システムソフトウェアと
オペレーティング・システム 90−10
（２００２．６．２７）

データの機密性と完全性を保護する ネットワークファイルシステムの設計と実装

藤田 智成 †　　小河原 成哲 †

†NTT 未来ねっと研究所
{fujita,ogawara}@exa.onlab.ntt.co.jp

ストレージリソースをアウトソーシングすることでコストを削減する企業が増えてきているが，既存のファイルシステムは，その管理者がユーザに気づかれることなく，保存されているデータを読むことに加え，変更することすら可能であるため，データの保護が不十分であるという新たな問題が生じている．

本稿では管理者を含めた全ての第三者に対して，対称鍵暗号を用いてデータの機密性，非対称鍵暗号を用いて完全性を保護するネットワークファイルシステムの設計と，その性能測定結果について報告する．中規模のソフトウェアパッケージのコンパイルという負荷に対して，実装したファイルシステムの性能を測定したところ，NFS version3 と比較して，そのオーバヘッドは 72.1%であった．

The design and implementation of network file system providing confidentiality and data integrity

Tomonori Fujita†　　Masanori Ogawara†

†NTT Network Innovation Laboratories

The number of companies outsourcing data management, and hiring outside consultants to administer servers has increased. In these cases, concern for data security grows because those who manage data on common file systems also have the ability to read it or even change it in subtle, difficult to detect ways.

This paper presents the filesystem providing strong confidentiality and data integrity guarantees by means of symmetric and asymmetric key cryptography. Compared with NFS version 3, an implementation takes 72% longer to compile the OpenSSH program, a medium sized software package.

1　はじめに

近年，企業では外部組織のストレージリソースをネットワークを介して利用する状況が増加している．ストレージリソースをアウトソーシングすることで，高額なストレージ機器への投資コストや，定期的なデータのバックアップ等の管理コストが削減される．

外部組織のストレージリソースを利用することで，サーバシステムとストレージが同一管理権限下において管理されていないという，これまでなかった状況となり，ストレージに保存されているデータに関して新たな問題が生じる．

我々はデータをストレージに保存する際に，そのデータが適切に保護されることを期待する．しかし，通常のファイルシステムはその管理者に対して，保存されているデータの機密性[*1]，完全性[*2] について何の保護機能も持たない．実際，管理者は我々に気づかれることなく，保存されているデータを読むことに加え，変更することすら可能である．言い換えると，我々は保存したデータに関して，その管理者を完全に信頼することを強制されているのである．

ストレージを自組織内で管理している状況では，その管理者を完全に信頼したとしても大きな問題とはな

[*1] データの内容を許可した者以外に不正に読まれないこと．
[*2] データの内容が変更された場合に検出可能であること．

らない．しかし，外部組織がストレージを管理している状況下でその管理者を完全に信頼することは，保存するデータの安全性に問題を抱えることになる．

上記の問題を解決するため，我々はストレージの管理者を含めた全ての第三者から，データの機密性と完全性を保護することのできるファイルシステム，Signed file system（以下，SignedFS）の研究を進めている．SignedFS は従来と同様のファイルオペレーションを提供するネットワークファイルシステムであり，対称鍵暗号を用いてデータの機密性，非対称鍵暗号とハッシュ関数を用いて完全性を保護する．

本稿では SignedFS の設計とその実装について説明し，性能測定の結果を述べる．

本稿の構成は以下のとおりである．まず，2 節で SignedFS と既存ファイルシステムとの関連について述べる．3 節では SignedFS の詳細な設計について，4 節ではその実装について述べる．5 節では実装した性能測定の結果を示す．最後に 6 節でまとめを行う．

2 関連研究

データの機密性を保護するファイルシステムには CFS[2], TCFS[3], Cryptfs[13] など，これまで多くの研究例がある．これらのファイルシステムは一般に暗号化ファイルシステムと呼ばれ，データをストレージに保存する前に対称鍵暗号を用いてデータを暗号化することで機密性を保護する．暗号化ファイルシステムはデータの機密性を保護するが完全性は保護しない．

ファイルシステムの機密性に関する研究例と比較すると，その完全性について取り組んだ研究例の数は限られている．PFS[10] と Cepheus[4] はメッセージ認証コードを用いてデータの完全性を保護するファイルシステムである．

メッセージ認証コードを用いて，ファイルシステムのデータの完全性を保護する方法には本質的な制限がある．メッセージ認証コードは，完全性を確認するためのコードを生成する者と完全性を確認する者が同じ鍵を利用するため，データを自分以外の第三者に公開する際には，その鍵をその相手に公開しなければならない．従って，鍵を受け取った第三者とファイルシステムの管理者が共謀することで，ファイルを変更した後に，鍵を使って完全性を確認するためのコードを生成し，変更を加えた不正なデータを正当なものであるかのように偽装することができる．

SignedFS が採用している非対称鍵暗号を用いて完全性を保護する方法は，データの作成者と完全性を確認する者が，それぞれ秘密鍵，公開鍵という二種類の異なる鍵を用いるため，上記のような制限は生じない．

SignedFS と同様に非対称鍵暗号を用いて完全性を保護するファイルシステムの研究例は SFSRO[5] と SUNDR[6] がある．SFSRO はセキュアなソフトウェア配布を目的とした読み取り専用のファイルシステムであり，従来同様のファイルオペレーションを提供する SignedFS とは目的が異なる．また，SUNDR は設計の提案に留まっている．

3 設計

3.1 機密性

前述のように機密性を保護する暗号化ファイルシステムの研究例はこれまでに幾つかあるが，データブロックを暗号化することで機密性を保護するという点は共通している．

ネットワークファイルシステムとして用いることのできる暗号化ファイルシステムは，書き込み時はユーザが使用するクライアントでデータを暗号し，その暗号化されたデータをサーバに保存する．また，読み取り時は暗号化されているデータをサーバに要求し，クライアントで復号化することで元の情報を復元する．従って，クライアントの外部に暗号化されていない情報が漏れることはなく，管理者を含めた全ての第三者からデータの機密性が保護される[*3]．

4 節で述べるが，SignedFS の機密性を保護する機能は，前述の Cryptfs を利用している．Cryptfs は上述のようにデータブロックを暗号化することで機密性を保護しているが，ディレクトリに関しては，データブロックを直接暗号化するのではなく，そのディレクトリに含まれるファイルシステムオブジェクト名を暗号化することで，その機密性を保護している．

3.2 完全性

3.2.1 通常ファイル

SignedFS は電子商取引等の分野で完全性を保護するために用いられる方法の一つであるデジタル署名を個々のファイル[*4]に適用することで，ファイルデータの完全性を保護する．

[*3] 鍵の紛失等の緊急の事態を想定し，管理者が特別な鍵でデータを復号化できる暗号化ファイルシステムもある．

[*4] 以後，単にファイルと表記した場合は通常ファイルを指すものとする．

具体的には，ファイルデータをオペレーティングシステムのページキャッシュの大きさに分割し，ハッシュ関数を適用することで，各々のブロックのメッセージダイジェストを求め，それらの値を組み合わせることでファイルデータ全体のメッセージダイジェストを求める．そして，求めたファイルデータ全体のメッセージダイジェストを非対称鍵暗号の秘密鍵で暗号化[*5]した値 (デジタル署名) が inode 構造体内に保存されている．

ファイルデータ全体にハッシュ関数を適用し，メッセージダイジェストを求めるのではなく，ファイルデータを分割してから，それぞれのメッセージダイジェストを求める設計は前述の Cepheus と同様である．ファイルデータ全体のメッセージダイジェストを求める方法では，ファイルの変更が一部分であっても，再度ファイルデータ全体のメッセージダイジェストを求める必要が生じるため，性能に悪影響を与える．

ファイルデータを分割してから，それぞれのメッセージダイジェストを求める設計では，求めた複数のメッセージダイジェストから全体のメッセージダイジェストを算出する方法が問題となる．Cepheus は求めた複数のメッセージダイジェストを木構造にして再帰的にメッセージダイジェストを計算する方法 [7] を用いているが，SignedFS は incremental hash 関数 [1] を用いている．これは，前者よりも後者の方法が高速であるためである．

以下，ファイルの完全性の保護に関連する基本操作について説明する．

Open　ファイルをオープンした時にファイルデータ全体の完全性を確認する[*6]．全てのファイルデータをサーバに要求し，ページキャッシュの大きさに分割し，各々のメッセージダイジェストを求める．この際，各々のメッセージダイジェストはページキャッシュを管理する構造体内に保存される（図 1）．分割したファイルデータの各々のメッセージダイジェストを求めた後に incremental hash 関数を用いてファイルデータ全体のメッセージダイジェストを生成する．この求めたメッセージダイジェストとサーバから別途受け取るデジタル署名[*7]を非対称鍵暗号の公開鍵を使って復号することで得られた値が一致すれば，ファイルデータの完全性を確認することができる．

Write　ファイルデータをページキャッシュの大きさに分割し，各々のメッセージダイジェストを求め，incremental hash 関数を用いてファイルデータ全体のメッセージダイジェストを生成し，署名する．得られたデジタル署名はサーバに保存し，オープン時にファイルデータの完全性を確認するために用いられる．また，分割されたページデータの各々のメッセージダイジェストはファイルデータが書き換えられる場合に備えて，ページキャッシュ管理用の構造体に保存される．

ファイルデータの一部が変更された場合は，対応するページキャッシュのメッセージダイジェストを再度求めてから，incremental hash 関数とページ構造体内に保存してあるデータ変更前のメッセージダイジェストを用いてファイルデータ全体のメッセージダイジェストを算出することができる．

Read　ファイルオープン時に全てのファイルデータを要求し，その完全性を確認するため，読み取り時に改めて確認する必要はない．但し，クライアントのページキャッシュからファイルデータを落ちていた場合は，再度サーバからファイルデータを受け取り，ファイルオープン時の動作と同様に完全性を確認する．

3.2.2　ディレクトリ

ディレクトリの完全性を保護する方法はファイルのそれとは異なる．SignedFS が NFS をベースに実装されていることが，その主たる理由である．

NFS でのディレクトリの扱いは，サーバがデータブロックをディレクトリとして解釈し，クライアントはディレクトリに含まれるファイル，ディレクトリなどのファイルシステムオブジェクトの名前をサーバから受け取る．従って，ファイルの完全性を保護する方法のように，ディレクトリのデータブロックを未加工のまま受け取って，デジタル署名を求めることはでき

[*5] 以降，ハッシュ関数を用いて求めたメッセージダイジェストを非対称鍵暗号の秘密鍵で暗号化する操作を「署名する」と表記する．

[*6] サーバから受け取るファイルデータは暗号化されており，それを復号化した後に完全性の確認作業が行われる．

[*7] ファイルの最終更新時に生成する．書き込み操作の説明を参照のこと．

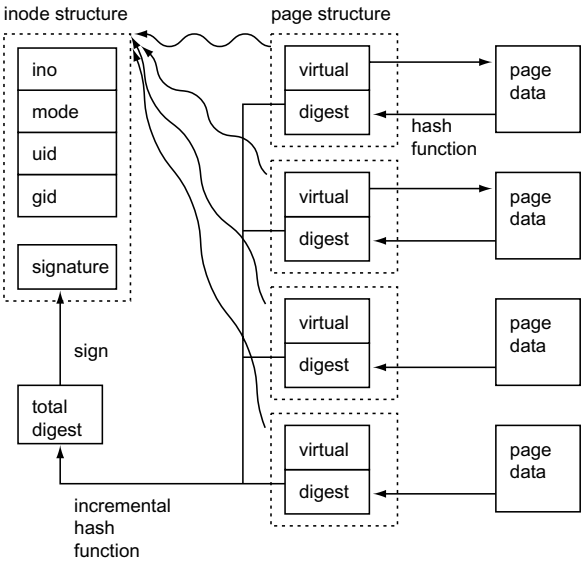

図 1　通常ファイルの完全性

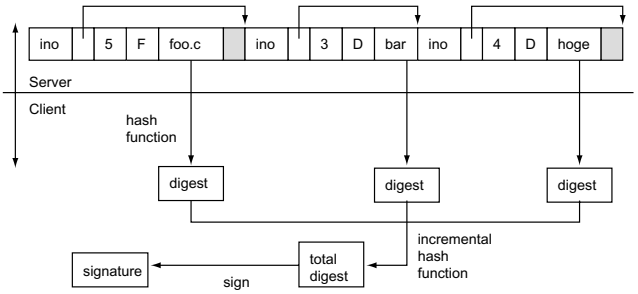

図 2　ディレクトリの完全性

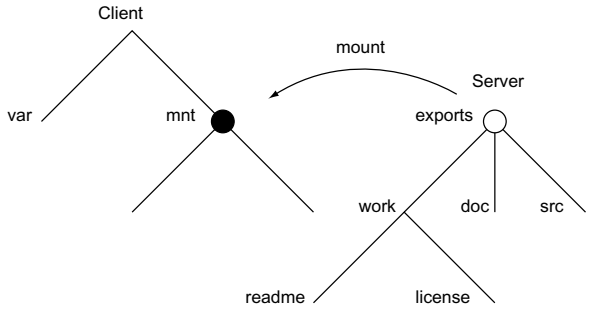

図 3　ディレクトリ構造例

ない．ファイルの完全性を保護する方法でディレクトリの完全性を保護するためには，ディレクトリのデータブロックの解釈をサーバではなくクライアントがデータブロックをディレクトリとして解釈するように NFS を変更することが必要となる．我々は NFS のディレクトリ解釈機能の変更を避け，ディレクトリに含まれるファイルシステムオブジェクト名ごとにメッセージダイジェストを求める方法を採用した．

まず，クライアントはディレクトリに含まれる全てのファイルシステムオブジェクトの名前をサーバから受け取り，それぞれの名前のメッセージダイジェストを求める．次に，求めた各々のメッセージダイジェストに incremental hash 関数を適用し，ディレクトリ全体のメッセージダイジェストを生成し，署名したものを inode 構造体内に保存する (図 2)[*8]．

以下，ディレクトリの完全性の保護に関連する基本操作について，図 3 に示すディレクトリ構造下で

```
open("/mnt/work/readme")
```

というファイルオープン動作を行った例を用いて説明する．

図 3 はクライアントがローカルディレクトリ */mnt* にサーバがエクスポートしているディレクトリをマウントしている状態を表しているため，完全性の確認が行われるパス名は *work* からになる．

1. パス解析中に未だ完全性を確認していないパス名 *work* に出会うと，親ディレクトリに当たる *exports* に含まれる全てファイルシステムオブジェクト名をサーバに問い合わせる．
2. サーバから *exports* に含まれる *work*，*doc*，*src* という 3 つのファイルシステムオブジェクト名を受取り，各々の名前ごとのメッセージダイジェストを求めた後は，incremental hash 関数でディレクトリ全体のメッセージダイジェストを求める．
3. 次にサーバから *exports* の inode 構造体内に保存されているデジタル署名を受取り，非対称鍵暗号の公開鍵を使って復号し，得られた値と (2) で求めたディレクトリ全体のメッセージダイジェストが一致すれば完全性を確認することができる．
4. パス名 *work* の完全性の確認が終了したので，次にパス名 *readme* の完全性の確認に進む．今度はディレクトリ *work* に含まれるファイルシステムオブジェクト名全てをサーバに問い合わせ，*readme* と *license* の 2 つの名前を受け取り，というように (2)〜(3) で説明した手順と同様にパス名 *readme* の完全性の確認が行われる．

[*8] 図中ではファイルシステムオブジェクト名が平文で示されているが，実際は暗号化されたた名前がサーバに保存されており，クライアントでファイルシステムオブジェクト名を復号化した後にメッセージダイジェストを求める．

表 1　暗号基礎操作に必要な時間

操作	時間 (μs)
Rijndael encrypt 4KB	100.639
Rijndael decrypt 4KB	83.376
SHA1 Hash 4KB	60.852
RSA Sign 20B	4205.347
RSA Verify 20B	55.591

ファイルシステムオブジェクトの新規作成，削除，名前の変更があった場合は，変更後のディレクトリのデジタル署名を求めて，inode 構造体内に保存する．その手順はページキャッシュの大きさに分割されたファイルデータがファイルシステムオブジェクト名に置き換わっていることを除けば，ファイルの完全性を保護する方法と同様である．

3.2.3　delayed signature

非対称鍵暗号は対称鍵暗号と比べて動作が低速である．従って，非対称鍵暗号処理の回数はファイルシステムの性能に大きな影響を与える．表 1 に SignedFS が利用する基本的な暗号処理の性能を示した．測定結果は AMD Athlon 1900+(1600MHz) を搭載したコンピュータで行われたものである．

SingedFS は機密性を保護するために，対称鍵暗号 rijndael[12] を鍵の長さ 128bit で用いている．また，完全性を保証するために，ハッシュ関数 SHA1[11] と非対称鍵暗号 RSA[8] を鍵の長さ 1024bit で用いている．

事前の暗号基礎操作の性能測定結果は，非対称鍵暗号処理が大きなオーバーヘッドになることを示しており，我々は書き込み時の性能を向上させるために非対称鍵暗号処理の回数を削減する方法を検討した．

最初に検討した方法は，ファイル毎に非対称鍵暗号処理を行うのではなく，ブロックサイズなど，より大きな単位に対して非対称鍵暗号処理を行うことで，その回数を削減するものである．しかし，この方法は完全性を保証する単位がファイルよりも大きくなり，筆者らが SignedFS の利用環境の一つとして想定している，異なる組織に所属するユーザ同士のファイルの共有 [14] を困難なものとする．

SignedFS ではファイル単位での完全性の保証という点を重視し，非対称鍵暗号処理を遅延させる方法を採用した．短時間に同一のファイルが何度も変更される場合は非対称鍵暗号処理を遅延させることで，その回数を削減することができる．

暗号処理を遅延させると，クライアントとサーバ間でデータが同期していない状態が長くなるので，障害発生時にファイルシステムの整合性に問題が生じる確立が高くなる．SignedFS の暗号処理の方針は，メタデータが同期処理，ファイルデータが遅延処理である．メタデータに関する暗号操作を同期処理としたのは，障害時のメタデータの不整合によって生じる，ファイルシステムの大きな被害を避けるためである．

ファイルデータの遅延処理は，NFS version3 の不安定な書き込み，と呼ばれるセマンティクスを利用している．このセマンティクスでは，サーバはクライアントからのファイルデータの書き込み要求を受け取った際に，変更があったバッファとストレージの同期を取る前に，クライアントに応答を返すことで性能を向上させる．クライアントは複数の書き込み要求を発行した後に，コミットと呼ばれる命令を発行することで，サーバでのバッファとストレージの同期を要求，確認することができる．

クライアントはサーバからバッファとストレージの同期完了をコミット命令の応答として受け取るまで，書き込みの内容を保存する．従って，サーバに障害が発生した場合には，サーバの再起動後書き込みの内容を再送し，書き込み内容が失われることを防ぐことができる．一方，クライアントに障害が発生した場合は，サーバからの最終同期完了応答以降の書き込みの内容は保証されない．

SignedFS ではこのセマンティクスを利用し，クライアントが複数の書き込みを行い，コミット命令を発行するまで，非対称暗号処理を遅延させる．コミット命令の直前に非対称鍵暗号処理を実行し，デジタル署名とコミット命令を一緒にサーバに送信する．

4　実装

我々は SignedFS を Linux kernel(Version 2.4.18) 上に実装した．その構成要素について説明する．

Cryptfs　機密性を保護する部分は Cryptfs の機能を利用している．その主な理由は，Cryptfs が Stacakable vnode[13] 機構を用いて設計されており，他の機能と組み合わせることが容易であるためである．

SignedFS Client/Server　NFS version3 プロトコルに完全性を保護する機能を追加し，kernel nfs daemon，nfs filesystem をベースに実装した．

SignedFS　サーバが用いるローカルファイルシステムは ext2 filesystem をベースに実装されている．具体的には，デジタル署名を保存するための inode 構造体の拡張があげられる．

5　性能

本節では実装した SignedFS の性能測定結果を示す．我々は SignedFS と，その実装のベースとなっている NFS verison3 の性能を 3 種類のベンチマークで比較した．

5.1　実験環境

測定に用いたコンピュータは，サーバ，クライアント共に CPU に AMD Athlon 1900+，1Gbps のイーサネットカード (3COM 3C996B-T)，512MB のメモリ，Seagate ST380021A 7200rpm IDE disk を搭載した同一のものである．サーバとクライアントは 1Gbps, full-duplex のスイッチで接続されている．

5.2　Micro-benchmark

各ファイルシステムの性能の特性を調べるために，LFS ベンチマーク [9] を実行した．LFS ベンチマークは多数の小さなサイズのファイルと 1 個の大きなサイズのファイルに対する操作という，対照的な 2 種類の操作のベンチマークから成り立っている．これらのベンチマークでは，前者がメタデータ，後者がファイルデータが主な操作対称となる．両者の負荷共に一般ユーザの使用環境で生成される負荷とは異なるが，ファイルシステムの性能の特性をよく表すものとなる．

これらのベンチマークの目的は暗号処理がファイルシステムの性能に与える影響を測定することである．そこで，サーバでのハードディスク操作によるオーバヘッドを測定時間から取り除くため，読み取りはサーバはベンチマークの対象となるデータをメモリにキャッシュしている状態からベンチマークを開始した．また，書き込みはサーバは更新されたバッファとストレージを同期させずにベンチマークを実行した．

なお，以下で示すそれぞれのベンチマークの結果は 5 回の試行の平均値である．

5.2.1　Small file benchmark

Small file benchmark はサイズが 1KB のファイル 4096 個に対する読み取り・書き込みに要する時間を測定するものである．ファイルはディレクトリごとに 32 個割り当てられている。読み取り，書き込みに要した時間を表 2 に示す．

表 2　Small file benchmark の測定結果

	読み取り (s)	書き込み (s)
NFS	2.7520	6.3501
SignedFS	3.7564	42.8967

表 3　Small file benchmark の暗号処理時間 (読み取り)

操作	時間 (μs)	回数	合計時間 (ms)
Decrypt 1KB	25.228	4096	103.33
Hash 1KB	15.315	4096	62.730
Decrypt 10B	0.476	4232	2.02
Hash 10B	0.221	4232	0.937
Verify 20B	55.591	4227	234.98

読み取りでは SignedFS は NFS と比較して，1.364 倍遅かった．その性能差の要因を明らかにするため，暗号処理によるオーバヘッドを解析した．

Small file benchmark の読み取りで実行された基礎暗号操作の種類とその回数を計測し，事前に測定した基礎暗号操作の処理時間と掛けた値を各々の暗号処理に要した時間とした．その結果を表 3 に示す．表の上から 2 項目がファイルデータの復号とハッシュ処理，次の 2 項目がディレクトリデータの復号，ハッシュ処理，最終項目が非対称鍵暗号の復号処理である．ファイルシステムオブジェクト名は平均の長さを 10 バイトとして計算した．

SignedFS の暗号処理以外のオーバヘッドとしては，サーバの inode に保存されているデジタル署名を要求する処理がある．この処理は NFS の独立したプロシージャとして実装されており，要求と応答に要する合計時間を測定したところ 175.938μs であった．Small file benchmark の読み込みでは，デジタル署名の要求が 4227 回あるため，要する時間は合計で 743.688ms となる．

SignedFS と NFS の性能差は 1.0044s であり，暗号処理とデジタル署名要求処理に要する時間 1.1477s とほぼ等しい．

書き込み時に実行される基礎暗号処理に関する情報を読み取り同様に表 4 にまとめた．SignedFS の NFS 性能差は 36.5466s であり，暗号処理に要する時間 35.1611s とほぼ等しい．

5.2.2　Large file benchmark

Large file benchmark はサイズが 40MB のファイル 1 個に対するシーケンシャルな読み取り・書き込みに要する時間を測定するものである．その結果を表 5 に

表 4　Small file benchmark の暗号処理時間 (書き込み)

操作	時間 (μs)	回数	合計時間 (ms)
Encrypt 1KB	25.105	4096	102.83
Hash 1KB	15.315	4096	62.730
Encrypt 10B	0.458	4232	1.94
Hash 10B	0.221	4232	0.935
Sign 20B	4205.347	8321	3.499269×10^4

表 5　Large file benchmark の測定結果

	読み取り (s)	書き込み (s)
NFS	1.2556	1.4186
SignedFS	2.5308	3.2694

示す.

Small file benchmark と同様に，読み取りで実行される暗号処理に関する情報を表 6 に示す．SignedFS と NFS の性能差は 1.2752s であり，暗号処理時間の合計 1.4770s と近い値となっている.

書き込みで実行される暗号処理に関する情報は表 7 に示す．SignedFS と NFS の性能差 1.8508s は，暗号処理時間の合計 1.8260s と非常に近い値である.

表中では署名処理の回数が 41 回となっているが，本来はページ単位での書き込み回数と同じく 10240 回に，ファイルの新規作成によるディレクトリへの署名の 1 回を加えた 10241 回となるはずである．回数が減っている要因は，3.2.3 節の delayed signature 機能によって，署名処理を遅延しているためである.

SignedFS では，コミット命令と一緒に署名処理を行うため，40 回の署名処理は，256 回 (= 10240 ÷ 40) の書き込み命令に 1 回の割合でコミット命令が発行されていることを意味する．3.2.3 節で説明したように，コミット命令を発行する割合を小さくすると署名処理の回数が減少して，さらに性能は向上するが，障害発生時の被害は大きくなる.

Linux の kernel nfs のデフォルトの設定では，書き込み命令が 256 個を越える (リクエスト数超過)，または，書き込み命令後 5 秒以上経過する (遅延書き込みタイムアウト)，のどちらかの条件が満たされた時にコミット命令を発行する．このベンチマークの結果では，256 回の書き込み命令に 1 回の割合でコミット命令が発行されていることから，リクエスト数超過によって全てのコミット命令が発生している.

delayed signature 機能によって署名処理を遅延しない場合を考えると，このベンチマークでは署名処理が 10241 回実行され，それに必要な時間は 43.06696s である．署名処理以外の暗号処理時間を加えると約 46s となり，SignedFS は NFS に比較して 32 倍程度遅くなることが予想される.

表 6　Large file benchmark の暗号処理時間 (読み取り)

操作	時間 (μs)	回数	合計時間 (ms)
Decrypt 4KB	83.376	10240	853.77
Hash 4KB	60.852	10240	623.12
Decrypt 10B	0.476	1	4.76×10^{-4}
Hash 10B	0.221	1	2.21×10^{-4}
Verify 20B	55.591	2	0.11118

表 7　Large file benchmark の暗号処理時間 (書き込み)

操作	時間 (μs)	回数	合計時間 (ms)
Encrypt 4KB	100.639	10240	1030.54
Hash 4KB	60.852	10240	623.12
Encrypt 10B	0.458	1	4.58×10^{-4}
Hash 10B	0.221	1	2.21×10^{-4}
Sign 20B	4205.347	41	172.4192

5.3　Software compile benchmark

ユーザの日常的な使用状況でのファイルシステム性能の指標として，ソフトウェアパッケージ (OpenSSH version 3.0.2) のアーカイブの展開，コンフィギュレーション，コンパイルに要する時間を各々測定する.

展開　OpenSSH のソースツリーを含む圧縮された tar アーカイブを展開する.

コンフィグレーション　コンパイルの段階で用いるための Makefile を生成する configure スクリプトを実行する．なお，この段階で参照される include ファイルやライブラリはクライアントのローカルファイルシステム (ext2) 上にある.

コンパイル　前段階で生成した Makefile を実行して，OpenSSH の実行ファイルを構築する.

このベンチマークでは，ユーザの日常的な使用状況を想定しているため，サーバでのストレージ操作による影響を含めて測定した．読み取りはベンチマークの対象となるデータがキャッシュにのっていない状態，書き込みではサーバは 3.2.3 節で説明した NFS version3 の不安定な書き込みと呼ばれるセマンティクスに従って，更新されたバッファとストレージを同期させる.

3 つの段階は連続して実行されるため，コンフィギュレーション，コンパイルの段階はページキャッシュが利用される.

表 8　OpenSSH のコンパイルに要した時間 (*s*)

	Unpack	Config	Build	Total
NFS	3.0352	27.7868	31.1092	61.9312
SignedFS	4.2670	47.0690	55.2558	106.5918

ベンチマークの結果を表 8 に示した．作業全体が完了するまで要した時間は，SignedFS は NFS と比較して 1.7211 倍遅かった．

このベンチマークは比較的メタデータを変更する操作が多いことから，Small file benchmark の 6.755 倍程度性能が低下することを予想していたが，実際の性能低下は 1.7211 倍に留まっている．これはサーバでのストレージ操作が大きなボトルネックとなり，暗号処理による性能低下が隠されているものと考えられる．

6　まとめ

本稿では，対称鍵暗号を用いてデータの機密性，非対称鍵暗号を用いて完全性を保護するネットワークファイルシステムの設計と，その性能測定結果について報告した．本ファイルシステムはデータを管理する者を含めた全ての第三者に対して，データの機密性，完全性の保護機能を提供する．

我々の行った実験によると，暗号処理がファイルシステムの性能へ与える影響は，書き込み時に必要となる非対称鍵暗号の暗号化処理が大きなオーバヘッドとなり，メタデータ中心の負荷で 6.8 倍，ファイルデータ中心の負荷で 32 倍程度，NFS と比べて遅くなることが分かった．ファイルデータ中心の負荷に対しては，非対称鍵暗号処理を遅延させる手法を適用することで，性能の低下を 2.3 倍に抑えることができた．

また，実装したファイルシステム上で中規模のソフトウェアパッケージ (OpenSSH 3.0.2) のコンパイルに要する時間を測定したところ，メタデータ中心の負荷であるにも関わらず，NFS version3 と比較して 1.721 倍であった．このことより，現在の高速な CPU を使うことで，サーバでのストレージ操作で発生する待ち時間に暗号処理を行い，そのオーバヘッドを隠蔽することが可能になりつつあることが確認できた．

参考文献

[1] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography and application to virus protection. In *the 27th ACM Symposium of the Theory of Computing*, pp. 45–56, May 1995.

[2] M. Blaze. A cryptographic file system for unix. In *1st ACM Conference on Communications and Computing Security*, pp. 158–165, 1993.

[3] G. Cattaneo and G. Persiano. Design and implementation of a transparent cryptographic file system for unix. Technical report, Dip. Informatica ed Appl, Università di Salerno, Jul 1997.

[4] Kevin Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, May 1999.

[5] Kevin Fu, Frans kaashoek, and David Mazieres. Fast and secure distributed read-only file system. In *4th USENIX Symposium on Operationg System Design and Implementation(OSDI 2000)*, October 2000.

[6] D. Mazières and Dennis Shasha. Don't trust your file server. In *The 8th Workshop on Hot Topics in Operating Systems*, 2001.

[7] R.C. Merkle. A digital signature based on a conventional encryption function. In *Advance in Cryptology - Crypto '87*, No. 293 in Lecture Notes in Computer Science, pp. 369–378, Berlin, 1987.

[8] R.L. Rivest, A. Shamir, and L.M. Adleman. A mechod for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, Vol. 21, No. 2, pp. 120–126, Feb 1978.

[9] M. Rosenblum and J. Outsterhout. The design and implementation of a log-structured file system. In *the 13th ACM Symposium on Operating Systems Principles*, pp. 1–15, CA, October 1991.

[10] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying file system protection. In *2001 USENIX Annual Technical Conference*, pp. 79–90, June 2001.

[11] National Institute of Science United State of American and Technology. Secure hash standard. Federal Information Processing Standard 180-1, April 1993.

[12] National Institute of Science United State of American and Technology. Announcing the advanced encryption standard(aes). Federal Information Processing Standard 197, November 2001.

[13] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical report, Computer Science Department, Columbia University, Jul 1998.

[14] 藤田智成, 小河原成哲. 機密性・完全性を保証するファイルシステムにおけるアクセス管理機構方法の提案. 情報処理学会第 64 回全国大会, pp. 11–12, 2002.