J Supercomput (2012) 61:642–672
DOI 10.1007/s11227-011-0627-z

Molecular solutions of the RSA public-key cryptosystem on a DNA-based computer

Weng-Long Chang · Kawuu Weicheng Lin · Ju-Chin Chen · Chih-Chiang Wang · Lai Chin Lu · Minyi Guo · Michael (Shan-Hui) Ho

Published online: 31 May 2011

Abstract The RSA public-key cryptosystem is an algorithm that converts a plain-text to its corresponding cipher-text, and then converts the cipher-text back into its corresponding plain-text. In this article, we propose five DNA-based algorithms—parallel adder, parallel subtractor, parallel multiplier, parallel comparator, and parallel modular arithmetic—that construct molecular solutions for any (plain-text, cipher-text) pair for the RSA public-key cryptosystem. Furthermore, we demonstrate that an

W.-L. Chang (✉) · K.W. Lin · J.-C. Chen · C.-C. Wang
Department of Computer Science and Information Engineering, National Kaohsiung University of Applied Sciences, No 415, Chien Kung Road, Kaohsiung 807, Taiwan, R.O.C.
e-mail: changwl@cc.kuas.edu.tw

K.W. Lin
e-mail: linwc@cc.kuas.edu.tw

J.-C. Chen
e-mail: joan@csie.ncku.edu.tw

C.-C. Wang
e-mail: Steven.cc.wang@gmail.com

L.C. Lu
Department of Industrial and Management Engineering, National Kaohsiung University of Applied Sciences, No 415, Chien Kung Road, Kaohsiung 807, Taiwan, R.O.C.
e-mail: rachel@cc.kuas.edu.tw

M. Guo
Department of Computer Software, The University of Aizu, Aizu-Wakamatsu City, Fukushima 965-8580, Japan
e-mail: minyi@u-aizu.ac.jp

M.(S.-H.) Ho
Computer Center and Institute of Electrical Engineering, National Taipei University, 151, University Rd., San Shia 237, Taipei County, Taiwan, R.O.C.
e-mail: MHoInCerritos@yahoo.com

eavesdropper can decode an encrypted message overheard with the linear steps in the size of the encrypted message overheard.

Keywords The RSA public-key cryptosystem · Plain-text · Cipher-text · Biomolecular computing · DNA-based computing

1 Introduction

The RSA public-key cryptosystem [1] is the primary cryptosystem used for security on the Internet and World Wide Web. Feynman [2] first offered molecular computations in 1961, but his idea was implemented 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. DNA-based algorithms had been offered to solve many computational problems, and these contained satisfiability [4], maximal clique [5], three-vertex-colouring [6], parallelism of three molecular operations by dealing with several natural questions [7], the energy barrier problem without pseudoknots and temporary arcs [8], the optimization problem nucleic acid sequence design [9], the maximum cut problem [10], and the binary integer programming problem [11]. From [12], several circuits that amplify nucleic acid signals were constructed and characterized. One potentially significant area of application for DNA algorithms is the breaking of encryption schemes [13, 14]. In [15], they introduced methods for controlling the asymptotic turnover of strand displacement-based DNA catalysts and showed how these could be used to construct linear classifier systems. From [16], the DNA-based algorithms are proposed to perform molecular verification of rule-based systems. From [17–19] DNA-based arithmetic algorithms are proposed, and from [20] DNA-based algorithms for constructing DNA databases are also offered.

DES (the United States Data Encryption Standard) is one of the most widely used cryptographic systems. It produces a 64-bit ciphertext from a 64-bit plaintext under the control of a 56-bit key. A cryptanalyst obtains a plaintext and its corresponding ciphertext and wishes to determine the key used to perform the encryption. The most naive approach to this problem is to try all 256 keys, encrypting the plaintext under each key until a key that produces the ciphertext is found and is called the plaintext-ciphertext attack. Adleman and his coauthors [14] provided a description of such an attack using the sticker model of molecular computation. Start with approximately 256 identical ssDNA memory strands each 11,580 nucleotides long. Each memory strand contains 579 contiguous blocks each 20 nucleotides long. As it is appropriate in the sticker model, there are 579 stickers—one complementary to each block. Memory strands with annealed stickers are called memory complexes. When the 256 memory complexes have half of their sticker positions occupied at the end of the computation, they weigh approximately 0.7 g and, in solution at 5 g/liter, would occupy approximately 140 ml. Hence, the volume of the 1303 tubes needs be no more than 140 ml each. It follows that the 1303 tubes occupy, at most, 182 L and can, for example, be arrayed in 1 m long and wide and 18 cm deep.

Adleman and his coauthors [14] indicated that at the end of computation for breaking DES, 256$\times$ (56 key bits + 64 ciphertext bits) pairs were generated and processed.

Adleman and his coauthors [14] also pointed out that this codebook for breaking DES has approximately 263 (8 × 1018) bits of information (the equivalent of approximately one billion 1 gigabyte CDs). The actual running time for the algorithm of breaking DES depends on how fast the operations can be performed. If each operation requires 1 day, then the computation for breaking DES will require 18 years. If each operation requires 1 hour, then the computation for breaking DES will require approximately 9 months. If each operation can be completed in 1 minute, then the computation for breaking DES will take 5 days. Finally, if the effective duration of a step can be reduced to 1 second, then the effort for breaking DES will require 2 hours. While it has been argued that special purpose electronic hardware [14] or massively parallel supercomputers (the IBM Blue Gene/L machine is capable of 183.5 TFLOPS or 183.5 × 1012 floating-point operations per second) might be used to break DES in a reasonable amount of time, it appears that today's most powerful sequential machines would be unable to accomplish the task.

The rest of the paper is organized as follows: in Sect. 2, we introduce DNA models of computation proposed by Adleman and his coauthors in detail. In Sect. 3, we give a high-level description of our DNA-based algorithm for the encryption function. By breaking this down into submodules in Sect. 4, we prove the operation of the various novel algorithms for arithmetic, shifted and comparative operations. In Sect. 5, based on our DNA-based algorithm to the encryption function, we also give a high-level description of our DNA-based algorithm for finding the corresponding plain-text from a cipher-text. In Sect. 6, we demonstrate that the time complexity of our DNA-based algorithm is cubic on the input size. In Sect. 7, we show how the basic operations within our model may be implemented by means of using standard laboratory operations on DNA strands. In Sect. 8, we conclude with a brief discussion.

2 Background

In this section, we review the basic structure of the DNA molecule and then discuss available techniques for dealing with DNA that will be used to figure out any (plain-text, cipher-text) pair in the RSA public-key cryptosystem.

2.1 The structure of DNA

From [6, 21, 22], it is indicated that DNA (deoxyribonucleic acid) encodes the genetic information of cellular organisms. It includes polymerchains, commonly referred to as DNA strands. Strands may be synthesized to order by an automated process. Each strand may be viewed as a sequence of nucleotides, or bases, attached to a sugar-phosphate "backbone." The four DNA nucleotides are adenine, guanine, cytosine, and thymine, commonly abbreviated to A, G, C, and T, respectively. Each strand has, in light of chemical convention, a $5'$ and a $3'$ end, thus any single strand has a natural orientation. This orientation (and, hence, the notation used) is due to fact that one end of the single strand has a free (i.e., unattached to another nucleotide) $5'$ phosphate group, and the other has a free $3'$ deoxyribose hydroxyl group. The classical double helix of DNA is formed when two separate strands bond. Bonding occurs by

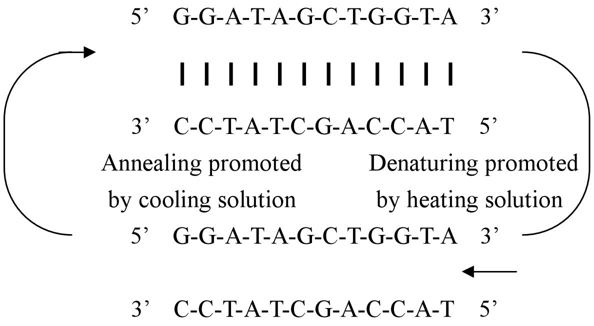
5' G-G-A-T-A-G-C-T-G-G-T-A 3'
3' C-C-T-A-T-C-G-A-C-C-A-T 5'
Annealing promoted by cooling solution
Denaturing promoted by heating solution
5' G-G-A-T-A-G-C-T-G-G-T-A 3'
3' C-C-T-A-T-C-G-A-C-C-A-T 5'

Fig. 1 DNA denaturing and annealing

the pairwise attraction of bases; A bonds with T and G bonds with C. The pairs (A, T) and (G, C) are therefore known as complementary base pairs. Double-stranded DNA may be dissolved into single strands (or denatured) by heating the solution to a temperature determined by the composition of the strand [6, 21, 22]. Heating breaks the hydrogen bonds between complementary strands (Fig. 1) from [6, 23]. Since a G–C pair is joined by three hydrogen bonds, the temperature required to break it is slightly higher than that for an A–T pair, joined by only two hydrogen bonds. This factor must be taken into account when designing sequences to represent computational elements. Annealing is the reverse of melting, whereby a solution of single strands is cooled, and allowing complementary strands to bind together (Fig. 1) from [6, 23]. In double-stranded DNA, if one of the single strands contains a discontinuity (i.e., one nucleotide is not bonded to its neighbor) then this may be repaired by DNA ligase from [6]. This allows us to create a unified strand from several bound together by their respective complements.

2.2 Adleman's experiment for solution of a satisfiability problem

Adleman and his coauthors [24, 25] performed experiments that were applied to, respectively, solve a 6-variable 11-clause formula and a 20-variable 24-clause 3-conjunctive normal form (3-CNF) formula. A Lipton encoding [4] was used to represent all possible variable assignments for the chosen 6-variable or 20-variable SAT problem. For each of the 6 variables $x_1, \ldots, x_6$, two distinct 15 base value sequences were designed. One represents true (T), $x_{k\mathrm{T}}$, and another represents false (F), $x_{k\mathrm{F}}$ for $1 \le k \le 6$. Each of the 26 truth assignments was represented by a library sequence of 90 bases consisting of the concatenation of one value sequence for each variable. DNA molecules with library sequences are termed library strands and a combinatorial pool containing library strands is termed a library. The 6-variable library strands were synthesized by employing a mix-and-split combinatorial synthesis technique [24]. The library strands were assigned library sequences with x_1 at the 5′-end and x_6 at the 3′-end ($5' - x_1 - x_2 - x_3 - x_4 - x_5 - x_6 - 3'$). Thus, synthesis began by assembling the two 15 base oligonucleotides with sequences $x_{6\mathrm{T}}$ and $x_{6\mathrm{F}}$. This process was repeated until all 6 variables had been treated. The similar method also is applied to solve a 20-variable of 3-SAT [25].

2.3 DNA manipulations

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

1. *Separate*(T, s, T_1, T_2). Given a tube T and a short strand of DNA, s, the operation produces two new tubes T_1 and T_2. Tube T_1 contains all of the molecules of DNA in the tube T which includes s as a sub-strand and tube T_2 consists of all of the molecules of DNA in the tube T which does not contain s as a substrand.
2. *Merge*$(\{T_i\})$. Given any n tubes $T_1, \ldots, T_n$, the operation yields *Merge*$(T_1, \ldots, T_n) = \bigcup_{i=1}^{n} T_i = T_1 \cup T_2 \cdots \cup T_n$. This implies that it is to pour any n tubes into one, without any change in the individual strands.
3. *Discard*(T). The operation sets T to be an empty set ($T \leftarrow \varnothing$).
4. *Amplify*$(T, \{T_i\})$. Given a tube T, the operation produces a number of identical copies, T_i, of tube T, and then *discard*(T).
5. *Concatenate*(s_1, s_2). Given two strands of DNA, s_1 and s_2, the operation returns a new strand of DNA, comprised of the concatenation of s_1 and s_2. If s_1 is a null strand of DNA, return s_2, and if s_2 is a null strand of DNA, return s_1.
6. *Appendhead*(T, s). Given a nonempty tube T and a short strand of DNA, s, the operation first creates a null tube, U, and then, in parallel, for each string $t_i \in T$ finishes the following: $U \leftarrow$ *Merge*$(U,$ *Concatenate*$(s, t_i))$ and return U. If T is initially empty, then U contains only s.
7. *Detect*(T). The operation returns *true* if T includes at least one DNA molecule ($T \neq \varnothing$), otherwise returns *false*.
8. *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.

3 Our DNA-based algorithm for the encryption function in the RSA public-key cryptosystem

In the RSA cryptosystem [1], the transformation of a plain-text M, whose value is less than the value of n, associated with a public key $P = (e, n)$ is $P(M) = Me \pmod{n}$. The transformation of a cipher-text C associated with a secret key $S = (d, n)$ is $S(C) = Cd \pmod{n}$. The encryption function and the decryption function, $P(M) = Me \pmod{n}$ and $S(C) = Cd \pmod{n}$, are used to finish encryption and decryption for a message, M. From [1], the following procedure is applied to compute encryption, $Me \pmod{n}$. Decryption, $Cd \pmod{n}$, can be performed similarly using d and C instead of e and M.

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

Assume that the length of the public-key, e, in the RSA public-key cryptosystem is k bits, and it can be represented as a k-bit binary number, $e_{k-1} \dots e_0$, where the value of each bit e_i is either 1 or 0 for $0 \le i \le k-1$. For every bit e_i, two distinct 15 base value sequences are designed to respectively represent the value "0" for e_i and the value "1" for e_i. For the sake of convenience in our presentation, assume that e_i 1 denotes the value of e_i to be 1 and e_i 0 defines the value of e_i to be 0. The following DNA algorithm is applied to first construct the solution space of every plaintext, then to implement the encryption function Encryption(M, e, n), and finally to construct the correspondence of between plain-text and cipher-text.

```
**Algorithm 1** Implementing the procedure, **Encryption**$(M, e, n)$.
(1) **MakeValue**$(T_n)$.
(2) **Init**$(T_0, T_n)$.
(3) **MakeInitialValue**$(T_e)$.
(4) **InitialEncryptedForm**$(T_0)$.
(5) **For** $i = k$ **down to** 1
    (5a) **InitialSum**$(T_0, ((2 * (k - i + 1) - 1) - 1) * (k + 1) + 1)$.
    (5b) **BinaryParallelMultiplier**$(T_0, 2 * (k - i + 1) - 1, 2 * (k - i + 1) - 1, C, C)$.
    (5c) **AssignmentOperator**$(T_0, 2 * (k - i + 1) - 1)$.
    (5d) **BinaryParallelDivider**$(T_0, 2 * (k - i + 1) - 1)$.
    (5e) **TruncatedAssignmentOperator**$(T_0, 2 * (k - i + 1) - 1, 2 * (k - i + 1))$.
    (5f) *Separate*$(T_e, e_{i-1}^1, T_e^{\mathrm{ON}}, T_e^{\mathrm{OFF}})$.
    (5g) **If** (*Detect*$(T_e^{\mathrm{ON}})$ == *true*) **then**
    (5h)     **InitialSum**$(T_0, (2 * (k - i + 1) - 1) * (k + 1) + 1)$.
    (5i)     **BinaryParallelMultiplier**$(T_0, 2 * (k - i + 1), 2 * (k - i + 1), C, M)$.
    (5j)     **AssignmentOperator**$(T_0, 2 * (k - i + 1))$.
    (5k)     **BinaryParallelDivider**$(T_0, 2 * (k - i + 1))$.
    (5l)     **TruncatedAssignmentOperator**$(T_0, 2 * (k - i + 1), 2 * (k - i + 1) + 1)$.
         **Else**
    (5m)     **TruncatedAssignmentOperator**$(T_0, 2 * (k - i + 1) - 1,$
               $2 * (k - i + 1) + 1)$.
         **EndIf**
    (5n) $T_e = $ *Merge*$(T_e^{\mathrm{ON}}, T_e^{\mathrm{OFF}})$.
  **EndFor**
**EndAlgorithm**
```

Theorem 1 *From those steps in Algorithm 1, the correspondence of between plain-text and cipher-text in the RSA public-key cryptosystem can be constructed.*

Proof On the execution of Step (1), it calls MakeValue(T_n) to yield tube T_n that contains a DNA strand encoding the production of two large odd prime numbers, n. Then, on the execution of Step (2), it calls Init(T_0, T_n) to generate tube T_0 that includes DNA strands encoding $2k$ plain-texts. The execution of Step (3) calls MakeInitialValue(T_e) to produce tube T_e that contains a DNA strand encoding the public key, e. Next, the execution of Step (4) calls InitialEncryptedForm(T_0) to perform the execution of Step (2) in the procedure, Encryption(M, e, n) and to generate the initial encrypted form of the corresponding cipher-text for every plain-text in tube T_0.

Step (5) is a loop and is mainly used to perform the function of the only loop (Step (3)) in the procedure, Encryption(M, e, n). Next, on the first execution of Step (5a), since the value of the loop index, i, is k, the value of the second argument is one. Therefore, it calls InitialSum(T_0, 1) and perform to set the initial value of the sum (product) of the first multiplication. On the first execution of Step (5b), because the value of the loop index, i, is k, the second and the third arguments are one and the fourth argument and the fifth argument are used to represent the multiplicand and the multiplier of the first multiplication. Hence, it calls BinaryParallelMultiplier(T_0, 1, 1, C, C) to perform computation of C2 in Step (3a) in Encryption(M, e, n). Next, on the first execution of Step (5c), since the value of the loop index, i, is k, the value of the second argument is one. Thus, it calls AssignmentOperator(T_0, 1) to perform to set the initial value of the dividend in the first division. On the first execution of Step (5d), because the value of the loop index, i, is k, the value of the second argument is one. Therefore, it calls BinaryParallelDivider(T_0, 1) to finish the division (modular operation) of Step (3a) in Encryption(M, e, n). Next, on the first execution of Step (5e), since the value of the loop index, i, is k, the second argument and the third argument are one and two. Hence, it calls TruncatedAssignmentOperator(T_0, 1, 2) to update the encrypted form of a cipher-text. This implies that the execution of Step (3a) in Encryption(M, e, n) can be finished by means of Step (5a) through (5e).

On the first execution of Step (5f), it generates tube T_e^{ON} that includes all of the strands having e_{k-1^1} because the value of the loop index, i, is k and tube T_e^{OFF} that consists of all of the strands having e_{k-1^0}. Next, the first execution of Step (5g), it uses the detect operations to check whether there is any DNA sequence in T_e^{ON}. This means that the execution of Step (3b) in Encryption(M, e, n) can be finished by means of Step (5f) through (5g). If Step (5g) returns a true, this implies that the execution of Step (3c) in Encryption(M, e, n) will be executed. So, Step (5h) through Step (5l) are used to finish the execution of Step (3c) in Encryption(M, e, n). If Step (5g) returns a false, then Step (5m) is applied to set the next encrypted form of a cipher-text to the current encrypted form. Next, the first execution of Step (5n) applies the merge operation to pour tubes T_e^{ON} and T_e^{OFF} into T_e. This is to say that tube T_e reserves the strand encoding the public-key, e. Repeat execution of Step (5a) through Step (5n) until the last time of the loop is processed. Finally, tube T_0 contains the strands encoding the final encrypted form of every cipher-text. Therefore, the correspondence of between plain-text and cipher-text in the RSA public-key cryptosystem can be constructed from those steps in Algorithm 1. □

4 Algorithm modules

We now introduce, in detail, the various modules which are combined to form the overall algorithm for the encryption function in the RSA public-key cryptosystem.

4.1 A DNA strand for representing n in the RSA public-key cryptosystem

Assume that the length of n is k bits and n is represented as a k-bit binary number, $n_k \dots n_1$, where the value of each bit n_j is either 1 or 0 for $1 \le j \le k$. The bits n_k and n_1 represent the most significant bit and the least significant bit for n, respectively. For every bit n_j, two *distinct* 15 base value sequences are designed to respectively represent the value "0" for n_j and 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($T_n$)
  (1) For $j = 1$ to $k$
        (1a) Appendhead($T_n$, $n_j$).
      EndFor
EndProcedure
```

Lemma 1 *A DNA strand for representing n in the RSA public-key cryptosystem can be constructed from the algorithm*, MakeValue(T_n).

Proof Each time Step (1a) is used to append *distinct* 15 base value sequences, representing the value "1" or "0" for n_j, onto the head of every strand in tube T_n. After repeating execution of Step (1a), it produces tube T_n that consists of a DNA strand representing n. □

4.2 Solution space of DNA sequences for any plain-text in the RSA public-key cryptosystem

Suppose that the length of a plain-text M is k bits and is represented as a k-bit binary number, $m_k \dots m_1$, where the value of each bit m_j is either 1 or 0 for $1 \le j \le k$. The bits m_k and m_1 represent the most significant bit and the least significant bit for M, respectively. For every bit m_j, from [24, 25] two *distinct* 15 base value sequences are designed to respectively represent the value "0" for m_j and the value "1" for m_j. Also, 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. The following algorithm is used to construct solution space of DNA sequences for any plain-text of k bits in the RSA public-key cryptosystem.

```
Procedure Init($T_0$, $T_n$)
  (1) For $j = 1$ to $k$
        (1a) Amplify($T_0$, $T_1$, $T_2$).
        (1b) Appendhead($T_1$, $m_j^1$).
        (1c) $Appendhead(T_2, m_j^0)$.
        (1d) $T_0 = Merge(T_1, T_2)$.
      EndFor
   (2) For $j = k$ down to 1
        (2a) $Separate(T_n, n_j^1, T_n^{ON}, T_n^{OFF})$.
        (2b) $Separate(T_0, m_j^1, T_0^{ON}, T_0^{OFF})$.
        (2c) If ($Detect(T_n^{ON}) == true$) then
             (2d) $T_0^{=} = Merge(T_0^{=}, T_0^{ON})$ and $T_0^{<} = Merge(T_0^{<}, T_0^{OFF})$.
             Else
             (2e) $T_0^{>} = Merge(T_0^{>}, T_0^{ON})$ and $T_0^{=} = Merge(T_0^{=}, T_0^{OFF})$.
             EndIf
        (2f) $T_0 = Merge(T_0, T_0^{=})$ and $T_n = Merge(T_n^{ON}, T_n^{OFF})$.
        (2g) $Discard(T_0^{>})$.
      EndFor
   (3) $T_0 = Merge(T_0, T_0^{<})$.
EndProcedure
```

Lemma 2 *Solution space of DNA sequences for any plain-text of k bits in the RSA public-key cryptosystem can be constructed from the algorithm*, Init(T_0, T_n).

Proof Similar to Algorithm 1 and Lemma 1. □

4.3 Solution space of DNA sequences for public-key in the RSA public-key cryptosystem

Algorithm 1 uses MakeInitialValue(T_e), as a sub-module, to construct a DNA strand for encoding e, the public-key denoted in Sect. 3 in the RSA public-key cryptosystem. The following algorithm, MakeInitialValue(T_e), is proposed to construct a DNA strand for encoding e.

```
Procedure MakeInitialValue($T_e$)
   (1) For $i = 0$ to $k - 1$
          (1a) $Appendhead(T_e, e_i)$.
      EndFor
EndProcedure
```

Lemma 3 *Solution space of DNA sequences for the public-key, e, in the RSA public-key cryptosystem can be constructed from the algorithm*, MakeInitialValue(T_e).

Proof Similar to Algorithm 1 and Lemma 1. □

4.4 Solution space of DNA sequences for any cipher-text in the RSA public-key cryptosystem

Assume that the length of a cipher-text C in the RSA public-key cryptosystem is k bits. From the procedure Encryption(M, e, n), the encrypted form of a cipher-text

C is finally obtained after at most updating $(2 * k + 1)$ times of the value for the cipher-text C. Therefore, suppose that a cipher-text C is represented as a k-bit binary number, $c_{a,k} \dots c_{a,1}$, where the value of each bit $c_{a,j}$ is either 1 or 0 for $1 \leq a \leq (2 * k + 1)$ and $1 \leq j \leq k$. The bits, $c_{a,k}$ and $c_{a,1}$, represent the most significant bit and the least significant bit for C, respectively. The first k-bit binary number, $c_{1,k} \dots c_{1,1}$, is used to represent the initial encrypted form of C. The last k-bit binary number, $c_{(2*k+1),k} \dots c_{(2*k+1),1}$, is used to represent the final encrypted form of C. For other k-bit binary numbers, they are applied to represent the intermediate encrypted form of C. For every bit $c_{a,j}$, two *distinct* 15 base value sequences were designed to respectively represent the value "0" for $c_{a,j}$ and the value "1" for $c_{a,j}$. For convenience, we assume that $c^1_{a,j}$ denotes the value of $c_{a,j}$ to be 1 and $c^0_{a,j}$ defines the value of $c_{a,j}$ to be 0. The following algorithm is used to construct solution space of DNA sequences for the initial encrypted forms to any cipher-text in the RSA public-key cryptosystem.

```
**Procedure InitialEncryptedForm**($T_0$)
(1) *Appendhead*($T_0$, $c^1_{1,1}$).
(2) **For** $j = 2$ **to** $k$
        (2a) *Appendhead*($T_0$, $c^0_{1,j}$).
    **EndFor**
**EndProcedure**
```

Lemma 4 *Solution space of DNA sequences for the initial encrypted forms to any cipher-text in the RSA public-key cryptosystem can be constructed from the algorithm*, InitialEncryptedForm(T_0).

Proof Similar to Algorithm 1 and Lemma 1. □

4.5 Solution space of DNA sequences for the initial value of the sum in a multiplier

A k-bit multiplier is used to finish multiplication of two binary numbers of k bits. It is done by successive additions and shifting of k times. The product obtained from the k-bit multiplier can be up to $(2 * k)$ bits long. A k-bit multiplier is done by successive additions and shifting of k times. Hence, suppose that the length of an integer Y is $(2 * k)$ bits. The integer Y is used to represent the augend and the sum of successive additions in a k-bit multiplier. From the procedure Encryption(M, e, n), the encrypted form of a cipher-text C is finally obtained after at most finishing $(2 * k)$ multiplication instructions. That is to say, at most $(2 * k^2)$ successive additions and shifting are completed. Therefore, suppose that Y is represented as a $(2 * k)$-bit binary number, $y_{f,(2*k)} \dots y_{f,1}$, where the value of each bit $y_{f,g}$ is either 1 or 0 for $1 \leq f \leq (2 * k^2 + 2 * k)$ and $1 \leq g \leq (2 * k)$. The bits, $y_{f,(2*k)}$ and $y_{f,1}$, represent the most significant bit and the least significant bit for Y, respectively. Two binary numbers $y_{f,(2*k)} \dots y_{f,1}$ and $y_{f+1,(2*k)} \dots y_{f+1,1}$ represent the augend and the sum of the successive fth addition and shift, respectively. This implies that the binary number $y_{f+1,(2*k)} \dots y_{f+1,1}$ is the augend of the successive $(f + 1)$th addition and shift. For every bit $y_{f,g}$, two *distinct* 15 base value sequences were designed to respectively represent the value "0" for $y_{f,g}$ and the value "1" for $y_{f,g}$. For convenience, we assume that $y^1_{f,g}$ denotes the value of $y_{f,g}$ to be 1 and $y^0_{f,g}$ defines the value of $y_{f,g}$ to be 0. The following algorithm is used to construct solution space of DNA sequences for the initial value of the sum in a k-bit multiplier.

```
Procedure InitialSum($T_0$, $f$)
(1) For $g = 1$ to $2 * k$
    (1a) Appendhead($T_0$, $y^0_{f,g}$).
    EndFor
EndProcedure
```

Lemma 5 *Solution space of DNA sequences for the initial value of the sum in a k-bit multiplier can be constructed from the algorithm*, InitialSum(T_0, f).

Proof Similar to Algorithm 1 and Lemma 1. □

4.6 The construction of a binary parallel multiplier

A binary parallel multiplier is a function that finishes the arithmetic multiplication for two binary numbers of k bits. The product obtained from the multiplication of two k-bit binary numbers can be up to $(2 * k)$ bits long. The following algorithm is proposed to finish the function of a binary parallel multiplier. The two parameters α and β in the algorithm are used to represent the multiplicand and the multiplier of a binary parallel multiplier. Assume that β^1_f is applied to represent the value of "1" for the fth bit of the multiplier (β) to the fth addition and shift.

```
Procedure BinaryParallelMultiplier($T_0, a, u, \alpha, \beta$)
    (1) For $f = 1$ to $k$
        (1a0) Appendhead($T_0$, $z^0_{(u-1)*(k+1)+f,0}$).
        (1a) Separate($T_0$, $\beta^1_f$, $T_3$, $T_4$).
        (1a1) If (Detect($T_3$) == true) then
        (1b) For $g = 1$ to $f - 1$
            (1b1) Separate($T_3$, $y^1_{f+(u-1)*(k+1),g}$, $T_1$, $T_2$).
            (1b10) If (Detect($T_1$) == true) then
            (1b2) Appendhead($T_1$, $y^1_{f+(u-1)*(k+1)+1,g}$) and
                      Appendhead($T_1$, $z^0_{f+(u-1)*(k+1),g}$).
            EndIf
            (1b20) If (Detect($T_2$) == true) then
            (1b3) Appendhead($T_2$, $y^0_{f+(u-1)*(k+1)+1,g}$) and
                      Appendhead($T_2$, $z^0_{f+(u-1)*(k+1),g}$).
            EndIf
            (1b4) $T_3$ = merge($T_1$, $T_2$).
        EndFor
    (1c) BinaryParallelAdder($T_3$, $f + (u-1) * (k+1) + 1, g, a$).
    (1c1) For $g = (k+f)$ to $(k+f)$
        (1c11) Separate($T_3$, $z^1_{f+(u-1)*(k+1),g-1}$, $T_1$, $T_2$).
```

```
            (1c12) **If** ($Detect(T_1) == true$) **then**
            (1c13) $Appendhead(T_1, y^1_{f+(u-1)*(k+1)+1,g})$ and
                      $Appendhead(T_1, z^0_{f+(u-1)*(k+1),g})$.
            **EndIf**
            (1c14) **If** ($Detect(T_2) == true$) **then**
            (1c15) $Appendhead(T_2, y^0_{f+(u-1)*(k+1)+1,g})$ and
                      $Appendhead(T_2, z^0_{f+(u-1)*(k+1),g})$.
            **EndIf**
            (1c16) $T_3 = Merge(T_1, T_2)$.
            **EndFor**
        (1d) **For** $g = (k + f + 1)$ **to** $2 * k$
              (1d1) $Appendhead(T_3, y^0_{f+(u-1)*(k+1)+1,g})$ and
                       $Appendhead(T_3, z^0_{f+(u-1)*(k+1),g})$.
            **EndFor**
        **EndIf**
        (1d2) **If** ($Detect(T_4) == true$) **then**
        (1e) **For** $g = 1$ **to** $2 * k$
              (1e1) $T_1 = Separate(T_4, y^1_{f+(u-1)*(k+1),g}, T_1, T_2)$.
              (1e10) **If** ($Detect(T_1) == true$) **then**
              (1e2) $Appendhead(T_1, y^1_{f+(u-1)*(k+1)+1,g})$ and
                       $Appendhead(T_1, z^0_{f+(u-1)*(k+1),g})$.
              **EndIf**
              (1e20) **If** ($Detect(T_2) == true$) **then**
              (1e3) $Appendhead(T_2, y^0_{f+(u-1)*(k+1)+1,g})$ and
                       $Appendhead(T_2, z^0_{f+(u-1)*(k+1),g})$.
              **EndIf**
              (1e4) $T_4 = Merge(T_1, T_2)$.
            **EndFor**
        **EndIf**
        (1f) $T_0 = Merge(T_3, T_4)$.
**EndFor**
**EndProcedure**
```

Lemma 6 *The algorithm*, BinaryParallelMultiplier(T_0, a, u, α, β), *can be applied to finish the function of a binary parallel multiplier.*

Proof The multiplication of the multiplicand and the multiplier of k bits is finished through k times of successive additions and left shift. Step (1) is the main loop and is applied to finish the function of a binary parallel multiplier. This means that the main loop is used to finish successive additions and left shift of k times for the k-bit multiplicand and the k-bit multiplier. With each addition and each left shift, the least significant position of the multiplicand and the multiplier of k bits is added; the input carry must be 0. So, each execution for Step (1a0) uses the *appendhead* operation to append 15-based DNA sequences for representing $z^0_{(u-1)*(k+1)+f,0}$ onto the head of every strand in T_0. Next, on each execution of Step (1a), it employs the *extract* operation to form tube T_3 including the strands having β_f^1, and tube T_4 consisting of the strands having β_f^0. Each execution of Step (1a1) is used to check whether contains any DNA strand for tube T_3 or not. If a *true* is returned, then the enclosed steps will be run.

Next, for tube T_3, Step (1b) is a loop and it is applied to finish additions and left shift of the front $(f - 1)$ bits for the fth addition and left shift. The number of shifted bit to the fth shift is $(f - 1)$ bits. This implies that the front $(f - 1)$ bits of the addend (multiplicand) for the fth addition are all zero. Therefore, it is inferred that the front $(f - 1)$ bits of the sum to the fth addition are equal to the front $(f - 1)$ bits of the augend to the fth addition. Hence, on each execution of Step (1b1), it employs the *extract* operation to form tube T_1 including the strands having $y_{f+(u-1)*(k+1),g} = 1$, and tube T_2 consisting of the strands having $y_{f+(u-1)*(k+1),g} = 0$. Each execution of Step (1b10) is applied to examine whether contains any DNA strand for tube T_1 or not. If a *true* is returned, then Step (1b2) will be run. Next, each execution of Step (1b2) uses the *appendhead* operations to append $y^1_{f+(u-1)*(k+1)+1,g}$ and $z^0_{f+(u-1)*(k+1),g}$ onto the head of every strand in T_1. Each execution of Step (1b20) is used to test whether contains any DNA strand for tube T_2 or not. If a *true* is returned, then Step (1b3) will be run. Each execution of Step (1b3) applies the *appendhead* operations to append $y^0_{f+(u-1)*(k+1)+1,g}$ and $z^0_{f+(u-1)*(k+1),g}$ onto the head of every strand in T_2. Then each execution of Step (1b4) applies the *merge* operation to pour tubes T_1 and T_2 into T_3. Tube T_3 contains the strands finishing addition and left shift of a bit. Repeat execution of Steps (1b1) through (1b4) until the front $(f - 1)$ bits are processed. Tube T_3 contains the strands finishing addition of the front $(f - 1)$ bits for the fth addition and left shift.

Next, each execution of Step (1c) calls the algorithm, BinaryParallelAdder(T_3, $f + (u - 1) * (k + 1) + 1, g, a$), to finish addition and left shift of k bits for the fth bit through the $(k + f - 1)$th bit. Because after BinaryParallelAdder(T_3, $f + (u - 1) * (k+1) + 1, g, a$) is performed, it perhaps generates the carry "1" for the $(k + f - 1)$th bit. Therefore, on each execution of Step (1c11), it employs the *extract* operation to form tube T_1 including the strands having $z_{f+(u-1)*(k+1),g-1} = 1$, and tube T_2 consisting of the strands having $z_{f+(u-1)*(k+1),g-1} = 0$. Each execution of Step (1c12) is applied to test whether contains any DNA strand for tube T_1 or not. If a *true* is returned, then Step (1c13) will be run. Next, each execution of Step (1c13) uses the *appendhead* operations to append $y^1_{f+(u-1)*(k+1)+1,g}$ and $z^0_{f+(u-1)*(k+1),g}$ onto the head of every strand in T_1. Each execution of Step (1c14) is used to examine whether contains any DNA strand for tube T_2 or not. If a *true* is returned, then Step (1c15) will be run. Each execution of Step (1c15) applies the *appendhead* operations to append $y^0_{f+(u-1)*(k+1)+1,g}$ and $z^0_{f+(u-1)*(k+1),g}$ onto the head of every strand in T_2. Then each execution of Step (1c16) applies the *merge* operation to pour tubes T_1 and T_2 into T_3. Tube T_3 includes the strands performing addition of a bit.

Step (1d) is a loop and is used to finish addition and left shift of $(k - f + 1)$ bits, because the last $(k - f + 1)$ bits of the addend (multiplicand) for the fth addition and left shift are all zero. Therefore, it is inferred that the last $(k - f + 1)$ bits of the sum to the fth addition and left shift are equal to the last $(k - f + 1)$ bits of the augend to the fth addition and left shift. Execution of Step (1d1) applies the *appendhead* operations to append $y^0_{f+(u-1)*(k+1)+1,g}$ and $z^0_{f+(u-1)*(k+1),g}$ onto the head of every strand in T_3. Repeat execution of Step (1d1) until the last $(k - f + 1)$ bits are processed. Tube T_3 contains the strands finishing the fth addition and left shift.

Since T_4 consists of all of the strands that have $\beta_f = 0$, the $(2 * k)$ bits of the addend (multiplicand) for the fth addition and left shift are all zero. This implies that the $(2 * k)$ bits of the sum to the fth addition and left shift are equal to the $(2 * k)$ bits of the augend to the fth addition and left shift. Each execution of Step (1d2) is used to check whether contains any DNA strand for tube T_4 or not. If a *true* is returned, then the enclosed steps will be run. Step (1e) is a loop and is employed to finish the fth addition and left shift for tube T_4. Each execution of Step (1e1) employs the *extract* operation to form two test tubes: T_1 and T_2. The first tube T_1 includes all of the strands that have $y_{f+(u-1)*(k+1),g} = 1$, and the second tube T_2 consists of all of the strands that have $y_{f+(u-1)*(k+1),g} = 0$. Each execution of Step (1e10) is applied to examine whether contains any DNA strand for tube T_1 or not. If a *true* is returned, then Step (1e2) will be run. Next, each execution of Step (1e2) uses the *appendhead* operations to append $y^1_{f+(u-1)*(k+1)+1,g}$ and $z^0_{f+(u-1)*(k+1),g}$ onto the head of every strand in T_1. Each execution of Step (1e20) is used to test whether contains any DNA strand for tube T_2 or not. If a *true* is returned, then Step (1e3) will be run. On each execution of Step (1e3), it applies the *appendhead* operations to append $y^0_{f+(u-1)*(k+1)+1,g}$ and $z^0_{f+(u-1)*(k+1),g}$ onto the head of every strand in T_2. Then, each execution of Step (1e4) applies the *merge* operation to pour tubes T_1 and T_2 into T_4. Tube T_4 contains the strands finishing addition and left shift of a bit. Repeat execution of Steps (1e1) through (1e4) until the $(2 * k)$ bits are processed. Tube T_4 contains the strands finishing addition and left shift of the $(2 * k)$ bits for the fth addition and left shift.

Each execution of Step (1f) applies the *merge* operation to pour tubes T_3 and T_4 into T_0. Tube T_0 contains the strands finishing the fth addition and left shift of $(2*k)$ bits. Repeat execution of Steps (1a) through (1f) until successive additions and left shift of k times are processed. Tube T_0 contains the strands finishing multiplication of k bits. □

4.7 The construction of a binary parallel adder

The BinaryParallelMultiplier(T_0, a, u, α, β) module uses, as a submodule, a parallel adder. We first describe the construction of a parallel adder for a single bit, and next demonstrate how this perhaps is applied as a building block for a parallel adder by means of using bit-strings of arbitrary length. A one-bit adder is a function that performs the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input bits represent augend and addend bits to be added, respectively. The third input represents the carry from the previous lower significant position. The first output gives the value of the sum for augend and addend bits to be added. The second output gives the value of the carry to augend and addend bits to be added. The truth table of the one-bit adder is shown in Table 1.

Suppose that two one-bit binary numbers denoted in Sect. 4.5, $y_{f,g}$ and $y_{f+1,g}$, represent the first input of a one-bit adder for $1 \le f \le (2 * k^2 + 2 * k)$ and $1 \le g \le 2 * k$, and the first output of a one-bit adder, respectively, a one-bit binary number

Table 1 The truth table of a one-bit adder

Augend bit	Addend bit	Previous carry bit	Sum bit	Carry bit
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

denoted in Sect. 4.4, $c_{a,j}$, represents the second input of a one-bit adder for $1 \leq a \leq (2 * k + 1)$ and $1 \leq j \leq k$, and two one-bit binary numbers, $z_{f,g}$ and $z_{f,g-1}$, represent the second output and the third input of a one-bit adder, respectively. Two *distinct* DNA sequences are also designed to represent the value "0" or "1" of every corresponding bit. For the sake of convenience in our presentation, assume that $z^1_{f,g}$ contains the value of $z_{f,g}$ to be 1 and $z^0_{f,g}$ contains the value of $z_{f,g}$ to be 0. Also, suppose that $y^1_{f+1,g}$ denotes the value of $y_{f+1,g}$ to be 1 and $y^0_{f+1,g}$ defines the value of $y_{f+1,g}$ to be 0. Similarly, assume that $z^1_{f,g-1}$ contains the value of $z_{f,g-1}$ to be 1 and $z^0_{f,g-1}$ contains the value of $z_{f,g-1}$ to be 0. The following algorithm is proposed to finish the function of a parallel one-bit adder.

```
Procedure ParallelOneBitAdder($T_3$, $f$, $g$, $a$, $j$)
    (1) Separate($T_3$, $y^1_{f,g}$, $T^{ON}_1$, $T^{OFF}_2$).
    (2) Separate($T^{ON}_1$, $c^1_{a,j}$, $T^{ON}_3$, $T^{OFF}_4$).
    (3) Separate($T^{OFF}_2$, $c^1_{a,j}$, $T_5$, $T_6$).
    (4) Separate($T^{ON}_3$, $z^1_{f,g-1}$, $T_7$, $T_8$).
    (5) Separate($T^{OFF}_4$, $z^1_{f,g-1}$, $T_9$, $T_{10}$).
    (6) Separate($T_5$, $z^1_{f,g-1}$, $T_{11}$, $T_{12}$).
    (7) Separate($T_6$, $z^1_{f,g-1}$, $T_{13}$, $T_{14}$).
    (8a) If (Detect($T_7$) = true) then
            (8) Appendhead($T_7$, $y^1_{f+1,g}$) and Appendhead($T_7$, $z^1_{f,g}$).
    EndIf
    (9a) If (Detect($T_8$) = true) then
            (9) Appendhead($T_8$, $y^0_{f+1,g}$) and Appendhead($T_8$, $z^1_{f,g}$).
    EndIf
    (10a) If (Detect($T_9$) = true) then
            (10) Appendhead($T_9$, $y^0_{f+1,g}$) and Appendhead($T_9$, $z^1_{f,g}$).
    EndIf
    (11a) If (Detect($T_{10}$) = true) then
            (11) Appendhead($T_{10}$, $y^1_{f+1,g}$) and Appendhead($T_{10}$, $z^0_{f,g}$).
    EndIf
    (12a) If (Detect($T_{11}$) = true) then
            (12) Appendhead($T_{11}$, $y^0_{f+1,g}$) and Appendhead($T_{11}$, $z^1_{f,g}$).
    EndIf
    (13a) If (Detect($T_{12}$) = true) then
        (13) Appendhead($T_{12}$, $y^1_{f+1,g}$) and Appendhead($T_{12}$, $z^0_{f,g}$).
    EndIf
    (14a) If (Detect($T_{13}$) = true) then
        (14) Appendhead($T_{13}$, $y^1_{f+1,g}$) and Appendhead($T_{13}$, $z^0_{f,g}$).
    EndIf
    (15a) If (Detect($T_{14}$) = true) then
        (15) Appendhead($T_{14}$, $y^0_{f+1,g}$) and Appendhead($T_{14}$, $z^0_{f,g}$).
    EndIf
    (16) $T_3$ = Merge($T_7$, $T_8$, $T_9$, $T_{10}$, $T_{11}$, $T_{12}$, $T_{13}$, $T_{14}$).
EndProcedure
```

Lemma 7 *The algorithm*, ParallelOneBitAdder(T_3, f, g, a, j), *can be applied to finish the function of a parallel one-bit adder.*

Proof Steps (1) through (7) employ the *extract* operations to form some different test tubes including different strands (T_1^{ON} to T_{14}). That is, T_1^{ON} includes all of the strands that have $y_{f,g} = 1$, T_2^{OFF} includes all of the strands that have $y_{f,g} = 0$, T_3^{ON} includes those that have $y_{f,g} = 1$ and $c_{a,j} = 1$, T_4^{OFF} includes those that have $y_{f,g} = 1$ and $c_{a,j} = 0$, T_5 includes those that have $y_{f,g} = 0$ and $c_{a,j} = 1$, T_6 includes those that have $y_{f,g} = 0$ and $c_{a,j} = 0$, T_7 includes those that have $y_{f,g} = 1, c_{a,j} = 1$ and $z_{f,g-1} = 1$, T_8 includes those that have $y_{f,g} = 1, c_{a,j} = 1$ and $z_{f,g-1} = 0$, T_9 includes those that have $y_{f,g} = 1, c_{a,j} = 0$ and $z_{f,g-1} = 1$, T_{10} consists of those that have $y_{f,g} = 1, c_{a,j} = 0$ and $z_{f,g-1} = 0$, T_{11} includes those that have $y_{f,g} = 0, c_{a,j} = 1$ and $z_{f,g-1} = 1$, T_{12} includes those that have $y_{f,g} = 0, c_{a,j} = 1$ and $z_{f,g-1} = 0$, T_{13} includes those that have $y_{f,g} = 0, c_{a,j} = 0$ and $z_{f,g-1} = 1$, and finally, T_{14} consists of those that have $y_{f,g} = 0, c_{a,j} = 0$ and $z_{f,g-1} = 0$. Having finished Steps (1) through (7), this implies that eight different inputs of a one-bit adder as shown in Table 1 were poured into tubes T_7 through T_{14}, respectively.

Steps (8a), (9a), (10a), (11a), (12a), (13a), (14a), and (15a) are, respectively, used to check whether contains any DNA strand for tubes $T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13},$ and T_{14} or not. If any a *true* is returned for those steps, then the corresponding *appendhead* operations will be run. Next, Steps (8) through (15) use the *appendhead* operations to append $y^1_{f+1,g}$ or $y^0_{f+1,g}$, and $z^1_{f,g}$ or $z^0_{f,g}$ onto the head of every strand in the corresponding test tubes. After finishing Steps (8) through (15), we can say that eight different outputs of a one-bit adder in Table 1 are appended into tubes T_7 through T_{14}. Finally, the execution of Step (16) applies the *merge* operation to pour tubes T_7 through T_{14} into tube T_3. Tube T_3 contains the strands finishing the addition of a bit. □

The one-bit adder just introduced figures out the sum and the carry of two input bits and a previous carry. Two k-bit binary numbers each can be added by means of this one-bit adder. A binary parallel adder is to finish the arithmetic sum for two k-bit binary numbers. The following algorithm is proposed to finish the function of a binary parallel adder.

```
Procedure BinaryParallelAdder($T_3$, $f$, $g$, $a$)
    (1) For $j = 1$ to $k$
        (1a) ParallelOneBitAdder($T_3$, $f - 1$, $g + j - 1$, $a$, $j$).
    EndFor
EndProcedure
```

Lemma 8 *The algorithm*, BinaryParallelAdder(T_3, f, g, a), *can be applied to finish the function of a binary parallel adder.*

Proof Similar to Algorithm 1 and Lemma 1. □

4.8 The construction of an assignment operator

An assignment operator is an instruction of two operands of $(2 * k)$ bits that the value of the first operand is set to the value of the second operand. Assume that the second operand is the product obtained from a k-bit multiplier; also assume that the length of an integer P, the first operand of an assignment operator, is $(2 * k)$ bits. Additionally, P is also applied to represent the minuend (dividend) and the difference of successive compare, shift and subtract operations in a divider. From the procedure, Encryption(M, e, n), the encrypted form of a cipher-text C is finally obtained after at most finishing $(2 * k)$ division instructions. This is to say that $((2 * k) * (k + 1))$ successive compare, shift, and subtract operations are at most finished. Therefore, suppose that P is represented as a $(2 * k)$-bit binary number, $p_{o,(2*k)} \dots p_{o,1}$, where the value of each bit $p_{o,q}$ is either 1 or 0 for $1 \le o \le (2 * k^2 + 4 * k)$ and $1 \le q \le (2 * k)$. The bits, $p_{o,(2*k)}$ and $p_{o,1}$, represent the most significant bit and the least significant bit for P, respectively. Two binary numbers $p_{o,(2*k)} \dots p_{o,1}$ and $p_{o+1,(2*k)} \dots p_{o+1,1}$ are applied to represent the minuend and the difference of the successive oth successive compare, shift, and subtract operations. That is, the binary number $p_{o+1,(2*k)} \dots p_{o+1,1}$ is the minuend of the successive $(o + 1)$th successive compare, shift, and subtract operations.

For every bit $p_{o,q}$, two *distinct* 15 base value sequences are designed to respectively represent the value "0" for $p_{o,q}$ and the value "1" for $p_{o,q}$. For the sake of convenience in our presentation, assume that $p^1_{o,q}$ denotes the value of $p_{o,q}$ to be 1 and $p^0_{o,q}$ defines the value of $p_{o,q}$ to be 0. The following algorithm is used to construct an assignment operator. This implies that the assignment operator can be used to set the initial value of the dividend in a divider. The second parameter, u, in the algorithm is applied to represent the uth arithmetic multiplication.

```
Procedure AssignmentOperator($T_0$, $u$)
    (1) For $q = 1$ to $2 * k$
        (1a) $Separate(T_0, y^1_{u*(k+1),q}, T_1, T_2)$.
        (1b) $Appendhead(T_1, p^1_{(u-1)*(k+2)+1,q})$.
        (1c) $Appendhead(T_2, p^0_{(u-1)*(k+2)+1,q})$.
        (1d) $T_0 = Merge(T_1, T_2)$.
    EndFor
EndProcedure
```

Lemma 9 *The algorithm*, AssignmentOperator(T_0, u), *can be applied to finish the function of an assignment operator.*

Proof Similar to Algorithm 1 and Lemma 1. □

4.9 The construction of a binary parallel divider

A binary parallel divider is a function that finishes the arithmetic division for a dividend of $(2 * k)$ bits and a divisor of k bits. The quotient obtained from the division to a dividend of $(2 * k)$ bits and a divisor of k bits can be up to $(k + 1)$ bits long. The remainder obtained from the division to a dividend of $(2 * k)$ bits and a divisor of k bits can be at most up to k bits long. Because the encryption function denoted in Sect. 3 needs to finish modular operations, the quotient can be neglected. The following algorithm is proposed to finish the function of a binary parallel divider.

```
Procedure BinaryParallelDivider($T_0$, $u$)
  (1) For $o = 1$ to $k + 1$
      (1a0) Appendhead($T_0$, $b^0_{(u-1)*(k+2)+o,0}$).
      (1a) ParallelComparator($T_0$, $T_n$, $T_0^>$, $T_0^=$, $T_0^<$, $u$, $o$).
      (1b) $T_0^{>=} = Merge(T_0^>, T_0^=)$.
      (1b0) If ($Detect(T_0^{>=}) == true$) then
      (2) For $q = 1$ to $(2 * k) - (o - 1) - (k - 1) - 1$
         (2a) Separate($T_0^{>=}$, $p^1_{o+(u-1)*(k+2),q}$, $T_1$, $T_2$).
         (2a0) If ($Detect(T_1) == true$) then
               (2b) Appendhead($T_1$, $p^1_{o+(u-1)*(k+2)+1,q}$) and
                    Appendhead($T_1$, $b^0_{o+(u-1)*(k+2),q}$).
         EndIf
         (2b0) If ($Detect(T_2) == true$) then
               (2c) Appendhead($T_2$, $p^0_{o+(u-1)*(k+2)+1,q}$) and
                    Appendhead($T_2$, $b^0_{o+(u-1)*(k+2),q}$).
         EndIf
         (2d) $T_0^{>=} = Merge(T_1, T_2)$.
      EndFor
      (3) BinaryParallelSubtractor($T_0^{>=}$, $o$, $q$, $u$).
      EndIf
      (4) If ($Detect(T_0^<) == true$) then
      (5) For $q = 1$ to $2 * k$
         (5a) Separate($T_0^<$, $p^1_{o+(u-1)*(k+2),q}$, $T_1$, $T_2$).
         (5a0) If ($Detect(T_1) == true$) then
               (5b) Appendhead($T_1$, $p^1_{o+(u-1)*(k+2)+1,q}$) and
                    Appendhead($T_1$, $b^0_{o+(u-1)*(k+2),q}$).
         EndIf
         (5b0) If ($Detect(T_2) == true$) then
               (5c) Appendhead($T_2$, $p^0_{o+(u-1)*(k+2)+1,q}$) and
                    Appendhead($T_2$, $b^0_{o+(u-1)*(k+2),q}$).
```

```
            EndIf
            (5d) $T_0^{<} = Merge(T_1, T_2)$.
        EndFor
        EndIf
        (6) $T_0 = Merge(T_0^{>=}, T_0^{<})$.
    EndFor
EndProcedure
```

Lemma 10 *The algorithm,* BinaryParallelDivider(T_0, u)*, can be applied to finish the function of a binary parallel divider.*

Proof The division to a dividend of $(2 * k)$ bits and a divisor of k bits is finished through successive compare, shift and subtract operations of $(k + 1)$ times. Step (1) is the main loop and is applied to finish the function of a binary parallel divider. When each successive compare, shift and subtract operations occur, the least significant position for a dividend of $(2 * k)$ bits and a divisor of k bits is subtracted, the input borrow bit must be 0. So, on each execution for Step (1a0) uses the *append-head* operation to append 15-based DNA sequences for representing $b^0_{(u-1)*(k+2)+1,0}$ onto the head of every strand in T_0. Next, on each execution of Step (1a), it calls ParallelComparator(T_0, T_n, $T_0^{>}$, $T_0^{=}$, $T_0^{<}$, u, o) to compare the divisor with the corresponding k bits in the dividend. After it is finished, tube $T_0^{>}$ includes the strands with the comparative result of greater than (">"), tube $T_0^{=}$ includes the strands with the comparative result of equal ("="), and tube $T_0^{<}$ consists of the strands with the comparative result of less than ("<"). Next, each execution of Step (1b) employs the *merge* operation to pour tubes $T_0^{>}$ and $T_0^{=}$ into $T_0^{>=}$. From each execution of Step (1b0), if a *true* is returned, then Step (2) through Step (3) will be executed.

Step 2 is a loop and is used mainly to reserve the least significant $((2 * k) - (o - 1) - (k - 1) - 1)$ bits of a dividend. This implies that the least significant $((2 * k) - (o - 1) - (k - 1) - 1)$ bits of the minuend (dividend) for the oth compare, shift and subtract operations are reserved and they are equal to the least significant $((2 * k) - (o - 1) - (k - 1) - 1)$ bits of the difference for the same operations. Therefore, on each execution of Step (2a), it uses the *extract* operation to form tube T_1 including the strands that have $p_{o+(u-1)*(k+2),q} = 1$, and tube T_2 consisting of the strands that have $p_{o+(u-1)*(k+2),q} = 0$. Next, Steps (2a0) and (2b0) are, respectively, used to check whether contains any DNA strand for tubes T_1 and T_2 or not. If any a *true* is returned for those steps, then the corresponding *appendhead* operations will be run. Each execution for Step (2b) and Step (2c) uses the *append-head* operations to append $p^1_{o+(u-1)*(k+2)+1,q}$, $b^0_{o+(u-1)*(k+2),q}$, $p^0_{o+(u-1)*(k+2)+1,q}$ and $b^0_{o+(u-1)*(k+2),q}$ onto the head of every strand in T_1 and the head of every strand in T_2. Then each execution to Step (2d) employs the *merge* operation to pour tubes T_1 and T_2 into $T_0^{>=}$. Tube $T_0^{>=}$ contains the strands finishing compare, shift and subtract operations of a bit. Repeat execution of Steps (2a) through (2d) until the least significant $((2 * k) - (o - 1) - (k - 1) - 1)$ bits of a minuend (dividend) are processed. Tube $T_0^{>=}$ contains the strands finishing successive compare, shift and subtract operations of the least significant $((2 * k) - (o - 1) - (k - 1) - 1)$ bits of a minuend (dividend). Next each execution of Step (3) calls the algorithm, BinaryParallelSubtractor($T_0^{>=}$, o, q, u), to accomplish subtraction of k bits.

Since $T_0^<$ consists of all of the strands with the comparative result of less than ("$<$"). This implies that the $(2 * k)$ bits of the difference to the oth compare, shift, and subtract operations are equal to the $(2 * k)$ bits of the minuend to the same operations. If a *true* is returned from each execution of Step (4), then Step (5) through Step (5d) will be executed. Step (5) is a loop and is employed to finish the oth compare, shift and subtract operations for tube $T_0^<$. On each execution of Step (5a), it employs the *extract* operation to form tube T_1 including the strands that have $p_{o+(u-1)*(k+2),q} = 1$, and tube T_2 consisting of the strands that have $p_{o+(u-1)*(k+2),q} = 0$.

Next, Steps (5a0) and (5b0) are, respectively, used to check whether contains any DNA strand for tubes T_1 and T_2 or not. If any a *true* is returned for those steps, then the corresponding *appendhead* operations will be run. Each execution for Step (5b) and Step (5c) uses the *appendhead* operations to append $p^1_{o+(u-1)*(k+2)+1,q}$, $b^0_{o+(u-1)*(k+2),q}$, $p^0_{o+(u-1)*(k+2)+1,q}$ and $b^0_{o+(u-1)*(k+2),q}$ onto the head of every strand in T_1 and the head of every strand in T_2. Then, each execution of Step (5d) applies the *merge* operation to pour tubes T_1 and T_2 into $T_0^<$. Tube $T_0^<$ contains the strands finishing compare, shift and subtract operations of a bit. Repeat execution of Steps (5a) through (5d) until the $(2 * k)$ bits are processed. Tube $T_0^<$ contains the strands finishing the oth compare, shift and subtract operations of the $(2 * k)$ bits to the comparative result of less than ("$<$").

Next, each execution of Step (6) applies the *merge* operation to pour tubes $T_0^{>=}$ and $T_0^<$ into T_0. Tube T_0 contains the strands finishing the oth compare, shift, and subtract operations of the $(2 * k)$ bits. Repeat execution of the above steps until successive compare, shift, and subtract operations of $(k + 1)$ times are processed. Tube T_0 contains the strands finishing a division for a dividend of $(2 * k)$ bits and a divisor of k bits. □

4.10 The construction of a parallel comparator

The BinaryParallelDivider(T_0, u) module uses, as a submodule, a parallel comparator. We now describe its construction in detail. A one-bit parallel comparator is a Boolean function that performs compared operation of the two input bits. From compared results in a one-bit parallel comparator, DNA strands encoding those pairs $(p_{o,q}, n)$ with compared results "$>$", DNA strands encoding those pairs $(p_{o,q}, n)$ with compared results "$=$" and DNA strands encoding those pairs $(p_{o,q}, n)$ with compared results "$<$" are, respectively, put into three different tubes.

Therefore, the submodule, OneBitComparator(T_0, T_n, $T_0^>$, $T_0^=$, $T_0^<$, u, o, j) is presented to compute the function of a one-bit parallel comparator. The first parameter and the second parameter, T_0 and T_n, respectively, contain those DNA strands that respectively encode $p_{o,q}$ and n. The third parameter, $T_0^>$, includes those DNA strands with the comparative result of greater than ("$>$") between $p_{o,q}$ and n. The fourth parameter, $T_0^=$, contains those DNA strands with the comparative result of equal ("$=$") between $p_{o,q}$ and n. The fifth parameter, $T_0^<$, consists of those DNA strands with the comparative result of less than ("$<$") between $p_{o,q}$ and n. The sixth parameter, u, is applied to represent the uth division operation. The seventh parameter, o, is used to represent the oth compared operation in parallel comparator of a k-bits. The eighth parameter, j, is employed to represent the jth bit of n to be compared.

```
Procedure OneBitComparator($T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, u, o, j$)
  (1) Separate($T_0, p^1_{o+(u-1)*(k+2),(2*k)-(o-1)-(j-1)}, T_0^{\mathrm{ON}}, T_0^{\mathrm{OFF}}$).
  (2) Separate($T_n, n^1_{k-(j-1)}, T_n^{\mathrm{ON}}, T_n^{\mathrm{OFF}}$).
  (3) If (Detect($T_n^{\mathrm{ON}}$) == true) then
      (3a) $T_0^{=} = Merge(T_0^{=}, T_0^{\mathrm{ON}})$ and $T_0^{<} = Merge(T_0^{<}, T_0^{\mathrm{OFF}})$.
    Else
      (3b) $T_0^{>} = Merge(T_0^{>}, T_0^{\mathrm{ON}})$ and $T_0^{=} = Merge(T_0^{=}, T_0^{\mathrm{OFF}})$.
    EndIf
  (4) $T_n = Merge(T_n^{\mathrm{ON}}, T_n^{\mathrm{OFF}})$.
EndProcedure
```

Lemma 11 *The algorithm*, OneBitComparator(T_0, T_n, $T_0^{>}$, $T_0^{=}$, $T_0^{<}$, u, o, j), *can be applied to finish the function of a one-bit parallel comparator.*

Proof Similar to Algorithm 1 and Lemma 1. □

The one-bit comparator just introduced compares the size of two input bits. Two k-bit binary numbers can be compared by means of this one-bit comparator. A binary parallel comparator is to perform the comparison for two k-bit binary numbers. The module, ParallelComparator(T_0, T_n, $T_0^{>}$, $T_0^{=}$, $T_0^{<}$, u, o) also is proposed to perform the function of a k-bit parallel comparator.

```
Procedure ParallelComparator($T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, u, o$)
  (1) For $j = 1$ to $o - 1$
      (1a) Separate($T_0, p^1_{o+(u-1)*(k+2),(2*k)-(j-1)}, T_0^{>}, T_0$).
      (1b) If (Detect($T_0$) ==false) then
          (1c) Terminate the execution of the loop.
          EndIf
    EndFor
  (2) If (Detect($T_0$) == true) then
      (3) For $j = 1$ to $k$
          (3a) OneBitComparator($T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, u, o, j$).
          (3b) If (Detect($T_0^{=}$) == false) then
          (3c) Terminate the execution of the loop.
              EndIf
        EndFor
    EndIf
EndProcedure
```

Proof Similar to Algorithm 1 and Lemma 1. □

4.11 The construction of a parallel subtractor

The BinaryParallelDivider(T_0, d) module uses, as a submodule, a parallel subtractor. We first describe the construction of a parallel subtractor for a single bit, and then

Table 2 The truth table of a one-bit subtractor

Minuend bit	Subtrahend bit	Previous borrow bit	Difference bit	Borrow bit
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

show how this may be used as a building block for a subtractor using bit-strings of arbitrary length.

A one-bit subtractor is a function that forms the arithmetic subtraction of three input bits. It consists of three inputs and two outputs. Two of the input bits represent minuend and subtrahend bits to be subtracted. The third input represents the borrow bit from the previous higher significant position. The first output gives the value of the difference for minuend and subtrahend bits to be subtracted. The second output gives the value of the borrow bit to minuend and subtrahend bits to be subtracted. The truth table of the one-bit subtractor is described in Table 2.

Suppose that the two one-bit binary numbers $p_{o,q}$ and $p_{o+1,q}$ denoted in Sect. 4.8, represent the first input and the first output of a one-bit subtractor for $1 \leq o \leq (2 * k^2 + 4 * k)$ and $1 \leq q \leq (2 * k)$. Also suppose a one-bit binary number n_j denoted in Sect. 4.1, represents the second input of a one-bit subtractor for $1 \leq j \leq k$, and two one-bit binary numbers $b_{o,q}$ and $b_{o,q-1}$ represent the second output and the third input of a one-bit subtractor. Two *distinct* DNA sequences are designed to encode every bit $b_{o,q-1}$ and $b_{o,q}$. For the sake of convenience in our presentation, assume that $b^1_{o,q}$ contains the value of $b_{o,q}$ to be 1 and $b^0_{o,q}$ contains the value of $b_{o,q}$ to be 0. Similarly, also suppose that $b^1_{o,q-1}$ contains the value of $b_{o,q-1}$ to be 1 and $b^0_{o,q-1}$ contains the value of $b_{o,q-1}$ to be 0. The following algorithm is proposed to finish the function of a parallel one-bit subtractor.

```
Procedure ParallelOneBitSubtractor($T_0^{>=}, o, q, j$)
    (1) Separate($T_0^{>=}$, $p^1_{o,q}$, $T_1^{ON}$, $T_2^{OFF}$).
    (2) Separate($T_1^{ON}$, $n^1_j$, $T_3^{ON}$, $T_4^{OFF}$).
    (3) Separate($T_2^{OFF}$, $n^1_j$, $T_5$, $T_6$).
    (4) Separate($T_3^{ON}$, $b^1_{o,q-1}$, $T_7$, $T_8$).
    (5) Separate($T_4^{OFF}$, $b^1_{o,q-1}$, $T_9$, $T_{10}$).
    (6) Separate($T_5$, $b^1_{o,q-1}$, $T_{11}$, $T_{12}$).
    (7) Separate($T_6$, $b^1_{o,q-1}$, $T_{13}$, $T_{14}$).
    (8a) If (Detect($T_7$) == true) then
        (8) Appendhead($T_7$, $p^1_{o+1,q}$) and Appendhead($T_7$, $b^1_{o,q}$).
    EndIf
    (9a) If (Detect($T_8$) == true) then
        (9) Appendhead($T_8$, $p^0_{o+1,q}$) and Appendhead($T_8$, $b^0_{o,q}$).
    EndIf
    (10a) If ($Detect(T_9) == true$) then
        (10) $Appendhead(T_9, p^0_{o+1,q})$ and $Appendhead(T_9, b^0_{o,q})$.
    EndIf
    (11a) If ($Detect(T_{10}) == true$) then
        (11) $Appendhead(T_{10}, p^1_{o+1,q})$ and $Appendhead(T_{10}, b^0_{o,q})$.
    EndIf
    (12a) If ($Detect(T_{11}) == true$) then
        (12) $Appendhead(T_{11}, p^0_{o+1,q})$ and $Appendhead(T_{11}, b^1_{o,q})$.
    EndIf
    (13a) If ($Detect(T_{12}) == true$) then
        (13) $Appendhead(T_{12}, p^1_{o+1,q})$ and $Appendhead(T_{12}, b^1_{o,q})$.
    EndIf
    (14a) If ($Detect(T_{13}) == true$) then
        (14) $Appendhead(T_{13}, p^1_{o+1,q})$ and $Appendhead(T_{13}, b^1_{o,q})$.
    EndIf
    (15a) If ($Detect(T_{14}) == true$) then
        (15) $Appendhead(T_{14}, p^0_{o+1,q})$ and $Appendhead(T_{14}, b^0_{o,q})$.
    EndIf
    (16) $T_0^{>=} = Merge(T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14})$.
EndProcedure
```

Lemma 12 *The algorithm*, ParallelOneBitSubtractor($T_0^{>=}, o, q, j$), *can be applied to finish the function of a parallel one-bit subtractor.*

Proof Similar to Algorithm 1 and Lemma 1. □

The one-bit subtractor just introduced figures out the difference bit and the borrow bit for two input bits and a previous borrow. Two k-bit binary numbers of can finish subtractions of k times by means of this one-bit subtractor. A binary parallel subtractor is a function that finishes the arithmetic subtraction for two k-bit binary numbers. The following algorithm is proposed to finish the function of a binary parallel subtractor.

```
Procedure BinaryParallelSubtractor($T_0^{>=}, o, q, u$)
    (1) For $j = 1$ to $k$
        (1a) ParallelOneBitSubtractor($T_0^{>=}, o + (u - 1) * (k + 2)$,
                $2 * k - (o - 1) - (k - j), j$).
        EndFor
    (2) For $j = (2 * k) - (o - 1) + 1$ to $2 * k$
        (2a) $Separate(T_0^{>=}, p^1_{o+(u-1)*(k+2),j}, T_1, T_2)$.
        (2b) $Separate(T_1, b^1_{o+(u-1)*(k+2),j-1}, T_3, T_4)$.
        (2c) $Separate(T_2, b^1_{o+(u-1)*(k+2),j-1}, T_5, T_6)$.
        (2d) If ($Detect(T_3) == true$) then
            (2d0) $Appendhead(T_3, p^0_{o+(u-1)*(k+2)+1,j})$ and
                        $Appendhead(T_3, b^0_{o+(u-1)*(k+2),j})$.
        EndIf
        (2e) If ($Detect(T_4) == true$) then
            (2e0) $Appendhead(T_4, p^1_{o+(u-1)*(k+2)+1,j})$ and
                    $Appendhead(T_4, b^0_{o+(u-1)*(k+2),j})$.
        EndIf
        (2f) If ($Detect(T_5) == true$) then
            (2f0) $Appendhead(T_5, p^1_{o+(u-1)*(k+2)+1,j})$ and
                    $Appendhead(T_5, b^1_{o+(u-1)*(k+2),j})$.
        EndIf
        (2g) If ($Detect(T_6) == true$) then
            (2g0) $Appendhead(T_6, p^0_{o+(u-1)*(k+2)+1,j})$ and
                    $Appendhead(T_6, b^0_{o+(u-1)*(k+2),j})$.
        EndIf
        (2h) $T_0^{>=} = Merge(T_1, T_2, T_3, T_4, T_5, T_6)$.
    EndFor
EndProcedure
```

Lemma 13 *The algorithm*, BinaryParallelSubtractor($T_0^{>=}, o, q, u$), *can be applied to finish the function of a binary parallel subtractor.*

Proof Similar to Algorithm 1 and Lemma 1. □

4.12 The construction of a truncated assignment operator

A truncated assignment operator is an instruction of the first operand of k bits and the second operand of $(2 * k)$ bits that the value of the first operand is set to the value of the least significant k bits to the second operand. Assume that the second operand is the remainder obtained from a dividend of $(2 * k)$ bits and a divisor of k bits in a divider. Also, suppose that the first operand is a cipher-text C denoted in Sect. 4.4. The following algorithm is applied to construct a truncated assignment operator. This implies that the truncated assignment operator can be used to update a cipher-text C. The second parameter, u, in the algorithm is employed to represent the uth arithmetic division. The third parameter, a, in the algorithm is used to represent the ath updating for a cipher-text C.

```
Procedure TruncatedAssignmentOperator($T_0, u, a$)
    (1) For $q = 1$ to $k$
        (1a) $Separate(T_0, p^1_{u*(k+2),q}, T_1, T_2)$.
        (1b) $Appendhead(T_1, c^1_{a,q})$.
        (1c) $Appendhead(T_2, c^0_{a,q})$.
        (1d) $T_0 = Merge(T_1, T_2)$.
    EndFor
EndProcedure
```

Lemma 14 *The algorithm*, TruncatedAssignmentOperator(T_0, u, a), *can be applied to finish the function of a truncated assignment operator.*

Proof Similar to Algorithm 1 and Lemma 1. □

5 Finding the correspondence between a plain-text and a cipher-text

The RSA public-key cryptosystem can be used to encrypt messages sent between two communicating parties. Assume that the encrypted message received by one of two communicating parties and also overheard by an eavesdropper can be represented as a k-bit binary number, $c_{(2*k+1),k} \ldots c_{(2*k+1),1}$, denoted in Sect. 4.4. An eavesdropper only needs to use the following procedure to find the correspondence of between a plain-text and a cipher-text. The first parameter, T_0, in the procedure, CorrespondtoPlaintextCiphertext(T_0), is generated from Algorithm 1.

```
Procedure CorrespondtoPlaintextCiphertext($T_0$)
  (1) For $j = 1$ to $k$
      (1a) $Separate(T_0, c_{(2*k+1),j}, T_1, T_2)$.
      (1b) $T_0 = Merge(T_0, T_1)$.
    EndFor
  (2) If $(Detect(T_0) == true)$ then
      (2a) $Read(T_0)$.
    EndIf
EndAlgorithm
```

Theorem 2 *From those steps in the procedure*, CorrespondtoPlaintextCiphertext(T_0), *an eavesdropper can find the correspondence of between a plain-text and a cipher-text.*

Proof Refer to Algorithm 1. □

5.1 The power of the proposed algorithm for solving an instance of the RSA public-key cryptosystem

It is supposed that the encrypted message $c^0_{9,4}c^0_{9,3}c^0_{9,2}c^1_{9,1}$ is overheard by an eavesdropper. It is assumed that the value of k is equal to four. The eavesdropper makes use of CorrespondtoPlaintextCiphertext(T_0) to find the corresponding plaintext. The first parameter, T_0, in CorrespondtoPlaintextCiphertext(T_0) is produced from Algorithm 1. Therefore, when Algorithm 1 in Sect. 3 is executed, on the execution of Step (1), MakeValue(T_n) is called. It is supposed that the value of n is equal to 15 (3×5), and its length is four bits. After each operation in MakeValue(T_n) is completed, tube $T_n = \{n^1_4 n^1_3 n^1_2 n^1_1\}$. Next, on the execution of Step (2) in Algorithm 1, Init(T_0, T_n) is invoked. The first parameter, T_0, is an empty tube and the second parameter, T_n, is $\{n^1_4 n^1_3 n^1_2 n^1_1\}$. It is assumed that the length of a plaintext M is four bits. After each operation in Init(T_0, T_n) is completed, the result for tubes T_0 and T_n is shown in Table 3.

Next, on the execution of Step (3) in Algorithm 1, MakeInitialValue(T_e) is called. The first parameter, T_e, is an empty tube. It is assumed that the length of a public

Table 3 The result for tubes T_0 and T_n is yielded by Init(T_0, T_n)

Tube	The result is yielded by Init(T_0, T_n)
T_0	$\{m_4^0 m_3^0 m_2^0 m_1^0, m_4^0 m_3^0 m_2^0 m_1^1, \ldots, m_4^1 m_3^1 m_2^1 m_1^1\}$
T_n	$\{n_4^1 n_3^1 n_2^1 n_1^1\}$

Table 4 The result for tubes T_e is generated by MakeInitialValue(T_e)

Tube	The result is produced by MakeInitialValue(T_e)
T_e	$\{e_3^0 e_2^1 e_1^1 e_0^1\}$

Table 5 The result for tube T_0 is produced by InitialEncryptedForm(T_0)

Tube	The result is yielded by InitialEncryptedForm(T_0)
T_0	$\{c_{1,4}^0 c_{1,3}^0 c_{1,2}^0 c_{1,1}^1 m_4^0 m_3^0 m_2^0 m_1^0, c_{1,4}^0 c_{1,3}^0 c_{1,2}^0 c_{1,1}^1 m_4^0 m_3^0 m_2^0 m_1^1, \ldots, c_{1,4}^0 c_{1,3}^0 c_{1,2}^0 c_{1,1}^1 m_4^1 m_3^1 m_2^1 m_1^1\}$

Table 6 The results for tubes T_0 and T_e are produced by Step (5a) through Step (5n) in Algorithm 1

Tube	The result is generated by Step (5a) through Step (5n)
T_0	$\{c_{9,4}^0 c_{9,3}^0 c_{9,2}^0 c_{9,1}^0 \cdots c_{1,4}^0 c_{1,3}^0 c_{1,2}^0 c_{1,1}^1 m_4^0 m_3^0 m_2^0 m_1^0,$ $c_{9,4}^0 c_{9,3}^0 c_{9,2}^0 c_{9,1}^1 \cdots c_{1,4}^0 c_{1,3}^0 c_{1,2}^0 c_{1,1}^1 m_4^0 m_3^0 m_2^0 m_1^1, \ldots\}$
T_e	$\{e_3^0 e_2^1 e_1^1 e_0^1\}$

Table 7 The result for tube T_0 is yielded by Steps (1a) and (1b) in CorrespondtoPlaintext-Ciphertext(T_0)

Tube	The result is produced by Steps (1a) and (1b)
T_0	$\{c_{9,4}^0 c_{9,3}^0 c_{9,2}^0 c_{9,1}^1 \cdots c_{1,4}^0 c_{1,3}^0 c_{1,2}^0 c_{1,1}^1 m_4^0 m_3^0 m_2^0 m_1^1\}$

key e is four bits and its value is 7. After each operation in MakeInitialValue(T_e) is performed, the result for tube T_e is shown in Table 4.

Next, on the execution of Step (4) in Algorithm 1, after each operation in InitialEncryptedForm(T_0) is completed, the result for tube T_0 is shown in Table 5.

Next, each operation from Step (5a) through Step (5n) in the only loop in Algorithm 1 is used to complete the encryption of each plaintext. This implies that those operations are applied to complete all of the computations for 0^7 (mod 15), 1^7 (mod 15), 2^7 (mod 15), and so on with 15^7 (mod 15). After those operations are all completed, the results for tubes T_0 and T_e are shown in Table 6.

Next, since each operation in Algorithm 1 is completed, an eavesdropper can continue to execute each operation from Step (1a) through Step (1b) in CorrespondtoPlaintextCiphertext(T_0). The value of k is equal to four, so Steps (1a) and (1b) will be executed four times. The eavesdropper overhears the encrypted message $c_{9,4}^0 c_{9,3}^0 c_{9,2}^0 c_{9,1}^1$, and the encrypted message $c_{9,4}^0 c_{9,3}^0 c_{9,2}^0 c_{9,1}^1$ is applied to find the corresponding plaintext. Hence, after each operation of Steps (1a) and (1b) is completed, the result for tube T_0 is shown in Table 7.

Next, a *true* is returned from the execution of Step (2) in CorrespondtoPlaintext-Ciphertext(T_0), so the execution of Step (2a) is used to obtain the answer (the corresponding plaintext), $m_4^0 m_3^0 m_2^0 m_1^1$. This indicates that for the encrypted message $c_{9,4}^0 c_{9,3}^0 c_{9,2}^0 c_{9,1}^1$ the corresponding plaintext is $m_4^0 m_3^0 m_2^0 m_1^1$.

6 Complexity assessment

Theorem 3 *Suppose that the length of a plain-text in the RSA public-key cryptosystem is k bits. The corresponding table for between each plain-text and each cipher-text in the RSA public-key cryptosystem can be constructed with $O(k^3)$ biological operations from solution space of DNA strands.*

Proof Refer to Algorithm 1. □

Theorem 4 *Suppose that the length of a plain-text in the RSA public-key cryptosystem is k bits The corresponding table for between each plain-text and each cipher-text in the RSA public-key cryptosystem can be constructed with $O(2^k)$ library strands from solution space of DNA strands.*

Proof Refer to Algorithm 1. □

Theorem 5 *Suppose that the length of a plain-text in the RSA public-key cryptosystem is k bits. The corresponding table for between each plain-text and each cipher-text in the RSA public-key cryptosystem can be constructed with $O(c)$ tubes from solution space of DNA strands, where c is a constant value.*

Proof Refer to Algorithm 1. □

Theorem 6 *Suppose that the length of a plain-text in the RSA public-key cryptosystem is k bits. The corresponding table for between each plain-text and each cipher-text in the RSA public-key cryptosystem can be constructed with the longest library strand, $O(k^3)$, from solution space of DNA strands.*

Proof Refer to Algorithm 1. □

7 Biological implementation

Experimental implementation of biological operations is very important for deciding whether any DNA-based algorithm to dealing with any problem can obtain the required answer(s) or not. The content of the following subsections cited from [23] is used to describe how biological operations used in the proposed algorithm are implemented.

7.1 Separate(T, s, T_1, T_2)

Affinity purification is used to extract any strands from T containing s. The implementation of the biological operation uses a probe sequence that is complementary to

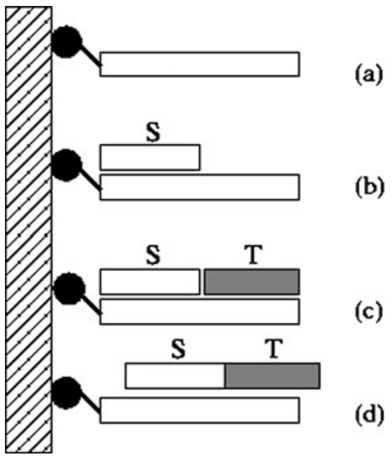

Fig. 2 Concatenation process: (**a**) Linker strand affixed to surface. (**b**) S anneals to linker strand. (**c**) T anneals to linker strand, adjacent to S. (**d**) S and T ligated to form a single strand, which is then freed by heating the solution

the target sequence being found for. Probes are fixed to a surface, and capture through annealing any strands consisting of the target sequence. Then captured strands perhaps are separated from the rest of the population by placing them in a separate solution, which is heated to break the bonds between the probes and the target sequence. Hence, the probe applied is the complementary sequence of s. Retained strands are placed in one new tube T_1 and the remainder are placed in another new tube T_2.

7.2 Merge($\{T_i\}$)

The contents of tubes $\{T_i\}$ are simply merged by means of pouring. The number of tubes will generally be low, so this is considered to be a constant-time operation.

7.3 Discard(T)

The contents of T are discarded, and T is replaced by a new, empty tube.

7.4 Amplify(T, $\{T_i\}$)

The polymerase chain reaction (PCR) is employed, with its initial input that is tube T. This reaction is applied to massively amplify (possibly small) amounts of DNA strands that begin and end with specific primer sequences. Since every strand in tube T is delimited by these sequences, they are all copied by the reaction. Then the result of the PCR is divided equally between the specified numbers of tubes (the number of PCR cycles may therefore be adjusted to ensure a constant DNA volume per tube, regardless of the number of tubes).

7.5 Concatenate(s_1, s_2)

Two strands (labeled S and T in Fig. 2) may be concatenated by the following process: create a *linker* strand, which has a sequence that is the complement of S followed by the complement of T. The linker strand is affixed to a surface with a magnetic bead (Fig. 2(a)). Then strand S is added to the solution, and anneals with the linker strand at the appropriate position (Fig. 2(b)). Then, strand T is added to the solution, and this also anneals with the linker strand, at a position immediately adjacent to strand S (Fig. 2(c)). Next, the ligase enzyme is added to the solution to seal the "nick" between S and T, forming a single strand which may be freed by heating the solution to break its bonds with the linker strand (Fig. 2(d)).

7.6 Append-head(U, s)

The implementation of the *concatenate*( ) operation denoted above may easily be applied to append a specific sequence, s, to the head of each strand in a tube U. The sequence s corresponds, in this case, to the strand S defined in Fig. 2, and strand T in Fig. 2 corresponds to the beginning sequence of every strand in the tube. In this case, only the beginning sequence of every strand anneals to the linker strand. Clearly, then after a series of *append-head*( ) operations has been completed on a strand, its sequence will be made up of a number of sequences representing bit-strings.

7.7 Detect(T)

The tube T is run through a gel electrophoresis process, which is generally used to sort DNA strands on length. Any DNA present in T shows up as a visible band in the gel; if DNA strands of the appropriate length are present, the operation returns *true*. If there are no visible bands corresponding to DNA of the correct length, then the operation returns *false*. The length criterion is applied to ensure that DNA fragments present do not cause a false positive result. If the DNA in the band corresponding to the contents of T is required in a subsequent processing step, the band may be excised from the gel by cutting and then soaked to remove the strands for further use.

8 Conclusions

The number of steps any classical computer requires in order to find discrete logarithm of a k-bit [26] and to factor integers of a k-bit natural number [1] increases exponentially with k, at least by means of using algorithms known at present. Shor's *quantum* factoring and discrete logarithm algorithm [27] includes that the two main components, modular exponentiation (computation of $a^x \bmod n$), and the inverse quantum Fourier transform (QFT) take only $O(k^3)$ operations. From [28], factoring integers of a k-bit natural number was solved by the proposed DNA-based algorithm with polynomial-time biological operations. From [29], for solving elliptic curve discrete logarithm it takes a series of steps that is polynomial in the input size. From [30], The security of the Diffie–Hellman public-key cryptosystem is broken by the proposed DNA-based algorithm in which it takes a series of steps that is polynomial in the input size.

The proposed algorithm (Algorithm 1) for constructing the corresponding table between each plain-text and each cipher-text in the RSA public-key cryptosystem is based on biological operations in Adleman's experiments. This algorithm has several advantages. First, the Adleman program [22] is used to generate good DNA sequences to construct the corresponding table between any plain-text and any cipher-text in the RSA public-key cryptosystem. Good DNA sequences are applied to decrease a rate of errors for hybridization. This indicates that the proposed algorithm actually has a lower rate of errors for hybridization. Secondly, the basic biological operations in Adleman's experiments are employed to finish the function of a k-bit parallel adder, the function of a k-bit parallel subtractor, the function of a k-bit parallel multiplier, the function of a $(2 * k)$-bit parallel divider, the function of a k-bit parallel comparator, the function of a $(2 * k)$-bit parallel assignment operator, and the function of a k-bit parallel truncated assignment operator. This means that the proposed algorithm has the computational capability of mathematics to finish addition ("$+$"), subtraction ("$-$"), multiplication ("$*$"), division ("$\div$"), and assignment ("$=$") operations. Thirdly, after the tube containing the strands encoding every (plain-text, cipher-text) pair is produced from Algorithm 1, the number of steps for decoding an encrypted message overheard by an eavesdropper is linear in the size of the encrypted message overheard.

Currently, the future of molecular computers is unclear. It is possible that in the future molecular computers will be the clear choice for performing massively parallel computations. However, there are still many technical difficulties to overcome before this becomes a reality. We hope that this paper helps to demonstrate that molecular computing is a technology worth pursuing.

Acknowledgements The authors would like to give many thanks to Dr. Amos who is the author of the 6th reference and the 23th reference for proposing valuable information on Sect. 7 entitled "Biological Implementation."

References

1. Rivest RL, Shamir A, Adleman L (1978) A method for obtaining digital signatures and public-key crytosystem. Commun ACM 21:120–126
2. Feynman RP (1961) There's plenty of room at the bottom. In: Gilbert DH (ed) Minaturization. Reinhold, New York, pp 282–296
3. Adleman L (1994) Molecular computation of solutions to combinatorial problems. Science 266:1021–1024
4. Lipton RJ (1995) DNA solution of hard computational problems. Science 268:542–545
5. Quyang Q, Kaplan PD, Liu S, Libchaber A (1997) DNA solution of the maximal clique problem. Science 278:446–449
6. Amos M (1997) DNA computation. PhD thesis, Department of Computer Science, the University of Warwick
7. Harju T, Li C, Petre I, Rozenberg G (2005) Parallelism in gene assembly. In: DNA computing. Lecture notes in computer science, vol 3384, p 686. doi:10.1007/11493785_12
8. Thachuk C, Manuch J, Rafiey A, Mathieson L-A, Stacho L, Condon A (2010) An algorithm for the energy barrier problem without pseudoknots and temporary arcs. Pac Symp Biocomput 15:108–119
9. Zadeh JN, Wolfe BR, Pierce NA (2010) Nucleic acid sequence design via efficient ensemble defect optimization. J Comput Chem. doi:10.1002/jcc.21633
10. Xiao D, Li W, Zhang Z, He L (2005) Solving the maximum cut problems in the Adleman–Lipton model. Biosystems 82:203–207
11. Yeh C-W, Chu C-P, Wu K-R (2006) Molecular solutions to the binary integer programming problem based on DNA computation. Biosystems 83(1):56–66

12. Zhang DY, Turberfield AJ, Yurke B, Winfree E (2007) Engineering entropy-driven reactions and networks catalyzed by DNA. Science 318(5853):1121–1125
13. Boneh D, Dunworth C, Lipton RJ (1996) Breaking DES using a molecular computer. In: Proceedings of the 1st DIMACS workshop on DNA based computers, 1995. DIMACS series in discrete mathematics and theoretical computer science, vol 27. American Mathematical Society, Providence, pp 37–66
14. Adleman L, Rothemund PWK, Roweis S, Winfree E (1999) On applying molecular computation to the data encryption standard. In: The 2nd annual workshop on DNA computing, Princeton University. DIMACS series in discrete mathematics and theoretical computer science. American Mathematical Society, Providence, pp 31–44
15. Zhang DY, Seelig G (2011) DNA-based fixed gain amplifiers and linear classifier circuits. In: DNA 16. Lecture notes in computer science, vol 6518, p 176
16. Yeh C-W, Chu C-P (2008) Molecular verification of rule-based systems based on DNA computation. IEEE Trans Knowl Data Eng 20(7):965–975
17. Guarnieri F, Fliss M, Bancroft C (1996) Making DNA add. Science 273:220–223
18. Ho M(S-H) (2005) Fast parallel molecular solutions for DNA-based supercomputing: the subset-product problem. Biosystems 80:233–250
19. Ahrabian H, Nowzari-Dalini A (2004) DNA simulation of nand Boolean circuits. Adv Model Optim 6(2):33–41
20. Schuster A (2005) DNA databases. Biosystems 81:234–246
21. Paun G, Rozenberg G, Salomaa A (1998) DNA computing: new computing paradigms. Springer, New York. ISBN:3-540-64196-3
22. Boneh D, Dunworth C, Lipton RJ, Sgall J (1996) On the computational power of DNA. Discrete Appl Math 71:79–94. Special Issue on Computational Molecular Biology
23. Amos M (2005) Theoretical and experimental DNA computation. Springer, Berlin
24. Braich RS, Johnson C, Rothemund PWK, Hwang D, Chelyapov N, Adleman LM Solution of a satisfiability problem on a gel-based DNA computer. In: Proceedings of the 6th international conference on DNA computation. Lecture notes in computer science. Springer, Berlin
25. Braich RS, Johnson C, Rothemund PWK, Hwang D, Chelyapov N, Adleman LM (2002) Solution of a 20-variable 3-SAT problem on a DNA computer. Science 296(5567):499–502
26. Diffie W, Hellman M (1976) New directions in cryptography. IEEE Trans Inf Theory IT-22(6):644–654
27. Shor PW (1997) Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J Comput 26(5):1484–1509
28. Chang W-L, Ho M, Guo M (2005) Fast parallel molecular algorithms for DNA-based computation: factoring integers. IEEE Trans Nanobiosci 4(2):149–163
29. Li K, Zou S, Xv J (2008) Fast parallel molecular algorithms for DNA-based computation: solving the elliptic curve discrete logarithm problem over GF(2^n). J Biomed Biotechnol 2008:518093. doi:10.1155/2008/518093
30. Chang W-L, Huang S-C, Lin KW, Ho M(SH) (2009) Fast parallel DNA-based algorithms for molecular computation: discrete logarithm. J Supercomput