Acta Universitatis Apulensis No. 28/2011
ISSN: 1582-5329 pp. 27-32

ON A PROBABILISTIC ALGORITHM SOLVING DISCRETE LOGARITHM PROBLEM

Murat Sahin and Ali Bulent Ekin

Abstract. Recently, Gadiyar et al. presented a probabilistic algorithm solving discrete logarithm problem over finite fields. In this paper, we compare the running time of this algorithm with Pollard's rho algorithm and we improve the required memory of the algorithm as a negligable memory by using some collision detection algorithms.

2000 *Mathematics Subject Classification*: 11Y16.

1. Introduction

Let p be an odd prime and g be a primitive root. For given an integer $y \in (\mathbb{Z}/p\mathbb{Z})^*$, discrete logarithm problem (DLP) for $(\mathbb{Z}/p\mathbb{Z})^*$ is to find the least non-negative integer x such that $g^x \equiv y\,(modp)$. When p is sufficiently large, DLP is believed to be hard. The security of many cryptographic algorithms, such as Diffie-Hellman key exchange protocol, ElGamal cryptosystem etc. depend on this hardness of DLP.

For given x, The repeated square and multiply algorithm is used to compute the modular exponentiation $(g^x modp)$ in polynomial time as follows : Assume that we write the integer x in base 2 expansion

$$x = x_0 + 2x_1 + 2^2x_2... + 2^kx_k.$$

Algorithm 1 (The repeated square and multiply)
Input : g, x and p
Output: y such that $g^x \equiv y(\text{mod } p)$
1. $y = 1$
2. $i = k$
3. While $i \geq 0$
4. $y = y^2(\text{mod } p)$

5. If $x_i = 1$ then $y = y * g(modp)$
6. $i = i - 1$

Gadiyar et al. ask the question what is the inverse of this algorithm. it is clear that the inverse of this algorithm is to divide and repeatedly extract square root as follows :

Algorithm 2 (Divide and repeated square root)
Input : g, y and p Where $g^x \equiv y \pmod p$
Output: x
1. $i = 0$
2. While $y \neq 1$
3. If $\left(\frac{y}{p}\right) = -1$ then
4. $x_i = 1$
5. $y = y * g^{-1} (\text{mod } p)$
6. $y = y^{\frac{1}{2}} (\text{mod } p)$
7. else
8. $x_i = 0$
9. $y = g^{\frac{\alpha}{2}} (\text{mod } p)$ Where $y = g^\alpha$
10. $i = i + 1$
11. $x = (x_i x_{i-1} x_{i-2} ... x_0)$

Note that : $\left(\frac{y}{p}\right)$ denotes the legendre symbol and since

$$\left(\frac{y}{p}\right) = \left(\frac{g^x}{p}\right) = \left(\frac{g}{p}\right)^x = (-1)^x$$

the least significant bit of x determined by legendre symbol for each time in the algorithm. In Algorithm 2, There exists polynomial time algorithms finding the two square root of an element y in $(\mathbb{Z}/p\mathbb{Z})^*$, but there is no known method which determines which one of these two square roots of $y = g^\alpha$ is $g^{\frac{\alpha}{2}}$. So, Algorithm 2 is useless.

2. Gadiyar's Algorithm

In [2], Gadiyar et al. gave the following probabilistic algorithm which bypasses the problem in Algorithm 2 by choosing a random square root.

Algorithm 3 (Gadiyar's probabilistic algorithm)
Input : g, y and p where $g^x \equiv y \pmod p$
Output: x
1. Choose an integer B and create Table A consisting of $(g^{k_j} \text{ mod } p \text{ , } k_j)$ where $j \leq B$. Here k_j is any subsequence of integers.
2. $i = 0$, $k = 0$, $l = 1$, $m = x$, $y[1] = y$, $m_1[1] = x$, $c_1[1] = y$, $c_2[1] = y$, $m_2[1] = x$.

3.1. If $\left(\frac{y}{p}\right) = -1$ then goto Step 4.
3.2. If $\left(\frac{y}{p}\right) = 1$ then goto Step 6.
4.1. $y = y * g^{-1} (\text{mod } p)$ and $m = m - 1$.
4.2. Goto Step 5.
4.3. If Step 5 does not solve for x, $i = i + 1$, store $y[i] = y$ and $m_1[i] = m$ in Table II.
4.4. Goto Step 6.
5.1. If $y \equiv g^{k_j} \pmod p$ for any $j \leq B$ in Table A, $Solve(m, k_j, k)$.
5.2. If $y \equiv y[j] \pmod p$ for any $j \leq B$ in Table B, $Solve(m, m_1[j], k)$.
5.3. If $y \equiv c_1[j] or c_2[j] \pmod p$ for any $j \leq B$ in Table C, $Solve(m, m_2[j], k)$.
6.1. $y = y^{\frac{1}{2}} (\text{mod } p)$ and $m = \frac{m}{2}$, $k = k + 1$, goto Step 5.
6.2. If Step 5 does not solve for x, $y = p - y (\text{mod } p)$, goto Step 5.
6.3. If Step 5 does not solve for x, $l = l + 1$, store $c_1[l] = y$, $c_2[l] = p - y$ and $m_2[l] = m$ in Table C.
6.4. $y = c_1[l]$ or $c_2[l]$ randomly. Goto Step 3.
Solve(a,b,t) : Solve the linear congruence:

$$2^t a \equiv 2^t b (mod p - 1)$$

The running time of this algorithm about $O\left(\sqrt{p}\right)$ with $O\left(\sqrt{p}\right)$ memory (See [2])

3. Pollard's Rho Method

Pollard's rho method is summarized as follows: To compute $\log_g y \pmod p$, Pollard used the iterating function $f_P(h)$ given by

$$f_P(h) = \begin{cases} gh & : \quad 0 < y \leq \frac{p}{3} \\ h^2 & : \quad \frac{p}{3} < y \leq \frac{2p}{3} \\ hy & : \quad \frac{2p}{3} < y < p \end{cases}$$

and defined a sequence (h_k) according to rule $h_0 = 1$, $h_{k+1} \equiv f_P(h_k) \pmod p$. If one finds a collision in (h_k), that is $h_m \equiv h_n \pmod p$ for $m \neq n$, we can recover the discrete logarithm x.

Pollard's rho algorithm has a running time about $O\left(\sqrt{p}\right)$ with $O\left(\sqrt{p}\right)$ memory requirements. There are collision-detection algorithms which can be applied for Pollard's rho algorithm to minimize the storage requirement. In this case, Pollard's rho algorithm needs only negligable storage requirement. See [3] for detailed information.

4. Comparison of Algorithms

In fact, Gadiyar's algorithm is very similar to Pollard's rho algorithm. We can interpret the Gadiyar's algorithm used the iterating function $f_G(h)$ given by

$$f_G(h) = \begin{cases} h^{\frac{1}{2}} mod\ p & : \quad \left(\frac{h}{p}\right) = 1 \\ hg^{-1} & : \quad \left(\frac{h}{p}\right) = -1 \end{cases}$$

and defined a sequence (h_k) according to rule $h_0 = 1$, $h_{k+1} \equiv f_G(h_k) \pmod p$. Note that $h^{\frac{1}{2}}\ (mod\ p)$ denote the randomly chosen square root of $h(\text{mod } p)$. Gadiyar's algorithm search for a collision in (h_k) using the Tables A,B and C in Algorithm 3.

The aim of this interpretation is comparison of the running times of these two algorithms by finding the performance of the iterating function $f_G(h)$.

For $B \approx \sqrt{p}$ in Algorithm 3, Table I shows the performance of the iterating function $f_G(h)$. In this experiment, we work with group orders between 10^{n-1} and 10^n, for $3 \leq n \leq 8$. For fixed n, we determine the prime field $(\mathbb{Z}/p\mathbb{Z})^*$ by choosing a random p such that $10^{n-1} < p < 10^n$ and find a primitive element g of $(\mathbb{Z}/p\mathbb{Z})^*$. We choose randomly $\mathcal{S}$ elements of this prime field. Then, we determine the required iteration number of each element for a collision. Let we define T as total iteration number, we calculate L_p given by

$$L_p = \frac{T}{\mathcal{S}\sqrt{p}}.$$

We repeat the above process for different groups by choosing random primes p. At last, we determine L which is arithmetic means of L_p's. The value L shows the performance of the $f_G(h)$, that is, we have determined experimentally the running time of the Gadiyar's algorithm in Table 1. The first column gives multiplicative groups of prime fields by the number of decimal digits in their order. The second column gives the values of L for each row. The third column gives the number of different groups. The values $\mathcal{S}$ is given in fourth column and the last column gives the number of total examples whose discrete logarithm is calculated for each row in the experiment.

Now, we can compare the performances of the Gadiyar's algorithm and the Pollard's rho algorithm by using Table 1. Pollard's rho algorithm finds a collision about $1.37\sqrt{p}$ iterations without a collision-detection algorithms in $(\mathbb{Z}/p\mathbb{Z})^*$ [1]. At the same time, Pollard's iteration function require only one group operation. On the other hand, We show that experimentally Gadiyar's algorithm finds a collision about $0.55\sqrt{p}$ iterations, but the function f_G require about $O(\log^3 p)$ group operation because of legendre symbol. Therefore, we have show that experimentally the

Table 1: The Performance of $f_G(h)$ for $(\mathbb{Z}/p\mathbb{Z})^*$

n	L	# Different groups	$\mathcal{S}$	# Examples
3	0.542	100	100	10000
4	0.558	100	100	10000
5	0.564	100	100	10000
6	0.569	100	100	10000
7	0.575	100	100	10000
8	0.627	100	100	10000
Avarege	0.555	-	-	32250

running time of the Pollard's rho algorithm is better than the running time of the Gadiyar's algorithm.

5. Improving The Algorithm

In order to minimize the storage requirement, a collision-detection algorithm can be applied with a small penalty in the running time for Pollard's rho algorithm. However, in Gadiyar's algorithm, we extract the square roots and here a decision function ρ will choose one of the square roots randomly and hence even if there is a collision $x_i = x_j$ for some $i \neq j$, the values x_{i+1} and x_{j+1} could be equal or negative of the other. As the function ρ is random the values ρ takes at x_i and x_j may not be same even x_i and x_j are equal. So it is most unlikely to get the cycle, that is a collision-detection algorithm can not be applied to Gadiyar's algorithm.

If we can convert the iteration function $f_G(h)$ in Gadiyar's algorithm as a deterministic function, we can use a collision detection algorithm for Gadiyar's algorithm.

Let replace the $f_G(h)$ with function $f_M(h)$ as follows:

$$f_M(h) = \begin{cases} \textit{The smaller square root of } h \textit{ mod } p & : \quad \left(\frac{h}{p}\right) = 1 \\ hg^{-1} & : \quad \left(\frac{h}{p}\right) = -1 \end{cases}$$

In this case, since the function $f_G(h)$ is deterministic if a collision occurs then we get a cycle for iterating function $f_M(h)$. So we can use a cycle-detection algorithm for iterating function $f_M(h)$. The question arises naturally that how this modification effects the running time of the algorithm. We reply this question experimentally in Table 2.

According to Table 2, this version of the Gadiyar's algorithm finds a collision about $1.211\sqrt{p}$ iterations using Teske's collision-detection algorithm (See [4]). That

Table 2: The Performance of $f_M(h)$ for $(\mathbb{Z}/p\mathbb{Z})^*$

n	L	# Different groups	$\mathcal{S}$	# Examples
3	1.164	100	100	10000
4	1.185	100	100	10000
5	1.261	100	100	10000
6	1.365	100	100	10000
7	1.217	100	100	10000
8	1.135	100	100	10000
Avarege	1.211	-	-	32250

is, there is a inconsiderable effect on running time of the Gadiyar's algorithm, but we decimate the required storage using a collision detection algorithm as a negligable storage requirement.

References

[1] Bai, S. and Brent, R.P., *On the Efficieny of Pollard's Rho Method for Discrete Logarithms*, Conferences in Research and Practice in Information Technology, James Harland and Prabhu Manyem Ed., Wollongong, Australia., Vol. 77, (2008), 207–216.

[2] Gadiyar, H.G., Maini, K.M.S., Padma, R.and Romsy, M, *What is the inverse of repeated square and multiply algorithm?*, Colloq. Math., Vol. 116, (2009), 1-14.

[3] Pollard, J. M, *Monte Carlo methods for index computation (mod p)*, Mathematics of Computation, Vol. 32, (1978), 918-924.

[4] Teske, E, *A space efficient algorithm for group structure computation*, Mathematics of Computation, Vol. 67, 224 (1998), 1637-1663.

Murat Sahin
Department of Mathematics
Ankara University
Tandogan, Ankara, Turkey.
E-mail : *musahin@science.ankara.edu.tr , muratsahin1907@gmail.com*

Ali Bulent Ekin
Department of Mathematics
Ankara University
Tandogan, Ankara, Turkey.
E-mail : *aekin@science.ankara.edu.tr*