理 工 学 部

2018年3月

卒 業 論 文

モンゴメリ法及び並列化を適用した耐サイドチャネル攻撃RSA復号回路の高位合成

27014556 大 窄 直 樹

(情 報 科 学 科)

卒業論文

モンゴメリ法及び並列化を適用した 耐サイドチャネル攻撃 RSA 復号回路の 高位合成

大窄 直樹

内容梗概

本研究では, モンゴメリ法及び並列化を適用した耐サイドチャネル攻撃 RSA 復号回路を高位合成により設計する手法を提案する. IoT (Internet of Things) 向け組み込み機器の通信やシステムのセキュリティを確保するためには暗号化機能が必要となるが, 暗号回路の漏洩情報を利用するサイドチャネル攻撃への対策もまた重要な課題となっている. 太田らは, Fournaris のアルゴリズムに基づき, サイドチャネル耐性を付加した RSA の復号回路を高位合成により設計したが, 回路の規模や実行時間の増加が課題となっている. そこで本研究では, モンゴメリ法と並列化の適用により耐サイドチャネル攻撃 RSA 復号回路の高速化を試みる. モンゴメリ法の適用により動作周波数が向上し, 並列化の適用によりサイクル数が減少した結果, 太田らの RSA 復号回路回路と比べて実行速度を約 2.4 倍に高速化することができた.

キーワード

高位合成, RSA 暗号, サイドチャネル攻撃, モンゴメリ法

目 次

第1章　序論

近年, 身の回りの様々なモノがインターネットに接続する IoT (Internet of Things) の登場によって, 世界中のあらゆるものが通信を行う社会が実現しようとしている. これらの IoT 向け組込み機器の通信やシステムのセキュリティを確保するために, AES [1], RSA [2] に代表される暗号化機能が必要となる.

暗号化及び復号の処理は複雑な計算を必要とするため, プロセッサで計算すると莫大な時間がかかり, 組み込み機器の厳しい時間制約を満たさなくなる. そこでソフトウェアの一部分をハードウェア化し計算能力を向上させることにより実行速度を向上させることが一つの選択肢となる.

ハードウェアの設計には時間がかかるため, 設計を効率的化する技術として高位合成 [3] がある. 高位合成は, C 言語などの動作記述を入力としてハードウェア記述言語を自動生成する技術である. また, 高位合成の一種にアセンブリや機械語を入力としてハードウェア記述言語を自動生成するバイナリ合成という技術がある. 伊藤らは, バイナリ合成を用いて RSA 暗号化/復号回路を設計している [4]. C 言語で記述した RSA 暗号化/復号 のプログラムに, 多倍長整数演算ライブラリ GMP (GNU Multi-Precision Library) をリンクしバイナリ合成システム ACAP (Assembly Compatible Architecture Prototyper) [5] を用いて RSA 暗号化/復号回路を設計している.

IoT 向け組み込み機器のもう一つの課題に, サイドチャネル攻撃への耐性の付加がある. サイドチャネル攻撃は, 暗号処理, 暗号化機能の実装上の脆弱性を利用した攻撃で, 暗号化方式の脆弱性をついた攻撃でなく暗号化装置の脆弱性をついた攻撃である. モジュール動作時に副次的に発生する熱や消費電力などの漏洩情報を利用する. オシロスコープやパソコンといった比較的安価な設備で実施でき, 攻撃の痕跡も残らないことから, 暗号モジュールに対する現実的な課題になっている.

そこで太田らは, 伊藤らの RSA 暗号化/復号回路にサイドチャネル耐性を付加している [6]. この回路は, RSA 復号回路にサイドチャネル攻撃への耐性を付加した Fournaris のアルゴリズム [7] に基づき, 復号部分にサイドチャネル攻撃への耐性を付加している. しかしサイドチャネル攻撃への耐性を付加したために, 実行時間と回路規模が大幅に増加している.

本研究では, モンゴメリ法の適用と並列化の適用により耐サイドチャネル攻撃 RSA 復号回路の高速化を試みる. モンゴメリ法の適用により動作周波数が向上し, 並列化の適用によりサイクル数が減少した結果, 太田らの RSA 復号回路回路と比べて実行速度が約 2.4 倍に高速化することができた [8].

以下, 第 2 章では, RSA 暗号に対するサイドチャネル攻撃, Fournaris の RSA 復号アルゴリズム, 高位合成システムについて, そして太田らの サイドチャネル耐性を付加した RSA 暗号化/復号回路について述べる. 第 3 章では, モンゴメリ法, モンゴメリの適用方法について述べる. 第 4 章では, 並列化, 並列化の適用方法について, 第 5 章では, 実験結果, 考察を述べ, 第 6 章では全体

の結論を述べる.

第2章　耐サイドチャネル攻撃RSA復号回路の高位合成

2.1　RSA 暗号に対するサイドチャネル攻撃

RSA [2] は公開鍵暗号の一種である. 公開鍵暗号では, 平文を暗号化する公開鍵と復号する秘密鍵が, 一組のペアになっている. 公開鍵は公開しているため誰でも平文を暗号化できるが秘密鍵は公開していないため鍵の保有者しか復号できない.

表 2.1: RSA 暗号に用いるデータ [9].

m	メッセージ
c	暗号文
e	公開鍵 (正整数)
p, q	任意の大きな素数
N	$p \cdot q$
d	秘密鍵 ($d = e^{-1} \pmod{(p - 1)(q - 1)}$)

RSA の暗号化/復号 に用いる鍵データを表 2.1 に示す. e と N は公開鍵, d は秘密鍵である. メッセージを m, 暗号文を c としたとき, 暗号化は次の演算により行える.

$$c = m^e \bmod N$$

また復号は次の演算により行える.

$$m = c^d \bmod N$$

現在, 安全を確保するためには鍵長は 2048 ビット以上が必要とされている.

サイドチャネル攻撃は, 暗号化方式の脆弱性をついた攻撃でなく暗号化装置の脆弱性をついた攻撃である. これはモジュール動作時に副次的に発生する漏洩情報を利用する. 漏洩情報として消費電力を利用する攻撃に, 単純電力解析攻撃と, 差分電力解析攻撃がある. 単純電力解析攻撃は, 測定した一つ以上の電力波形を直接解析する手法で, 差分電力解析攻撃は測定した大量の電力波形を統計的手法を用いて秘密鍵を推定する手法である. RSA の復号では, 秘密鍵中の 0 と 1 どちらの計算しているかによって消費電力が異なることを利用する.

その他の攻撃の一種に故障利用攻撃がある. 故障利用攻撃は, 高電圧を加えたり, 瞬間的にクロック周波数を上げたりすることにより, 他の機能に影響を与えない範囲の限定的な障害を与え攻撃者の望む異常な処理を行わせる攻撃である. 有名な RSA に対する故障利用攻撃として, Bellcore 攻

撃 [10], KQ 攻撃 [11] がある．Bellcore 攻撃は, 差分故障攻撃の一種で, 差分のあるメッセージをもう一組生成しそれらとの差分を計測して解析する手法である．KQ 攻撃は, 中国人剰余定理を用いた復号の後に故障を挿入する．

2.2 Fournaris の RSA 復号アルゴリズム

Fournaris のアルゴリズム [7] は, サイドチャネル攻撃に対する RSA 復号処理の脆弱性の解決を図ったものである．このアルゴリズムはハイディングとマスキングを伴うモンゴメリ冪剰余算, および故障挿入の検出に基づいている．ハイディングは, モジュールの消費電力と処理される値の依存関係を隠し, 値に関係なく消費電力を一定あるいはランダムにするものである．マスキングは, 暗号化/復号に用いる入力値にあらかじめ乱数による変換を施すものである．このとき処理する値に依存した消費電力となるが, その消費電力はマスキング前の真の中間値とは無関係であるため, 解析が不可能とされる [12]．故障挿入の検出は，値を返すとき検算により誤りが存在する場合, 出力を停止する．故障の挿入があった場合中間結果が変更するのでこれにより故障利用攻撃を防ぐことができる．このアルゴリズムは一般的な単純電力解析攻撃や差分電力解析攻撃の他, Bellcore 攻撃, KQ 攻撃, YLMH 攻撃 [13] 等の故障利用攻撃への耐性を持つ．

アルゴリズムを図 2.1 と図 2.2 に示す． 図 2.1 が復号アルゴリズムの本体であり，ここから図 2.2 の耐攻撃モンゴメリ冪剰余算を計算する FSCAME を呼び出している． 図 2.1 は, $c, b, b^{-1}, p, q, d_p, d_q, i_q, N$ を入力として $c^d \bmod N$ を出力する． また b はマスキング用の乱数であり, b^{-1} は N を法とする b の逆数である．$d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$ であり, $i_q = q^{-1} \bmod p$ である．N を法とする計算を高速化するため, $N = p \cdot q$ となる p と q をそれぞれ法として $s_0^p, s_1^p, s_2^p, s_4^p$ および $s_0^q, s_1^q, s_2^q, s_4^q$ を FSCAME で計算し, ここから中国人剰余定理によって S_0, S_1, S_2, S_4 を求めている．返り値である $(S_0 \cdot S_4 \bmod N)$ は, $c^d \bmod N$ と等しく, これは RSA の復号である．故障が挿入された場合には下線部が成立しないため, 故障の挿入を検出できる．このアルゴリズムでは, $c^d \bmod N$ を直接計算するのではなく, c に乱数のマスク b を乗じた $S_0 = (b \cdot c)^d \bmod N$ を計算し, 最後にこれに $S_4 = b^{-d} \bmod N$ を乗じて復号結果を返すようにしている．これ以外の S_1, S_2 の値も b を用いている．これによって, 特定の c の入力によって鍵が推定されるのを防いでいる．

図 2.2 は, 耐攻撃モンゴメリ冪剰余算である．c, b, b^{-1}, d, M が入力であり, (s_0, s_1, s_2, s_4) が出力である．また b はマスキング用の乱数であり, b^{-1} は N を法とする b の逆数である．d_i (d の i ビット目) が 0 でも 1 でも同じ演算が行うようにすることによりハイディングを行っている．また下線部で故障の挿入を検出している．R^{-1} を乗じているのは, 計算量を削減するためにモンゴメリリダクションを利用している．モンゴメリリダクションは第三章で説明する．

RSA 復号

Input: $c, b, b^{-1}, p, q, d_p, d_q, i_q = q^{-1} \bmod p, N$

Output: $c^d \bmod N$

$(s_0^p, s_1^p, s_2^p, s_4^p) = \mathrm{FSCAME}(c, b, b^{-1}, d_p, p);$

$(s_0^q, s_1^q, s_2^q, s_4^q) = \mathrm{FSCAME}(c, b, b^{-1}, d_q, q);$

$S_0 = s_0^q + q \cdot ((s_0^p - s_0^q) \cdot i_q \bmod p);$

$S_1 = s_1^q + q \cdot ((s_1^p - s_1^q) \cdot i_q \bmod p);$

$S_2 = s_2^q + q \cdot ((s_2^p - s_2^q) \cdot i_q \bmod p);$

$S_4 = s_4^q + q \cdot ((s_4^p - s_4^q) \cdot i_q \bmod p);$

if ($\underline{S_0 \cdot S_1 \bmod N = S_2}$ **and** p, q not modified)

{ **return** $(S_0 \cdot S_4 \bmod N)$; }

else { **return** error; }

図 2.1: Fournaris のアルゴリズム [7].

耐攻撃モンゴメリ冪剰余算

Function: FSCAME
Input: $c, b, b^{-1}, d = (1, d_{t-2}, ...d_0), M$
Output: (s_0, s_1, s_2, s_4)
$R = 2^{n+2}$; $T = R^2 \bmod M$;
$b_R = b \cdot R \bmod M$;
$b_{R-1} = b^{-1} \cdot R \bmod M$;
$s_0 = s_1 = b_R$;
$T_R = T \cdot c \cdot R^{-1} \bmod M$;
$s_2 = b_R \cdot T_R \cdot R^{-1} \bmod M$;
$s_3 = s_4 = s_5 = b_{R-1}$;
for $(i = 0$ to $t - 1)$ {
  if $(d_i = 1)$ {
    $s_0 = s_0 \cdot s_2 \cdot R^{-1} \bmod M$;
    $s_4 = s_4 \cdot s_3 \cdot R^{-1} \bmod M$;
  } **else** {
    $s_1 = s_1 \cdot s_2 \cdot R^{-1} \bmod M$;
    $s_5 = s_5 \cdot s_3 \cdot R^{-1} \bmod M$;
  }
  $s_2 = s_2^2 \cdot R^{-1} \bmod M$;
  $s_3 = s_3^2 \cdot R^{-1} \bmod M$;
}
$s_0 = s_0 \cdot b^{-1} \cdot R^{-1} \bmod M$;
$s_1 = s_1 \cdot c \cdot R^{-1} \bmod M$;
$s_2 = s_2 \cdot 1 \cdot R^{-1} \bmod M$;
$s_4 = s_4 \cdot b \cdot R^{-1} \bmod M$;
if (i and d are not modified **and** $s_0 \cdot s_1 \cdot R^{-1} \bmod M = s_2 \cdot 1 \cdot R^{-1} \bmod M$)
  { **return** (s_0, s_1, s_2, s_4); }
else { **return** error; }

図 2.2: 関数 FSCAME [7].

2.3 バイナリ合成システム ACAP

高位合成は, C 言語などの動作記述を入力としてハードウェア記述言語を自動生成する技術である．また高位合成の一種にアセンブリや機械語を入力としてハードウェア記述言語を自動生成するバイナリ合成という技術がある．

本研究ではバイナリ合成システム ACAP (Assembly Compatible Architecture Prototyper) [5]

を利用する．ACAP は, 石浦研究室で開発しているバイナリ合成システムであり, MIPS R3000 の機械語プログラムを入力として, レジスタ転送レベルのハードウェア記述を合成する．ハードウェア化することにより演算を並列に処理できるため処理が高速化される．ACAP の処理の流れを 図 2.3 に示す．ACAP は, MIPS 用 GCC で得られるアセンブリを CDFG に変換し, データフロー解析, スケジューリング, バインディングを行い最終的に Verilog HDL を出力する．

ACAP には, 分割コンパイルモード, 全体合成モード, アクセラレータモードの三つの合成モードが実装されている．分割コンパイルモードは, C 言語で書かれた関数群を, ソフトウェアから呼び出せるハードウェアに合成する．全体合成モードは, リンク済みの実行可能なバイナリコードから, それを実行する MIPS と 機能等価なハードウェアを合成する．アクセラレータモードは, リンク済み実行可能コードの一部を指定して, 等価な動作をより拘束に行うハードウェアを合成する．

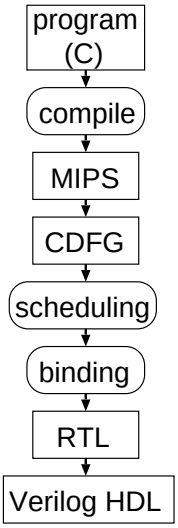

図 2.3: ACAP の高位合成の処理の流れ [9].

2.4 RSA 復号回路の高位合成

伊藤らは, バイナリ合成を用いて RSA 暗号化/復号回路を設計している [4]．C 言語で記述した RSA 暗号化/復号 のプログラムに, 多倍長整数演算ライブラリ GMP (GNU Multi-Precision Library) をリンクし, バイナリ合成システム ACAP を用いて RSA 暗号化/復号回路を設計している．

GMP は, GNU プロジェクトが提供する算術ライブラリであり, 任意精度の符号付き整数, 有理数, 浮動小数点数の演算を扱うことができる．多倍長整数の表現方法は, 図 2.4 に示す構造体 `__mpz_struct` によって表現する． このメンバである `_mp_d` は整数の値を表現する配列であり, `_mp_size` は使用されている要素数である．ここでの `mp_limb_t` は MIPS をターゲットにした場合, 32 ビットの符号なし整数となり, これは `unsigned int` に等しい．表 2.2 は, GMP の関数の

例である．高位合成する際, 多倍長整数演算ライブラリ GMP を高位合成に不要な関数を削除し, 動的な割当てを静的な割当てに書き換えたものを用いている．

太田らは, Fournaris のアルゴリズムを C 言語で記述し, 高位合成により設計した．このとき, ACAP の全体合成モードで合成することにより耐サイドチャネル攻撃 RSA 復号回路を設計している．しかし, サイドチャネル耐性を付加したことにより回路規模が増加し, 実行速度が低下する等の課題がある．

```
typedef struct
{
  int _mp_size;
  mp_limb_t *_mp_d;
} __mpz_struct;
```

図 2.4: __mpz_struct 型の構造体

表 2.2: GMP の関数例

mpz_add (r, a, b)	$r = a + b$
mpz_mul (r, a, b)	$r = ab$
mpz_powm (r, a, b, m)	$r = b^d \bmod M$

第3章　モンゴメリ法の適用

3.1　モンゴメリ法

モンゴメリ法は除算を行うことなく, 加算, 減算, 乗算, シフト演算のみで乗算剰余算を行う方法である. 除算及び剰余算を他の演算に代替できるため, 回路から除算と剰余算を削除することができる.

モンゴメリ法の処理の流れを図 3.1 に示す. R を 2 の冪乗数, N を定数とした時, R_2 もまた定数となる.

$$R_2 = R^2 \bmod N$$

モンゴメリ法はモンゴメリリダクションを使うことにより乗算剰余算を行うことができる. 次の関数がモンゴメリリダクションである.

$$MR(x) = x \cdot R^{-1} \bmod N$$

この関数は, 以下の方法で除算を用いずに計算できる.

1. $N' = -1 \bmod R$ を事前に計算
2. $s = (x \bmod R)N' \bmod R$
3. $t = (T + s) / R$
4. if $t \geq N$ then t = t – n

R は 2 の冪乗数のため, 除算はシフト演算, 剰余算は論理積で求めることができる. モンゴメリ表現への変換は次の関数を使うことにより行うことができる.

$$M(x) = MR(xR_2)$$

モンゴメリ表現からの逆変換は次の関数で行うことができる.

$$M^{-1}(x) = MR(x)$$

モンゴメリ表現での積は乗算剰余算同じ意味をなす. 次のものがモンゴメリ表現での積である.

$$x \otimes y = MR(xy)$$

モンゴメリ表現での積を逆変換することにより乗算剰余算を行うことができる.

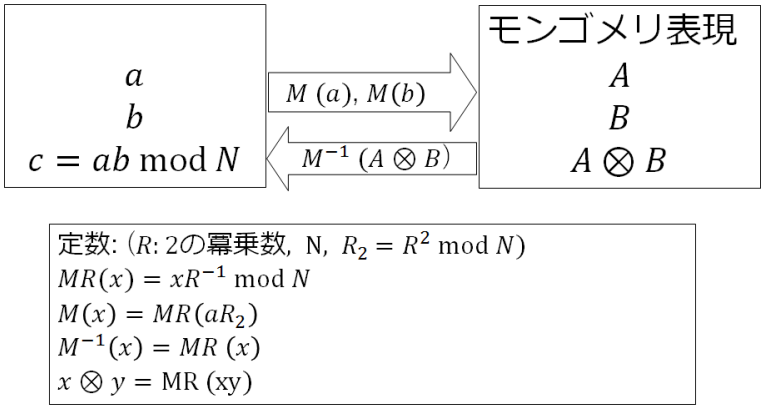

図 3.1: モンゴメリ法を用いた乗算剰余算

3.2 関数の追加

本研究では, モンゴメリ法を利用するために GMP を利用した新しい関数を追加する. 追加する関数は, 乗算剰余算を計算する関数 mmm と, 冪剰余算を計算する関数 powm である.

関数 mmm の記述を, 図 3.2 に示す. これは, モンゴメリ法を用いて乗算剰余算を行う関数である. この関数は, $r, a, b, m, minv, r\_shift, t$ を入力として $r = a \cdot b \bmod m$ を計算する. m は, RSA の秘密鍵であり, r_shift はモンゴメリ法の定数である. $minv$ は $m \cdot minv \equiv -1 \bmod 2^{r\_shift}$ を, t は $t = 2^{r\_shift} \bmod m$ を満たす数である. `N_SIZE` は鍵の格納に必要な語数であり, 鍵の最大ビット数を語長 (32 ビット) で割った値である. 関数 mr は, モンゴメリリダクションを計算する.

関数 powm の記述を, 図 3.3 に示す. これはモンゴメリ法を用いて冪剰余算を行う関数である. この関数は, $r, b, e, m, rr, invv, m\_r, r\_m$ を入力として, $r = b^e \bmod m$ を計算する.

3.3 モンゴメリ法の適用

太田らの RSA 復号回路は関数 fscame の一部だけモンゴメリ法を用いて計算を行っていた. 本研究では, 残りの乗算剰余算全てにモンゴメリ法を適用する. 残りの乗算剰余算を用いている箇所は, 図 3.4 と図 3.5 中の下線部である.

図 3.6 はモンゴメリ法への適用方法である. $r = a \cdot b \bmod N$ を計算する際, モンゴメリ法の適用により図の左の計算を右のように変更する.

また, モンゴメリ法を適用する $R, T, p^{-1}, q^{-1}, N^{-1}$ を事前に計算して定数として与える.

```
static void
mmm(mpz_t r, const mpz_t a, const mpz_t b, const mpz_t m,
    const mpz_t minv, const mp_size_t r_shift, const mpz_t t){

    mp_limb_t A_d[N_SIZE];
    mp_limb_t B_d[N_SIZE];
    mp_limb_t tmp1_d[N_SIZE*2];
    mp_limb_t abs_d[N_SIZE];
    mp_limb_t rev_d[0];

    __mpz_struct A= {0, A_d};
    __mpz_struct B= {0, B_d};
    __mpz_struct tmp1= {0, tmp1_d};
    __mpz_struct abs= {0, abs_d};
    rev_d[0]=1;
    __mpz_struct rev = {-1, rev_d};

    mpz_abs(&abs, a);
    mpz_mul(&tmp1, &abs, t);
    mr(&A, &tmp1, m, minv, r_shift);//
    mpz_mul(&tmp1, b, t);
    mr(&B, &tmp1, m, minv, r_shift);
    mpz_mul(&tmp1, &A, &B);
    mr(&tmp1, &tmp1, m, minv, r_shift);
    mr(r, &tmp1, m, minv, r_shift);
    if(a->_mp_size<0){
        mpz_sub (r, m, r);
    }
}
```

図 3.2: 関数 mmm

```
static void
mpz_powm
(mpz_t r, const mpz_t b, const mpz_t e, const mpz_t m,
const mpz_t rr, const mpz_t invv, const mp_size_t m_r, const mpz_t r_m){
    mp_limb_t minv_d[N_SIZE];
    mp_limb_t tmp1_d[N_SIZE*2];
    mp_limb_t tmp2_d[N_SIZE*2];
    mp_limb_t A_d[N_SIZE];
    mp_limb_t bb_d[N_SIZE];
    mp_limb_t X_d[N_SIZE];
    mp_limb_t t_d[N_SIZE];
    __mpz_struct tmp1 = {0, tmp1_d};
    __mpz_struct tmp2 = {0, tmp2_d};
    __mpz_struct minv = {0, minv_d};
    __mpz_struct A = {0, A_d};
    __mpz_struct X = {0, X_d};
    __mpz_struct t = {0, t_d};
    __mpz_struct bb = {0, bb_d};
    mp_size_t r_shift;

    r_shift =  m_r;
    mpz_set(&minv, invv);
    mpz_set(&t, rr);
    set_2n(&tmp1, r_shift);
    mpz_set(&A, r_m);
    mmm(&X, b, &tmp1, m, &minv, r_shift, &t);
    int i, en = get_digits(e), bits, index;

    for (i = en; i != 0;){
        mpz_mul(&tmp1, &A, &A);
        mr(&A, &tmp1, m, &minv, r_shift);
        i--;
        bits = i%GMP_LIMB_BITS;
        index = i/GMP_LIMB_BITS;
        if (((1 << bits) & e->_mp_d[index])){
        mpz_mul(&tmp1, &A, &X);
        mr(&A, &tmp1, m, &minv, r_shift);
    }
}
    mr(r, &A, m, &minv, r_shift);
}
```

図 3.3: 関数 mpz_powm

RSA 復号

Input: $c, b, b^{-1}, p, q, d_p, d_q, i_q = q^{-1} \bmod p, N$

Output: $c^d \bmod N$

$(s_0^p, s_1^p, s_2^p, s_4^p) = \mathrm{FSCAME}(c, b, b^{-1}, d_p, p);$

$(s_0^q, s_1^q, s_2^q, s_4^q) = \mathrm{FSCAME}(c, b, b^{-1}, d_q, q);$

$S_0 = s_0^q + q \cdot (\underline{(s_0^p - s_0^q) \cdot i_q \bmod p});$

$S_1 = s_1^q + q \cdot (\underline{(s_1^p - s_1^q) \cdot i_q \bmod p});$

$S_2 = s_2^q + q \cdot (\underline{(s_2^p - s_2^q) \cdot i_q \bmod p});$

$S_4 = s_4^q + q \cdot (\underline{(s_4^p - s_4^q) \cdot i_q \bmod p});$

if ($\underline{S_0 \cdot S_1 \bmod N = S_2}$ **and** p, q not modified)

{ **return** $(S_0 \cdot S_4 \bmod N)$; }

else { **return** error; }

図 3.4: Fournaris のアルゴリズム [7].

耐攻撃モンゴメリ冪剰余算

Function: FSCAME
Input: $c, b, b^{-1}, d = (1, d_{t-2}, ...d_0), M$
Output: (s_0, s_1, s_2, s_4)
$R = 2^{n+2}$; $T = R^2 \bmod M$;
$b_R = \underline{b \cdot R \bmod M}$;
$b_{R-1} = \underline{b^{-1} \cdot R \bmod M}$;
$s_0 = s_1 = b_R$;
$T_R = T \cdot c \cdot R^{-1} \bmod M$;
$s_2 = b_R \cdot T_R \cdot R^{-1} \bmod M$;
$s_3 = s_4 = s_5 = b_{R-1}$;
for $(i = 0$ to $t - 1)$ {
  if $(d_i = 1)$ {
    $s_0 = s_0 \cdot s_2 \cdot R^{-1} \bmod M$;
    $s_4 = s_4 \cdot s_3 \cdot R^{-1} \bmod M$;
  } **else** {
    $s_1 = s_1 \cdot s_2 \cdot R^{-1} \bmod M$;
    $s_5 = s_5 \cdot s_3 \cdot R^{-1} \bmod M$;
  }
  $s_2 = s_2^2 \cdot R^{-1} \bmod M$;
  $s_3 = s_3^2 \cdot R^{-1} \bmod M$;
}
$s_0 = s_0 \cdot b^{-1} \cdot R^{-1} \bmod M$;
$s_1 = s_1 \cdot c \cdot R^{-1} \bmod M$;
$s_2 = s_2 \cdot 1 \cdot R^{-1} \bmod M$;
$s_4 = s_4 \cdot b \cdot R^{-1} \bmod M$;
if (i and d are not modified **and** $s_0 \cdot s_1 \cdot R^{-1} \bmod M = s_2 \cdot 1 \cdot R^{-1} \bmod M$)
  { **return** (s_0, s_1, s_2, s_4); }
else { **return** error; }

図 3.5: 関数 FSCAME [7].

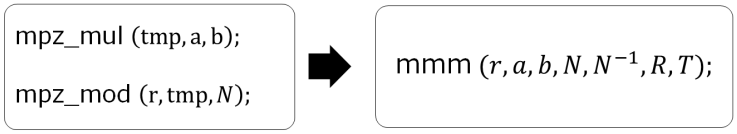

図 3.6: モンゴメリ法の適用

第4章　並列化

4.1　並列化の対象

RSA の計算時間の大半は冪剰余算であり, モンゴメリ冪剰余算を計算する関数 FSCAME が計算の大部分を占める. そこで FSCAME の並列化の適用による効果は大きいと考えられるため, 図 2.1 に二箇所ある関数 FSCAME を並列に計算する. RSA においては, この 2 つの処理は互いに独立であるため並列に計算を行うことができる.

具体的な並列化の構成を図 4.1 に示す. FSCAMEp は秘密鍵 p の冪剰余算で, FSCAMEq は秘密鍵 q の冪剰余算である. CRT は図 2.1 中国人剰余定理の処理を行う. 並列化適用前は図の上のように FSCAMEp, FSCAMEq, CRT と順に処理を行っていた. 並列化適用後は図の下のように, FSCAMEp と FSCAMEq を並列に計算を行い双方の処理が終え次第 CRT の処理に移行する. この際モジュールを FSCAMEp + CRT と FSCAMEq の 2 個にした. FSCAMEp, FSCAMEq, CRT の 3 個にした場合, 加算器や乗算器などが 3 個必要になり回路規模が 2 個の場合よりも回路規模が増加する.

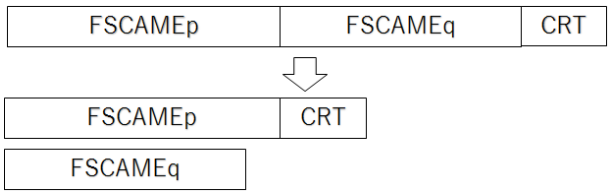

図 4.1: 復号処理の並列化

4.2　並列化を適用した RSA 復号回路

本研究では, C 言語のプログラムを並列に計算できるようにし高位合成を行うことにより並列化をする. 図 4.2 は, 並列化適用後のハードウェア構成を示している. 2 箇所の FSCAME を並列に計算するために, RSA 復号回路本体から FSCAMEq を分離する. メモリは共有する.

図 4.3 は C 言語での並列化の記述である. 1 行目は, 分離させた FSCAMEq を呼び出すための関数である. 2 行目で FSCAMEp の計算を行う. 3 行目で双方の処理が終わるまで待機する.

図 4.4 は, 4.3 の 2 行目の FSCAMEq を呼び出すための関数である. 引数は, 関数 fscame と等しい. `ARGq_s0`, `ARGq_s1` ,`ARGq_s2`, `ARGq_s4`, `ARGq_c`, `ARGq_b`, `ARGq_binv`, `ARGq_e`, `ARGq_n`, `ARGq_minv`, `ARGq_r_m`, `RUNq` は, C 言語の型である void* volatile で外部変数

として宣言する．16 行目から 26 行目でメモリに書き込むことにより FSCAMEq に値を渡す．28 行目は RUNq に 1 を代入する．RUNq が 1 になることにより, FSCAMEq が起動する．

図 4.5 は, FSCAMEq のメイン関数である．4 行目から 18 行目は, RSA 復号回路とメモリを共有するために一度呼び出している．一度も使用しない場合 ACAP は不要な変数として認識しメモリを確保しない．20 行目の RUNq の初期値は 0 であるため, RSA 復号回路から呼び出されるのを待っている．o 呼びだされ RUNq の値が 1 になることにより 実行状態へと移る, 22 行目, 23 行目で fscame で FSCAMEq の計算を行う．25 行目で処理が終わったことを RSA 復号回路に伝えるために RUNq の値を 0 に戻す．

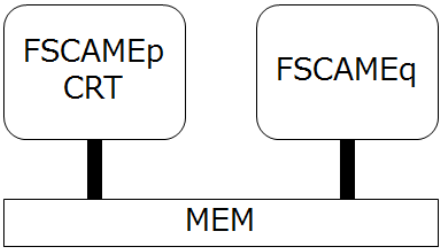

図 4.2: ハードウェアの構成

```
1:  fscameq(&s0q, &s1q, &s2q, &s4q, &c, &b, &binv, &dq, &q, &minv2, &r_m2);
2:  fscame(&s0p, &s1p, &s2p, &s4p, &c, &b, &binv, &dp, &p, &minv, &r_m1);

3:  while(RUNq){;}
```

図 4.3: 並列化の C 言語の記述

```
 1: static
 2: fscameq(
 3:   mpz_t s0,
 4:   mpz_t s1,
 5:   mpz_t s2,
 6:   mpz_t s4,
 7:   const mpz_t c,
 8:   const mpz_t b,
 9:   const mpz_t binv,
10:   const mpz_t e,
11:   const mpz_t m,
12:   const mpz_t minv,
13:   const mpz_t r_m
14: )
15: {
16:     ARGq_s0 = s0;
17:     ARGq_s1 = s1;
18:     ARGq_s2 = s2;
19:     ARGq_s4 = s4;
20:     ARGq_c = c;
21:     ARGq_b = b;
22:     ARGq_binv = binv;
23:     ARGq_e = e;
24:     ARGq_m = m;
25:     ARGq_minv = minv;
26:     ARGq_r_m = r_m;
27:
28:     RUNq = (int*)1;
29: }
```

図 4.4: 関数 fscameq

```
 1: int main(void){
 2:     void* x;
 3:     int a;
 4:     x = _RUN_sub;
 5:     x = ARGq_s0;
 6:     x = ARGq_s1;
 7:     x = ARGq_s2;
 8:     x = ARGq_s4;
 9:     x = ARGq_c;
10:     x = ARGq_b;
11:     x = ARGq_binv;
12:     x = ARGq_e;
13:     x = ARGq_m;
14:     x = ARGq_minv;
15:     x = ARGq_r_m;
16:     x = RUNq;
17:     a = _ERR_sub;
18:     rsa_t rsa2 = rsa1;
19:
20:     while(!RUNq){;}
21:
22:     fscame(ARGq_s0, ARGq_s1, ARGq_s2, ARGq_s4, ARGq_c, ARGq_b,
23:     ARGq_binv, ARGq_e, ARGq_m, ARGq_minv, ARGq_r_m);
24:
25:     RUNq = (int*)0;
26:
27:     return 0;
28: }
```

図 4.5: fscameq.c のメイン関数

第5章　実験結果

5.1　耐サイドチャネル攻撃 RSA 復号回路の合成

バイナリ合成システム ACAP によりサイドチャネル耐性を付加した RSA 復号回路の合成を行った. 具体的なコマンドは次の通りである.

(1) 下記のコマンドで, スケジューリングまでを行い中間ファイルを出力する.

```
acap.pl -Z1 -O2 -Tsch srr.c
```

“acap.pl” で ACAP を起動して合成を行う. “-Z1” は ACAP のオプションで全体合成モードで行う. “-O2” は gcc の最適化オプションであり, GCC のバージョンは 5.4.0 を用いた. “-Tsch” は高位合成の処理のうち, スケジューリングまでを行う. “srr.c” は合成対象のプログラム名である. スケジューリング結果は “srr.sch” というファイルが出力される.

(2) 下記のコマンドでバインディングを行う.

```
ilpbind.pl -c0 -I30 -L30 -P1.03 -t240 -m reg+2 srr.sch
```

バインディングには整数線形計画法によるもの [14] を用い, ソルバーには CPLEX 12.6.1 を用いた. “ilpbind.pl” は整数線形計画法を用いたバインディングを行うコマンドである. “-c0” はチェイニングを行わないことを指定する. “-I30”, “-L30”, “-P1.03” は 1 回の繰り返しで解くバインディング問題の規模を制御するパラメータである. また, “-t240” は, 1 回の繰り返しで整数線形計画法に使用できる計算時間の上限である. “-m reg+2” は, レジスタ数を必要最小数より 2 つまで増やしてバインディングを行う. 入力ファイルは上記のコマンドによってスケジューリングされた “srr.sch” であり, バインディングの結果は “srr.bnd” というファイルが出力される.

(3) 下記のコマンドで, “srr.bnd” を入力することにより Verilog HDL 記述である “srr.v” に出力した.

```
acap.pl -Z1 srr.bnd
```

5.2　回路規模および実行速度

生成された Verilog HDL の動作確認は RTL シミュレーション (Xilinx 社の Isim) により行った. また, 論理合成は Xilinx Vivado (2016.4) を用いて FPGA (kintex-7 xc7k70) をターゲットに行った.

合成結果を表 5.1 に示す. “cycles” は実行にかかるサイクル数, “#LUT” は LUT 数, “freq” は動作周波数を表している.

“MIPS” はソフトコアでの実行結果を表している. “RSA” は伊藤 [4] の攻撃に耐性のない RSA 復号回路回路, “SRR” は太田 [6] の攻撃に耐性のある RSA 復号回路, “SRR+M” は “SRR” にモンゴメリ法を適用した回路, “SRR+M+P” は “SRR+M” に並列化を適用した回路である. モンゴメリ法の適用により動作周波数の約 1.4 倍した. また回路規模を約 68% に削減できた. さらに並列化により回路規模は SRR の約 1.4 倍になったが, サイクル数は約 56% に削減できた. 結果, RSA の約 5.2 倍の実行時間, 約 2.0 倍の回路規模でサイドチャネル攻撃耐性を付加できた. 実行速度は, 遅延時間とサイクル数を乗じた値から求めた.

表 5.1: 合成結果

design	cycles	#LUT	freq. [MHz]
RSA (MIPS)	432,256	3,180	121.7
RSA [4]	68,261	11,721	107.8
SRR (MIPS)	3,557,349	3,180	121.7
SRR [6]	627,615	16,801	77.5
SRR+M	680,284	11,464	105.5
SRR+M+P	353,489	22,590	105.6

5.3 消費電力

伊藤の RSA 復号回路と 太田のサイドチャネル耐性を付加した RSA 復号回路 SRR に単純電力解析を行った. 消費電力は Xilinx Vivado (2016.4) を用いた.

RSA, SRR の電力の波形の一部を図 5.1 と図 5.2 に示す. 縦軸が電力, 横軸が時間である. 図 5.1 の RSA の電力波形は電力の最小値と最大値に大きな差がある. また波長が一定でなく, 波形も様々である. 0 の処理と 1 の処理にそれぞれ特有の波形があるように見えるが, 鍵を特定できないと考えられる.

それに対し, 図 5.2 の SRR の電力波形は, 電力に大きな差がない. 波長と波形に大きな差はないため, SRR も鍵を特定できないと考えられる.

鍵推定はできなかったが RSA と SRR の電力波形でサイドチャネル耐性に大きな差があると考えられる.

5.4 考察

モンゴメリ法及び並列化を適用したサイドチャネル攻撃に耐性を持つ RSA 復号回路の高位合成を行った, モンゴメリ法の適用により動作周波数が約 1.4 倍増加した. 動作周波数が増加した一因は, モンゴメリ法のプログラムのコードと比べ, 元の剰余算の計算に利用していたプログラムのコードが長くこれを削除したことによりプロクラムが短くなったためである. 回路規模は, 約 68%

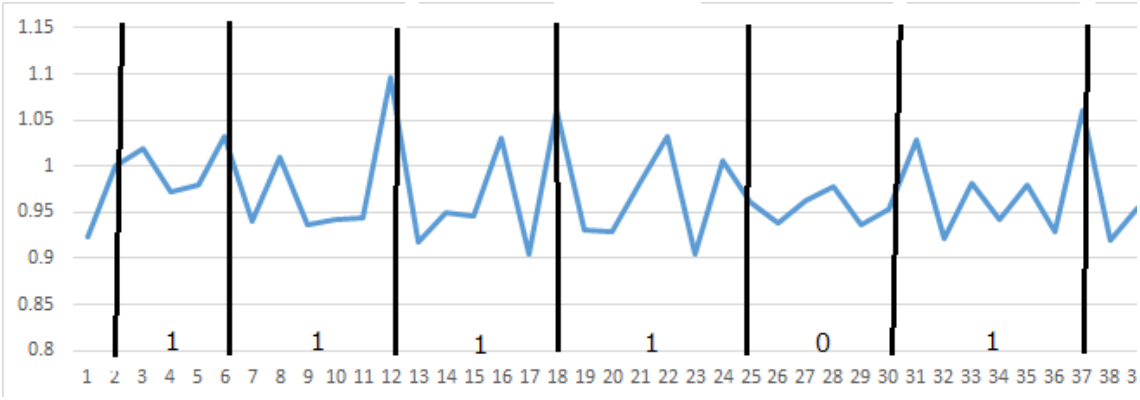

図 5.1: 回路 “RSA” の電力の波形

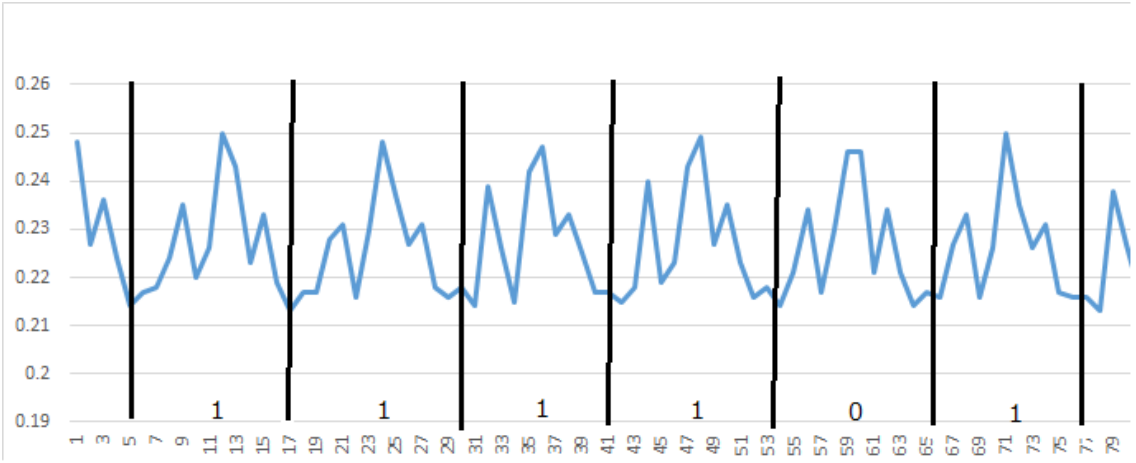

図 5.2: 回路 “SRR” の電力の波形

に削減できた. これはモンゴメリ法では除算, 剰余算の回路を使用しないため削除することにより削減できた. 並列化の適用によりサイクル数を約 50% に削減した. サイクル数を約半分に削減できたのは, 計算の大半が冪剰余算のためである. 回路規模は約 2.0 倍に増加した. これは回路の大半を冪剰余算で使用しているためである. 結果, 並列化及びモンゴメリ法の適用前と比べて, 約 1.3 倍の回路規模で約 2.4 倍の高速化を実現した.

伊藤の RSA 復号回路と 太田のサイドチャネル耐性を付加した RSA 復号回路に単純電力解析を行った. 秘密鍵の特定は双方ともできなかったが, サイドチャネル耐性を付加した回路のほうが鍵の推定が難しいと考えられる.

第6章　結論

本論文では, モンゴメリ法及び並列化を適用した耐サイドチャネル攻撃 RSA 復号回路の高位合成により設計する手法を提案した. モンゴメリ法の適用により動作周波数を向上させ, 並列化の適用によりサイクル数を削減し, バイナリ合成システム ACAP を用いてハードウェア化を行った.

実験の結果, モンゴメリ法及び並列化を適用した RSA 復号回路は, 適用前と比べて約 2.4 倍高速化を実現できた.

消費電力を測ったが, サイドチャネル攻撃に耐性を持つことを確認することができなかった. これは, データ数が少ないため統計的に処理できなかったためと考えられる. したがってこの解決は, データ数を増やし差分を取ることにより解析できると考えられる.

今後の課題としては, さらなる実行の高速化, 回路規模の削減および実機への移行が挙げられる. また ACAP 以外の高位合成システムを用いた実験をすること, 及び多くの電力波形を出力し攻撃への耐性を確認することも重要な課題である.

謝辞

本研究に際し, 多くの方々から御指導, 御支援を賜りました. ここに感謝の意を表します.

高位合成の研究に携わる機会を与えていただき, 研究に関する多くの御指導, 御支援を賜りました石浦菜岐佐教授に心より感謝いたします.

本研究に関する有益な議論の場を提供して頂き, 様々な視点から御指導いただきました京都高度技術研究所の神原弘之氏に感謝いたします. 本分野に対する高い見識を持ち, 様々な面で御助力いただきました立命館大学の冨山宏之教授に感謝いたします. 技術的な面で多くの御助言をいただきました元立命館大学の中谷嵩之氏に感謝いたします. 本研究に関して御協力, 御討議頂いた関西学院大学の太田小百合氏をはじめ, HLS チームの諸氏に深く感謝いたします. 最後に, 研究のみならず生活面においても多くの御支援をいただいた関西学院大学理工学部石浦研究室の皆様に深く感謝いたします.

参考文献

[1] 本間尚文, 青木孝文, 佐藤証: “暗号モジュールへのサイドチャネル攻撃とその安全性評価の動向,” 電子情報通信学会論文誌, vol. J93-A, no. 2, pp. 42–51 (Feb. 2010).

[2] 岡本栄司: 暗号理論入門 第 2 版, 共立出版 (2002).

[3] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[4] 伊藤直也, 竹林陽, 神原弘之, 石浦菜岐佐: “多倍長整数演算ライブラリをリンクしたバイナリコードからの RSA 暗号回路の高位合成,” 情報処理学会関西支部大会, A-04 (Sept. 2015).

[5] N. Ishiura, H. Kanbara, and H. Tomiyama: “ACAP: Binary synthesizer based on MIPS object codes,” in *Proc. International Technical Conference on Circuit/Systems, Computers and Communications (ITC-CSCC 2014)*, pp. 725–728 (July 2014).

[6] 太田小百合, 由良駿, 石浦菜岐佐: “電力解析攻撃/故障利用攻撃耐性 RSA 復号回路の高位合成,” 電子情報通信学会ソサイエティ大会, A-6-6 (Sept. 2016).

[7] A. P. Fournaris and O. Koufopavlou: “Protecting CRT RSA against fault and power side channel attacks,” in *Proc. 2012 IEEE Computer Society Annual Symposium on VLSI*, pp. 159–164 (Aug. 2012).

[8] N. Osako, S. Ota, S. Yura and N. Ishiura: “High-level synthesis of side channel attack resistant RSA decryption circuit” in *Proc. Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2018)*, (Mar. 2018 to appear).

[9] 太田小百合: “電力解析攻撃および故障利用攻撃に耐性を持つ RSA 復号回路の高位合成,” 関西学院大学理工学部情報科学科卒業論文 (Mar. 2017).

[10] D. Boneh, R. A. DeMillo, and R. J. Lipton: “On the importance of checking cryptographic protocols for faults,” in *Proc. International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 37–51 (May 1997).

[11] C. H. Kim and J.-J. Quisquater: “How can we overcome both side channel analysis and fault attacks on RSA-CRT?” in *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pp. 21–29 (Sept. 2007).

[12] S. Mangard, E. Oswald, and T. Popp: *Power Analysis Attacks—Revealing the Secrets of Smart Cards*, Springer (2007).

[13] S.-M. Yen, W.-C. Lien, S.-J. Moon, and J. Ha: "Power analysis by exploiting chosen message and internal collisions—Vulnerability of checking mechanism for RSA-decryption," in *Proc. International Conference on Cryptology & Malicious Security*, pp. 183–195 (Sept. 2005).

[14] 大迫裕樹, 石浦菜岐佐: "高位合成のバインディングの整数線形計画法による分割解法における実数制約の導入," 電子情報通信学会ソサイエティ大会, A-6-7 (Sept. 2016).