All rights are reserved and copyright of this manuscript belongs to the authors.
This manuscript has been published without reviewing and editing as received from the authors: posting the manuscript to SCIS 2008 does not prevent future submissions to any journals or conferences with proceedings.

SCIS 2008 The 2008 Symposium on Cryptography and Information Security
Miyazaki, Japan, Jan. 22-25, 2008
The Institute of Electronics, Information and Communication Engineers

共通言語基盤上における暗号アルゴリズムの効率的な実装手法

Efficient Implementation Techniques of Cryptographic Algorithms for Common Language Infrastructure

及川 一樹* 児玉 英一郎 * 王 家宏 * 高田 豊雄 *
Kazuki Oikawa Eiichiro Kodama Jiahong Wang Toyoo Takata

あ 近年，共通言語基盤上で動作するプログラムの開発が数多く行われている．また，情報やプライバシー保護の観点から，データを暗号化する必要性が高まっている．このため，共通言語基盤上で暗号化を行うプログラムの開発が頻繁に行われるようになった．こういった状況の中，共通言語基盤では，仮想実行システムと呼ばれる仮想マシン上でプログラムを実行する方式をとっているため，機械語から直接実行する場合と比べると，パフォーマンスの点で劣ってしまうということが特に問題視されている．そこで本稿では，共通言語基盤上における暗号アルゴリズムの効率的な実装手法を提案する．また，我々は，これを実際に適用して，代表的な共通鍵暗号アルゴリズムである Rijndael と Camellia，及び，公開鍵暗号である楕円曲線暗号を共通言語基盤上で実装し，性能評価を行った．評価の結果，本提案手法により，特に Camellia において，提案手法を使わない場合に比べ 40%程度の速度の向上が認められた．また，NTT による Camellia の C 言語を用いた実装よりも高速という結果が得られた．

キー 共通言語基盤, 共通鍵暗号, 公開鍵暗号, 楕円曲線暗号, 多倍長演算

1 はじめに

1.1 背景

近年，Microsoft が設計し，Ecma International や ISO，JIS などの標準規格でもある共通言語基盤 [1] を利用した開発が数多く行われている．この共通言語基盤は，共通中間言語と呼ばれる中間言語を用いて，仮想実行システムと呼ばれる仮想マシン上で実行する仕組みを採っている．そのため，直接機械語にコンパイルする場合と比べるとパフォーマンスの点で劣ってしまうが，言語に依存しないため，言語間での相互運用性が高く，豊富なクラスライブラリや，ガベージコレクションにより開発が容易といった利点を持っている．

一方，近年のアプリケーション開発において，情報やプライバシ保護の観点から，データを暗号化する必要性が高まっている．しかし，高速かつ強度の高い共通鍵暗号アルゴリズムとして知られる Camellia[2] や Rijndael[3] においても，CPU に強く依存するアセンブラによる実装でなければ，ギガビットイーサネットワークなどの広帯域を生かすような，高速な暗号化処理は望めない．

また，公開鍵暗号系は計算量の多い多倍長演算を必要とするため，パフォーマンスに難がある共通言語基盤上で動作するプログラムと，機械語から直接実行されるプログラムとでは，性能差が大きくなってしまう．

そのため，IPv6 の普及と共に今後登場するであろう，クライアント同士が相互に認証し，データをやりとりするような P2P アプリケーションや，PDA/スマートフォンのような，様々なアーキテクチャが存在する携帯端末向けのネットワークを活用したアプリケーションなどでは，暗号化や認証などの計算コストが無視できない割合を占めることになる．

そこで，本稿では共通言語基盤上における暗号アルゴリズムの効率的な実装手法を提案する．評価においては，共通鍵暗号方式として良く知られた Camellia や Rijndael，及び，公開鍵暗号方式である楕円曲線を利用したディジタル署名方式である ECDSA(Elliptic Curve Digital Signature Algorithm)[4] に対し，提案手法を適用して実装を行い，性能評価を行った．

1.2 共通言語基盤の実装

共通言語基盤は仕様であり，これに従った共通言語基盤の実装がいくつか存在している．特に有名なのは，共通言語基盤を設計した Microsoft 自身による実装で，Windows 上でのみ動作する.NET Framework[6] である．

* 岩手県立大学, 〒 020-0193 岩手県岩手郡滝沢村滝沢字巣子 152-52, Iwate Prefectural University, 152-52, Sugo, Takizawa, Takizawa village, Iwate 020-0193

また，オープンソースの実装として，Linux や Mac OS X, Solaris，BSD，Windows といった OS や s390 や SPARC, PowerPC，x86，x86-64，IA64，ARM といったアーキテクチャ上で動作する Mono[7] なども知られている．

1.3 共通言語基盤の特徴

共通言語基盤の特徴として，言語に依存しないという点が挙げられる．共通言語基盤上で動作するプログラムを開発する際には，共通中間言語にコンパイルできるプログラミング言語を用いれば良いため，開発のために新たな言語を習得する必要はない．現在では，Basic，C#, C/C++，COBOL，Fortran，JavaScript，LISP，Perl, Pascal など，50 種類以上のプログラミング言語が利用可能で，これらの言語間の相互運用も可能となっている．

また，共通言語基盤におけるメモリ管理には，ガベージコレクションを利用しているため，メモリリークなどのバグを排除でき，開発者への負担が軽減される点や，豊富なクラスライブラリを備えているため，開発が容易といった点が特徴として挙げられる．

1.4 本研究で用いる暗号アルゴリズムの概要

1.4.1 共通鍵暗号方式

本研究では，共通鍵暗号方式として，高い安全性と処理効率を誇る Camellia と Rijndael を利用する．

Camellia は 2000 年に NTT と三菱電機により共同開発されたブロック暗号アルゴリズムである．高い安全性を誇っており，欧州連合の暗号標準である NESSIE (New European Schemes for Signatures, Integrity, and Encryption)[8] や日本の暗号標準である CRYPTREC (Cryptography Research and Evaluation Committees)[9], ISO/IEC 18033，IETF の SSL/TLS や IPsec などの暗号標準として利用されている．

Rijndael は 1998 年に Vincent Rijmen と Joan Daemen によって設計されたブロック暗号アルゴリズムであり，Camellia と同じく NESSIE や CRYPTREC などの暗号標準となっているほか，アメリカ合衆国の暗号規格である DES の後継規格である AES として採用されている．

1.4.2 公開鍵暗号方式

本研究では，公開鍵暗号方式として，近年普及が進んでいる楕円曲線を利用したディジタル署名方式である ECDSA を利用する．

楕円曲線上の点の演算は群を成すため，群法則に基づく暗号を構成することができる．ECDSA は楕円曲線上の離散対数問題を安全性の根拠とするディジタル署名方式である．楕円曲線上の離散対数問題を解くアルゴリズムは指数的計算量を持つため，準指数的計算量で解くことのできる有限体の離散対数問題と比較すると，短い鍵長で同等の安全性を確保できる．即ち，RSA や DSA の 1024bit 相当の安全性が，ECDSA の場合 160bit 程度の鍵長で確保できる．このことは，計算量や通信量の削減などの利点を持つ．

なお，今回は 4.2 節で後述する通り，素体上の楕円曲線に基づくもののみを想定するため，演算の大部分は，素数を法とする多倍長演算が占める．

2 既存の実装手法

前述の暗号アルゴリズムには，プログラミング言語や共通言語基盤などの実行環境に依存しない，計算量を削減する実装手法が提案されている．

2.1 共通鍵暗号方式

Camellia や Rijndael に対する既存の高速化手法として，以下の 1~3 が仕様書に記載されている [2][3].

1. 演算の処理単位を CPU アーキテクチャの処理単位に合わせる
2. 副鍵をメモリ上にすべて展開する
3. 置換表に対して事前に適用できる関数を適用する

2.2 公開鍵暗号方式

楕円曲線暗号に限らず，公開鍵暗号方式は数論上の問題に依存している場合が多いため，公開鍵暗号方式の演算は有限体上の多倍長演算に依るところが大きく，剰余演算の高速化が公開鍵暗号方式の高速化につながる．また，今回は楕円曲線上の演算を利用するため，楕円曲線上の点の倍算の速度も ECDSA の高速化につながる．

剰余演算の実装方式としては，以下の 4 種類の方法が知られている．

- 除算を利用するクラシックな方法
- Barrett 還元を利用する方法 [10]
- Montgomery 算法を利用する方法 [11]
- 一般化メルセンヌ素数を利用する方法

Montgomery 算法を利用する方法は，数値を Montgomery 表現に変換してから，各種演算を行う方式である．数値を Montgomery 表現に変換するには，計算量の多い除算を行わなければならず，Montgomery 表現への変換コストは極めて高い．しかし，Montgomery 表現における乗算の計算効率は，そのコストを帳消しにするほど効率がよく，Montgomery 算法を利用した剰余演算は高速であることが知られている．また，一般化メルセンヌ素数を利用した方法は，法が一般化メルセンヌ素数

でなければならないという条件付きではあるが，Montgomery 算法を上回る計算効率で，高速な剰余演算を可能にする．

楕円曲線上の点の倍算の実装手法としては，主として以下の 5 種類の方法が知られている．

- バイナリ法
- 移動窓法
- バイナリ法 + NAF
- 移動窓法 + NAF
- 符号付き m 進展開窓移動法 [12]

楕円曲線上の点の倍算においてバイナリ法を利用した場合，一桁ごとに点の 2 倍算を行い，桁の値が 1 であるごとに点の加算を行うため，0 でない桁数が多いほど，計算量が多くなるという性質を持っている．NAF は $\{-1, 0, 1\}$ の 3 値で桁を表現する方式のことで，0 でない桁が少なくなるという傾向を持っている．そのため，NAF を用いることによって 0 でない桁が少なくなり，点の加算回数が減少する分，計算量を減らすことができる．

移動窓法は，m 進展開法の拡張版で，窓のサイズを n ビットとしたとき，事前計算として点の 2 倍算 1 回と，$2^{n-1}-1$ 回の点の加算を必要とする方式である．窓の最上位の桁が 0 の時は，その桁をスキップすることによって計算効率を向上させる手法をとっている．

符号付き m 進展開窓移動法は，符号付きの値を考えることによって，移動窓法と同じ事前計算量で窓のサイズを 1 ビット大きくすることを可能にする手法である．楕円曲線上の点の演算において，加算と減算の計算量がほぼ同じという性質を生かした手法である．

3 共通言語基盤向けの実装手法の提案

我々は，2 節で挙げた既存の手法と組み合わせて利用することができる，共通言語基盤において効果的な以下の実装手法を提案する．

1. 境界チェックの回避
2. 一時メモリ領域のスタック領域への確保
3. 論理積よりも高速なキャスト変換の利用
4. 参照頻度の高い変数の宣言場所の変更

以下，これらの手法の詳細を示す．

3.1 境界チェックの回避

共通言語基盤において配列を用いる場合，添え字の値の範囲が配列の大きさを超えていないかをチェックするための，境界チェックプログラムが自動的に挿入される．この処理は表のルックアップを多く行う共通鍵方式や，多倍長演算の処理速度に大きな影響を与える．そのため，これを回避するために配列をポインタに変換してから表のルックアップ処理を行うように変更することで，速度の向上が見込まれる．

3.2 一時メモリ領域のスタック領域への確保

共通言語基盤で配列を確保する場合，通常は，ガベージコレクションの対象となるヒープ領域に確保される．しかし，これをガベージコレクションの対象外であるスタック領域に確保することによって高速な確保，解放が可能となるほか，要素へのアクセス速度が向上する．但し，宣言したメソッド内でしか利用できないという制約があるため，一時領域という用途にしか利用できない．しかし，共通鍵暗号方式における副鍵生成部分では十分な効果を発揮する．

また，スタック領域に確保した配列はポインタを経由しなければアクセスできないため，自動的に 3.1 節の境界チェックを回避するという手法も適用される．

この手法の有用性を確認するために，表 1 の実行環境において，以下のヒープ領域に配列を確保するプログラム TestHeap と，配列をスタック領域に確保するプログラム TestStack を 10,000,000(1 千万) 回繰り返した時の実行時間を計測し，一回あたりの実行時間を算出した．二種類のアーキテクチャ上で二種類の共通言語基盤の実装を利用し実行時間を計測した結果，表 2 に示す通り，どの環境でも速度の向上が見られた．

- ヒープ領域に配列を確保するプログラム

```
static int TestHeap (int start) {
  int[] tmp = new int[8];
  tmp[0]=start;          tmp[1]=tmp[0]+1;
  tmp[2]=tmp[1]+tmp[0]; tmp[3]=tmp[2]+tmp[1];
  tmp[4]=tmp[3]+tmp[2]; tmp[5]=tmp[4]+tmp[3];
  tmp[6]=tmp[5]+tmp[4]; tmp[7]=tmp[6]+tmp[5];
  return tmp[7];
}
```

- スタック領域に配列を確保するプログラム

```
static unsafe int TestStack (int start) {
  int* tmp = stackalloc int[8];
  tmp[0]=start;          tmp[1]=tmp[0]+1;
  tmp[2]=tmp[1]+tmp[0]; tmp[3]=tmp[2]+tmp[1];
  tmp[4]=tmp[3]+tmp[2]; tmp[5]=tmp[4]+tmp[3];
  tmp[6]=tmp[5]+tmp[4]; tmp[7]=tmp[6]+tmp[5];
  return tmp[7];
}
```

表 1: 各提案手法を評価するのに利用した PC の構成

PC 名	CLR	Mono (x86)	Mono (PPC)
CPU	Opteron 270 (2GHz)		PowerPC 970 FX (1.8GHz)
OS	Windows Vista Business (32bit)	Gentoo Linux 2007.0 (Kernel 2.6.22)	
CLI 実装	Microsoft .NET Framework 2.0	Mono 1.2.5	

表 2: ヒープ領域とスタック領域上での処理時間

PC 名	ヒープ領域	スタック領域
CLR	56.83 ナノ秒	22.50 ナノ秒
Mono (x86)	165.13 ナノ秒	11.56 ナノ秒
Mono (PPC)	514.55 ナノ秒	56.10 ナノ秒

表 3: 論理積とキャスト変換の処理時間

PC 名	論理積	キャスト
CLR	0.7488 秒	0.4992 秒
Mono (x86)	0.8021 秒	0.8002 秒
Mono (PPC)	2.1130 秒	2.0713 秒

3.3 論理積よりも高速なキャスト変換の利用

今回の実装では 32bit アーキテクチャを対象とするため，32bit の数値から 8bit の値を取り出すという処理が，共通鍵暗号方式の実装において多く用いられる．通常は 32bit の数値と 0xFF との論理積をとることにより，8bit の数値を得るが，共通言語基盤の実装の一つである .NET Framework では，論理積をとるよりも 8bit 符号なし数値型である byte 型へキャスト変換した方が，高速に 8bit の数値を得られる．

32bit 数値から 8bit 数値を取り出す処理を利用したプログラムを以下に示す．また，本プログラムの実行時間を計測した結果を表 3 に示す．但し，ループ数である N には時間計測の精度を考慮して 100,000,000(1 億) を設定し，OS などの処理が割り込むことにより実行時間が延びてしまう可能性があることを考慮し，10 回計測した値のうち最も小さな値を採用した．

- 論理積を用いたプログラム

```
for (int i = 0; i < N; i ++)
  x = T[x&0xFF] ^ T[(x>>8)&0xFF]
      ^ T[(x>>16)&0xFF] ^ T[(x>>24)&0xFF];
```

- キャスト変換を用いたプログラム

```
for (int i = 0; i < N; i ++)
  x = T[(byte)x] ^ T[(byte)(x>>8)]
      ^ T[(byte)(x>>16)] ^ T[(byte)(x>>24)];
```

3.4 参照頻度の高い変数の宣言場所の変更

参照頻度の高い置換表などを，静的メンバ変数よりも少ない命令数でアクセスできるローカル変数やメソッドの引数を経由してアクセスすることで，高速化を図る．

即ち，置換表へアクセスする場合，クラス内からならどこからでも参照可能な，静的メンバ変数として宣言されている置換表を直接参照するのではなく，置換表を利用するメソッドの引数に置換表を渡し，メソッド内からは引数として渡された置換表を参照するように変更する．

4 性能評価

性能評価では，提案手法を用いて共通鍵暗号方式を実装し，提案手法を用いない場合と，C 言語による実装との暗号化速度の比較を行った．また，同様に提案手法を用いて公開鍵暗号方式である ECDSA を実装し，C 言語による実装との署名作成・検証速度の比較を行った．

なお C 言語による実装として，Camellia では設計元である NTT が公開している 32bit 最適化済みのプログラム [13] を利用し，Rijndael と ECDSA については，最もよく利用されている暗号通信プロトコルである SSL/TLS のオープンソース実装である OpenSSL[14] のバージョン 0.9.8g(執筆時点で最新) のものを利用した．

評価には，最も高速な共通言語基盤の実装である Microsoft .NET Framework 2.0 が動作する，表 1 の PC 名 CLR を用いた．

4.1 共通鍵暗号方式

共通鍵暗号方式の評価には，提案手法を利用し，現在，主に使われている共通鍵暗号方式である Rijndael と Camellia の実装を行い，1MB のデータの暗号化に要する時間を計測し，そこから暗号化速度を算出した．また，評価に利用した PC の CPU は 32bit CPU であるため，処理単位を 32bit とし，メモリも十分にあるため副鍵や置換表もすべてメモリ上に展開するといった，2.1 節で挙げた既存の実装手法をすべて取り入れた．なお，ここ

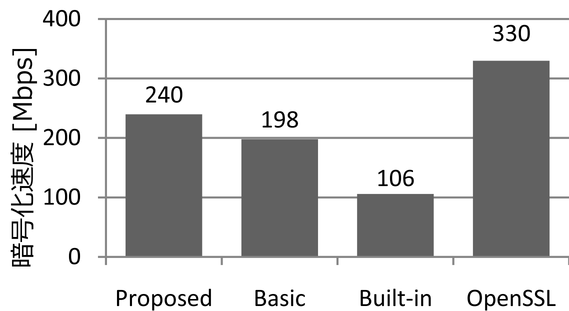

図 1: Rijndael の暗号化速度

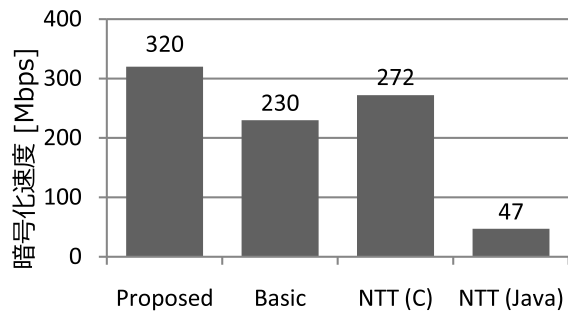

図 2: Camellia の暗号化速度

では鍵長・ブロック長は共に128bit，暗号利用モードはECB を利用し評価を行った．本評価結果を図 1 と図 2 に示す．

図 1 は Rijndael の暗号化速度を表しており，図 2 は Camellia の暗号化速度を表している．図中の Proposed が提案手法を利用した実装であり，Basic が既存の実装手法のみを利用した場合の実装，Built-in は.NET Framework に含まれる Rijndael 実装，OpenSSL は OpenSSL に含まれる C 言語による実装，NTT(C) は NTT が公開している 32bit 最適化済みの C 言語による実装，NTT(Java) は NTT が公開している Java による実装となっている．

図 1，図 2 より，提案手法を使うことによって提案手法を使わない場合と比べ，Rijndael においては 20%の高速化が確認でき，Camellia においては 40%の高速化が確認できた．また，Camellia においては NTT が公開している C 言語による実装よりも 17%高速という結果となっており，提案手法を利用した Camellia 実装の性能が非常に高いことが確認できる．

また，図 1 より，.NET Framework に組み込まれている Rijndael 実装は極めて遅いことがわかる．共通言語基盤上で動作し，暗号を用いるプログラムの殆どは，この実装を利用しているものと思われるため，本提案手法を利用した Rijndael 実装に置換することで，2.2 倍の性能向上が見込まれる．

Rijndael 実装において C 言語による実装である OpenSSL のものと，本稿提案手法を利用した Rijndael 実装の性能にはかなりの差が出た．Rijndael を実装する場合，既存の実装手法を用いると殆どが置換表の参照のみになってしまうため，提案手法を用いても，共通言語基盤における配列の参照速度と，機械語による参照速度には差があることに起因していると考えられる．

4.2 公開鍵暗号方式

公開鍵暗号方式の評価では，提案手法と既存の実装手法を利用して ECDSA の実装を行い，楕円曲線として SEC2[5] で定められている 192bit 楕円曲線 (secp192r1) を利用し，署名の作成時間と検証時間を計測した．

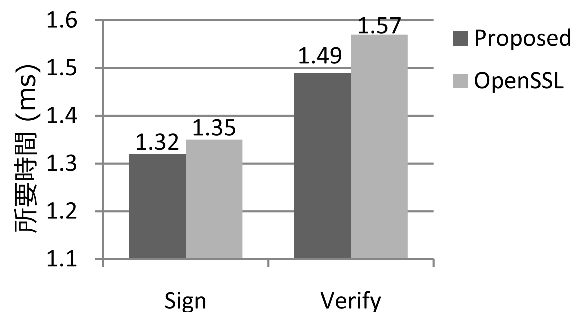

図 3: ECDSA の署名作成，検証所要時間 (1)

既存の実装手法としては，剰余演算として一般化メルセンヌ素数に基づく手法，点の倍算手法として，符号付き m 進展開窓移動法を利用している．

ECDSA では，署名の作成時に乱数を用いてパラメータを決定するため，その乱数の値によって計算量に差が出てしまう．そのため，一万回署名の作成と検証を行い，最も短い時間で処理を終えた組み合わせを結果として採用した．

本評価結果を図 3 と図 4 に示す．図中の Proposed が本提案手法を利用し共通言語基盤上で動作する実装であり，OpenSSL が OpenSSL プロジェクトで利用されている C 言語による実装，Bouncy Castle C#は，既存の共通言語基盤上で動作する ECDSA の実装となっている．

図 3 より，本提案手法を利用した実装は，C 言語による実装とほぼ同等の処理時間で，署名の作成と検証が行われていることが認められる．また，図 4 より，既存の共通言語基盤上で動作する ECDSA の実装と比べても 60 分の 1 の時間で，署名の作成と検証処理が行われていることが認められ，本提案手法を利用した ECDSA の実装が非常に速いことが確認できる．

5 おわりに

本稿では，共通言語基盤上で暗号アルゴリズムを効率的に実装する手法の提案を行った．また，本提案手法の実装を利用した性能評価結果について報告を行った．その結果，共通鍵暗号方式である Rijndael では 20%，Camellia では 40%の速度の向上が認められ，公開鍵暗号方式であ

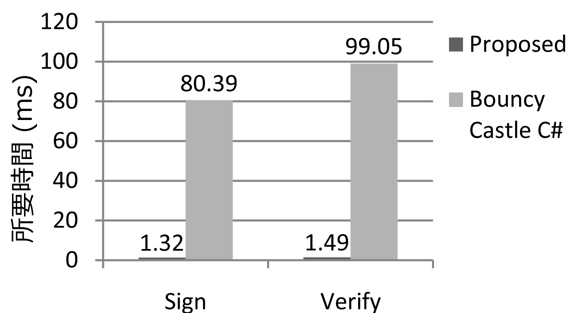

図 4: ECDSA の署名作成，検証所要時間 (2)

る ECDSA では，C 言語による実装とほぼ同等の処理時間で，署名の作成と検証が行えることが確認された．以上のように，本提案手法により仮想マシン上で動くプログラムながら，直接，環境固有の機械語にコンパイルされる C 言語で記述されたプログラムに近い速度を達成できた．

なお，本稿で実装したプログラムは Web 上で公開しており，修正 BSD ライセンスの下，自由に利用することが出来る．より詳しい情報やソースコードは `http://trac.panicode.com/crypto/` で入手することが出来る．

謝辞

本研究は岩手県立大学 2006 年度 PBL(Project Based Learning) の補助によって行われた．学部 2 年生であってもプロジェクトを申請することによって，学部からの補助が受けられる PBL という制度に感謝の意をここに記す．

また，プロジェクトメンバーとして協力してくれた，岩手県立大学の酒匂大輔氏と牧原大樹氏に感謝する．

参考文献

[1] Ecma International, “Standard ECMA-335 4th Edition June 2006 Common Language Infrastructure (CLI) Partitions I to VI”, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf

[2] 市川哲也, 松井充, 中嶋純子, 時田俊雄, 青木和麻呂, 神田雅透, 盛合志帆, “128 ビットブロック暗号 Camellia アルゴリズム仕様書”, http://info.isl.ntt.co.jp/crypt/camellia/dl/01jspec.pdf

[3] Joan Daemen, Vincent Rijmen, “AES submission document on Rijndael, Version 2”, http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf

[4] Certicom Research, Standards for Efficient Cryptography Group, “SEC1: Elliptic Curve Cryptography”, http://www.secg.org/download/aid-385/sec1_final.pdf

[5] Certicom Research, Standards for Efficient Cryptography Group, “SEC2: Recommended Elliptic Curve Domain Parameters”, http://www.secg.org/download/aid-386/sec2_final.pdf

[6] Microsoft .NET Framework, http://msdn2.microsoft.com/netframework/

[7] Mono, http://www.mono-project.com/

[8] NESSIE (New European Schemes for Signatures, Integrity, and Encryption), http://www.cryptonessie.org/

[9] CRYPTREC (Cryptography Research and Evaluation Committees), http://www.cryptrec.jp/

[10] P.D.Barrett, “Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor”, Advances in Cryptology, Proc. Crypto’86, LNCS 263, A.M. Odlyzko, Ed., Springer-Verlag, pp. 311–323, 1987.

[11] P.L.Montgomery, “Modular multiplication without trial division”, Mathematics of Computation, Vol. 44, pp. 519–521, 1985.

[12] Ian Blake, Gadiel Seroussi, Nigel Smart, *Elliptic Curves in Cryptography*, London Mathematical Society Lecture Note Series 265, Cambridge University Press, 1999

[13] NTT, Camellia 暗号エンジン, http://info.isl.ntt.co.jp/crypt/camellia/engine.html

[14] OpenSSL: The Open Source toolkit for SSL/TLS, http://www.openssl.org/