名前

- opcount - 暗号方式中の演算をカウントする

書式

```
opcount [{preamble} ...] [-t {postamble} ...] < {filename} | gp -q
```

説明

opcount は ペアリングに基づく暗号方式記述言語 (cryptographic scheme description language, または C-like scheme description language, CSDL) で記述された暗号方式中の演算をカウントし, GP (PARI Calculator) 形式の 代入文の列を出力するフィルタプログラムである. CSDL を標準入力から読み カウント結果を標準出力に出力する. {preamble} ... が指定されたときは, 結果を標準出力に出力する前にファイル {preamble} ... の内容を順次 そのまま標準出力に出力する. -t {postamble} ... が指定されたときは, 結果を標準出力に出力した後にファイル{postamble} ... の内容を順次 そのまま出力する. 出力全体が一つの GP のプログラムになるような preamble および postamble のファイルを準備することが望ましい.

多重冪乗の演算コストの代数

$n \in \mathbb{N}_0$ および $x_0 \in \{0,1\}$ なる $\prod_{i=0}^{n} g_i^{x_i}$ 型の 多重冪乗を考える. $x_0, \ldots, x_n$ は単位的可換環 $\mathbb{L}$ の元を表現する変数 で冪と呼ばれる. $g_0, \ldots, g_n$ はある群 ($\mathbb{L}$-加群)$\mathbb{G}$ に属する元を表現する変数で基底と呼ばれる. 暗号方式を設計する際は, 個々の冪や基底が実際にはどの元に対応するかを考える必要は無く, 仮想的にそのような元があると考える (形式的多重冪乗). 例えば, $x_0 = 0$ のときは g_0 という元をそも そも考える必要は無いが, 仮想的にそのような元があると考える. 他の x_i と異なり x_0 だけ $\{0,1\}$ 上に制限される理由は $\mathbb{G}$ 上の乗算と冪乗をまとめて考えるのに便利だからである. 基底が全て $\mathbb{G}$ 型の単一の変数である 時, $\prod_{i=0}^{n} g_i^{x_i}$ を計算するのに必要な演算コストを (n, x_0) と書くとする. 例えば何の冪乗も乗算も無い $\mathbb{G}$ 型変数 g を計算する 演算コストは $(0,1)$ と書くことが出来る. 勿論

$$(0,1) = 0$$

である. 複数の形式的多重冪乗の代数的な演算コストを考察するため, 代数的表現 上の全ての多重冪乗を該当する演算コストに置き換えた表現 (演算コストの代数的表現) を考える. 例えば, 二つの $\mathbb{G}$ 型変数の積を計算する演算コストを mul と定義すると $g, h, k \in \mathbb{G}$ に対して次のような対応を考える事が出来る.

実際の演算	演算コストの代数的表現	実際の演算コスト
$h \times k \to g$	$(0,1) \times (0,1) \to (0,1)$	mul.

これらの対応により,

$$(0,1) \times (0,1) = (0,1) + \texttt{mul}$$

と考える事ができる. このような演算コストの代数的表現を一般の多重冪乗で 考える. 即ち

$$\text{実際の演算}\qquad\qquad\text{演算コストの代数的表現}$$
$$\left(h_0^{y_0}\prod_{i=1}^{m}h_i^{y_i}\right)\times\left(k_0^{z_0}\prod_{i=1}^{\ell}k_i^{z_i}\right)\to g_0^{x_0}\prod_{i=1}^{n}g_i^{x_i},\qquad (m,y_0)\times(\ell,z_0)\to(n,x_0),$$

但し, $n,m,\ell\in\mathbb{N}_0$, $x_0,y_0,z_0\in\{0,1\}$, を考察する.

$$g_i=\begin{cases}h_0^{y_0}\times k_0^{z_0} & i=0,\\ h_i & i\in\{1,\dots,m\},\\ k_{i-m} & i\in m+\{1,\dots,\ell\}\end{cases}$$

と定義すると

$$n=m+\ell,$$
$$x_0=y_0\vee z_0$$

として良い. $y_0\wedge z_0=1$ なる時は新しい基底 $g_0=h_0\times k_0$ を 計算する為に基底の乗算が一回必要となる. 従って

$$(m,y_0)\times(\ell,z_0)=(m+\ell,y_0\vee z_0)+(y_0\wedge z_0)\cdot\texttt{mul}$$

とする事が出来る. また, 多重冪乗の冪乗も同様にして考える.

$$\text{実際の演算}\qquad\qquad\text{演算コストの代数的表現}$$
$$\left(h_0^{y_0}\prod_{i=1}^{m}h_i^{y_i}\right)^{z}\to g_0^{x_0}\prod_{i=1}^{n}g_i^{x_i},\qquad (m,y_0)^z\to(n,x_0),$$

但し, $n,m\in\mathbb{N}_0$, $x_0,y_0\in\{0,1\}$.

$$g_i=\begin{cases}h_i & y_0=0,\ i\in\{0,\dots,m\},\\ h_{i-1} & y_0=1,\ i\in\{0,\dots,m+1\},\end{cases}$$

但し h_{-1} は適当な変数, と定義すると

$$n=m+y_0,$$
$$x_0=0$$

として良い. 即ち

$$(m,y_0)^z=(m+y_0,0)$$

とする事が出来る. `opcount` はこの多重冪乗の演算コストをカウントして出力する.

ビルド方法

まず, 以下を準備せよ.

- UNIX-ライクな開発環境 (`sh`,`mv`,`cp`,`rm`,`chmod`,`cat`,`sed`,`awk`,`perl`,`make`,`...`)
- yacc or bison パーサジェネレータ
- lex or flex レキシカルアナライザジェネレータ
- clang or gcc c99/c++11 コンパイラ
- boost C++ LIBRARIES
- PARI/GP 計算代数システム (ビルドには不要)
- `opcount` のソースコード

ソースコードが展開された (Makefile のある) ディレクトリにてコマンドライン (シェル) より以下を入力せよ.

```
make
```

opcount は適切に設定された次の環境下で正しくビルドできる事が確認されている.

環境	バージョン	アーキテクチャ	Note
FreeBSD	10.1	amd64	yacc/lex/clang
CentOS	6.8	x86_64	bison/flex/gcc

オプション

- -t 以降指定されたファイルを postamble として出力する.
- -v 指定の位置にバージョン情報を記述した式が出力される.
- -p 結果を表示する postamble が自動生成される.

暗号方式記述言語

暗号方式記述言語 (Cryptographic Scheme Description Language, CSDL) はペアリングに基づく暗号方式のデータフロー解析を目的としたドメイン特化言語で, その文法はプリプロセッサの無い C 言語 (C89) と概ね同じ (形式的文法はほとんど一緒) である.

CSDL のコメントは C++ 言語のコメントとだいたい同じで, /* で始まり */ で終わるか, // で始まり次の改行で終わるかのどちらかである.

CSDL と C 言語の文法上の最も大きな違いは ^ 演算子の解釈である. ^ 演算子は C 言語では bitwise xor 演算子を表現し算術演算より弱い優先度 を持つ. 一方 CSDL では ^ 演算子は TeX のようにべき乗を表現し, どの算術演算より強い優先度を持ち右結合する. xor 演算を記述する場合は xor または (+) と書く. (xor 代入演算の場合は xor_eq または (+)=).

CSDL ではアルゴリズムは関数の中に記述する.

```
main(){
    ...
}
```

関数名は main でなくても良い. 同名でなければいくつ定義しても良い. 例えば:

```
setup(){ ... }
sign(){ ... }
verify(){ ... }
```

データ型

C の組み込みデータ型の他に, CSDL は次のデータ型を持つ.

型名	意味
`group`	ペアリングのソース群 $\mathbb{G}$
`group0`	非対称ペアリングのソース群 $\mathbb{G}_0$
`group1`	非対称ペアリングのソース群 $\mathbb{G}_1$
`target`	ペアリングの標的群
`integer`	整数 (上記群の自己準同型環 (の可換部分環))
`list`	リストデータ型

変数の宣言は次のように記述する. (宣言文).

```
group0   A,B,C ;
group1   X,Y,Z ;
integer alpha, beta, gamma, delta ;
target  gT ;
```

変数は宣言時に初期化する事も出来る (初期化文). 例えば

```
group1 W = X*Y*Z ;
```

宣言文や初期化文は C++ や C99 のように, スコープ内で同名の変数が既に宣言されていない限りは複文中のどの位置でも可能.

変数に値を割り当てるには, 次のように記述する (代入文).

```
X = Y^alpha * Z^beta ;
```

左辺はいわゆる左辺値である必要がある. 例えば

```
X^2 = Y^alpha * Z^beta ;
```

のような代入は出来ない.

リスト値とリスト変数

CSDL と C 言語の間には形式的文法上の違いはほとんど無い. opcount がオペレー ションをカウントするだけの機能しか持たないので, opcount が想定する CSDL の 意味を論ずることにあまり深い意味は無いが,(カンマ) 演算子の解釈の仕方に 大きな違いがある. ,(カンマ) 演算子は C 言語では左から右に評価され, 左の式の値は捨てられる. 一方 CSDL では左の式の値は捨てられないと想定されている. ,(カンマ) 演算子の優先度は C 言語と同様に最も低く設定されているので, リスト 値に何らかの操作を行う場合は

```
(a, b, c)
```

のようにカッコで括る. リスト値の要素が全て左辺値なら, そのリスト値は左辺値 となる. 即ち代入出来る.

```
integer a, b, c, d, e, f ;
(a,b,c) = (d^2, e^3*f, e*d/f) ;
```

リストは入れ子にする事が出来る.

```
integer a, b, c, d, f ;
((a,b),c) = ((d^2, d^3*f), d/f) ;
```

`list` 型の変数 (リスト変数) にはリスト値を代入することが出来る.

```
integer a, b, c ;
list L = (a, b, c) ;
```

関数

関数は C 言語っぽく定義する. 返り値を指定しない場合は `list` と解釈される. プロトタイプ宣言があればその関数を定義に利用できる.

```
integer random() ;

setup(){
  group   g   ;
  integer msk = random() ;
  group   y   = g^msk ;
  list    pp  = (g,y) ;
  return  (pp,msk) ;
}
```

特殊形式

以下の特殊形式が存在する. 同じ名前の変数定義や関数宣言をしてしまうと, そのスコープでは使用出来なくなるので注意が必要である. ついうっかり変数 `e` を宣言してしまう事が良くある.

関数	想定機能
`target e (group0 g0, group1 g1)`	pairing
`group  Repeat (integer n, group  g)`	繰り返し
`target product(integer n, target t)`	積

それぞれオペレーションの数を表すと想定されている適当な関数などに変換されるので, pari/gp 側で該当する関数を定義して使用する.

文	変換	想定機能
`for(x;y;z){ ... }`	`for_statement(x,y,z,body)`	繰り返し
`if(x){ ... }`	`if_statement(x,body,orelse)`	分岐
`inline "s" ;`	`s`	文字列リテラル `"s"` をそのまま出力

`for` 文や `if` 文は, コードが複雑になると, 繰り返し回数の平均値や確率の代数的な表現をプログラムで自動的に解析する事は現実的ではないので `opcount` には, そうした繰り返し回数や確率をユーザが自分で設定できるインターフェイスが用意されている. `opcount` は

```
for(x;y;z){
    body ;
}
```

なる CSDL の文を

```
for_statement("x","y","z",{body})
```

なる計算コストとして計上し,

```
if(x){
    body ;
}else{
    orelse ;
}
```

なる CSDL の文を

```
if_statement("x",{body},{orelse})
```

なる計算コストとして計上する. ここで, `"x","y"` および`"z"` は それぞれ, 正規化された 式 `x,y` および式 `z` を表現する文字列で, `{body}`, `{orelse}` はそれぞれ `body` 節および `orelse` 節の 計算コストである. `if` 文に `else` 節がない場合は `{orelse}` は 0 となる. pari/gp 側に適切な `for_statement()` 関数および `if_statement()` 関数をユーザが定義する事により, 繰り返し回数や確率の代数的な表現を得る事が出来る. `inline` 文は後続の文字列リテラルをそのまま pari/gp 側に渡す. 例えば, 以下のような記述を CSDL の先頭の方に記述しておけば比較的単純なループや条件分岐は処理できるであろう.

```
inline "

  for_statement(A_,B_,C_,D_) = {
    eval(A_);
```

```
    A_=0;
    while(eval(B_),A_++;eval(C_));
    return(A_ * D_) ;
  }

  if_statement(A_,B_,C_) = {
    return (if_prob * B_ + (1 - if_prob) * C_) ;
  }

  ni = 10;
  if_prob = 1/2 ;

" ;
```

例

例 1

hello_world.csdl を次のような内容のテキストファイルとする.

```
group    hello ;
integer  world ;

main(){
  return hello^world ;
}
```

hello_world.csdl を opcount に入力すると次のような出力を得る. (但し先頭が$の行はコマンド入力行を表すとする.)

```
$ ./opcount < hello_world.csdl
call_main = (1)*GROUP_batch(1*1,0)+(1)*LIST_VARIABLE_assign ;
```

csdl 中で関数が定義されると, 関数内の演算コストがカウントされ call_関数名という変数にその値を代入するスクリプトが出力される. 上記の場合は main() 関数が定義されているので, call_main への代入式が出力されている. ここで GROUP_batch(n,b) は group 型の多重冪乗の演算コスト (n, b) を表現している. 計算代数システム gp がこのスクリプトをこのままで 評価することは出来ないが, 例えば preamble に

```
GROUP0_batch(n,b) = n * GROUP0_pow + (n-1+b) * GROUP0_mul ;
```

等と記述しておけば gp が評価可能な出力を得る事ができる. LIST_VARIABLE_assign は return 文によって生成される項で通常無視してよく, 例えば preamble に

```
LIST_VARIABLE_assign = 0 ;
```

等と設定しておく. preamble を次のような内容の gp スクリプトとする.

```
GROUP_batch(n,b)     = n * GROUP_pow   + (n-1+b) * GROUP_mul   ;
GROUP0_batch(n,b)    = n * GROUP0_pow  + (n-1+b) * GROUP0_mul  ;
GROUP1_batch(n,b)    = n * GROUP1_pow  + (n-1+b) * GROUP1_mul  ;
TARGET_batch(n,b)    = n * TARGET_pow  + (n-1+b) * TARGET_mul  ;
pairing_batch(n)     = n * Miller_Loop + (n-1)   * TARGET_mul + Final_Exponentiation ;

LIST_VARIABLE_assign = 0 ;
LIST_LITERAL_assign  = 0 ;
GROUP_assign         = 0 ;
GROUP0_assign        = 0 ;
GROUP1_assign        = 0 ;
TARGET_assign        = 0 ;
INTEGER_assign       = 0 ;
```

preamble と結果を表示する postamble が自動生成される-p オプションを 指定して hello_world.csdl を opcount に入力し, 出力を gp に渡すと 次のような出力を得る.

```
$ ./opcount preamble -p < hello_world.csdl | gp -q
main = GROUP_pow
```

例 2

elgamal.csdl を次のような内容のテキストファイルとする.

```
group   g ;
integer random() ;

list setup(){
  integer x = random() ;
  group   y = g^x ;
  return  (x,y) ;
}

list enc(group y, group m){
  integer r = random() ;
  return  (g^r, m * y^r) ;
}
```

```
group dec(integer x, list c){
  group c1 , c2 ;
  (c1, c2) = c ;
  return c2/c1^x ;
}
```

この例では適当な関数 random() をプロトタイプ宣言している. プロトタイプ 宣言された関数は定義が無くとも, 他の関数の定義に使用できる. elgamal.csdl を opcount と gp で処理すると次のような出力を得る.

```
$ ./opcount preamble -p < elgamal.csdl | gp -q
setup = GROUP_pow + call_random
enc = 2*GROUP_pow + (GROUP_mul + call_random)
dec = GROUP_pow + GROUP_mul
```

定義またはプロトタイプ宣言された関数を, 他の関数の定義の中で呼び出すと call_関数名 としてカウントされる. 例えば上記の random() 関数は call_random としてカウントされている.

例 3

boneh_franklin.csdl を次のような内容のテキストファイルとする.

```
group   g0 ;
group   g1 ;
group   hash2point(char * ID) ;
integer random() ;

setup(){
  integer x = random() ;
  group   y = g1^x ;
  return  (x,y) ;
}

keygen(char * ID, integer x){
  return hash2point(ID)^x ;
}

enc(target m, char * ID, group  y){
  integer r  = random() ;
  group   c1 = g0^r ;
  target  c2 = m * e(hash2point(ID),y^r) ;
  return (c1,c2) ;
}
```

```
dec(list c, group  d_ID){
  group  c1 ;
  target c2 ;
  (c1,c2) = c ;
  target m = c2/e(c1,d_ID) ;
  return m ;
}
```

この例では定義済み関数 e(·,·) (ペアリング) を方式の定義に 使用している. boneh_franklin.csdl を opcount と gp で処理すると 次のような出力を得る. (但し行末の\\は行継続を表すとする.)

```
$ ./opcount preamble -p < boneh_franklin.csdl | gp -q
setup = GROUP_pow + call_random
keygen = GROUP_pow + call_hash2point
enc = 2*GROUP_pow + (call_random + (call_hash2point + (TARGET_mul\\
 + (Miller_Loop + Final_Exponentiation))))
dec = (Miller_Loop + (Final_Exponentiation + TARGET_div))
```

ペアリングはターゲットグループとは独立した演算コストとしてカウントされ, n 個のペアリングの積が pairing_batch(n) としてカウントされる.