暗号実装最適化
— 暗号実装屋の不毛な戦い

青木 和麻呂

NTT

自己紹介

- 某電話会社に勤めています。
- 暗号学者です。
- 暗号の解読と高速実装が好きです。
- 本業では **Camellia** 暗号の設計とか、**768** ビット数の素因数分解とかやりました。

注

- **(詳細な)** 知識は **Pentium 4** と初代 **Opteron** で止まってます。なるべく最新の話も入れますけど。
- 紹介するネタは他人が発見したネタもあります。

内容

- 暗号実装
- 公開鍵暗号の実装
- 共通鍵暗号の実装
- まとめ

暗号実装

- 高速実装が求められる最後の領域**(?)**
  (画像屋は**GPU**にいっちゃったし…**)**

暗号実装

- 高速実装が求められる最後の領域**(?)**
  (画像屋は**GPU**にいっちゃったし…**)**
- **(**一般人には**)** 安全性は謎。優劣は速度でしか比較できない。

暗号実装

- 高速実装が求められる最後の領域**(?)**
  (画像屋は**GPU**にいっちゃったし…**)**
- **(**一般人には**)** 安全性は謎。優劣は速度でしか比較できない。
- でも、暗号アプリケーションからは
  - いくら高速にしたところで間に合わない

暗号実装

- 高速実装が求められる最後の領域**(?)**
  (画像屋は**GPU**にいっちゃったし…**)**
- **(**一般人には**)** 安全性は謎。優劣は速度でしか比較できない。
- でも、暗号アプリケーションからは
  - いくら高速にしたところで間に合わない
  - 十分に高速で速くする意味がない

暗号実装

- 高速実装が求められる最後の領域**(?)**
  (画像屋は**GPU**にいっちゃったし…**)**
- **(**一般人には**)** 安全性は謎。優劣は速度でしか比較できない。
- でも、暗号アプリケーションからは
  - いくら高速にしたところで間に合わない
  - 十分に高速で速くする意味がない

でも、最適化しますけどね。楽しいから。

公開鍵暗号の実装

- アルゴリズムレベルでの最適化がメイン (x86 だから…という話はかなり末端処理)
- 1 単位の処理が 100μ 秒前後 (≈100k cycle)
- イマドキだと、自作セマフォを使った thread 同期処理とか (by 光成さんペアリング)
- お題を 5 つ: OEF、素体逆元、多倍長整数演算、篩処理、高速完全性検証

OEF

- 最適拡大体 **(Optimal Extension Fields)**。有限体の一種。
- イマドキの **CPU (**通常の回路を起こすと乗算とかは遅いが、回路面積を割いて高速になっている**)** で高速になるように有限体のパラメータを選択。
- 高速な楕円曲線暗号の設計などで利用。
- **1998** 年 **Bailey** らにより発表。**(**チーム内で同じようなことを考えていたけど、査読でまわってきて負けを悟る**)**

OEFの定義

- $\mathbf{GF}(p^n) \approx \mathbf{GF}(p)[x]/(f(x))$
  ($\deg f = n$, f は $\mathbf{GF}(p)$ 上既約)
- $p = 2^m \pm c$ (c は小さい、m はワードサイズぐらいかその半分)、
  $f(x) = x^n \pm b$ (b は小さい)
- 色々拡張されている。

OEFでの演算

内部では、ワードサイズに納まる

$$a_L + c \times a_H \qquad (c \text{は小})$$

という演算多数。

解決策は

OEFでの演算

内部では、ワードサイズに納まる

$$a_L + c \times a_H \qquad (c \text{は小})$$

という演算多数。

解決策は

lea

lea

- `lea r1, [r2 + scale * r3]`
  (r1, r2, r3…一般レジスタ、scale=1,2,4,8)
  $\mathtt{r1} \leftarrow \mathtt{r2} + \mathbf{scale} \times \mathtt{r3}$
- 入力を潰さない…神!
  (最近はv始まり命令があるが…)
- まさに「$a_L + c \times a_H$」が出来る。

(αだと`s{4,8}{add,sub}{l,q}`とか**)**

lea の注意点

- 昔は **AGI** とかあった。
- フラグが変わらないのは善し悪し。
- 使える **ALU** が少ないかも。

mod$p$のC言語実装

- $a + b \bmod p$ (a, b, pはワードサイズ以内、$0 \leq a, b < p$)
- ```
  a = a + b;
  if(a >= p){  /* 分岐確率50% */
      a -= p;
  }
  ```
- ```
  a = a + b;
  a -= (-(a >= p)) & p;
  ```
  SIMD系の命令ではありがちの方法

mod$p$ の **asm** 実装

```
      add a, b
      mov t, a
      sub a, p
      jnc l0
      mov a, t
l0:
```

```
      add a, b
      sub a, p
      jnc l1
      add a, p
l1:
```

```
  add a, b
  mov t, a
  sub a, p
  cmovc a, t
```

cmovc がない環境では setc や sbb で同様のことが出来るが、そういう **CPU** では分岐はあんまり遅くない…

素体逆元

- $a^{-1} \bmod p$の計算
  ($ax + py = 1$となるxの計算**)**
- 最大公約数**(GCD)**計算の変形で計算できる

「拡張**Euclid**の互除法」**vs**「バイナリ**GCD**」

- 拡張**Euclid**の互除法**:** 割算がメイン
- バイナリ**GCD:** ループ内部で**4**分岐**(**それぞれ確率**1/4)**

GCD 計算の原理

素体逆元と基本は同じなので **GCD** の説明

$\mathrm{GCD}(a, b)$ の計算 **(**$a \geq b$**)**

Euclid の互除法:

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(b, a \bmod b)$$

バイナリ **GCD:** $\mathrm{GCD}(a, b) =$

$$\begin{cases} 2\mathrm{GCD}(a/2, b/2) & a\text{: even, } b\text{: even} \\ \mathrm{GCD}(a/2, b) & a\text{: even, } b\text{: odd} \\ \mathrm{GCD}(a, b/2) & a\text{: odd, } b\text{: even} \\ \mathrm{GCD}(a - b, b) & a\text{: odd, } b\text{: odd} \end{cases}$$

Euclidの互除法の最適化

- 基本的に割算、しかもレジスタ縛り (rax,rdx) なので、最適化の余地は少ない
- 実は、商 $(a \bmod b)$ に偏りあり

```
if(a - b < b){
    商=1, 余りa-b
}else if(a - b - b < b){
    商=2, 余りa-b-b
}else ...
```

拡張Euclidの互除法の場合P3で10回展開ぐらいまでは速い？

バイナリGCDの最適化(1/2)

最内周ループの分岐予測が問題。

$$\mathrm{GCD}(a,b) = \begin{cases} \mathrm{GCD}(a-b,b) & a > b \\ \mathrm{GCD}(a,b-a) & a < b \end{cases}$$

$a + b \bmod p$と同様に分岐を削ると

```
a -= (-(a > b)) & b;
b -= (-(b > a)) & a;
```

cmovでも同様に実現可。
拡張バイナリ**GCD**では、複数の変数を最内周ループで更新する必要がある。効率微妙。

バイナリ GCD の最適化 (2/2)

x[0]=a, x[1]=b と変数を割り当て

```
i      = x[0] > x[1];
x[i] -= x[1 - i];
```

addressing でも分岐除去可。

Euclid の互除法とバイナリ **GCD** のどちらが速いかは、**architecture** と **code** により、すぐに入れ替わるので、どちらともいえない。

多倍長整数演算

- **de facto** の公開鍵暗号では多倍長整数の四則演算が必要。
- 例えば、**RSA** では **1024** ビット **(128** バイト**)** とか **2048** ビットとか。
- **64** ビットレジスタでも **16(32)** ワードも必要。

多倍長整数加算(素朴)

r_i, s_i を 64 ビットレジスタ、3 倍長整数:

$$a = \sum_{i=0}^{2} r_i 2^{64i}, \quad b = \sum_{i=0}^{2} s_i 2^{64i}$$

$a + b$ の素朴な実装:

```
add r0, s0
adc r1, s1
adc r2, s2
```

問題点 1: **critical path** が長い

問題点 2: **carry** の 1 ビットを足すためだけに 1μop 使われる

多倍長整数加算(冗長表現)

r_i, s_i を 64 ビットレジスタ、3 倍長整数:

$$a = \sum_{i=0}^{3} r_i 2^{48i}, \quad b = \sum_{i=0}^{3} s_i 2^{48i}$$

$a + b$ の実装:

```
add r0, s0
add r1, s1
add r2, s2
add r3, s3
```

○ μop 数が減少
○ 命令依存性なし
× 時々正規化が必要

扱う計算 (Karatsuba 乗算とか) により空ける上位ビット数の評価が必要

篩処理

- 多倍長の合成数の素因数分解が出来る
  ⇒**RSA** が解読できる
- 現状、ある程度以上大きな数に対しては数体篩法がもっとも高速な素因数分解法

```
forprime p
    compute staring point x_p (≈ 0)
    while(x < H)
        s[x] += log p
        x += p
    end while
end forprime
```


篩処理の最適化

問題点:

- p が小さいものについては問題なし
- p が H に近くなってくると $s[x]$ へのアクセスは殆んどキャッシュミス (H は実メモリの大きさに近い)

解決策: 加算は可換であることを利用し、前もってアクセス先により更新内容 $(x, \log p)$ を整列しておく
整列なんかしていたら…と思うが、実は数倍速くなる (ことがある)

高速完全性検証(宣伝)

- 転送したファイルの **SHA-256** ハッシュ値の検証と同等の機能
- 比較対象をメモリにおいて単純に **memcmp()** するより高速
- 残念ながら、タグ **(SHA-256** ハッシュ値のようなもの**)** 生成は超遅く、しかもそれなりに大きい

興味ある人は「**SCIS2007 1C2-3** 高速データ検証」を見て下さい。

共通鍵暗号の実装

- ブロック暗号限定ということで(その他の実装経験はないので)
- **1** 単位は数百 **cycle**
- 対象暗号毎、**x86** のコアのアーキテクチャ毎にかなりの違い
- 一般論: **(self-)bitslice**, 等価変形, 副鍵埋め込み, **T-box**, エンディアン変換
- 個別: **SWAPMOVE, 4-bit table (**pshufb**)**

bitslice 実装

- 64(32) ビットレジスタを 1-bit が 64 並列の SIMD と思う
- 全ての演算を and, or, xor などの論理式で
- 速くなればラッキー (鍵の全数探索とか)
- 実行時間が命令には依らなくなる (cache とかの影響少)
- 最適化対象によっては、1(暗号化) 単位の中で bitslice できることもある (self-bitslice)

等価変形による依存性緩和

仕様通りではなく、同じ結果を得る違う計算 (順序) を考える

Feistel 構造 ($1 \leq i \leq 16$ とか):

$$\begin{cases} L_{i+1} \leftarrow f(L_i \oplus k_i) \oplus R_i \\ R_{i+1} \leftarrow L_i \end{cases}$$

$$\Downarrow$$

$$\begin{cases} L_{i+1} \leftarrow f(L_i) \oplus R_i \oplus (k_{i-1} \oplus k_{i+1}) \\ R_{i+1} \leftarrow L_i \end{cases}$$

副鍵埋め込み

- 通常、共通鍵ブロック暗号は、秘密鍵を鍵スケジューリングというものを用いて副鍵に変形し、暗号化では副鍵のみを用いる
- 多ブロックの暗号化では副鍵の実行コードへの埋め込みが有利
- 最近は χ**byak** とかがあり便利

副鍵埋め込み例

関係箇所のみ抽出

NASM

```
%macro SMcamelliaRound 7
global Camelliak%5r
xor esp, 0x89abcdef
Camelliak%5r:
%endmacro
```

C 言語から

```
#define OFFSET(label) ((uint32_t)&(label) -\
    (uint32_t)SMcamelliaEnc128)
extern uint32_t Camelliak1r;
memcpy(code, SMcamelliaEnc128,
    OFFSET(endOfSMcamelliaEnc128));
(uint32_t*)&code[OFFSET(Camelliak1r)-4]=r[2];
```

T-box実装

AESなどの**byte-oriented**暗号は、1バイト$\rightarrow$4バイトとかの表$T_i[]$と論理演算の組合せが多い。平文データを(eax, ebx, ecx, edx)にもち

```
movzx esi, al
movzx edi, ah
xor ebp, T0[4*esi]
xor esp, T1[4*edi]
rol eax, 16
movzx esi, al
movzx edi, ah
xor ebp, T2[4*esi]
xor esp, T3[4*esi]
```

といった処理の繰返し。

endian 変換

32 ビットデータ x の **endian** 変換

- $(x \ll 24) + ((x\&\texttt{0xff00}) \ll 8) + ((x \gg 8)\&\texttt{0xff00}) + (x \gg 24)$

endian 変換

32 ビットデータ x の **endian** 変換

- $(x \ll 24) + ((x\&\texttt{0xff00}) \ll 8) + ((x \gg 8)\&\texttt{0xff00}) + (x \gg 24)$
- $((x\&\texttt{0xff00ff}) \ggg 8) + ((x \lll 8)\&\texttt{0xff00ff})$

endian 変換

32 ビットデータ x の **endian** 変換

- $(x \ll 24) + ((x\&\texttt{0xff00}) \ll 8) + ((x \gg 8)\&\texttt{0xff00}) + (x \gg 24)$
- $((x\&\texttt{0xff00ff}) \ggg 8) + ((x \lll 8)\&\texttt{0xff00ff})$
- `bswap`

SWAPMOVE

SWAPMOVE(A,B,N,M)

$$T = ((A \gg N) \oplus B)\&M;$$
$$B = B \oplus T;$$
$$A = A \oplus (T \ll N);$$

B の M で **mask** されたビットが、A の $M \ll N$ で **mask** されたビットと入れ替わる

- **DES の IP/FP は SWAPMOVE 5 回**
- 行列転置とかでも便利
- **punpck の一般形**

pmovmskb

DES IP:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

→

62	54	46	38	30	22	14	6
60	52	44	36	28	20	12	4
58	50	42	34	26	18	10	2
56	48	40	32	24	16	8	0
63	55	47	39	31	23	15	7
61	53	45	37	29	21	13	5
59	51	43	35	27	19	11	3
57	49	41	33	25	17	9	1

pmovmskb 8 回と位置合わせで完了

興味ある人は「**SCIS2002 12C2-2 Pentium III** 上の **Triple DES** 実装最適化記」参照のこと

4-bit table lookup

SSSE3 で **pshufb** 命令が追加された!

xmm0: $t[0], t[1], \ldots, t[15]$

xmm1: $x_0, x_1, \ldots, x_{15}$

⇓**pshufb xmm0, xmm1**

xmm0: $t[x_0], t[x_1], \ldots, t[x_{15}]$

- 定数時間実行
- $\mathbf{GF}(2^8) \cong \mathbf{GF}(2^4)[x]/$(**2** 次既約式) を利用した x^{-1} の計算

などの応用。

まとめ

まとめ

- 結局はスケジューリング

まとめ

- 結局はスケジューリング
- **Out-Of-Order** 嫌い。直接 μ**op** 書きたい。

まとめ

- 結局はスケジューリング
- **Out-Of-Order** 嫌い。直接 μ**op** 書きたい。
- 個別技ばっかり。研究者としては体系化したいんだけど。**(**せめて **Hacker's delight** 程度には**)**

まとめ

- 結局はスケジューリング
- **Out-Of-Order** 嫌い。直接 μ**op** 書きたい。
- 個別技ばっかり。研究者としては体系化したいんだけど。**(**せめて **Hacker's delight** 程度には**)**
- **AES-NI** で玉砕

まとめ

- 結局はスケジューリング
- **Out-Of-Order** 嫌い。直接 μ**op** 書きたい。
- 個別技ばっかり。研究者としては体系化したいんだけど。**(**せめて **Hacker's delight** 程度には**)**
- **AES-NI** で玉砕
- 一子相伝にならないようにするには…