UltraSPARC Tx におけるメモリプールを用いた暗号処理のオフローディング方式の高速化

天野 桂輔† 渥美 裕太† 笠原 竜大† 村上 智祐† 齋藤 孝道‡

†明治大学大学院 ‡明治大学

あらまし 複数の暗号モジュールを搭載したプロセッサ上で，複数のアプリケーションプロセスから暗号モジュールへのオフローディングを行う際，メモリ確保及び解放の時間がオーバーヘッドとなる場合がある．そこで，本論文では，その解決策の一つとして，各プロセスのメモリ確保の回数を減らすために，UltraSPARC Tx 上で，メモリを事前に確保し，各プロセスがそのメモリを利用する方式を実装した．また，その評価を行った．

An Off-loading of Cryptographic Operation Using Memory Pool on UltraSPARC Tx

Keisuke Amano† Yuta Atsumi† Ryuta Kasahara† Tomosuke Murakami†
Takamichi Saito‡

†Meiji University ‡Graduate School of Meiji University

Abstract When a cryptographic module on UltraSPARC Tx is used by application program, the time of memory allocation could be an overhead. In the paper, we implement a memory allocator to decrease a frequency of memory allocation, and evaluate the performance.

1 はじめに

インターネットを利用する通信システムにおいて，個人情報など秘匿性の高い情報を盗聴や改ざんから保護するためにSSL（Secure Socket Layer）/TLS（Transport Layer Security）やIPSec [1]などのセキュリティプロトコルは欠かせないものとなっている．SSL/TLSやIPsecを利用した通信では，暗号化・復号処理をする必要がある．しかし，これらの処理は汎用的な処理を行うプロセッサコアにとって負担が大きく，数多くの暗号化・復号処理を同時に行うWebサーバなどでは，大きな負担となり，通信のボトルネックとなってしまう場合がある．

そのような背景の中，暗号化・復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[2][3][4]．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増えてきた[5][6].

本論文では，複数の暗号モジュールを搭載したプロセッサとして，Oracle社のUltraSPARC T2を利用して，暗号化・復号処理のオフロード技術について検討する．UltraSPARC T2は8個のコアを搭載したCPUであり，それぞれのコアに，共通鍵暗号方式や公開鍵暗号方式，ハッシュ処理などをサポートする暗号モジュール（後述）を搭載している．また，UltraSPARC T2用のOSであるSolarisでは，暗号モジュールを利用するためのフレームワークとして，PKCS#11ライブラリ（後述）が提供されている．しかし，PKCS#11を利用する場合，プログラマは直接暗号処理の制御ができないため，暗号モジュールの利用は，PKCS#11に依存することになる．

そこで，本論文では，UltraSPARC Txの暗号処理の制御を直接行うために，Solarisに単純に移植したOCF（OpenBSD/FreeBSD Cryptographic Framework）[7]及びOCFから暗号モジュールを利用するためのカーネルモジュール（以降，実装モジュールと呼ぶ）[8]を利用し，暗号処理を効率化する．しかし，単純に移植したOCFでは，複数のプロセスによるメモリ確保と解放がCPUのオーバーヘッドとなっていると推測され，性能を引き出していない．そこで，本論文では，新たにメモリプールを利用することで，各プロセスによるメモリ確保の回数を減らし，暗号処理の高速化を試みた．この実装の評価として，実装システムでの暗号処理に加えて，PKCS#11での暗号処理のパフォーマンスを計測した．

2　UltraSPARC T2 のアーキテクチャ

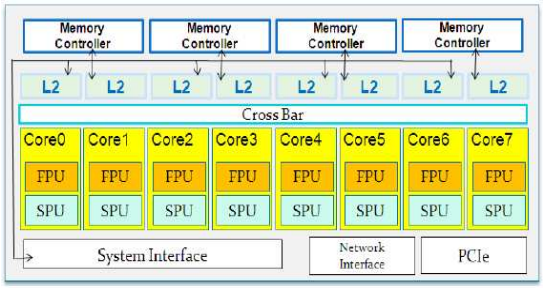

図1：UltraSPARC T2プロセッサ

UltraSPARC T2プロセッサのアーキテクチャを図1に示す．

UltraSPARC T2プロセッサはOracle社のチップ・マルチスレッディング・テクノロジー（CMT: Chip Multithreading Technology）に基づくマルチコアプロセッサであり，1つのプロセッサに8つのコアが搭載されている．それぞれのコアは，論理的に8つのCPUとして動作するため，最大64個のスレッドを同時に実行可能である．また，各コアに浮動小数演算ユニットであるFPU(Floating point/Graphics Unit)と暗号処理ユニットであるSPU(Stream ProcessingUnit)を搭載している．これらはメインメモリを共有しており，各コア，SPU及びFPUは並列に動作できる．

UltraSPARC T2のMemory Controllerはオンチップ化により，メインメモリへのアクセス・レイテンシを低減させている．合計4MBのL2キャッシュが8つのコアに均等に分割され，Cross Bar経由で，それぞれのコアとデータ転送を行っている．

暗号処理ユニットSPUは主にMAU（Modular Arithmetic Unit)と暗号/ハッシュ・ユニットから構成される．MAUはFPUを利用し，公開鍵暗号方式RSA，及び楕円曲線暗号を処理する．暗号/ハッシュ・ユニットは共通鍵暗号方式やハッシュ関数に対応しており，DES，3DES，AES，RC4，SHA1，SHA256とMD5を利用できる．

3　PKCS#11

PKCS#11では，アプリケーションからのインターフェースとなるライブラリとしてlibpkcs11.so [9]を用意しており，これを用いて，暗号モジュールへ処理をオフロードすることができる．また，暗号モジュールを利用するための仕組みとして，暗号モジュールの利用状況を監視し，暗号処理の実行対象を決定するスケジューラ／ロードバランサと，暗号モジュールを制御するためのデバイスドライバである暗号化プロバイダを備えている（図2）．

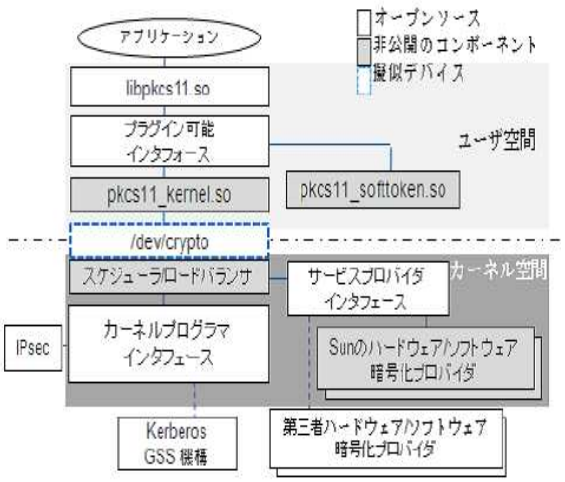

図2：PKCS#11暗号フレームワーク

4　OCF

4.1　OCF の概要

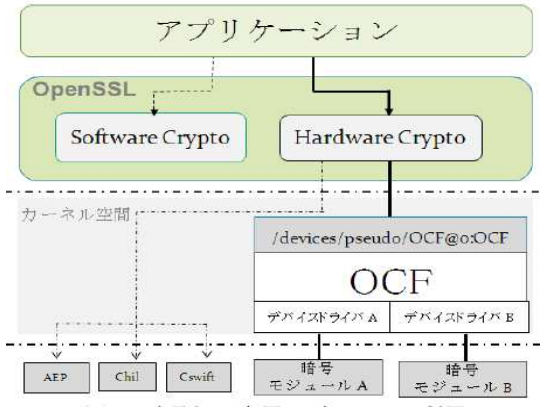

図3：暗号処理専用モジュールの利用

OCF(図 3)とは，OpenSSL [10]，OpenSSH 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ス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ー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ため，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．

4.2　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．

4.3　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/devices/pseudo/OCF@0:OCF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．ここでセッションとは，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．

5　OpenSSL

OpenSSL（図3）は，SSLやTLS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容易なインター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．特に，共通鍵暗号化方式とハッシュ関数については，共通のインターフェースでの利用を可能とするEVP APIを提供している．

また，OpenSSLは，AEP，ChilやCswift をはじめとした様々なハードウェアアクセラレータに対応しており，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（図4）．

5.1　OpenSSL からの 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4に示す所定の手続きを行う必要がある．

```
1. ENGINE *e;
2. ENGINE_load_cryptodev();
3. if( !(e=ENGINE_by_id("cryptodev")))
4.   /*エラー処理*/
5. else if( !ENGINE_set_default( e,ENGINE_METHOD_ALL) )
6.   /*エラー処理*/
```

図4：OpenSSLからのオフロードの手続き

図4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．3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．5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．

これらを利用することで，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に暗号処理を実行することができる．

5.2　OpenSSL 暗号処理の受け渡し

図5に，OpenSSLからEVP APIを用いてOCFに暗号処理を受け渡す一連の処理を示す．図5のEVP_ Encryptinit関数，EVP_ Ecrypt・Update/Final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．

```
7.  EVP_CIPHER_CTX ctx;
8.  EVP_EncryptInit( &ctx, EVP_des_cbc(), key,  iv );
9.
10. EVP_EncryptUpdate( &ctx, output, &outlen, input, len);
11. EVP_EncryptFinal( &ctx, outbuf+outlen, &tmplen);
12.
13. EVP_CIPHER_CTX_cleanup( &ctx );
```

図5：ENGINEを利用した暗号処理

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に受け渡され，OCFとのセッションを生成される．次に，EVP_ EncryptUpdate関数とEVP_ EncryptFinal関数が呼び出され，暗号処理が行われる．この関数内でCIOCCRYPT制御リクエストが発行され，OCFに暗号処理を受け渡される．暗号処理が終わると，EVP_ CIPHER_ CTX_ Cleanup関数を呼び出される．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．

6　提案システム

オリジナルのOCFではセッション生成時，暗号処理を行うアプリケーションプロセス（以降，暗号プ

ロセス)から，OpenSSL経由で，送られてくる平文データをOCFに受け渡すために，カーネル空間上で一時的に平文を確保するためのメモリ領域(以降，平文用メモリ領域と呼ぶ)を毎回用意する．これがオーバーヘッドとなる可能性が高い．そこで，本論文では，平文用メモリ領域を事前に確保し，セッション生成時には，それらを利用することでメモリ確保の回数を減らす．しかし，平文用メモリ領域が利用出来なかった場合は，新しくメモリ領域を確保する．

6.1 提案システムの概要

提案システムでは，実装モジュールがカーネル空間に組み込まれる際に，1Mbytes の平文用メモリ領域を 64 個確保し，プールする．プールされた平文用メモリ領域に対し，以下のような 2 つのメンバーを持つ構造体をそれぞれ用意し，それらを用いてメモリ領域へのアクセスを管理する．

```
struct buffer {
  unsigned char *buf;
  int status;
};
```

各メンバーの役割は，以下の通りである．

- buf:平文用メモリ領域の先頭アドレスを指すポインタ
- status:平文用メモリ領域が利用可能か否かを判定する変数．変数の値は平文用メモリ領域が利用可能な場合は -1，利用不可能な場合は，当該プロセスのセッションIDが格納されている．

提案システムでは，64個の平文用メモリ領域を均等に利用するため，各暗号プロセスが保持するセッションIDを64で割った余りを用いて，平文用メモリ領域を割り当てる．しかし，セッションIDを64で割った余りの値が他の暗号プロセスのそれと同一のものとなる場合がある．

そこで，平文用メモリ領域が割り当てに先立ち，その平文用メモリ領域の構造体のstatusを確認する．ただし，statusの確認の際には，ロックを用意し，複数のプロセスによる同時アクセスを制限する．statusの値が -1であれば，当該平文用メモリ領域は，利用可能な状態であるので，暗号プロセスはstatusの値に自らのセッションIDを格納し，bufで示された領域を利用する．statusの値が -1でなければ，その平文用メモリ領域は他の暗号プロセスによって利用されているため，statusの値を操作せずに，新しくメモリ領域を確保し，そのメモリ領域を利用する．

bufで示された領域を利用したプロセスは，暗号処理の終了後，平文用メモリ領域の構造体のstatusの値を -1に戻し，他の暗号プロセスが再利用できるようにする．新しくメモリ領域を確保した場合，そのメモリ領域を開放する．

6.2 提案システムの動作例

ここでは，ユーザ空間で実行されるOpenSSLの暗号処理を，OCFと実装モジュールを介して，SPUへオフロードする例を示す．以下の説明は，図8中の番号と対応している．

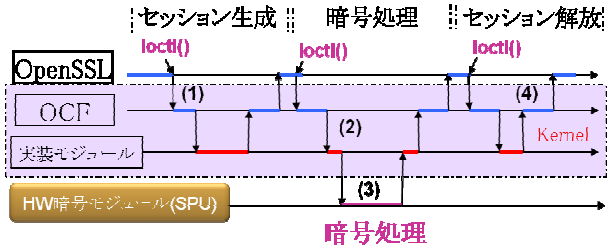

図8:処理の詳細

(1) OpenSSLとOCF間でのセッション確立時に，各暗号プロセスに平文用メモリ領域を割り当てる．statusの値が -1でない場合，新たにメモリを確保する．

(2) OpenSSLがOCFにセッションID，平文とIVを受け渡す．さらに，実装モジュールがセッションIDに対応した暗号鍵，OCFから受け取った平文及びIVをSPUで使用できる形式に変換する．

(3) SPUが暗号処理を行う．ここで，暗号処理が終わると，ハードウェア割り込みが発生し，結果をOCF経由でOpenSSLへ返す．

(4) 暗号処理の終了を指示し，利用した平文用メモリ領域の構造体の初期化(status = -1)を行う．新たにメモリ確保を行った場合，そのメモリを開放する．

7　評価

7.1　評価環境

提案システムの評価のために，表1に示す環境を用意した．この環境で，単純に移植したOCF（以降，単純移植版OCFと呼ぶ）を用いた場合とPKCS#11ライブラリを用いた場合に加えて，本論文で実装したOCF（以降，提案OCFと呼ぶ）の処理時間を計測し，比較した．

表1：評価環境

CPU	1.2GHz Ultra SPARC T2（コア）	メモリ	16GB
カーネル	Solaris kernel build 117 [11]	その他	OCF-20041201 OpenSSL-0.9.8a
OS	Solaris Express		

7.2　評価項目

性能評価として，EVP APIを使用し，AESのCBCモードを行うオリジナルの計測用コード（以降，計測用コードと呼ぶ）を用いた．計測用コードは，fork()関数により複数の暗号プロセスを生成し，それぞれの暗号プロセスで10Kbytesのデータを暗号化処理している．計測方法は，gettimeofday()関数を使用し，暗号プロセスの生成から暗号プロセスの終了までの処理時間を求めた．

8　計測結果

図9に，計測用コードによる計測結果について示す．図9の計測結果より，実装モジュールは，単純移植版OCFを利用したものより高速化できた．

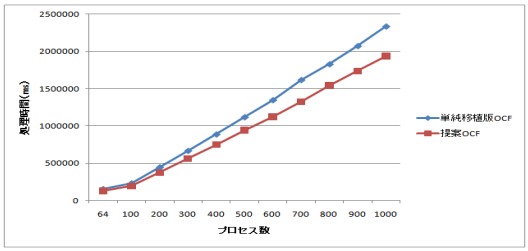

図9：暗号処理時間

処理時間の差分は，プロセス数に比例して大きくなっており，最終的にプロセス数が1,000個の時には，単純移植版OCFと比べ8割程度の処理時間で暗号化に成功している．また，PKCS#11に関しては，プロセス数が1,000個の時には，26,375,799msの処理時間を費やし，提案OCFの10倍以上の時間が掛かることがわかった．

9　まとめ

本論文では，メモリ確保及び解放に伴うオーバーヘッドを減らすため，メモリプールを利用したオフローディング方式を導入した．その評価を行った結果，単純移植版OCFより高速化することができた．

参考文献

[1] IPSec, http://www.ietf.org/rfc/rfc1825.txt

[2] Intel® Xeon® Processor 5600 Series, http://download.intel.com/jp/business/japan/pdf/323501-003JA.pdf

[3] Intel® IXP425 Network Processor, http://download.intel.com/design/network/ProdBrf/27905105.pdf

[4] Oracle SPARC Enterprise T5120 サーバ, http://www.oracle.com/jp/products/servers-storage/servers/sparc-enterprise/t-series/035999.pdf

[5] Hughes ,J., Morton, G., Pechanec, J., Schuba, C., Spracklen, L. and Yenduri, B.: Transparent Multi-core Cryptographic Support on Niagara CMT Processors, Sun Microsystems, Inc. 10 Network Circle Menlo Park, CA 95025 - USA

[6] 齋藤，大釜，羅，杉浦，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（2010），pp1530-1541.

[7] OCF, http://ocf-linux.sourceforge.net/

[8] 羅，大釜，杉浦，齋藤，UltraSPARC T2 における暗号モジュールの利用と評価，暗号と情報セキュリティシンポジウム（2010）

[9] libpkcs11.so, http://docs.sun.com/

[10] John Viega・Matt Messier・Pravir Chandra 共著，齋藤孝道 監訳，OpenSSL -暗号・PKI・SSL/TLS, http://www.openssl.org/

[11] Solaris kernel build 117 http://dlc.sun.com/osol/on/downloads/b117/