情報処理学会論文誌 Vol.56 No.9 1846–1856 (Sep. 2015)

暗号モジュールを搭載したプロセッサにおける暗号処理のオフローディング方式の効率化の提案と評価

金子 洋平[1,a)] 齋藤 孝道[2] 菊池 浩明[2]

受付日 2014年12月5日, 採録日 2015年6月5日

概要：近年，モバイル端末において暗号処理を行う機会が増えている．この処理は汎用プロセッサにとって負荷が高く，特に，モバイル端末では，システム全体の処理のボトルネックなどの原因となる．この負荷を軽減する方法として，暗号処理に最適化されたハードウェアモジュールを搭載したプロセッサを利用し，暗号処理をオフロードする方法がある．そこで本論文では，モバイル端末上で暗号処理のオフロードを行うために AM3358 プロセッサを採用し，オフロードを行う．さらにオフロードによって計算資源の空く CPU も同時に利用することで，モバイル端末における暗号処理の高速化を行い，その性能を測定した．

キーワード：オフロード，暗号処理，暗号モジュール

Cryptographic Operation Load-balancing between Cryptographic Module and CPU

Yohei Kaneko[1,a)] Takamichi Saito[2] Hiroaki Kikuchi[2]

Received: December 5, 2014, Accepted: June 5, 2015

Abstract: Mobile devices such as smartphones and tablets have permeated into our daily lives and are now often indispensable because of the constant Internet access they provide. Furthermore, with ever increasing concerns regarding privacy and security, it has become popular to utilize cryptographic operations when accessing Web application servers from such devices. However, since such operations cause high loading on the central processing units (CPUs) of personal computers (PCs) or servers, mobile device CPUs now often come equipped with hardware cryptographic modules. These cryptographic modules are frequently utilized by many mobile device applications via a process known as offloading. However, when all cryptographic operations can be offloaded to cryptographic modules, device CPUs may become idle, which is an ineffective use of total computing resources. In this paper, we propose the simultaneous balanced offloading of cryptographic operations to the cryptographic module of an AM3358 processor and the CPU via load-balancing. We evaluated the performance of our implementation, and concluded that it is capable of working effectively.

Keywords: off-loading, cryptographic operation, cryptographic module

1. はじめに

近年，モバイル端末において，個人情報など秘匿性の高い情報を扱う機会が増えてきている．それにともない，これらの情報を保護するためにモバイル端末上で暗号処理を行う機会も増えている．しかし，暗号処理は汎用プロセッサコアにとって負担が大きく，システム全体の処理のボトルネックとなる場合がある．特にモバイル端末に搭載される CPU は，一般的なサーバや PC などに搭載される CPU よりも性能が低く，システム全体の処理のボトルネックとなる．

そこで，暗号処理をオフロードする目的で，暗号処理を専門に行うハードウェアモジュール（以降，暗号モジュールと呼ぶ）をコア内に持つ CPU が数多く登場した [1], [2]．また，その種の CPU における効率的な暗号化/復号処理のオフロード技術についての研究も行われており，我々

[1] 明治大学大学院
Graduate School of Meiji University, Kawasaki, Kanagawa 214–8571 Japan
[2] 明治大学
Meiji University, Kawasaki, Kanagawa 214–8571 Japan
[a)] youhei.amd13@gmail.com

の研究グループでも，オフロード技術の研究を行っている [3], [4], [5].

本論文では，Texas Instruments 社の AM3358 プロセッサ [6] を利用し，モバイル端末上の暗号処理問題を研究する．AM3358 プロセッサは，ARM Cortex-A8 コアベースで，共通鍵暗号化方式やハッシュ処理をサポートする暗号モジュールを搭載したシングルコア CPU である．AM3358 プロセッサ用の OS として Texas Instruments 社から Linux OS および Android OS が提供されており，本研究では Linux OS を採用する．また，暗号ライブラリとして，広く使われている OpenSSL [7] を利用する．OpenSSL は，AM3358 プロセッサの暗号モジュールを直接利用するための仕組みを持っていない．そこで，OCF（OpenBSD Cryptographic Framework）[8], [9] と AM3358 プロセッサ用の暗号モジュールのデバイスドライバを介し，暗号モジュールにアクセスする．

この環境で，暗号処理以外の処理を行わず，暗号処理のオフロードを行う場合，TI 社によると，CPU 使用率が大きく下がることが分かった [10]．また，モバイル端末では，PC やサーバなどと比べると，用途からしても同時に複数のアプリケーションを大量に利用することは少ないと考えられる．このことは，計算資源の少ないモバイル端末において，オフロード時，CPU の計算資源が有効活用されていない状況にあるとも解釈できる．そこで，本論文では独自に，暗号モジュールと CPU のいずれかに暗号処理を振り分ける仕組みを提案し，暗号処理の高速化を試みる．また，いくつかの代表的な処理を振り分けるアルゴリズムを実装し，提案システムと比較を行い，その評価を行った．

2. 研究環境

2.1 AM335x プロセッサ

本論文で利用した AM3358 プロセッサのアーキテクチャを図 1 に示す．

AM3358 プロセッサは，ARM Cortex-A8 アーキテクチャベースのシングルコアプロセッサである．SIMD の拡張命令である NEON に対応している．L1 キャッシュはデータキャッシュが 32 KB，命令キャッシュが 32 KB で，いずれも 1 ビットエラー検出可能である．L2 キャシュは 256 KB 搭載していて，ECC に対応している．動作周波数は，600 MHz，800 MHz，1,000 MHz で動作し，処理速度はそれぞれ，600 MIPS，1,600 MIPS，2,000 MIPS である．DRAM は，LPDDR-400，DDR2-532，DDR3-606 に対応しており，本論文で AM3358 プロセッサの評価に使用した評価ボード TMDXEVM3358 では，DDR2 を 512 MB 搭載している．稼働させることが可能な OS は，Linux，Android，Windows Embedded CE である．

暗号モジュールは，AM3358 プロセッサに 1 つ搭載されていて，共通鍵暗号化方式やハッシュ関数に対応しており，

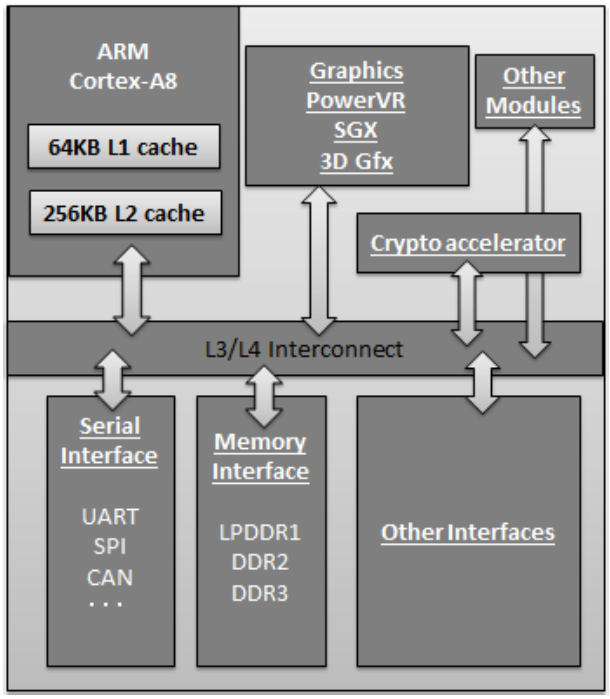

図 1 AM3358 プロセッサ
Fig. 1 AM3358 processor.

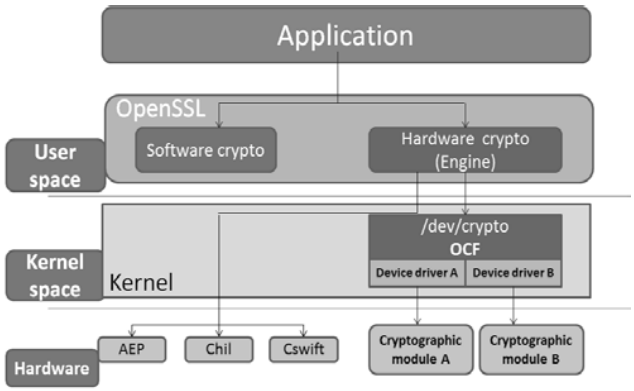

図 2 暗号モジュールの利用
Fig. 2 Utilization of cryptographic module.

AES, SHA-1, MD5 と RNG（Random Number Generator）を利用可能である．AES は，鍵長 128 bit，192 bit，256 bit にそれぞれ対応しており，暗号化利用モードは ECB, CBC, CTR, CFB, OFB, XTS, GCM, CCM, ICM, F8 および CBC-MAC に対応している．

2.2 OCF

2.2.1 OCF の概要

OCF（図 2 におけるカーネル空間参照）は，OpenSSL, OpenSSH [11] などのアプリケーションから様々なハードウェアアクセラレータが提供する暗号処理機能を利用するための共通のインタフェースを提供する API と，ハードウェアアクセラレータのデバイスドライバから構成されたミドルウェアである．ただし，OCF は BSD や Linux

のカーネルミドルウェアとして開発されているので，他のOS で利用するには，その環境に合わせた改修が必要となる．また，OCF がサポートしていない暗号モジュールに対しては，それに対応するデバイスドライバを用意する必要がある．AM3358 プロセッサの暗号モジュールは，標準対応していないため，別途デバイスドライバを用意する必要がある．

2.2.2 OCF の利用方法

OCF の利用について，まず，アプリケーションから OCF へのアクセス方法について説明する．アプリケーションはキャラクタ型のデバイスノードに対して，システムコールを発行することで，各システムコールに対応する OCF 内で定義された関数を呼び出すことができる．これは，標準的な file operations 構造体を利用しており，アプリケーションは open()，ioctl() システムコールを呼び出すことで，OCF へアクセスする．

暗号処理を，OCF を経由して暗号モジュールにオフロードする場合には，まず，アプリケーションと OCF 間で，後述するセッションを生成する．セッションの生成後，アプリケーションが OCF に対して，暗号処理の実行を指示することで，暗号モジュールの暗号処理機能を利用できる．暗号処理の終了後，アプリケーションと OCF 間のセッションを解放する．

2.2.3 OCF の制御

OCF を制御するためには，アプリケーションから制御リクエストを引数として ioctl() システムコールを /dev/crypto に対して発行する．その OCF への制御リクエストには，以下の 3 種類があり，次のように使い分ける．

CIOCGSESSION

アプリケーションと OCF 間でセッションの生成を行う際の制御リクエストである．セッションを生成すると，セッション ID を返り値としてアプリケーションへ渡す．このセッションとは，アプリケーションと OCF 間で暗号鍵や暗号化方式などの暗号情報と OCF 内の暗号情報を格納した構造体への識別子であるセッション ID を共有した状態である．

CIOCCRYPT

暗号処理をアプリケーションから暗号モジュールにオフロードする際の制御リクエストである．この制御命令を発行すると，まず，アプリケーションから OCF にセッション ID，IV と平文を転送する．OCF 側では，セッション ID により，暗号情報を格納した構造体を取得する．その後，暗号モジュールに暗号処理をオフロードする．

CIOCFSESSION

アプリケーションと OCF 間のセッションを解放する際の制御リクエストである．セッションの解放とは，暗号情報を格納した構造体の削除とセッション ID の削除を行うことである．

2.3 OpenSSL

OpenSSL（図 2 のユーザ空間参照）は，SSL [12] や TLS [13] だけでなく，証明書の発行といった PKI（Public Key Infrastructure）関連の処理や公開鍵暗号化方式や共通鍵暗号化方式などを共通インタフェースで利用可能とした API ライブラリを含むツールキットである．これらの機能は，libssl と libcrypto という共有ライブラリによってアプリケーションから利用することができる．libssl は SSL/TLS 通信に関する機能を提供し，libcrypto は，暗号処理に関する機能を提供する．また，libcrypto は，暗号処理を共通のインタフェースで利用を可能にする EVP API を提供している．

OpenSSL は，AEP，Chil や Cswift といったいくつかの暗号モジュールに標準対応しており，アプリケーションからそれらを利用するために ENGINE API を提供している．また，標準対応していない暗号モジュールについても，OCF などのミドルウェアとデバイスドライバを介し，ENGINE API から利用することができる．

2.3.1 OpenSSL から OCF 利用の手続き

OpenSSL から OCF を利用するには，ENGINE API に用意されたオブジェクト（以降，ENGINE オブジェクトと呼ぶ）を利用し，**図 3** に示す所定の手続きを行う必要がある．

図 3 の 2 行目で ENGINE_load_cryptodev 関数を呼び出すと，この関数内で，OCF に対応する ENGINE オブジェクトを生成し，それらを ENGINE オブジェクト専用のリスト（以降，ENGINE リストと呼ぶ）に登録する．4 行目では，暗号処理モジュールの識別子を引数として指定して ENGINE_by_id 関数を呼び出し，引数に対応した ENGINE オブジェクトを ENGINE リストから取得する．ここでは，引数に指定した “cryptodev” に対応する ENGINE オブジェクトを取得している．6 行目の ENGINE_set_default 関数を呼び出すことで，暗号処理モジュールが対応している暗号アルゴリズムを ENGINE オブジェクトに登録する．

このようにして，アプリケーションが EVP API を用いて暗号処理を実行する際に，OCF に暗号処理を実行させることができる．

2.3.2 OpenSSL 暗号処理の受け渡し

図 4 に，OpenSSL から EVP API を用いて OCF に

```
1 ENGINE *e;↓
2 ENGINE_load_cryptodev();↓
3   ↓
4 if(!(e = ENGINE_by_id("cryptodev"))){↓
5   /*Error Handling codes*/  ↓
6 else if(!ENGINE_set_default(e,ENGINE_METHOD_ALL))↓
7   /*Error Handling codes*/↓
```

図 3 OpenSSL からのオフロードの手続き
Fig. 3 Off-loading procedure with OpenSSL.

```
1 EVP_CIPHER_CTX ctx;↓
2 EVP_EncryptInit(&ctx, EVP_aes_128_cbc(),key, iv);↓
3 ↓
4 EVP_EncryptUpdate(&ctx, output,&outlen,input,len);↓
5 EVP_EncryptFinal(&ctx, outbuf + outlen, &tmplen);↓
6 ↓
7 EVP_CIPHER_CTX_cleanup(&ctx);↓
```

図 4 ENGINE を利用した暗号処理
Fig. 4 Encryption with ENGINE API.

暗号処理を受け渡す一連のソースコードを示す．図 4 の EVP_EncryptInit 関数，EVP_EncryptUpdate 関数，EVP_EncryptFinal 関数，EVP_CIPHER_CTX_cleanup 関数は，それぞれ OCF 内部の CIOCGSESSION，CIOCCRYPT, CIOCFSESSION 制御リクエストに対応している．

OpenSSL が OCF に暗号処理を受け渡すには，まず，EVP_EncryptInit 関数を呼び出す．この関数内で CIOCGSESSION 制御リクエストが発行され，暗号方式，鍵などの暗号情報を OCF に渡し，OCF とのセッションを生成する．次に，EVP_EncryptUpdate 関数と EVP_EncryptFinal 関数を呼び出し，暗号処理を行う．この関数内で CIOCCRYPT 制御リクエストが発行され，OCF に暗号処理が受け渡される．暗号処理が終わると，EVP_CIPHER_CTX_cleanup 関数を呼び出す．この関数内で CIOCFSESSION 制御リクエストを発行することで，OCF とのセッションが解放される．なお，ENGINE API を利用する場合は，EVP_EncryptInit_ex 関数の引数で使用するエンジンへのポインタを指定できるが，本論文の実装では，2.3.1 項の方法でこれを行ったため，“ex” のついていない関数を利用した．

3. 提案システム

3.1 提案システムの概要

提案システムは，OpenSSL から OCF を介して，暗号モジュールで暗号処理を行うスレッド（以降，暗号モジュールスレッドと呼ぶ），OpenSSL から CPU で暗号処理を行うスレッド（以降，CPU スレッドと呼ぶ）および，暗号処理を行うデータを保持するキューの 3 つから構成される．**図 5** および，3.2 節にこれらのスレッドとキューの関係を表す．本研究では，これらの 3 つを独自に設計し用意した．これらの設計により，CPU 上で行われている提案システム以外のプログラムの処理への影響を抑え，暗号処理を暗号モジュール単体で行う場合より高速化する．

3.2 提案システムの実装

3.2.1 キューの説明

提案システムで用意したキューは，CPU スレッドと暗号モジュールスレッドを起動する前に，平文データをファイルから読み込み保持する．キューに保持されるデータは，読み込んだファイルそのもので，分割はしない．その

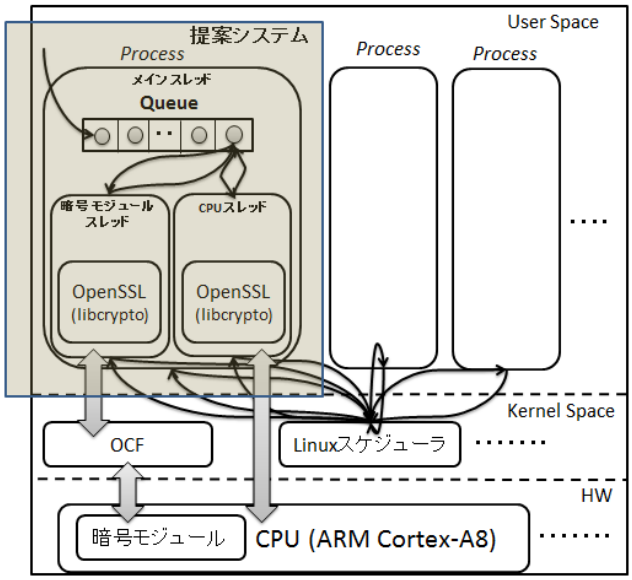

図 5 提案システムの概念図
Fig. 5 Concept of proposed system.

うえでファイルごとに暗号化する．これらのデータは，暗号モジュールスレッドと CPU スレッドのいずれかによってキューから取り出される．

3.2.2 暗号モジュールスレッドの実装

暗号モジュールスレッドとは，OpenSSL の EVP API と OCF を利用し，もっぱら暗号モジュールにおいて暗号処理を行うスレッドである．

初めに，暗号モジュールスレッドでは，2.3.1 項で説明したとおり，ENGINE_load_cryptodev 関数，ENGINE_by_id 関数，および ENGINE_set_default 関数を呼び出し，OpenSSL から OCF を利用する手続きを行う．

その後，2.3.2 項で示した OpenSSL から EVP API を用いて暗号処理を行う関数を実行する．EVP_EncryptInit 関数により，暗号処理に利用する IV，鍵および暗号アルゴリズムなどを決定する．次に，キューから暗号処理を行うデータを取り出し，そのデータに対し，EVP_EncryptUpdate 関数と EVP_EncryptFinal 関数を利用し，暗号処理を行う．暗号処理が終了したら，再度，キューからデータの取り出しを行い，そのデータに対し暗号処理を行う．これらの，データの取り出しから暗号処理の流れを，キューのデータがなくなるまで繰り返す．

最後に，EVP_CIPHER_CTX_cleanup 関数を呼び出し，暗号処理に利用した IV，鍵および暗号アルゴリズムなどの情報を破棄する．

これらの暗号モジュールスレッドにおける処理の流れを**図 6** に示す．

3.2.3 CPU スレッドの実装

CPU スレッドは，OpenSSL の EVP API を利用し，CPU で暗号処理を行うスレッドである．

CPU スレッドでは，OCF を利用しないので，それらに

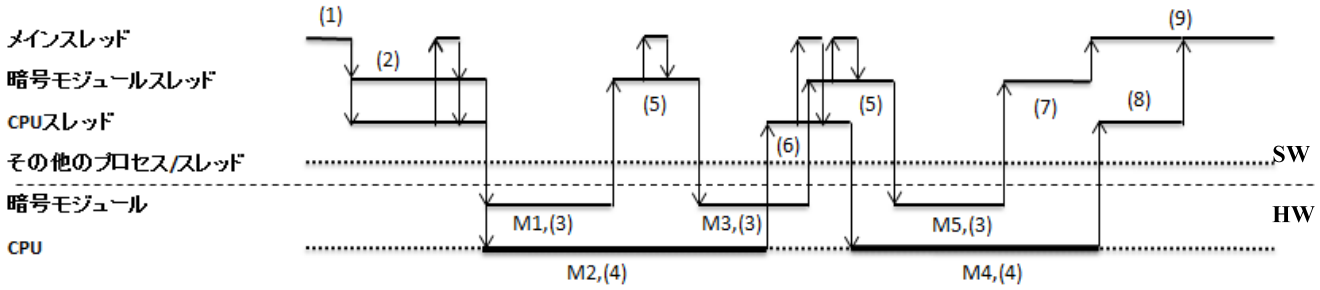

図 8 提案システムの動作例
Fig. 8 An execution process of proposed system.

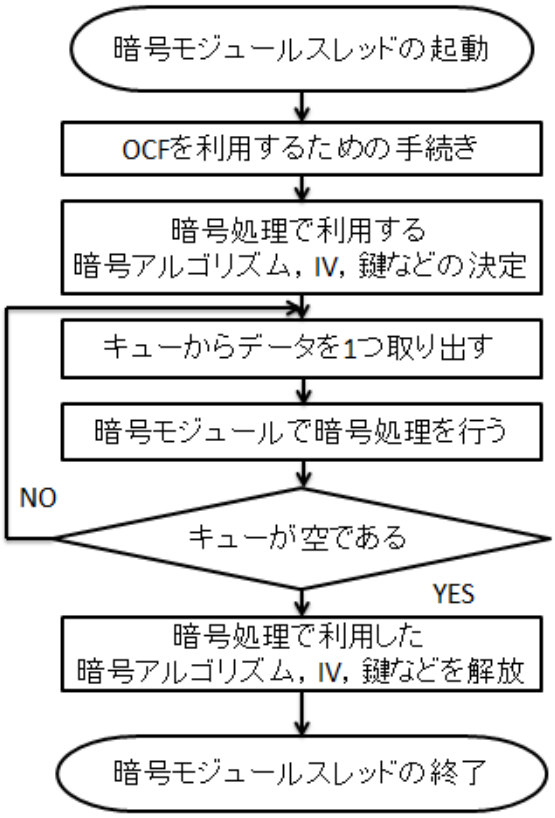

図 6 暗号モジュールスレッドの動作
Fig. 6 Process of cryptographic module thread.

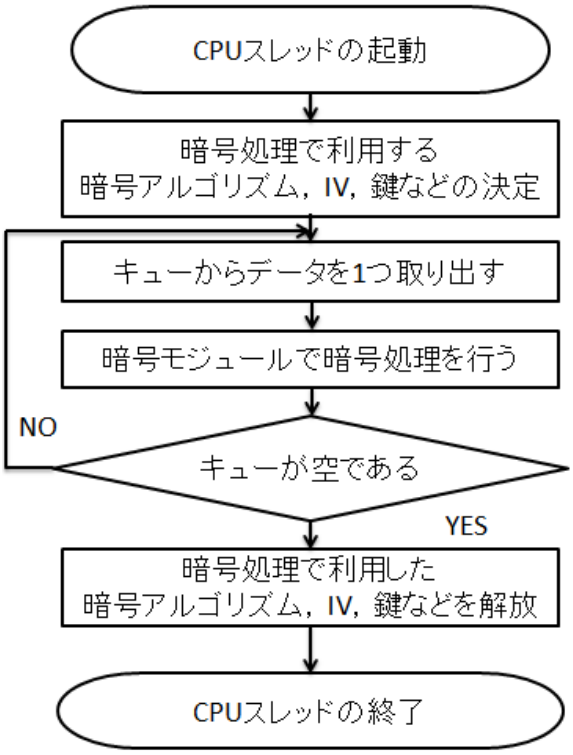

図 7 CPU スレッドの動作
Fig. 7 Process of CPU thread.

関する処理を行わない以外は，暗号モジュールスレッドと同様の手順で処理を行う．

CPU スレッドにおける処理の流れを図 **7** に示す．

3.3 提案システムの動作例

図 **8** を用いて提案システムの動作例を説明する．ここで，暗号モジュールスレッド，CPU スレッド，これらを呼び出すメインスレッド，および，提案システムと関係のないその他のプロセス/スレッドの動作の流れを図示する．なお，実線が暗号処理に関連する処理の実行，破線がその他のプロセス/スレッドの実行を表すものとする．図 8 では，キューが暗号処理を行うデータ M1，M2，M3，M4，M5 を保持している場合を想定している．ここで，この節の番号 (1)〜(9) は，図 8 の番号と対応するものとする．

(1) メインスレッドは，キューに平文データを格納する．また，暗号モジュールスレッドと CPU スレッドを起動する．
(2) 暗号モジュールスレッドと CPU スレッドがそれぞれ独立に，任意のタイミングで，キューから平文データを取り出し，暗号処理を行う．
(3) 暗号モジュールにより，暗号処理を行う．
(4) CPU により，暗号処理を行う．
(5) 暗号モジュールが暗号処理を終えると，再度暗号モジュールスレッドはキューからデータを取り出し暗号処理を行う．
(6) CPU が暗号処理を終えると，再度 CPU スレッドが，キューからデータを取り出し暗号処理を行う．
(7) 暗号モジュールが暗号処理を終えると，さらにキューにデータがないことを確認し，処理を終了する．

(8) CPU が暗号処理を終えると，さらにキューにデータがないことを確認し，処理を終了する．
(9) 暗号モジュールスレッドと CPU スレッドの待ち合わせを行い，それぞれスレッドを終了する．

図 8 では，CPU で行う処理として，CPU スレッドによる暗号処理とその他のプロセス/スレッドによる処理を示した．実際には，メインスレッド，暗号モジュールスレッド，および，CPU スレッドによる暗号化といった，各々の処理に関する管理処理も CPU で行われる．これらの管理処理やその他のプロセス/スレッドによる処理は，スレッドやプロセスごとに Linux スケジューラによって適切にスケジューリングされる．本論文では，暗号処理以外のプロセス/スレッドの他の処理のパフォーマンスを下げることなく，暗号処理の高速化を追求した．しかし，PC やサーバに比べ同時に実行するアプリケーションが少ないと考えられるモバイル端末でも，複数のアプリケーションを同時に実行している場合があることも考えられる．そのため，本論文では，CPU がその他のプロセス/スレッドを処理することを考慮し，暗号処理を行っているときも CPU リソースが Linux スケジューラによって適切にスケジューリングされる実装とした．

4. 評価

4.1 評価環境

提案システムの評価を行うために，**表 1** に示す環境を用意した．CPU は，AM3358 プロセッサを利用した．この CPU は，ARM Cortex-A8 を搭載し，600 MHz，800 MHz および，1,000 MHz で動作し，処理速度はそれぞれ，600 MIPS，1,600 MIPS および 2,000 MIPS である．また，プログラムの実行環境および開発環境は，Sitara SDK 6.00.00 を利用した．これは Linux Kernel の 3.2.0 が使われている．

4.2 計測方法

4.1 節で説明した評価環境上で，1 回に 4,000 個のファイルをキューに保持しそのファイルを暗号化し，処理時間およびそれぞれのスレッドが暗号化したファイルの個数を計測した．

CPU スレッドと暗号モジュールスレッド単体での暗号処理と提案システムの比較だけでなく，ラウンドロビンとポーリングについても処理速度の比較を行った．以降で説明する 5 つの評価プログラムにおいて評価を行い，それぞれ結果を比較した．

(1) CPU スレッドのみで暗号化をした場合（「CPU スレッドのみ」と表記）
(2) 暗号モジュールスレッドのみで暗号化をした場合（「暗号モジュールスレッドのみ」と表記）
(3) CPU スレッドと暗号モジュールスレッドにキューのデータをラウンドロビンで使い，重み付けせずに均等に割り振った場合（ラウンドロビンと表記）
(4) CPU スレッドと暗号モジュールスレッドをつねに監視し，暗号処理を終えたスレッドにキューのデータを割り当てる場合（ポーリングと表記）
(5) 提案システムで暗号化した場合（提案システムと表記）

ファイルサイズは，OpenSSL での暗号処理の速度を計測する openssl speed コマンドの出力が 16，64，256，1,024，8,192 byte であるので，基本的にはそれに合わせた．さらに小さい 8 byte，SSL/TLS の最大フレームサイズの 16,384 byte，また，1,024 byte と 8,192 byte では 8 倍の差があるので，4,096 byte を加えたので，8，16，64，256，1,024，4,096，8,192，および 16,384 byte の 8 種類のファイルを 4,000 個ずつ用意し，評価を行った．最大のファイルサイズである 16,384 byte は SSL/TLS のレコードサイズの最大値である．

暗号アルゴリズムは AES128，暗号化利用モードは CBC，鍵長は 128 bit で評価を行った．

時間の計測には，ファイルの入出力処理を計測時間に含めず，暗号モジュールスレッドおよび CPU スレッドを生成する直前から，両スレッドが終了した直後までの時間を計測した．

計測は，それぞれ 5 回行い，それらの平均値を評価に用いる．

表 1 評価環境
Table 1 Evaluation environment.

CPU	AM3358 プロセッサ
メモリ	512MB DDR2
Linux カーネル	Linux 3.2.0
OpenSSL	OpenSSL 1.0.0e
OCF	ocf-linux-20120127
暗号方式	aes-128-cbc

4.3 計測結果

4.2 節で説明した提案システムによる暗号処理時間の計測結果を**表 2**，**表 3**，**図 9**，および**図 11** に示す．

表 2，図 9，および図 11 の計測結果より，ファイルのサイズが，約 1,500 byte から 2,048 byte の間で，CPU スレッドのみで暗号化した場合と提案システムの速度が入れ替わり，これ以下のファイルサイズでは CPU スレッドが最も高速に暗号化している．これは，ファイルサイズが小さい場合に，暗号化のオフロードによる高速化の効果が少なく，オーバヘッドの割合が大きいためである．このオーバヘッドは，暗号モジュールにオフロードする際のメモリ転送などの OCF を経由する処理などにより発生する．

ファイルサイズが 1,500 byte から 2,048 byte の間を超えると，提案システムが最も高速に暗号処理をする．本実験のファイルサイズで最も大きい 16,384 byte では，提案シ

表 2 方式ごとの暗号処理時間
Table 2 Comparison of processing time: Way of encryption.

ファイルサイズ (byte)	8	16	64	256	1,024	4,096	8,192	16,384
CPU スレッド (ms)	81.24	85.42	97.47	141.69	334.26	1162.63	2153.26	4715.85
暗号モジュールスレッド (ms)	347.59	421.69	566.04	594.39	740.57	1254.06	1829.71	2937.71
ラウンドロビン (ms)	215.49	257.93	341.25	375.61	518.4	1041.05	1592.8	3081.24
ポーリング (ms)	401.86	407.07	485.6	694.03	843.93	2073.09	2603.75	3348.39
提案システム (ms)	147.71	168.12	198.91	208.68	442.44	1022.77	1351.68	2231.81

表 3 CPU スレッドと暗号モジュールスレッドの暗号化ファイル数
Table 3 Number of encrypt files in each thread.

ファイルサイズ (byte)	8	16	64	256	1,024	4,096	8,192	16,384
CPU スレッド（注）	4000/0	4000/0	4000/0	4000/0	4000/0	4000/0	4000/0	4000/0
暗号モジュールスレッド（注）	0/4000	0/4000	0/4000	0/4000	0/4000	0/4000	0/4000	0/4000
ラウンドロビン（注）	2000/2000	2000/2000	2000/2000	2000/2000	2000/2000	2000/2000	2000/2000	2000/2000
ポーリング （注）	2483/1517	2557/1443	2627/1373	2347/1653	1936/2064	1521/2479	1225/2775	933/3067
提案システム （注）	3001/999	3087/913	3158/842	3460/540	2704/1296	1726/2274	1115/2885	1027/2973

注）*CPU スレッドの暗号処理ファイル数 / 暗号モジュールスレッドの暗号処理ファイル数*

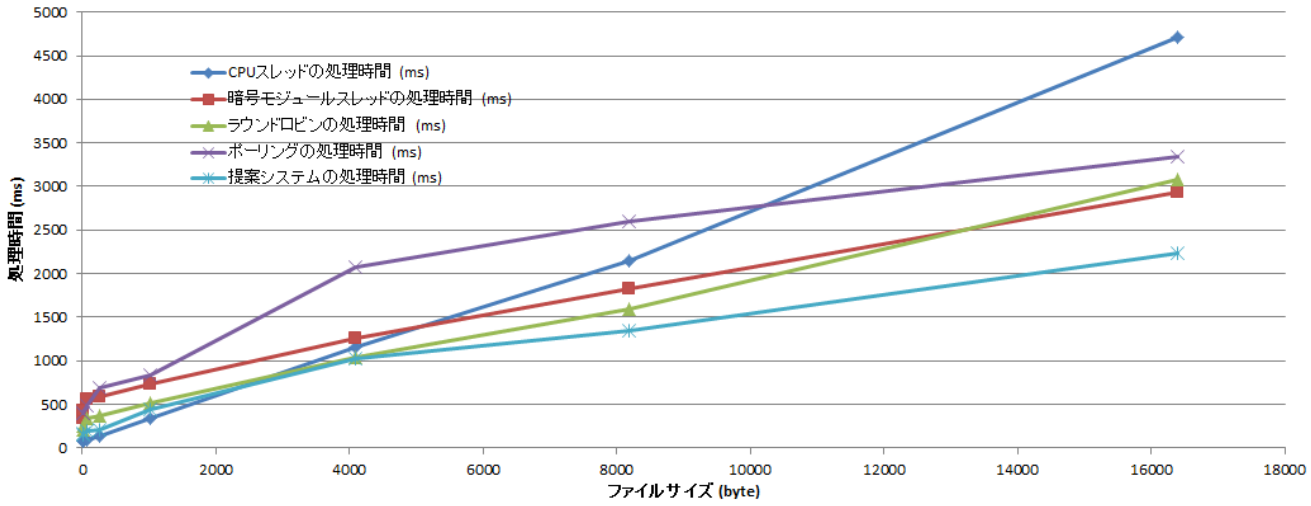

図 9 ファイル長ごとの暗号化処理時間
Fig. 9 Comparison of processing time: File size.

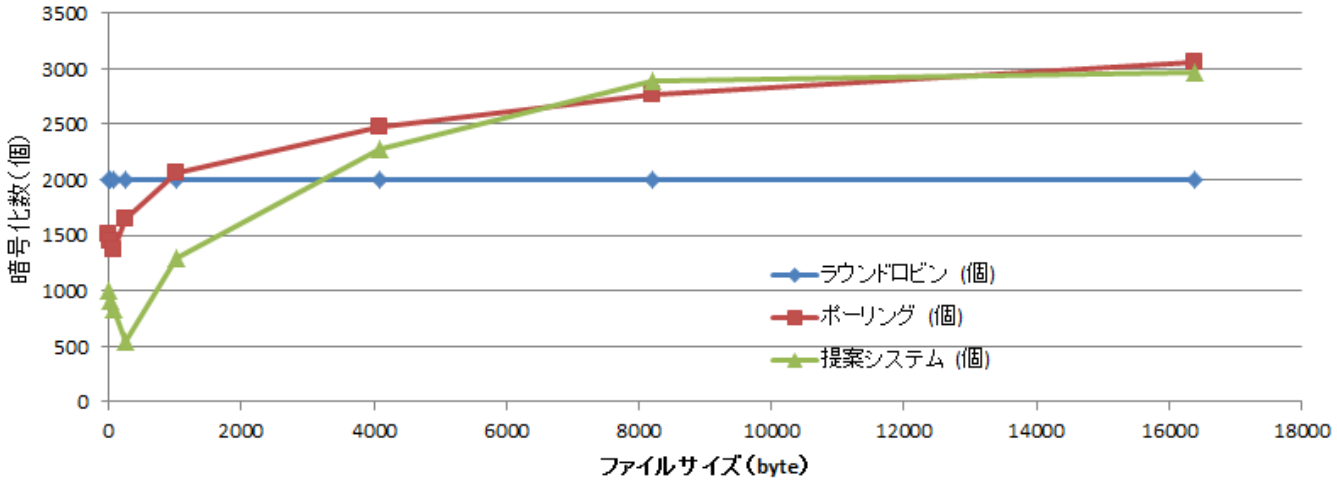

図 10 暗号モジュールスレッドでの暗号化数
Fig. 10 Number of encrypt files in crypto module thread.

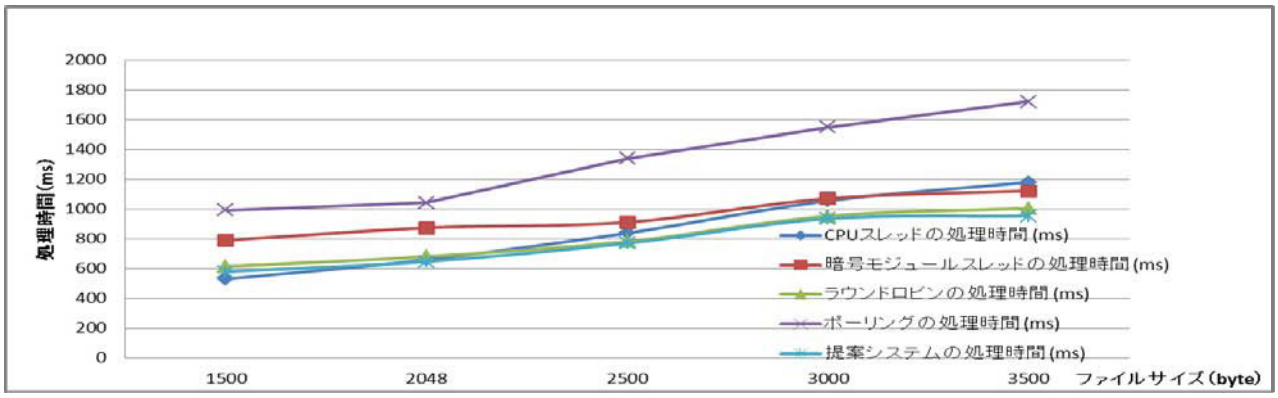

図 11 1,500 から 3,500 byte における暗号化処理時間
Fig. 11 Comparison of processing time: 1,500 byte to 3,500 byte.

ステムは，CPU スレッドのみの約 2.11 倍，暗号モジュールスレッドのみの約 1.32 倍，ラウンドロビンの約 1.38 倍，ポーリングの 1.5 倍の速度で暗号化をしている．

図 9 および表 2 より，ラウンドロビンは，約 13,000 byte 以下では，暗号モジュールスレッドのみの場合と提案システムの中間の性能で暗号化している．ファイルサイズが 1,024 byte から 4,096 byte の間で，提案システムにきわめて近い速度で暗号処理を行うことができる．しかし，この範囲よりファイルサイズが小さい場合や大きい場合には提案システムの方が高速に暗号化を行う．また，約 13,000 byte を超えるファイルサイズでは，ラウンドロビンは，暗号モジュールスレッドのみで暗号化した場合よりも処理速度が遅くなる．

図 9 および，表 2 より，ポーリングは，ファイルサイズがきわめて小さい場合を除いて，約 10,000 byte 以下では，暗号化の速度が低い．また，約 10,000 byte 以上のファイルサイズでは，ポーリングは，CPU のみで暗号化を行った場合より暗号処理速度が速かった．

表 3 および**図 10** に各評価プログラムで，CPU スレッドと暗号モジュールスレッドのそれぞれでファイルを暗号化した平文数を示す．CPU スレッドのみ，暗号モジュールスレッドのみで暗号化を行った場合は，4,000 個すべてのファイルがそれぞれ一方で暗号化されている．ラウンドロビンは，本実験では，CPU スレッドと暗号モジュールスレッドに 1 : 1 の割合で 2,000 個ずつのファイルの暗号化を行っている．ポーリングと提案システムは，ファイルサイズが小さい場合は，CPU スレッドで暗号処理をした方の効率が良いので，暗号化ファイル数が多く，ファイルサイズが大きくなると，暗号モジュールスレッドの暗号化ファイルの処理数が多くなっている．

4.4 考察

提案システムは，ファイルサイズが 1,500 byte から 2,048 byte の間を超えた場合，最も高速に暗号化を行っていた．ここでは，適用先ごとの暗号処理の効果，および，スレッドごとの暗号処理の配分についての観点で提案システムの有効性を考察する．

4.4.1 暗号化するファイルサイズ

提案システムをファイルの暗号化や SSL/TLS 通信にともなう暗号処理に適用することを考える．文献 [14] によると，Web サイトで通信するデータの平均サイズは約 1,600 kbyte 前後である．また，SSL/TLS では，レコード単位で暗号処理を行い，その最大サイズは 2^{14} byte である．よって，SSL/TLS 通信における暗号処理は提案システムが有効である 1,500 byte から 2,048 byte の間より大きいサイズで行われることも多く，提案システムの効果が高いと考えられる．

次に，ファイルの暗号化の応用例を考える．Ubuntu は現在広く普及している Linux ディストリビューションの 1 つであり，また，スマートフォンやタブレット向けのものもリリースされている．そのため，Ubuntu をインストールし，それらのファイルをすべて暗号化する場合を想定した．Ubuntu 14.04.1 LTS を minimal インストールした際，それに含まれるファイルの平均サイズは，約 26,676 byte である．よって，1,500 byte から 2,048 byte の間より大きいため，提案システムの適用は，効果があると考えられる．

4.4.2 スレッドごとの暗号処理の配分

ポーリングを用いた場合は，約 10,000 byte 以下のファイルサイズでは最も処理速度が遅かった．これは，CPU スレッドと暗号モジュールスレッドの状況を監視するためのループが，CPU 負荷を高めてしまい，結果として全体の処理性能を下げているためである．表 3 および図 10 において，CPU スレッドで暗号処理されるファイルの比率は提案システムより少ないことからも裏付けられる．たとえば，1,024 byte のファイルサイズのときに，ポーリングの比率 1,936/2,064 は提案方式の比率 2,704/1,296 よりも低い．

ラウンドロビンは，13,000 byte 以下のファイルサイズでは，ポーリングと提案システムの中間の性能を示し，1,024 byte から 4,096 byte の間では，提案システムの性能と同等の性能を示した．

CPU スレッドと暗号モジュールスレッドの処理能力の比はファイルサイズによって異なる．このため，ラウンドロビンでは，ファイルサイズが大きい場合や小さい場合では，CPU スレッドと暗号モジュールスレッドのどちらかが振り分けられた処理を終えてしまい，もう一方の処理の終了を待つことになる．表 3 および図 10 のとおり，1,024 byte から 4,096 byte の間では，CPU スレッドと暗号モジュールスレッドの処理能力の比が 1 : 1 に近づく．このため，ラウンドロビンは，本実験では，1,024 byte から 4,096 byte の間では提案システムの性能に近づいた．提案システムでは，3 章で説明したとおり，各スレッドがキューにデータをとりに行くので，ポーリングのように常時 CPU スレッドと暗号モジュールスレッドを監視する必要がない．さらに，両スレッドに処理させるファイルを動的に決めることができるので，ファイル長に適応して最適な比率で暗号処理を行うことができる．たとえば，CPU に負荷があるときには，暗号モジュールの活用の割合が増える．以上により，提案システムは，つねに高い性能を示すと結論付ける．

4.4.3 メモリの使用量について

本論文では，キューのサイズを 4,000 とし，16,384 byte のファイルの暗号化を行うときには，64 Mbyte のメモリを消費した．本論文で利用した環境では，512 Mbyte のメモリを搭載しているため，12.5%に相当し非常に大きなものとなる．しかし，最近では数 GB 以上のメモリを搭載するモバイル端末もリリースされている．また，4,000 というキューのサイズは今回の実験で，4,000 個のファイルを用意し，ファイルの入出力は考慮せず暗号処理のみの時間を計測するため，すべてのファイルをメモリに格納する必要があるために利用した．そのため，一度にすべてのファイルをメモリに格納せずに，たとえばキューのサイズを半分の 2,000 とし，ファイルの入出力を行いながら，暗号処理を同時に実行することも可能である．たとえば，このようにキューのサイズを半分にすればメモリの消費量も半分となり，メモリ消費量が大きな問題にはならないと考えられる．

5. まとめ

本研究では，モバイル端末など計算資源に制約がある端末上で行われる暗号処理を，暗号モジュールと CPU の両方を効果的に使うことで，高速化を目指した．提案システムは，ラウンドロビンやポーリングを活用した処理の振り分けを行う場合よりも高速に暗号処理を行うことを実験的に示した．ファイルサイズが，1,500 byte から 2,048 byte の間以下では，CPU 単体で暗号処理を行った場合が最も高速であったが，1,500 byte から 2,048 byte の間以上では，提案システムが最も高速であった．提案システムの適用例に鑑みても，1,500 byte から 2,048 byte の間以上での高速化は十分に意義がある．このことから，比較的大きいデータを一度に扱うような処理では，提案システムが有効性を持つことを示した．

今後の課題として，CPU 使用率を考慮したうえで，CPU スレッドと暗号モジュールスレッドにキューのデータを割り振ることにより，暗号処理が他のプログラムに影響を及ぼすことを最小限に抑える仕組みの実装と評価がある．また，提案システムを実際に SSL/TLS やファイルの暗号化に適用することも課題である．

参考文献

[1] Intel® IXP425 Network Processor, available from ⟨http://download.intel.com/design/network/ProdBrf/27905105.pdf⟩ (accessed 2014-11).
[2] UltraSPARC Tx Processor, available from ⟨http://www.opensparc.net/pubs/preszo/09/hotChips_spracklen-final.pdf⟩ (accessed 2014-11).
[3] Hughes, J., Morton, G., Pechanec, J., Schuba, C., Spracklen, L. and Yenduri, B.: Transparent Multi-core Cryptographic Support on Niagara CMT Processors, *Proc. IWMSE09* (2009).
[4] 齋藤孝道，大釜正裕，羅　鏡栄，杉浦　寛：IXP425 における暗号処理の効率的なオフロード方式の実装と評価，情報処理学会論文誌，Vol.51, No.9, pp.1530–1541 (2010).
[5] 齋藤孝道，杉浦　寛：Cell/B.E. における暗号処理の効率的なオフロード方式の提案と実装，情報処理学会論文誌，Vol.53, No.2, pp.815–824 (2012).
[6] AM3358 Processor, available from ⟨http://www.ti.com/product/am3358⟩ (accessed 2014-11).
[7] Viega, J., Messier, M. and Chandra, P.（共著），齋藤孝道（監訳）：OpenSSL—暗号・PKI・SSL/TLS, 入手先 ⟨http://www.openssl.org/⟩（参照 2014-11）.
[8] OCF, available from ⟨http://ocf-linux.sourceforge.net/⟩ (accessed 2014-11).
[9] Keromytis, A.D., Wright, J.L. and de Raadt, T.: The Design of the OpenBSD Cryptographic Framework, *Proc. USENIX Annual Technical Conference* (June 2003).
[10] AM335x_Crypto_Performance, available from ⟨http://processors.wiki.ti.com/index.php/AM335x_Crypto_Performance⟩ (accessed 2014-11).
[11] OpenSSH, available from ⟨http://www.openssh.org/⟩ (accessed 2014-11).
[12] SSL, available from ⟨http://tools.ietf.org/html/rfc6101⟩ (accessed 2014-11).
[13] TLS, available from ⟨http://tools.ietf.org/html/rfc5246⟩ (accessed 2014-11).
[14] HTTP Archive, available from ⟨http://httparchive.org⟩ (accessed 2014-11).

付　　録

図 A·1，**図 A·2**，**表 A·1**，**表 A·2** では，AES192 および AES256 の実験結果を示した．AES192 と AES256 は，暗号アルゴリズムは AES，暗号化利用モードは CBC および鍵長はそれぞれ 192 bit と 256 bit である．この結果，AES192 および AES256 でも AES128 と同様にある一定以上のファイルサイズでは，提案システムが最も高速であるという結果が得られた．

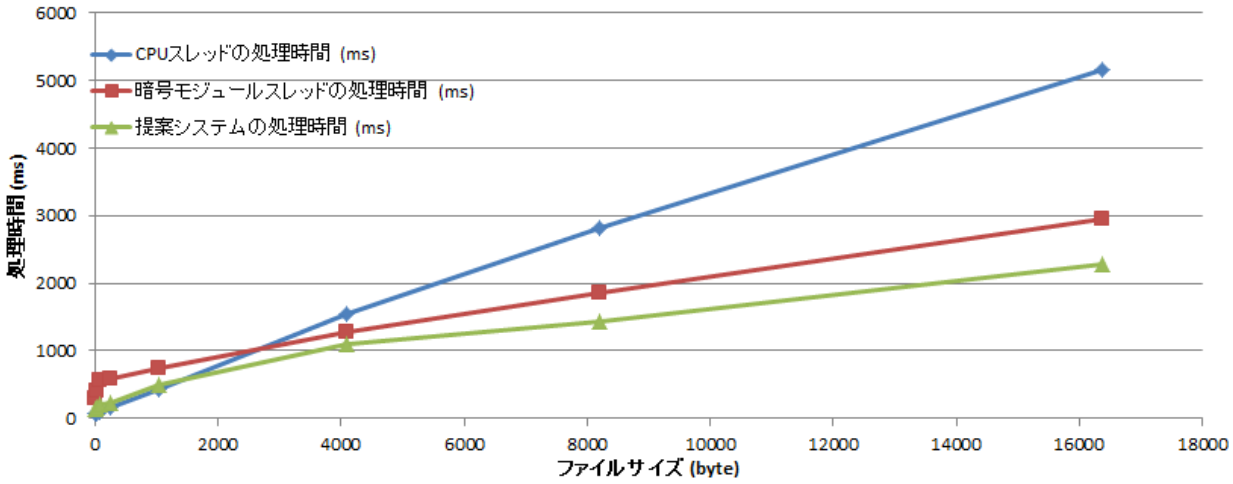

図 A·1 AES192 におけるファイル長ごとの暗号化処理時間
Fig. A·1 Comparison of methods in AES192.

表 A·1 AES192 におけるファイル長ごとの暗号化処理時間
Table A·1 Comparison of methods in AES192.

ファイルサイズ (byte)	8	16	64	256	1024	4096	8192	16384
CPU スレッド (ms)	78.95	82.18	99.79	164.64	422.36	1539	2813.54	5163.3
暗号モジュールスレッド (ms)	291.6	412.57	566.77	590.18	739.87	1278.41	1854.09	2954.13
提案システム (ms)	141.08	154.24	212.49	235.81	501.65	1094.15	1428.92	2274.2

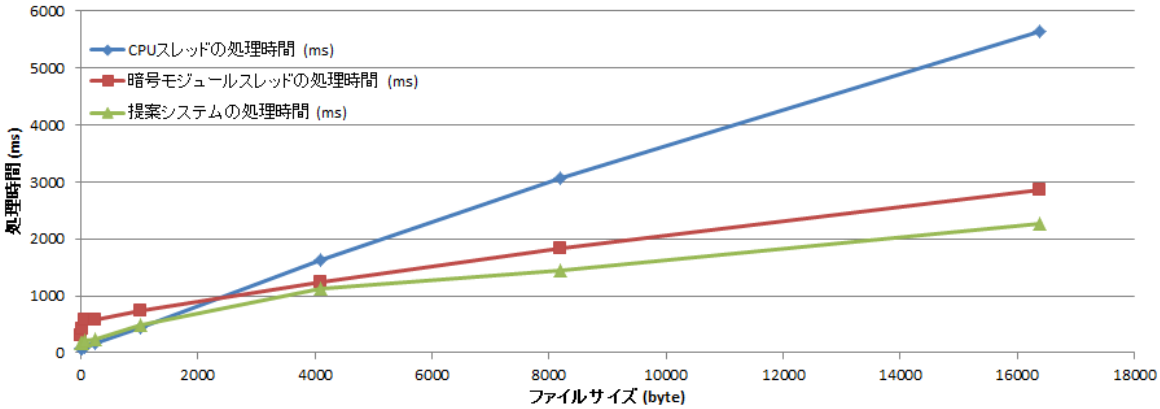

図 A·2 AES256 におけるファイル長ごとの暗号化処理時間
Fig. A·2 Comparison of methods in AES256.

表 A·2 AES256 におけるファイル長ごとの暗号化処理時間
Table A·2 Comparison of methods in AES256.

ファイルサイズ (byte)	8	16	64	256	1024	4096	8192	16384
CPU スレッド (ms)	83.44	84.41	102.08	172.97	452.39	1639.53	3066.38	5648.8
暗号モジュールスレッド (ms)	315.06	409.03	571.9	587.59	737.49	1237.73	1829.96	2861.9
提案システム (ms)	163.45	180.57	212.04	248.6	483.22	1127.04	1455.78	2274.02

復号処理についても同様に 8，16，64，256，1,024，4,096 byte のファイルサイズにおいて AES128 で計測を行った（図 **A·3** 参照）．その結果，ファイルサイズが小さい場合は，暗号化の場合と多少の差異はあるものの，全体としては各手法において順位が大きく変動することはない．また，暗号化と同様に，ある一定のファイルサイズ以上で，提案システムが最も高速に復号することができた．

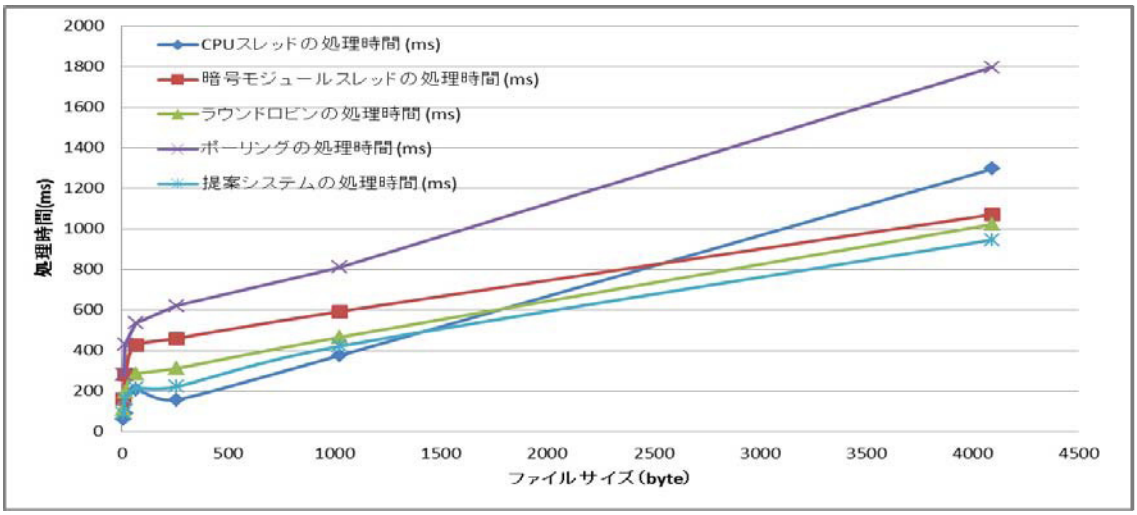

図 A·3 ファイル長ごとの復号処理時間
Fig. A·3 Comparison of methods in decryption.

金子 洋平 （正会員）

2013 年明治大学理工学部情報科学科卒業．2015 年同大学大学院理工学研究科博士前期課程修了．同年株式会社野村総合研究所入社．

齋藤 孝道 （正会員）

2005 年明治大学理工学部情報科学科助教授．現在，同学部同学科准教授．博士（工学）．

菊池 浩明 （フェロー）

1994 年東海大学工学部電気工学科助手．1995 年同専任講師．1999 年同助教授，2000 年同電子情報学部情報メディア学科助教授，2006 年同情報理工学部情報メディア学科教授．2008 年同情報通信学部通信ネットワーク工学科教授．1997 年カーネギーメロン大学計算機科学学部客員研究員．2013 年明治大学総合数理学部先端メディアサイエンス学科教授．