L-031

ユーザに負担をかけない OS レベルの動的な暗号化と復号化
Dynamic and Nonintrusive Coding and Decoding at OS Level

杉浦 秀幸† 伊藤 孝行†‡
Hideyuki Sugiura Takayuki Ito

1. はじめに

近年，暗号化技術は需要の増加とともに大きく進化してきている．なかでもファイルの暗号化は情報流出防止の観点から重要視されている．国外への情報流出は，流出した個人や企業だけでなく「知財立国」を目指す日本とって大きな損害となる．しかし，実際は暗号化技術の進化にかかわらず，P2Pソフトによる流出や，悪意あるユーザによる情報の持ち出しが後を絶たず，完全に情報流出を防ぐことはできていない．

情報流出を止めることができない原因として，従来の暗号化ソフトの方式が考えられる．従来の暗号化ソフトのほとんどは，ユーザが能動的に暗号化行う方式である．したがって，ユーザは少なくとも暗号化ソフトを使うだけの技術や知識を持ち，そして悪意は持っていないということが前提にある．しかし実際は，重要なファイルでも暗号化されていない，内部の人間による機密情報の持ち出しがおこるなど，ユーザ自身が原因の場合が多い．また暗号ソフトが難しくて使えないという声も多い．JNSAの2008年上半期での個人情報漏洩インシデント概要データ[1]によると，管理ミスや誤操作，紛失といったユーザが原因の流出は全体の70%を超える．

本論文では**ユーザに負担をかけないOSレベルの動的な暗号化，復号化**を提案する．本機構の実現は，動的な暗号および復号化機能をもつAPIを開発することで実現される．本機構では，新しいAPIを提供するのではなく，従来のアプリケーションと互換性を持たせつつ，従来のファイルAPIに暗号化，復号化の機能を持つように書き換える．本機構の導入により，ユーザに意識させることなく暗号化，復号化を実現できる．

暗号化APIを使用すると書き込みを行う全てのファイルを暗号化してしまい非効率である．本問題を解決するため，OSのシステムファイルやクッキーなど暗号化しなくても良いファイルや，暗号化APIを適用しないアプリケーションをシステム管理者に選択させるAPIの管理機構も合わせて開発する．APIの管理機構は，復号化APIに関しても適用するアプリケーションをシステム管理者に選択させることにより，システム管理者の意図しない情報流出を防止するように設計する．

本論文の構成を以下に示す．２．では暗号化，復号化の流れについて説明する．３．では暗号化および復号化アルゴリズムについて解説する．次に４．では暗号化，復号化APIを導入するためのOS側での変更点について述べる．そして５．ではAPIの管理機構について提案し，６．で実際に暗号化，復号化APIを組み込んだOSの動作例を示し，最後に本論文のまとめと今後の課題について議論する．

†名古屋工業大学 (Nagoya Institute of Technology)
‡マサチューセッツ工科大学（Massachusetts Institute of Technology）

2. 暗号化と復号化の流れ

本章では暗号化と復号化の流れについて解説する．図１では，ブロック図を用いて，どのようにデータが暗号化，復号化されるかを示している．

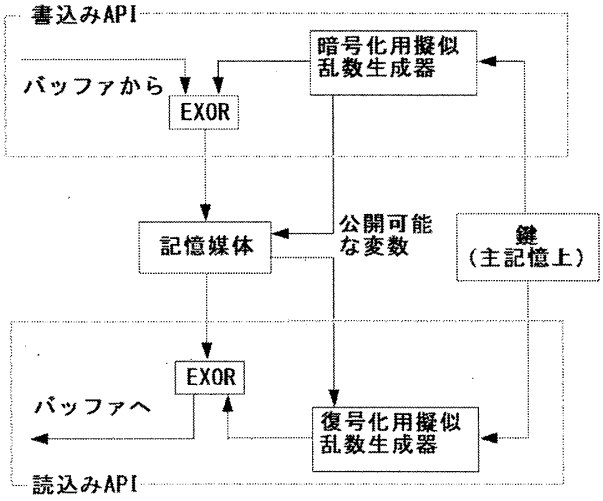

図１．暗号化，復号化の流れ

通常であればバッファから渡されたデータはそのまま記憶媒体へと書き込まれる．しかし図１から分かるように，本論文の書込み API では暗号化用疑似乱数生成器の出力との EXOR を計算し記憶媒体に書き込む．また乱数生成に必要な鍵は主記憶上から読み込み，復号化に必要であるが公開可能な変数も記憶媒体に書き込む．読込み API に関しても読み込んだデータをそのままバッファに渡すのではなく，復号化用疑似乱数生成器との EXOR を計算しバッファに渡すようにする．復号化の際，鍵は主記憶上から読み込み，必要な公開可能な変数は記憶媒体から読み込むようにする．

バッファから渡されたデータやバッファに渡すデータは暗号化されていないので，従来のアプリケーションに変更を施す必要がない．また，暗号化，復号化は自動的に行われるため，ユーザ側にも悟られる事なく暗号化，復号化が可能である．

3. 暗号化アルゴリズム

本章では書込み API で用いる暗号化アルゴリズムについて詳しく解説する．本論文では暗号化方式にストリーミング暗号方式を採用し，疑似乱数の生成には Mersenne Twister[2]に変形を施した Custom Mersenne Twister(以下 CMT)を用いる．

3.1 疑似乱数生成アルゴリズム： CMT

Mersenne Twister は長周期で高次元（６２３次元）に均等分布を持つ高速な疑似乱数生成アルゴリズムである．しかし，線形漸化式に基づく乱数生成を行っているため，生成される乱数は計算量理論的に安全な乱数ではない[3]．一方， CMT では Mersenne Twister の出力である 32bit の変数に不可逆圧縮かけ， 8bit の変数として出力としている．また， CMT では任意長の鍵の読込みに対応し，読込んだ鍵を用いて内部ベクトルの初期化ルーチンを実行することで，初期化空間を拡大できる．

表１． Mersenne Twister と CMT の実行時間の比較

アルゴリズムと鍵長	初期化に要する時間(msec)	出力速度（plots/sec）
Mersenne Twister 鍵なし	0.0169	11.3×10^7
CMT 鍵長 64bit	0.1457	6.0×10^7
CMT 鍵長 1024it	0.1640	6.0×10^7

表１の初期化に要する時間とは， Mersenne Twister と CMT において実行してから，鍵があれば読込み，内部ベクトルを初期化し終えるまでの時間を 100 回測定した平均を示している．また出力速度とは， Mersenne Twister と CMT において初期化終了後の乱数の生成速度を 100 回測定した平均を示している.なお動作環境を以下に示す

- プロセッサ： 2.4GHz Intel Core 2 Duo
- メモリ： 2 GB
- OS ： Windows VISTA
- 使用言語： C 言語

表１が示すように， CMT は Mersenne Twister よりも初期化に時間がかかるようになり，また乱数の生成速度が低下している．また， CMT は初期化にかかる時間は鍵長に依存するが，乱数生成速度は鍵長に非依存であることが CMT(鍵長 64bit)と CMT(鍵長 1024bit)の比較から分かる．

3.2 読み飛ばしアルゴリズム

また，同じ鍵を使い続ける限り， CMT の出力は毎回同じ数列になる．しかし，全てのファイルを同じ数列で暗号化する事は，非常に危険である．そこで，本機構ではファイルごとにランダムな数の要素を読み飛ばし，暗号化を開始する数列の要素を変更することで，計算量的安全性を強化する．

CMT の乱数生成速度は動作環境に依存するので，同じ要素数を読み飛ばすのにかかる時間も動作環境によって変化する．そこで本論文では，動作環境に応じて内部変数を動的に変更する事で，読み飛ばしにかかる時間の平均を決定できる読み飛ばしアルゴリズム Expectation One （以下 EO ）を提案する．

まず EO の疑似コードを以下に示す．

- genrand() ： CMT で乱数を発生させる関数
- rand() ： CPU 時間をシードとし乱数生成を行う関数
- match[ｎ] ：公開可能な変数（以下変数 match ）の前から ｎビット目
- IoM ：一致させるビット数． EO は IoM の増減により実行時間を調節できる．

```
Function ExpectationOne(match){
    Index = 0;
    while(match==0)
      match = rand();
    prev = genrand();
    while(index < IoM){
      temp = genrand();
      if(match[index] == '0' && temp < prev)
        index++;
      else if(match[index] == '1' && temp > prev)
        index++;
      else
        index = 0;

      prev = temp;
    }
    return match;
  }
```

EO では CMT の n 番目の出力と n+1 番目の出力とを大小比較したとき， n 番目の方が大きい確率と n+1 番目の方が大きい確率がともに５０％であると近似している．

また変数 match の前から n ビット目が 0 である確率と 1 である確率もともに５０％であると近似している．

EO は index が IoM を超えたとき変数 match を返して終了する． index がインクリメントされる条件は， match[n] が０で CMT の出力が直前の出力よりも小さい時か，match[n]が１で CMT の出力が直前の出力よりも大きいときであるので， index がインクリメントされる確率は 0.5*0.5+0.5*0.5=0.5 である．

index はインクリメントされなければ，０になるので，EO が終了する確率は， index が IoM 回連続でインクリメントされる確率に等しい．そこで P =CMT の出力速度（ plot/sec ）、S=時間（ sec ）とし,平均 S 以内に EO が終了する場合の関係式を以下に示す．

index がインクリメントされる確率とされない確率は等しく 0.5 である．また k 回目のインクリメントするかどうかの判定（以下 判定）がすんだ時点で、IoM 回連続インクリメントしている確率を $A_{IoM}(k)$とする。１度、IoM 回連続インクリメントしてしまったら EO は終了するが、便宜上、EO を続行したとして、後の判定に関係なく終了した状態のままであると考えることにする。また、k 回目の判定がすんだ時点で n 回連続インクリメントした確率を $A_n(k)$とする。k 回目の判定で index が０になる確率を $A_0(k)$とすると，

$A_0(k+1)=0.5*A_0(k)+0.5*A_1(k)+...+0.5*A_{IoM-1}(k)$
$A_1(k+1)=0.5*A_0(k)$
$A_2(k+1)=0.5*A_1(k)$
....
$A_{IoM-1}(k+1)=0.5*A_{IoM-2}(k)$
$A_{IoM}(k+1)=A_{IoM}(k)+0.5*A_{IoM-1}(k)$

である。ただし、$A_0(0)=1,A_1(0)=A_2(0)=...=A_{IoM}(0)=0$
$A_{IoM-1}(k),...,A_0(k)$を消去して、

$A_{IoM}(k+1)=A_{IoM}(k)+0.5^{IoM}*0.5*(1-A_{IoM}(k-IoM))$
ただし、
$A_{IoM}(0)=A_{IoM}(1)=...=A_{IoM}(IoM-1)=0, A_{IoM}(IoM)=0.5^{IoM}$
となる。つまり平均 S 以内に EO を終了させたいのならば k=P*S のとき $A_{IoM}(k)=1.0$ に近くなるように IoM の値を定めてやればよい．表２に IoM の値を変化させたとき， EO の実際の実行時間を示す．

表２． IoM の値と EO の実行時間

IoM の値	実行時間(sec)
25	0.044
26	0.074
27	0.112
28	0.166
29	0.292
30	0.486
31	1.104
32	1.684
33	3.752

各 IoM の値に対し EO を１００回実行したときにかかった時間の平均を EO の実行時間とする．なお表１と動作環境が同じであるので CMT の出力速度は 6.0×10^7(plots/sec)とする．

表２が示すように IoM の値が増加すると実行時間が増加する．一方， IoM の値が増加すると，計算量的安全性も増加する．実際に IoM を設定する場合には，動作環境やユーザの要望にあわせて IoM の値を適切に設定する事が重要になる．

3.3 暗号化の手順

CMT と EO を用いた暗号化の手順について解説する．図１から分かるように，本論文では暗号化用疑似乱数生成器の出力とバッファからのデータとの排他的論理和を計算する事で暗号化を行っている．暗号化用疑似乱数生成器とは CMT と EO を以下の手順で組み合わせたアルゴリズムに基づいて動作する．

(1) CMT に鍵を読み込ませ，内部ベクトルを初期化する．
(2) match=0 として EO を実行し， CMT の出力を読みとばす
(3) EO が終了したら変数 match を記憶媒体に書き込む
(4) EO が終了した以降の CMT の出力とバッファから渡された書き込むデータとの EXOR を１バイトずつ計算する
(5) (4)の計算結果を記憶媒体に書き込む

3.4 復号化アルゴリズム

復号化アルゴリズムについて解説する．復号化アルゴリズムも図１から分かるように，本論文では復号化用疑似乱数生成器の出力と記憶媒体から読み込んだデータとの排他的論理和を計算する事で復号化を行っている．復号化用疑似乱数生成器とは CMT と EO を以下の手順で組み合わせたアルゴリズムに基づいて動作する．

(1)CMT に鍵を読み込ませ，内部ベクトルを初期化する
(2)変数 match の値を記憶媒体から読み込む
(3)EO を実行し， CMT の出力を読みとばす
(4)EO が終了した以降の CMT の出力と記憶媒体から読み込んだデータとの EXOR を１バイトずつ計算する
(5) (4)の計算結果をバッファに渡す．

4. OS 側の変更点と API への実装

本性では，実際に，暗号化，復号化アルゴリズムを実装するにあたり， OS 側に施した変更点と， API への実装を解説する．

4.1 鍵へのアクセス制御

本論文では鍵を主記憶に読み込んで使用する．そこで API や権限のあるユーザ以外鍵へのアクセスを制限する事で鍵の漏洩や改変を阻止した．また，鍵は Microsoft の BitLocker[4]のような記憶媒体ごと暗号化するような技術と併用し安全に保管される必要がある．

4.2 ファイルハンドルの変更点

ファイルハンドルの変更点について解説する． CMT は 624 ワードの内部ベクトルと１ワードのカウンタを用いて乱数の生成を行う．ファイルハンドルに各ファイル固有の 625 ワードの配列を用意し， CMT の乱数の生成に用意した配列を用いる事により，暗号化，復号化の中断，再開ができるようになり，並列化も可能になる．

4.3 書き込み API への実装

書き込み API への実装について解説する．書き込み API を書き込みモードでのファイルオープン API と記憶媒体へ書き込む API とに分け，それぞれについて疑似コードを用いて説明する．書き込み API の実装のために，変数および関数の定義と上述の関数に下記の拡張を行う．

- mt[]はファイルハンドルに追加した 625 ワードの配列
- genrand()を genrand(Array)とし，配列 Array を CMT の内部ベクトルとカウンタとして乱数を生成する関数とする
- ExpectationOne(match)を ExpectationOne(match, Array)とし自関数内の genrand(Array)へ配列 Array を渡せるようにする．

4.3.1 書き込みモードでのファイルオープン API

書き込みモードでのファイルオープン API の疑似コードを以下に示す．引数 filename は開くべきファイルの名前を意味する．

```
Function FileOpen_w_API(filename){
  //標準的なファイルオープン API の処理を実行する
  FileHandle fh = Standard_FileOpen_w_API(filename);
  //ここから追加の処理をする
  match=0;
  match=ExpectationOne(match, fh->mt[]);
```

```
  //ファイル情報のリザーブ部に match を書き込む
  FileWrite_reserve(fh, match);
}
```

ファイルが暗号化されているのならばリザーブ部に match の値(0 以外)が書き込まれているので，暗号化されていないファイルと判別できる．

4.3.2 記憶媒体へ書き込む API

記憶媒体に書き込む API について解説する．記憶媒体へ書き込む API の疑似コードを以下に示す．引数 fh は書き込むべきファイルのファイルハンドルを意味する．引数 buffer は書き込むべきデータを意味する

```
Function FileWrite(fh, buffer){
  //buffer を暗号化する
  for(i=0; i<buffer.length; i++)
    data[i]=buffer[i] ^ genrand(fh->mt[]);
  //暗号化したデータを書き込むため
  //標準的な記憶媒体に書き込む API を実行する。
  Standard_FileWrite(fh, data);
}
```

記憶媒体には暗号化されたデータのみが書き込まれる．

4.4 読み込み API への実装

読み込み API について解説する．読み込み API を読み込みモードでのファイルオープン API と記憶媒体から読み込む API とに分け，それぞれについて解説する

4.4.1 読み込みモードでのファイルオープン API

まず，読み込みモードでのファイルオープン API について解説する．読み込みモードでのファイルオープン API の疑似コードを以下に示す．引数 filename は開くべきファイルの名前を意味する．

```
Funtcion FileOpen_r_API(filename){
  //標準的なファイルオープン API の処理を実行する
  FileHandle fh = Standard_FileOpen_r_API(filename);
  //ここから追加の処理をする
  //ファイル情報のリザーブ部から match を読み込む
  FileRead_reserve(fh, match);
  //match の値が 0 ならば暗号化されていないので，
  //以降は復号化処理を行わないようにする．
  if(match==0)
    exit(NotCoded);
  ExpectationOne(match, fh->mt[]);
}
```

4.4.2 記憶媒体から読み込む API

記憶媒体から読み込む API について解説する．記憶媒体から読み込む API の疑似コードを以下に示す．引数 fh は読み込むべきファイルのファイルハンドルを意味する．引数 buffer は読み込んだデータを格納する変数を意味する．

```
Function FileRead(fh, buffer){
  //標準的な記憶媒体から読み込む API を実行する
  Standard_FileRead(fh, data);
  //data を復号化し buffer に格納する
  for(i=0; i<data.length; i++)
    buffer[i]=data[i] ^ genrand(fh->mt[]);
}
```

バッファに渡されるのは復号化された情報である．

5. API の管理方法と情報流出

暗号化，復号化機能を有した API をどのように管理，使用するのかを解説する．本論文の書き込み API を用いると書き込もうとする全てのファイルが暗号化されてしまう．しかし，ウェブブラウジング時のキャッシュや機密情報の含まれない重要度の低いファイルなど暗号化しなくてもよいファイルや， OS のシステムファイルなど暗号化すると危険なファイルまで暗号化されてしまう．そこでシステム管理者に暗号化しないアプリケーションやユーザの振る舞いを選択させる事で，効率的に暗号化を行う事ができる．

また復号化に関しても，システム管理者に復号化を行うアプリケーションを選択させる．

こうする事でユーザが勝手にシステムにインストールしたアプリケーションには復号化の権限が与えられていないため，暗号化された情報が復号化されて流出するのを防ぐ事ができる．例えばユーザが勝手に P2P ソフトやキーロガーをインストールし，インストールしたソフトで何らかの重要な情報をアップロードしようとしても，アップロードされるのは暗号化された情報であり，情報流出による被害を最小限にとどめる事ができる．

6. 動作例

本章で暗号化，復号化機能を有した API を組み込んだ OS の動作を示す．次に示す OS は川合秀実氏著の「30 日でできる！ OS 自作入門」[5]を参考に作成した OS で，本論文にあわせて機能を拡張し，暗号化，復号化機能を有した API を組み込んである．なお、動作は実機上ではなく，エミュレーター「QEMU」[6]上での動作である．また暗号化，復号化機能はコマンドで簡易的に有効，無効を切り替えられるようにした．まず，作成した OS が正しく動作している事を示す．

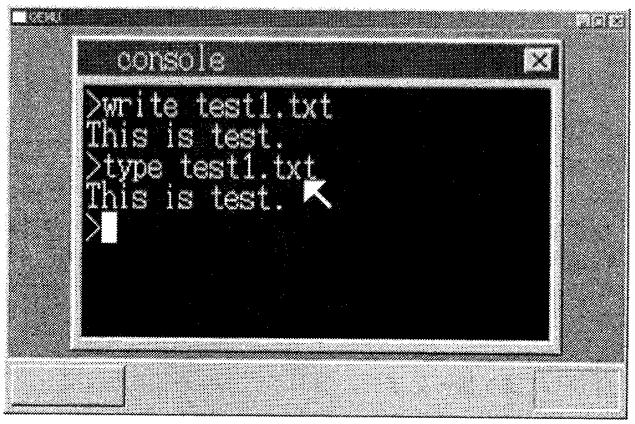

図２．動作確認

write コマンドはファイル名を引数とし，引数で指定されたファイルへ、入力した文字列を書き込むコマンドである． type コマンドはファイル名を引数とし，引数で指定されたファイルの内容を出力するコマンドである．図２．に示す通り， OS やコマンドが正しく動作していることがわかる．なお図２の時点では，暗号化，復号化機能は無効にしてある．

次に暗号化，復号化機能を有効にした時の動作を示す．

図３．暗号化，復号化機能を有効にした時の動作

図３より暗号化，復号化機能を有効にする前に作成した「test1.txt」が正しく表示され，暗号化，復号化機能を有効にした後作成した「test2.txt」も正しく表示されている事が分かる．また図３を見る限り，暗号化や復号化している事かどうかはユーザ側から判断できない事が分かる．しかし，暗号化，復号化機能を無効にすると，「test2.txt」が正しく表示できない事を次に示す．

図４．暗号化，復号化機能を無効にした時の動作

図４．より，「test2.txt」が暗号化されていて正しく表示できない事が分かる．つまり，「test2.txt」が暗号化されて保存されている事が分かる．

また本論文の手法では API の引数や呼び出し方，入出力は変更していないので，従来のアプリケーションも変更する事なく動作する．そこで次に暗号化，復号化機能を有する API を有効にしたときのアプリケーションの動作を示す．

次に示すアプリケーションは川合秀実氏が作成したテキストビューワ「tview」である．「tview」はファイル名を引数として，実行すると，指定されたファイルの中身を表示する．川合秀実氏が作成してからソースコードは変更していない．

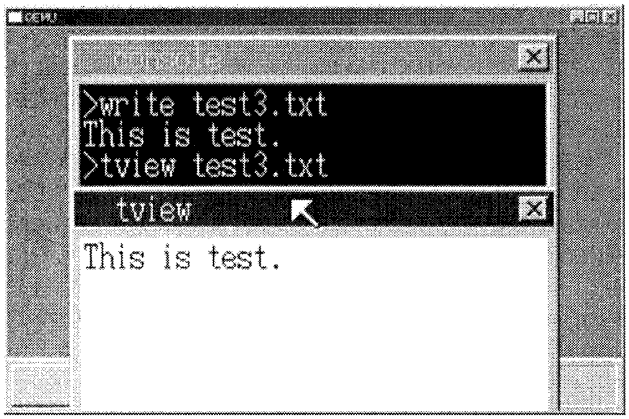

図５．暗号化，復号化機能有効時の「tview」の動作

図５より，「test3.txt」が正しく「tview」上で表示されているので，暗号化，復号化機能有効時でもアプリケーションが変更する事なく正しく動作している事が分かる．しかし，暗号化，復号化機能を無効にすると，「test3.txt」正しく表示できない事を次ぎに示す．

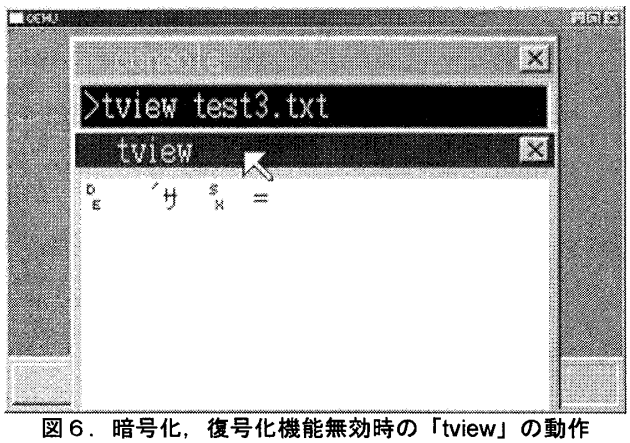

図６．暗号化，復号化機能無効時の「tview」の動作

図６．より，「test3.txt」が正しく表示できない事が分かる．つまり，暗号化，復号化機能がアプリケーションにも問題なく適用できていると言える．

また「test2.txt」と「test3.txt」の内容は「This is test」で，同一であるが，図４，図６に表示されている暗号化結果は異なっている．つまり， EO を実行する事で，ファイルごとに異なる暗号化ができている事を示している．

7. まとめと今後の課題

本論文ではユーザに負担をかけない OS レベルの動的な暗号化と復号化について提案した．ユーザが能動的に暗号化，復号化を行う従来の方法とは違い， OS レベルで暗号化，復号化を行う事で，ユーザが原因となる情報流出を減らす事ができると考えられる．また自作の OS 上で暗号化，復号化機能をもつ API と管理機構を実装する事ができた．

本論文では「ユーザに負担をかけない」とあるが，裏を返せば「ユーザ(システム管理者ではない)を信頼しない」という事である．情報流出事件が多発している現在，技術的な教育だけでなく，倫理的な教育も行い，ユーザの情報管理に対する意識を改善させることが急務である．特に今後はさらに情報技術が発達し，様々な場面で情報処理の技術が使われる事になる．その時に，情報が流出し放題では話にならない．どれだけ技術が進んだとしても最終的に使うのはユーザである．ユーザの意識を改善する一つの方法として，強制的に暗号化，復号化させる本研究を提案した．

今後の課題としてまず，暗号化アルゴリズムの改良があげられる．現在 CMT の出力は Mersenne Twister の出力を非可逆圧縮しているだけだが，SHS[7]の様により攻撃に耐性のあるハッシュ関数を用いる事で暗号化を強固なものにする．また Mersenne Twister の高速版である SFMT[8]や Mersenne Twister を用いた暗号化アルゴリズム CryptMT[9]を参考にして CMT の速度向上や改良をするといった方法が考えられる．

他には API の管理機構もシステム管理者の選択だけでなく，アプリケーションやユーザの振る舞いを解析し，ヒューリスティックに判断できれば，より実用的なものになると考えている．

参考文献

[1] 日本ネットワークセキュリティ協会（JNSA），“2008 年上半期 情報漏えいインシデント報告書”(2009).

[2] M. Matsumoto, T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”, ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998).

[3] 松本 眞，“擬似乱数ユーザーの方へ-Mersenne Twister 法開発者より(<特集>統計科学における乱数)”，日本統計学会誌. シリーズ J, Vol.35, No,2, pp.165-180 (2006).

[4] Microsoft TechNet, “BitLocker Drive Encryption”, http://technet.microsoft.com/en-us/windows/aa905065.aspx

[5] 川合秀実，“30 日でできる！ OS 自作入門”，毎日コミュニケーションズ(2006)

[6] QEMU CPU Emulator. http://fabrice.bellard.free.fr/qemu/.

[7] National Institute of Standards and Technology, ” Secure Hash Standard”, FIPS Vol.180,No.2(2002).

[8] Mutsuo Saito, Makoto Matsumoto, ″SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator″, Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, 2008, pp. 607 - 622.

[9] M Matsumoto, M Saito, T Nishimura, M Hagita, “CryptMT stream cipher version 3”, eSTREAM, ECRYPT Stream Cipher Project, Report(2005)