平成 13 年度

学士学位論文

秘密分散法のファイルシステムへの応用

An Application of Secret Sharing Scheme to UNIX File System

1020290　　澤野 充明

指導教員　　菊池 豊

2002 年 2 月 15 日

高知工科大学 情報システム工学科

要 旨

秘密分散法のファイルシステムへの応用

澤野 充明

情報に秘匿性や対災害性を持たせて保管する方法に秘密分散法 (SSS: Secret Sharing Scheme) がある。現在、SSS の実装として UNIX 上の ssar コマンドがある。ssar コマンドはユーザに SSS を利用していることを意識させてしまうという問題点がある。本研究の目的はユーザに SSS を意識させないようなファイルの保管手法を与えることである。今回は、SSS を UNIX ファイルシステムへ応用することの提案を行い、それの実装に必要な暗号化/復号化モジュールの作成を行った。

キーワード　秘密分散法、対災害性、秘匿性、UNIX ファイルシステム

Abstract

An Application of Secret Sharing Scheme to UNIX File System

SAWANO Mitsuharu

SSS (Secret Sharing Scheme) is a method to data storages to have disaster prevent property and secarity. There is a command named ssar implemented on UNIX. The command has a problem that user must be conscious of SSS. The purpose of this research shows a method for file storage that users are not conscious of SSS. I made proposal of the application of SSS to UNIX File System. I created the encrypt/decrypt module that necessary to implementation of SSS.

key words Secret Sharing Scheme, disaster prevention property, Security, UNIX File System

目次

第 1 章　はじめに **1**

第 2 章　秘密分散法 **3**

2.1　秘密分散法とは 3

2.1.1　(k, n) 閾値 SSS 4

2.1.2　(k, n) 閾値 SSS のアルゴリズム 4

分散符号化 4

分散復号化 5

分散符号化と復号化の簡単な例 7

2.2　SSS を利用した実装 9

第 3 章　UNIX ファイルシステム **10**

3.1　UNIX ファイルシステムの概要 10

3.2　スタッカブルファイルシステム 11

3.2.1　Null ファイルシステム 12

3.2.2　Null ファイルシステムの処理 13

第 4 章　SSS のファイルシステムへの応用 **16**

4.1　Secret Sharing File System 16

4.2　SSFS の設計 17

4.2.1　分散符号化/復号化の処理 17

4.2.2　シェアの保存の方法 18

4.3　分散符号化/復号化モジュール 21

4.3.1　仕様 21

4.3.2　分散符号化モジュール 22

4.3.3 復号化モジュール 22

第 5 章 暗号化/復号化モジュールの動作テスト 24

5.1 復元性 24

5.1.1 任意のファイルでも暗号化/復号化することができるか 24

5.1.2 任意のシェアファイル 2 個から復号化することができるか 26

5.2 エラー処理 27

5.2.1 シェアファイルが 0 個、1 個のときはエラーを返すか 27

第 6 章 まとめ 29

6.1 まとめ 29

6.2 今後の課題 29

謝辞 31

参考文献 32

付録 A 暗号化/復号化モジュール 33

A.1 暗号化モジュール 33

A.2 復号化モジュール 34

図目次

2.1 分散符号化 3
2.2 復号化 4

3.1 Virtual File System 11
3.2 スタッカブルファイルシステム 12
3.3 mount_null 前の UNIX ファイルシステム階層構造 14
3.4 mount_null 後の UNIX ファイルシステム階層構造 15
3.5 Null ファイルシステム 15

4.1 Secret Sharing File System 18
4.2 分散符号化モジュールの返値 22
4.3 復号化モジュールの返値 (成功) 23
4.4 復号化モジュールの返値 (失敗) 23

5.1 実行結果 1 25
5.2 実行結果 2 25
5.3 実行結果 3 27
5.4 ssfs_decode23_error 28

表目次

4.1 Null ファイルシステムのオペレーション評価 18

第 1 章

はじめに

今日では、ありとあらゆる情報がコンピュータ内に保存されている。その情報はデジタル化されているため、検索や編集が容易に行えるという利点を持っている。

情報を保存しているコンピュータに人為的な破壊や自然災害などでシステムに支障が起きると、情報が消滅して取り出せなくなってしまう可能性がある。このような情報の消失の解決策に、データのコピーをとり複数のコンピュータに分散して保存しておく方法がある。重要な情報の場合、一ヶ所だけでなく複数の場所のコンピュータにコピーをとっておくとよい。もし、システムに支障が起きた場合にはコピーから情報を取り出す。そうすることによってこれらの危険から回避することが出来る。しかし、秘密情報のコピーを作るということはそれだけで危険である。なぜなら、コピーを作成することにより、より他人に秘密情報を見られる危険性を増加させるからである。

十分な秘匿性や対災害性をもつ情報の保管の方法に秘密分散法 (Secret Sharing Scheme) がある。SSS を利用することによって秘匿性や対災害性をもたせて情報を保管できるようになる。現在、SSS の実装として UNIX 上のコマンド ssar がある。ssar コマンドはユーザに SSS 機構の存在を意識させてしまうという問題点がある。

そこで、本研究では、ユーザに SSS 機構を意識させないようなファイルシステムを提案する。今回は、SSS を UNIX ファイルシステムへ応用することの提案と暗号化/復号化モジュールの作成を行った。

本論文では、第 2 章で秘密分散法の理論について説明を行う。また、UNIX のファイルシステムへ応用するため、基礎知識として第 3 章で UNIX ファイルシステムに関する説明を行う。第 4 章で秘密分散法のファイルシステムへの応用手法について説明し、第 5 章で暗号

化/復号化モジュールの動作テストの結果、第 6 章でまとめを述べる。

第 2 章

秘密分散法

本章では、まず秘密分散法の特性について説明を行う。次に、本研究で使用する Shamir により提案された (k, n) 閾値 SSS について説明を行う。最後に、既存の秘密分散法の実装である ssar について説明する。

2.1　秘密分散法とは

秘密分散法 (SSS: Secret Sharing Scheme) とは 1979 年に Shamir[5] と Blakley[1] により独立に提案された符号化の理論である。秘密分散法には Shamir により提案された (k, n) 閾値 SSS などがある。

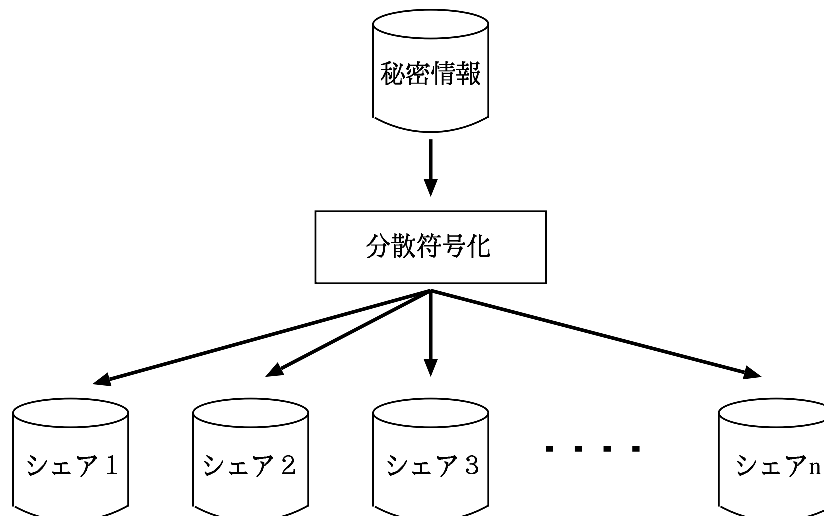

図 2.1　分散符号化

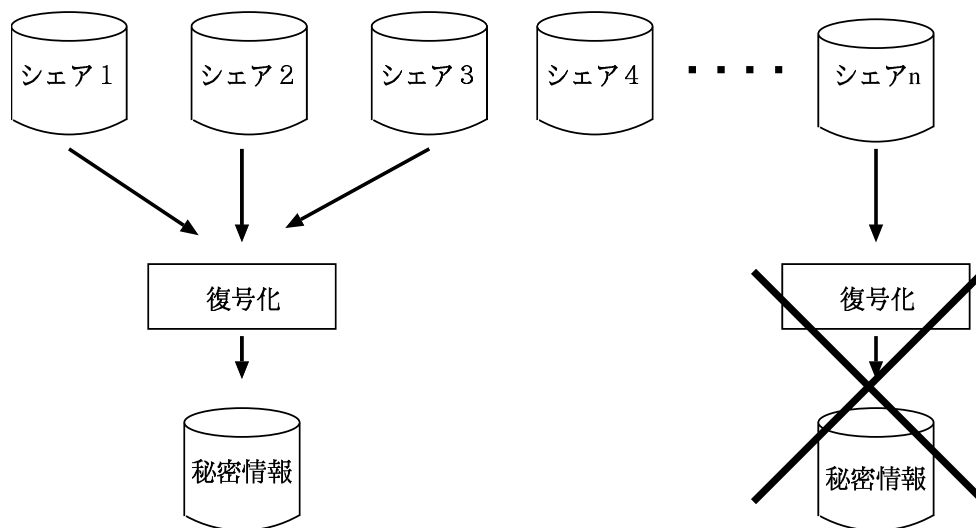

図 2.2　復号化

2.1.1　(k,n) 閾値 SSS

SSS の理論の 1 つである (k,n) 閾値 SSS について説明する。(k,n) 閾値 SSS では、秘密情報を n 個のシェアと呼ばれる分散情報に分散符号化する (図 2.1)。任意の分散情報を k 個集めると元の秘密情報を復元することができる。ただし、分散情報を $k-1$ 個以下集めても元の秘密情報を復元することはできない (図 2.2)。また、シェアの一つ一つを見ても秘密情報の部分的な情報はわからないようになっている [7, 10]。

2.1.2　(k,n) 閾値 SSS のアルゴリズム

ここでは、(k,n) 閾値 SSS のアルゴリズムについて説明する。説明には一部「秘密分散法における効率的なアルゴリズム [9]」から引用を行った。

分散符号化

秘密情報を分散符号化するために、最大次数 $k-1$ のランダムな多項式 $f(x)$ を構築する。その際、多項式の定数項を秘密情報 S とする。分散情報は、この多項式の一点 $(x_i, f(x_i)) = (x_i, w_i)$ を得る。この w_i がシェアである。処理は素体 $\boldsymbol{GF}(p)$ 上で行われる

(p は素数、$p \geq n+1$)。また、x_i は ID といい、原始根を用いる。原始根とは $p-1$ 乗したときの値だけが 1 となる数である。分散情報としてシェアと ID を持つ。

分散符号化の手順

- 初期設定
  1. $\mathbf{GF}(p)$ の元から、0 でない別々のものを n 個選びだし、$x_i(i=1,2,\ldots,n)$ とする ($p \geq n+q$)
  2. $i=1,2,\ldots,n$ に対し、値 x_i を H_i を与える
- 分散符号化
  1. 秘密情報 $S \in \mathbf{GF}(p)$ を分散保管したいとすると、$\mathbf{GF}(p)$ から $k-1$ 個の元 $r_j(j=1,2,\ldots,k-1)$ をランダムに選ぶ
  2. $w_i = f(x_i)$ を以下の式で計算する。$(i=1,2,\ldots,n)$

$$f(x_i) = S + \sum_{j=1}^{k-1} r_j x_i^j \pmod p \tag{2.1}$$

  3. w_i と ID を保管する

分散復号化

分散情報 k 個から秘密情報 S を求めるとする。それぞれ分散情報として x_i と w_i が存在し、k 個の x_i と w_i 及び多項式 $f(x)$ から k 個の未知数 $S, r_1, r_2, \ldots, r_{k-1}$ の線形方程式が k 個得られる。この線形方程式は

$$\begin{gathered}
S + r_1 x_{i_1} + r_2 x_{i_1} + \ldots + r_{k-1} {x_{i_1}}^{k-1} = w_{i_1} \\
S + r_1 x_{i_2} + r_2 x_{i_2} + \ldots + r_{k-1} {x_{i_2}}^{k-1} = w_{i_2} \\
\vdots \\
S + r_1 x_{i_k} + r_2 x_{i_k} + \ldots + r_{k-1} {x_{i_k}}^{k-1} = w_{i_k}
\end{gathered}$$

となり、行列を用いて

$$\begin{bmatrix} 1 & x_{i_1} & {x_{i_1}}^2 & \ldots & {x_{i_1}}^{k-1} \\ 1 & x_{i_2} & {x_{i_2}}^2 & \ldots & {x_{i_2}}^{k-1} \\ 1 & x_{i_3} & {x_{i_3}}^2 & \ldots & {x_{i_3}}^{k-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{i_k} & {x_{i_k}}^2 & \ldots & {x_{i_k}}^{k-1} \end{bmatrix} \begin{bmatrix} S \\ r_1 \\ r_2 \\ \vdots \\ r_{k-1} \end{bmatrix} = \begin{bmatrix} w_{i_1} \\ w_{i_2} \\ w_{i_3} \\ \vdots \\ r_{i-k} \end{bmatrix}$$

と表すことができる。これらの線形方程式は互いに独立であるため一意に解が得られ、秘密情報 S を得る。

復号化の手順

- 初期設定
  1. k 個の x_{i_j} と w_{i_j} $(1 \leq j \leq k)$ を集める
- 復号化
  1. 以下の式に x_{ij} と w_{ij} を代入して線形方程式を求める
     $$f(x) = S + r_1 x + r_2 x^2 + \ldots + r_{k-1} x^{k-1} \pmod 7$$
  2. 1 を k 回繰り返す
  3. k 個の線形方程式から k 個の未知数 $S, r_1, r_2, \ldots, r_{k-1}$ を求める。
  4. 秘密情報 S が得られる

式 2.1 において $y = f(x)$ とすると秘密情報 S は y 軸切片を示している。式 2.1 は $k-1$ 次の多項式なので、分散情報 $w_i(i = 1, 2, \ldots, n)$ が k 個集まると k 個の座標点から多項式 $y = f(x)$ が一意に求まる。しかし, 分散情報を $k-1$ 個集めただけでは $k-1$ 個の座標点しか分からず多項式が一意に定まらないので、全ての y 切片を通る可能性がある。そのため秘密情報 S が分からないようになっている。

分散符号化と復号化の簡単な例

分散符号化

p =7, k =3, n =5 とする。これは **GF**(7) 上で $(3,5)$ 閾値法を実行することを意味している。公開されている $x_1, x_2, \ldots, x_5$ の値を $x_1 = 3, x_2 = 2, x_3 = 6, x_4 = 4, x_5 = 5$ とする。係数行列 G は、

$$G = \begin{bmatrix} 1 & 3 & 3^2 \\ 1 & 2 & 2^2 \\ 1 & 6 & 6^2 \\ 1 & 4 & 4^2 \\ 1 & 5 & 5^2 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 2 \\ 1 & 2 & 4 \\ 1 & 6 & 1 \\ 1 & 4 & 2 \\ 1 & 5 & 4 \end{bmatrix} \pmod 7$$

となる。秘密情報 $S = 3$ とし、$k - 1$ 個すなわち 2 個の乱数を $r_1 = 1, r_2 = 1$ とする。式 2.1 を用いてシェアを求める。

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 2 \\ 1 & 2 & 4 \\ 1 & 6 & 1 \\ 1 & 4 & 2 \\ 1 & 5 & 4 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 2 \\ 5 \end{bmatrix} \pmod 7$$

保管する (ID, シェア) は、(3,1),(2,2),(6,3),(4,2),(5,5) となる。

復号化

公開されている $x_1, x_2, \ldots, x_5$ は $x_1 = 3, x_2 = 2, x_3 = 6, x_4 = 4, x_5 = 5$ である。3 人の参加者 H_1, H_3, H_4 がそれぞれのシェア $w_1 = 1, w_3 = 3, w_4 = 2$ を持ち寄る。式より多項式 $f(x)$ は

$$f(x) = S + r_1 x + r_2 x^2$$

となる。集まった 3 組の ID とシェア $(x_1, w_1) = (3, 1), (x_3, w_3) = (6, 3), (x_4, w_4) = (4, 2)$ より $f(x_1), f(x_3), f(x_4)$ を求める。参加者 H_1 によって求められる多項式は $x_1 = 3$ を用いて

$$f(3) = S + 3r_1 + 3^2 r_1 = 1$$

となる。同様に H_3, H_4 のシェアから以下の 2 つの多項式が求められる。

$$f(6) = S + 6r_1 + 6^2 r_1 = 3$$

$$f(4) = S + 4r_1 + 4^2 r_1 = 2$$

こうして求められた 3 つの多項式を行列式で表すと

$$\begin{bmatrix} 1 & 3 & 3^2 \\ 1 & 6 & 6^2 \\ 1 & 4 & 4^2 \end{bmatrix} \begin{bmatrix} S \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}$$

となる。この連立方程式を解くと

$$\begin{bmatrix} S \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix}$$

が求められる。よって秘密情報 $S = 3$ が得られた。

2.2　SSS を利用した実装

既存の SSS の実装には Shamir の (k, n) 閾値 SSS をソースファイルに対して適用する UNIX 上のコマンド ssar (Secret Sharing ARchiver) がある [8]。ssar コマンドを使用すると秘密情報から分散情報ファイル (シェアファイル) を作成したり、シェアファイルから秘密情報を復号化することができる。

コマンドを実行するとソースファイルからオプションにより設定した数のシェアファイルを出力する。分散符号化の構文とオプションは以下の通りである。

```
ssar -c -s number -t number -b number -f basename sourcefilename
```

`-c` 暗号化の指示

`-s` シェアファイルの数

`-t` 閾値の値

`-f` シェアファイルにつける basename

`-b` 暗号化を行うブロック長の指定

復号化のときは、コマンドを実行すると閾値個のシェアファイルから元のデータをオプションで指定されたファイル名で出力する。復号化の場合の構文とオプションは以下の通りである。

```
ssar -x -f decodefilename sharefilename1 sharefilename2 ...
```

`-x` 復号化の指示

`-f` 復元するファイルの名前

第 3 章

UNIX ファイルシステム

第 4 章で SSS を UNIX ファイルシステムへ応用することを提案する。その技術的準備として本章では、まず基本的な UNIX ファイルシステムについて説明を行う。次に、今回 SSS を UNIX ファイルシステムに応用するために使用するスタッカブルファイルシステムについて説明を行う。

3.1 UNIX ファイルシステムの概要

UNIX ファイルシステムは BSD や Solaris などのさまざまの UNIX で使用されているファイルシステムである。

ファイルシステムの内部では、1 つのファイルに対して 1 つの i ノード (index node) が割り当てられている。i ノードによってファイルの実データを参照できるようになっている。

初期の UNIX のカーネル内部では、すべてのファイル関連の処理を i ノードをベースに処理を行っていた。しかし、その後ネットワークファイルシステム、他の OS のファイルシステムや ISO9660 規格のファイルシステム等が登場し、それらを扱うようになってきた。しかし、それぞれのファイルシステムでインターフェースが異なればユーザにとって使いにくい。そこで、ファイルシステムと i ノードを抽象化して扱うことにより、ユーザプロセスに統一的なインターフェースを提供するようになった。実際の処理はおのおののファイルシステムを実装するモジュールに任せる。ファイルシステムを抽象化したものを仮想ファイルシステム (Virtual File System)、i ノードを抽象化したものを v ノード (virtual node) と呼ぶ。VFS の構成を図 3.1 に示す。

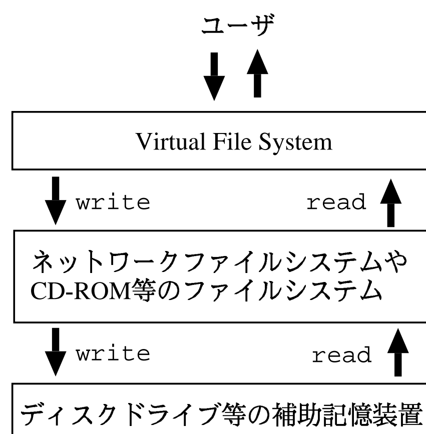

図 3.1　Virtual File System

仮想ファイルシステムには大きな制限がある。インターフェースが継承をサポートしていないため、既存のファイルシステムのいくつかの機能を継承した新しいファイルシステムを構築することができない。また、インターフェースが拡張することができないため、ファイルシステムに新しい機能を追加したり、既存の操作の動作を変更することができない。

そこで、スタッカブル層を利用したスタッカブルファイルシステムを利用することにより、ファイルシステムに新しい機能を追加したり、既存の操作の動作を変更することができるようになる。

3.2　スタッカブルファイルシステム

スタッカブルファイルシステム [6, 2, 3] は、UCLA の Ficus プロジェクトの成果として作られたファイルシステムで、4.4BSD に実装されている。スタッカブルの名のととおり、ファイルシステムを積み重ねることで、元となるファイルシステムに何らかの付加的な処理を行うことができるようになる。スタッカブルファイルシステムの構成を図 3.2 に示す。スタッカブルファイルシステムには、テンプレートである Null ファイルシステムがある。

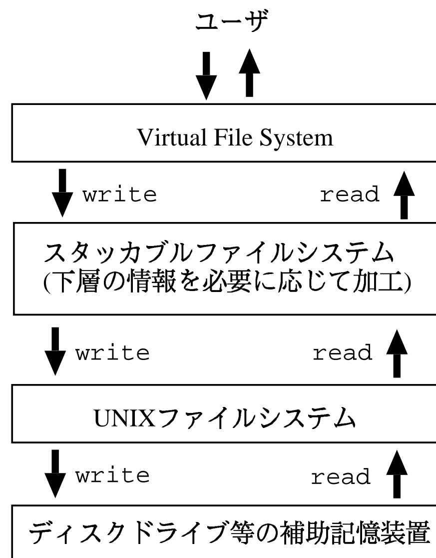

図 3.2　スタッカブルファイルシステム

3.2.1　Null ファイルシステム

Null ファイルシステム [4] はある既存のファイルシステム上の空間 (ディレクトリツリー) を既存のファイルシステム上の任意の位置にマウントできるようになる。これにより、ディレクトリーツリーの各ファイルは、第 2 のパス名を持つことができるようになる。

新しい Null ファイルシステムは `mount_null` コマンドで作る。`mount_null` は 2 つの引数をとる。1 つは下位のレイヤーの仮想ファイルシステムのパス名 (target-pn) で、もう 1 つは Null ファイルシステムが現れる名前空間内のパス名 (mount-point-pn) である。Null ファイルシステムが適切な場所に置かれたあと、目的のディレクトリ階層 (target-pn) の中身がマウント先 (mount-point-pn) にエイリアスされる。`mount_null` の書式は以下の通りである。

```
mount_null [-o options] target mount_point
```

オプションは-o の後にコンマで区切って指定する。オプションには以下のようなものがある。

- `async`

指定したファイルシステムのすべての I/O を非同期に行う。

- force

  ファイルシステムのマウント状態を読み書き可能から読み込み専用へ変更するときに、すでにオープンされているファイルへの書き込み許可を強制的に取り消す。また、正常かどうか不明なファイルシステムも強制的に読み書き可能にマウントする。

- sync

  ファイルシステムのすべての I/O を同期的に行う。

- rdonly

  マウントするファイルシステムを読み書き専用にする。スーパユーザでも書き込みできない。

- noexec

  マウントしているファイルシステム上のバイナリの実行を許可しない。

実際に mount_null コマンドの使用例について説明する。図 3.3 に示すファイルシステムの構造があるとする。そこで、

```
mount_null /usr/src /home/guest/src
```

と実行すると、図 3.4 に示す木構造が出来上がる。これにより、実線で囲んだ/usr/src 配下が破線で囲んだ/home/guest 配下にエイリアスされる。

3.2.2　Null ファイルシステムの処理

この節では、Null ファイルシステムの処理について説明する。Null ファイルシステムはすべてのオペレーションを下位のレイヤーに処理させるためにバイパスするだけである。構成を図 3.5 に示す。ほとんどの v ノードに対する操作はバイパスする。その動作のほとんどはバイパスルーチンに集中している。バイパスルーチンは下位レイヤーの任意の v ノードに対する操作を受け付ける。まず、v ノード に対する操作の引数を検査し、null-node を下位層において等価となるものに置き換える。次に、下位レイヤーの操作を起動する。最後

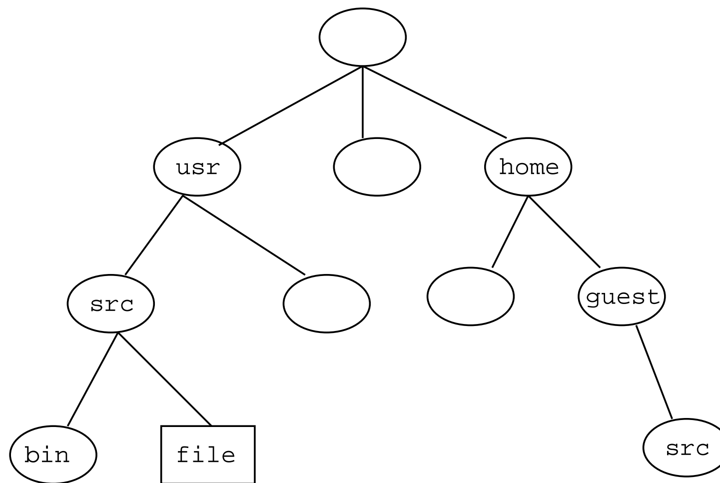

図 3.3　mount_null 前の UNIX ファイルシステム階層構造

に、引数中の `null-node` を置き換える。もしその操作によって v ノードが返ってきたら、その返ってきた v ノードの上に `null-node` を積む。

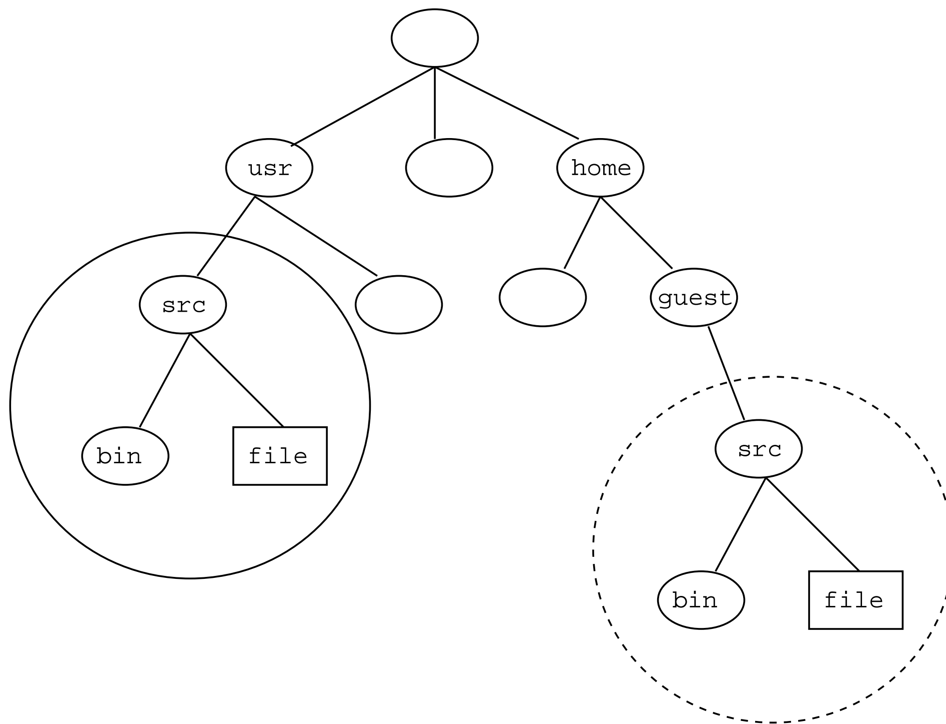

図 3.4　mount_null 後の UNIX ファイルシステム階層構造

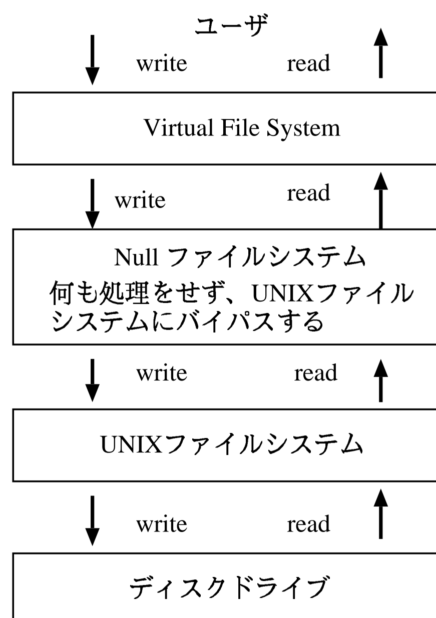

図 3.5　Null ファイルシステム

第 4 章

SSS のファイルシステムへの応用

本章では、まず SSS を利用したファイルシステムの提案を行う。今回、ファイルシステムへ実装するのに必要な分散符号化/復号化モジュールを作成したので、分散符号化/復号化モジュールについての説明も行う。

4.1 Secret Sharing File System

現在、Shamir の (k, n) 閾値 SSS をソースファイルに対して適用する UNIX 上のコマンド ssar (Secret Sharing ARchiver) がある。ssar コマンドを使用すると秘密情報から分散情報ファイル (シェアファイル) を作成したり、シェアファイルから復号化したりできる。ssar コマンドを使用することによりユーザは SSS を使って秘密情報に秘匿性や対災害性を持たせて保存することができる。

しかし、ssar コマンドには問題がある。明示的にコマンドを使用しなければならないので、ユーザに SSS 機構の存在を意識させてしまうことがある。また、シェアファイルの一つ一つがユーザに見えてしまうのも SSS 機構の存在を意識させてしまう原因になっている。

UNIX ファイルシステムでファイルを保存するときはどのようにしてファイルが保存されているかを考えながらデータを保存するユーザはいないだろう。誰もが何も考えずにファイルを保存しているはずである。何も考えずに SSS を使用できるようになると、ユーザにとって便利 SSS を UNIX ファイルシステムに応用することを提案する。SSS をファイルシステムへ応用するとユーザは通常の通りファイルを操作するだけでよい。このとき、バックグラウンドでは SSS を使いファイルを保存しているので、安全にファイルを保存することができ

る。SSS を UNIX ファイルシステムへ応用したものを SSFS (Secret Sharing File System) と呼ぶことにする。

4.2　SSFS の設計

本提案では、UNIX のスタッカブルファイルシステムを用いる。これにより、UNIX ファイルシステムに自作のモジュールを利用することが可能となり、付加的な分散符号化/復号化の処理を行えるようになる。

4.2.1　分散符号化/復号化の処理

今回は、スタッカブルファイルシステムのテンプレートである Null ファイルシステムを利用する。Null ファイルシステムに分散符号化/復号化モジュールを組み込むことによって SSS を利用した UNIX ファイルシステムを実現する。

SSS の処理は分散符号化と復号化の 2 つに分かれている。同様に、SSFS の処理もこの 2 つにわかれる。分散符号化/復号化の処理は、ユーザプロセスからの write/read システムコールをそれぞれ Null ファイルシステムに埋め込む SSS の分散符号化/復号化モジュールに経由させる。そうすることによって分散符号化/復号化の処理を行い (k, n) 閾値 SSS を UNIX ファイルシステムに応用する。

SSFS として機能するには write/read の処理だけでは成立しない。その他にも SSFS がファイルシステムとして成り立つには様々な機能を必要とする。Null ファイルシステムはスタッカブルファイルシステムを作るのに必要な要素はすべて兼ね備えている。そのため、その他のファイルシステムとして成り立つために必要な機能は Null ファイルシステムのものを利用する。SSFS の構成を図 4.1 に示す。

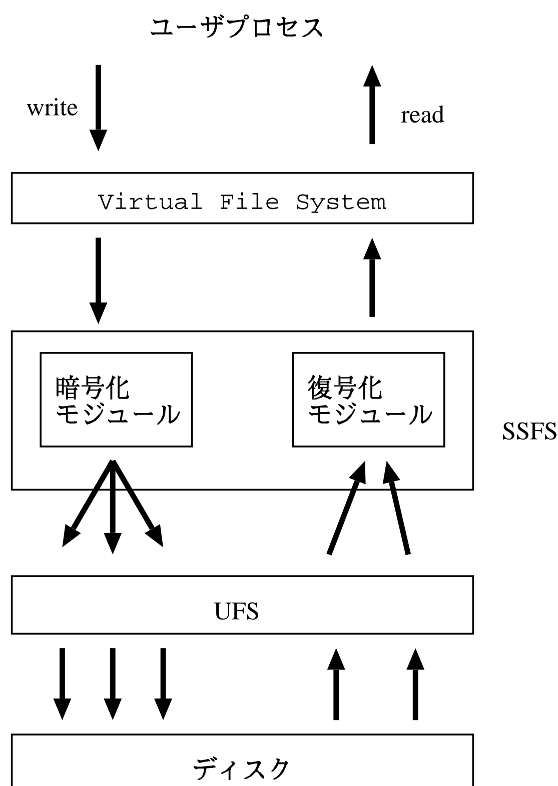

図 4.1　Secret Sharing File System

4.2.2　シェアの保存の方法

秘密情報から分散符号化してできたシェアをどのようにして保存するか、いくつかの方法を示す。シェアの保存方法には、

1. 同じパーティションに保存する方法
2. 別々のパーティションに保存する方法
3. 別々のディスクに保存する方法

がある。この 3 つの方法で Null ファイルシステムのものが、どれぐらいそのまま利用できるかを調べた。その結果を、表 4.1 に示す。

表 4.1　Null ファイルシステムのオペレーション評価

	同一パーティション	別パーティション	別ディスク
nullfs_fhtovp	○	△	△
nullfs_checkexp	○	△	△
nullfs_mount	○	×	×
nullfs_quotactl	○	△	△
nullfs_root	○	△	△
nullfs_start	○	△	△
nullfs_statfs	○	△	△
nullfs_sync	○	△	△
nullfs_unmount	○	×	×
nullfs_vget	○	△	△
nullfs_vptofh	○	○	○
nullfs_extattrctl	○	△	△
null_bypass	×	×	×
null_access	○	○	○
null_getattr	○	○	○
null_inactive	○	○	○
null_lock	○	○	○
null_lookup	○	○	○
null_print	○	○	○
null_reclaim	○	○	○
null_setattr	○	○	○
null_unlock	○	○	○
nullfs_init	○	○	○

	同一パーティション	別パーティション	別ディスク
nullfs_uninit	○	○	○
null_node_find	○	×	×
null_node_alloc	○	×	×
null_node_create	○	×	×
null_checvp	○	○	○

○:そのままでも使える

△:多少の改造が必要

×:大幅に改造が必要

同一のパーティションにシェアを保存する方法では、表 4.1 を見てみると手を加えなくてはならないのは、null_bypass だけである。そのため、Null ファイルシステムのほとんどのオペレーションをそのまま利用できると考える。null_bypass には分散符号化/復号化モジュールを組み込む作業が必要である。改造を加えなければならないのが 1 つのオペレーションなので、比較的容易に SSS を UNIX ファイルシステムへ応用できると考える。しかし、この場合はシェアを同じパーティションに保存するため、そのパーティションがクラッシュしたりすると秘密情報を復号化することができなくなる。そのため、同一のパーティションに保存する方法は SSS を使用する意味がないと考える。

別々のパーティションに保存する方法では、パーティションの一つがクラッシュしても、残りのシェアから秘密情報を復号化することが可能である。しかし、この場合は、分散符号化/復号化モジュールを組み込むだけではく、Null ファイルシステムの mount/umount の操作を複数のパーティションにマウント/ アンマウントできるように改造しなくてはならない。また、それにともない、その他の多くのオペレーションも複数のパーティションに対応するように少しずつ改造しなくてはならなくなる。

同じように、別々のディスクに保存する方法も分散符号化/復号化モジュールを組み込むだけでなく、複数のディスクをマウント/アンマウントできるように改造を加えなければならない。また、その他の多くのオペレーションが複数のディレクトリに対応するように少し改造をしなくてはならない。

3 つの方法を比較してみると、一つのパーティションに保存する方法は、SSS を利用する意味があまりないので問題外で、同様に改造が必要である観点より、別々のディスクに保存する方のが良いと方法と考える。なぜなら、別々のディスクに保存する方が対災害性が強くなるからである。

その他にもシェアをディスクに保存するときに、考えなくてはならないことがある。それは、シェアの一つがディスクの容量オーバーで保存できないときがでてくることである。今回作成した、分散符号化/復号化モジュールは $(2,3)$ 閾値 SSS なのでシェアの一つがなくても残りの二つのシェアがあれば秘密情報を復号化することはできる。しかし、保存時にシェアの 1 つがなくなってしまうと、あまり有益でないので、エラーを返し保存できないようにした方がよいと考える。

また、SSFS をマウントする時にも同様のことが言える。だから、ディスクの容量がいっぱいのディスクをマウントしようとするとエラーを返しマウントできないようにする。そうすることによってシェアの保存時に最初からシェアの一つが欠けることを防ぐ。

4.3　分散符号化/復号化モジュール

今回、SSS を実装するのに必要な分散符号化/復号化モジュール (付録**??**参照) を作成した。

4.3.1　仕様

今回 C 言語を用いて作成した、分散符号化/復号化モジュールは、秘密情報を char 型とし、分散符号化計算の単位のブロック長は 9 ビットにした。また、返値は int 型とした。SSS

は、1 バイトの秘密情報からは、9 ビットのシェアができる。int 型は 4 バイトなので、シェアは 3 個までいれることができる。そして、残りのビットをチェック用に使用することにしたので、分散するシェアを 3 個にした。シェアが 3 個だと、$(1,3)$,$(2,3)$,$(3,3)$ 閾値が可能になる。しかし、$(1,3)$ は、シェアが一つで秘密情報が復号化できてしまう。また、$(3,3)$ は、シェアが 3 個必要なので、シェアを 1 つでも喪失してしまうと復号化することができなくなる。よって、今回は、$(2,3)$ 閾値 SSS を採用し、Shamir のアルゴリズムを使用した。素数はブロック長より大きくなければならないので 257 にした。また、素数 257 の最小原始根は 3 である。乱数は 1 に固定した。

4.3.2　分散符号化モジュール

今回、作成した分散符号化モジュールの引数は、char 型で 1 バイト。返値は int 型で 4 バイトにした。分散符号化モジュールは、引数で与えられた秘密情報からシェアを 3 個作成する。作成されたシェアは返値の下位 27 ビットに入れる。シェアが有効かどうかを表すために、返値の上から 3~5 ビットにフラグを立てた。上から 3 ビット目にビットが立っているとシェア 1、4 ビット目だとシェア 2、5 ビット目だとシェア 3 が有効である。

返値のフォーマットを図 4.2 に示す。

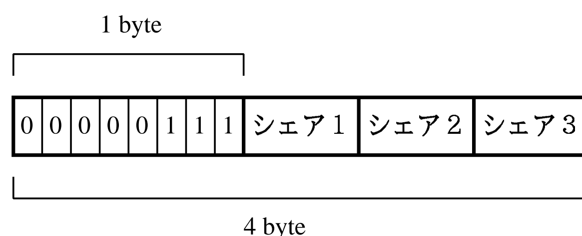

図 4.2　分散符号化モジュールの返値

4.3.3　復号化モジュール

今回、作成した復号化モジュールの引数と返値はともに、int 型で 4 バイトとした。復号化モジュールは引数で与えられた 2 個のシェアから秘密情報を復号化する。どのシェアで復

号化するかは、引数の上位 1 バイトの下位 3 ビットにどのシェアが有効か表すフラグが立っているので、その有効な 2 つのシェアより復号化する。もし、有効なビットが 3 ビット共立っている場合は、シェア 1 とシェア 2 から復号化するようにする。復号化に成功すると、返値の下位 1 バイトに秘密情報を入れる。また、復号化に成功したことを表すフラグを上位 1 ビットに立てた。復号化に成功したときの返値のフォーマットを図 4.3 に示す。また、有効なシェアが 0 個や 1 個の場合は復号化することができない。このときは、なにもフラグを立てないようにした。復号化に失敗したときの返値のフォーマットを図 4.4 に示す。成功したかどうかを表すフラグは復号化の失敗をユーザに知らせるために使用する。

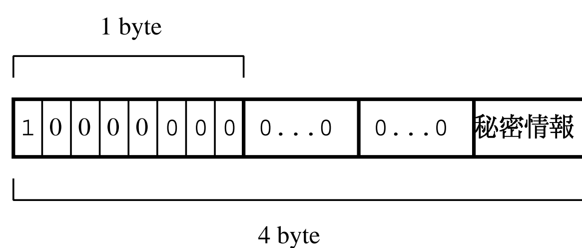

図 4.3　復号化モジュールの返値 (成功)

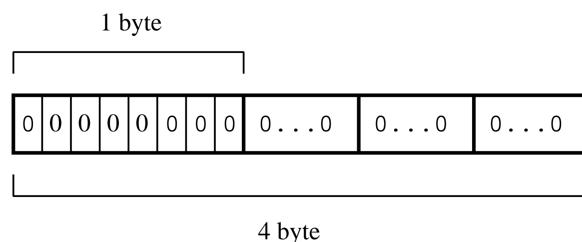

図 4.4　復号化モジュールの返値 (失敗)

第 5 章

暗号化/復号化モジュールの動作テスト

本章では、今回作成した、暗号化/復号化モジュールに動作テストについて説明する。そのために、ファイルから暗号化/復号化モジュールに入出力を行うようなプログラムを作成して実験した。

5.1 復元性

今回作成した暗号化/復号化モジュールが本当に暗号化/復号化できるかを実際にプログラムを作成した実験した。

5.1.1 任意のファイルでも暗号化/復号化することができるか

どんなファイルでも暗号化/復号化することができるかどうかを検証した。今回は、秘密ファイル `test.txt` で実験を行った。`test.txt` にはバイナリで `0x00`〜`0xFF` までが書かれている。復号化されたファイルを `testout` とした。秘密ファイルと復号化したファイルを `cmp` コマンドを用いて同じかどうかを調べた。`test.txt` を暗号化/復号化した結果は図 5.1 のようになった。

図 5.1 からわかるように、復号化に失敗した。原因を調べるために、ファイルから 1 バイトづつ読み込むときに、読み込んだ値を表示させるようにした。また、暗号化モジュールを経由して作成された、シェアもそれぞれ表示させた。結果は図 5.2 のようになった。

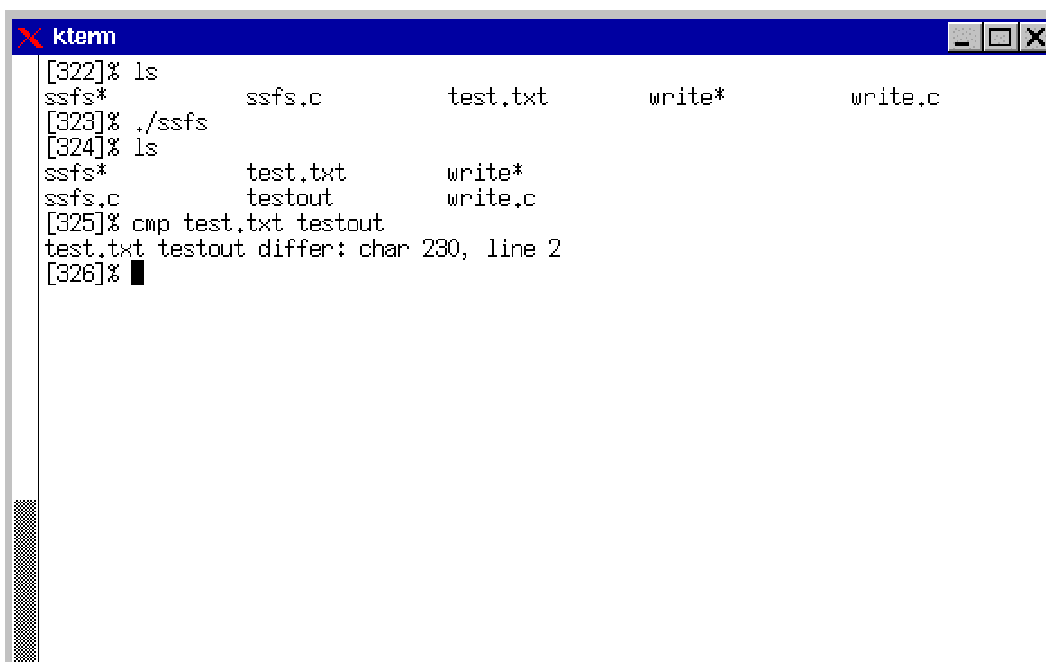

図 5.1 実行結果 1

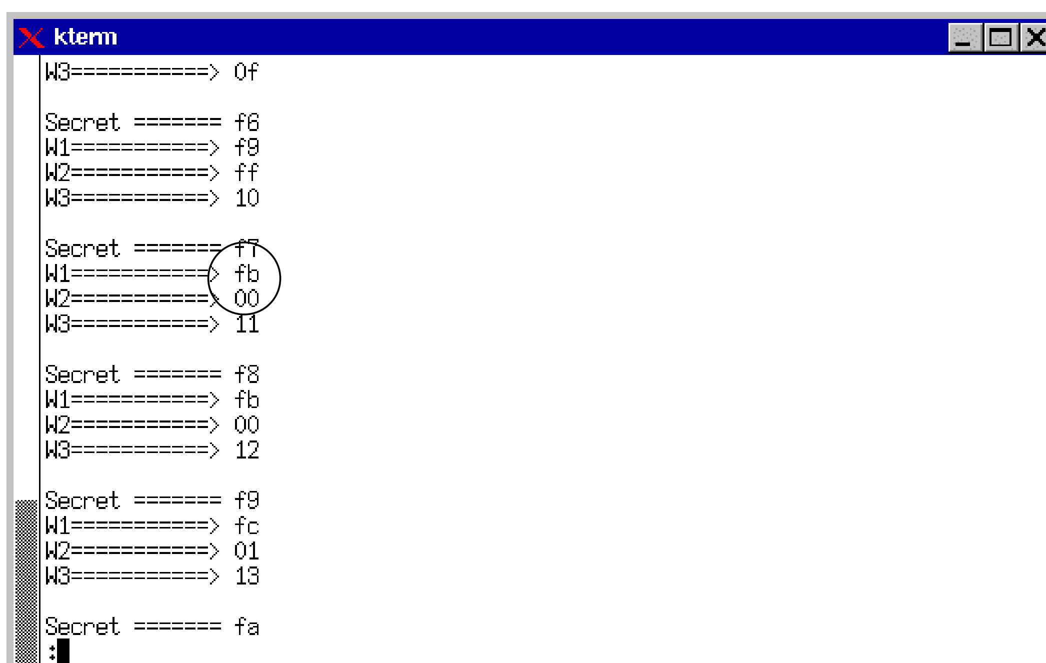

図 5.2 実行結果 2

ファイルから読み込んだ値が、f7 のときは、シェアはそれぞれ、16 進数表記で fa、100、11 になる。しかし、シェアのサイズは 1 バイトなのに、シェア 2 は 1 バイトを越えているので、シェア 1 が fb、シェア 2 が 00 になってしまった。この他にも、読み込んだ値が fd のときは、シェア 1 が 1 バイトを越えてしまい、00 になってしまった。また、e5 のときは、シェア 3 が 1 バイトを越えてしまい、00 になってしまった。このシェアのサイズが 1 バイトを越えたことが失敗した原因になっていた。

以後の実験では、秘密ファイル test.txt から f7、fd、e5 を削除した秘密ファイル test2.txt を用いて実験を行った。

5.1.2　任意のシェアファイル 2 個から復号化することができるか

本当に任意のシェアファイル 2 個から復号化することができるかどうかを検証した。今回作成した復号化モジュールは $(2, 3)$ 閾値 SSS なので復号化するパターンは

1. シェア 1 とシェア 2
2. シェア 1 とシェア 3
3. シェア 2 とシェア 3

の 3 パターンになる。

本当に復号化されているかを調べるために、復号化されたファイルと秘密ファイルを cmp コマンドを用いて調べた。

test2.txt をパターン 1 で復号化したファイルを test2out1、パターン 2 で復号化したものを test2out2、パターン 3 で復号化したものを test2out3 というファイル名にした。その実験の結果は、図 5.3 になった。この実験により、正確に秘密ファイルを復号化することができていることがわかる。

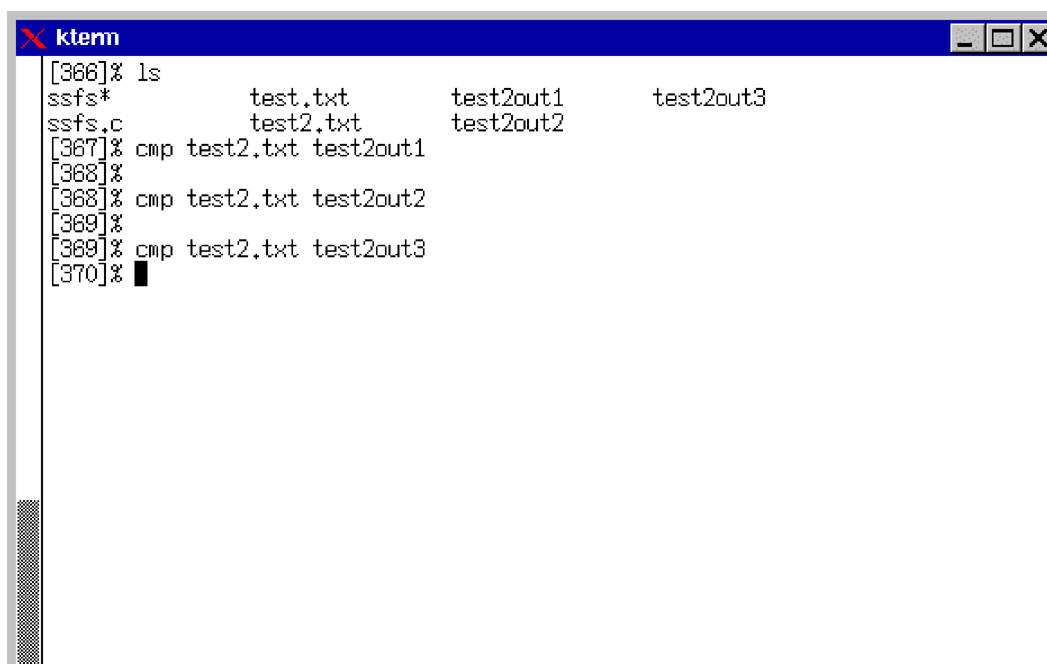

図 5.3　実行結果 3

5.2　エラー処理

今回作成した暗号化/復号化モジュールにはエラー処理を行えているかを確認するために、プログラムを作成して実験した。

5.2.1　シェアファイルが 0 個、1 個のときはエラーを返すか

シェアファイルが 0、1 個の場合に本当にエラーを返すかどうかを調べた。復号化モジュールの引数には、3 つのシェアのうち、どの 2 つのシェアが有効かを表すフラグをたてている。

今回の実験では、

1. 全部のシェアにフラグをたてる
2. シェア 1 とシェア 2 にフラグをたてる
3. シェア 1 とシェア 3 にフラグをたてる
4. シェア 2 とシェア 3 にフラグをたてる
5. シェア 1 だけにフラグをたてる

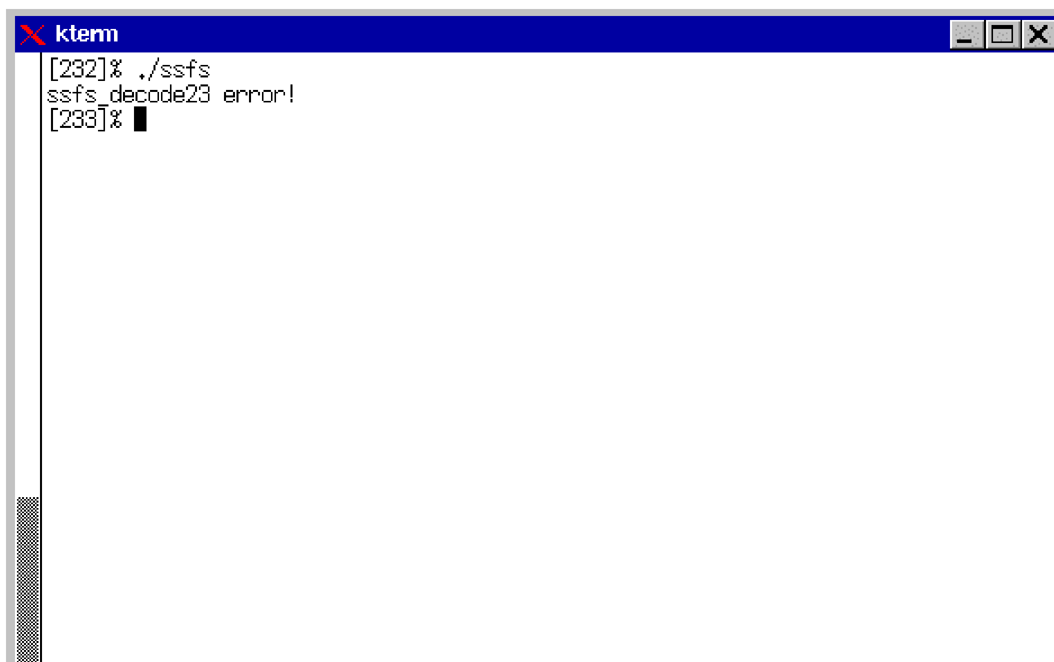

図 5.4　ssfs_decode23_error

6. シェア 2 だけにフラグをたてる
7. シェア 3 だけにフラグをたてる
8. シェア全部にフラグをたてない

の 8 つのパターンで実験を行った。

作成した復号化モジュールは復号化に成功すると、返値の上位 1 ビットにフラグがたつようにした。もし、フラグがたっていないと、エラーと判断して

```
ssfs_decode23 error!
```

と表示するように実験用のプログラムを作成した。

有効なシェアが 2 個以上ある 1〜4 の場合は、問題なく復号化された。また、シェアが 0、1 個の 5〜8 の場合は図 5.4 のようにエラーが表示された。

第 6 章

まとめ

6.1　まとめ

本研究では、ユーザに秘密分散法を意識させないようにするということを目的として、SSS を UNIX ファイルシステムへ応用することの提案と暗号化/復号化モジュールの作成とを行った。SSS をファイルシステムへ応用することによってユーザは SSS 機構を意識することなく、かつ安全に情報を保存することができるようになる。

6.2　今後の課題

本研究により Null ファイルシステムのいくつかのオペレーションを改造することによって SSS を UNIX ファイルシステムへ応用することが可能だと考える。今回作成した暗号化/復号化モジュールはシェアのサイズが 1 バイトを越えるものが入っていると暗号化/復号化に失敗したので、今後は、暗号化モジュールの返値のシェアをいれるサイズを 1 ビット大きくするという対処法をとることによって、完全に暗号化/復号化できるようになる。

本研究では、SSS を UNIX ファイルシステムへ応用することを提案と暗号化/復号化モジュールを作成しただけである。実際に SSS を UNIX ファイルシステムに実装するためには、まず、Null ファイルシステムの write/read 機構を解析して暗号化/復号化モジュールを組み込む必要がある。また、Null ファイルシステムの mount を複数のディスクにマウントできるように改良を加えなくてはならない。それにともない、その他のオペレーションに関しても、若干の改良を加えることによって実際にファイルシステムとして SSFS を使用でき

るようになると考える。

謝辞

本研究を行うにあたり、御指導、御助言を頂いた舟橋 釈仁氏に心から感謝致します。また、福本昌弘助教授をはじめ、福本研究室のみなさま代数学の輪講でお世話になりました。本研究の全過程を通じて、研究のテーマから文章の詳細部分に至るまで数多くの御助言を頂いた菊池豊 助教授に心から感謝の意を表します。加えて、本論文をまとめるにあたって御協力を頂いた菊池研究室の諸氏に感謝致します。

参考文献

[1] G.R Blakley. Safeguarding cryptographic keys. *Proc. of AFIPS 1979 Nat. Computer Conf.*, Vol. 48, pp. 313–317, 1979.

[2] John Shelby Hedemann. Stackable layers: An architecture for file system development, Aug 1991.

[3] John Shelby Hedemann and Gerald J.Popek. File system development with stackable layers, Jul 1993.

[4] Jan-Simon Pendry and M.K.McKusick. Union mounts in 4.4bsd-lite, Jan 1995. Conference Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems.

[5] A. Shamir. How to share a secret. *Communication of the ACM*, Vol. 22, No. 11, pp. 612–613, 1979.

[6] Uresh Vahalia. 最前線 UNIX のカーネル, pp. 421–426. ピアソン・エデュケーション, May 2000.

[7] 岡本龍明, 山本博資. 現代暗号. 産業図書, June 1997.

[8] 舟橋釈仁. 秘密分散法に基づくファイル保管コマンドの実装. 高知工科大学, 2001.

[9] 庄田亜輝. 秘密分散法における効率的なアルゴリズム. 高知工科大学, 2001.

[10] 尾形わかは, 黒沢馨. 秘密分散共有法とその応用. 電子情報通信学会誌, Vol. 82, No. 10, pp. 1228–1236, Dec 1999.

付録 A

暗号化/復号化モジュール

A.1　暗号化モジュール

```
1   #define RANDOM 1  /*乱数*/
2   #define PRIME 257 /*素数*/
3
4   int ssfs_encode23(char secret){
5
6     int w1,w2,w3;
7     int result = 0;
8     int G [3][3] = {
9       1,3,9,
10      1,9,81,
11      1,27,729
12    };
13    /* シェア 1 を計算する */
14    w1 = (G[0][0]*secret + G[0][1]*RANDOM) % PRIME;
15    /* シェア 2 を計算する */
16    w2 = (G[1][0]*secret + G[1][1]*RANDOM) % PRIME;
17    /* シェア 3 を計算する */
18    w3 = (G[2][0]*secret + G[2][1]*RANDOM) % PRIME;
```

```
19
20      /*1 バイト目にフラグを立てる*/
21      result = (result & 0x00ffffff) | (0x07 << 24);
22      /*2 バイト目にシェア 1 を格納*/
23      result = (result & 0xff00ffff) | (w1 << 16);
24      /*3 バイト目にシェア 2 を格納*/
25      result = (result & 0xffff00ff) | (w2 << 8);
26      /*4 バイト目にシェア 3 を格納*/
27      result = (result & 0xffffff00) |  w3;
28
29      return  result;
30
31    }
32
```

A.2　復号化モジュール

```
1   #define PRIME 257 /*素数*/
2
3   int ssfs_decode23(int share){
4
5     int check;
6     int w1,w2,w3;
7     int result=0;
8     int G1 [2][2] = {
9       387,-129,
```

```
10      -43,43
11    };
12
13    int G2 [2][2] = {
14      19320201,-2146689,
15      -715563,715563
16    };
17
18    int G3 [2][2] = {
19      387,-129,
20      -3698,3698
21    };
22
23    /*引数からフラグを取り出す*/
24    check = (share & 0xff000000) >> 24;
25    /*引数からシェア1を取り出す*/
26    w1 = (share & 0x00ff0000) >> 16;
27    /*引数からシェア2を取り出す*/
28    w2 = (share & 0x0000ff00) >> 8;
29    /*引数からシェア3を取り出す*/
30    w3 = (share & 0x000000ff);
31
32    switch(check){
33    case 0x06:
34      result = (w1 *G1[0][0] + G1[0][1] * w2) % PRIME;
35      result = (result & 0x00ffffff) | (0x07 << 24);
```

```
36        break;
37      case 0x05:
38        result = (w1 *G2[0][0] + G2[0][1] * w3) % PRIME;
39        result = (result & 0x00ffffff) | (0x07 << 24);
40        break;
41      case 0x03:
42        result = (w2 *G3[0][0] + G3[0][1] * w3) % PRIME;
43        result = (result & 0x00ffffff) | (0x07 << 24);
44        break;
45      case 0x07:
46        result = (w1 *G1[0][0] + G1[0][1] * w2) % PRIME;
47        result = (result & 0x00ffffff) | (0x07 << 24);
48        break;
49      default:
50        break;
51      }
52
53      return  result;
54
55    }
```