UNEXPOSED EXPONENTS

The Ornery Case of the Discrete Log Problem in Cryptography

Simon Spicer
0939537
Math 414, Winter 2010, University of Washington

March 10, 2010

Abstract

The discrete log problem is the name given to the fact that, while computing exponentiation in finite cyclic groups is easy, the reverse operation - the discrete analogue of the classical logarithm - is in general very difficult. For this reason the discrete log is central to many public cryptographic schemes in active use today.

To illustrate this, we show how the difficulty of solving discrete logs ensures the security of three example cryptographic schemes. We also describe two simple generic algorithms that solve the discrete log, and quantify mathematically just how inefficient they are. Finally, we provide implementations of these two methods in Sage.

1 Overview

Note: For this project we will assume a basic working knowledge of finite group theory and elementary number theory.

The problem we which to solve is as follows (using multiplicative notation):

Given a generic finite cyclic group G with generator g i.e. $G =< g >$, and given some $a \in G$, find the smallest x such that $g^x = a$.

This is known as the *Discrete Logarithm Problem*. The reason that it is labelled a 'problem' is because, while there are algorithms that perform exponentiation in cyclic groups quickly and efficiently [1, pg. 50], no efficient methods are known for the inverse operation – the discrete logarithm described above. It is thus an example of an *asymmetric function*: one for which it is easy to compute the output given the input, but extremely difficult (though still possible) to compute the input given the output.

Asymmetric functions form the basis of many cryptographic schemes. We will describe three that rely on the intractability of the discrete log problem: Diffie-Hellman Key Exchange, the ElGamal cryptosystem and the related ElGamal signature scheme.

2 Context

As mentioned, the discrete log is a problem that is integral to many areas of cryptography. Diffie-Hellman Key Exchange and the ElGamal Cryptosystem and Signature Scheme are perhaps three of the most elegant schemes in which it occurs.

2.1 Diffie-Hellman Key Exchange

Suppose Alice and Bob want to exchange private information, but are worried about Eve listening in on their communication. They know that they can achieve this end by sharing a secret key, which could then be used to encrypt a larger piece of data that they want to exchange.

Diffie-Hellman Key Exchange is a method for Alice and Bob to arrive at a secret key in such a way that is provably secure, even if Eve listens to every word of the communication between the two.

The scheme was first published by Whitfield Diffie and Martin Hellman in 1976 (although the British Intelligence forces had come up with the same idea independently a few years prior, but had to keep such musings classified) [7],[1, pp. 49-50]. It was a landmark event in the field of cryptography: Diffie-Hellman was the first practicable example of a publicly published scheme using *public keys*, sets of information that are available for anyone to scrutinize. This allows for a much more formally rigorous analysis of the cryptographic scheme. Contrast this with *security through obscurity*, where (you hope) your transmissions are secure mostly because no-one has figured out yet how they work.

Like the other schemes discussed in this project, Diffie-Hellman Key Exchange is perhaps most often seen in the context of multiplicative group of integers modulo a prime; however, it can be applied to any cyclic group in which the discrete log problem is difficult.

The generic scheme is thus:

1. Alice and Bob (publically) agree upon a large group G for which the discrete log is difficult, and a generator $g \in G$.
2. Alice chooses a random $x \in G$, and calculates $a = g^x$. She tells no-one about x, but sends a to Bob.
3. Bob chooses a random $y \in G$, and calculates $b = g^y$. He tells no-one about y, but sends b to Alice.
4. Alice calculates $s = b^x = (g^y)^x = g^{xy}$. Bob calculates $a^y = (g^x)^y = g^{xy} = s$.
5. Alice and Bob have arrived upon the same value - that of the shared secret $s = g^{xy}$.

Note that throughout the whole procedure Eve only has access to the following: p, g, g^x and g^y. Because of the difficulty of the discrete log problem, for sufficiently large groups G it is computationally infeasible for Eve to calculate x or y. Since the only way to get s from the above known values is exponentiation by x or y, Eve can never determine the value of the shared secret s.

2.2 The ElGamal Cryptosystem

While Diffie-Hellman allows for the sharing of a secret key, Alice and Bob generally have no control as to what that key actually is. The scheme thus can only be used to securely share a random key, and allows for no transmission of meaningful information. It is also a two-way

process, requiring active negotiation between both parties at the time the shared secret is generated. The next scheme bypasses both of these shortcomings.

The Egyptian cryptographer Taher Elgamal puplished his eponymous cryptographic scheme in 1985 (only 25 years ago!) [8]. The scheme shares the same foundation as Diffie-Hellman key exchange, and as such can be implemented for any large cyclic group in which the discrete log problem is difficult. In particular, it has commercial implementations exist over both the multiplicative integers modulo p and elliptic curves.

Suppose Bob wants to send a secret message to Alice.

1. (Alice Generates a Reusable Key)
   - Alice chooses a large cyclic group G and a generator $g \in G$.
   - Alice chooses a random $x \in G$, and calculates $a = g^x$.
   - Alice publishes the triplet (G, g, a) for the world to see, but tells no-one about x.
2. (Bob Encrypts)
   - Bob encodes his plaintext message to Alice as an integer $M \in G$.
   - Bob chooses a random $y \in G$, and calculates $b = g^y$.
   - Bob calculates the shared secret $s = a^y$.
   - Bob encrypts the message: he calculates the ciphertext $C = M \cdot s$.
   - Bob transmits the pair (C, b) to Alice, but tells no-one about y or s.
3. (Alice Decrypts)
   - Alice calculates the inverse of the shared secret $s^{-1} \equiv (b^x)^{-1}$.
   - Alice decrypts the message. She recovers M by multiplying C and s^{-1}: $C \cdot s^{-1} = M \cdot s \cdot s^{-1} = M$.

Eve has access to the following information: p, g, g^x, g^y and C. Again, due to the discrete log problem being such a tough nut to crack, she cannot recover x or y, and so is unable to determine s. Thus Eve cannot determine the plaintext M from the ciphertext C.

2.3 The ElGamal Signature Scheme

Suppose now that Bob wants to ensure that a given message claiming to have been sent by Alice, is actually from Alice. To do this the two can employ a *digital signature scheme*, of which the The ElGamal Signature Scheme is but one of many.

Also invented by Taher Elgamal, the ElGamal Signature Scheme shares some similarity with the eponymous cryptosystem, most notably in the key generation stage. However, it differs notably from the cryptosystem in that it relies on the specific properties of the $(\mathbb{Z}/p\mathbb{Z})^*$, the multiplicative integers modulo some prime p.

Theorem 2.1. *If p is a positive prime integer, then the multiplicative group of integers modulo p $(\mathbb{Z}/p\mathbb{Z})^* = (\mathbb{Z}/p\mathbb{Z}) \setminus \{0\}$ is a cyclic group of size $p - 1$.*

Definition 2.2. A *cryptographic hash function* H is a algorithmic procedure that takes a (possibly variable-sized) message M and outputs a smaller fixed-length bit output $H(M)$, subject to the following:

- $H(M)$ is easy to compute;
- a small change in M produces a large change in M;

- It is computationally infeasible to find a message M with a given hash;
- It is computationally infeasible to find two different messages with the same hash.

Cryptographic hashes provide a simple way to check if a message has been corrupted, as doing so will significantly alter the hash of that message. The last two features listed above also ensure against deliberate tampering. These characteristics make hashes attractive for use in digital signature schemes.

As such, the full scheme is as follows:

1. (Alice Generates a Reusable Key)
   - Alice chooses a large prime p and a generator $g \in (\mathbb{Z}/p\mathbb{Z})^*$.
   - Alice chooses a random $x \in (\mathbb{Z}/p\mathbb{Z})^*$, and calculates $a \equiv g^x \pmod{p}$.
   - Alice publishes the triplet (p, g, a) for the world to see, but tells no-one about x.
2. (Alice Signs Her Message)
   - Alice chooses a random $y \in (\mathbb{Z}/p\mathbb{Z})^*$, and calculates $b \equiv g^y \pmod{p}$.
   - Alice hashes her message: $h = H(M)$.
   - Alice calculates $t = (h - xb) \cdot y^{-1} \pmod{p-1}$ (note the modulus is different!). There is a chance that $t = 0$; in this case, choose a new y and try again.
   - Alice attaches the pair (b, t) to her message M. She keeps the value of y secret.
3. (Bob Verifies)
   - Bob hashes Alice's message to obtain $h = H(M)$.
   - Bob calculates whether $g^h \equiv a^b \cdot b^t \pmod{p}$. If this is the case, then Bob accepts the message as authentic; if not, Bob rejects it as counterfeit.

To see why this works, observe that $t = (h - xb) \cdot y^{-1} \pmod{p-1} \Rightarrow h \equiv xb + ty \pmod{p-1}$.
Now from Fermat's Little Theorem we have that for any $u, v \in \mathbb{Z}$,
$g^u \equiv g^v \pmod{p} \Leftrightarrow u \equiv v \pmod{\varphi(p)}$.
Since $\varphi(p) = p - 1$, we thus have:

$$g^h \equiv g^{xb+ty} \equiv (g^x)^b \cdot (g^y)^t \equiv a^b \cdot b^t \pmod{p}$$

Thus for an authentic message from Alice Bob will indeed determine $g^h \equiv a^b \cdot b^t \pmod{p}$.

However, suppose an insidious Eve wants to forge a message from Alice. She has access to the following information published by Alice: p, g, and $g^x \pmod{p}$.
Because of the intractability of the discrete log problem, Eve cannot determine the value of either x.
Thus, even though she can pick her own y, Eve cannot generate an appropriate $t = (h - xb) \cdot y^{-1} \pmod{p-1}$, as it relies upon knowing x.
Hence Eve has no feasible way of producing a pair (b, t) such that $g^h \equiv a^b \cdot b^t \pmod{p}$.

(Note that Eve also cannot take a legitimate message from Alice and modify it while leaving the signature unchanged, as the hash function ensures that the new h will be significantly different from its original value, and so g^h and $a^b \cdot b^t \pmod{p}$ will no longer match.)

3 Attacking the Discrete Log Problem

We have stated often enough that solving the discrete log problem is intractable for some large cyclic groups, but what exactly does this mean?

3.1 Big O Notation

When discussing the efficiency of an algorithm, we will use *Big O* notation. This gives us an indication of how quickly the computing time of a given algorithm grows as a function of the size of the input.

Definition 3.1. A function on the integers $f(n)$ is said to be $O(g(n))$, or *order* $g(n)$, for some function $g(n)$ if

$$\limsup_{n\to\infty} \left| \frac{f(n)}{g(n)} \right| < \infty.$$

Informally, $f = O(g)$ if f and g eventually differ by at most a constant factor.

For example, if k is the number of bits used to describe the integer fed into the algorithm f, saying $f = O(k^3)$ tells us that for large inputs, the (average) computing time of f is at worst proportional to the cube of the number of input bits.

Note that a function that is $O(k^2)$ is also $O(k^3)$. We usually, however, give the best upper bound on such an algorithm when discussing its complexity. Thus when we say that an algorithm is $O(k^3)$, it is understood that it will, for sufficiently large k, run slower than an algorithm that is $O(k^2)$. This is because a cubic function will always eventually outstrip a quadratic in magnitude. For the same reason, an algorithm that is $O(e^k)$ will eventually run slower than an algorithm of $O(k^r)$ for any value of $r \geq 0$.

Thus we can formalize just what we mean when we say an algorithm is efficient:

Definition 3.2. An algorithm acting on a k-bit input is *efficient* if it is $O(k^r)$ for some $r \geq 0$. We say then that the algorithm runs in *polynomial time*.

Definition 3.3. An algorithm is *inefficient* if its complexity is not polynomial time.

For example, an exponential time function i.e. one which is $O(e^k)$, is (horribly) inefficient.

In the context of number theoretic algorithms we more often describe the input by the value of the number being fed into the algorithm. In the context of the discrete log problem, the yardstick is n, the size of the group under scrutiny. Note that an n can be described in approximately $\log_2(n) \propto \log(n)$ bits. Thus if an efficient discrete log algorithm existed, it would be $O(\log(n)^r)$ for some $r \geq 0$. On the other hand, any algorithm that is $O(n^r) = O(e^{k^r})$ for $r > 0$ is inefficient.

3.2 Badly Chosen Groups

First and foremost, we must note that the discrete log problem is difficult for *most* large cyclic groups; it is certainly the case that solving discrete logs is easy - efficient - in some cases. It is up to the implementer of a given cryptographic scheme to find a suitable large group in which taking the discrete logarithm is provably difficult.

Theorem 3.4. *If G is a finite abelian group of order n, and $n = q_1^{\alpha_1} q_2^{\alpha_2} \cdots q_k^{\alpha_k}$ is the prime decomposition of n,*
then $G \simeq H_1 \times H_2 \times \cdots \times H_k$,
where H_i is the product of finite cyclic groups of order some power of q.

That is, a finite abelian group is always isomorphic to the product of a bunch of cyclic p-groups.

To illustrate the potential shortfalls of picking a bad group, suppose we are working in $(\mathbb{Z}/p\mathbb{Z})^*$, but have chosen a p such that $p-1$ is the product of only small primes. From the above, since $(\mathbb{Z}/p\mathbb{Z})^*$ is a finite abelian group of order $p-1$, it is isomorphic to a the product of some number of cyclic groups with really small order.

The smaller a finite group, the easier it is to calculate discrete logs in it. It turns out that if we can solve the problem modulo each of the small group orders, then we can use the Chinese Remainder Theorem to reconstruct the answer in $(\mathbb{Z}/p\mathbb{Z})^*$. If $p-1$ has only small prime factors, then the former is easily done. Thus discrete logs in $(\mathbb{Z}/p\mathbb{Z})^*$ can be recovered efficiently for this choice of p.

There is a discrete log attack built specifically on this premise: the Pohlig-Hellman Algorithm. See [3, pp. 166-170] and [1, p. 39-42] for more details.

3.3 The Brute Force Method

Suppose, however, that we know nothing about our cyclic group G, perhaps not even its order n. A number of generic discrete log-calculating methods do exist; it is just that they are very inefficient.

The simplest way to compute discrete logs is simply to calculate higher and higher powers of the given generator, and see which one equals our target element. A generic brute force algorithm would look something like this:

Given:

- a finite cyclic group G of unknown order,
- a generator $g \in G$,
- an element $a \in G$.

Find:
(The smallest) x such that $g^x = a$.

The Algorithm:

1. Set $b = 1$
2. Set $x = 0$
3. While $b \neq a$:
   - if $b = a$, output x and halt;
   - otherwise set $b = b \cdot g$, and set $x = x + 1$.

Note that, assuming given a random$a \in G$, we will on average exponentiate g through half the elements of G before we find a. If we assume that multiplication is a roughly constant-time operation on all the elements of G, we deduce that the average total computing time will be proportional to the size of G. Thus the brute force method is $O(n) = O(e^k)$, where k is the number of bits needed to describe the order of G. Hence we see that the brute force method runs in exponential time, making it very inefficient indeed.

3.4 The Baby-Step, Giant-Step Algorithm

Thankfully, the naïve method above is not the only way to calculate discrete logs. We will explore one such improvement, the curiously-named 'baby-step giant-step method' (the reason for its name will become clear below). We describe the baby-step giant-step (BSGS) algorithm in full generality for a generic cyclic group.

Given:

- a finite cyclic group G of order n,
- a generator $g \in G$,
- an element $a \in G$.

Find:
(The smallest) x such that $g^x = a$.

The Premise:
Suppose $g^x = a$.
Since g generates G, it follows that $0 \leq x < n$.
Let $m = \lceil \sqrt{n} \rceil$. Then $x = qm + r$, where $0 \leq q < m$ and $0 \leq r < m$.
Thus $a \equiv g^x \equiv g^{qm+r} \equiv (g^m)^q g^r \pmod{p}$,
and so $a \cdot (g^{-m})^q \equiv g^r \pmod{p}$.

Hence we can calculate and store the values of g^r (the 'baby step') for $0 \leq r \leq m-1$.
We then multiply a by successively higher powers of g^{-m} (the 'giant step'). We are guaranteed that the result will equal one of the stored values before the exponent of g^{-m} hits m.

The Algorithm:

1. Set $m = \lceil \sqrt{n} \rceil$.
2. For every $0 \leq r < m$, calculate g^r and store the pair (r, g^r) in a lookup table.
3. Set $b = g^{-m}$.
4. For every $0 \leq q < m$:
   - if a equals any of the stored g^r, output $x = qm + r$ and halt;
   - otherwise set $a = a \cdot b$.

Note that as outlined here, we are required to know the n, the size of G. However, we need only have an estimate for n for this algorithm to suffice. More complex implementations of BSGS exist for when even this is not known.

3.5 Analysis of the Baby-Step, Giant-Step Algorithm

3.5.1 Running Time

The algorithm is comprised of two distinct parts:

- calculating and storing the m baby-step powers; and
- calculating the product of a and powers of g^{-m}, and checking to see if this equals any of the stored baby-step values.

For the first part, we must perform $m \approx \sqrt{n}$ exponentiations in G. Assuming that exponentiation in G and storing the resulting values in a table are roughly constant-time operations, we thus have that this section completes in $O(\sqrt{n})$ time.

For the second part - assuming that a is drawn with equal probability from anywhere in G - we will complete on average $m/2$ multiplications of a by $b = g^{-m}$ before we strike gold when comparing the result with the m stored values.

Now the last part of this step - comparing our computed product with the values stored in the lookup table - could potentially take a significant amount of time. However, there are efficient ways of storing the values of g^r - such as using a hash table - that reduce lookup time to $O(1)$ (ignoring log factors) [6]. That is, we can code the lookup table in such a way that the lookup time is negligible compared to that of multiplication in G. Hence the running time of the second part of BSGS can be reduced to $O(\sqrt{n})$.

Since both parts have the same order of running time, we have the total running time of the BSGS algorithm to be $O(\sqrt{n})$.

This is a considerable improvement to the brute force method, which has $O(n)$ running time.

3.5.2 Memory Requirements

The obvious drawback to BSGS is that we must store $m \approx \sqrt{n}$ powers of g in a lookup table. No matter how this is done it means that we will require $O(\sqrt{n})$ memory in order to use the baby-step giant-step algorithm (for a 20-digit n this translates to a gigabyte or more-sized lookup table; a 50-digit n would require more memory than is available in all the world's computers combined). This is in marked contrast to the brute force method, whose memory requirements do not scale significantly with the size of n.

Note that we can reduce the memory requirements for BSGS by computing and storing less powers of g in a lookup table. We would then, however, have to on average calculate and compare $a \cdot (g^{-m})^q$ for more values of q before we find an equality. The time and memory requirements of BSGS are in fact inversely proportional: for any $0 \leq d \leq n$ we may adapt BSGS so that required memory is $O(d)$, but then running time becomes $O(\frac{n}{d})$.

3.6 Can We Do Better?

It was proven in 1997 by Victor Shoup, then of IBM research in Zurich, Switzerland, that any generic attack on the discrete log problem has complexity no better that $O(\sqrt{n})$ [2]. Shoup proved this by showing that a generic algorithm must perform a minimum of $O(\sqrt{q})$ group operations before halting, where q is the largest prime divisor of the group size n. Since we can

expect q to scale roughly with n, it follows that any generic discrete log attack is at best $O(\sqrt{n})$.

Thus BSGS is close to optimal in terms of the time complexity of a generic discrete log algorithm. However, generic algorithms with better memory complexity do exist. The most notable of these is the Pollard Rho discrete log algorithm. Like BSGS, it also has complexity $O(\sqrt{n})$, but unlike BSGS it does not have memory requirements that scale significantly with n.

Of course, once we go beyond the framework of generic attacks, all bets are off. Faster discrete log algorithms have been constructed for groups with more structure, such as $(\mathbb{Z}/p\mathbb{Z})^*$ and those defined over elliptic curves. Notable examples for the multiplicative group of integers modulo p are the Pohlig-Hellman algorithm mentioned previously, and the Index Calculus Method. The latter is the fastest known discrete log algorithm for $(\mathbb{Z}/p\mathbb{Z})^*$, and has subexponential time complexity: using heuristic arguments we can estimate the running time to be $O(e^{a\sqrt{k \log k}})$ for some small constant a (where k is the number of bits in the binary representation of n) [3, p. 172].

4 Implementing BSGS in Sage

Although we have described both the brute force method and BSGS for solving discrete logs over a generic cyclic group, they are perhaps most easily realized in the context of $(\mathbb{Z}/p\mathbb{Z})^*$, the multiplicative groups over the integers modulo the prime p. We therefore provide a toy implementation of BSGS - and, for comparison, the brute force attack - in Sage. The code for both implementations defines a function that solves the following problem:

Input:

- a prime p,
- a generator $g \in (\mathbb{Z}/p\mathbb{Z})^*$,
- an integer a with $1 \leq a < p$.

Return:
(The smallest) x such that $g^x \equiv a \pmod{p}$.

4.1 Brute Force Implementation

```
def discrete_log_brute(p,g,a):
     # First check that p is indeed prime and that a and g are in (Z/p)*
     assert is_prime(p), "p is not prime"
     assert gcd(p,g)==1 and gcd(p,a)==1, "g or a not in (Z/p)*"

     # Convert g and a to integers modulo p
     g = mod(g,p)
     a = mod(a,p)

     # iterate over b = g^x and see when the result equals a
     x = 0
     b = 1
     for i in range(p):
         if b == a: return x
         x = x+1
         b = b*g

     # If we get here, then g is not a generator of (Z/p)*
     return "a not in <g>"
```

4.2 BSGS Implementation

```
def bsgs(p,g,a):
    # First check that p is indeed prime and that a and g are in (Z/p)*
    assert is_prime(p), "p is not prime"
    assert gcd(p,g)==1 and gcd(p,a)==1, "g or a not in (Z/p)*"

    # Convert g and a to integers modulo p
    g = mod(g,p)
    a = mod(a,p)

    # The size of (Z/p)* is p-1, so m = ceil(sqrt(p-1))
    m = ceil(sqrt(p-1))

    # The first two values of the lookup table are (0, g^0) and (1,g)
    t = [[0,1],[1,g]]

    # Populate the rest of the lookup table
    b = g
    for r in range(2,m+1):
        b = b*g
        t = t + [[r,b]]

    # Re-use b; it is now the size of the giant step
    b = g^(-m)

    # Calculate a*b^q for 0 <= q < m, and see if the result equals any values in the table
    for q in range(0,m):
        for x in t:
            if a == x[1]:
                return q*m + x[0]
        a = a*b

    # If we get here, then g is not a generator of (Z/p)*
    return "a not in <g>"
```

Note that we have taken no great steps to optimize the code. Specifically, the lookup process as implemented here will contribute significantly to the running time of the algorithm; this is thus expected to be worse than $O(\sqrt{n})$.

4.3 Running Time Comparison

We implement the following code in Sage:

```
time x = discrete_log_brute(100003,7,77)
time y = bsgs(100003,7,77)
print(x,y)
time x = discrete_log_brute(10000019,7,77)
time y = bsgs(10000019,7,77)
print(x,y)
time x = discrete_log_brute(100000007,7,77)
```

```
time y = bsgs(1000000017,7,77)
print(x,y)
```

This compares the running time of our implementations of the brute force method and BSGS on three multiplicative groups, for primes $p = 100003, 10000019$ and 100000007 (we have checked independently that 7 is a generator of each group). The above code also prints out the result for each computation, to make sure that the exponents calculated using both algorithms agree.

The resulting output is thus (computations were done on a 2.53GHz MacBook Pro):

```
Time: CPU 0.01 s, Wall: 0.01 s
Time: CPU 0.00 s, Wall: 0.00 s
(6243, 6243)
Time: CPU 2.22 s, Wall: 2.25 s
Time: CPU 0.85 s, Wall: 0.86 s
(4311737, 4311737)
Time: CPU 92.32 s, Wall: 102.44 s
Time: CPU 16.04 s, Wall: 18.40 s
(64045025, 64045025)
```

What we see, then, is that computing discrete logs using either method is essentially instantaneous when p is small - here of the order of 10^6. However, the baby-step giant-step method is noticeably faster for p of the order of 10^8. Increase by p another factor of 10, and the brute force method is essentially unusable. The rapid outstripping of even this unoptimized version BSGS relative to the brute force method lends support to the idea that it really is a much faster algorithm.

And yet, even for p around 10^9 the BSGS has essentially reached its usable limit. For primes bigger than this it will still be too slow to be of any practical use.

Note that these prime sizes are tiny in comparison to the record group sizes that have been 'broken' using more advanced discrete log solving methods. For example, the current record for an Elliptic Curve-based method is for a 112-bit prime – or a prime of 34 decimal digits [4].

However, this is again miniscule in comparison to the key sizes used in commercial cryptography. The ElGamal Cryptographic scheme, for instance, generally uses keys of size 512 or 1024 bits, or primes of length around 154 or 308 digits respectively. This puts the keys beyond the range of feasible attack for any known discrete log algorithm, ensuring the security of the data that such a scheme encrypts.

References

[1] R. A. Mollin, *RSA and Public-Key Cryptography,*
Chapman & Hall/CRC, Boca Raton, 2002.

[2] V. Shoup, *Lower Bounds For Discrete Logarithms and Related Problems,*
Advances in Cryptology - EUROCRYPT 97. Springer-Verlag, 1997.

[3] D. R. Stinson, *Cryptography: Theory and Practice*
CRC Press, Boca Raton, 1995.

[4] J. W. Bos, M. E. Kaihara et al, *PlayStation 3 Computing Breaks* 2^{60} *Barrier,*
École Polytechnique Fédérale de Lausanne, 8 July 2009,
`http://lacal.epfl.ch/page81774.html`

[5] Wikipedia contributors, *Big O Notation,*
Wikipedia, The Free Encyclopedia, 6 March 2010,
`http://en.wikipedia.org/wiki/Big_o_notation`

[6] Wikipedia contributors, *Baby-Step, Giant-Step,*
Wikipedia, The Free Encyclopedia, 26 September 2009,
`http://en.wikipedia.org/wiki/Baby-step_giant-step`

[7] Wikipedia contributors, *Diffie-Hellman Key Exchange,*
Wikipedia, The Free Encyclopedia, 6 March 2010,
`http://en.wikipedia.org/wiki/Diffie-Hellman_Key_Exchange`

[8] Wikipedia contributors, *ElGamal Encryption,*
Wikipedia, The Free Encyclopedia, 23 December 2009,
`http://en.wikipedia.org/wiki/ElGamal`

[9] Wikipedia contributors, *ElGamal Signature Scheme,*
Wikipedia, The Free Encyclopedia, 31 December 2009,
`http://en.wikipedia.org/wiki/ElGamal_signature_scheme`