Breaking DES Using a Molecular Computer

Dan Boneh
dabo@cs.princeton.edu

Christopher Dunworth
ctd@cs.princeton.edu

Richard J. Lipton*
rjl@cs.princeton.edu

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

Recently Adleman [1] has shown that a small traveling salesman problem can be solved by molecular operations. In this paper we show how the same principles can be applied to breaking the Data Encryption Standard (DES). Our method is based on an encoding technique presented in Lipton [8]. We describe in detail a library of operations which are useful when working with a molecular computer. We estimate that given one arbitrary (plain-text, cipher-text) pair, one can recover the DES key in about 4 months of work. Furthermore, if one is given cipher-text, but the plain text is only known to be one of several candidates then it is still possible to recover the key in about 4 months of work. Finally, under chosen cipher-text attack it is possible to recover the DES key in one day using some preprocessing.

1 Introduction

Due to advances in molecular biology it is nowadays possible to create a soup of roughly 10^{18} DNA strands that fits in a small glass of water. Adleman [1] has shown that each DNA strand can be used to perform computations. Thus, a small test tube containing DNA strands seems to have more computing power than the most powerful parallel computers. The drawback of this approach is that basic operations using DNA take a long time (e.g. 3 hours). Thus, we are capable of performing 10^{18} basic operations at once, though each operation requires several hours to complete. Throughout the paper we will refer to operations done on DNA strand as bio-steps. This will be explained in more detail in Section 2.

Recently, Lipton [8] has come up with an encoding scheme that enabled him to solve the satisfiability problem of formulas with a small number of variables. A generalization of this scheme [3] can be used to solve the satisfiability problem for circuits. Using these methods, the number of bio-steps required to find a satisfying assignment is proportional to the size of the circuit. Lipton's approach raises the hope of using molecular computing to solve hard problems that come up in practice. However, to make this procedure practical one needs to encode the problem at hand so as to minimize the number of bio-steps required to solve it.

In this paper we give an example of a hard problem that can be solved using a molecular computer. We present a "molecular program" for breaking the Data Encryption standard [6] or DES for short. We precisely describe the steps needed to implement this scheme and show efficient ways of reducing the number of total bio-steps required.

*Supported in part by NSF CCR–9304718.

DES is a widely used encryption procedure. It encrypts 64 bit messages and uses a 56 bit key. By breaking DES we mean that given one (plain-text, cipher-text) pair we can find a key mapping the plain-text to the cipher-text. In fact, we can do even better. Suppose we were able to intercept some cipher-text, but we do not know what the plain-text is. What we do know is that the plain-text is one of several possible candidates. Then we can recover the set of keys mapping the candidate plain-texts to the intercepted cipher-text for the same amount of work as when the plain-text was known. Finally, using chosen cipher text attack[1] it is possible to recover the key in one day of work using some preprocessing. This means that after the preprocessing work is done it is possible to break many DES systems for very little work.

Several researchers [11, 2, 10] have come up with various methods for implementing non-deterministic Turing Machines using molecular computers. Clearly a non-deterministic Turing Machine can break any crypto-system, including DES, by guessing the correct key. Though these results are very important theoretically, they are not useful in practice. For instance, breaking DES using a Turing Machine would require millions of biological operations. Running such experiments will take hundreds of years.

It should be pointed out that attacks using differential cryptanalysis methods have proven to be very useful for breaking DES [7, 9]. The central achievement of these attacks is that DES can be broken on a conventional computer in 2^{43} steps. This means that DES can be broken within several days on a conventional computer. Note however, that the differential cryptanalysis methods require 2^{43} pairs of (plain-text, cipher-text), while our molecular computer requires only one such pair. In fact, as was mentioned above, a molecular computer can break DES with even less information.

Another conventional attack on DES was suggested by Wiener [12]. Wiener predicted that using dedicated hardware which costs $1 million it is possible to run through all 2^{56} DES keys in 7 hours. An advantage of the molecular attack is that the plain-text need not be known exactly. Furthermore, if DES was made to use 64 bit keys, the conventional attack would slow down by a factor of 256, while the molecular attack would be uneffected.

Throughout this paper we will assume an error free model. This means that all the molecular biology experiments work perfectly without any errors. Even though this assumption does not hold in practice, it will do for now since the objective of this paper is to present general procedures for performing efficient computations using DNA. The first steps towards making molecular computers resistant to errors was taken in [5].

In Section 2 we introduce the basic functions of a molecular computer. We then go on to briefly sketch the DES circuit in Section 3. Sections 4 and 5 describe the main steps of the procedure.

2 Overview of molecular computing

In order to understand the computation described here, it is necessary to gain a fundamental understanding of DNA's structure and function. The reader who is familiar with this information may wish to skip to Section 2.1, which covers our notation.

DNA (deoxyribonucleic acid) is found in every living creature as the storage medium for genetic information. It is comprised of subunits called nucleotides that are strung together into polymer chains (the chemical structure of a single nucleotide is shown in Figure 1). DNA polymer chains are more commonly called DNA *strands*, and short strands are called *oligonucleotides*, or simply *oligos*.

There are four kinds of nucleotides in DNA, distinguished by the chemical group, or base, attached

[1] Chosen cipher text attack means that we can obtain a (plain-text, cipher-text) pair where we get to choose the cipher-text.

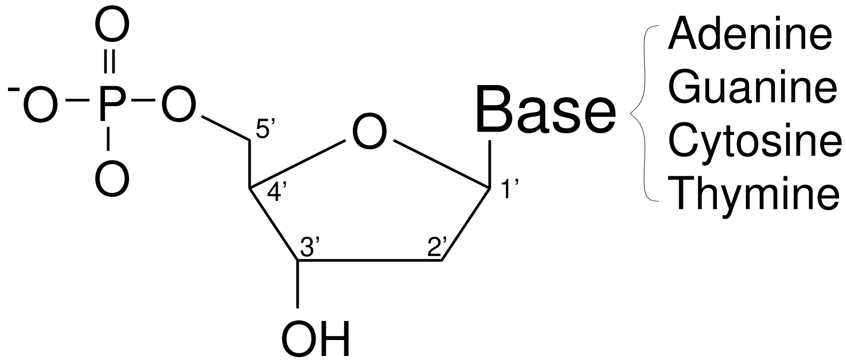

Figure 1: Chemical structure of a nucleotide

to it. The four bases are adenine, guanine, cytosine, and thymine, abbreviated as A, G, C, and T (we will henceforth use these letters to refer to nucleotides containing these bases). Single nucleotides are linked together end-to-end to form DNA strands in a process called polymerization. This linking occurs via a reaction between the 5' phosphate of one nucleotide and the 3' hydroxyl of another (see Figure 2). Note that every DNA strand will have two distinct ends—one with a free 5' PO_4 group and the other with a free 3' OH group, referred to as the 5' and 3' ends, respectively. Thus, we can think of the sequence of nucleotides in any strand as having a natural orientation. This orientation is important to remember because it restricts the kinds of operations that one can perform on the strands.

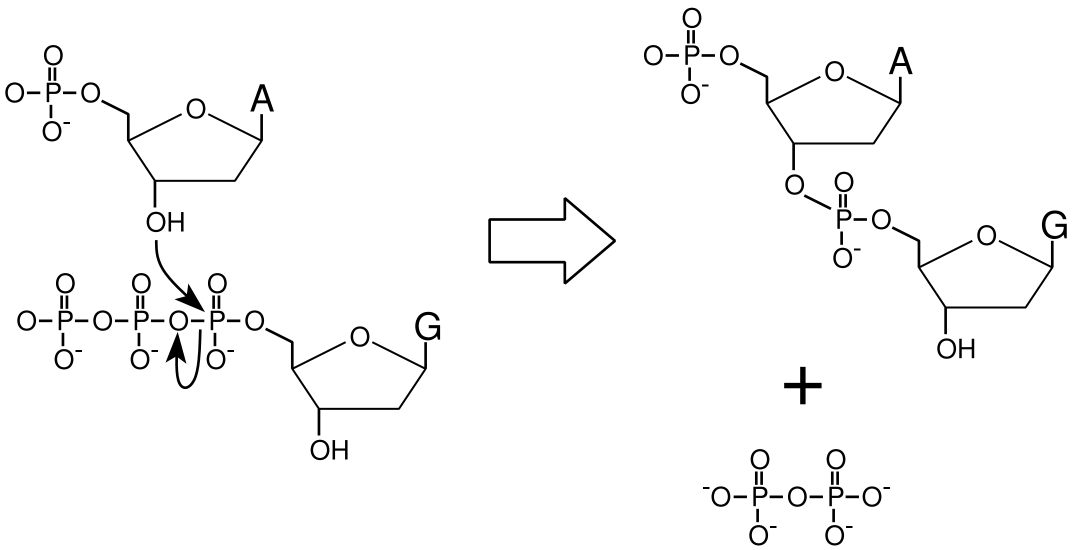

Figure 2: DNA polymerization reaction

DNA does not usually exist in nature as free single strands, though. Under appropriate conditions single strands will pair up and twist around each other, forming the famous double helix structure discovered by Watson and Crick. This pairing occurs because of a mutual attraction, called hydrogen bonding, that exists between As and Ts and between Gs and Cs. Hence, A/T and G/C are called *complementary* base pairs. A further condition is that hydrogen bonding will only occur if the pair of complementary strands are oriented in an *antiparallel* fashion. That is, as one strand of a double helix runs in the 5' → 3' direction, its complement runs in the 3' → 5' direction. For example, if we have the strands `5'-AAGCGTAG-3'`, `3'-TTCGCATC-5'`, and `5'-TTCGCATC-3'`, the first two can pair up into a double strand—namely, $\begin{array}{l}\texttt{5'-AAGCGTAG-3'}\\ \texttt{3'-TTCGCATC-5'}\end{array}$ —but the first and last ones cannot, because when lined up in an antiparallel fashion we see that they are not complementary: $\begin{array}{l}\texttt{5'-AAGCGTAG-3'}\\ \texttt{3'-CTACGCTT-5'}\end{array}$. This process of complementary strands coming together to form a double helix is called *annealing*, and the reverse process—a double helix coming apart to yield its two constituent single strands— is called *melting*.

Of course, DNA strands would not be of much use to us unless we had some way to manipulate them. To accomplish this we use any of a number of commercially available enzymes that perform the tasks we require. One class of enzymes, called restriction endonucleases, will recognize a specific short sequence in a strand and then "cut" the strand at that location. Another class of enzymes, the DNA polymerases, will read a single DNA strand, called the "template" strand, and create its complementary strand. Still another enzyme, called DNA ligase, will hook together, or ligate, the free 5' end of one strand to the free 3' end of another strand. There are many other enzymes that could potentially be useful, but for our computation these are sufficient.

2.1 Notation

We have devised a notation for abstractly representing DNA strands in order to succinctly describe the changes they undergo during various operations. The full details of this notation are described in [4].

First, we represent strings over the alphabet {`A`,`C`,`G`,`T`} using letters for variable names (e.g. x = `ACCTGAC`). Note that x is a *string*, but not yet a *strand*—that is, it's just a sequence of characters, with no orientation implied. We indicate the concatenation of two or more strings by juxtaposition (e.g. if y = `AAATAAG`, then xy = `ACCTGACAAATAAG`).

Two operations that we would like to apply to strings are complement and reverse. The *complement* of a string x, denoted $\hat{x}$, is the string that results when each character of x has been replaced by its Watson-Crick complement. That is, we apply the following mapping to every character in x: `A`→`T`, `C`→`G`, `G`→`C`, `T`→`A`. Thus, for x = `ACCTGAC`, we have $\hat{x}$ = `TGGACTG`. The *reverse* of a string x, denoted x^R, is just the string x in reverse order. Thus for the same example, x^R = `CAGTCCA`.

DNA strands are more than just strings, though—they are strings *with an orientation.* To this end we introduce some new operators that, when applied to a string, denote a DNA strand based on that string.

The first such operator is "$\upharpoonright$". $\upharpoonright x$ denotes the single DNA strand whose sequence is x and whose orientation is $5' \rightarrow 3'$, as x is read from left to right. For example, if x = `ACCTGAC`, then $\upharpoonright x$ is the strand `5'-ACCTGAC-3'`. The second such operator is "$\downharpoonleft$". $\downharpoonleft x$ denotes the strand complementary to $\upharpoonright x$. For our example, $\downharpoonleft x$ = `3'-TGGACTG-5'`. Finally, we have the operator "$\updownharpoons$". $\updownharpoons x$ denotes the double strand that results when $\upharpoonright x$ and $\downharpoonleft x$ anneal in solution.

The table below summarizes these operators:

$$
\begin{aligned}
x &= \texttt{ACCTGAC} \\
y &= \texttt{AAATAAG} \\
\\
xy &= \texttt{ACCTGACAAATAAG} \\
\hat{x} &= \texttt{TGGACTG} \\
x^R &= \texttt{CAGTCCA} \\
\hat{x^R} &= \texttt{GTCAGGT} \qquad (= \hat{x}^R) \\
\\
\uparrow x &= \texttt{5'-ACCTGAC-3'} \\
\downarrow x &= \texttt{3'-TGGACTG-5'} \qquad\qquad (\text{note: } \uparrow \hat{x}^R = \texttt{5'-GTCAGGT-3'} = \downarrow x) \\
\updownarrow x &= \begin{smallmatrix}\texttt{5'-ACCTGAC-3'} \\ \texttt{3'-TGGACTG-5'}\end{smallmatrix}
\end{aligned}
$$

The convenience of this notation becomes clearer when we try to describe various operations on DNA strands in solution. For example, we can easily see that the strand $\uparrow xy$ and the strand $\downarrow yz$ will anneal under appropriate conditions to form the expected duplex $\uparrow x \updownarrow y \downarrow z$. We will rely on this notation to describe aspects of our computation as appropriate.

2.2 Biological operations

Our fundamental model of computation will be to apply a sequence of operations to a set of strands in a test tube. The operations that we make use of are derived from a collection of experiments commonly used in molecular biology today [1].

2.2.1 Extract

We will need the ability to extract from a test tube all strands that contain any specific short nucleotide sequence. To accomplish this we will use the method of biotin-avidin affinity purification as described in [1]. This technique works in the following way. If we want to extract all strands containing the sequence $\uparrow x$, then we first create many copies of its complementary oligo, namely $\downarrow x$. To these oligos we attach a biotin molecule, which will in turn be anchored to an avidin bead matrix. If we then melt the strands in our test tube and pour them over this matrix, those strands that contain $\uparrow x$ will anneal to the $\downarrow x$ oligos anchored to the matrix. A simple wash procedure will whisk away all strands that did not anneal, leaving behind only those strands that contain $\uparrow x$, which can then be retrieved from the matrix. We refer to this operation as an *extract* using beads of type $\downarrow x$.

2.2.2 Polymerization via DNA Polymerase

Given a particular single strand of DNA, we may wish to create its Watson-Crick complementary strand. To do this we will use the enzyme DNA polymerase. DNA polymerase will "read" the given strand, called the template strand, in the $3' \to 5'$ direction and build the complementary strand in the $5' \to 3'$ direction, one nucleotide at a time. In order to work, DNA polymerase actually requires that there be a short portion of the template that is double stranded, and it is onto the end of this short complementary piece, called the *primer*, that the enzyme will add the new nucleotides. For example, if we have some strand $\uparrow xyz$, DNA polymerase cannot create its complement. However, if we add $\downarrow z$ to the solution and let it anneal to $\uparrow xyz$, we will have $\uparrow xy \updownarrow z$, and DNA polymerase will be able to add nucleotides onto the free 3' end of z to create $\updownarrow xyz$. Note that because DNA polymerase only works in one direction, the partial duplex $\uparrow x \updownarrow y \uparrow z$ will yield $\updownarrow xy \uparrow z$ and not the full duplex $\updownarrow xyz$.

2.2.3 Amplification via PCR

At times we will need to make copies of all the DNA strands in a test tube. This can be done with a straightforward application of the polymerase chain reaction (PCR). PCR is a process that uses DNA polymerase to make many copies of a DNA sequence that exists between two primer sequences. PCR works in the following way. If we have the duplex $\updownarrow xyz$, we first melt it to form $\uparrow xyz$ and $\downarrow xyz$. To this solution we will add the primer oligos $\downarrow z$ and $\uparrow x$, which anneal to form the partial duplexes $\uparrow xy \updownarrow z$ and $\updownarrow x \downarrow yz$. DNA polymerase can then elongate the primers to create full duplexes of the form $\updownarrow xyz$. Note that we now have *two* copies of our original strand. If we just repeat this process, we will again double the number of copies of the original strand in solution. Soon we will have four copies, then eight, then sixteen, and so on, until we have enough copies for our purposes. Thus, if we can guarantee that the primer sequences that we use occur on the ends of every strand, and only on the ends, then we can use PCR to duplicate every strand in the test tube. We will call this operation *amplify*.

3 The DES circuit

In this section we give a brief overview of the DES circuit. DES encrypts a 64 bit plain-text into a 64 bit cipher-text using a 56 bit key. We denote by $\mathrm{DES}(M, k)$ the DES encryption of the plain-text M using the key k.

As will be seen in the next sections, our goal is to run the DES circuit on a fixed 64 bit string M using all possible keys k. In other words, we plan to evaluate the function $f(k) = \mathrm{DES}(M, k)$ for all possible k. The circuit computing the function $f(k)$ is shown in figure 3. The figure displays a convoluted version of the DES circuit since in our case the message to be encrypted is fixed and the key varies. Traditionally the circuit is drawn the other way around, i.e. the key is fixed and the message varies. For a more detailed description of the DES circuit see [6].

We now explain the various components of figure 3. The circuit is composed of 16 levels. We refer to each level as a round. The figure shows the first four rounds and the last round. The input to the circuit is the 56 bit key shown on the left. The high order 32 bits of M are denoted by M_h and the low order bits are denoted by M_l.

A P-box is a box which permutes the bits of its input. Suppose the input to a P-box contains x bits and the output contains y bits. If $x = y$ then the box will only permute the bits of the input, e.g. the box may swap the second and third bits, mapping 010 to 001. If $x > y$ then the box will output a subset of the bits of the input in some order. If $x < y$ then the box will replicate some of the bits of the input to obtain a y bit number. The exact operation performed by each P-box in the figure is predetermined and can be found in [6]. For our purposes we note that the P-boxes are insignificant and may be ignored. The reason for this is that P-boxes simply change the order of the bits that arise during the computation. When going through a P-box there is no need to physically change the order of the bits in the DNA-strand. We can simply keep track in our mind which bits on the strand encode which bits of the computation.

An S-box is more complicated. An S-box takes 48 bits of input and outputs 32 bits. The S-box groups the 48 input bits into 8 groups of 6 bits each. For each group of 6 bits the S-box performs a table lookup and outputs 4 bits. This can be seen in figure 4. The 8 tables used can be found in [6]

To summarize, we see that each DES round involves a 48 bit Xor, an S-box and another 32 bit Xor. Thus, each round requires 80 Xor operations and 8 table lookups.

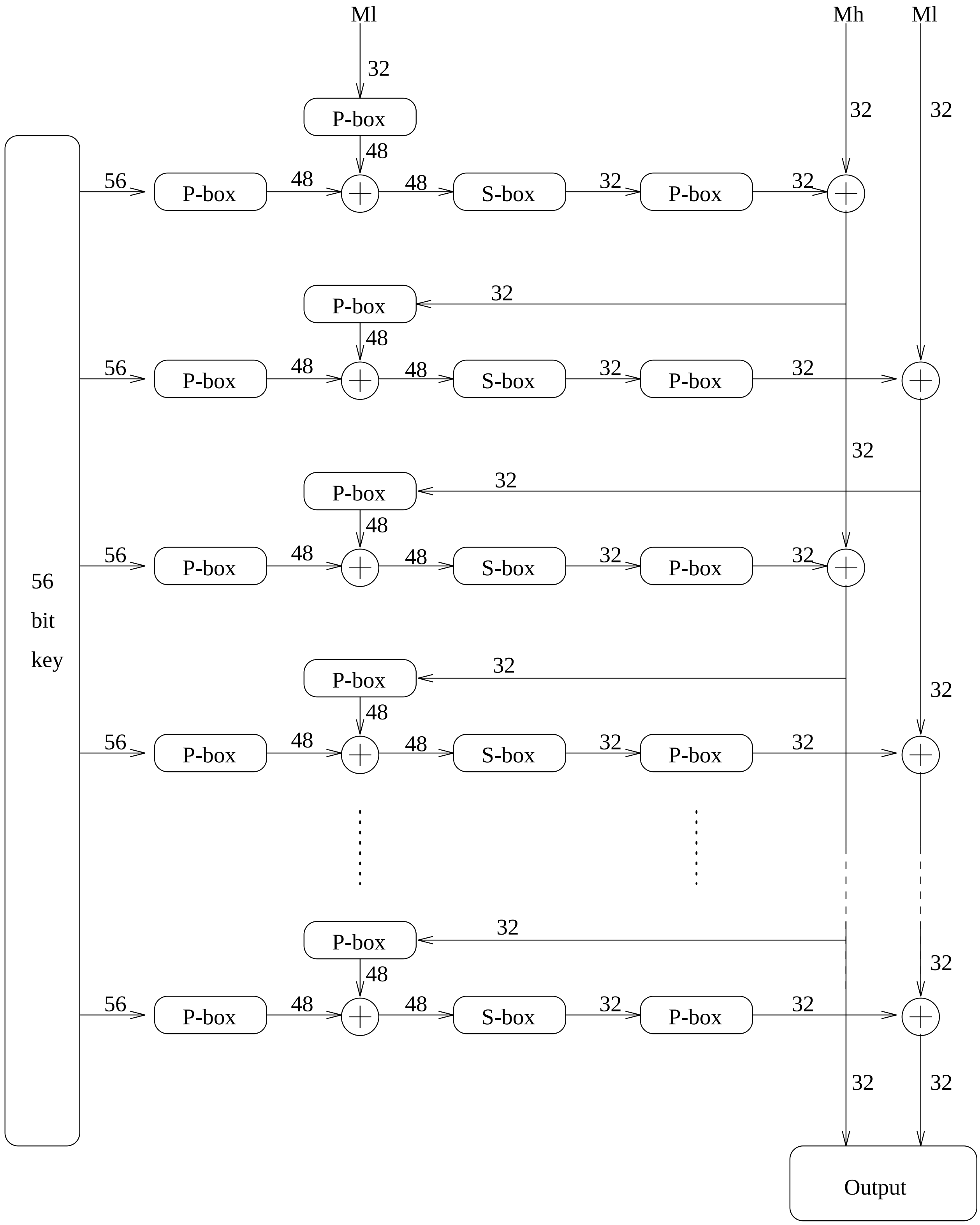

Figure 3: DES circuit

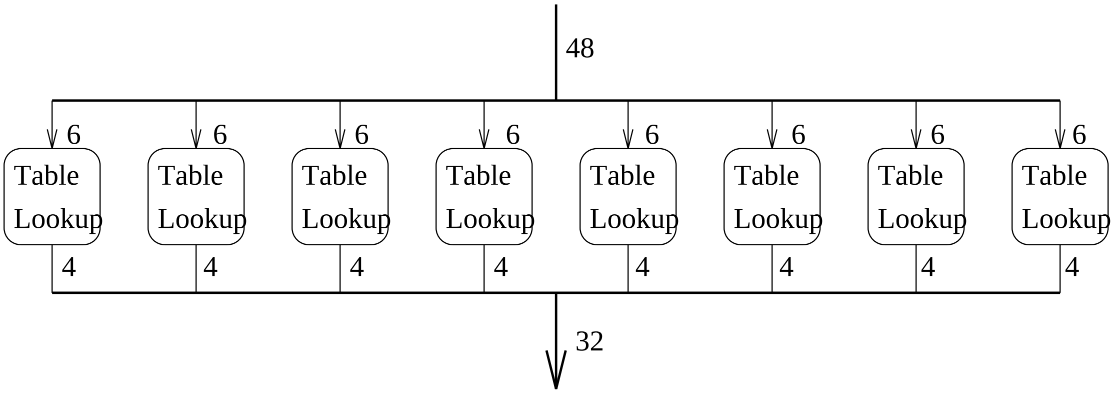

Figure 4: S-box circuit

A useful property of DES is that the decryption procedure is very similar to the encryption procedure. We denote the decryption procedure by DES^{-1}. That is, if $E = \mathrm{DES}(M, k)$ then $M = \mathrm{DES}^{-1}(E, k)$. The only difference between the DES circuit and the DES^{-1} circuit is in the P-boxes. Thus, as far as our biology experiment is concerned the two circuits may be regarded as the same circuit.

4 Representing binary strings

Before we explain how to evaluate the DES circuit we first describe a convenient method for encoding binary strings as DNA strands. The encoding is based on the one described in [8].

Let $x = x_1 \ldots x_n$ be an n-bit binary string. The idea is to assign a unique 30-mer to each bit position and bit value.

1. Let $B_i(0)$ be the 30-mer used to encode the fact that the i-th bit of x is 0. Similarly, let $B_i(1)$ be the 30-mer used to encode the fact that the i-th bit is 1.
2. For $i = 0, \ldots, n$ let S_i be a 30-mer which will be used as a separator between consecutive bits.

The DNA strand representing the binary string $x \in \{0,1\}^n$ will be:

$$\updownarrow S_0\ B_1(x_1)\ S_1\ B_2(x_2)\ S_2\ , \ldots , S_{n-1}\ B_n(x_n)\ S_n$$

There are several points to be made regarding this encoding. First of all, it is crucial that the strings $B_i(x)$ and S_i should all be distinct. Furthermore, it is desirable that any pair of these strings should not contain long common substrings. For example, there shouldn't be a string of length 15 which is a substring of both S_0 and S_1. This can be achieved by choosing these words randomly.

Suppose a solution T contains DNA strands representing a set of n-bit binary strings. By definition, all these strands will start with S_0 and end with S_n. This is convenient for doing PCR, since S_0, S_n can be used as primers.

A potential problem may arise if one strand contains two portions which are Watson-Crick complements of one another. Such a single stranded DNA molecule is likely to anneal to itself, making it useless for our purposes. A simple way of avoiding this problem is to choose the strings $B_i(x)$ and S_i to only contain the bases `A` and `C`. Since `A` and `C` are not Watson-Crick complements, a strand encoding an n bit number will never contain two complementary portions.

Finally, it is worth noting that the strings S_i are not essential to the encoding. It is possible to represent an n-bit number as $\updownarrow B_1(x_1)\ B_2(x_2)\ \ldots B_n(x_n)$, however it is more convenient to encode the strings using the separators.

Notation: For notational convenience, given an n-bit string x, we denote by $R_i(x)$ the string encoding x at position i, i.e.

$$R_i(x) = B_i(x_1)\ S_i\ B_{i+1}(x_2)\ S_{i+1}\ \ldots S_{i+n-2}\ B_{i+n-1}(x_n)\ S_{i+n-1}$$

4.1 Operations on binary strings

We now explain the basic primitive operations that can be performed on binary strings using their encoding as DNA strands. Throughout this section we let T be a test tube containing a collection of DNA strands which represent some binary strings.

As was discussed in section 2, it is possible to extract all the strands in T that contain a certain sequence of basis. This will be the fundamental operation used for evaluating circuits.

Suppose we wish to extract all strands in T representing binary strings whose i'th bit is 1. We know that all such strands contain the string $B_i(1)\ S_i$ and they are the only strands containing this string. Hence, by extracting all strands in T containing the sequence $B_i(1)\ S_i$ we extract exactly those strings whose i'th bit is 1. We denote this operation by

$$\text{Extract}(\ T\ \ ,\ \ x_i = 1)$$

It is possible to perform a more general extract operation. Suppose we wish to extract all strands in T who have a 1 in their i'th position and a 0 in their $i+1$'st position. We denote such an operation by

$$\text{Extract}(\ T\ \ ,\ \ x_i x_{i+1} = 10)$$

Clearly this can be done by extracting all strands containing the string

$$R_i(10) = B_i(1)\ S_i\ B_{i+1}(0)\ S_{i+1}$$

Hence, we were able to perform an "and" operation on consecutive bits for the cost of one extract operation. Note that the fact that the bits are adjacent is crucial for this to work.

An even more general extract operation is the following:

$$\text{Extract}(\ T\ \ ,\ \ x_i x_{i+1} x_{i+2} = 101\ \text{ or }\ x_i x_{i+1} x_{i+2} = 001)$$

This can be done using two types of beads at once. One bead will attract all strands containing the string $R_i(101)$ and the other will attract all strands containing $R_i(001)$. Hence, for the cost of one extract operation we were able to perform a combination of an "and" and an "or". For technical reasons it is only possible to perform this simultaneous "or" operation when the strings being extracted have the same length. For instance, in the above example, both strings 101 and 001 have the same length 3.

Important Note: An important observation is that these operations on DNA strands work no matter where the the bits are on the strand. For instance, the string 01 can be represented in two equivalent ways:

$$\updownarrow S_0\ B_1(0)\ S_1\ B_2(1)\ S_2 \qquad \text{or} \qquad \updownarrow S_0\ B_2(1)\ S_2\ B_1(0)\ S_1$$

All we did was to put the string representing the second bit in the first position. Clearly, the Extract$(T, b_2 = 1)$ operation works exactly as before. The point is that the fact $b_2 = 1$ is encoded by the presence of the string $B_2(1)\ S_2$ on the strand. It makes no difference where this string is on the strand. Finally we note that the string 01 can be represented in a third way as:

$$\updownarrow S_2\ B_2(1)\ S_0\ B_1(0)\ S_1$$

This representation will be useful in Section 6.6.

5 Plan of DES attack

Now that we know how to represent binary strings and how to perform primitive operations on them, we are ready to explain our attack on DES.

In the next section we will show that given a message M it is possible to create a solution that contains for each $k \in \{0,1\}^{56}$ a DNA strand of the form:

$$\updownarrow\ S_0\ \ R_1(\ k\ )\ \ R_{57}(\ \mathrm{DES}(M,k)\ )$$

Each strand in this solution encodes a key k and the encryption of the fixed message M using the key k. We denote this solution by $\mathrm{DES}(M)$. As was stated in Section 3, the decryption circuit DES^{-1} may be regarded as the same circuit as DES. Thus, we can just as easily create a solution $\mathrm{DES}^{-1}(E)$ which is defined analogously to $\mathrm{DES}(M)$, i.e. contains all strands of the form

$$\updownarrow\ S_0\ \ R_1(\ k\ )\ \ R_{57}(\ \mathrm{DES}^{-1}(E,k)\ )$$

Let (M,E) be a (plain-text, cipher-text) pair. We wish to find a key k s.t. $M = \mathrm{DES}^{-1}(E,k)$, i.e. the decryption of the cipher-text E is the plain-text M. This can be done as follows:

1. Create the solution $\mathrm{DES}^{-1}(E)$ described above.
2. Extract from $\mathrm{DES}^{-1}(E)$ all strands that contain the pattern $R_{57}(\ M\ )$.
3. The extracted strands encode pairs of strings (k, M) where $M = \mathrm{DES}^{-1}(E,k)$. The key k can easily be recovered by sequencing any of the extracted DNA strands.

Step 2 and 3 of this process can be done very quickly, i.e. within a few days. The laborious part is step 1.

The same procedure can be used even if the plain-text is only known to be one of a number of candidates. Simply repeat step 2 for each one of the candidates. The running time of the process is hardly effected by this since most of the work is done in step 1. Thus, suppose we are able to intercept one cipher-text and we have an idea of what the encoded message is. Then the above procedure will enable us to recover the key and eavesdrop on the rest of the conversation.

Once we generate the solution $\mathrm{DES}^{-1}(E_0)$ for some fixed 64 bit word E_0, we can very quickly break any DES system if we can use chosen-cipher-text attack. Simply generate a pair (M_0, E_0) and run through steps 2 and 3 above. Clearly the same is true if we can use chosen-plain-text attack. Thus, potentially, a chemical company can spend the time to generate the solution $\mathrm{DES}^{-1}(E_0)$ and then sell it to anyone who wants to quickly break a DES system.

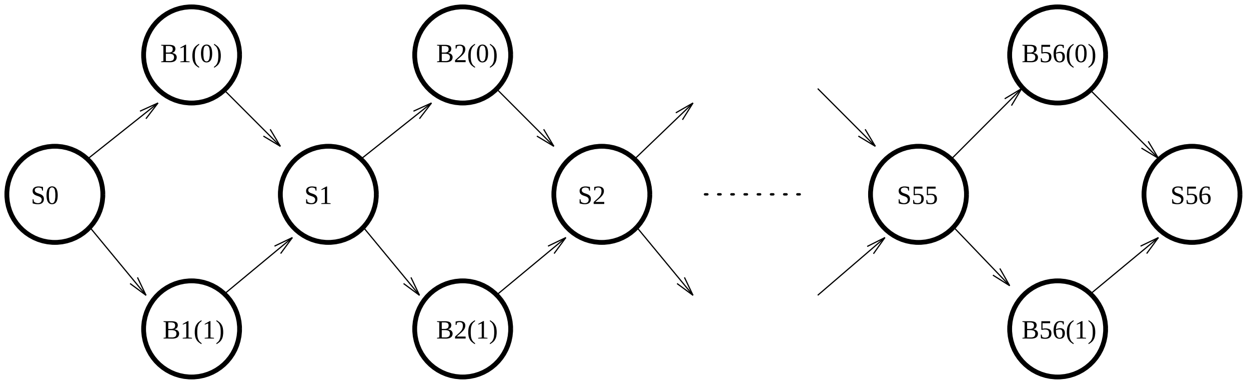

Figure 5: Initialization graph

6 Evaluating the DES circuit

In this section we describe how to generate the solution DES(M). As we saw in the previous section, this solution is the key ingredient in the molecular attack on DES.

We give a quick outline of how to construct this solution and then describe the details. The process is carried out in three steps.

1. Construct an initial solution encoding all 56-bit strings. Thus, the solution contains all strands of the form
$$\updownarrow S_0\ R_1(k)$$
for all $k \in \{0,1\}^{56}$.

2. Evaluate the gates of the DES circuit by tagging on their values. For instance, suppose the first gate in the DES circuit tells us to Xor bit 3 and bit 7 of the key. We do this by appending the value of $k_3 \oplus k_7$ to each of the strands in the initial solution. The resulting solution will contain strands of the form:
$$\updownarrow S_0\ R_1(k)\ B_{57}(k_3 \oplus k_7)\ S_{57}$$
for all $k \in \{0,1\}^{56}$ where k_3 and k_7 are the third and seventh bits of k.

3. We evaluate the gates in the circuit one by one. When we are done, all strands in the solution look like:
$$\updownarrow S_0\ R_1(k)\ \ R_{57}(\ I\ )\ \ R_r(\ \mathrm{DES}(M,k)\ )$$
where I is a string of bits corresponding to the intermediate values generated by the circuit.

6.1 Constructing the initial soup

We begin the detailed discussion by explaining how to construct the initial solution of 2^{56} DNA strands corresponding to each of the possible DES keys. We use a procedure similar to the one described in [8].

The basic idea is to use the graph G shown in figure 5. Each path from S_0 to S_{56} corresponds to one DES key. To create a soup of DNA strands which correspond to all paths in G we use the encoding of [1]. We follow the following steps:

1. Recall that each vertex in the graph represents a string of length 30. Denote the string assigned to vertex V by $V_l V_r$ where V_l is the 15 left letters and V_r are the 15 right letters.

2. For each vertex v create a test tube containing the strands $\uparrow V_l V_r$. For each directed edge (u, v) create a test tube containing the strands $\downarrow U_r V_l$. To handle the end points, create two test tubes containing $\uparrow (S_0)_l$ and $\downarrow (S_{56})_r$.

3. Mix all the test tubes and allow the strands to anneal and ligate. Then extract from the resulting mixture all the strands that contain the words S_0. Call the set of extracted strands A. From the solution A extract all strands that contain the word S_{56}.

It is not difficult to see that at the end of step 2 we get a mixture that corresponds to various paths in the graph. At the end of step 3 the resulting strands will correspond to all paths from S_0 to S_{56}.

6.2 Tagging

Before we explain how the various components of the DES circuit are evaluated we must first explain how to append a given string to all strands in a test tube. This "append to end" primitive can be implemented in various ways. Here we describe one possible method.

Suppose T' is a test tube containing DNA strands which encode various n bit binary strings. Since all the strings have the same length n, all the DNA strands will end with word S_n.

We explain the tagging procedure through an example. Suppose we wish to append the string 01 to all strings in T'. Formally, this means that we append the DNA strand

$$R_{n+1}(\ 01\ ) = B_{n+1}(0)\ S_{n+1}\ B_{n+2}(1)\ S_{n+2}$$

to all strands in T'. We refer to this operation as tagging the strands with 01. To do this, we create a test tube T_{01} containing many copies of the single stranded DNA:

$$\downarrow S_n\ R_{n+1}(\ 01\ )$$

We then pour the contents of T_{01} into T'. Since all the strands in T' end with S_n the new strands will anneal to them. The resulting strands will look like:

$b_1\ b_2\ \ldots\ b_{n-1}\ b_n \quad S_n$
$\qquad\qquad\qquad\quad S_n\ B_{n+1}(0)\ S_{n+1}\ B_{n+2}(1)\ S_{n+2}$

We can now apply a polymerase enzyme to complete this DNA molecule to a double stranded molecule, as explained in Section 2.2.2.

6.3 Evaluating a lookup table

It remains to show how to evaluate the various components of the DES circuits. There are only two types of gates in the DES circuit: an Xor gate and a lookup table (used in the S-boxes). We begin by explaining how to evaluate a lookup table.

A lookup table is a function $f : \{0,1\}^6 \to \{0,1\}^4$ which maps 6 bit strings to 4 bit strings. The function is given as a table of values. For example,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	12	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Columns correspond to the right 4 bits and rows to the left 2 bits. For example, 2 would be mapped to 12. This look up table is one of the eight tables used in the DES S-boxes.

In the DES circuit it is possible to arrange things so that the 6 bits to be looked up are always consecutive bits. This can be arranged by appending the bits in the "right" order.

Say we wish to perform a table look up on the value of the 6 bits in position i to $i+5$. Thus, the strands look like

$$\updownarrow W_l \; B_i(x_i) \; S_i \; B_{i+1}(x_{i+1}) \; \ldots \; B_{i+5}(x_{i+5}) \; S_{i+5} \; W_r$$

where W_l and W_r are the bits on the left and right of the critical 6-bit positions $i, \ldots, i+5$. For a given strand, let $x = x_i \ldots x_{i+5}$ be the bit string encoded on the strand at position i. We wish to append to every strand the 4 bit string $f(x)$ where f is the function computed by the look up table. This is done in two steps:

1. Separate the solution into 16 containers $T_0, \ldots T_{15}$ according to the value to be appended. Formally this entails the following extract operation:

$$T_j \leftarrow \text{Extract}(\ T\ ,\ f(x) = j)$$

2. For each $j = 0, \ldots 15$, append the 4 bit binary string representing j to all strands in container T_j. Formally, we tag all strands in container j by the string $R_n(j)$ where n is the current length of all strands.

This procedure is best explained through an example. Using the above table we see that container T_0 should contain all strands that have one of the 6-bit patterns 15, 16, 31 and 45 in position $i, \ldots, i+5$. We can extract all these DNA strands using the following extract:

$$T_0 \leftarrow \text{Extract}(\ T\ ,\ x = 15 \text{ or } x = 16 \text{ or } x = 31 \text{ or } x = 45)$$

We then tag all strands in T_0 with the string 0000. Note that all 4 bits can be tagged on at once.

Step 1 of this procedure requires 64 different types of beads corresponding to all 6 bit combinations. These beads can be prepared using the same method used to construct the initial soup of all 56 bit strings, i.e. by constructing all paths in a graph similar to Figure 5 which is made up of 6 diamonds. As we will see in Section 6.6 the same set of 64 different beads can be used for all rounds of the DES circuit. Hence, they need only be prepared once at the beginning of the computation. The same method can be used to generate the 16 bit patterns that are appended to the strands in step 2 of the procedure.

To summarize, we have just seen that a look up table operation can be done using 16 extract operations which may be done in parallel followed by 16 parallel tag operations. Hence, if we allow 16 extracts to be done at once, then table look up can be done in time which is approximately equivalent to one extract operation and one tag operation.

6.4 Evaluating Xor gates

Finally, we explain how to evaluate an Xor gate. Suppose we wish to evaluate the Xor of the i'th and j'th bits of the strings in the current solution T. As usual, this is done by appending the value $x_i \oplus x_j$ to the strands representing x. The first step towards performing this operation is to separate the solution T into two solutions T_0, T_1 where T_0 contains those strands in T satisfying $x_i \oplus x_j = 0$ and T_1 contain those satisfying $x_i \oplus x_j = 1$. We then tag each of T_0, T_1 with the appropriate value.

We can write $x_i \oplus x_j$ as

$$x_i \oplus x_j = (x_i + x_j)(\bar{x}_i + \bar{x}_j) \quad ; \quad \overline{x_i \oplus x_j} = (\bar{x}_i + x_j)(x_i + \bar{x}_j)$$

This leads the following procedure for generating T_0, T_1:

1. Set $T^{11} = \text{extract}(T, x_i = 1 \text{ or } x_j = 1)$ and $T^{01} = \text{extract}(T, x_i = 0 \text{ or } x_j = 1)$.
2. Set $T_1 = \text{extract}(T^{11}, x_i = 0 \text{ or } x_j = 0)$ and $T_0 = \text{extract}(T^{01}, x_i = 1 \text{ or } x_j = 0)$.

This procedure requires 4 extract operations. However, if we allow two extract operations to be done in parallel, then the procedure can be carried out in only two steps.

As we saw in Section 4.1, "and" gates are inherently sequential, while an "or" of many variables can be done in one step. This suggests that when writing out a circuit, one should try to minimize the number of "and" gates in the circuit. This is the reason why we wrote the formulas for Xor as a CNF expression. Had we used DNF expressions there would have been 4 and gates (for the Xor and its complement) as opposed to two in the CNF expression. Consequently, evaluating the Xor using CNF expressions would have required 6 extract steps.

Thus, one Xor gate can be evaluated in time equivalent to two extract steps and one tagging step. If we allow more extract steps to be carried out in parallel, then we can evaluate more than one Xor gate at a time. We discuss this improvement next.

6.5 Evaluating three Xors at once

We show that if one can perform 32 extract operations at once, then it is possible to evaluate three Xor gates in two steps. It is possible to arrange things so that the bits to be Xored are x_i, x_{i+1}, x_{i+2} and x_j, x_{j+1}, x_{j+2}, i.e. the value to be appended to the strands is $x_i \oplus x_j, x_{i+1} \oplus x_{j+1}, x_{i+2} \oplus x_{j+2}$. The point is that we are Xoring two groups of three consecutive bits. The fact that the bits to be Xored can be made to be consecutive is a useful property of the DES circuit. It can be shown by examining the precise action of the P-boxes.

As usual, we plan to separate the solution T into 8 solutions according to the value to be tagged. We show how to do this separation in 2 steps. The first step consists of 8 extracts and the second consists of 32 extracts.

For $z_1 z_2 z_3 \in \{0,1\}^3$ define

$$Z(z_1 z_2 z_3) = z_1 \; z_2 \; (z_1 \oplus z_2 \oplus z_3) \in \{0,1\}^3$$

i.e. $Z(z_1 z_2 z_3)$ is a 3 bit string whose third bit is $z_1 \oplus z_2 \oplus z_3$. To separate T into $T_{000}, \ldots, T_{111}$ according to the value to be tagged perform the following steps:

1. For each $z_1 z_2 z_3 \in \{0,1\}^3$ do

$$T_{z_1 z_2 z_3} \leftarrow \text{Extract}(\; T \;,\; x_i x_{i+1} x_{i+2} = Z(z_1 z_2 z_3) \;\text{ or }\; x_j x_{j+1} x_{j+2} = Z(z_1 z_2 0)\;)$$

2. For each $z_1 z_2 \in \{0,1\}^2$ and $a = a_1 a_2 a_3 \in \{0,1\}^3$ do

$$T_{a, z_1 z_2} \leftarrow \text{Extract}(\; T_{z_1 z_2 b} \;,\; x_i x_{i+1} x_{i+2} = a \oplus Z(z_1 z_2 b) \;\text{ or }\; x_j x_{j+1} x_{j+2} = a \oplus Z(z_1 z_2 0)\;)$$

where $b = \overline{a_1 \oplus a_2 \oplus a_3}$.

3. For all $a \in \{0,1\}^3$ do
$$T_a \leftarrow T_{a,00} \bigcup T_{a,01} \bigcup T_{a,10} \bigcup T_{a,11}$$

For each in $a = a_1a_2a_3 \in \{0,1\}^3$ we can now tag all strands in T_a by the bits $a_1a_2a_3$. The reason this extract procedure works follows from the fact that for any $a_1a_2a_3 \in \{0,1\}^3$ we have:

$$\begin{aligned} x_1x_2x_3 \oplus y_1y_2y_2 = a_1a_2a_3 \quad \Longleftrightarrow & \\ \exists z_1z_2 \in \{0,1\}^2 \quad \text{s.t.} \quad & \Big(x_1x_2x_3 = Z(z_1z_2b) \quad \text{or} \quad y_1y_2y_3 = Z(z_1z_20)\Big) \quad \text{and} \\ & \Big(x_1x_2x_3 = a \oplus Z(z_1z_2b) \quad \text{or} \quad y_1y_2y_3 = a \oplus Z(z_1z_20)\Big) \end{aligned}$$

where $b = \overline{a_1 \oplus a_2 \oplus a_3}$.

It is worth noting that there is a simple extraction procedure for evaluating three Xors in two steps if we are allowed to perform 64 extractions per step. Simply separate the solution T into 8 solutions according to the value of $x_ix_{i+1}x_{i+2}$ and then separate each of the resulting 8 solutions into 8 solutions according to the value of $x_jx_{j+1}x_{j+2}$. Using this procedure we perform 8 extractions on the first step and 64 extractions on the second step. This should be contrasted with the performance of the procedure given above which only requires 32 extractions on the second step.

6.6 Shortening the strands

The process of continually appending values to the strands would make the strands very long by the time we complete the DES circuit. It is possible to shorten the strands by throwing away segments which are no longer needed. For instance, this can be done at the end of every round. The only values that have to be kept on the strands is the 56 bit key and the 64 bits that propagated out of the previous round. For a strand that looks like $\updownarrow\ K\ J\ P$ where K is the 56 bit key, J is the junk portion and P are the 64 bits from the previous round that have to be propagated to the next round, we would like to remove that junk portion J.

Removing the middle of a strand is a complicated operation. Fortunately there is a simple fix. As was discussed in Section 4.1, it makes no difference where the string P is on the strands. Hence, when tagging the strands with the bits of P we can place P next to the key K. At the end of the round the strands will look like: $\updownarrow\ P\ K\ J$. We can now use a restriction enzyme to cut the J portion of the strand. At the beginning of the next round all strands look like $\updownarrow\ P\ K$.

This procedure shows that cutting the junk portion of the DNA strands is a simple operation, if the data is arranged with foresight. It involves the use of a restriction enzyme and the filtering of the junk strands out of the solution.

Since the B and S strings used by the junk portion are no longer present in the solution they may be reused. For instance, if the 57'th bit was cut away from all strands then the strings $B_{57}(1), B_{57}(0)$ and S_{57} can be reused in the next round. Specifically, the same set of 64 beads used in the table lookup step can be used in all rounds.

6.7 Summary thus far

Before going any further we briefly calculate the number of steps required to evaluate the DES circuit using the methods sketched thus far. First, observe that the first 48 + 32 + 32 = 112 Xors may be ignored. These Xors are computing an Xor with the message bits. Since the message bits are fixed, these Xors are either a no-op (i.e. no operation performed) or a negation. A negation only requires that we make a mental note remembering that the string representing 1 is now representing 0 and vice versa.

Suppose we allow 32 parallel extractions per step. We saw that it is possible to evaluate 3 Xors in two parallel extract steps and one table lookup per step. Hence, the total number of parallel extractions required is

$$2[15 * (48/3) + 14 * (32/3)] + 128 = 916$$

We can only roughly estimate the amount of lab time required to run this algorithm. Assuming 10 extract operations per day, this experiment would require roughly 4 months of lab work. Clearly many factors have been excluded form this estimate and it should only be regarded as a rough approximation.

7 Speeding computations using Join

We now describe a general method which enables us to perform more steps in parallel. The major bottleneck of the algorithm described in Section 6 is the evaluation of Xors. When the algorithm had to evaluate 32 Xors, it had to evaluate them one by one (or in groups of three). A natural thing to try is to make two copies T_1, T_2 of the solution T. Then evaluate the left 16 Xors in T_1 and the right 16 Xors in T_2 and then combine the solutions T_1 and T_2. Unfortunately there is no simple way of combining the solutions T_1 and T_2 after the Xors have been evaluated.

The problem can be stated as follows. Suppose a strand representing the string x is present in the solution T. We wish to compute two functions $f(x)$ and $g(x)$ and eventually get strands representing the string $x\|f(x)\|g(x)$. By making two copies T_1, T_2 of T we can form two solutions: one containing strings of type $x\|f(x)$ and the other containing strings $x\|g(x)$. The problem is combining these two solutions to form the required answer: $x\|f(x)\|g(x)$. We refer to this operation as a *join*.

For technical reasons we formally define the join operations in the following way: Let T_1 be a solution containing strands of the form $\updownarrow\ x\ y\ \alpha_1$ and T_2 a solution containing strands of the form $\updownarrow\ \alpha_2\ z\ x$, where α_1, α_2 are fixed 30-mers. That is, all strands in T_1 have an α_1 on their 5' end and all strands in T_2 have an α_2 on their 3' end. The join of T_1 and T_2 is defined as

$$\text{join}(T_1, T_2) = \{\updownarrow\ \alpha_1\ z\ x\ y\alpha_2 \ \mid\ zx \in T_1 \text{ and } xy \in T_2\}$$

None of the biology primitives we have discussed so far enable us to carry out a join of two solutions. We first show how to implement the join operation and then discuss how to apply it when evaluating a group of Xors.

Let T_1, T_2 be two solutions as above. The join of T_1 and T_2 may be carried out as follows:

1. Pour T_1 and T_2 into one container and allow the solutions to mix.

2. Raise the temperature in the container so as to melt the hydrogen bonds and form single stranded DNA.

3. Let the container cool down so that complementary strands will reanneal. We get four types of strands:

   (a) $\updownarrow\ x\ y\ \alpha_1$

   (b) $\updownarrow\ \alpha_2\ z\ x$

(c) $\uparrow \alpha_2 \; z \; \updownarrow x \; \downarrow y \; \alpha_1$

(d) $\downarrow \alpha_2 \; z \; \updownarrow x \; \uparrow y \; \alpha_1$

4. Use a polymerase enzyme to complete the sticky ends in the strands. Note that due to the orientation of the strands the polymerase enzyme will only work on strands of the third type.

5. Extract all strands from the resulting solution that contain both α_1 and α_2. This step requires two sequential extracts.

At the present it is not clear whether this procedure works in practice. It is not clear that after we cool the container in step 3 every single stranded DNA will find its mate. The hope is that if initially, the solutions T_1, T_2 contained enough copies of each strand then after cooling, enough strands of type 3 will be formed.

In the spirit of this paper, we will assume for now that the this procedure correctly implements the join operation. We now return to our example of computing two functions $f(x), g(x)$ in parallel. We evaluate the function $f(x)$ in the solution T_1 and $g(x)$ in the solution T_2. For our join operation to work, the strands in T_1 must look like $\updownarrow x \| f(x)$ and the strands in T_2 must look like $\updownarrow g(x) \| x$. Hence, the value of $g(x)$ must be tagged on the left of x as opposed to tagging on the right of x as we have been doing all along. To be more precise, let $x = x_1 \ldots x_n$ be an n bit binary string. Let $f(x) = y_1 \ldots y_r$ and $g(x) = z_1 \ldots z_r$ be r bit strings then the strands in T_1 look like

$$\updownarrow S_0 \; B_1(x_1) \; S_1 \; B_2(x_2) \; S_2 \; \ldots \; B_n(x_n) \; S_n \; B_{n+1}(y_1) \; S_{n+1} \; \ldots \; B_{n+r}(y_r) \; S_{n+r}$$

and the strands in T_2 look like

$$\updownarrow S_{n+2r} \; B_{n+2r}(z_r) \; S_{n+2r-1} \; \ldots \; S_{n+r+1} \; B_{n+r+1}(z_1) \; S_0 \; B_1(x_1) \; S_1 \; B_2(x_2) \; S_2 \; \ldots \; B_n(x_n) \; S_n$$

Note that we treat the bits of $g(x)$ as though they were in position $n + r$ in the string. The two ends S_{n+r} and S_{n+2r} can be used as the strings α_1 and α_2. When we perform the join operation on the two solutions T_1 and T_2 the resulting strands look like:

$$\begin{aligned} &\updownarrow S_{n+2r} \; B_{n+2r}(z_r) \; S_{n+2r-1} \; \ldots \; S_{n+r+1} \; B_{n+r+1}(z_1) \; S_0 \\ &\qquad B_1(x_1) \; S_1 \; B_2(x_2) \; S_2 \; \ldots \; B_n(x_n) \; S_n \; B_{n+1}(y_1) \; S_{n+1} \; \ldots \; B_{n+r}(y_r) \; S_{n+r} \end{aligned}$$

The procedure outlined above is a general method for evaluating two functions in parallel on a molecular computer. For the DES circuit there is a natural way of breaking up the work. Recall that one round of the DES circuit consists of the following steps:

1. 48 Xors.

2. 8 table lookups on disjoint groups of 6 bits.

3. 32 Xors.

At the beginning of each round we may break up the work into 8 groups. Each group will evaluate the Xor of 6 bits, perform a table lookup on the resulting 6 bits and finally perform an Xor on the 4 bits resulting from the lookup. We end up with 8 solutions each representing 4 bits of the value coming out of the current round. To combine the 8 solutions we pair them up and do a join to reduce to 4 solutions. We iterate this twice more until we end up with one solution containing the 32 bits coming out of the current round. The entire process is shown in figure 6.

This method of breaking up the work is appealing since it has a number of advantages.

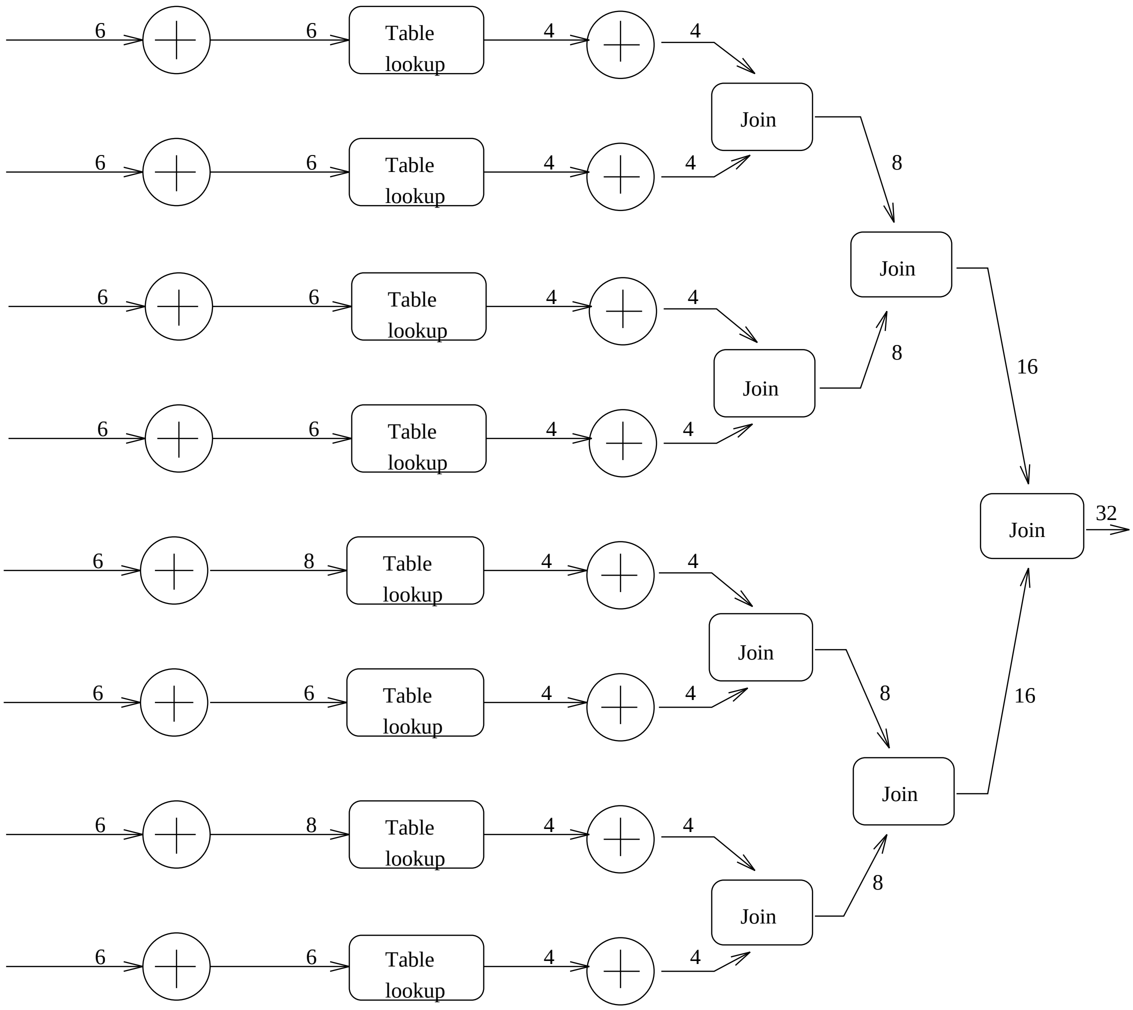

Figure 6: One round of DES circuit

1. Assuming as we did before that it is possible to perform 16 extractions per step, we can evaluate 8 Xor gates at once. This is done by working on the 8 solutions in parallel. Recall that each Xor requires two sequential steps. Thus, we need a total of 292 steps to evaluate the 1168 Xors in the DES circuit.

2. Recall that the table look up operation involved the extraction of strands according to the value of a 6 bit pattern. This step required that we prepare 64 different types of beads. Since now each table lookup is done in a separate solution, we can use the same set of 64 beads for all 8 table lookups in one round. This is a significant reduction in the overhead work for the experiment.

   Note that since each table lookup requires 16 parallel extractions, we can still only perform one table lookup per step. The 128 table lookups in the DES circuit still require 128 steps.

We now need to estimate how many steps it would take to combine the 8 solutions at every round into one solution. There are 3 joins to be made: combine the 8 groups into 4, combine the 4 groups into 2 and combine the 2 groups into 1 (combining the 8 group into 4 requires 4 join operations which can be done in parallel). We saw that each join step requires 2 extractions, i.e. the joins require 6 extraction steps.

Unfortunately there is an additional complication. We explain the problem through an example. Suppose we have 4 solutions T_1, T_2, T_3, T_4 containing strings of type yx, xz, wx, xt respectively. After we do a join on T_1, T_2 and on T_3, T_4 we get two solutions T_{12}, T_{34} containing strings of type yxz and wxt respectively. We would like to do a join on T_{12}, T_{34} to get $zyxtw$, but we can't since the ends of the strings are different(xyz will not join with wxt). This shows that in T_{12} we must copy z to the left to get string of type zyx. Similarly in T_{34} we must copy w to the right to get strings of the form xtw. Doing a join on the resulting solutions will generate the desired $zyxtw$.

This example shows that after we join the 8 groups of solutions to 4 groups we must move some information from one side of the strands to the other. Let T be a solution of strands. Moving 4 bits from on side the strands in T to the other can be done as follows:

1. Separate T into 16 solutions $T_0, \ldots, T_{15}$ according to the 4 bits to be moved. Assuming we are allowed to perform 16 extractions in parallel, this can be done in one step.

2. For all $i = 0, \ldots, 15$ tag T_i with the bit string i on the appropriate side.

3. Set $T \leftarrow T_1 \bigcup \ldots \bigcup T_{15}$ and use a restriction enzyme to cut the out the 4 bits that have just been copied.

We have just seen that copying 4 bits requires one extract step and one tag step. Similarly, copying 8 bits would require 2 extract steps and two tag steps.

After we join the 8 solutions into 4 we must perform 4 move operations to prepare the solutions for the next join. Each move involves copying 4 bits and hence takes one step of 16 parallel extractions. Similarly, after we join the 4 solution into 2 we must make another 2 move operations. Each move this time involves copying 8 bits and hence takes 2 extraction steps. This brings us to a total of $4 \times 1 + 2 \times 2 = 8$ steps for rearranging information on the strands.

To summarize, combining the 8 solutions into 1 requires 6 steps for the join operations and 8 steps for the move operations. Thus we need 14 steps per round where each step consists of 16 parallel extractions. Since there are 16 rounds we get a total of 224 steps.

Evaluating Xors takes 292 steps, table lookup take 128 steps and recombination takes 224 steps. Thus, in 664 steps we can evaluate the DES circuit. This figure should be contrasted with the 916 steps required without using the join operation.

8 Summary

The objective of this paper was to provide a concrete example of how a molecular computer might be programmed. We emphasized the fact that boolean gates are evaluated by tagging on their value. Furthermore we saw how to evaluate Xor gates and lookup tables.

The tagging method is inherently sequential since values have to be tagged on one by one. To overcome this problem we introduced the join operation. The importance of this operation is that it enables us to evaluate functions in parallel and then combine the results. Consequently we saw how this improves our ability to evaluate the DES circuit. It should be pointed out that the join operation has never been experimented with in practice. It is currently unknown whether it actually works.

We now briefly summarize our results regarding breaking DES. We showed that assuming one can perform 32 extractions per step, it is possible to break DES in 916 steps. Furthermore, we saw that by using the join operation it is possible to reduce this figure to 664 steps of 16 parallel extractions. We can only guess that approximately 10 extraction steps per day is a reasonable figure. Under such an assumption we see that one can break DES in 4 months without using joins and in 3 months using joins.

References

[1] L. Adleman, "Molecular Computation of Solutions to Combinatorial Problems", Science 266:1021-1024 (Nov. 11) 1994.

[2] D. Beaver, "A Universal Molecular Computer", Penn State University Tech Report CSE-95-001.

[3] D. Boneh, C. Dunworth, R. Lipton, J. Sgall, "On Computational Power of DNA", to appear.

[4] D. Boneh, C. Dunworth, R. Lipton, "Notation for Molecular Biology", unpublished manuscript.

[5] D. Boneh, R. Lipton, "Making DNA Computers Error Resistant", to appear.

[6] National Bureau of Standards, "Data Encryption Standard", U.S. department of commerce, FIPS, pub. 46, January 1977.

[7] E. Biham, A. Shamir, "Differential Cryptanalysis of the Full 16-round DES", Proceedings of Crypto 1992.

[8] R. Lipton, "Using DNA to solve NP-Complete Problems", Science, to appear.

[9] M. Matsui, "The first experimental cryptanalysis of the Data Encryption Standard", Proceedings Crypto 1994, pp. 1–11.

[10] J. Reif, "Parallel Molecular Computation", To appear, SPAA 95.

[11] W. Smith, "DNA Computers in Vitro and Vivo", unpublished manuscript.

[12] M. Wiener, "Efficient DES Key Search", Crypto 93 rump session.