

Поиск дискретного логарифма

Сергей Николенко

Криптография — CS Club, осень 2009

Outline

- 1 \sqrt{n} -методы
 - Введение. Атака на гладкие модули
 - Алгоритм Шенкса и ρ -метод Полларда
 - λ -метод Полларда
- 2 Алгоритмы index calculus: первая фаза
 - Введение. Основная идея
 - Проверка на гладкость одного числа
 - Проверка на гладкость многих чисел

Задача

- На прошлой лекции мы узнали, как раскладывать числа на множители.
- Теперь попробуем решать другую базовую задачу криптографии: дискретный логарифм.

Задача

- На прошлой лекции мы узнали, как раскладывать числа на множители.
- Теперь попробуем решать другую базовую задачу криптографии: дискретный логарифм.
- *Дискретный логарифм*: в циклической группе G по $g \in G$ и $y \in G$ найти такой x , что $g^x = y$.
- Этот x определяется с точностью до порядка g ; если $\langle g \rangle = G$, то логарифм определён с точностью до $|G| = n$. Мы будем считать, что $\langle g \rangle = G$.

Сложность общей задачи

- Известно, что если не пользоваться ничем, кроме групповой операции и взятия обратного, то ничего лучше, чем \sqrt{n} , не будет: когда алгоритм обращается за определёнными умножениями, можно по ходу строить группу так, что ему придётся обращаться $\Omega(\sqrt{p})$ раз, где p — наибольший простой делитель n [Shoup, 1997].
- Мы сначала рассмотрим методы, достигающие этой цели, а потом перейдём к специфически числовым методам, работающим не во всех группах.

Тривиальный подход

- Тривиальный подход: возводить g, g^2, g^3, \dots , пока не наткнёмся на y .
- Требуется примерно $\frac{n}{2}$ операций, имеет смысл только для маленьких n .

Атака на гладкие модули

- Пусть $n = p_1^{k_1} p_2^{k_2} \dots p_l^{k_l}$.
- Заметим, что для каждого из этих p порядок элемента g^{n/p^k} равен p^k , и порядок элемента y^{n/p^k} не превосходит p^k .
- Иначе говоря, g^{n/p^k} порождает подгруппу G порядка p^k , а y^{n/p^k} лежит в этой подгруппе.
- И если мы можем найти логарифм в этой подгруппе:

$$\left(g^{n/p^k}\right)^{x'} = y^{n/p^k}, \text{ то, с другой стороны,}$$

$$\left(g^{n/p^k}\right)^x = y^{n/p^k}, \text{ и тем самым}$$

$$x' \equiv x \pmod{p^k}.$$

Атака на гладкие модули

- Тогда, если мы найдём логарифмы по модулям простых чисел

$$\begin{aligned} \left(g^{n/p_1^{k_1}} \right)^{x_1} &= y^{n/p_1^{k_1}}, \\ \left(g^{n/p_2^{k_2}} \right)^{x_2} &= y^{n/p_2^{k_2}}, \\ &\vdots \\ \left(g^{n/p_l^{k_l}} \right)^{x_l} &= y^{n/p_l^{k_l}}, \end{aligned}$$

то сможем по китайской теореме об остатках восстановить x , потому что

$$\begin{aligned} x &\equiv x_1 \pmod{p_1^{k_1}}, \\ x &\equiv x_2 \pmod{p_2^{k_2}}, \\ &\vdots \\ x &\equiv x_l \pmod{p_l^{k_l}}. \end{aligned}$$

Атака на гладкие модули

- Оказывается, найти логарифм по модулю p^k для маленького простого p легко, даже если k большое.
- Разложим предполагаемый логарифм x' по основанию p :

$$x' = z_0 + z_1p + z_2p^2 + \dots + z_kp^k.$$

- Положим сначала $y_0 = y^{n/p}$, $g_0 = g^{n/p}$. Порядок g_0 не больше p , значит,

$$y_0 = y^{n/p} = g^{x \cdot n/p} = g_0^x = g_0^{z_0}.$$

- Тем самым мы нашли z_0 . Теперь можно его вычесть, положить $y_1 = (yg_0^{-z_0})^{n/p^2}$ и продолжать.
- В итоге найдём логарифм по модулю p^k за k поисков логарифма по модулю p .

Атака на гладкие модули

- Значит, гладкие модули использовать нельзя.
- Нужно выбирать такие n , у которых есть большие простые делители.
- Либо, в крайнем случае, разложение n неизвестно, но есть причины полагать, что большие простые делители есть.

Baby-step–Giant-step

- Shanks, 1973: алгоритм, работающий за $O(\sqrt{n})$; стандартный time-space tradeoff.
 - 1 Запишем x в виде $x = im + j$ для какого-то m . Тогда $y \cdot (g^{-m})^i = g^j$.
 - 2 Предвычислим g^j и будем перебирать i , умножая y на g^{im} и проверяя, нет ли его среди g^j .
- Если записать g^j в хеш-таблицу, можно считать, что проверка на равенство происходит в среднем за константное время.

Baby-step–Giant-step

- Алгоритм записывает два массива. Первый (giant steps):

$$S = \left\{ \left(i, g^{i \lceil \sqrt{n} \rceil} \right) \mid i = 0.. \lceil \sqrt{n} \rceil \right\}.$$

- Второй (baby steps):

$$T = \left\{ \left(j, y \cdot g^j \right) \mid j = 0.. \lceil \sqrt{n} \rceil \right\}.$$

- Как только списки пересекутся, логарифм можно будет найти как

$$\log_g y \equiv i \lceil \sqrt{n} \rceil - j \pmod{n}.$$

- Однако этот алгоритм требует экспоненциальной памяти.

ρ -метод Полларда

- Pollard, 1978. Суть — «birthday paradox»: мы выбираем псевдослучайную последовательность элементов в группе и ждём цикла. Цикл будет в среднем через $O(\sqrt{n})$ элементов.
- Разобьём группу на три части (не подгруппы) S_1, S_2, S_3 . Будем вычислять

$$a_{i+1} = \begin{cases} y \cdot a_i, & \text{если } a_i \in S_1, \\ a_i^2, & \text{если } a_i \in S_2, \\ g \cdot a_i, & \text{если } a_i \in S_3. \end{cases}$$

ρ -метод Полларда

- Если в последовательности найдётся цикл, это с большой вероятностью приведёт к тому, что мы найдём дискретный логарифм, потому что мы найдём соотношение вида $g^a y^b = g^c y^d$.
- Но, казалось бы, всё равно надо хранить всю последовательность, и с памятью лучше не становится. Что делать?

Алгоритм Флойда для поиска цикла

- Алгоритм Флойда, он же «tortoise-and-hare algorithm».
- Общая постановка: хотим найти цикл в последовательности $a_{i+1} = f(a_i)$.
- Давайте будем хранить всего два указателя: u и v , причём $u_i = a_i$ (черепаха), а $v_i = a_{2i}$ (заяц).
- Если в последовательности есть цикл периода r , начинающийся с позиции s ($a_i = a_{i+r}$ для $i \geq s$), то для любого $i \geq s$, делящегося на r , $a_i = a_{2i}$.
- Т.е. нам придётся искать не более чем на длину периода (т.е. примерно вдвое) дольше.

Алгоритм Brenta

- Другой алгоритм для того же самого — алгоритм Brenta.
- Теперь черепаха останавливается на степенях двойки, а заяц прыгает шаг за шагом к следующей степени.
 - 1 Пока $tortoise \neq hare$:
 - если $i == pow$, то
 $tortoise := hare$
 $pow := 2 \cdot pow$
 $i := 0$
 - $hare = f(hare)$
 - $++ i$
- Шагов в любом случае не больше, чем в алгоритме Флойда, но каждый шаг — это одно вычисление f , а не три.

λ -метод Полларда

- Раньше были зайцы и черепахи, теперь — кенгуру.
- λ -метод Полларда ещё называется «kangaroo method».
- Предположим, что мы знаем некий интервал $[a, b]$, на котором должен лежать неизвестный логарифм x .
- Как это использовать?

λ -метод Полларда

- Определим хеш-функцию h , делящую G на r множеств S_1, S_2, \dots, S_r : $S_i = h^{-1}(i)$.
- Поставим каждому множеству в соответствие расстояние d_1, d_2, \dots, d_r и длину прыжка $g^{d_1}, g^{d_2}, \dots, g^{d_r}$.
- Теперь путь кенгуру определяется как

$$c_{i+1} = c_i \cdot g^{d_{h(c_i)}}.$$

λ -метод Полларда

- Нам будут нужны два кенгуру: дикий и ручной.
- Ручной кенгуру начнёт прыгать из какой-нибудь точки внутри интервала $[a, b]$, например, $g^{\frac{a+b}{2}}$.
- Дикий кенгуру начнёт прыгать из неизвестной точки u .
- Однако, суммируя d_j , мы можем хранить общее пройденное расстояние для обоих кенгуру.

λ -метод Полларда

- Когда ручной и дикий кенгуру встретятся, причём ручной пройдёт к тому времени расстояние t , а дикий — расстояние w , у нас получится, что

$$g^{\frac{a+b}{2}} g^t = g^x g^w, \text{ и } x = \frac{a+b}{2} + t - w.$$

- Пересечение можно найти, например, храня только $t_1, t_2, t_4, t_8, \dots$ и $w_1, w_2, w_4, w_8, \dots$, потому что после пересечения пути кенгуру сойдутся навсегда.
- В результате (без доказательства) ожидаемое время работы получается $O(\sqrt{b-a})$.

ρ - и λ -методы

- Почему ρ - и λ -методы названы этими буквами?

ρ - и λ -методы

- Почему ρ - и λ -методы названы этими буквами?
- Потому что то, что происходит в алгоритмах, похоже на эти буквы:
 - ρ -метод строит последовательность элементов, которая в какой-то момент возвращается к одному из промежуточных значений, создавая цикл;
 - λ -метод строит две последовательности элементов, которые в какой-то момент сливаются и затем совпадают.

Outline

- 1 \sqrt{n} -методы
 - Введение. Атака на гладкие модули
 - Алгоритм Шенкса и ρ -метод Полларда
 - λ -метод Полларда
- 2 Алгоритмы index calculus: первая фаза
 - Введение. Основная идея
 - Проверка на гладкость одного числа
 - Проверка на гладкость многих чисел

От общих групп к частным случаям

- Algorithms better than the above-described rather simple mappings, for general groups are unknown.
- However, it is possible to do better with additional assumptions on the structure of the group.
- They are performed, in particular, in groups of numbers \mathbb{Z}_p .

От общих групп к частным случаям

- Предположения простые: можно выбрать разумную базу факторизации p_1, \dots, p_s , для которой многие элементы будут представляться в виде

$$r = p_1^{k_1} p_2^{k_2} \dots p_s^{k_s}.$$

- Для чисел это легко: берём простые числа, меньшие B ; «многие» — это в точности B -гладкие элементы.
- В дальнейшем будем считать, что мы работаем над \mathbb{Z}_p .

Общая идея index calculus

- Алгоритм index calculus очень похож на алгоритм факторизации, использующий квадратичное решето.
- Так что заодно в каком-то смысле и повторим прошлую лекцию.
- Мы знаем свойства логарифма, а именно

$$\log_g(ab) = \log_g a + \log_g b,$$

$$\log_g(a^e) = e \log_g a.$$

Общая идея index calculus

- Общая идея: логарифм гладкого элемента можно представить как

$$\log_g r \equiv k_1 \log_g p_1 + k_2 \log_g p_2 + \dots + k_s \log_g p_s \pmod{p-1}.$$

- Если мы знаем $\log_g r$ (например, сами выбрали u и вычисляли $r = g^u$) и наберём достаточно много таких соотношений, у нас получится линейная система на $\log_g p_i$.
- Её можно решить и найти $\log_g p_i$, а затем с их помощью найти $\log_g y$.

Общая идея index calculus

- Итак, получаются три фазы.
 - 1 Найти достаточно много соотношений на $\log_g p_i$.
 - 2 Решить линейную систему.
 - 3 Найти логарифм интересующего нас y , зная логарифмы p_i .
- Линейные системы будем решать так же, как в алгоритме факторизации.
- А остальные фазы сейчас рассмотрим.

Гладкие числа

- Нам нужно выбрать границу гладкости B , а затем найти кучу соотношений на $\log_g p_i$, $p_i \leq B$, при помощи гладких чисел u .
- Иначе говоря, нужно проверить кучу чисел на гладкость.
- Мы начнём с методов проверки индивидуальных чисел на гладкость (тоже пригодится), а потом вспомним метод полиномиального решета.

Метод Полларда

- Если просто проверять на B -гладкость перебором, сложность будет порядка $O(\pi(B))$.
- Можно воспользоваться методом, очень похожим на ρ -метод Полларда: определим последовательность чисел

$$a_{i+1} \equiv a_i^2 + 1 \pmod{n}, \text{ где } n \text{ — интересующее нас число.}$$

- По birthday paradox, она начнёт повторяться в среднем через $O(\sqrt{n})$.
- Более того, если у n есть простой делитель q , то в среднем через $O(\sqrt{n})$ начнёт повторяться последовательность $a_i \pmod{q}$.

Метод Полларда

- Мы не знаем q , но можем проверять просто каждый раз a_i и a_{2i} , не даёт ли

$$\gcd(n, a_{2i} - a_i) \text{ или } \gcd(n, a_{2i} + a_i)$$

чего-нибудь интересного. При таком подходе мы ожидаем найти делитель n за $O(\sqrt{q})$, где q — наименьший простой делитель n .

- Значит, на гладкость проверить ожидаем за $O(\sqrt{B})$; если через $O(\sqrt{B})$ шагов совпадений не найдено, можно просто предположить с большой вероятностью, что не гладкое.

Алгоритм Ленстры

- Мы знаем эффективные алгоритмы разложения чисел на множители.
- У нас были алгоритмы, работающие за время $L_n \left[\frac{1}{2}; \sqrt{2} \right]$ и даже $L_n \left[\frac{1}{2}; 1 \right]$.
- Но непонятно, как их обобщить так, чтобы оценка зависела от размера простых делителей (от B), а не от n .

Алгоритм Ленстры

- Алгоритм Ленстры (ECM, elliptic curve method) делает как раз это. Он основан на эллиптических кривых, и мы его разбирать не будем.
- Важно, что работает он за время

$$O\left(e^{\sqrt{(2+o(1)) \log B \log \log B}} (\log n)^2\right) = L_B \left[\frac{1}{2}; \sqrt{2}\right].$$

Итоги

- Итак, у нас есть два разумных подхода к проверке *одного* числа на гладкость:

- метод Полларда проверяет на B -гладкость за $O(\sqrt{B})$;
- ЕСМ проверяет на B -гладкость за

$$L_B \left[\frac{1}{2}; \sqrt{2} \right] = O \left(e^{\sqrt{(2+o(1)) \log B \log \log B}} \right).$$

Задача

- Нам нужно на первой фазе породить много соотношений вида

$$\log_g r = k_1 \log_g p_1 + k_2 \log_g p_2 + \dots + k_s \log_g p_s, \quad p_i \leq B.$$

- Для этого нужно проверить массу чисел на B -гладкость. Вообще говоря, мы должны выбрать много случайных u , а потом проверить $g^u \pmod{p}$ на B -гладкость.

Квадратичное решето

- Мы для подобной задачи знаем метод квадратичного решета.
- Рассмотрим последовательность $Q(x) = x^2 - n$ для $x = x_0 = \lceil \sqrt{n} \rceil, x_0 + 1, \dots$
 - Если n — квадрат по модулю p , то $x^2 - n \equiv 0 \pmod{p}$ iff $x \equiv a$ или $b \pmod{p}$, где a и b — корни из n по модулю p .
 - Если n — не квадрат \pmod{p} , то делиться никогда не будет.
- Значит, можно просто так же вычёркивать те $Q(x)$, для которых x делится на a или b .
- Причём этот алгоритм можно применить к любому многочлену (нам нужны будут квадратичные и линейные).
- Сложность этого алгоритма:
 $O(\pi(B)(1 + \log B)^{o(1)} + N \log \log B)$, где N — количество проверяемых чисел.

Проблема

- Но сейчас у нас не всё так просто.
- Если выбирать u , то g^u , которые нужно проверять на гладкость, не похожи ни на какой многочлен, и так просто всё не получится.
- Как обойти эту проблему?

Решение

- Рассмотрим $H = \lceil \sqrt{p} \rceil$ и будем рассматривать последовательность $(H + c_1)(H + c_2)$ для маленьких c_1 и c_2 .
- Тогда для $p_i \leq B$ получаются соотношения вида

$$\log_g((H + c_1)(H + c_2)) = k_1 \log_g p_1 + k_2 \log_g p_2 + \dots + k_s \log_g p_s.$$

- Если $H^2 = p + J$, то

$$(H + c_1)(H + c_2) \equiv J + (c_1 + c_2)H + c_1 c_2 \pmod{p},$$


и это линейный многочлен, к которому можно применить решето (если в каждый конкретный момент фиксировать c_1 и варьировать c_2).

- Но ведь мы по прежнему не знаем $\log_g((H + c_1)(H + c_2))$, и отдельных $\log_g(H + c_1)$ тоже не знаем. :) Чем же нам стало лучше?

Решение

- Нам стало лучше тем, что теперь с одними и теми же c_1 и c_2 получаются сразу много соотношений!
- Мы просто добавляем $\log_g(H + c_i)$ как новые неизвестные.
- Но количество уравнений растёт быстрее, чем количество неизвестных, и на практике получается, что для базы B нужно не больше $4\pi(B)$ уравнений.
- А затем мы их решим при помощи алгоритма Видеманна, за время $\pi(B)^2$.
- Будем варьировать $0 \leq c_1 < c_2 \leq C$, C выберем позже.

Спасибо за внимание!

- Lecture notes и слайды будут появляться на моей homepage:
<http://logic.pdmi.ras.ru/~sergey/>
- Присылайте любые замечания, решения упражнений, новые численные примеры и прочее по адресам:
sergey@logic.pdmi.ras.ru, snikolenko@gmail.com
- Заходите в ЖЖ  [smartnik](#).