社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－ARC－170 (4)
2006／11／28

プログラム保護を行うプロセッサの公開鍵暗号ハードウェアの評価

酒井智也† 田端猛一† 北村俊明††

近年，プログラムを暗号化し，復号をプロセッサチップ内で行うことで，プログラムをリバースエンジニアリングから保護する機能を持ったセキュアプロセッサが各種研究されている．その中で，我々は暗号化されたプログラムとそれを復号する鍵との対応付けに，仮想記憶の枠組みを利用したプログラム保護システムを提案している．これまでの評価では，提案システムにおける公開鍵暗号ハードウェアの評価が含まれていなかった．そこで本稿では，RSA 暗号を用いた公開鍵暗号ハードウェアを設計し，提案システムをプロセッサチップに搭載した際の評価を行なった．その結果，提案システムを搭載する前と比べて，約 29%の面積増加でおさまることがわかった．

Evaluation of Public Key Cryptosystem Hardware of Processor that has Program Protection Feature

TOMOYA SAKAI,† TAKEKAZU TABATA† and TOSHIAKI KITAMURA††

Recently, the secure processor has been researched actively. It protects the program from reverse engineering by encrypting programs and decrypting it inside the processor. We propose the program protection system which has the mechanism of correspondence between the program and the decryption key using virtual memory system. In the former evaluation, the public key cryptosystem hardware was not included. Therefore, we designed RSA decryption unit and evaluated the proposal program protection system including RSA decryption unit. As a result, the amount of all area increased by about 29% compared with a processor without the proposed protection system.

1. はじめに

近年，PCだけでなく家電製品や通信機器などにプロセッサが搭載されるようになり，そこで処理されるプログラムの不正利用が問題となっている．たとえば，プリンタの発色管理や，カメラのレンズ制御を行なうようなプログラムはシステム開発者にとって，保護すべき重要なアルゴリズムが含まれている場合があるが，リバースエンジニアリングによってそのようなプログラムが解析され，不正に利用される危険性がある．

悪意のあるユーザーがプログラムの不正な入手をしようとした場合，システム上のプログラムへの攻撃方法としては，次のようなものが考えられる．

- 二次記憶に存在する実行プログラムに対してアクセスする方法
- OSの特権を利用して，通常のユーザーが知り得ない情報を取得する方法(OSを改変できる場合)
- メモリバスなどの命令が転送されるバスを観測するハードウェア的な攻撃方法

などである．これらの対策として，プロセッサ外部ではプログラムを暗号化しておき，プロセッサ内部でプログラムの復号を行うことで，プロセッサチップ内にのみ復号された命令が存在し，命令が転送されるバスをロジックアナライザなどでハードウェア的に観測するような攻撃からもプログラムを保護する機能を有したプロセッサであるセキュアプロセッサが提案されている[1)~3)]．

セキュアプロセッサでは，処理されるプログラムはそれぞれ異なるベンダによって開発される可能性が高く（マルチベンダ環境），各ベンダで異なる鍵を用いて暗号化できるという自由度を持つことが望ましい．そのため，公開鍵暗号と共通鍵暗号を組み合わせたハイブリッド方式を採用している．そこで，セキュアプロセッサでは，処理するプログラムに対応した鍵を用意する機構が必要となるが，我々は，暗号化されたプログラムとそれを復号する鍵との対応付けに，仮想記憶の枠組を利用したプログラム保護システムを提案している[4)]．提案システムでは，プログラムはページ単位で復号する鍵と対応付けられるため，1つのプロセスの中に，異なる鍵で暗号化されたプログラムを存在

† 広島市立大学大学院情報科学研究科
Graduate School of Information Sciences, Hiroshima City University
†† 広島市立大学情報科学部
Faculty of Information Sciences, Hiroshima City University

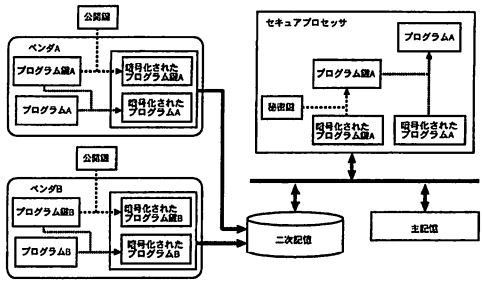

図 1　プログラムの暗号化と復号の概略図

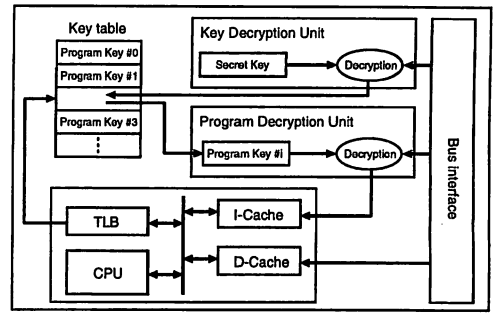

図 2　提案プロセッサのブロック図

させることができる．

これまでの評価では，提案システムにおける公開鍵暗号ハードウェアの評価が行なわれていなかった．そこで本稿では，RSA 暗号を用いた公開鍵暗号ハードウェアを設計し，シンプルなプロセッサチップに提案システムを実装した場合の評価を行なう．

以下本稿では，2 章でセキュアプロセッサについて述べ，3 章では，本研究で提案するプロセッサを用いたプログラム保護システムについて説明する．4 章では，提案システムで用いる共通鍵復号回路について述べ，5 章では，提案システムで用いる公開鍵復号回路について説明する．6 章では，提案システムをプロセッサチップに搭載した際の影響について説明し，7 章では，まとめを述べる．

2. セキュアプロセッサ

セキュアプロセッサを用いたシステムでは，プログラム開発者が不特定多数のマルチベンダ環境においても，それぞれのプログラムを保護できなければならない．そのような環境を提供するために公開鍵暗号と共通鍵暗号を組み合わせたハイブリッド方式が採用されている．これはプログラムの暗号化/復号に共通鍵暗号を用い，共通鍵の暗号化/復号に公開鍵暗号を用いるものである．

セキュアプロセッサにおいてのプログラムの暗号化と復号の概略を図 1 にしたがって説明する．セキュアプロセッサ側は公開鍵暗号で用いる鍵のペアを作成しておき，2 つの鍵のうち 1 つは共通鍵を暗号化するための鍵としてプログラム開発者やベンダに公開する．もう 1 つの鍵は，復号のための鍵としてプロセッサ内に秘密に保持する．ベンダ A は作成したプログラム A を共通鍵（以下，プログラム鍵）A によって暗号化する．そして，セキュアプロセッサの公開鍵でプログラム鍵 A を暗号化し，プログラムとともに配布する．セキュアプロセッサ側では暗号化されたプログラム鍵 A を実行プログラムが起動される際にプロセッサ内部の秘密鍵を用いて復号する．暗号化されたプログラムの復号は，復号したプログラム鍵 A を用いてキャッシュミス時に逐次実行される．

以上のようにプログラム開発者は自身のプログラム鍵によりプログラムを暗号化できるという自由度を持つことができ，プログラム鍵を公開鍵暗号で暗号化/復号することでプログラム開発者から安全にプロセッサ内部にプログラム鍵を渡すことができる．

公開鍵暗号のみでプログラムを暗号化/復号することも可能であるが，公開鍵暗号は共通鍵暗号に比べて非常に計算量が多いため，多くのセキュアプロセッサでは，ハイブリッド方式を採用している．

3. 提案システムの概要

この章では我々が提案しているプロセッサのプログラム保護システムについて簡単に説明する．既存のプロセッサに以下の機構を追加することでプログラム保護を実現する．

(1)　プログラム鍵管理機構
(2)　共通鍵暗号，公開鍵暗号復号機構

本研究のプロセッサのブロック図を図 2 に示す．

3.1 プログラム鍵管理機構

セキュアプロセッサでは，システム上に存在する実行プログラムはそれぞれ異なるプログラム鍵で暗号化されているため，それぞれのプログラム鍵を格納するレジスタスタック（以下，鍵テーブル）と，復号するプログラムがどのプログラム鍵に対応しているかを管理する機構が必要である．通常，プロセスごとに ID を割り当て，その ID とプログラム鍵を対応付ける方法が用いられるが、共有ライブラリなどの複数のプロセスから共有されるプログラムは，様々なプロセス ID で処理されるので暗号化することができない．提案プロセッサでは，仮想記憶を利用してページ単位で実行プログラムとプログラム鍵を対応付けることにより，1 つのプロセスに複数のプログラム鍵の対応付けができるため、共有ライブラリなどの暗号化も可能である．

つまり，暗号化されたプログラム鍵の復号を行う命令（以下，KEYDEC 命令）が発行され，OS が仮想アドレスと実アドレスの変換対をページテーブルに登

録するときに，実行プログラムを復号するプログラム鍵の鍵テーブル上のインデックスも登録しておく（インデックスはKEYDEC命令により指定）．また，プロセッサ内のTLBにも同じように登録する．キャッシュミス時にはこのインデックスにより，TLBによるアドレス変換の際にそのページの命令に対するプログラム鍵のインデックスを知ることができる．キャッシュヒット時は，すでに復号された命令がキャッシュ上にあるため復号処理は必要ない．

3.2 共通鍵暗号，公開鍵暗号復号機構

提案システムでは，暗号化されたプログラム鍵を復号する公開鍵暗号復号機構と，暗号化されたプログラムを復号する共通鍵復号機構が必要である．詳細は4章，5章で述べる．

4. 共通鍵暗号の復号回路について

セキュアプロセッサでは暗号化されたプログラムは共通鍵暗号の復号回路によって平文に復号される．本システムでは，プログラムを復号する共通鍵暗号に，NIST (National Institute of Standard and Tecnology) によって，標準暗号として選定されている，AES (Advanced Encryption Standard) を採用した[5]．

AES復号回路は先行研究によりすでに設計，評価がなされている．Synopsys社 Design CompilerでHITACHI 0.18 μmプロセス用に京都大学で作成されたスタンダードセルライブラリを用いて論理合成を行った結果を**表1**に示す．

ただし，通常レジスタファイルには6T-SRAMを使用するが，評価に使用したセルライブラリには6T-SRAMがない．そのためAES復号回路内のラウンド鍵を保存するレジスタファイルの面積については，6T-SRAMでの実現を仮定した値としている．6T-SRAMの回路面積は，同程度のトランジスタサイズを持つライブラリ中のインバータの回路面積を参考にし，1ビットあたり$48\mu m^2$とした．

5. 公開鍵暗号の復号回路について

セキュアプロセッサでは暗号化されたプログラム鍵は公開鍵暗号の復号回路によって平文に復号される．この復号処理はプログラム実行時にOSの発行するKEYDEC命令によって実行される．起動中の実行プログラムの数が鍵テーブルの数を越える場合は，再び復号を行なう場合があるが，ほとんどのプログラムでは，復号はプログラム実行時の1回でよい．また，復号処理時間はプログラムを二次記憶から主記憶にローディングしている間にこの復号処理を並行して行なうことができるため，ある程度隠蔽できると考えられる．このため，公開鍵暗号の復号回路は，回路規模の小面積化に重点を置いた設計にすることで影響を最小限にできる．

表1 AES復号回路の諸元

	面積 (mm^2)	クロック数 (cycle)	遅延時間 (ns)	鍵長 (bit)
AES-128	0.67	10	5.0	128

提案システムでは公開鍵暗号としてデファクトスタンダードであるRSA (Rivest-Shamir-Adleman) を採用した[6]．RSA暗号では$C = P^e modM$なる計算で平文を暗号化する．復号は$P = C^d modM$である．ここで，平文はP(Plaintext)，暗号文はC(Ciphertext)，法はM(Modulas)，暗号化べき指数はe，復号べき指数はdである．また，公開鍵は(e, M)で秘密鍵はdである．公開鍵の1つであるMのビット数nはセキュリティパラメータと呼ばれ，RSAの強度と関連があり，現在，安全性の観点から1024ビット以上が利用されることが多い．暗号化べき指数e(復号べき指数d)はnビットまでの値をとりうるため，非常に高い次数でのべき乗演算を必要とする．また，べき乗剰余演算は$X{\cdot}YmodM$の乗算剰余演算に分解できるが，乗算，除算を行うと非常に計算コストがかかる．よって，これらの演算を効率よく処理するアルゴリズムが必要である．提案システムでは復号回路のみを有するため以後の説明では復号を例に説明するが，暗号化についても同様に処理できる．

5.1 べき乗剰余演算のアルゴリズム

RSAの復号処理では，べき乗剰余演算が必要である．Mを法とするべき乗剰余演算は$C^d modM$を計算する演算である．

5.1.1 バイナリ法

C^dの計算において，指数dを$d = 2d_1 + d_1^{'}$($d_1^{'}$は0または1)と分解する．すると$C^d = (C^{d_1})^2 \cdot C^{d_1^{'}}$となり，そのまま$C$を$d$回乗算するよりも乗算回数は少なくなる．さらに，$C^{d_1}$を$d_1 = 2b_2 + d_2^{'}$と分解していくことで，乗算回数を削減できる．この性質を利用したものがバイナリ法である．**図3**にアルゴリズムを示す．

このアルゴリズムではdを2進表記した場合のビット数$i(i = \lfloor log_2 d \rfloor + 1)$回の2乗剰余と1が立っているビット数分の乗算剰余が必要である．Mのビット数を1024ビットとした場合，最悪の乗算剰余回数は2048回，平均1536回となる．

5.2 乗算剰余演算のアルゴリズム

バイナリ法を用いる場合，べき乗剰余演算は乗算剰余演算に分解される．Mを法とする乗算剰余演算は，$A{\cdot}BmodM$を計算する演算である．

5.2.1 モンゴメリ法

$A{\cdot}BmodM$においてMによる剰余を求める処理は除算を利用すると非常に処理時間がかかる．そこで，剰余を除算を用いることなく処理する計算手法としてモンゴメリ法が考案されている[7),8)]．

Input: C,d,M
Output: $P = C^d \bmod M$

STEP1: $P = 1$;
STEP2: $i = \lfloor log_2 d \rfloor + 1$;
STEP3: $P = P \cdot P mod M$; $\quad i = i - 1$;
STEP4: if $(d_i = 1)$ $P = P \cdot C mod M$;
STEP5: if $i \neq 0$ return STEP3;
else output P;

図 3 バイナリ法

モンゴメリ法は

$$AB + MN = WR \qquad (1)$$

という方程式において W を求める方法である．ここで，$R = 2^n$ であり，A，B は与えられた非負の整数，M は n ビットの法で，$R/2 < M < R$ という範囲にあるものとする．N，W は未知数である．このとき，M は奇数であり，R，M は互いに素となるから，式 1 は無限個の解をもつ．式 1 の両辺を M で mod をとると，

$$W \equiv ABR^{-1} (mod M)$$

となる．したがって，式 1 を満たすような n ビットの N を構成すれば，W が得られることになる．N は式 1 の両辺を R で mod をとると，

$$N \equiv -ABM^{-1} (mod 2^n)$$

となる．このことから，N を求めるのに必要な剰余計算は，$mod 2^n$ の形となる．$mod 2^n$ の形の剰余計算は単に下位 n ビットを取り出すだけでよいので簡単である． 上記のモンゴメリ法で求まる剰余の値 W は $ABmodM$ そのものではなく，$ABR^{-1}modM$ である．以下，$Mont(A, B, M) = ABR^{-1}modM$ とする．ここで，モンゴメリ形式として $A^* = ARmodM$ とすると

$$\begin{aligned} & Mont(A^*, B^*, M) \\ &= (AR)(BR)R^{-1} modM \\ &= ABRmodM \\ &= (AB)^* \end{aligned}$$

となる．つまり，モンゴメリ形式のデータ同士をモンゴメリ法で演算したとき，結果は，積のモンゴメリ形式になる．したがって，べき乗剰余演算を実行する際は，データをモンゴメリ形式に変換しておけば，その後の計算も，データ形式を修正することなく行なえる．ただし，最後に Z^* から Z に変換する必要がある．この変換は $Mont(Z^*, 1, M)$ で行なえる．また，A から A^* への変換は $Mont(A, R^2 modM, M)$ で行なえる．これらの変換は計算全体の最初と最後に行なうだけでよい．

M が n ビットの奇数で $R = 2^n$ であるとき，$Mont(A, B, M)$ は乗数 B の下位ビットから 1 ビットずつ逐次計算することで求めることができる．アルゴリズムを図 4 に示す．ここで，B は 2 進表現で $[b_{n-1} b_{n-2} ... b_0]$ としている．

Input: A,B,M
Output: $W = Mont(A, B, M)$
$= ABR^{-1} modM$

STEP1: $Q_0 = 0$;
STEP2: $for\ j = 0\ to\ n-1$;{
$R_j = Q_j + A \cdot b_j$;
if $(2 | R_j)$ $Q_{j+1} = R_j / 2$;
else $Q_{j+1} = (R_j + M)/2$;}
STEP3: if $(Q_{n-1} \geq M)$ $W = Q_{n-1} - M$;
else $W = Q_{n-1}$;

図 4 モンゴメリ法

5.2.2 基数 4 のモンゴメリ法

基数 4，つまり乗数の 2 ビットずつ計算を行なうモンゴメリのアルゴリズムが提案されている[9]．これによりイタレーションが半分になり，サイクル数の削減ができる．乗数の 2 ビットずつ計算を行なう場合，部分積 P_j は 2 次ブースのデコードを用いて生成する．これはシフトと正負反転で行なえる．また，法 $M = (m_{m-1}, ... m_1, m_0)$ の m_1 と累算結果 $Q_j + P_j$ の LSB の 2 ビット (t_{j1}, t_{j0}) により加算する法の値 (2M,M,-M, または 0) を決定する．**図 5** にアルゴリズムを示す．また，2 次ブースのデコード規則を**表 2** に示す．

5.3 RSA 復号回路の設計

先に述べた，バイナリ法，基数 4 のモンゴメリ法に基づき，RSA 復号回路を設計する．**図 6** に設計する RSA 復号回路のブロック図を示す．

5.3.1 回路構成

モンゴメリ法を用いる場合は，最初にデータ形式をモンゴメリ形式へ変換する必要がある．**図 3** における，P の初期値 1 と暗号文 C が変換対象である．暗号文 C を変換するには $Mont(C, R^2 modM, M)$ を計算すればよい．$R^2 modM$ と 1 を変換した値 $RmodM$ はあらかじめ計算しておく．変換した結果はそれぞれ REG_P，REG_C に格納する．

べき乗剰余回路では 1 サイクルごとに REG_D を上位から 1 ビットずつ読み込み，乗算剰余回路の被乗数を MAX1 により選択する．乗数は REG_P の値である．乗算剰余回路では，入力が決定すると，**表 2** のデコード規則にしたがって部分積生成回路 Booth decorder で部分積を生成し，REG1 に保持する．ここまでの処理を 1 サイクルで行なう．

次サイクルで，1024 ビットの加算を行う．ここで 1024 ビットの加算器を構成するのは現実的ではないため，より小規模の加算器を複数回使うようにする．本システムでは 256 ビットの加算器を 4 回使う構成とした．つまり，1024 ビットの加算を 4 サイクルか

Input: A,B,M
Output: $W = Mont(A, B, M)$
$= ABR^{-1} mod M$

STEP1: $Q_0 = 0$
STEP2: *for* $j = 0$ *to* $n/2 - 1${
$(t_{j1}, t_{j0}) = (Q_j + P_j)(mod4)$;
if $(t_{j0} = 0)${
if $(t_{j1} = 0)$
$Q_{j+1} = (Q_j + P_j)/4$;
else
$Q_{j+1} = (Q_j + P_j + 2M)/4$;}
else {
if $(t_{j1} = m_1)$
$Q_{j+1} = (Q_j + P_j - M)/4$;
else
$Q_{j+1} = (Q_j + P_j + M)/4$;}}
STEP3: if $(Q_{n/2-1} < 0)$ $W = Q_{n/2-1} + M$;
else $W = Q_{n/2-1}$;

図 5 基数 4 のモンゴメリ法

表 2 2 次ブースのデコード規則

b_{2j+1}	b_{2j}	b_{2j-1}	b_j	出力 P_j
0	0	0	0	0
0	0	1	1	1A
0	1	0	1	1A
0	1	1	2	2A
1	0	0	-2	-2A
1	0	1	-1	-1A
1	1	0	-1	-1A
1	1	1	0	0

けて計算する．

ADDER1 で 1 回目の 256 ビットの加算が終わると，次のサイクルで加算結果の LSB2 ビットと M の値から加算する法の倍数を決定し，ADDER2 で加算する．このとき，ADDER1 では 2 回目の 256 ビット加算を行なう．この方式で計算すると，最初に REG_Q に結果が保存されるのに 7 サイクルかかる．次から結果が保存されるのは，ADDER2 で 4 回目の加算をしているときに，ADDER1 では次の新しい部分積の 1 回目の計算を行なうことができるため，4 サイクルで行なえる．Booth decorder により乗数の 2 ビットずつ計算しているため，1024 ビット加算は 512 回行なわれる．このため，1 回の乗算剰余計算は 2051 サイクル (511*4+7) となる．結果が負の場合は，最後に法 M を加算する必要があるため (STEP3)，4 サイクル余分に必要である．バイナリ法の説明で述べたように，この乗算剰余計算は，最悪 2048 回，平均 1536 回行なわれる．

REG_D を最後まで読み込み，乗算剰余回路での計算が終わると REG_P に復号した結果が格納されるが，

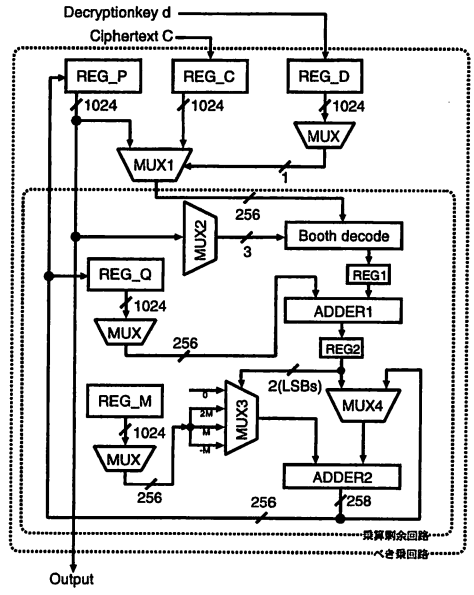

図 6 RSA 復号回路の構成

この値はモンゴメリ形式なので通常の形式に変換する必要がある．これは $Mont(P, 1, M)$ の計算を行なえばよい．

ここで，加算器は自動合成で得られる最適化された CLA（Carry Look ahead Adder）を用いている．

6. 評　　価

RSA 復号回路を Verilog-HDL で設計し，Synopsys 社 Design Compiler で HITACHI 0.18 μm プロセス用に京都大学で作成されたスタンダードセルライブラリを用いて論理合成を行った．論理合成時の遅延制約は 5.0 [ns](動作周波数 200 [MHz]) としている．

表 3 に合成結果を示す．ここで，秘密鍵/公開鍵レジスタはチップ製造時にヒューズ回路を設けて，データ書き込み，検査後に回路を切る方法で実装できると考えた．その場合，面積は無視できるほど小さいと考えられるので評価結果の面積には含めていない．

1 回の乗算剰余計算に必要なクロックサイクル数は最悪の場合，2055 サイクルであり，処理時間は 5.0ns × 2055 = 10μs となる．また，法 M のビット数が 1024 ビットであれば，すべてのビットが 1 である最悪の場合，形式変換も含め，2050 回の乗算剰余計算が必要となる．したがって，復号処理時間はおよそ 21ms である．平均的な場合，1 回の乗算剰余計算は 2051 サイクル，それが 1538 回行なわれるので，復号処理時間はおよそ 16ms となる．

表 3 RSA 復号回路の合成結果

面積 (mm^2)	クロック数 (cycle)	遅延時間 (ns)	復号処理時間 (ms)
1.75	4,212,750	5.0	21

表 4 Luehdorfia の構成

CPU コア	ARM version 5 互換命令セット 6 段パイプライン バレルシフタ 1 個 ALU1 個 32×32 乗算器 1 個 汎用レジスタ (32 ビット ×16 本) 特権レジスタ (32 ビット ×20 本) 動作周波数 200 MHz
命令キャッシュ	8KB ダイレクトマップ方式 ラインサイズ : 16B
データキャッシュ	8KB ダイレクトマップ方式 ラインサイズ : 16B
TLB	16 エントリ フルアソシアティブ方式 ページサイズ : 4KB

6.1 面積への影響

実際に本システムをプロセッサチップに組み込んだ場合，どの程度面積が増加するのかを評価する．本システムを組み込む対象は，本研究室で設計された ARMversion5 互換命令セットを持つシンプルなプロセッサコア (Luehdorfia) とする．Luehdorfia の構成を表 4 に示す．

Luehdorfia に本システムを組み込む場合，新たに追加が必要な回路は図 2 の RSA 復号回路，AES 復号回路，鍵テーブルである．本稿では，同時に実行されるプログラムはそれほど多くはないと考え，鍵テーブルのエントリ数は 16 としている．また，これに伴いページテーブルと TLB の各エントリにプログラム鍵のインデックスが 4 ビット追加されることになるが，ハードウェア量の増加はわずかである．表 5 にそれぞれの回路面積を示す．なお，CPU コア内のレジスタファイル，命令キャッシュ，データキャッシュと鍵テーブルの面積は 4 章で述べた 6T-SRAM での実現を仮定した値である．プロセッサチップに組み込んだ場合の全体の面積は $11.32mm^2$ であり，もとの約 29%の面積増加で抑えられる．

7. ま と め

本稿では，暗号化されたプログラムとそれを復号する鍵との対応付けに，仮想記憶の枠組を利用したプログラム保護システムにおける公開鍵暗号復号ハードウェアを設計し，システム全体をプロセッサチップに搭載する場合の影響について評価を行なった．

その結果，面積は約 29%の面積増加で抑えられることがわかった．この回路面積は LSI に十分実装可能な大きさである．現在，我々は提案プロセッサの FPGA

表 5 全体の回路面積

	面積 (mm^2)
CPU コア	1.04
命令キャッシュ (8KB)	3.74
データキャッシュ (8KB)	3.74
TLB	0.28
(Luehdorfia 合計)	8.80
AES 復号回路	0.67
RSA 復号回路	1.75
鍵テーブル	0.10
(追加回路合計)	2.52
合計	11.32

による実装を進めており，今後 FPGA 実装による検証，評価を行なう予定である．

謝辞 本研究は，東京大学大規模集積システム設計教育研究センターを通し，シノプシス株式会社ならびに株式会社日立製作所の協力で行われたものである．また本研究の一部は，文部科学省科学研究費補助金基盤研究 (C) 課題番号 17500044 の支援により行った．

参 考 文 献

1) D.Lie, C.Thekkath, M.Mitchell, P.Lincoln, D.Boneh, J.Mitchell and M. Horowitz: Architectural Support for Copy and Tamper Resistant Software, *ASPLOS-IX*, pp. 168–177 (2000).
2) G.E.Suh, D.Clarke, B.Gassend, M.van Dijk and S.Devadas: AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing, *ICS03* (2003).
3) 橋本幹生，春木洋美:敵対的な OS からソフトウェアを保護するプロセッサアーキテクチャ，Vol.45, No.SIG 3 (ACS 5), pp.1–10 (2004).
4) 城本正尋，田端猛一，酒井智也，島田貴史，窪田昌史，川端英之，北村俊明 : 公開鍵暗号を用いてプログラムの保護を行なうプロセッサの提案，Vol.47, No.SIG 18 (ACS 16), (2006 年 11 月発刊予定).
5) National Institute of Standard and Tecnology: Announcing the ADVANCED ENCRYPTION STANDARD (AES), FIPS PUB 197 (2001).
6) R.L.Rivest, A.Shamir and L.Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communication of the ACM*, Vol.21, No.2, pp.120–126 (1978).
7) P.L.Montgomery: Modular multiplication without trial division, *Math, Computation*, Vol.44, pp.519–521 (1985).
8) 神永正博，渡邊高志 : 情報セキュリティの理論と技術，森北出版，(2005).
9) J.Hong and C.Wu: Radix-4 Modular Multiplication and Exponentiation Algorithms for the RSA Public-Key Cryptosystem, *DAC 2000*, pp. 565–570 (2000).