How Discreet is the Discrete Log?

Douglas L. Long[†]
Avi Wigderson[‡]
Department of EECS
Princeton University
Princeton, NJ

Abstract: Blum and Micali [4] showed how to hide one bit using the discrete logarithm function. In this paper we show how to hide $c \cdot \log\log p$ bits for any constant c, where p is the modulus.

1. Introduction

A function $f: [1,N] \to [1,N]$ is said to be one-way if computing $f(x)$ from x is easy, but computing x from $f(x)$ is hard. One-way functions are extremely attractive in cryptographic applications (e.g. see [4], [5], [12], [13], [14]). However, one should use them with care. The reason is that while x is hard to compute from $f(x)$, it may be possible to obtain *almost all* bits of x easily. It is clear, however, that some bits of x are hard to obtain.

In a recent paper, Blum and Micali [4] introduced the notion of hiding a bit in a one-way function. A Boolean predicate $B: [1,N] \to \{0,1\}$ is said to be hard for f if an oracle for $B(f(x))$ will allow to invert f easily. They prove that a certain Boolean predicate is hard for the Discrete Logarithm function, and show how to use that to construct good pseudo random number generators.

Some natural questions arise in light of their work. Given a one-way function f

1) What are its hard bits?
2) How many hard bits are there?
3) How many bits are hard simultaneously?

To explain what we mean by the last question, consider B, a hard Boolean predicate for f, and its complement, $\bar{B}$. Clearly, $\bar{B}$ is also hard for f. However, some partial information about the pair of bits $(B(f(x)), \bar{B}(f(x)))$ is available, namely the information that they are different. Therefore this two bit predicate is not hard for f. We say that a k-bit

This research was partially supported by
†ARO Grant DA AG29-80-K-0090 and Garden State Fellowship
‡IBM Fellowship

© 1983 ACM 0-89791-099-0/83/004/0413 $00.75

predicate $B^k: [1,N] \to \{0,1\}^k$ is hard for f if for *every* Boolean predicate $B:\{0,1\}^k \to \{0,1\}$, an oracle for $B(B^k(f(x)))$ will allow us to invert f easily. If such a B^k exists, we say that f hides k bits.

It should be clear that if f is one-way, it hides more than $c \cdot \log|N|$ bits for any constant c ($|N|$ is the number of bits in the binary representation of N). Otherwise it would be possible to invert f with an algorithm polynomial in $|N|$. (Just find the easy bits "easily" and the hard bits by a brute force enumeration of their possible values.)

The fact that hiding only one bit suffices for many applications seem to reduce the motivation for the questions above. We would like to argue that this is not the case. From the practical point of view, the ability to hide k bits cuts the computation / communication per secure bit by a factor of k. One can flip k coins at a time or generate k bits at a time with a pseudo random number generator without hurting security.

But what really motivates us is the theoretic point of view. Very little is known about one-way functions. In particular, a major open problem is whether they exist. We believe that discovering and studying the "hard core" (the collection of hard bits) of functions that are believed to be one-way may shed some light on this problem.

In this paper we focus our attention on the Discrete Logarithm function, which is widely believed to be one-way. Our main result states that the discrete log function hides $c \cdot \log|p|$ bits for any constant c, where p is the modulus.

Section 2 contains definitions and some technical lemmas. In sections 3 and 4 we extend the techniques of [4] to prove our main result for the case where the oracle is always correct and the case where it is more correct than incorrect, respectively. In section 5 we discuss the difficulties i extending our methods to hide more bits. urprisingly, this yields a polynomial time al ithm for finding discrete logarithms when $p = 2^m+1$, which is different than the intuitive o..e [10].

2. Definitions and Other Preliminaries

We start with some definitions. From this point on p will represent a fixed prime, Z_p^* the group of units mod p, and g a generator of Z_p^*. Let k be an integer such that $k \le \log p$. Let S_p be the set of integers 1 to $p-1$. This set is divided into 2^k intervals $I_i^k = [L_i^k, U_i^k]$ for i from 0 to 2^k-1 where $L_i^k = \left\lfloor \frac{p-1}{2^k} i \right\rfloor + 1$, and $U_i^k = \left\lfloor \frac{p-1}{2^k}(i+1) \right\rfloor$. This is a partition at the k^{th} level. For $x \in S_p$ let $I^k(x)$ be the interval which contains x, i.e. the integer i such that $x \in I_i^k$. Let $R^k = \left\lfloor \frac{p-1}{2^k} \right\rfloor$. The lower indices of I_i^k are computed modulo 2^k, i.e., I_{r+s}^k means $I_{r+s(\bmod 2^k)}^k$. In general all numbers in this paper are taken modulo their range.

Definition 2.1. Suppose $x \in S_p$. The **index of x**, index(x), is the unique $z \in S_p$ such that $x \equiv g^z(\bmod p)$. z is also known as the **discrete logarithm of x**.

In order to hide the k bits representing the integer i, $0 \le i < 2^k$, choose a random $z \in I_i^k$ and compute $g^z(\bmod p)$. Thus an integer $x \in S_p$ hides the bits which represent the interval of its index.

We need the following lemmas which are given without proof.

Lemma 2.1. $|I_i^k| = R^k$ or R^k+1.
(The intervals have almost the same size.)

Lemma 2.2. $L_i^{k-1} = L_{2i}^k$, $U_i^{k-1} = U_{2i+1}^k$.
(Each interval in the $(k-1)^{th}$ level contains two intervals of the k^{th} level.)

Throughout this paper, given an element $x\in[1,p-1]$, we will apply the operations $w\leftarrow zg^{iR^k}$, $w\leftarrow\sqrt{z}$ (z is a quadratic residue), and $w\leftarrow zg^{-1}$ (z is a non-residue). The effect of these operations on the index of z is shifting by a multiple of R^k, dividing it by two (when even), and subtracting one (when odd), respectively. The following lemmas show into which intervals the new index may fall.

Lemma 2.3. Let $x\in I_j^k$. Then for every $1\le i\le 2^k-1$

(1) $x+iR^k\in I_{i+j}^k$, or

(2) $x+tR^k = L_{t+j}^k$ for some $0\le t<i$.

Proof. By induction on i.

$i=1$. From Lemma 2.1 $x+R^k\in I_j^k$ if and only if $|I_j^k|=R^k+1$ and $x=L_j^k$.

$i>1$. Let $x'=x+(i-1)R^k$. If $x'\notin I_{i+j-1}^k$ then case (2) holds for x' and therefore for x. Otherwise $x'\in I_{i+j-1}^k$ so $x+iR^k=x'+R^k$ and the lemma follows by the same argument as for $i=1$. ∎

Lemma 2.4. If $2x\in I_j^{k-1}$, then

(1) $x\in I_j^k$ or $x\in I_{j+2^{k-1}}^k$, or

(2) $2x = L_j^{k-1}$ or $2x=U_j^{k-1}$.

Proof. Suppose $x\le\frac{p-1}{2}$. (The case $x>\frac{p-1}{2}$ is analogous.) Assume $L_{2j}^k<2x$.

Then $1+\left\lfloor\frac{p-1}{2^k}2j\right\rfloor<2x$. So $\frac{p-1}{2^k}2j<2x$ and $\left\lfloor\frac{p-1}{2^k}j\right\rfloor\le\frac{p-1}{2^k}j<x$.

Therefore $L_j^k=1+\left\lfloor\frac{p-1}{2^k}j\right\rfloor\le x$. Similarly if $2x<U_{2j+1}^k$ then $x<U_j^k$. ∎

Lemma 2.5. If $x\in I_j^k$ (and x is odd), then either

(1) $(x-1)\in I_j^k$, or

(2) $x = L_j^k$.

We can easily determine if $z=g^{L_j^k}$ or $z=g^{U_j^k}$ and thereby find index(z) by maintaining a sorted list of these values. There are $O(2^k)$ values to maintain so sorting will cost $O(k2^k)$ time and each query $O(k)$ time. If z is not one of these values then the above operations will result in the index of z falling in the intervals given in case (1) of lemmas 2.3, 2.4, or 2.5. Henceforth, we will assume that whenever one of these operations takes place a test for case (2) of lemmas 2.3, 2.4, or 2.5 is performed. To make the $O(k2^k)$ cost of the sorting operation polynomial in $|p|$ we must restrict ourselves to $k=O(\log|p|)$.

Definition 2.2. A **decision on k bits, d,** is a nonconstant function d:$[0,2^k-1]\to\{0,1\}$.

Definition 2.3. The **period of a decision, d,** is the smallest positive integer, t, such that $d(i)=d(i+t)$ for all $i\in[0,2^k-1]$.

Clearly $t = 2^l$, for some l, $1\le l\le k$.

Lemma 2.6. Let d be a decision with period 2^l. Then there exists an i such that $d(i)\ne d(i+2^{l-1})$. Moreover, such an i can be found in $O(2^k)$ time by linear search.

Proof. Immediate from Definition 2.3. ∎

3. The Oracle is Always Correct

We introduce the concept of an oracle for a decision.

Definition 3.1. Let d be a decision on k bits. We say that Θ_d is an **oracle** for d on S_p if for all $x\in S_p$, $\Theta_d(g^x) = d(I^k(x))$.

In this definition the oracle is correct for every g^x that it is given. This section considers what happens if we have such an oracle. In the next section we will consider what happens if the oracle is allowed to make mistakes.

Definition 3.2. Let z be a quadratic residue mod p and $2s$ the unique index of z such that $2s\in[1,p-1]$. Then g^s will be called the **principal square root** of z, and $g^{s+(p-1)/2}$ the non principal root of z. (This definition is taken from [4].)

In [4] it was shown that determining principal square roots is as hard as inverting the discrete log. In the following we will show how to use the information provided by any decision on k bits to solve the principal square root problem. For every j, $1 \le j \le k$, consider the partition of the set of intervals $\{I_i^j\}$ into two types, even and odd, depending on whether i is even or odd. Our method is based on the following observation. The roots of quadratic residues in even (odd) intervals at level j fall in even (odd) intervals at level $j+1$ (except at interval boundaries). Note that principal square roots all belong to the even interval, I_0^1, and non principal roots all belong to the odd interval, I_1^1. A decision gives us information about the intervals at some level. To translate this information to information about the first level we repeatedly take square roots as shown by procedure REDUCE and the following lemmas.

For $z \in [1, p-1]$ and $0 \le l \le k$ define the procedure REDUCE(z, l) as follows:

```
procedure REDUCE($z$,$l$)
  $w \leftarrow z$
  for $i=1$ to $k-l$ do
    if $w$ is a nonresidue then $w \leftarrow wg^{-1}$
    $w \leftarrow \sqrt{w}$ (either root)
  endfor
  return($w$)
```

Since a quadratic nonresidue, g, is known, the extraction of square roots in the above procedure can be performed in time $O(|p|^3)$ ([1],[6]). REDUCE takes time polynomial in $|p|$ (including boundary tests).

Lemma 3.1. Let $z \in [1, p-1]$ and let w = REDUCE(z, l). Let x = index(z) and y = index(w). If $I^k(x) \equiv 0 (\bmod 2^{k-1})$ then $I^k(y) \equiv 0 (\bmod 2^{l-1})$.

Proof. The case $l=k$ is clear. Assume the lemma is true for $l+1$. Let w' = REDUCE$(z, l+1)$ and y' = index(w'). If y' is odd let $w' = w'g^{-1}$ and $y' = y'-1$. (This operation does not change $I^k(y')$ by lemma 2.5.) Let $w = \sqrt{w'}$ and y = index(w). (I.e. w=REDUCE(z, l) and $y' \equiv 2y (\bmod p-1)$). By induction we have $I^k(y') \equiv 0 (\bmod 2^l)$. Then $y' \equiv 2y (\bmod p-1)$ so by Lemma 2.4

$$I^k(y) = \left\lfloor \frac{I^k(y')}{2} \right\rfloor \text{ or } I^k(y) = \left\lfloor \frac{I^k(y')}{2} \right\rfloor + 2^{k-1}.$$

Therefore $I^k(y) \equiv 0 (\bmod 2^{l-1})$. ∎

The next lemma shows how to solve the principal square root problem for integers with index in I_0^k.

Lemma 3.2. Let $z \in \{g^{2x} \mid 2x \in I_0^k\}$, and d be a decision with period 2^l. Suppose we are given Θ_d, an oracle for d. Then we can find the principal square root of z in time polynomial in $|p|$.

Proof. By Lemma 2.6 we can choose an i such that $d(i) \ne d(i+2^{l-1})$ and $0 \le i < 2^{l-1}$. Let z_0 be one of the square roots of z. Let w = REDUCE(z_0, l) and $w_0 = wg^{iR^k}$. We will show that if $\Theta_d(w_0) = d(i)$ then the principal square root of z is z_0 otherwise it is $z_0 g^{(p-1)/2}$. Let x_0 = index(z_0), y = index(w), and y_0 = index(w_0). Since $2x_0 \in I_0^k$, $I^k(x_0) \equiv 0 (\bmod 2^{k-1})$ and $I^k(y) \equiv 0 (\bmod 2^{l-1})$, so $I^k(y) \equiv 0 (\bmod 2^l)$ or $I^k(y) \equiv 2^{l-1} (\bmod 2^l)$. This implies either $I^k(y_0) \equiv i (\bmod 2^l)$ or $I^k(y_0) \equiv i + 2^{l-1} (\bmod 2^l)$. Since $\Theta_d(w_0) = d(I^k(y_0)) = d(I_k(y_0) \bmod 2^l)$, then z_0 is the principal square root of z if and only if $\Theta(w_0) = d(i)$. The time is dominated by the REDUCE procedure which takes polynomial time. ∎

The next lemma shows how to actually find the index, if we know that the index is in I_0^k.

Lemma 3.3. Let $z \in \{g^x \mid x \in I_0^k\}$, and suppose Θ_d is an oracle for a decision, d. Then we can find index(z) in time polynomial in $|p|$.

Proof. Find the bits of the index from right to left as in [4], using lemma 3.2. Note that each time we find a principal root its index remains in I_0^k, so we can apply lemma 3.2 repeatedly. The algorithm follows:

```
procedure INDEX($x$)
  $y \leftarrow x$
  $index \leftarrow \phi$ (empty string)
  while $y \neq 1$ do
    if $y$ is a quadratic residue mod $p$ then
      $index \leftarrow 0\,index$ (concatenate 0 to $index$)
    else
      $y \leftarrow yg^{-1}$
      $index \leftarrow 1\,index$ (concatenate 1 to $index$)
    endif
    $y \leftarrow$ the principal square root of $y$
  endwhile
```

Each use of Lemma 3.2 to extract the principal square root takes polynomial time. There are at most $|p|$ iterations so the total time is again polynomial in $|p|$. ∎

We are now ready for the main theorem.

Theorem 3.1. Let Θ_d be an oracle for some decision d on k bits, where $k = O(\log|p|)$. Then for every $x \in [1,p-1]$ we can determine index(x) in time polynomial in $|p|$ (including boundary tests).

Proof. By Lemma 3.3 we can find the index if it is in the first interval. To do the same for every x we "guess" the interval to which index(x) belongs, shift it to the first interval and apply Lemma 3.3. The algorithm follows:

```
for $i = 0$ to $2^k-1$ do
  $w \leftarrow xg^{iR^k}$ (Guess that index($x$) $\in I_{-i}^k$)
  $candidate \leftarrow$ INDEX($w$)$-iR^k$
  if $g^{candidate} = x$ then stop.
endfor
```

Each iteration can be implemented in polynomial time. There are most a polynomial number of iterations, so the total time is polynomial in $|p|$. ∎

4. An Oracle which is Sometimes Wrong

Now we consider what happens if the oracle is sometimes wrong. If the oracle is correct more often than it is wrong, e.g. answers correctly for 51% of inputs, then our result still holds. We will redefine an oracle as follows:

Definition 4.1. Let d be a decision on k bits and $0<\varepsilon\leq\frac{1}{2}$. Θ_d is an **ε-oracle for d,** if $\Theta_d(g^x) = d(I^k(x))$ for $1-\varepsilon$ of the x's in $I^k(x)$.

Note that this definition requires the ε-oracle to more correct than incorrect on *every* interval and not over the whole range $[1,p-1]$. This is a stronger assumption than the one used in [4], and in fact stronger than what we need. This definition can be modified so that for $k=1$ it gives the definition in [4]. To simplify the presentation, we first use the stronger version.

How can we find $d(I^k(x))$, given only g^x and an ε-oracle that makes mistakes? We use the technique of "concentrating a stochastic advantage" developed in [4]. It is based on the weak law of large numbers.

> If $y_1, y_2, \cdots, y_m$ are m independent 0-1 variables so that $y_i=1$ with probability α, and $S_m = y_1+y_2+\cdots+y_m$, then for positive real numbers ψ and φ, $m > \frac{1}{4\varphi\psi^2}$ implies that $\Pr\left(\left|\frac{S_m}{m} - \alpha\right| > \psi\right) < \varphi$.

Define trials$(\psi,\varphi) = \frac{1}{4\psi\varphi^2}$.

Lemma 4.1. Let $n = \text{trials}(\varepsilon,\delta)$. Let $x_1, x_2, \ldots, x_n$ be randomly chosen elements of I_j^k. Let $z_i = g^{x_i}$. Then given Θ_d we can find $d(j)$ with probability $1-\delta$.

Proof. (Sketch) Let $\alpha = \text{majority}\{\Theta_d(z_i)\}$. By the weak law of large numbers $\alpha = d(j)$ with probability $1-\delta$. ∎

Since we don't know where $z = g^x$ is in the interval, we face the problem of generating random x_i in $I^k(x)$ without falling into neighboring intervals. This can be solved by assuming for the moment that x is in the "beginning" of $I^k(x)$. Later we will consider what to do if it isn't.

Lemma 4.2. Let $n = \text{trials}(\varepsilon,\delta)$. Suppose $x \in \{g^s \mid L_i^k \le s \le L_i^k + R^k/n\}$ for some i. Then we can find $d(i)$ using Θ_d with probability $1-\delta$ in time polynomial in δ^{-1} and ε^{-1}.

Proof. (Sketch) Pick n random elements in $[1,R^k]$, $s_1,s_2,\ldots,s_n$. Let $x_i = xg^{s_i} = g^{s+s_i}$. Since the s_i are uniformly distributed and x is in the beginning of I_i^k, very few of them (in fact, the expected number is one) will cause $x+s_i \in I_{i+1}^k$. Therefore we can still use lemma 4.1 to obtain $d(i)$ with probability $1-\delta$. ∎

Lemma 4.3. Let $n = \text{trials}(\varepsilon,\delta)$. Let $x \in \{g^{2x} \mid 0 \le 2x \le R^k/n\}$ (i.e. $2x$ is in the beginning of the first interval). Then we can find the principal root of x with probability $1-\delta$.

Proof. (Sketch) The proof is the same as for lemma 3.2 except we use lemma 4.2 to evaluate $d(i)$. ∎

Lemma 4.4. Let $n = \text{trials}(\varepsilon,\delta|p|^{-1})$. Let $x \in \{g^x \mid 0 \le x \le R^k/n\}$. Then index($x$) can be computed with probability $1-\delta$ in time polynomial in ε^{-1}, δ^{-1}, and $|p|$.

Proof. (Sketch) Same as lemma 3.3 except we use lemma 4.3 to evaluate the principal square root. Since we find principal square roots at most $|p|$ times, and the probability of success in each is greater than $1-\delta|p|^{-1}$, the total probability of success is greater than $(1-\delta|p|^{-1})^{|p|} > 1-\delta$.

Theorem 4.1. Let Θ_d be an ε-oracle for some decision d on k bits, where $k = O(\log|p|)$. Then for any $x \in [1,p-1]$ we can find index(x) in time polynomial in ε^{-1}, δ^{-1}, and $|p|$ with probability $1-\delta$. ∎

Proof. Let $n = \text{trials}(\varepsilon,\delta|p|^{-1})$. By Lemma 4.4 we can find index(x) for x with index in the "beginning" of the first interval. For other x, we first guess the interval of index(x) and shift it into the first interval as in Theorem 3.1. Then we break the first interval into n subintervals of size roughly $\lfloor R^k/n \rfloor$ each, and guess to which subinterval the index belongs. This subinterval is then shifted to the first subinterval. The algorithm follows:

```
for $i = 0$ to $2^k - 1$ do
    $w \leftarrow xg^{iR^k}$ (Guess that index($x$) $\in I^k_{-i}$)
    for $j = 0$ to $n-1$ do
        $w \leftarrow xg^{j\lfloor R^k/n \rfloor}$ (Guess which subinterval)
        $candidate \leftarrow \text{INDEX}(w) - iR^k - j\left\lfloor \frac{R^k}{n} \right\rfloor$
        if $g^{candidate} = x$ then stop.
    endfor
endfor
```

We use the procedure INDEX from the previous section with the modifications outlined in this section for finding the principle square roots. The total time is polynomial in $|p|$, ε^{-1}, and δ^{-1}. ∎

We conclude this section with a weaker definition of an ε-oracle.

Definition 4.2. Let d be decision on k bits with period 2^l, and let i be so that $d(i) \ne d(i+2^{l-1})$. Let T be the union of the intervals I_j^k such that $j \equiv i \pmod{2^{l-1}}$. Θ_d is an ε-oracle for d if $\Theta_d(g^x) = d(I^k(x))$ for $1-\varepsilon$ of x in the set T.

We leave it to the reader to show that for the case $k=1$ this definition is equivalent to the one given in [4], and that all the results in this section hold with this definition of an ε-oracle.

5. Discussion

Consider again the case where the oracle is always correct. The algorithm in theorem 3.1 essentially gives an oracle reduction from the discrete log problem to the problem of evaluating any decision on the k bits which encode the number of the interval. This reduction takes polynomial time in $|p|$ only if $k=O(\log|p|)$. Is it possible to have a polynomial time reduction for higher values of k? There are two places in the algorithm in which polynomial time in the number of intervals ($=2^k$) is required:

(1) Maintaining a list of the boundaries of the intervals. This is done in order to avoid case 2 of lemmas 2.3, 2.4 and 2.5.

(2) A linear search over all intervals to "guess" in which of them the index is. This is done since lemma 3.2 holds only if the index is in the first interval.

Is there a way around doing (1) and (2)? We can not answer this question in general. There are, however, special cases in which (1) and (2) are not needed.

The problem with the boundaries is simpler when all intervals are of exactly the same size. In particular, if $p=2^m+1$, then case 1 holds in all lemmas 2.3, 2.4 and 2.5 if $k>1$.

When can we avoid enumerating the intervals? Let d be a decision with period 2^l. Lemma 2.6 stated that there must be an interval i for which $d(i)\neq d(i+2^{l-1})$. However, there may be very few such intervals, and in order to use lemma 3.2 we had to know where w was, so that we could shift it to i or to $i+2^{l-1}$. Call a decision *complementary* if for every i, $d(i)\neq d(i+2^{l-1})$. If d is complementary, there is no need to shift w at all, and so lemma 3.2 would hold for every element in Z_p^*. Then there is no need for the linear search over the intervals in the algorithm.

Of course, we cannot use the above to show that the discrete log hides more bits, since not all primes and decisions share those properties. On the other hand, a reduction is a two-way tool: if we could show that there is a partition of S_p into intervals and a complementary decision on the intervals which is easy to evaluate, we would have a polynomial time algorithm for the discrete log in the case $p=2^m+1$. But this is easy to do. Partition S_p into 2^m intervals, i.e. each x is in an interval by itself. Our complementary decision will be $d(x)=1$ if and only if x is odd. $d(x)$ is easily evaluated from g^x by the Euler Criterion. Therefore, by the discussion above, the algorithm of theorem 3.1 can be made polynomial in $|p|$.

This algorithm is somewhat strange and far less natural than the one given in [10]. However it shows that the reduction can be used to solve the discrete log in special cases when an easy bit (x is odd) reveals a hard bit (x is in the first half of S_p). Are there any other cases where this happens?

We conclude by noting that there is nothing magic in partitioning $[1,p-1]$ into 2^k contiguous intervals. Other partitions may yield other hard bits.

References

[1] L. Adleman, K. Manders, G. Miller, "On Taking Roots in Finite Fields", 18th FOCS (1977), 175-178.

[2] L. Adleman, "A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography", 20th FOCS (1979), 55-60.

[3] E. Berlekamp, "Factoring Polynomials Over Large Finite Fields", *Mathematics of Computation*, 24, (1970), 713-735.

[4] M. Blum & S. Micali, "How To Generate Cryptographically Strong Sequences Of Pseudo Random Bits", 23rd FOCS (1982), 112-117.

[5] W. Diffie & M. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, IT-22, 6 (1976), 644-654.

[6] J. Finn, "Probabilistic Methods in Number-Theoretic Algorithms and Digital Signature Schemes", Ph.D. Dissertation, Princeton University, (June 1982).

[7] S. Goldwasser & S. Micali, "Probabilistic Encryption and How to Play Mental Poker Keeping Secret all Partial Information", 14th STOC (1982), 365-377.

[8] K. Ireland & M. Rosen, ***Elements of Number Theory***, Bogden & Quigly, Inc., New York, 1972.

[9] R. Lipton, "How to Cheat at Mental Poker", Unpublished Manuscript, 1979.

[10] S. Pohlig & M. Hellman, " An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance", *IEEE Transactions on Information Theory*, IT-24, 1 (1978), 106-110.

[11] M. Rabin, "Probabilistic Algorithms in Finite Fields", *SIAM Journal of Computing*, 9 No. 2, (May 1980), 273-280.

[12] A. Shamir, R. Rivest, L. Adleman, "Mental Poker", MIT Technical Report (Feb. 1979).

[13] A. Yao, "Theory and Applications of Trapdoor Functions", 23rd FOCS (1982), 80-91.

[14] A. Yao, "Protocols for Secure Computations", 23rd FOCS (1982), 160-164.

421