Fast Parallel Bio-Molecular Logic Computing Algorithms of Discrete Logarithm

Michael (Shan-Hui) Ho and Yu-Ying Shih

Abstract—**The discrete logarithm problem is one of the well-known NP problems. It has important applications in such fields as cryptography. The discrete logarithm problem is the basis for the security of many cryptosystems including the Elliptic Curve Cryptosystem and Diffie-Hellman protocol. In this paper, we proposed newly developed parallel bio-molecular logic computing algorithms based on bio-molecular logic computing model to solve discrete logarithm problem.**

I. Introduction

Diffie & Hellman [1] in 1976 proposed their key exchange protocol. The security of this protocol depends on the discrete logarithm. Like factoring problem, the discrete logarithm problem is believed to be difficult and to be the hard direction of a one-way function. These two are the basis for public key cryptography. The purpose of the algorithm lets two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communication channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

Public-key cryptography based on elliptic curve over finite fields was proposed by Miller and Koblitz in 1985. Elliptic curves over finite fields have been used to implement the Diffie-Hellman key passing scheme and also the elliptic curve variant of the Digital Signature Algorithm. The security of these cryptosystems relies on the difficulty of solving the elliptic curve discrete logarithm problem. If P is a point with order m on an elliptic curve, and Q is some other point on the same curve, then the elliptic curve discrete logarithm problem is to determine an L such that Q = LP where L is an integer and $0 \le L \le m-1$. If this problem can be solved efficiently, then elliptic curve based cryptosystems can be broken efficiently.

Feynman [2] first proposed bio-molecular computations in 1961, but his idea was not experimented with for several decades. In 1994 Adleman [3] succeeded in solving an instance of the Hamiltonian path problem in a test tube by handling DNA strands. Lipton [4] demonstrated that the Adleman techniques could be used to solve the satisfiability problem (the NP-complete problem). Adleman and his co-authors [5] proposed sticker for enhancing the error rate of hybridization.

Manuscript received July 5, 2008.

Michael (Shan-Hui) Ho is with the Department of Information Management, Ming Chuan University, Taoyuan, Taiwan. (phone: 886-3-3507001 ext. 3408; fax: 886-3-3593875; e-mail: mhoincerritos@yahoo.com).

Yu-Ying Shih is with Department of Graduate Institute of Industrial and Business Management, National Taipei University of Technology, Taipei, Taiwan. (e-mail: amy_shyy@yahoo.com.tw).

Through advances in molecular biology [6], it is now possible to produce roughly 10^{18} DNA strands that fit in a test tube. Those 10^{18} DNA strands can also be applied to represent10^{18} bits of information. In the future (perhaps after many years) if biological operations can be applied to deal with a tube with 10^{18} DNA strands and they are run without errors, then 10^{18} bits of information can simultaneously be correctly processed. Hence, it is possible that bio-molecular computation can provide a huge amount of parallelism for dealing with many computationally- intensive problems in the real world.

The fastest supercomputers can execute approximately 10^{12} integer operations per second. This implies that (128×10^{12}) bits of information can be simultaneously processed in a second. The fastest supercomputers can process (128×10^{15}) bits of information in 1000 seconds. The extract operation is one of basic biological operations of the longest execution time. It could be approximately done in 1000 seconds [10]. In the future (perhaps after many years) if an extract operation can be used to deal with a tube with 10^{18} DNA strands and it is run without errors, then 10^{18} bits of information can simultaneously be correctly processed in 1000 seconds. If it becomes true in the future, then basic biological operations will perhaps be faster than the fastest super computer in the future. In [9], it was pointed out that storing information in molecules of DNA allows for an information density of approximately 1 bit per cubic nm (nanometer). Videotape is a kind of traditional storage media and its information density is approximately 1 bit per 10^{12} cubic nanometers. This implies that an information density in molecules of DNA is better than that of traditional storage media.

II. Bio-molecular Computing

A. Biological Operations of Bio-molecular Computing

A (test) tube is a set of molecules of DNA (a multi-set of finite strings over the alphabet {A, C, G, T}). Given a tube, one can perform the following operations:

1. *Extract*: Given a tube T and a short single strand of DNA, "s", produce two tubes + (T, s) and – (T, s), where + (T, s) is all of the molecules of DNA in T which contain the strand "s" as a sub-strand and – (T, s) is all of the molecules of DNA in T which do not contain the short strand "s".
2. *Merge*: Given tubes T_1 and T_2, yield $\cup(T_1, T_2)$, where $\cup(T_1, T_2) = T_1 \cup T_2$. This operation is to pour two tubes into one, with no change of the individual strands.

3. *Amplify*: Given a tube T, the operation, Amplify (T, T_1, T_2), will produce two new tubes T_1 and T_2 so that T_1 and T_2 totally a copy of T (T_1 and T_2 are identical) and T becomes an empty tube.
4. *Append*: Given a tube T and a short strand of DNA, *"s"*, the operation will append the short strand, *"s"*, onto the end of every strand in the tube T. It is denoted by append (T, s).
5. *Append-head*: Given a tube T and a short strand of DNA, *"s"*, the operation will append the short strand, *"s"*, onto the head of every strand in the tube T. It is denoted by append-head (T, s).
6. *Detect*: Given a tube T, say 'yes' if T includes at least one DNA molecule, and say 'no' if it contains none. It is denoted by detect (T).
7. *Discard*: Given a tube T, the operation will discard the tube T. It is denoted by discard (T).

Read: Given a tube T, the operation is used to describe a single molecule, which is contained in tube T. Even if T contains many different molecules each encoding a different set of bases, the operation can give an explicit description of exactly one of them. It is denoted by read (T).

B. *Optimal Bioinformatics Logic Computing System*

In [12, 13], we developed a new bio-molecular logic computing model. We use logic true tables to optimize and complete logic bio-circuit operations that can construct most basic DNA logic circuits. These DNA logic circuits (gates) work in test tubes to implement basic logic operations. These gates are AND, OR, XOR,… etc. Through these logic gates, we construct a set of parallel bio-molecular adder, subtractor, multiplier, and divider. In this paper, we use the new bio-molecular logic computing model to solve the problem of discrete logarithm. All operations of optimal bioinformatics logic computing are shown in Figure 1.

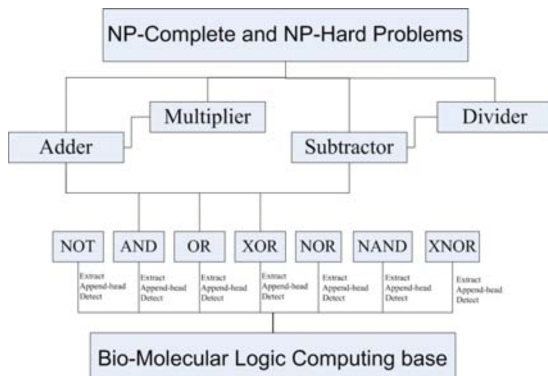

Fig.1. Bio-molecular logic computing model

III. Bio-molecular Logic Computing Solution for NP Problem: Discrete Logarithm

A. *Introduction of Discrete Logarithm Problem*

For any integer d and any positive integer n, there are unique integers s and r such that $0 \le r < n$ and $d = s * n + r$. The value $s = d / n$ is the *quotient* of the division. The value $r = d \bmod n$ is the remainder of the division. We have that $n \mid d$ if and only if $d \bmod n = 0$. Given a well-defined notion of the remainder one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If ($d \bmod n$) = ($b \bmod n$), we write $d \equiv b \pmod n$ and say that d is equivalent to b, modulo n. In other words, $d \equiv b \pmod n$ if d and b have the same remainder divided by n. The integer can be divided into n equivalence classes according to their remainders modulo n. The equivalence class modulo n containing an integer d is $[d]_n = \{d + h * n$, where h is an integer$\}$. The set of all such equivalence classes is $\mathbf{Z}_n = \{[d]_n : 0 \le d \le n - 1\}$. One often sees the definition $\mathbf{Z}_n = \{0, 1, \ldots n - 1\}$ [7].

The *greatest common divisor* of two integers d and n, not both zero, is the largest of the common divisors of d and n; it is denoted gcd(d, n). Two integers d and n are said to be *relatively prime* if their only common divisor is 1, that is, if gcd(d, n) = 1. Because the equivalence class of two integers uniquely determines the equivalence class of their product, thus, we define multiplication modulo n, denoted $*_n$, as follows: $[d]_n *_n [h]_n = [d * h]_n$. Using the definition of multiplication modulo n, we define the multiplicative group modulo n as $(\mathbf{Z}_n^*, *_n)$, where $\mathbf{Z}_n^* = \{[d]_n \in \mathbf{Z}_n: \gcd(d, n) = 1\}$.

Just as it is natural to consider the multiples of a given element d, modulo n, it is often natural to consider the sequence of power of d, modulo n, where $d \in \mathbf{Z}_n$: $d^0, d^1, d^2, \ldots$, modulo n. Indexing from 0, its value in this sequence is $d^0 \bmod n = 1$, and the ith value is $d^i \bmod n$. We denote $<d>$ as the subgroup of $\mathbf{Z}_n^*$ generated by d, and we also denote $\mathrm{ord}_n(d)$ (the "order of d, modulo n") as the order of d in $\mathbf{Z}_n^*$. For example, $<2> = \{1, 2, 4\}$ in $\mathbf{Z}_7^*$, and $\mathrm{ord}_7(2) = 3$.

If $\mathrm{ord}_n(M)$ is equal to the number of elements in $\mathbf{Z}_n^*$, then every element in $\mathbf{Z}_n^*$ is a power of M, modulo n, and we say that M is a *primitive root* or a *generator* of $\mathbf{Z}_n^*$ [7]. For example, there is a primitive root, modulo 7 and $<3> = \{1, 3, 2, 6, 4, 5\}$. If $\mathbf{Z}_n^*$ possesses a primitive root, we say that the group $\mathbf{Z}_n^*$ is *cyclic*. If M is a primitive root of $\mathbf{Z}_n^*$ and C is any element of $\mathbf{Z}_n^*$, then there exists an e such that $M^e \equiv C \pmod n$. This e is called the discrete logarithm of C, modulo n, to the base M. No method in a reasonable amount of time can be applied to solve the problem of discrete logarithm. The following method is used to figure out $M^e \equiv C \pmod n$ [8].

Procedure Encryption(M, e, n)
(1) Let $e_{k-1} \ldots e_0$ be the binary representation of e.
(2) $C = 1$.
(3) **For** $i = k - 1$ **down to** 0
  (3a) Set C to the remainder of (C^2) when divided by n.
  (3b) If $e_i = 1$ then
  (3c) Set C to the remainder of ($C * M$) when divided by n.
  EndFor
(4) Halt. Now C is the result of $M^e \pmod n$.
EndProcedure

Fig.2. Procedure Encryption(M, e, n)

B. Bio-molecular Optimization Solution for Discrete Logarithm Problem

Assume that the length of e is k bits. Also suppose that e is represented as a k-bit binary number, $e_{k-1} \dots e_0$, where the value of each bit e_j is either 1 or 0 for $0 \le j \le k-1$. The bits e_{k-1} and e_0 represent the most significant bit and the least significant bit for e, respectively. The form of an expression, M^e (mod n), can be transformed into another form: $(\dots((1 * M^{e_{k-1}} \text{ (mod } n))^2 * M^{e_{k-2}} \text{ (mod } n))^2 * M^{e_{k-3}} \text{ (mod } n) \dots)^2 * M^{e_0} \text{ (mod } n)$. In the Diffie-Hellman public-key cryptosystem, n is a prime number. Therefore, in this paper, we also assume that n is a prime number. Because n is a prime number, $\langle M \rangle = \{M^0 \text{ (mod } n), M^1 \text{ (mod } n) \dots M^{n-2} \text{ (mod } n)\}$. This is to say that $0 \le e \le n-2$. The following pseudo algorithm is applied to solve the problem of discrete logarithm.

Method 1: Solving the problem of discrete logarithm.
(1) All of the computation for M^0 (mod n), M^1 (mod n) … M^{n-2} (mod n) are simultaneously performed on a parallel molecular computer.
(2) For any given C, from the result finished in Step (1), find $M^e \equiv C$ (mod n).
(3) Output("discrete logarithm is: ", e).
EndMethod

Fig.3. Method 1: Solving the problem of discrete logarithm.

C. DNA Algorithm to Solve Discrete Logarithm Problem

The procedure, **Encryption**(M, e, n), denoted in Subsection A, is used to finish computation of an exponential modular operation. The following DNA algorithm is applied to implement the procedure, **Encryption**(M, e, n).

Algorithm of Discrete Logarithm: Implementing the procedure, **Encryption**(M, e, n).
(0) $T_0 \leftarrow \varnothing$; $T_\theta \leftarrow \varnothing$; $T_n \leftarrow \varnothing$; $T_1 \leftarrow \varnothing$.
(1) **Init**(T_0, k).
(2) **SelectDiscreteLogarithm**(T_0, T_θ, k).
(3) **MakeValue**(T_n, k).
(4) **InitialValue**(T_0, k).
(5) **For** $j = k-1$ **down to** 0
  (5a) **ModularMultiplication**(T_0, T_n, $(2 * (k-1-j)) * (4 * k + 1) + 1$, $2 * (k-j)$, C, C).
  (5b) $T_0 = +(T_0, e_j^1)$ and $T_1 = -(T_0, e_j^1)$.
  (5c) **ModularMultiplication**(T_0, T_n, $(2 * (k-1-j) + 1) * (4 * k + 1) + 1$, $2 * (k-j) + 1$, C, M).
  (5d) **For** $r = 0$ **to** $4 * k$
    (5e) **ReservedValue**(T_1, $(2 * (k-1-j) + 1) * (4 * k + 1) + r$).
  EndFor
  (5f) **AssignmentOperator**(T_1, $(2 * (k-1-j) + 1) * (4 * k + 1) + 1 + 4 * k$, $2 * (k-j) + 1$).
  (5g) $T_0 = \cup(T_0, T_1)$.
EndFor
EndAlgorithmOfDiscreteLogarithm

Fig.4. Algorithm of Discrete Logarithm

Theorem 1: From those steps in **Algorithm of Discrete Logarithm**, the problem of discrete logarithm can be solved.

Proof:

From the execution of Step (0), tubes T_0, T_θ, T_n, and T_1 are set to empty tubes. On the execution of Step (1), it calls **Init**(T_0, k) to construct solution space for 2^k possible discrete logarithms. This means that tube T_0 includes strands encoding 2^k possible discrete logarithms. Next, the execution of Step (2) calls **SelectDiscreteLogarithm**(T_0, T_θ) to perform selection of legal discrete logarithms with its range is from 0 to $n-2$. This implies that those legal discrete logarithms are encoded in tube T_0. On the execution of Step (3), it calls **MakeValue**(T_n) to encode a prime number, n. This indicates that tube T_n contains a strand encoding it. Next, the execution of Step (4) calls **InitialValue**(T_0) to finish the execution of Step (2) in the procedure, **Encryption**(M, e, n). This is to say that the initial value for C is set to one.

Step (5) is a loop and is mainly used to finish the function of the only loop (Step (3)) in the procedure, **Encryption**(M, e, n). Next, the first execution of Step (5a) calls **ModularMultiplication**(T_0, T_n, $(2 * (k-1-j)) * (4 * k + 1) + 1$, $2 * (k-j)$, C, C) to perform Step (3a) in **Encryption**(M, e, n). On the first execution of Step (5b), it employs the *extract* operation to form two tubes: T_0 and T_1. The first tube T_0 includes all of the strands that have $e_j = 1$. The second tube T_1 consists of all of the strands that have $e_j = 0$. This indicates that the execution of the step finishes Step (3b) in **Encryption**(M, e, n). Because the jth bit of e encoded in tube T_0 is one, next, the first execution of Step (5c) calls **ModularMultiplication**(T_0, T_n, $(2 * (k-1-j) + 1) * (4 * k + 1) + 1$, $2 * (k-j) + 1$, C, M) to perform Step (3c) in **Encryption**(M, e, n). Since the jth bit of e encoded in tube T_1 is zero, Step (5d) is the loop and is mainly used to maintain the consistency of the intermediate value for Y. On the first execution of Step (5e), it calls **ReservedValue**(T_1, $(2 * (k-1-j) + 1) * (4 * k + 1) + r$) to copy the current intermediate value of Y to the next intermediate value of Y. Repeat to execute Step (5e) until the value of r reaches $(4 * k)$. Next, the first execution of Step (5f) calls **AssignmentOperator**(T_1, $(2 * (k-1-j) + 1) * (4 * k + 1) + 1 + 4 * k$, $2 * (k-j) + 1$) to perform updating of the value for C. Because the jth bit of e encoded in tube T_1 is zero, the updated value of C is still equal to the previous value.

On the first execution of Step (5g), it uses the *merge* operation to pour tube T_1 into T_0. Repeat execution of Steps (5a) through (5g) until the value of j is zero. After all of the steps are processed, every strand in tube T_0 performs computation of an exponential modular operation, M^e (mod n). This implies that **Algorithm of Discrete Logarithm** performs Step (1) of **Method 1**. Therefore, the discrete logarithm problem can be solved from those steps in **Algorithm of Discrete Logarithm**. In the following section, we will describe, in detail, the various modules that are combined to form the overall DNA-based algorithm for solving the discrete logarithm problem.

D. Solution Space for $\text{Order}_n(M)$

Because $\text{Order}_n(M)$ is equal to $n-1$, suppose that $n-1$ is

represented as a k-bit binary number, $\theta_{k-1} \ldots \theta_0$, where the value of each bit θ_j is either 1 or 0 for $0 \le j \le k-1$. The bits θ_{k-1} and θ_0 are used to represent the most significant bit and the least significant bit for $n-1$, respectively. From [9, 10], for every bit θ_j, two *distinct* 15 base value sequences are designed. One represents the value "0" for θ_j and the other represents the value "1" for θ_j. For the sake of convenience in our presentation, assume that θ_j^1 denotes the value of θ_j to be 1 and θ_j^0 defines the value of θ_j to be 0. The following algorithm, **SelectDiscreteLogarithm**(T_0, T_θ), is proposed to construct a DNA strand for encoding $n-1$ and select legal discrete logarithms.

```
Procedure SelectDiscreteLogarithm(T0, Tθ, k)
  (1) For j = 0 to k – 1
        (1a) Append-head(Tθ, θj).
      EndFor
  (2) For j = k – 1 down to 0
        (2a) T0^ON = +(T0, ej^1) and T0^OFF = –(T0, ej^1).
        (2b) Tθ^ON = +(Tθ, θj^1) and Tθ^OFF = –(Tθ, θj^1).
        (2c) If (Detect(Tθ^ON) = = true) then
               (2d) T0^= = ∪(T0^=, T0^ON) and T0^< = ∪(T0^<, T0^OFF).
             Else
               (2e) T0^> = ∪(T0^>, T0^ON) and T0^= = ∪(T0^=, T0^OFF).
             EndIf
        (2f) Tθ = ∪(Tθ^ON, Tθ^OFF).
        (2g) Discard(T0^>).
        (2h) T0 = ∪(T0, T0^=).
      EndFor
  (3) Discard(T0).
  (4) T0 = ∪(T0, T0^<).
EndProcedure
```

Fig.5. Procedure SelectDiscreteLogarithm(T_0, T_θ, k)

E. Solution Space for MODULE n

Assume that the length of n denoted in Subsection A is k bits. Also suppose that n is represented as a k-bit binary number, $n_{k-1} \ldots n_0$, where the value of each bit n_j is either 1 or 0 for $0 \le j \le k-1$. The bits n_{k-1} and n_0 represent the most significant bit and the least significant bit for n, respectively. From [9, 10], for every bit n_j, two *distinct* 15 base value sequences are designed. One represents the value "0" for n_j and the other represents the value "1" for n_j. For the sake of convenience in our presentation, assume that n_j^1 denotes the value of n_j to be 1 and n_j^0 defines the value of n_j to be 0. The following algorithm, **MakeValue**(T_n), is proposed to construct a DNA strand for encoding n.

```
Procedure MakeValue(Tn, k)
(1) For j = 0 to k – 1
      (1a) Append-head(Tn, nj).
EndFor
EndProcedure
```

Fig.6. Procedure MakeValue(T_n, k)

F. Solution Space for a Primitive Root M and the Result of an Exponential Modular Operation C

Suppose that the length of a primitive root M for $\mathbf{Z}_n^*$ is k bits. Also assume that M is represented as a k-bit binary number, $m_{k-1} \ldots m_0$, where the value of each bit m_j is either 1 or 0 for $0 \le j \le k-1$. The bits m_{k-1} and m_1 represent the most significant bit and the least significant bit for M, respectively. From [9, 10], for every bit m_j, two *distinct* 15 base value sequences are designed. One represents the value "0" for m_j and the other represents the value "1" for m_j. For the sake of convenience in our presentation, assume that m_j^1 denotes the value of m_j to be 1 and m_j^0 defines the value of m_j to be 0.

Assume that the length of C, the result of an exponential modular operation denoted in Subsection A, is k bits. From the procedure **Encryption**(M, e, n), C is finally obtained after at most updating $(2 * k + 1)$ times of the value for C. Therefore, suppose that C is represented as a k-bit binary number, $c_{a,k-1} \ldots c_{a,0}$, where the value of each bit $c_{a,j}$ is either 1 or 0 for $1 \le a \le (2 * k + 1)$ and $0 \le j \le k-1$. The bits, $c_{a,k-1}$ and $c_{a,0}$, represent the most significant bit and the least significant bit for C, respectively. The first k-bit binary number, $c_{1,k-1} \ldots c_{1,0}$, is used to represent the initial value to C. The last k-bit binary number, $c_{(2*k+1),k-1} \ldots c_{(2*k+1),0}$, is used to represent the final result of C. For other k-bit binary numbers, they are applied to represent the intermediate computed form of C. From [9, 10], for every bit $c_{a,j}$, two *distinct* 15 base value sequences were designed. One represents the value "0" for $c_{a,j}$ and the other represents the value "1" for $c_{a,j}$. For the sake of convenience in our presentation, assume that $c_{a,j}^1$ denotes the value of $c_{a,j}$ to be 1 and $c_{a,j}^0$ defines the value of $c_{a,j}$ to be 0. The following algorithm is used to construct solution space for the initial value for C and the primitive root M.

```
Procedure InitialValue(T0, k)
(1) For j = 0 to k – 1
   (1a) Append-head(T0, mj).
EndFor
(2) Append-head(T0, c1,0^1).
(3) For j = 1 to k – 1
   (3a) Append-head(T0, c1,j^0).
EndFor
EndProcedure
```

Fig.7. Procedure InitialValue(T_0, k)

G. Algorithm of a Modular Multiplication

The procedure, **Encryption**(M, e, n), denoted in Subsection A, is used to finish computation of an exponential modular operation. In the procedure, it uses successive operations of square and multiplication to perform the exponential modular operation. We now give details of the **ModularMultiplication**(T_0, T_n, f, a, α, β) module used by the main algorithm. The following DNA-based algorithm, **ModularMultiplication**(T_0, T_n, f, a, α, β), is applied to perform all of the steps to a modular multiplication. This implies that Steps (3a) and (3c) in the procedure, **Encryption**(M, e, n), are performed through the following DNA-based algorithm, **ModularMultiplication**(T_0, T_n, f, a,

α, β). The two parameters, α and β, in **ModularMultiplication**(T_0, T_n, f, a, α, β) represent the multiplicand and the multiplier of a modular multiplication. Assume that β_j^1 is applied to represent the value of "1" for the jth bit of the multiplier (β).

Procedure **ModularMultiplication**(T_0, T_n, f, a, α, β)
(1) **InitialSet**(T_0, f).
(2) **For** $j = k - 1$ **down to** 0
(2a) **ParallelLeftShifter**(T_0, $f + (k - 1 - j) * 4$).
(2b) **ParallelComparator**(T_0, T_n, $T_0^>$, $T_0^=$, $T_0^<$, $f + (k - 1 - j) * 4 + 1$).
(2c) $T_0 = \cup(T_0^>, T_0^=)$.
(2d) **ParallelSubtractor**(T_0, T_0, $T_{f + (k - 1 - j) * 4 + 1}$).
(2e) **ReservedValue**($T_0^<$, $f + (k - 1 - j) * 4 + 1$).
(2f) $T_0 = \cup(T_0, T_0^<)$.
(2g) $T_0 = +(T_0, \beta_j^1)$ and $T_1 = -(T_0, \beta_j^1)$.
(2h) **If** (Detect(T_0) = = *true*) **then**
(2i) **ParallelAdder**(T_0, $T_{f + (k - 1 - j) * 4 + 2}$, T_a).
(2j) **ParallelComparator**(T_0, T_n, $T_0^>$, $T_0^=$, $T_0^<$, $f + (k - 1 - j) * 4 + 3$).
(2k) $T_0 = \cup(T_0^>, T_0^=)$.
(2l) **ParallelSubtractor**(T_0, T_0, $T_{f + (k - 1 - j) * 4 + 3}$).
(2m) **ReservedValue**($T_0^<$, $f + (k - 1 - j) * 4 + 3$).
(2n) $T_0 = \cup(T_0, T_0^<)$.
EndIf
(2o) **If** (Detect(T_1) = = *true*) **then**
(2p) **ReservedValue**(T_1, $f + (k - 1 - j) * 4 + 2$).
(2q) **ReservedValue**(T_1, $f + (k - 1 - j) * 4 + 3$).
EndIf
(2r) $T_0 = \cup(T_0, T_1)$.
EndFor
(2s) **AssignmentOperator**(T_0, $f + k * 4$, a).
EndProcedure.

Fig.8. Procedure ModularMultiplication(T_0, T_n, f, a, α, β)

H. Solution Space for the Initial Value to Computation of a Modular Multiplication

For any given two positive integers d and b, Blakley [11] proposed the fastest method to perform computation of $(d * b) \pmod n$. Blakley used adder and subtractor of $(4 * k)$ times to perform computation of $(d * b) \pmod n$. Assume that $Y \equiv (d * b) \pmod n$ and the length of Y is k bits. From Blakley's method, Y is finally obtained after at most updating $(4 * k + 1)$ times of the value for Y. From the procedure **Encryption**(M, e, n), Blakley's method is at most called $(2 * k)$ times. That is to say, at most updating $(8 * k^2 + 2 * k)$ times of the value for Y are completed. Therefore, suppose that Y is represented as a k-bit binary number, $y_{f, k-1} \dots y_{f, 0}$, where the value of each bit $y_{f, g}$ is either 1 or 0 for $1 \le f \le (8 * k^2 + 2 * k)$ and $0 \le g \le k - 1$. The bits, $y_{f, k-1}$ and $y_{f, 0}$, represent the most significant bit and the least significant bit for Y, respectively. If updating of fth time for Y is finished through an adder, then two binary numbers $y_{f, k-1} \dots y_{f, 0}$ and $y_{f+1, k-1} \dots y_{f+1, 0}$ represent the augend and the sum of the fth updating, respectively. If updating of fth time for Y is finished through a subtractor, then two binary numbers $y_{f, k-1} \dots y_{f, 0}$ and $y_{f+1, k-1} \dots y_{f+1, 0}$ represent the minuend and the difference of the fth updating, respectively.

From [9, 10], for every bit $y_{f, g}$, two *distinct* 15 base value sequences were designed. One represents the value "0" for $y_{f, g}$ and the other represents the value "1" for $y_{f, g}$. For the sake of convenience in our presentation, assume that $y_{f, g}^1$ denotes the value of $y_{f, g}$ to be 1 and $y_{f, g}^0$ defines the value of $y_{f, g}$ to be 0. The following algorithm is used to construct solution space for the initial value to computation of a modular multiplication.

Procedure InitialSet(T_0, f)
(1) **For** $g = 0$ **to** $k - 1$
(1a) Append-head(T_0, $y_{f, g}^0$).
EndFor
EndProcedure

Fig.9. Procedure InitialSet(T_0, f)

I. Reserving the Result to Intermediate Computation of a Modular Multiplication

The procedure, **Encryption**(M, e, n), denoted in Subsection A, is applied to perform computation of an exponential modular operation. The following DNA-based algorithm, **ReservedValue**(T_2, f), is employed to reserve the result to intermediate computation of a modular multiplication.

Procedure ReservedValue(T_2, f)
(1) **For** $j = 0$ **to** $k - 1$
(1a) $T_3 = +(T_2, y_{f, j}^1)$ and $T_4 = -(T_2, y_{f, j}^1)$.
(1b) Append-head(T_3, $y_{f+1, j}^1$).
(1c) Append-head(T_4, $y_{f+1, j}^0$).
(1d) $T_2 = \cup(T_3, T_4)$.
EndFor
EndProcedure

Fig.10. Procedure ReservedValue(T_2, f)

J. Construction of Assignment Operator

An assignment operator is an instruction of the first operand of k bits and the second operand of k bits that the value of the first operand is set to the value of the second operand. The following algorithm is applied to construct an assignment operator. This implies that the assignment operator can be used to update the value of C denoted in Subsection G. The third parameter, a, in the algorithm is used to represent the ath updating for C.

Procedure AssignmentOperator(T_0, f, a)
(1) **For** $j = 0$ **to** $k - 1$
(1a) $T_1 = +(T_0, y_{f, j}^1)$ and $T_2 = -(T_0, y_{f, j}^1)$.
(1b) Append-head(T_1, $c_{a, j}^1$).
(1c) Append-head(T_2, $c_{a, j}^0$).
(1d) $T_0 = \cup(T_1, T_2)$.
EndFor
EndProcedure

Fig.11. Procedure AssignmentOperator(T_0, f, a)

K. The Attacking Plan of Breaking the Diffie-Hellman Public-key Cryptosystem

The Diffie-Hellman public-key cryptosystem can be

used to encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode them. Assume that the public key between two parties is represented as a k-bit binary number, $c_{(2 * k + 1), k-1} \ldots c_{(2 * k + 1), 0}$, denoted in Subsection G. An eavesdropper only uses the following algorithm to figure out the corresponding secret key.

Algorithm: The attacking plan of breaking the Diffie-Hellman public-key cryptosystem.
(1) Call **Algorithm of Discrete Logarithm**.
(2) **For** $j = 0$ **to** $k - 1$
 (2a) $T_1 = +(T_0, c_{(2 * k + 1), j})$ and $T_2 = -(T_0, c_{(2 * k + 1), j})$.
 (2b) $T_0 = \cup(T_0, T_1)$.
 EndFor
(3) **If** (Detect(T_0) = = *true*) **then**
 (3a) Read(T_0).
 EndIf
EndAlgorithm

Fig.12. Algorithm: The attacking plan of breaking the Diffie-Hellman public-key cryptosystem

L. Complexity Assessment

Theorem 2: Suppose that the length of a secret key (discrete logarithm) in the Diffie-Hellman public-key cryptosystem is k bits. Its public-key cryptosystem can be broken with $O(k^3)$ biological operations proposed by Adleman [3, 9, 10] from solution space.
Proof: Refer to **Algorithm of Discrete Logarithm**.

Theorem 3: Suppose that the length of a secret key (discrete logarithm) in the Diffie-Hellman public-key cryptosystem is k bits. The Diffie-Hellman public-key cryptosystem can be broken with $O(2^k)$ library strands from solution space.
Proof: Refer to **Algorithm of Discrete Logarithm**.

Theorem 4: Suppose that the length of a secret key (discrete logarithm) in the Diffie-Hellman public-key cryptosystem is k bits. The Diffie-Hellman public-key cryptosystem can be broken with O(1) tubes from solution space.
Proof: Refer to **Algorithm of Discrete Logarithm**.

Theorem 5: Suppose that the length of a secret key (discrete logarithm) in the Diffie-Hellman public-key cryptosystem is k bits. The Diffie-Hellman public-key cryptosystem can be broken with the longest library strand, $O(k^3)$, from solution space.
Proof: Refer to **Algorithm of Discrete Logarithm**.

IV. Conclusions

The number of steps any classical computer requires in order to find discrete logarithm of a k-bit increases exponentially with k, at least by means of using algorithms [3] known at present. In this paper, Our *molecular* discrete logarithm algorithm demonstrates how basic biological operations can be used to solve discrete logarithm problem with $O(k^3)$ biological operations. It can simultaneously deal with 2^{1024} bit information to find the discrete logarithm of 1024 bits used in the Diffie-Hellman public-key cryptosystem. Due to current technical difficulties, the proposed algorithm currently does not in fact find the discrete logarithm of 1024 bits. This implies that if a molecular computer is *really* constructed in the future (perhaps after many years), then our discrete logarithm algorithm has very high feasibility for solving the discrete logarithm problem.

References

[1] Diffie W., and Hellman M., "New directions in cryptography," *IEEE Transaction on Information Theory*, IT-22, 6, 1976, pp. 644-654.
[2] R. P. Feynman. "In minaturization," *D.H. Gilbert, Ed*., Reinhold Publishing Corporation, New York, 1961, pp. 282-296.
[3] L. Adleman, "Molecular computation of solutions to combinatorial problems," *Science*, 266:1021-1024, Nov. 11, 1994.
[4] R. J. Lipton, "DNA solution of hard computational problems," *Science*, 268:542:545, 1995.
[5] S. Roweis, E. Winfree, R. Burgoyne, N. V. Chelyapov, M. F. Goodman, Paul W.K. Rothemund and L. M. Adleman, "A sticker based model for DNA computation," *2nd annual workshop on DNA Computing*, Princeton University. Eds. L. Landweber and E. Baum, DIMACS: series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999, pp. 1-29.
[6] J. Watson, N. Hoplins, J. Roberts, A. Gann, M. Levine, and R. Losick, *Molecular Biology of the Gene*, Benjamin/Cummings Menlo Park CA, 1987.
[7] Koblitz N., "A course in number theory and cryptography," *Springer-Verlag*, 1987, ISBN 0387942939.
[8] Rivest R. L., Shamir A., and Adleman L. 1978. "A method for obtaining digital signatures and public-key cryptosystem," *Communication of the ACM*, Volume 21, pp. 120-126.
[9] Braich R. S., Johnson C., Rothemund P. W. K., Hwang D., Chelyapov N., and Adleman L. M., "Solution of a satisfiability problem on a gel-based DNA computer," *Proceedings of the 6th International Conference on DNA Computation in the Springer-Verlag Lecture Notes in Computer Science series*, 2000, pp. 27-41.
[10] Adleman L. M., Braich R. S., Johnson C., Rothemund P. W.K., Hwang D., and Chelyapov N., "Solution of a 20-variable 3-SAT problem on a DNA computer," *Science*, Volume 296, Issue 5567, 2002, pp. 499-502.
[11] Blakley G. R., "A computer algorithm for calculating product AB modulo M," *IEEE Transaction on Computer*, Vol. c-32, No. 5,1983, pp 497-500.
[12] Yu-Ying Shi, Michael(Shan-Hui) Ho, Yu-Jen Wang, and Chun-Yu Huang, "Constructing bio-molecular parallel adder and multiplier with basic logic operations in the Adleman-Liption model," in *ICSC &ISIS*, 2008 (Submitted for publication).
[13] Yu-Ying Shi, Michael(Shan-Hui) Ho, Chun-Yu Huang, and Yu-Jen Wang, "Constructing bio-molecular parallel subtractor and divider with basic logic operations in the Adleman-Liption model," in *ICSC &ISIS*, 2008(Submitted for publication).