SSSによる分散データ作成コマンドの実装

舟橋 釈仁* 福本 昌弘* 菊池豊*

情報を分散保存する目的の暗号理論にSSS (Secret Sharing Scheme) がある. 今回, 我々はSSSに基づく暗号化を行うコマンドをUNIX上に実装した. 性能を測定したところ, 用途によっては十分実用になることが判明した. 本稿では, コマンドの実装と測定について述べ, より実用的な実装を行うための議論を行う.

Implementation of a command for data distribution using SSS

FUNAHASHI Sekiji†, FUKUMOTO Masahiro† and KIKUCHI Yutaka†

Secret Sharing Scheme, or SSS, is a scheme of cryptology, that distributes portions of information safely. We implemented an encrypt/decrypt command on UNIX based on SSS. It turns out that the command is useful in some applications because of a performance result of the command. This paper describes the implementation of the command, the measurement of it, and discussions about more practical implementation.

1 はじめに

さまざまな分野にIT技術が用いられるようになり、情報の保管に対する安全性はますます重要になって来ている。分野によっては、たとえコストが若干余計にかかろうとも十分な秘匿性や対災害性を持たせたいという要望も出て来た。例えば、住民基本台帳等を管理する地方自治体や保健・医療・福祉等の重要な個人情報を扱う組織である。

情報を複数に分散させ、なおかつ秘匿性をも与える暗号理論にSSS (Secret Sharing Scheme) がある。SSSの時間計算量や空間計算量は大きく、これまでは実用性が低いと考えられていた。近年、ハードディスクの低価格化やネットワーク伝送の広帯域化によって、SSS技術が現実味を帯びて来ている。今回、我々はSSS理論を基にしたUNIXコマンドを実装し性能計測を行った。

本稿では, 最初にSSSとRAIDを比較し, SSSの特徴を述べる. 次に使用したSSSのアルゴリズムについて説明する. そして, SSSの実装における方針を示し, 実装するコマンドの仕様と, どのような実装を行ったか詳しく述べる. また, 実装したコマンドについて性能測定を行った結果を示し, その結果についての考察を述べる. 最後に今後の予定について述べる.

*高知工科大学情報システム工学科
†Department of Information Systems Engineering, Kochi University of Technology

2 分散型データ保管法

データを冗長にすることにより対障害性を持つ分散型データ保管法にはSSSの(k, n)閾値法やRAID[3]がある.

これらの技術はデータをいくつかのハードディスに分散して保管する. 保管されたデータは冗長になりハードディスクが壊れてもデータを失うことが少なくなり, 単一のハードディスクに保管するより安全性が高い.

このような特徴をもつSSSとRAIDにおいてどちらがデータ保管の技術に適しているか調べるため両者を比較した. また, 暗号化したデータをコピーして冗長性を持たせた分散データも比較した.

2.1 (k, n)閾値SSS

(k, n)閾値法はShamir[2]によって提案されたデータを分散する方法で, 秘密情報からn個のシェアをつくる. これをシェアと呼ぶ. そして, シェアをつくる時に$0 < k \leq n$の範囲で閾値kを設定する. シェアをk個集めると元の秘密情報が復元できる. また, シェアが$n-k$個壊れても残りのシェアから元のデータを復元でき, $k-1$個のシェアを集めても元の秘密情報が全く分からないという特徴をもつ. この閾値の値は任意に決めることが可能である. また, 暗号化が行われるので, シェアの

どれからも部分的な情報は分からない.

一方, 作成されたシェアのサイズの合計は元のデータに比べて非常に大きくなる. 本稿では単にSSSと書いてあれば (k,n) 閾値法を指す.

2.2 RAIDの特徴

ここでRAIDはRAID5 (分散データガーディング) を想定する. RAIDはデータを複数のハードディスクに分散して保管する方法である. これによって, データの安全性の向上と読み書き性能の改善を行っている. RAIDではパリティによるデータ復元を行うため, 2台以上のハードディスクがクラッシュするとデータの復元ができない. しかし, 分散データの合計は元のデータに比べてパリティのサイズ大きくなるだけである.

また, 暗号化が行われないので, それぞれの分散データの中身を見ると, 元のデータの部分的な内容が分かる.

2.3 SSSとRAIDの比較

SSSとRAIDの比較を表1に示す. また, 単純に暗号化したデータを複製した場合も表に併記した. 表で x は元のデータのサイズを示している. また, α と γ は暗号化にともなうデータの増加分であり, β はRAIDのパリティによる増加分を示す.

まず, SSSはRAIDに比べ対災害性が格段に高い. RAIDは, 冗長性が低く2つ以上の分散データが損失すると元のデータに復元しない. SSSは閾値を任意に設定できるので, この数値を低くすることにより, 冗長性を高くすることができる.

次にSSSは分散データのセキュリティが高い. SSSでは分散と同時に暗号化が行われる. この暗号化により分散データが閾値個集まらなければ部分的な情報も分からないような仕組みになっている. 一方, RAIDには暗号化機能がついていない. それぞれの分散データをみると, 元のデータの部分的な情報が分かる.

これらの理由によりSSSはRAIDにはない優れた特徴をもつデータ分散手法であるといえる.

	SSS	RAID	暗号化複製
分散データサイズ	$n(x+\alpha)$	$x+\beta$	$n(x+\gamma)$
冗長分散データ数	$n-k$	1	$n-1$
分散データ安全性	高	低	中
処理速度	遅い	速い	遅い

表 1: SSSとRAIDと素朴な暗号化複製との比較

3 コマンドの実装

(k,n) 閾値法のアルゴリズムを与えられたソースファイルに対して適用する**ssar** (Secret Sharing Archiver) コマンドを作成した. 本節では, まず, Shamirのアルゴリズムを説明し, 次にコマンドを実装する際に立てた方針と仕様と設計, そして演算の実装の方法について説明を行う.

3.1 (k,n) 閾値法のアルゴリズム

今回のプログラムでは, (k,n) 閾値法において最も基本的なShamirの (k,n) 閾値法を選択した. 本節ではこのアルゴリズムを説明する.

最初に秘密情報Sより大きな素数 p とその原始根 α を求める. (k,n) 閾値法では暗号化, 符号化ともに演算は素数 p の有限体上で行われる.

次に暗号化では $GF(p)$ 上の秘密情報 S から $GF(p)$ 上のシェア w_i を n 個作成する. この w_i を作成する式は

$$w_i = f(x_i) \quad (i = 1, 2, \ldots, n)$$

と表され, 式(1)の行列計算を行う. ここで $r_j (j = 1, 2, \ldots, k-1)$ は $GF(p)$ 上の乱数とする.

復号化では閾値個シェア w_i を集め, 式(2)の行列計算を行い秘密情報 S を求める.

3.2 基本方針

今回の実装はSSSアルゴリズムを使用した分散暗号化システムにおいて初めての実装である. そのため, Shamirの (k,n) 閾値法を忠実に実装し, 基本的な特性を調べる事を目的とした. また, 素数計算と原始根計算は非常に時間がかかる事が分かっているものの今回は, シンプルに実装して処理にかかる時間の特性をみた.

$$
\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \equiv \begin{bmatrix} 1 & \alpha_1 & (\alpha_1)^2 & \cdots & (\alpha_1)^{k-1} \\ 1 & \alpha_2 & (\alpha_2)^2 & \cdots & (\alpha_2)^{k-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha_n & (\alpha_n)^2 & \cdots & (\alpha_n)^{k-1} \end{bmatrix} \begin{bmatrix} S \\ r_1 \\ \vdots \\ r_{k-1} \end{bmatrix} \pmod{p} \quad (1)
$$

$$
\begin{bmatrix} S \\ r_{j_1} \\ \vdots \\ r_{j_{k-1}} \end{bmatrix} \equiv \begin{bmatrix} 1 & \alpha^{j_1} & \alpha^{2j_1} & \cdots & \alpha^{(k-1)j_1} \\ 1 & \alpha^{j_2} & \alpha^{2j_2} & \cdots & \alpha^{(k-1)j_2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha^{j_k} & \alpha^{2j_k} & \cdots & \alpha^{(k-1)j_k} \end{bmatrix}^{-1} \begin{bmatrix} w_{j_1} \\ w_{j_2} \\ \vdots \\ w_{j_k} \end{bmatrix} \pmod{p} \quad (2)
$$

3.3 仕様

暗号化部分ではソースファイルが入力されたらそのファイルにSSSアルゴリズムを使用し暗号化を行いシェアファイルを出力する．復号化部分ではシェアファイルが入力されると復号化を行った後，ソースファイルを出力する．(図1参照)

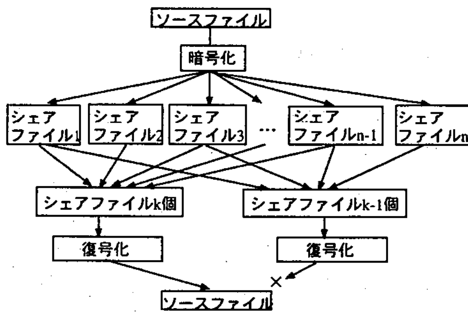

図 1: SSS による暗号化と復号化

3.3.1 コマンド形式

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．シェアファイルの名前はオプションで指定された prefix に 0〜シェアファイル数 −1 までの番号を組み合わせて作成する．

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

-c 暗号化オプション
-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

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ソース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．

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

-x 復号化オプション
-f 復元するファイルの名前

以下ではまず，ソースファイル afo からシェアファイル bfo[01234] を作成し，そのうちの bfo1, bfo3, bfo4 を使用してソースファイル cfo に戻した．

```
> ls
afo
> ssar -c -s 5 -t 3 -b 2 -f afo bfo
> ls
afo bfo0 bfo1 bfo2 bfo3 bfo4
> ssar -x -f cfo bfo1 bfo3 bfo4
> ls
afo bfo0 bfo1 bfo2 bfo3 bfo4 cfo
> cmp afo cfo
>
```

3.4 設計

シェアファイルの保存形式を設定し，コマンドの機能を暗号化と復号化に分けて設計した．

3.4.1 シェアファイルの形式

ソースファイルからつくられるシェアファイルの保存形式を表2のようにした．

HEAD 部分にシェアの情報を格納する．また, BODY 部分の一番最初に素数, 2 番目に ID, 3 番目以降にシェアデータを格納する．

	格納する情報
HEAD 情報	[HEAD] タグ シェアファイル数 閾値 basename
BODY 情報	[BODY] タグ 素数 ID シェアデータ ⋮ シェアデータ

表 2: シェアファイルの形式

3.4.2 暗号化

1 オプション解析
2 HEAD 情報をシェアファイルに書き込む
3 ブロック長より大きな素数 p を計算し, シェアファイルに書き込む.
4 素数 p の原始根を計算し, シェアファイルに書き込む.
5 乱数 r_x を関数により求めるか/dev/random から求める.
6 ソースファイルからブロック長単位で秘密情報データ S を読み込む.
7 SSS 暗号計算
式 1 の行列計算によりシェア w_i と ID を求める．ここで α を原始根とし, $\alpha_1 \cdots \alpha_n$ が ID となる.
8 シェアファイルにシェア w_i とその ID をバイナリで書き出す.
9 ソースファイルの終りまで 6〜8 を繰り返す.

3.4.3 復号化

1 オプション解析
2 HEAD 情報を読み込む.
3 シェアファイルから素数と ID を読み込む.
4 シェアファイルからシェア w_i を読み込む.
5 SSS 復号計算
式 2 の計算を行う．ここで $\alpha^{j_1}, \ldots, \alpha^{j_k}$ はシェアファイルの ID. である.
6 ソースファイルとして秘密情報データ S を書き込む.
7 シェアファイルの終りまで 4〜6 を繰り返す.

3.5 実装

本節では, 多倍長整数型と演算について説明する.

3.5.1 ブロック長

プログラムではソースファイルから秘密情報データ S を一定のブロック長で読み込む．このブロック長を 1〜512Byte (UNIX における標準的なデータ量) の可変値にした．ブロック長の指定は b オプションで行われ, 何も指定されないときはデフォルトの 1Byte になる.

3.5.2 多倍長整数型

ブロック長が 512Byte のとき 10 進整数に直すと $0 \sim 2^{4096} - 1$ の整数になる．このような大きな整数を扱うために GMP3.0 (The GNU Multiple Precision Arithmetic Library) を使用した．GMP とは多倍長の整数, 実数, 小数を扱うためのライブラリである．入出力, 乱数の取得, 各種計算関数などを持っている．GMP では多倍長整数を mpz 型としてもち, これを int 型の列として実現している.

今回のプログラムでは, 秘密情報 S や, シェアデータ w_i などを mpz 型にしている.

3.5.3 素数計算

素数を求める計算には確率的に求める方法と確定的に素数を求める方法がある．一般に確率的に素数を求める方が計算が速い．今回は処理速度を優先して確率的に求める方法を使用した．もし, 素数でなければ原始根計算の部分で判別されるのでもう一度素数計算をやり直す.

具体的には GMP のライブラリの中の素数を求める *mpz_probab_prime_p*() 関数を使用した．この関数では Miller-Rabins 法が使用されている.

3.5.4 原始根計算

このプログラムでは最初に 2 の 1 乗から 2 の $p-1$ 乗までの計算を行い, 計算結果をハッシュ構造に保存する. そして, すべての計算した値を比べ同じ値が 2 個以上あれば, 原始根でないと判断する. 次は 3 の 1 乗から 2 の $p-1$ 乗までの計算を行い, 同じ値がないか調べる. もし, 同じ値が 2 個以上あれば, 4 を試す. これを原始根がみつかるか, $p-1$ まで行う. 見付からなければ素数を求め直す.

4 性能測定

SSS アルゴリズムによる暗号化と復号化の処理速度を調べるため以下の項目について性能測定を行った. また, この性能測定を行ったマシンは CPU Pentium3 600MHz, メモリ 128MByte である. OS は FreeBSD-4.1.1-RELEASE, コンパイラは gcc version 2.95.2 を使用した. 処理速度を計る関数として *clock*() を用いた.

1 ファイルサイズによる処理速度の比較:
ブロック長を 2 バイトで一定にし, 1MByte〜100MByte のファイルを入力して, 入力ファイルサイズの違いによる暗号化と復号化にかかる処理時間を計測した. 結果を図 2 に示す.

2 ブロック長の違いによる素数と原始根を求める時間の比較:
ファイルサイズを一定にして, 素数と原始根を求める計算にかかる時間を計測した. 結果を表 3 に示す.

3 ブロック長の違いによるシェアファイルのサイズ比較:
1M と 100M の入力ファイルを与え, ブロック長の違いによるシェアファイルのサイズの違いを計測した. 結果を表 4 に示す.

ブロック長 (Byte)	原始根を求める時間 (s)
1	0
2	39

表 3: ブロック長の違いによる処理時間の変化

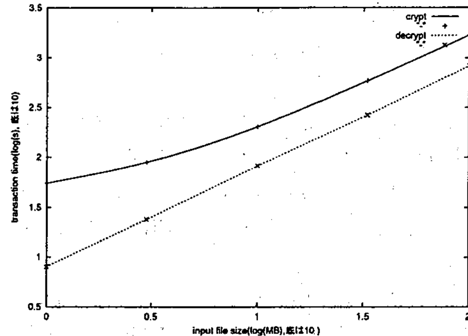

図 2: ファイルサイズの違いによる処理時間

入力ファイルサイズ (MB)	ブロック長 (Byte)	出力ファイルサイズ (MB)
1	1	5
100	1	500
1	2	3
100	2	300

表 4: ブロック長の違いによる出力ファイルサイズ

5 考察

実装を行った結果より, SSS を実装する上での課題について考察を行った.

5.1 処理速度

ブロック長が 2Byte の時, 10MB のデータを分散する時間は 203 秒で, 復号時間は 82 秒であった. 処理時間 $f(x)$ はファイルサイズの大きさ x(KB) と次のような関係式が成り立つ. ここで傾きは 16 秒/s であり, グラフの y 切片は素数と原始根を計算する時間で, 39 秒である.

$$f(x) = 16x + 39$$

この式の傾きはブロック長と関係しており, ブロック長を大きくすることにより傾きを小さくすることができる.

5.2 計算量

ブロック長が 3Byte のとき, 全体の 8 分の 1 まで計算したところでプログラムが止まった. 止まるまでの時間は 10 時間であったので予想終了時間は 80 時間程度である.

止まった原因は, 原始根を計算する過程において, プロセスが使用可能なメモリより大きなメモリを必要としたためである.

今回のアルゴリズムでは, 原始根を判別する処理の中間データをメモリ上に保持している. この中間データをファイルに保存することにより止まらなくすることができるが, 処理時間がかなり長くなると予想される.

また, SSS の全体の計算量において支配的なのは原始根を求める計算である. 一般的に原始根を求める方法は時間がかかる. 次回は計算量を減らすために違うアルゴリズムを試す予定である. しかし, 劇的な効果は期待できないと予想される.

5.3 データ量

SSS では, 一般に以下の式が成り立つ.

シェアのサイズ $\geq$ 秘密情報のサイズ

つまり最も効率のよい分散ファイルを作成したとするとそのサイズは元のファイルと同じ程度になる.

しかし, 実際のシェアファイルのサイズはソースファイルより大きくなる. なぜならシェアデータを格納する mpz 型を保存する時に, 数値自体に加えて 4Byte のオーバヘッドが生まれるためである.

例えば, 各分散ファイルのサイズはブロック長が 1 バイトの時, ソースファイルの 5 倍になり, ブロック長が 2 バイトの時ソースファイルの 3 倍になった.

ブロック長が大きくなると mpz のオーバヘッドが相対的に小さくなる. よって, シェアファイルのサイズをソースファイルに近づけることができる.

6 今後の予定

本節では今後どのような機能を実装する予定であるか説明する.

6.1 2 の拡大体における SSS

SSS においては, 素数と原始根を求める計算は必須であり, 素数と原始根を求める計算は非常に時間がかかる. この問題を解決するための一つの方法は 2 の拡大体上で SSS を使うことである. 2 の拡大体を使うと原始根を求める計算が不要になる. よって, 劇的な効率改善が期待できる. ただし, 2 の拡大体上の加法乗法の計算量が支配的になる. そのため, 実装する際には工夫する必要がある. 例えば, 最初に有限体上の全ての元に対する加法と乗法の表を構成する方法が有望だろう.

6.2 ネットワーク転送機能の付加

作成したシェアファイルを安全に保管するためには, ftp などで遠隔地にシェアファイルを送らなければならない. また, ソースファイルを復元するためには, 閾値個のシェアファイルを遠隔地から持って来る必要がある. この作業を毎回手動で行うのは手間がかかる. 次のバージョンアップでは, 遠隔地との通信機能を持たせ, シェアファイルの遠隔地への転送と遠隔地からの転送を自動で行う機能をつける予定である.

6.3 ファイルシステムへの応用

今回のプログラムはファイルに対して明示的に SSS を適用している. SSS をファイルシステムに実装することにより, すべてのデータに対してユーザが指示することなく分散暗号化を行うことが可能になる.

参考文献

[1] B. Chor, S. Goldwasser, S. Michli, and B. Awebuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. *Proc. of FOCS*, pp. 383–395, 1985.

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

[3] 宇野俊夫. デイスクアレイテクノロジ RAID. エーアイ出版, 2000.

[4] 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. 12, pp. 1228–1236, Dec 1999.

[5] 岡本龍明, 山本博資. 現代暗号, 14. 秘密分散法. 産業図書, 1997.