Библиотека целочисленной арифметики произвольной точности MPL

К.В. Никулов, <knikulov@yandex.com>

Г.А. Ситкарев, <sitkarev@unixkomi.ru>

Сыктывкарский Государственный Университет
Лаборатория Прикладной Математики и Программирования
http://amplab.syktsu.ru

РЕФЕРАТ

Приводится краткое описание библиотеки целочисленной арифметики произвольной точности и теоретико-числовых алгоритмов MPL, реализованных алгоритмов и её API. Даётся краткое описание используемых методов повышения скорости вычислений.

1. Введение

Криптографические преобразования на основе асимметричных шифров, таких как опубликованный в [RSA78] RSA, обычно оперируют с целыми числами имеющими разрядность выходящую за область значений целочисленных типов, поддерживаемых аппаратно. Языки программирования общего применения обычно не поддерживают арифметику произвольной точности, а лишь отображают свои типы на машинные. В связи с этим поддержка «длинной» арифметики должна реализовываться программно. В представленной работе даётся описание реализации такой программной библиотеки.

Название библиотеки это аббревиатура, образованная из начальных букв «Multiple Precision Library». Помимо основных арифметических операций, таких как «сложение», «вычитание», «умножение», «деление» и «возведение в квадрат», библиотека включает в себя ряд специальных функций модульной арифметики и теоретико-числовых алгоритмов, выполняющих операции «возведение в степень по модулю», «редукция по модулю», «нахождение обратного числа по модулю», «тест Миллера — Рабина числа на простоту» предложенный в [Rabin80], «наибольшй общий делитель». Преследовалась цель сделать библиотеу максимально переносимой и независимой от окружения времени исполнения. В связи с этим для реализации библиотеки был выбран язык Си.

Инициализация числа произвольной точности и управление памятью, выделяемой для хранения его разрядов, осуществляется служебными функциями. Последние выделяют для хранения разрядов блоки памяти, и высвобождают их по завершению жизненного цикла числа произвольной точности.

Библиотека арифметики произвольной точности содержит функции выгрузки разрядов числа в буфер и загрузки их из буфера. Они соответствуют примитивам преобразования данных I2OSP (Integer-to-Octet-String primitive) и OS2IP (Octet-String-to-Integer primitive) из [PKCS1v2].

Иногда числа хранятся или передаются в виде ASCII-строк. Значение числа произвольной точности может быть установлено из ASCII-строки с основанием системы счёта на выбор от 2-х до 36-ти (обычно используется 10 или 16).

2. Особенности реализации

Типы и константы

В заголовочном файле *"mpl.h"* содержится определение структуры **mpl_int**. Эта структура содержит несколько полей:

- адрес блока, хранящего разряды числа произвольной точности;
- знак числа;
- индекс верхего разряда;
- данные, связанные с управлением памятью.

Все поля структуры считаются закрытыми и не предназначены для доступа со стороны прикладных программ напрямую.

Некоторые свойства библиотеки настраивается константами в файле *"mpl_common.h"*:

MPL_INT_BITS
: количество бит задействованных для хранения одного разряда числа: 28 бит для 32-х разрядных процессоров x86;

MPL_INT_BASE
: основание системы счёта для арифметики произвольной точности: для 32-х разрядного процессора x86 это $2^{MPL\_INT\_BITS} = 268435456$;

MPL_INT_MASK
: битовая маска, выделяющая биты в слове, задействованные для хранения одного разряда числа;

MPL_INT_ALLOC_DEFAULT
: количество разрядов в блоке, выделяемых по умолчанию из кучи при инициализации числа;

MPL_INT_APPEND
: количество разрядов, дополнительно выделяемых к блоку из кучи;

Целые числа произвольной точности представляются в виде

$$\pm a = \pm(a_n, a_{n-1}, \ldots, a_0)_\beta,$$

где $a_n, a_{n-1}, \cdots, a_0$ значения разрядов по основанию системы счёта β. Арифметическое значение такого числа образуется суммированием разрядов, как коэффициентов полинома при степенях основания

$$\pm a = \pm a_n\beta^n + a_{n-1}\beta^{n-1} + \ldots + a_0.$$

Большинство функций возвращают значения, по ним программист определяет как завершилась запрошенная операция. Библиотека определяет константы кодов возврата:

MPL_OK
: успешное завершение операции;

MPL_ERR
: неверный аргумент или ошибка;

MPL_NOMEM
: не удалось выделить память в куче;

MPL_COMPOSITE
: число не является простым.[1]

Принятые соглашения

В функциях библиотеки произвольной точности выходные аргументы всегда располагаются перед входными аргументами. Такое соглашение имитирует поведение оператора присвоения языков программирования и соответствует порядку в алгебраической записи. Все функции и типы, определённые в библиотеке, имеют префикс ‘‘*mpl_**’’, все константы имеют префикс ‘‘*MPL_**’’.

```
/* Помножить a на b, сохранить результат в c: c = a * b. */
mpl_mul(c, a, b);
```

Большинство функций позволяют использовать входные аргументы как выходные. В этом случае программисту нет необходимости держать временные переменные для хранения результатов промежуточных вычислений, что весьма удобно.

```
/* Возвести a в квадрат. */
mpl_mul(a, a, a);
```

Выделение памяти

Числа произвольной точности должны быть предварительно инициализированы функциями *mpl_init*() или *mpl_initv*(). Блоки памяти для хранения разрядов выделяются функциями библиотеки динамически, по необходимости. Программист должен позаботиться только о том чтобы освободить ресурсы тогда, когда числа больше не нужны. Для этого библиотека предоставляет функции *mpl_clear*() и *mpl_clearv*().

mpl_initv() и *mpl_clearv*() получают переменное число аргументов. Для обозначения конца переменных аргументов, в последний аргумент помещается NULL.

```
mpl_int a, b, c;
/* Инициализировать переменные a, b и c. */
mpl_init(&a);
mpl_initv(&b, &c, NULL);
```

Когда переменные больше не нужны, их ресурсы нужно высвободить.

```
/* Высвободить ресурсы занимаемые переменными a, b и c. */
mpl_clear(&a);
mpl_clearv(&b, &c, NULL);
```

Для выделения памяти из кучи библиотека использует функции стандартной библиотеки Си *malloc*(), *realloc*() и *free*().

3. Замечания по реализации

Далее приводятся замечания по реализации некоторых функций библиотеки арифметики произвольной точности. Все используемые алгоритмы широко известны, а их теоретические аспекты неоднократно исследовались и публиковались в различных источниках. В практических же их реализациях в том или ином алгоритме даже в распространённых библиотеках

[1] Это значение может возвращать только *mpl_primality_miller_rabin*().

встречаются неточности и ошибки, которые зачастую вызваны буквальным пониманием теоретического описания или опубликованного псевдокода.

3.1. Умножение

Реализация операции умножения в библиотеке фактически включает в себя три различных алгоритма:

1. классический алгоритм умножения;
2. умножение по методу [Comba90];
3. умножение по методу А.А. Карацубы, опубликованного в [Karatsuba62].

Все три метода имеют ограничения и обладают разными свойствами. Реализация автоматически выбирает наиболее подходящий метод и выполняет умножение, используя его. Несколько констант, определённых в библиотеке, задают пороговые значения для алгоритмов умножения, по которым реализация выбирает тот или иной метод:

MPL_COMBA_STACK
: размер массива для хранения временных значений в методе Comba;

MPL_COMBA_DEPTH
: максимальное количество складываемых разрядов при умножении по методу Comba;

MPL_KARATSUBA_CUTOFF
: пороговое значение для количества разрядов в минимальном из перемножаемых чисел, при котором будет выбран метод А.А. Карацубы.

Реализация пытается выбрать наиболее подходящий метод из возможных, начиная с метода Карацубы. Если порог MPL_KARATSUBA_CUTOFF не достигнут минимальным из операндов, проверяется возможность использования метода Comba, и в случае если это не осуществимо, алгоритм переходит к классическому «школьному» умножению.

Отметим, что пороговое значение MPL_KARATSUBA_CUTOFF установлено экспериментально, путём измерения процессорного времени, занимаемого операциями метода для различного количества разрядов перемножаемых чисел. Дело в том что метод Карацубы даёт практическое преимущество лишь при достижении определённого количества перемножаемых разрядов. Это обусловлено промежуточными расходами в функции, реализующей метод, связанные с организацией стека вызовов и локальных переменных. При существенном количестве разрядов (2048 бит и выше), метод Карацубы даёт весьма существенный выигрыш в скорости умножения.

3.2. Возведение в квадрат

Возведение в квадрат так же, как и умножение, включает в себя три метода:

1. модификация классического алгоритма умножения;
2. модификация умножения по методу [Comba90];
3. модификация умножения по методу [Karatsuba62].

Все три метода используют тот факт, что при возведении в квадрат умножаются два одинаковых числа. Это позволяет сократить количество выполняемых операций и ускорить выполнение действия, по сравнению с умножением двух чисел методами без модификаций.

3.3. Возведение в степень по модулю

Скорость операции возведения в степень по модулю существенно сказывается на производительности алгоритмов асимметричного шифрования, таких как RSA. Наивная реализация такого алгоритма в буквальном смысле

$$c = x^a \bmod y$$

практически реализуема только для малых значений a, в то время как в алгоритме RSA к примеру примитив расшифрования использует значение экспоненты в тысячи бит (от 1024 и выше). Если число a состояло хотя бы из 1024 бит, при наивной реализации уже на первом шаге алгоритма промежуточный результат возведения в степень уже насчитывал бы $1024 \times 2 = 2048$ бит. Всего бы для хранения такого числа потребовалось

$$\log_2 \left(2^{1024} - 1\right)^{2^{1024}-1} = (2^{1024} - 1) \cdot \log_2 \left(2^{1024} - 1\right) \approx (2^{1024} - 1) \cdot 1024 \approx 2^{1034}$$

бит. Очевидно, что такой метод вряд ли стоит применять на практике.

Если представить показатель степени a в двоичном виде из n двоичных разрядов, как полином по основанию степеней двоек, на что мы имеем полное право, так как в любом случае все целые числа таким образом представимы, а фактически разряды числа в машинном представлении соответствуют этому виду, то получим:

$$\begin{aligned} x^a \bmod y &= x^{a_0 2^0 + a_1 2^1 + a_2 2^2 + \ldots + a_{n-1} 2^{n-1}} \bmod y \\ &= 1 \cdot x^{a_0 2^0} \cdot x^{a_1 2^1} \cdot \ldots \cdot x^{a_{n-1} 2^{n-1}} \bmod y, \end{aligned}$$

где $a_0, a_1, \ldots, a_{n-1}$ принимают значение 1 или 0. Если алгоритм будет поддерживать на i-ом шаге, начиная с $i = 0$, степень x^{2^i}, то достаточно будет проверить бит a_i и выполнить одно умножение если $a_i = 1$. Псевдокод, представленный ниже, иллюстрирует метод возведения x в степень a по модулю y.

```
mod_exp(x, a, y)
{
    if (a & 0x1)
        c = x;
    else
        c = 1;
    a >>= 1;
    tmp = x;
    while (a > 0) {
        tmp = tmp^2 mod y;
        if (a & 0x1)
            c = (c * tmp) mod y;
        a >>= 1;
    }

    return c;
}
```

Такой метод для возведения числа в степень по модулю, называемый *быстрым возведением в степень*, в худшем случае выполнит до $2n$ умножений. Количество операций умножения можно сократить, если заранее вычислить множители для всех возможных комбинаций некоторого числа бит показателя степени a. Если бы мы выбрали количество бит равное k, тогда, заранее составив таблицу всех возможных значений окна из k бит $(x^0, x^1, x^2, \ldots, x^{2^k-1})$ *mod* y, мы могли бы за один раз сразу выбрать k бит из показателя степени a. Если взять округление до ближайшего большего целого $m = \left\lceil \frac{n}{k} \right\rceil$, то для k бит показателя степени тогда:

$$x^a \ mod \ n = x^{b_0 + b_1 2^k + b_2 2^{2k} + b_3 2^{3k} + \ldots + b_{m-1} 2^{(m-1)k}} \text{ mod } n$$
$$= x^{b_0} x^{b_1 2^k} x^{b_2 2^{2k}} x^{b_3 2^{3k}} \ldots x^{b_{m-1} 2^{(m-1)k}} \text{ mod } n.$$

Такой метод называется *оконным методом возведения в степень*, так как он использует окно в k бит, которое пробегает по всем битам показателя степени a. Функция *mpl_mod_exp*() основана на этом методе, с небольшими оптимизациями, в частности, предвычисляются множители x^z только для значений показателя степени z с установленным старшим битом, т.е. от $x^{2^{k-1}}$ до x^{2^k-1}. Это позволяет уменьшить размер памяти, занимаемый предвычисляемыми значениями x^z, в два раза. Совершенно ясно, что бинарный алгоритм является по сути всего лишь частным случаем оконного, так как в алгоритме быстрого возведения в степень полагается $k = 1$.

```
mod_exp_windowed(a, e, b)
{
    w[0] = a;
    for (i=1; i < (2^k - 1); i++)
        w[i] = (a * w[i-1]) mod b;

    c = 1;

    while (e > 0) {

        z = e & (2^k - 1);
        c = (c * w[z]) mod b;

        for (i = 0; i < k; i++)
            c = c^2 mod b;

        e >>= k;
    }

    return c;
}
```

Редукция по модулю

Алгоритм взятия остатка от деления по модулю

$$c = a \text{ mod } b$$

может быть реализован эффективнее, чем просто взятие остатка от операции деления. В основе этого метода, предложенного в [Barrett86], лежит следующая аппроксимация для вычисления результата деления a на b:

$$\frac{a}{b} \approx \left\lfloor \left(a \cdot \left\lfloor \frac{2^q}{b} \right\rfloor \right) \cdot \frac{1}{2^q} \right\rfloor.$$

Если деление на b нужно делать много раз, то взяв достаточно большое 2^q, и вычислив только один раз $\lfloor 2^q/b \rfloor$, можно получать достаточно хорошую аппроксимацию к $\frac{a}{b}$, используя при этом только битовые сдвиги и умножение. Выигрыш очевиден, так как операция деления задействуется в чистом виде только один раз.

Взяв за основу этот метод, полагая что β есть основание системы счисления, a занимает максимально $2m$ разрядов, а b занимает максимум m разрядов, аппроксимация модульной редукции $c = a \ mod \ b$ представима в следующем виде:

$$c \approx a - b \cdot \left\lfloor \left(\frac{a}{\beta^{m-1}} \cdot \left\lfloor \frac{2^q}{b} \right\rfloor \right) \cdot \frac{\beta^{m-1}}{2^q} \right\rfloor .$$

Если положить $2^q = \beta^{2m}$, и обозначить как $\mu = \lfloor 2^q / b \rfloor$, то гарантируется выполнение соотношения:

$$c \approx a - b \cdot \left\lfloor \left\lfloor \frac{a}{\beta^{m-1}} \right\rfloor \cdot \mu \cdot \frac{1}{\beta^{m+1}} \right\rfloor < 3b.$$

Собственно, реализация алгоритма разделена на две функции: *mpl_barret_reduce_setup*() подготавливает параметр μ, а *mpl_barrett_reduce*() выполняет редукцию с его использованием. В реализации есть несколько оптимизаций, например при перемножении μ и $\lfloor a/\beta^{m-1} \rfloor$ младшие $m-1$ разрядов не вырабатываются, так как далее следует операция сдвига на $m+1$ разрядов вправо, а сдвигаемые младшие $m+1$ разрядов в любом случае теряются.

3.4. Деление

Алгоритм деления основан на классическом алгоритме, который известен всем со школы. Его реализация основана на алгоритме 14.20 из [Handbook96]. Из всех операций арифметики произвольной точности деление самая затратная и сложная, потому всегда стремятся избежать её применения, если это возможно.

Перед началом основного цикла алгоритма, делимое $u = (u_n, u_{n-1}, \ldots, u_0)_\beta$ и делитель $v = (v_t, v_{t-1}, \ldots, v_0)_\beta$ нормализуются сдвигом разрядов влево, для того чтобы выполнялось условие $v_t \geq \beta / 2$; здесь и далее $\beta = MPL\_INT\_BASE$. Аппроксимация $\hat{q} \approx q$ для значения разряда частного q_{i-t-1} на текущей итерации i выполняется в несколько шагов:

1. Если самый старший разряд делимого u_i и делителя v_t совпадают, то $\hat{q} = \beta - 1$.
2 В противном случае, $\hat{q} = \lfloor (u_i \beta + u_{i-1}) / v_t \rfloor$.
2 Пока выполняется условие $\hat{q} \times (v_t \beta + v_{t-1}) > u_i \beta^2 + u_{i-1} \beta + u_{i-2}$, аппроксимация разряда частного $\hat{q}$ уменьшается на единицу.

После чего гарантируется, что $\hat{q}$ будет максимум больше на единицу чем действительное значение q текущего разряда частного, формально $q \leq \hat{q} \leq q + 1$. Если $\hat{q}$ окажется равным $q + 1$, это будет обнаружено по отрицательному значению делимого u сразу же после того как из него было вычтено $\hat{q}v$. В этом случае u корректируется, а в q_{i-t-1} записывается $\hat{q} - 1$. В конце алгоритма остаток денормализуется, т.е. сдвигается на столько бит вправо, на сколько были сдвинуты делитель или делимое влево.

4. Заключение

Библиотека готова к практическому использованию в составе различных криптографических систем и исследовательских проектов. Компактность библиотеки позволяет использовать её во встраиваемых системах. На её основе были получены рабочие прототипы алгоритма шифрования RSA с дополняющей схемой, описанной в [OAEP95]. Тестовые вектора сверялись с представленными в [PKCS1v2]. Планируется перенос библиотеки на FreeBSD, NetBSD, OpenBSD, MacOS и Windows. Исходные тексты библиотеки доступны на репозитории subversion в [MPLsvn] и для просмотра через Web по адресу [MPLweb].

Ссылки

[RSA78]
Rivest, R.; A. Shamir; L. Adleman (1978). «A method for obtaining digital signatures and public-key cryptosystems», *Communications of the ACM 21 (2): 120-126.*

[Rabin80]
Michael O. Rabin, «Probabilistic algorithm for testing primality», *Journal of Number Theory, Volume 12, Issue 1, February 1980, Pages 128-138.*

[PKCS1v2]
RSA Laboratories, «PKCS #1 v2.1: RSA Cryptography Standard», *June 14, 2002.*

[Comba90]
Comba, P. G., «Exponentiation cryptosystems on the IBM PC», *IBM Systems Journal, Vol. 29, No. 4, December 1990.*

[Karatsuba62]
Карацуба А., Офман Ю. «Умножение многозначных чисел на автоматах», *Доклады Академии Наук СССР. — 1962. — Т. 145. — № 2.*

[Barrett86]
P.D. Barrett, «Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor», *Advances in Cryptology — CRYPTO'86, Springer, 1986.*

[Handbook96]
Alfred J. Menezes, Scott A. Vanstone, Paul C. Van Oorschot. *Handbook of Applied Cryptography (1st ed.). 1996. CRC Press, Inc., Boca Raton, FL, USA.*

[OAEP95]
M. Bellare, P. Rogaway. «Optimal Asymmetric Encryption — How to encrypt with RSA», *Extended abstract in Advances in Cryptology — Eurocrypt '94 Proceedings, Lecture Notes in Computer Science Vol. 950, A. De Santis ed, Springer-Verlag, 1995.*

[MPLsvn]
https://github.com/knikulov/libmpl

[MPLweb]
http://github.com/knikulov/libmpl