MAS 335 **Cryptography**

Notes 10: Public-key cryptography **Spring 2008**

In this section we look at two other schemes that have been proposed for public-key ciphers. The first is interesting because it was the earliest such scheme to be invented after Diffie and Hellman came up with the idea of public-key cryptography.

Neither of these schemes is used commercially as far as I know. But we will see later a good reason why it is not wise to rely exclusively on encryption based on number-theoretic problems.

The knapsack cipher

One of the earliest problems to be shown to be NP-complete was the *knapsack problem*. Unofficially, we are given a knapsack with a volume of b units, and items of volume $a_1, a_2, \ldots, a_k$ units. We want to know whether we can fill the knapsack using some of the items.

More formally, the input data for this problem consists of the number b and the list $(a_1, a_2, \ldots, a_k)$ of numbers. Since a number between 2^m and $2^{m+1} - 1$ can be written in base 2 using m bits, we see that the size of a number a when regarded as input data is about $\log_2(a)$, and so the size of the data for this problem is about

$$\log_2(b) + \sum_{i=1}^{k} \log_2(a_i).$$

We are asked to find a k-tuple $(e_1, e_2, \ldots, e_k)$, where each e_i is equal to 0 or 1, such that

$$\sum_{i=1}^{k} e_i a_i = b$$

if possible, or discover that no such tuple exists. This problem is in NP, since we can very easily check a purported solution by simple arithmetic. But finding a solution is harder. In principle, we have 2^k possible k-tuples to check, and if there is no solution we might have to look at all of them. This is not a proof that the problem is hard, since there may be a smarter way to do it; but this problem is indeed known to be hard:

Theorem 13 *The knapsack problem is* NP*-complete.*

Recall that this theorem makes two assertions:

(a) The problem is in NP; that is, we can *check* whether a proposed solution $(e_1, e_2, \ldots, e_k)$ is correct in a polynomial number of steps. (The check is just integer addition!)

(b) If an algorithm to *solve* the problem in a polynomial number of steps were found, then we would know that $\mathsf{P} = \mathsf{NP}$ (which is believed not to be the case).

For example, suppose that we are given the list

$$(323, 412, 33, 389, 544, 297, 360, 486)$$

and a target number 1228. If we try the *greedy algorithm*, which says "at each stage, put the largest item which will fit into the knapsack", we obtain

$$1228 = 544 + 684 = 544 + 486 + 198 = 544 + 486 + 33 + 165,$$

and then we are stuck. So the greedy algorithm fails to solve the problem.

In the end, exhaustive search of some kind reveals that

$$1228 = 412 + 33 + 297 + 486.$$

As can be imagined, a similar problem with 100 numbers of 50 digits each would present quite formidable difficulty.

Now we can make a cipher based on this hard problem as follows. The public key consists of a k-tuple $(a_1, a_2, \ldots, a_k)$ of integers. In order to encrypt a message, we first write it as a string of bits, and break it into blocks of length k. Now the block $(e_1, e_2, \ldots, e_k)$ is encrypted as the integer

$$a = \sum_{i=1}^{k} e_i a_i = b,$$

and this integer is transmitted.

In order to break the cipher it is necessary to solve this instance of the knapsack problem, which is hard! Of course, we also need a secret key so that the intended recipient can decrypt the message.

The way the key is constructed illustrates one important thing about computational complexity, which we haven't stressed so far. For a problem to be easy, it is necessary that there is an algorithm which solves *any* instance efficiently. It may be that some (perhaps just a few) instances are hard; then the problem will be classified as hard, even if most cases are actually easy. In other words, we are measuring ‘worst-case complexity’ rather than ‘average-case complexity’.

Now there are indeed some instances of the knapsack problem which are easy to solve. These correspond to the so-called super-increasing sequences.

The sequence $(a_1, a_2, \ldots, a_k)$ of positive integers is called *super-increasing* if each term is greater than the sum of its predecessors, that is, if

$$\sum_{j=1}^{i-1} a_j < a_i$$

for $i = 1, \ldots, k$. If the data in the knapsack problem is super-increasing, then the greedy algorithm we met earlier, that is, “put into the knapsack the largest object which will fit”, is guaranteed to solve the problem. In other words, let i be the largest index for which $a_i \leq b$; then set $e_i = 1$ and $e_j = 0$ for $j > i$, and (recursively) solve the knapsack problem for the integer $b - a_i$ with the sequence $(a_1, \ldots, a_{i-1})$. The reason for this is that, if the ith item is the largest one which fits in the knapsack, then we must use it; the larger objects don’t fit and, even if all the smaller objects were used, they would not fill the knapsack. (This argument shows a bit more: if a solution exists, then it is unique.)

For example, the sequence $1, 2, 4, 8, \ldots$ of powers of 2 is super-increasing; the above algorithm is exactly what we do when we express an integer in base 2. For example,

$$27 = 16 + 11 = 16 + 8 + 3 = 16 + 8 + 2 + 1,$$

where we take at each step the largest power of 2 not exceeding what we have left.

We cannot just use a super-increasing sequence as public key, since Eve could recognise that it is super-increasing and use the greedy algorithm to decrypt the cipher. So we have to disguise it. This can be done as follows. Bob chooses a super-increasing sequence $(a_1, a_2, \ldots, a_k)$. Then he chooses an integer $n > \sum a_i$ and an integer u with $\gcd(n, u) = 1$, and builds the new sequence $(a_1^*, a_2^*, \ldots, a_k^*)$, where

$$a_i^* = ua_i \bmod n$$

for $i = 1, \ldots, k$. It is very unlikely that these numbers will still be super-increasing, so Bob can use them as the public key.

Now to encipher the binary string $(e_1, \ldots, e_k)$, Alice computes $b^* = \sum e_i a_i^*$, and sends this to Bob. To decrypt this, he calculates the inverse v of u mod n, using Euclid’s Algorithm (as we have seen before). Then he calculates $b = vb^*$ mod n. Now we have

$$b \equiv vb^* \pmod{n}$$

$$\begin{aligned} &= v\sum e_i a_i^* \\ &\equiv v\sum e_i(ua_i) \pmod{n} \\ &= (uv)\sum e_i a_i \\ &\equiv \sum e_i a_i \pmod{n}. \end{aligned}$$

But both b and $\sum e_i a_i$ are smaller than n. (Remember that we chose $n > \sum a_i$.) So, if they are congruent mod n, then they are actually equal:

$$b = \sum e_i a_i.$$

So Bob has only to solve an easy instance of the knapsack problem (with super-increasing data) in order to decrypt the message.

For example, suppose that we take the super-increasing sequence

$$(1,3,7,15,31,63,127,255).$$

Take the modulus 557, which is greater than the sum of the terms in the sequence, and multiply by the coprime inteteger 323 to get the sequence

$$(323,412,33,389,544,297,360,486).$$

Now the bit string 01100101 (character `e` in 8-bit ASCII) is encoded as $412+33+297+486 = 1228$. To decrypt this without solving a 'hard' instance of the knapsack problem, Bob knows that the inverse of 323 mod 557 is 169 (having found that $169\cdot 323 - 98\cdot 557 = 1$); then he calculates $1228\cdot 169$ mod 557, which is 328; and then he applies the greedy algorithm to get

$$328 = 255+73 = 255+63+10 = 255+63+7+3$$

so that the bit string is 01100101 as required.

For added security one can apply the 'disguising' transformation of multiplying by u mod n several times over (with different choices of n and u) before publishing the key.

This was the first practical public-key cryptosystem to be proposed; it was invented by Merkle and Hellman, soon after the basic principles of public-key cryptography had been stated by Diffie and Hellman. It is not actually used today. The problem is that keys obtained by disguising super-increasing sequences in this way are somehow special, and the knapsack problem for such keys turns out to be easier than it is for completely general instances of the knapsack problem. A polynomial time algorithm to break this system was obtained by Shamir in 1982. (Shamir was one of the inventors of the RSA system: so he has a vested interest in breaking any competing systems!)

McEliece's cipher

(Not lectured in 2008)

Another system was proposed by McEliece, based on the theory of error-correcting codes. This is an entirely different topic, which we summarise briefly. Consider the following list of sixteen binary strings of length 7:

$$\begin{matrix}
0 & 0 & 0 & 0 & 0 & 0 & 0\\
1 & 1 & 1 & 0 & 0 & 0 & 0\\
1 & 0 & 0 & 1 & 1 & 0 & 0\\
1 & 0 & 0 & 0 & 0 & 1 & 1\\
0 & 1 & 0 & 1 & 0 & 1 & 0\\
0 & 1 & 0 & 0 & 1 & 0 & 1\\
0 & 0 & 1 & 1 & 0 & 0 & 1\\
0 & 0 & 1 & 0 & 1 & 0 & 0\\
1 & 1 & 1 & 1 & 1 & 1 & 1\\
0 & 0 & 0 & 1 & 1 & 1 & 1\\
0 & 1 & 1 & 0 & 0 & 1 & 1\\
0 & 1 & 1 & 1 & 1 & 0 & 0\\
1 & 0 & 1 & 0 & 1 & 0 & 1\\
1 & 0 & 1 & 1 & 0 & 1 & 0\\
1 & 1 & 0 & 0 & 1 & 1 & 0\\
1 & 1 & 0 & 1 & 0 & 0 & 1
\end{matrix}$$

A little checking shows that any two of these 7-tuples differ in at least three positions. This means that, if one of them is transmitted through a noisy channel which might make a single *error* (that is, change a 0 to a 1 or *vice versa*), the received sequence will still be closer to the transmitted sequence than to any other sequence in the list.

The sixteen 7-tuples have another important property. They consist of all possible linear combinations of four of them (over the integers mod 2); that is, they form a 4-dimensional subspace of the 7-dimensional vector space over $\mathrm{GF}(2)$, the field of integers mod 2. We can take a basis as the rows of the matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1\\ 0 & 1 & 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 0 & 1 & 1 & 0\\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

This provides a very simple way to encode information. If the message to be transmitted is the binary 4-tuple $e = (e_1, e_2, e_3, e_4)$, then we encode it as the 7-tuple

$$eG = e_1a_1 + e_2a_2 + e_3a_3 + e_4a_4$$

using matrix multiplication over $\mathrm{GF}(2)$ (where $a_1,\ldots,a_4$ are the rows of G).

Decoding is more difficult, since (assuming that an error might have occurred) we have in principle to compare the received word to all 16 codewords to see which is nearest.

We can generalise all this. If G is a $k \times n$ matrix over $\mathrm{GF}(2)$ with rank k, then we can encode a binary k-tuple e into an n-tuple eG by matrix multiplication, which is easy. If some errors occur (in a pattern which the code can correct), then to decode we must find the particular one of the 2^k codewords which is nearest to the received word. This looks hard; and indeed it has been shown that the problem of decoding an arbitrary linear code is NP-complete.

However, there are some codes with particular algebraic structure for which efficient decoding algorithms exist. These are widely used in practice; for example, Reed–Solomon codes in CD players, Reed–Muller and Golay codes in space probes.

Our small example gives us an indication of how there can be a 'hard way' and an unexpected 'easy way' to decode. Suppose we are using the 16-word code of length 7 given earlier. The hard way to decode is to compare the received word with each transmitted word to find out which is nearest. For example, if (0111001) is received then we find that the seventh row of the table, (0011001), differs from it in the second position, and must be the transmitted word (assuming at most one error). However, the following *syndrome decoding* method is more straightforward. Let H be the matrix

$$H = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

If the received word is v, we calculate vH, which is a string of three bits. Regard this string as the base 2 representation of an integer m in the range $0\ldots7$. If $m = 0$, then the received word is correct; if $m = 1, \ldots, 7$, there is an error in the mth position.

In our case,

$$(0,1,1,1,0,0,1)H = (0,1,0)$$

and $(0,1,0)$ is the number 2 in base 2, so the second digit is wrong.

You might like to try to explain why this works. This material is covered in the Coding Theory course (MAS309), or in books such as Ray Hill, *A First Course in Coding Theory*.

McEliece's idea is to use the fact that encoding is easy and decoding is difficult as the base of a public-key cipher.

Suppose that Alice wants to send a message to Bob. First, Bob chooses a large code for which an efficient decoding algorithm exists. He also chooses a random permutation and applies it to the columns of the matrix G. The resulting matrix G^* is the public key.

If Alice wants to send the binary k-tuple e to Bob, she first calculates eG^*, and then randomly changes a few of the entries (this corresponds to making some random errors). This is transmitted to Bob.

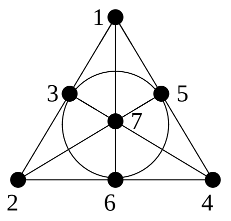
By applying the inverse of his permutation to the cipher, Bob obtains a word encoded using G, which he can decode efficiently (correcting the errors at the same time!) using the decoding algorithm for G.

However, Eve is faced with decoding a word encoded with G^*, which looks like an 'arbitrary' linear code. Without the benefit of the algebraic structure, it is hard to decode.

In terms of the last section of the notes, the encryption function is just matrix multiplication $e \mapsto eG$. Decryption consists of error-correction followed by recovering e from eG. The function g from secret key to public key is applying a permutation to the columns of G; the inverse function involves finding a permutation which converts the 'unknown' code into one for which an efficient decoding algorithm is known.

Any public-key cipher can be attacked in two ways: either try to decrypt directly, or try to reconstruct the private key from the public key. In the case of McEliece's cipher, the latter attack is more likely. We may be able to use the structure of the code in some way.

In our example, some sets of four columns are linearly independent and some are linearly dependent. If we take the set of triples of columns whose complements are linearly dependent, we get a recognisable picture which gives the structure of the code:



Even if the code is presented in arbitrary order, we can build a similar picture and map it onto this one; this will tell us how to rearrange the columns into an order for which our syndrome decoding algorithm will work.