクライアントのファイルシステム上において，対応するパスにも実体のフォルダを作成する．次にこの作成したフォルダに対応する index ファイルの作成である．作成した際に振り分けられるファイル ID を path_index に保存する．この処理によってフォルダに関する情報が初期化され，次回以降 path_index と index ファイルの利用が可能となる．

フォルダを削除する際には，path_index からパスを鍵として pop し，更新した内容をクラウドへ上書きする．そしてクラウドへフォルダの削除要求を行う．中に含まれているファイルごと削除される．

ファイルやディレクトリにアクセスする際には，親ディレクトリに含まれるエンティティ一覧の取得が発生することとなる．アクセス前後に発生するメタデータの取得やそれに付随する path_index の扱いに関しては前章 4 や次節 5.3 で述べる．ディレクトリ内エンティティ一覧の取得は，readdir という関数で処理を行う．まず，システムを通さず直接クラウドストレージに対して処理を行った場合を考慮し，readdir はクラウドストレージに問い合わせる必要があるためディレクトリの ID を特定する．処理が発生したディレクトリのパスを鍵として path_index を読み取ったデータからファイル ID を取り出し，それを用いて含まれるファイル一覧をクラウドへリクエストする．リスト形式でレスポンスを得，次にリストのそれぞれの要素に対して繰り返し処理によって操作を行う．クラウドから得られるデータにはメタデータが含まれており，mimeType がフォルダである場合には，path_index にデータがあるかの確認を行い，ない場合には追加する．そして実体ディレクトリにおける該当パスにフォルダを作成する．fusepy では readdir の返り値はリスト形式ではなく順次返す形式となっており，yield によってファイル名を返す．

5.3 メタデータ取得

FUSE ではファイルやディレクトリの存在確認や，フォルダに含まれるファイル一覧の取得，アクセス前などに getattr という関数を実行する．この getattr 関数によりファイルの作成日時 st_ctime や更新日時 st_mtime，ファイル種別を表す st_mode とユーザやグループの id とサイズを辞書型で渡す必

要がある．階層が下のファイルに対してこの関数が呼び出される際には，ルートフォルダから下位のフォルダに対して順に処理が行われる．この時それぞれのフォルダに関するメタデータ取得を毎回クラウドストレージに対して行うと，前節で述べたファイル ID 管理上のルックアップ処理と同様オーバーヘッドにつながる．前節ではパス管理クラウド上のルックアップでは 1 度のリクエストで可能と述べたが，この FUSE における getattr の仕様により，複数回のリクエストが必要となる．これを回避するため，path_index にはパスと index ファイルの情報とともに getattr で求められる作成日時などのメタデータの反映も行うこととする．最初に発生する getattr 時に path_index を取得してキャッシュとして保持することにより，以降の getattr に対してはキャッシュから閲覧してメタデータを与えることが可能となる．

この時，path_index の更新時刻を保存しておくことにより次の読み取り時に path_index の更新日時と比較し更新があったときのみダウンロードを行う実装とすることによって処理時間の短縮を行う．起動後初回の問い合わせ時には必ずダウンロードを行い，その際に更新日時をパスを鍵とした辞書型変数に保存する．以後の更新確認時には更新日時の取得をまず行い変数の値と比較する．それが異なっていればダウンロードをして上書きをし，同じならば何も変更をせずに path_index ファイルを読み取る．

5.4 ファイル書き込みと読み取り

ファイルの新規作成や既存の物を開く時には，通常のディスクに対する書き込み処理に割込み暗号化を行いクラウドへアップロードする．新規作成時に共通鍵が生成され自身の公開鍵で暗号化，クラウドに保存してサービスから得られたファイル ID を親ディレクトリの index ファイルへ保存する．共通鍵の生成には pycryptodome の”Random”を使用する．”Random.get_random_bytes”関数の引数に鍵長を指定することによって鍵長文のランダムなバイトが得られ，それを AES 鍵とする．再利用時には index ファイルから得られたファイル ID によって鍵ファイルのダウンロードを行う．ダウンロードした鍵ファイルは秘密鍵によって復号され，以降ファイルの再利用時の暗号化に使用される．

ファイルの書き込み時には write 関数で割込み処理を行い，暗号化したファイルをアップロードする．AES 暗号でファイルの暗号化を行い，作成されたファイル鍵は UTF-8 でエンコードしたのち RSA 暗号である PKCS1_OAEP で暗号化しアップロードする．

ファイルを読み取る際には read 関数によって割込みを行う．再利用時と同様に index ファイルからデータを取得し，ファイル鍵を秘密鍵で復号しファイルの復号に使用する．

5.5 共有の有効化と無効化

他者に共有を行う際には公開鍵を受け渡す必要があるため，本実装ではユーザ ID をクエリとして公開鍵を受け渡すサーバを導入して実装した．共有処理により，被共有者の公開鍵によってファイル鍵が暗号化され，被共有者の index ファイルが作成または更新，必要に応じて path_index の更新が行われる．

共有を無効化する際には無効化対象ユーザの path_index から該当パスとそれに付随するメタデータを削除し，共有ファイルが含まれるフォルダの index ファイルから鍵ファイル ID を消す．ファイル鍵を再暗号化したのち，所有者とその他の被共有者の鍵ファイルに上書きすることによって鍵の更新を行う．

5.6 Intel®SGX

Intel SGX は Trusted Execution Environment(TEE) 実装の 1 つである [11]．SGX は Intel 製の CPU によってメモリ上に暗号的に隔離された実行環境 (Enclave) を構築し保護する．Enclave に対しては非保護領域からのアクセスが制限されており，事前に設定した専用の命令セットを用いてのみ可能である．SGX はホスト OS やハイパーバイザを信頼していないため，ホストからの攻撃に対しても耐性を持つ一方，非特権で実行するがゆえに特権が必要となる処理にも制限がある．

Enclave 内部で利用し計算したデータは揮発性であり，終了すると再利用は不可能となっているがデータを暗号化して外部へ出力する Sealing の利用によって保存を実現可能である．

5.7 path_index 名サーバ

Enclave 外のプログラムでは Enclave の実行開始と TLS サーバの起動関数の呼び出しを行う．Enclave 内部で TLS 通信の処理を操作可能とするため，上で述べた SGX-OpenSSL を導入し OpenSSL の API を利用するには指定の Wrapper ヘッダーを include する必要があり，この内部で socket 操作などが定義されている．これによってカーネルで実現されている socket 操作を Ocall として Enclave 内から操作可能となり，更に用意されている ssl や crypto のヘッダーファイルを信頼可能なライブラリとしてコンパイル時にリンクすることで TLS プロトコルを使用したサーバの構築が可能となる．TLS プロトコルだけではなく，他の OpenSSL の API が多く利用可能となっており，公開鍵暗号の使用や sha256 などのハッシュ関数が利用可能である．

Enclave 内部では，TLS サーバの処理とランダムな文字列である path_index 名の導出を実行する．ランダムな文字列を導出する方法としてユーザ ID を元としたハッシュ値を用いることとした．初回アクセス時にダイジェスト値を計算し，Sealing によって暗号化を施し，Ocall によって外部関数に暗号化データを渡しファイルに保存する．ダイジェスト値の計算には EVP ヘッダーの Digest 生成を行う関数を利用し sha256 アルゴリズムで値を取得する．この際，所有者と共有者のユーザ ID に加え pepper を追加し計算を行った．取得した digest はバイト文字列であり，path_index 名として不適であるため 16 進数の文字列に変換し response する．

2 回目以降に path_index 名リクエストを受信する際には，外部に保存したファイル内容を Unsealing によって復号し，受信した所有者のユーザ ID と，共有相手の ID をキーとして該当データを探索し返送する．

6 評価

評価では，path_index 名取得の方法として隠蔽を行わず ID 名を使用して探索する方法と実装を行った TEE 内のサーバ管理を利用して取得する方法を比較する．path_index 名がリクエストされた時刻とクラウドストレージ上に存在する path_index のファイル ID を特定した時刻を計測し、差を求めた。時刻の計測はクライアントファイルシステム上で Python の time.perf_counter_ns 関数を用いて取得した。どちらも 1000 回ずつ実行し、算出した平均実行時間をミリ秒に直し有効数字 6 桁で表した (表 1)。TEE 内部の処理によって平

均 16.992 ミリ秒のオーバーヘッドが発生している。

Naming rule	TEE request
330.503	347.496

表 1　path_index 名取得時間 (ms)

7　考察

暗号化しているデータをクラウドストレージ上で共有することを実現するため，ユーザ間とファイルを紐づける物として path_index と file_index を導入した．しかしながら，共有関係に対応する path_index を容易に識別可能であること，それと同時に外部からの識別を困難にし，共有関係が容易に暴露することへの対策が必要である．

path_index とユーザを外部から容易に紐づけられることを防ぐため，path_index 名を逆算困難であるランダムな文字列とし，その計算と管理を TEE という暗号的に保護された領域で行う仕組みを実装した．これらによってファイルとその共有情報はクライアントマシン上と同等に保護された状態でクラウドストレージを介した共有が可能となっている．

クライアント上ではファイルシステムとしてファイルを利用することが可能である．保存したファイルは透過的に自動生成した公開鍵と共通鍵によって暗号化され，API を利用することでクラウドストレージへアップロードされる．これによりクラウドストレージ上では暗号化した状態で存在することとなり，機密性は保護されていると考えられる．

更に暗号化した状態のファイルを共有するため，ファイルの暗号化に使用した共通鍵を共有相手の公開鍵によって暗号化した後クラウドストレージで受け渡し，実装したファイルシステムによって共有相手自身の秘密鍵を利用した復号が透過的に行われる．そして共有相手のクライアント上で平文としてファイルを利用可能となり，共有を実現した．ファイル所有者と共有者のクライアント外では共通鍵とファイル本体は平文で存在することがなく，暗号化された状態である．そしてファイルとユーザ，共通鍵を紐づける file_index とアクセス可能なファイルの管理を行う為，path_index を導入し，共有関係に対応する path_index を識別するためのファイル名の導出を TEE 内で完結していることにより，共有関係の推察は容易ではない．

一方で利用した Intel SGX に寄る懸念点も残っており，利用可能なメモリサイズが制限されていることにより，大規模なサービス化が困難である可能性や，Sealing/Unsealing では多くのユーザからのアクセスの場合には，暗号化処理のため可用性に問題が生じる可能性がある．今回の実装では共有関係と path_index 名のマッピング情報に関するファイルの Unsealing ではメモリ上にファイルデータを展開しているためデータの量に合わせたファイルの分散やサーバの複数導入など，対策の実行が必要となる．

しかし評価で行った実験では Intel SGX の使用による path_index 名取得のために増加する処理時間は平均 16.992 ミリ秒であり、クラウド上のファイル ID の特定までの処理時間の 4.89% に留まっている。その後の path_index ファイルをクラウドからダウンロードする時間や，ファイルシステム起動時に 1 度 path_index 名を取得後ファイル ID をキャッシュすることを考慮すれば，十分に小さいと考えられる。以上によってファイルの機密性を保ったまま共有を行い，共有関係の隠蔽に関しても対策を実現したと考えられる．本研究では安全なクラウドストレージ経由のファイル共有を透過的に実現することを目的としているため，研究の目的は達成していると考える．

8　結論

本研究では FUSE を用いて実装したデーモン内で透過的な暗号処理とクラウドストレージの操作を行うことによって，機密性を保持したクラウド環境にあるストレージの利用を可能とした．更に暗号化鍵とファイル，ユーザのマッピングを行う file_index と path_index の導入によりクラウド上に存在するファイルの共有を暗号化ファイルの共有を行う．共有情報である path_index と共有関係のクラウド上ファイルの閲覧による推測に対しては隔離された実行環境である TEE を用いることによって秘匿性を高めている．

文　献

[1] Mohamad Ahtisham Wani. *Privacy Preserving Anti-forensic Techniques*, pp. 89–108. Springer Singapore, Singapore, 2021.

[2] Osama A. Khashan. Parallel proxy re-encryption workload distribution for efficient big data sharing in cloud computing. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0554–0559, 2021.

[3] vgough. encfs. `https://github.com/vgough/encfs`.

[4] D. Leibenger, J. Fortmann, and C. Sorge. Encfs goes multi-user: Adding access control to an encrypted file system. In *2016 IEEE Conference on Communications and Network Security (CNS)*, pp. 525–533, 2016.

[5] Osama Ahmed Khashan. Secure outsourcing and sharing of cloud data using a user-side encrypted file system. *IEEE Access*, Vol. 8, pp. 210855–210867, 2020.

[6] Chunhua Xiao, Lei Zhang, Weichen Liu, Linfeng Cheng, Pengda Li, Yanyue Pan, and Neil Bergmann. Nv-ecryptfs: Accelerating enterprise-level cryptographic file system with non-volatile memory. *IEEE Transactions on Computers*, Vol. 68, No. 9, pp. 1338–1352, 2019.

[7] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez 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, Santa Clara, CA, February 2017. USENIX Association.

[8] fusepy. `https://github.com/fusepy/fusepy`.

[9] intel. `https://github.com/intel/linux-sgx`.

[10] sparkly9399. Sgx-openssl. `https://github.com/sparkly9399/SGX-OpenSSL`.

[11] インテル ソフトウェア・ガード・エクステンションズによるデータ保護. `https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/software-guard-extensions-enhanced-data-protection.html`.