Notes on Cryptography

Charles Blair
Business Administration Dept.
University of Illinois
ceblair@ux1.cso.uiuc.edu

Contents

1 Encryption Systems

An encryption system is a procedure which takes the original message (*plaintext*) and a small piece of information arranged in advance between sender and receiver (the *key*) and creates an encoded version of the message (the *ciphertext*).

When we are considering the quality of an encryption system, we assume the person trying to decode the message knows what the general procedure is and is looking at the ciphertext— the only thing he does not have is the key. We also assume the person sending messages does not spend time trying to contrive a difficult-to-read message by using unusual words or letter frequencies— the sender is counting on the system to provide all the needed security.

Usually one assumes the person trying to break the code is only working with the ciphertext. However, there are situations in which both plaintext and ciphertext of a previously encoded message are available. For example, I often keep encrypted versions of examinations on a mainframe computer, only decoding them just before having them printed, and deleting the plaintext file immediately afterward. If a student was able to look at my files, he could keep a copy of the encoded test and compare this with the test he took. As we will see, this may be very useful in decoding future tests.

[One countermeasure against this type of *known-plaintext attack* is to continually change keys, assuming an encryption using one key is not helpful on a different key. It can become difficult to keep track of the different keys in use, especially if they are long.]

A more demanding standard is that a code may be safe against a *chosen-plaintext attack*. We are imagining that the encryption is done by a machine, and that unauthorized persons may have access to the machine (although we assume they are only using it in the normal way, not allowed to take it apart).

Example 1: Simple substitution

This is the simple letter-for-letter method found in Poe's "The Gold Bug" and many other stories. The key is a rearrangement of the 26 letters:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
actqgwrzdevfbhinsymujxplok
```

Using this key, the plaintext:

```
THE SECURITY OF THE RSA ENCODING SCHEME RELIES ON THE
FACT THAT NOBODY HAS BEEN ABLE TO DISCOVER HOW TO TAKE
CUBE ROOTS MOD N WITHOUT KNOWING NS FACTORS
```

becomes the ciphertext:

```
UZG MGTJYDUO IW UZG YMA GHTIQDHR MTZGBG YGFDGM IH UZG
WATU UZAU HICIQO ZAM CGGH ACFG UI QDMTIXGY ZIP UI UAVG
TJCG YIIUM BIQ H PDUZIJU VHIPDHR HM WATUIYM
```

The messages can be made harder to decode (but also harder to read!) by leaving out the spaces between words.

Most messages can be decoded by looking for frequently occuring pairs of letters (TH and HE are by far the most common), using these to identify a few letters to begin, and filling in the remaining letters one at a time ("The Gold Bug" gives a good description, as do many books).

In a known-plaintext situation, the whole code is obtained almost immediately. However, in our example, the letters `J`, `P`, and others do not occur in the plaintext, so we could not tell how they are encoded. If we were allowed a chosen plaintext, we would use all the letters to get the entire key.

Example 2: The Vigenère cipher and one-time pads

This cipher works by replacing each letter by another letter a specified number of positions further in the alphabet. For example `J` is 5 positions further than `E`. `D` is 5 positions after `Y`. (`Y,Z,A,B,C,D`) The key is a sequence of shift amounts. If the sequence is of length 10, the 1st, 11th, 21st, ...letters of the plaintext are processed using the first member of the key. The second member of the key processes plaintext letters 2, 12, 22, ...and so forth. If we omit spaces from the plaintext on page **??** and use the key-sequence:

$$3\ 1\ 7\ 23\ 10\ 5\ 19\ 14\ 19\ 24$$

we obtain

```
WILPOHNFBR BPMQRJKGTC QDVASSZGVF
HNLOOQBSLM QUOBPFVHMF DUULLTWMAY VCLBXFUZXR
REPPMTOSKF RXALDFDSVS EFYLYYLAHB QXPQRTNHDL
RXPKQSLTTA WPYP
```

(We have divided the ciphertext into groups of ten letters for convenience. The division into lines is arbitrary.)

This type of cipher was considered very secure at one time (~ 1600), but is not really difficult. Suppose we guess that the first letter is `T`. This implies the eleventh letter is `Y`, the 21st letter is `N`, etc. Now look at the two-letter combinations that occur from different possiblilities for the second letter:

```
TI YP ND EN NU AU SC OE OX BF NX OX TP   (no shift of 2nd letter)
TJ YQ NE EO NV AV SD OF OY BG NY OY TQ
TK YR NF EP NW AW SE OG OZ BH NZ OZ TR
TL YS NG EQ NX AX SF OH OA BI NA OA TS
   (skipping over some in the middle)
TF YM NA EK NR AR SZ OB OU BE NU OU TM
TG YN NB EL NS AS SA OC OV BD NV OV TN
TH YO NC EM NT AT SB OD OW BE NW OW TO
```

The last line is the "right answer." Although it shows several bad combinations (NC NT SB NW), mostly caused by the last letter of one word being adjacent to the first letter of the next word, it looks better than the other possible rows. Once the second letter has been identified, the same approach can be used to get the third letter. This approach is easily automated using a table of digrams.

It is necessary to know the first letter and the length of the key-sequence. If we assume the length is not too large, a program can just try all possibilities, eventually choosing the plaintext which looks best.[1]

One-time pads

A long key-sequence makes this approach more difficult, since we have fewer rows. The extreme case is that in which the key-sequence is as long as the plaintext itself. This leads to a theoretically *unbreakable* cipher. For any possible plaintext, there is a key for which the given ciphertext comes from that plaintext.

This type of cipher has reportedly been used by spies, who were furnished with notebooks containing page after page of randomly generated key-sequence. Notice that it is essential that each key-sequence be used only once (hence the name of the system). Otherwise the approach for Vigenère systems described above could be tried, since we would have at least two rows to work with.

One-time pads seem practical in situations where one agent is communicating with a central command. They become less attractive if several agents

[1]Mike Mendelson, a student in this course in 1989, wrote a program to implement this algorithm.

may need to communicate with each other. The one-time feature is lost if X and Y inadvertently use the same page to talk as W and Z are using. Also capture of X's equipment makes it possible to overhear a conversation between Y and Z.

Example 3: A Transposition System

In this system, we will assume every line of the message is 63 characters long. The key is a permutation of the numbers from 1 to 63, and each line of the plaintext is rearranged using the permutation to produce the corresponding ciphertext. For example if the key is

$$1\ 11\ 21 \ldots 61\ 8\ 18 \ldots 54$$

(we would really want to use a more complicated permutation) and we use the same plaintext as in the previous two examples, we obtain:

```
TTRNRT UHOMO SFECE HYSGEH REDEN E NHS E A LE I I   CTCE   O SI
FN  ET AHBCT  DNDO AOBTRA TALOO TY IW CBEO K  SEV H AS  TOE HE
C HNO  OWOA     S  UMOGR  TIWC  RNK   BOU S  STIT  O NF  EDTN
```

We are using the version of the plaintext including blanks. The second line of the plaintext has 55 characters, so we add 8 blanks on the end.

One method of decoding looks at a column of the ciphertext and asks what other column could immediately follow it. For example, it is possible that the column following `OBO` (the tenth ciphertext column) is `UAO` (the 8th), but the column `TFC` would yield the improbable two-letter combination `BF`.

As always, a longer message is easier to decode. Unlike simple substitution, it seems that blanks make the decoding process more difficult.

What about a known-plaintext attack? Since there is only one Y in the first line of the plaintext, we can tell that column 12 of the plaintext is column 21 of the ciphertext, but there are other things we can't tell. In this example, there are 8 columns of three blanks at the end of the plaintext, and we can't be sure which of these corresponds to which of the all-blank ciphertext columns. (it doesn't matter for this message, but we would like to know the entire key to deal with longer plaintexts in the future) A carefully chosen plaintext can give us the entire key at once.

2 Introduction to Number Theory

2.1 Congruences

The congruence $a \equiv b \bmod n$ ("a is congruent to b mod n") says that, when divided by n, a and b have the same remainder.

$$100 \equiv 34 \bmod 11 \qquad -6 \equiv 10 \bmod 8$$

In the second congruence, we are using $-6 = 8(-1) + 2$. We always have $a \equiv b \bmod n$ for some $0 \leq b \leq n-1$, and we are usually concerned with that b. If $a \equiv b \bmod n$ and $c \equiv d$, we can add or multiply

$$a + c \equiv b + d \bmod n \qquad ac \equiv bd \bmod n$$

Division does not always work: $6 \equiv 18 \bmod 12$ but $3 \not\equiv 9$.

2.2 The Greatest Common Divisor

For a and b, the number (a, b) is the largest number which divides a and b evenly.

$$(56, 98) = 14 \qquad (76, 190) = 38$$

Theorem 1 *For any a, b there are integers x, y with $ax + by = (a, b)$*

Proof: The equation can be solved by making a sequence of simplifying substitutions:

$$\begin{aligned} 30x + 69y &= 3 \\ 30x' + 9y &= 3 \quad [x' = x + 2y] \\ 3x' + 9y' &= 3 \quad [y' = y + 3x'] \\ 3x'' + 0y' &= 3 \quad [x'' = x' + 3y'] \end{aligned}$$

It is easy to see that $x'' = 1$, $y' = 0$ is a solution to the final equation and we get a solution to the original equation by working backwards:

$$x' = x'' - 3y' = 1 \quad y = y' - 3x' = -3 \quad x = x' - 2y = 7$$

We could also solve an equation like $30x + 69y = 15$ by multiplying our solution: $y = -15$, $x = 35$. It should be clear that the equation will have no solution in integers if 15 is replaced by something that is not a multiple of $(30, 69) = 3$.

All other integer solutions of $30x+69y = 15$ may be obtained by changing the first solution:

$$y = -15 + \frac{30}{3}t \quad x = 35 - \frac{69}{3}t \quad \text{for } t \text{ integer}$$

If we do the process illustrated on the previous page for any equation $ax+by = (a, b)$, we eventually get one of the coefficients as zero and the other as (a, b). [In fact, this process is usually presented as "Euclid's algorithm for finding the greatest common divisor."]

It is important that this process is feasible [on a computer] even if a and b are several hundred digits long. It is easy to show that the larger of the two coefficients decreases by at least 1/2 every two equations, hence that in twenty equations the larger coefficient has decreased by $2^{-10} < 10^{-3}$, so a 600-digit number would not require more than 4000 equations. [this analysis can be improved]

We pointed out earlier that division does not work with congruences. An important application of Theorem **??** is that it does work for prime numbers.

Corollary 2 *If p is a prime number, $ar \equiv as \bmod p$ and $a \not\equiv 0$, then $r \equiv s$.*

Proof: Since p is a prime, $(a, p) = 1$, so Theorem **??** says there are integer x, y with $ax + py = 1$. Hence

$$ax \equiv 1 \bmod p \quad \text{and} \quad r \equiv (1)r \equiv axr \equiv xar \equiv xas \equiv s \bmod p$$

Corollary 3 *If p is a prime number and $a \not\equiv 0 \bmod p$, then for any b, there is y with $ay \equiv b \bmod p$.*

Proof: We showed in the preceding proof that there is x with $ax \equiv 1 \bmod p$. Let $y = bx$.

Corollary 4 (The "Chinese Remainder Theorem") *If $(p, q) = 1$, then for any a, b, there is an n with*

$$n \equiv a \bmod p \quad \text{and} \quad n \equiv b \bmod q$$

Proof: Theorem **??** implies there are integers x, y such that

$$px + a = qy + b \quad \text{so let } n = px + a$$

2.3 Powers modulo a prime

The sequence

$$a \quad a^2 \quad a^3 \dots \quad \bmod p$$

has many applications in cryptography. Before looking at theoretical properties, the example below (done using a pocket calculator) should make clear that it is practical to compute these numbers, even when many digits are involved.

Suppose we want to compute $432^{678} \bmod 987$. The basic trick is to start with a number and keep squaring:

$$432^2 = 186624 \equiv 81 \quad 432^4 \equiv 81^2 \equiv 639 \quad 432^8 \equiv 639^2 \equiv 690 \dots 432^{512} \equiv 858$$

Since $678 = 512 + 128 + 32 + 4 + 2$,

$$432^{678} \equiv (81)(639)\dots(858) \equiv 204 \qquad \text{(I hope!)}$$

Calculations with exponents involve not-too-many multiplications. If the numbers have several hundred digits, however, it is necessary to design special subroutines to do the multiplications (see Knuth, volume 2).

Let us look at the sequence of powers of 2 mod 11:

$$2\ 4\ 8\ 5\ 10\ 9\ 7\ 3\ 6\ 1$$

Each number from 1 to 10 appears in the sequence.

Theorem 5 *Let p be a prime. There is an a such that for every $1 \leq b \leq p-1$, there is $1 \leq x \leq p-1$ such that $a^x \equiv b \bmod p$.*

It is not always the case that $a = 2$. The powers of 2 mod 7 are 2, 4, 1 after which the sequence repeats and we never get 3, 5, or 6.

The proof of Theorem **??** requires several steps, so we will give it later. For now, we want to look at some of its consequences.

Corollary 6 *Let a be as in Theorem* **??**. *Then $a^{p-1} \equiv 1 \bmod p$.*

Proof: We know that $a^d \equiv 1$ for some $1 \leq d \leq p-1$. If $d < p-1$, the sequence of powers of a would start repeating before we got all the numbers: $a^{d+1} \equiv a$, $a^{d+2} \equiv a^2$, etc.

Corollary 7 *For any* $b \not\equiv 0$*,* $b^{p-1} \equiv 1$ *mod* p*.*

Proof: Let a be as in Theorem ??. Using Corollary ??

$$b^{p-1} \equiv a^{x(p-1)} \equiv \left(a^{p-1}\right)^x \equiv 1$$

Corollary 8 *If* $x \equiv y$ *mod* $(p-1)$*, then* $b^x \equiv b^y$ *mod* p

Proof: For some integer r, $y = r(p-1) + x$ and by Corollary ??

$$b^y \equiv \left(b^{p-1}\right)^r b^x \equiv b^x \text{ mod } p$$

Lemma 9 *Let* $b \not\equiv 0$*,* d *the smallest positive number such that* $b^d \equiv 1$*. Then for any* $e > 0$ *with* $b^e \equiv 1$ d *divides* e *evenly. In particular, by Corollary* ??*,* d *divides* $p-1$ *evenly.*

Proof: If d does not divide e, then $e = dr + s$ for some $0 < s < d$, but

$$b^s \equiv b^e \left(b^d\right)^{-r} \equiv 1$$

would contradict the definition of d.

2.4 Primitive roots

Theorem ?? showed that if p is a prime, there is an a such that the equation

$$a^x \equiv b \text{ mod } p$$

has a solution for any $b \not\equiv 0$. Such an a is called a *primitive root* of p, and x is called the *discrete logarithm* of b.

We showed in the beginning of section ?? that it is easy to obtain b given a and x. Finding x given a and b is much harder. Many modern encryption systems are based on the fact that no efficient way of computing discrete logarithms is known.

No efficient method for always finding primitive roots is known. However, it is often possible to find one in special cases. We will use $p = 1223$ as an example. $p - 1 = 2 \cdot 13 \cdot 47$. By Lemma ??, if a is not a primitive root, then we will either have a^{26}, a^{94}, or $a^{611} \equiv 1$ mod 1223. $a = 2$ and 3 fail, but

$a = 5$ satisfies all three conditions, so it is a primitive root. (we could tell that $a = 4$ would not be a primitive root without testing. Why?)

It is easy to show that, if a is a primitive root, a^x is a primitive root if and only if $(x, p-1) = 1$. In this example, this means the number of primitive roots is

$$1222\left(\frac{1}{2}\right)\left(\frac{12}{13}\right)\left(\frac{46}{47}\right) = 552$$

Thus, if we had just chosen a at random, the probability that it would be a primitive root is $\approx .45$. Choosing a at random and testing until we found a primitive root would not be expected to take too long.

This is an example of a *probabilistic algorithm.* It is possible for it to take a long time, but the amount of time needed on average is reasonably small. We will see many other probabilistic algorithms later.

Proof of Theorem ??

Let $a \not\equiv 0$, d be the smallest positive number for which $a^d \equiv 1$ (there must be such a d since $a^K \equiv a^L$ implies $a^{K-L} \equiv 1$). If $d = p-1$, a is a primitive root. If $d < p-1$, we will find a', d' with $d' > d$. If $d' < p-1$ the process is repeated until we eventually obtain a primitive root.

Lemma 10 *There are at most d solutions to a congruence involving a polynomial of degree d:*

$$x^d + \alpha_1 x^{d-1} + \ldots \alpha_d \equiv 0 \ mod\ p$$

In particular, there are at most d x with $x^d \equiv 1$.

Proof: This can be proved in the same way as the corresponding theorem in ordinary algebra: if $x = \beta$ is a solution, the polynomial can be written as $(x - \beta)$ times a polynomial of degree $d-1$, which by induction has $\leq d-1$ solutions.

We return to the proof of Theorem ??. The sequence

$$a \quad a^2 \quad a^3 \ldots a^d \equiv 1 \text{ mod } p$$

consists of d different solutions of $x^d \equiv 1$. If $d < p-1$, let b be any nonmember of the sequence, with e the smallest positive number with $b^e \equiv 1$.

If $e > d$, we may take $a' = b$, so we will assume $e \leq d$ from now on. By Lemma **??**, $b^d \not\equiv 1$, which implies e does not divide d and $e/(d,e) > 1$.

$$\text{Let} \quad a' = a^{(d,e)}b \qquad c = \frac{d}{(d,e)}$$

To complete the proof, we will show that if $a'^x \equiv 1$, then x is divisible by $ce = de/(d,e) > d$.

Since $(c,e) = 1$, Theorem **??** implies there are K, L with $cK + eL = 1$. If $a'^x \equiv 1$, then $a'^{cx} \equiv b^{cx} \equiv 1$. By Lemma **??**, $cx = eM$ for some integer M and

$$x = (cK + eL)x = e(KM + Lx)$$

so $x = ex'$ for some integer x'. Together with $a'^x \equiv 1$ and Lemma **??**, this implies for some integer N

$$(d,e)ex' = dN \Rightarrow ex' = cN \Rightarrow (cK + eL)x' = c(Kx' + LN)$$

so x' is divisible by c.

3 Encryption techniques based on powers and congruences

3.1 The Diffie-Hellman key exchange procedure

A and B are communicating. C hears everything A and B say. A and B want to agree on a number, without C knowing what the number is. It may be, for example, that A and B plan to use the number as the key for future encoded messages. The procedure (also often called a *protocol*):

A and B agree on a (large) prime p and a primitive root a. These numbers are also known to C. A secretly chooses a (large) number X_1, B secretly chooses X_2. a^{X_1} and a^{X_2} mod p are publicly announced (hence known to C). The secret number will be $S = a^{X_1 X_2}$ mod p.

$$\text{A calulates } S \equiv \left(a^{X_2}\right)^{X_1} \qquad \text{B calculates } S \equiv \left(a^{X_1}\right)^{X_2}$$

A possible drawback to this system is that neither A nor B controls what S is. If S is not a satisfactory number, they may have to repeat the protocol.

Diffie and Hellman suggest the procedure can also be used in a situation in which n people must find, for each pair of people, an agreed-upon number. For $1 \leq i, j \leq n$ the number is $a^{X_i X_j}$.

3.2 The Rivest-Shamir-Adleman public key system

A sets up a system so that anyone can send him an encoded message, but only A will be able to decode it. The message is represented as a number M. The encoding is done by a publicly known function $f(M)$, with A the only person who knows how to compute f^{-1}. A chooses two large primes p, q which he keeps secret. He announces $n = pq$ and another number d, with $(d, p-1) = (d, q-1) = 1$ (one way to do this is to choose d a prime larger than $p/2$ and $q/2$.) The encoding is

$$f(M) \equiv M^d \text{ mod n}$$

where M and $f(M)$ are both $\leq n-1$. We have seen f can be computed in a realistic amount of time even if M, d, n are many digits long.

A computes M from M^d using his knowledge of p, q. By Corollary 8,

$$\text{If } de \equiv 1 \text{ mod } (p-1) \text{ then } \left(M^d\right)^e \equiv 1 \text{ mod } p$$

Similarly $\left(M^d\right)^e \equiv M \text{ mod } q$ if $de \equiv 1 \text{ mod } (q-1)$. e satisfies these two conditions if $ed \equiv 1 \text{ mod } (p-1)(q-1)$. Theorem 1 says we can let $e = x$, where x is a solution of

$$dx + (p-1)(q-1)y = 1$$

Since $\left(M^d\right)^e - M$ is divisible by p and by q, it is divisble by pq, hence we can recover M from M^d by taking to the e-th power mod pq.

It is crucial to the security of this system that knowledge of n does not allow an eavesdropper to calculate p and q. The crude approach of dividing n by all numbers up to $\sqrt{n}$ would take $\sim 10^{50}$ steps for a 100-digit n. However, many famous mathematicians have been unable to devise significantly better factoring algorithms, and this problem has been studied for at least 100 years.

One practical difficulty in using this system is the need to do calculations with many-digit numbers, especially to find primes. Another difficulty is that the inventors of this system have patented it. Amateur programmers who have posted implementations on electronic bulletin boards have received letters from "RSA Security, Inc" warning of possible patent infringement.

3.3 A public key system as hard as factoring

It is possible in theory that there is some way of computing f^{-1} for the system in the previous section that does not involve determining p and q. In the original RSA paper, the authors say

> It may be possible to prove that any general method of breaking our scheme yields an efficient factoring algorithm. This would establish that any way of breaking our scheme must be as difficult as factoring. We have not been able to prove this conjecture, however.

To see the difficulties involved in trying to prove such a thing, suppose that one could show that knowledge of a ciphertext $f(M)$ and a plaintext M enabled one to find p and q. Then one could factor n as follows:

1. Choose any M.
2. Compute $f(M)$. [Remember, we are assuming f is publicly available. Furthermore, $f(M)$ can't be too hard to compute, or the code would be impractical.]
3. Use the assumed method to obtain p, q.

In words, we are unable to distinguish between the situation in which $f(M)$ is obtained from M (easy) and the (presumably difficult) situation in which M is obtained from $f(M)$.

Rabin has suggested an alternative to the RSA system in which there is a direct connection to factoring. As in RSA, $n = pq$ is announced publicly, with primes p, q kept secret. For technical reasons, we assume $p, q \equiv 3 \bmod 4$. The encoding function is

$$f(M) \equiv M^2 \bmod n$$

The way we avoid the difficulty described above is that there are *four* numbers M_1, M_2, M_3, M_4 with $f(M_i) \equiv f(M)$. The key facts are:

1. If p, q are known, it is easy to compute all the M_i given $f(M)$.

2. If we are given n, $f(M)$, and all the M_i, we can calculate p, q.

We are *not* able to obtain p, q from just one of the M_i, so the approach based on M and $f(M)$ described above won't work. One drawback of this system is that, even with knowledge of p and q, one can only say the number sent is one of the four M_i, without being able to identify which one. In practice, this is not serious, since it is very unlikely that more than one of the M_i would correspond to a feasible message.

proof of 1: Since $p \equiv 3 \bmod 4$, there is an integer k with $4k = p+1$. If $x \equiv (f(M))^k \bmod p$, then using Corollary 8:

$$x^2 \equiv \left(\left(M^2\right)^k\right)^2 \equiv M^{4k} \equiv M^2 \bmod p$$

Similarly if $y \equiv (f(M))^L \bmod q$ $[4L = q+1]$, then $y^2 \equiv M^2 \bmod q$. The M_i are given from Corollary 4 by

$$\begin{array}{llll} M_1 \equiv x \bmod p & M_2 \equiv x \bmod p & M_3 \equiv -x \bmod p & M_4 \equiv -x \bmod p \\ M_1 \equiv y \bmod q & M_2 \equiv -y \bmod q & M_3 \equiv y \bmod q & M_4 \equiv -y \bmod q \end{array}$$

proof of 2: If we know the M_i, there will be two like M_1 and M_3 above with $M_1 + M_3 \equiv 0 \bmod p$ and $M_1 + M_3 \not\equiv 0 \bmod q$. Thus we can calculate $(M_1 + M_3, n)$ to obtain p.

Unfortunately, this cryptosystem is vulnerable to a chosen-plaintext attack, even if we assume that the person trying to break the code gets only one of the M_i, chosen randomly. The attacker keeps generating pairs M, $f(M)$ until he gets an M_i with $(M + M_i, n) = p$ or q.

4 Subset-Sum (Knapsack) problems and their uses

4.1 Subset-sum problems are hard

A subset of the numbers

$$267 \quad 493 \quad 869 \quad 961 \quad 1000 \quad 1153 \quad 1246 \quad 1598 \quad 1766 \quad 1922$$

adds up to 5842. Spend a few minutes trying to find such a subset. Whether you succeed or not, I hope you are convinced the task is not trivial.

This is an example of a *subset-sum* problem. In general, we are given n natural numbers a_i and a target number T and asked whether there is a $S \subset \{1, \ldots, n\}$ with

$$\sum_{i \in S} a_i = T \qquad (*)$$

A seemingly simpler problem is the *subset-sum decision* problem. For a given a_i and T, decide whether there is an S for which $(*)$ holds, without being required to identify such an S. However, it can be proved that the decision problem is just as difficult as the subset-sum problem in this sense:

Theorem 11 *Suppose we had a method of solving the subset-sum decision problem. Then we could solve a subset-sum problem using the assumed method n times.*

(the n is not particularly important— the main thing is that the number of uses of the method is not too large.)

Proof: Begin by using the method to decide if T is a sum of the a_i— if not, we can stop immediately. Then use the method to determine if $(*)$ holds for some $S \subset \{2, \ldots, n\}$. If the answer is yes, we ignore a_1 for the rest of the analysis. If the answer is no, we know we must have $1 \in S$. In this second case, we continue by using the method to decide whether there is $S \subset \{3, \ldots, n\}$ with

$$\sum_{i \in S} a_i = T - a_1$$

A yes answer means we can assume $2 \notin S$, otherwise $2 \in S$.

The idea of this proof is more important than the specific result. We show that one problem is as difficult as another by showing that a method of solving the supposedly easier problem can be used to solve another problem. This involves constructing one or several easier problems whose solution answers the hard problem. We saw another example of this idea in our discussion of the Rabin encryption system (section **??**).

Using more elaborate versions of the techniques in Theorem **??**, it can be shown that a method of solving the subset-sum decision problem could be used to solve many other problems, including:

- Factoring
- The Travelling Salesman Problem
- Any Integer Programming Problem
- The Satisfiability Problem

Don't worry if you are not familiar with the details of these problems. The important point is that they are all well-known problems for which many people have been unable to find efficient solution methods, which makes it unlikely that there is a method which solves all subset-sum decision problems efficiently (we will go into more detail on this in section **??**).

The discussion above makes it plausible that some subset-sum problems are difficult. Further, there is some evidence that the "typical" subset-sum problem is not easy. V. Chvatal[2] has shown that if the a_i and T are randomly chosen, then with high probablility (i) there will be no S for which $(*)$ holds (ii) certain simple ways of proving this will not work.

4.2 A proposed public-key system based on subset-sum

As an example of an easy subset sum problem, consider the task of determining what subset of

$$1 \quad 3 \quad 6 \quad 14 \quad 27 \quad 60 \quad 150 \quad 300 \quad 650 \quad 1400$$

adds up to 836. The a_i in this problem have the special property that every number is greater than the sum of all preceding numbers ($27 > 1+3+6+14$, etc). 1400 clearly cannot be in the set. If 650 is not in the set, we would be in trouble, since the sum of the remaining numbers is < 650, hence < 836. Thus 650 must be in the set, and we now have the task of finding numbers which add up to $836 - 650 = 186$. 300 is too big, and the same reasoning as before shows that 150 must be in the set. If we continue, it is easy to identify the desired set as $\{650, 150, 27, 6, 3\}$.

[2] "Hard Knapsack Problems," *Operations Research*, vol. 28, pp 1402–1411

We began with the problem of identifying a subset of

$$267 \quad 493 \quad 869 \quad 961 \quad 1000 \quad 1153 \quad 1246 \quad 1598 \quad 1766 \quad 1922$$

which adds up to 5842. What we didn't mention before was that the a_i were carefully chosen to make them directly related the a_i in the easy subset-sum problem:

$$267 \equiv 300(1000) \bmod 2039 \quad 493 \equiv 27(1000) \bmod 2039 \quad 869 \equiv 60(1000) \ldots$$

where 2039 is a prime chosen in advance (larger than any of the numbers in the easy subset-sum problem) and 1000 is an arbitrarily chosen number.

To find the subset, begin by solving $1000x \equiv 1 \bmod 2039$, which gives $x = 1307$. If we let a_i be the numbers in the easy problem, the hard problem can be written as

$$\sum_{i \in S}(1000a_i) \equiv 5842 \bmod 2039$$

When we multiply by 1307, this becomes

$$\sum a_i \equiv 1307(5842) \equiv 1478$$

It is easy to identify $\{1400, 60, 14, 3, 1\}$ as a subset which adds to 1478, and the desired subset of the original system is

$$\{1246 \equiv 1400(1000) \bmod 2039, 869 \equiv 60(1000), 1766, 961, 1000\}$$

This would seem to give us a good public-key system: a problem which is easy once some special information (the 2039 and the 1000) is known, difficult without the information. Unfortunately, the special type of subset-sum problem created in this way can be solved even without the special information. There is a sequence of papers showing how to solve special subset-sum problems and proposing a refinement which, in turn, was solved by the next paper in the sequence. This has not happened with the RSA system, but there is no guarantee that it won't!

4.3 Other uses of the subset-sum problem

The results mentioned at the end of the last section do *not* contradict the presumed difficulty of subset-sum problems in general. It is only the specially constructed problems which are known to be easy. There are security problems other than public-key codes for which subset-sum problems are useful.

4.3.1 Computer passwords

A computer needs to verify a user's identity before allowing him or her access to an account. The simplest system would have the machine keep a copy of the password in an internal file, and compare it with what the user types. A drawback is that anyone who sees the internal file could later impersonate the user.

I believe this alternative is actually implemented on some systems: the computer generates a large number (say 500) of a_i. They are stored in the internal file. A password is a subset of $\{1, \ldots, 500\}$. (in practice, there is a program to convert a normal sequence-of-symbols password to such a subset.) Instead of having the password for the user, the computer keeps the total associated with the appropriate subset. When the user types in the subset, the computer tests whether the total is correct. It does not keep a record of the subset. Thus impersonation is possible only if somebody can reconstruct the subset knowing the a_i and the total.

4.3.2 Message verification

A sender (S) wants to send messages to a receiver (R). Keeping the message secret is not important. However, R wants to be sure that the message he is receiving is not from an imposter and has not been tampered with. S and R agree on a set of a_i (say 500) and a set of totals T_j (say 200). These numbers may be publicly known, but only S knows which subsets of the a_i correspond to which T_j. The message sent by S is a subset of size 100 of $\{1, \ldots, 200\}$. He does this by sending 100 subsets of the a_i corresponding to the message he wants to send.

5 Subset-Sum Problems and NP-Completeness

The phrase "NP-complete" has an intimidating sound. In this section, we will first define a new problem involving formulas in logic, called the Satisfiability Problem (SP). We will use the abbreviation (SSP) for the subset-sum problem. Our main results will be:

1. If there is an algorithm which efficiently solves (SSP), it can be used to efficiently solve (SP).

2. If there is an algorithm which efficiently solves (SP), it can be used to solve (SSP).
3. An algorithm to solve (SP) efficiently would give efficient solutions to factoring and many other problems.

5.1 The Satisfiability Problem

We will use capital letters A_i, B_i, to stand for logical variables. These stand for statements like "221 is a prime number" or "TH is the most common two-letter sequence in English," which are either true or false, i. e., these variables have values of either **T** or **F**. $\sim A_i$ ("not A_i") is the statement that A_i is false, so it has value **T** if A_i has value **F**, and $\sim A_i$ has value **F** if A_i has value **T**. We will also be interested in more elaborate formulas:

$$A_1 \text{ or } \sim A_2 \text{ or } \sim A_4 \text{ or } A_7 \text{ or } A_8$$

This formula says that either A_1 is true or A_2 is false, or A_4 is false, etc. The value of this formula will be **T** unless $A_1 =$ **F**, $A_2 =$ **T**, $A_4 =$ **T**, $A_7 =$ **F**, $A_8 =$ **F**. Thus, there is only one way in which the formula will be false.

Figure **??** illustrates a satisfiability problem. We want to assign **T**, **F** to

$$\begin{gathered} A_1 \text{ or } A_2 \\ \sim A_1 \text{ or } A_5 \\ \sim A_1 \text{ or } A_3 \text{ or } A_4 \\ A_3 \text{ or } \sim A_5 \\ A_4 \text{ or } A_5 \\ \sim A_3 \text{ or } \sim A_4 \\ \sim A_2 \text{ or } A_3 \end{gathered}$$

Figure 1: A small (SP)

all the variables so that all of the formulas have value **T**. Even in this small example, it may take you a minute or so to find such an assignment.

5.2 Converting (SP) to (SSP)

We want to construct numbers a_i and a target number T so that there is a subset adding up to T if and only if there is an assignment for (SP) which makes all the formulas true. Using this construction allows us to use an algorithm for (SSP) to solve (SP). It implies that solving (SSP) is at least as hard as solving (SP).

We will illustrate the construction using the example in Figure ??. We will have numbers $a_1, \ldots, a_5$ corresponding to the logic variables $A_1, \ldots, A_5$ with $A_i = \mathbf{T}$ corresponding to a_i being included in the subset. We will also need additional a_i, $i > 5$ for technical reasons.

$a_1 =$	1	01	01	00	00	00	00
$a_2 =$	2	00	00	00	00	00	01
$a_3 =$			2	01	00	01	02
$a_4 =$			4	00	01	02	00
$a_5 =$		2	00	02	02	00	00
$a_6 =$	1	00	00	00	00	00	00
$a_7 =$	2	00	00	00	00	00	00
$a_8 =$		1	00	00	00	00	00
$a_9 =$		1	00	00	00	00	00
$a_{10} =$		4	00	00	00	00	00
$a_{11} =$			1	00	00	00	00
$a_{12} =$			2	00	00	00	00
$a_{13} =$			3	00	00	00	00
$a_{14} =$			8	00	00	00	00
$a_{15} =$				1	00	00	00
$a_{16} =$				3	00	00	00
$\ldots =$					$\ldots$		
$T =$	4	04	08	04	04	04	04

Figure 2: Subset-sum problem

The subset-sum problem is shown in Figure ??. For clarity, we have divided the numbers into zones. T will be a sum of a subset of the a_i if and only if the totals within each zone are appropriate. Each zone corresponds to one of the logic formulas. For a_1 through a_5, the value in a zone is 0 if the corresponding A_i does not appear in the logic formula. If A_i does appear, a power of 2 is used. (the reason for using powers of 2 is that different subsets of $\{a_1, \ldots, a_5\}$ will have different totals)

The leftmost zone corresponds to the first logic formula in our example: A_1 or A_2. By making suitable decisions about inclusion of a_6 or a_7, we will be able to get a total of 4 in this zone, unless both a_1 and a_2 are left out of the set, which is precisely what would make the logic formula have value $\mathbf{F}$.

The second zone corresponds to $\sim A_1$ or A_5, which has value $\mathbf{T}$ unless a_1

is in the set and a_5 is not. a_8, a_9, and a_{10} can be used to get the total for the zone equal to 4 in any other case.

Similarly, each of the other zones[3] has a_i associated with it which can be used to obtain the correct total except in one case.

The general problem is that we want numbers which can be used to obtain any total between 1 and 2^n, except for one "forbidden total" M. [In the two zones discussed above M is 4 in one case and 3 in the other] We start with the powers of 2 from 1 to 2^n. If $2^j \leq M < 2^{j+1}$, replace 2^j by the two numbers $M - 2^j$ and $M + 1$. [We did not follow exactly the procedure described in this paragraph in constructing Figure **??**.]

5.3 Converting (SSP) to (SP)

Suppose we have a subset-sum problem with 50 a_i, all between 1 and 2^{20}, with $T < 50(2^{20}) < 2^{26}$. Solving the SSP may be crudely broken into two steps:

1. Decide which a_i are in the subset.

2. Verify that the sum of the chosen a_i is T.

Our (SP) will also carry out these steps. The first is simple: we will have logic variables $A_1, \ldots, A_{50}$ with $A_i = \mathbf{T}$ corresponding to a_i being in the subset. To carry out the second step, we need to construct a set of logic formulas which acts as an "adding machine" to check the total.

We will represent numbers in base 2. Since all relevant numbers are $< 2^{26}$, numbers may be represented by 26 logical variables. For each $1 \leq i \leq 50$, we will have variables $B_{i1}, \ldots, B_{i26}$. These will represent a_i if $A_i = \mathbf{T}$, 0 if $A_i = \mathbf{F}$. To do this, we need formulas which show how the value of A_i determines the value of all the B_{ij}:

$$\sim A_i \text{ or } B_{ij} \quad A_i \text{ or } \sim B_{ij} \quad \text{if } j\text{th digit of } a_i = 1$$

with the simple formula $\sim B_{ij}$ if the jth digit of a_i is 0.

[3] We have omitted the a_i for the last three zones in Figure **??**.

Next, we need, for $2 \le i \le 50$, variables $C_{i1}, \ldots C_{i26}$ which represent the total of the first i of the numbers given by the B-variables. Formulas are needed which show the B-variables determining the C-variables.

We begin with a set of formulas $\mathcal{S}(V, W, X, Y, Z)$ which have Y get value **T** if and only if an odd number of V, W, X have value **T**. Z gets value **T** if and only if 2 or 3 of V, W, X have value **T**:

$$\begin{array}{ll}
V \text{ or } W \text{ or } X \text{ or } \sim Y & \sim V \text{ or } \sim W \text{ or } Z \\
\sim V \text{ or } W \text{ or } X \text{ or } Y & \sim V \text{ or } \sim X \text{ or } Z \\
V \text{ or } \sim W \text{ or } X \text{ or } Y & \sim W \text{ or } \sim X \text{ or } Z \\
V \text{ or } W \text{ or } \sim X \text{ or } Y & V \text{ or } W \text{ or } \sim Z \\
\sim V \text{ or } \sim W \text{ or } X \text{ or } \sim Y & V \text{ or } X \text{ or } \sim Z \\
\sim V \text{ or } W \text{ or } \sim X \text{ or } \sim Y & W \text{ or } X \text{ or } \sim Z \\
V \text{ or } \sim W \text{ or } \sim X \text{ or } \sim Y & \\
\sim V \text{ or } \sim W \text{ or } \sim X \text{ or } Y &
\end{array}$$

If V, W, X represent three one-digit numbers (0 or 1), the formulas $\mathcal{S}(V, W, X, Y, Z)$ have the effect that Y is the number in the column with the three numbers, while Z shows whether there is a number carried into the next column.

We will use $D_{i1},\ldots,D_{i27}$, $2 \le i \le 50$, to keep track of numbers being carried. Since there are no numbers carried in the rightmost column, we have the formulas $\sim D_{i1}$. The formulas

$$\mathcal{S}(B_{1j}, B_{2j}, D_{2j}, C_{2j}, D_{2(j+1)}) \qquad 1 \le j \le 26$$

have the effect of making C_{2j} represent the sum of the numbers B_{1j} and B_{2j}. To have C_{ij} represent the sum of $C_{(i-1)j}$ and B_{ij}, we use

$$\mathcal{S}(B_{ij}, C_{(i-1)j}, D_{ij}, C_{ij}, D_{i(j+1)}) \qquad 3 \le i \le 50 \quad 1 \le j \le 26$$

These logic formulas together have the effect that the A_i determine the B_i, which determine C_{2j} through C_{50j} successively. This last group of variables corresponds to the base-2 representation of the sum of the a_i which are in the set we have chosen. Finally, we add the formulas C_{50j} or $\sim C_{50j}$ depending on whether the jth digit of T is 1 or 0. As planned, a solution to this satisfiability problem gives a solution to the subset-sum problem (look at which A_i have value **T**), which implies that a method of solving (SP) can be used to solve (SSP).

The (SP) we have constructed is rather large, involving approximately 15(26)(50) formulas. However, the rate of growth if we had more a_i with a larger upper limit is not too bad.

5.4 Cook's Theorem

It is more important to understand the general idea of what we did in section ?? than to get involved in the details of the construction of the "adding machine." We used logical variables (the A_i) to represent our guess as to what a solution to the (SSP) was, then constructed a set of formulas to verify the guess was correct.

You should be able to convince yourself that a similar thing could be done with a factoring problem. We could have variables A_i and B_i represent our guesses as to two factors, then construct a "multiplication machine" to verify that the product is what we want. Thus an efficient algorithm for (SP) would lead to an efficient algorithm for factoring[4].

Many other problems can be viewed as making some guesses as to what the correct answer is, with the process of verification relatively easy. An example might be an integer programming problem:

$$\begin{aligned}&\max cx\\ &Ax \leq b\\ &x_i = 0 \text{ or } 1\end{aligned}$$

We ask if there is a feasible solution with objective function value $> K$. For a given x, it is easy to check that it satisfies all the problem constraints and tell if the value is big enough.

The detailed construction in section ?? was intended primarily to convince you that, if a verification can be done efficiently, it can be simulated by a set of logic formulas. It should make you willing to believe

Theorem 12 (Cook) *Any "guess and verify" problem can be converted to a satisfiability problem. Thus, an efficient algorithm for (SP) can efficiently solve any "guess and verify" problem.*

[4]Unlike (SSP), nobody has been able to show that an algorithm for factoring would give an algorithm for (SP).

5.5 Terminology

The concept we have vaguely described as solving efficiently is technically known as "polynomial time." The types of problem that can be considered as "guess and verify" are called NP (for Nondeterministic [the guessing stage] Polynomial [the verification stage]). Cook's Theorem says that (SP) is as hard as any NP problem— the usual terminology is to say (SP) is NP-complete. Since we showed in section **??** that (SSP) could be used to solve (SP), we essentially proved that (SSP) is also NP-complete.

Computers and Intractability, by Garey and Johnson, is strongly recommended for more information.

6 A probabilistic test for primality

Suppose we want to test whether 247 is a prime number. Recall two facts about prime numbers p:

1. $a^{p-1} \equiv 1 \bmod p$ if $a \not\equiv 0$.
2. If $a^2 \equiv 1 \bmod p$, then $a \equiv 1$ or $a \equiv -1$.

Suppose we randomly choose $a = 2$ and test for consistency with these conditions. Since $2^{246} \equiv 220 \bmod 247$ we can conclude immediately that 247 is *not* a prime.

Perhaps we were lucky with $a = 2$. If we try $a = 27$, we get $a^{246} \equiv 1 \bmod 247$. However,

$$27^{246} \equiv \left(27^{123}\right)^2 \text{ and } 27^{123} \equiv 170 \bmod 247$$

which is inconsistent with the second condition, again implying 247 is not a prime.

Not every choice of a is inconsistent with the conditions. For example, $160^{123} \equiv -1$ (hence $160^{246} \equiv 1$) and $178^{123} \equiv 1$. However, the fact that some choices of a give a proof that a number is not prime suggests the following test:

Rabin's Primality Test. Let $p-1=2^k m$, where m is an odd number. Choose a at random. Compute the sequence

$$a^m \quad a^{2m} \quad a^{4m} \ldots a^{p-1} \qquad \bmod p$$

This sequence is consistent with p being a prime if $a^m \equiv 1$ or if the sequence has -1 at some point, followed by 1 for all subsequent terms. In all other cases, a provides a proof that p is not prime (this is usually described by saying that a is a *witness* that p is not prime). Repeat this test for some number of random choices of a, and conclude that p is a prime if none of the chosen a is a witness.

Two features of the test should be emphasized. It does not provide an absolute guarantee that p is a prime, only that it is probably a prime (we will analyze exactly how probable in the next section). Secondly, when we know p is not a prime, we do not know what its factors are— factoring is much more difficult than testing for primality.

[A different probabilistic test is described near the end of the RSA paper.]

6.1 Analysis of the Rabin test

We will calculate how many a are witnesses that 247 is not a prime. Our analysis will make use of the fact that $247 = 13(19)$ and that 2 is a primitive root for both 13 and 19. However, it should be emphasized that this information (which will not be available in general) was not used when we did the test itself.

How many a satisfy $a^{123} \equiv 1 \bmod 247$? We must have $a^{123} \equiv 1 \bmod 13$ and $a^{123} \equiv 1 \bmod 19$. Let $a \equiv 2^x \bmod 13$. Then we must have $123x$ divisible by 12. This gives the possible values $0, 4, 8$ for x, which implies $a \equiv 1, 3$, or 9 mod 13. Similarly, if $a \equiv 2^x \bmod 19$, $123x$ must be divisible by 18, which leads to $a \equiv 1, 7$, or 11 mod 19. (we actually found 178 above by solving $a \equiv 9 \bmod 13$ and $a \equiv 7 \bmod 19$) The 3 choices mod 13 and mod 19 imply there are 9 a with $a^{123} \equiv 1$.

How many a satisfy $a^{123} \equiv -1 \bmod 247$? If $2^{123x} \equiv -1 \bmod 13$, we must have $123x \equiv 6 \bmod 12$, which leads to $a \equiv 4, 7$, or 17 mod 13. Similarly, we get $a \equiv 8, 18$, or 12 mod 19. Thus we get 9 a satisfying this condition.

If we choose $1 \leq a \leq 246$ at random, the chances of getting an a that is not a witness are $18/246 \approx .073$. If we do the test 5 times, the chance of incorrectly concluding 247 is a prime is $\approx 2(10^{-6})$.

[We actually did more work than necessary, identifying the exact set of numbers which would lead to a wrong conclusion. If we only want to count how many numbers there are, we could make use of observations such as that, for any k, an equation $123x \equiv k \bmod 12$ will either have 3 solutions or no solutions.]

Theorem 13 (Rabin) *If p is not a prime, at least $3/4$ of $1 \leq a \leq p-1$ are witnesses.*

This implies that for any non-prime p, the chance of being incorrectly identified after 5 tests is $\leq 4^{-5} < .001$.

7 Probabilistic Encryption

[References to "the paper" in this section are to "Probabilistic Encryption," in *Journal of Computer & System Sciences* 28, pp. 270–299. I have also used *Primality and Cryptography*, by E. Kranakis]

So far, the public key systems have been functions f such that the message M presumably cannot be computed from the encoding $f(M)$. A further concern arises as to whether, even if the adversary cannot identify M exactly, he may be able to obtain some partial information about M, for example tell whether M is an even number, a square, a power of 2, etc.

An extreme case of this would be a scenario in which the adversary knows the message is one of two possibilities, M_1 or M_2. Since we have been assuming that the function f is easy to calculate, all the adversary needs to do is compare $f(M_1)$ and $f(M_2)$ with the ciphertext.

Probabilistic encryption is a system designed to avoid these problems. Instead of $f(M)$ being a single number, the calculation of $f(M)$ involves the sender doing some things randomly during the calculation, so that M has many different encryptions. Indeed, the probability should be very close to 1 that if the same message is sent twice, the encryptions should be different.

7.1 The Goldwasser-Micali encryption system

As in many previously discussed systems, the person receiving messages chooses two primes (~ 100 digits) p, q and announces $n = pq$. This system is concerned with whether, for a given number a, there is x with $x^2 \equiv a \bmod n$. Such a are called *squares* or (in most books and papers) *quadratic residues.* For technical reasons, when we refer to squares mod n, we will exclude a which are divisible by p or q. The following facts are easy to prove, in some cases using primitive roots.

Lemma 14 *If a, b are squares, then ab is a square. If a is a square and b is not a square, then ab is not a square.*

Lemma 15 *a is a square mod n if and only if it is a square mod p and a square mod q.*

Lemma 16 *Let $h = \frac{p-1}{2}$. If a is a square mod p, $a^h \equiv 1$ mod p. If a is not a square, $a^h \equiv -1$.*

This implies that, if p and q are known, it is easy to decide whether a is a square. The encryption system depends on the assumption (called QRA in the paper [p. 294]) that this problem is very difficult if p, q are unknown.

Lemma 17 $1/2$ *of the numbers from 1 to $p - 1$ are squares mod p. Take the numbers from 1 to n and leave out those divisible by p or by q. Divide the remaining $(p-1)(q-1)$ numbers into four groups according to whether they are squares or not mod p and also mod q. There are $(p-1)(q-1)/4$ numbers in each group.*

The numbers which are not squares mod p and also not squares mod q are called *pseudo-squares.* Example: If $p = 5$, $q = 7$, the squares mod 35 are 1, 4, 9, 16, 29, 11 ($29 \equiv 8^2$, $11 \equiv 9^2$; note we don't include 25 and 14, because they're divisible by p, q). The pseudo-squares must be congruent to 2 or 3 mod 5 and to 3, 5, or 6 mod 7. Thus the pseudo-squares are 17, 12, 27, 3, 33, 13.

The encryption system is primarily concerned with the union of the set of squares and pseudo-squares— this set is unfortunately denoted both by Z_n^1 (p. 291) and by Z_n^{+1}. Since exactly half the members of Z_n^1 are squares, the crude idea of saying "this is a square" all the time will only be right half the time. (QRA) says that no algorithm that runs in a reasonable amount of time can do much better than this. [the precise definitions of "reasonable" and "much better" are what require the concepts of circuits of size k and "ϵ-approximating"]

In addition to announcing n, the person receiving messages announces one pseudo-square y. To send a sequence of 0's and 1's, the sender converts them into numbers as follows: for each number in the sequence, an x is chosen *at random*. 0 is converted into x^2 mod n, 1 is converted into yx^2. Each 0 or 1 in the sequence can be converted (depending on the choice of x) into one of $(p-1)(q-1)/4$ different numbers. If the message is of length 500 (about one line of ordinary text), and $p, q \approx 10^{100}$, the message can be encoded into $(1/4)10^{100000}$ different possible ciphertexts.

By Lemma **??**, 0's are converted to squares, 1's are converted to pseudo-squares. Since the receiver knows p, q, Lemmas **??** and **??** show he can efficiently decode the message.

In the subsequent sections, we will give the essential ideas of Goldwasser & Micali's proof that (assuming QRA) this system will prevent the adversary from obtaining any partial information about the plaintext.

7.2 Weak laws of large numbers

Both the encryption algorithm and the hypothetical algorithms used by the adversary involve random events. We will need a theorem that says that, if an event with probability p is tried r times, the chance that the number of successes is not close to pr is small. The paper uses[5] (p. 293)

Lemma 18 *Let S_r be the number of successes in r tries. For any ψ*

$$\Pr\left(\left|\frac{S_r}{r} - p\right| > \psi\right) < \frac{1}{4r\psi^2}$$

[5]The usual central limit theorem cannot be used because it does not tell you how large r must be for the normal distribution to give a good estimate.

Proof: S_r is a random variable, which is the sum of r independent random variables, each having value 0 or 1. Let V be the variance of S_r. Each of the 0–1 variables has variance $\leq 1/4$, so

$$r^2\psi^2 \Pr(|S_r - rp| > r\psi) < V \leq \frac{r}{4}$$

Lemma **??** provides a very rough estimate of the probability. An improvement requiring much more work is:

Lemma 19 *With the same notation as Lemma* **??**,

$$\begin{aligned}
&\Pr\left(\frac{S_r}{r} \geq p + \psi\right) \\
&\leq \frac{1}{\sqrt{2\pi r(p+\psi)(1-p-\psi)}}\left(\frac{(1-p)(p+\psi)}{\psi}\right)\exp\left(-\frac{r\psi^2(1+\psi)}{2(1-p)(p+\psi)}\right) \qquad (*)
\end{aligned}$$

For comparison, if $p = .5$, $r = 1000$, the probability that there are ≥ 520 successes is .1087. Lemma **??** gives[6] an upper limit of .3125, while Lemma **??** gives .1498. (these figures courtesy of Mathematica)

One reason the paper does not use Lemma **??** is that it does not give a simple formula for how large r would have to be in terms of the other quantities. We will not use this result later, and you should skip to section **??** unless you like to manipulate formulas.

Proof: We will assume $pr + r\psi$ is integer. ¿From the binomial theorem:

$$\begin{aligned}
\Pr(S_r \geq rp + r\psi) &= \sum_{i \geq pr + r\psi} \binom{r}{i} p^i (1-p)^{r-i} \\
&\leq \binom{r}{pr + r\psi} p^{pr+r\psi}(1-p)^{r-pr-r\psi}(1 + \alpha + \alpha^2 + \ldots) \\
&\qquad \text{where } \alpha = \frac{p(r - pr - r\psi)}{(1-p)(pr + r\psi + 1)}
\end{aligned}$$

$p + \psi \leq 1$ implies $p - p\psi - p^2 > 0$ and

$$\sum \alpha^i = \frac{1}{1-\alpha} = \frac{(1-p)(pr + r\psi + 1)}{r\psi + 1 - p} \leq \frac{(1-p)(p+\psi)}{\psi}$$

[6] We divide by 2 to eliminate the probability of ≤ 480.

which gives the second factor of $(*)$. We use Stirling's formula on the binomial coefficient and group it with the powers of p and $1-p$ to obtain:

$$\left(\frac{1}{\sqrt{2\pi r(p+\psi)(1-p-\psi)}}\right)\left(\frac{p}{p+\psi}\right)^{pr+r\psi}\left(\frac{1-p}{1-p-\psi}\right)^{r-pr-\psi r} \qquad (**)$$

The first factor of $(**)$ is the first factor of $(*)$. We obtain upper bounds on the rest of $(**)$, using

$$-A-\frac{A^2}{2(1-A)} \leq \ln(1-A) \leq -A-\frac{A^2}{2}$$

(the lower bound on $\ln(1-A)$ involves a geometric series)

$$\begin{aligned}
(pr+r\psi)\ln\left(1-\frac{\psi}{p+\psi}\right) &\leq -r\psi-\frac{r\psi^2}{2(p+\psi)} \\
(pr+\psi r-r)\ln\left(1-\frac{\psi}{1-p}\right) &\leq \frac{(r-pr-r\psi)\psi}{1-p}+\frac{(r-pr-r\psi)\psi^2(1-p)}{2(1-p)^2(1-p-\psi)} \\
&= r\psi-\frac{\psi^2 r}{1-p}\left(-1+\frac{1}{2}\right)
\end{aligned}$$

Adding these and using exp gives the remaining factor of $(*)$.

7.3 The magic of sampling

We have 10^6 envelopes. Inside each envelope is a piece of paper with 0 or 1 written on it. If we want to know exactly how many envelopes have each number, we have to open them all. Suppose we want to estimate the fraction of the envelopes of each kind, and we want the proportion to be accurate to within .05. Now we need only open $9(10^5)$ envelopes.

The situation changes dramatically if we only want to estimate the proportion with high probability. If we are willing to accept a .01 probability of an error $> .05$, Lemma **??** implies we only need to open a randomly chosen sample of 10^4 envelopes[7].

The special feature of problems involving squares and pseudo-squares is that sampling is possible. We saw in our discussion of the Rabin system that

[7] Lemma **??** and Mathmatica suggest 400 envelopes are enough.

every number mod n has four square roots. Thus if we choose one of the $(p-1)(q-1)$ numbers x not divisible[8] by p or q and compute x^2 mod n, each square has a $(p-1)(q-1)/4$ chance of being chosen. It is also important that it is possible to sample from Z_n^1 (the set of squares and pseudo-squares) even if p, q are not known.

Lemma 20 *There is an efficient algorithm for deciding if $a \in Z_n^1$.*

The proof of this is difficult, involving "quadratic reciprocity" and the "Jacobi symbol." The algorithm itself is not that complicated, and is given in the RSA paper.

Given this lemma, we can sample in Z_n^1 by choosing x at random and testing if it is in the set. If not, another x is chosen. Since roughly half of $1 \leq x \leq n$ is in Z_n^1, this won't take too long.

The different sampling possibilities we have discussed so far have all assumed that only n was known. If we are given a single pseudo-square y, we can sample among all pseudo-squares by calculating yx^2 for x randomly chosen.

The possibility of doing these various kinds of sampling is closely related to properties 2(a) and (c) in the paper (p. 277).

7.4 Determining algorithm performance by sampling

We are interested in algorithms for deciding whether a given number is or is not a square. As with the algorithm in Section ??, there is some probability that, for a given input a, the algorithm may give the wrong answer.

Let p_a be the probability that a given algorithm gives the correct answer for input a. We are also interested in p_S, which is the average of p_a over all squares a, and p_{PS}, the average over all pseudo-squares, and p_Z, the average of p_a over all $a \in Z_n^1$.

If we are given an algorithm, we can easily determine p_S by running it with input $a = x^2$ on a sample of randomly chosen x and counting the number of times the algorithm answers "this is a square."

[8] Even though p, q are unknown, the gcd of x, n can be computed.

The procedure for determining p_Z is more elaborate. Suppose we have an algorithm for which $p_S = .6$. Using Lemma **??**, generate a sample of 100 members of Z_n^1, and run the algorithm on each of them. Suppose we get the answer "this is a square" 65 times. There are ~ 50 squares in the sample, on which there have been .6(50) correct responses and 20 incorrect. Pseudo-squares have been identified as squares $65 - 30 = 35$ times, which suggests $p_{PS} \approx 15/50$. Finally $p_Z = (p_S + p_{PS})/2 \approx .45$.

Lemma **??** or **??** can be used to determine the probability that these estimates come within a specified amount.

7.5 Two versions of QRA

1. There is no efficient algorithm for distinguishing squares from pseudo-squares with $p_a > 1 - \epsilon$ for all $a \in Z_n^1$.

2. There is no efficient algorithm with $p_Z > .5 + \epsilon$

It would seem that (1) is not as strong as (2). Note that (2) would rule out an algorithm with $p_S = .9$ and $p_{PS} = .2$. This would be something that says "this is a square" most of the time, occasionally correctly identifying a pseudo-square. However, the paper (p. 293) shows that (1) implies (2).

Suppose we are given an algorithm. We estimate p_S, p_{PS}, p_Z with high probability using the techniques in Section **??**. To take a specific example, we will assume we find $p_S = .6$, $p_{PS} = .45$. We want to test whether a is a square. Run the algorithm on ax^2 for 1000 randomly chosen x. If a is a square, the algorithm will say "this is a square" ≈ 600 times. If a is a pseudo-square, the answer will be "this is a square" ≈ 550 times.

7.6 Knowing a pseudo-square does not help much

QRA talks about the ability to identify squares when only n is known. In the proposed encryption system, a pseudo-square y is also announced. The paper shows (p. 295) that this does not make the problem easier.

Suppose we have an algorithm which takes as input a, y and tries to decide if a is a square. Assume $p_Z = .55$ whenever y is a pseudo-square. Choose $y \in Z_n^1$ at random, then use the techniques from Section **??** to estimate p_Z.

Since half the numbers in Z_n^1 are pseudo-squares, you will quickly find a y for which $p_Z = .55$.

7.7 The inability to distinguish two plaintexts

Theorem 5.1 of the paper addresses the issue we mentioned at the beginning of section **??**. It shows that if we have an algorithm which can identify messages m_1 and m_2 and efficiently tell the difference between an encryption of m_1 and an encryption of m_2, then we could construct an algorithm which efficiently distinguishes squares from pseudo-squares. Thus (QRA) implies we cannot tell the difference between m_1 and m_2.

Proof:[9] Suppose we are trying to decide whether $a \in Z_n^1$ is a square and that the two distinguishable messages are

$$\begin{aligned} m_1 &= 01001011 \\ m_2 &= 11101101 \end{aligned}$$

Choose 8 x_i randomly and consider the sequences

$$\begin{array}{cccccccc} x_1^2 & ax_2^2 & x_3^2 & x_4^2 & ax_5^2 & x_6^2 & ax_7^2 & ax_8^2 \\ ax_1^2 & ax_2^2 & ax_3^2 & x_4^2 & ax_5^2 & ax_6^2 & x_7^2 & ax_8^2 \end{array}$$

If a is a pseudo-square, these will be randomly chosen encodings of m_1 and m_2. In this case, the performance of our assumed algorithm on the two sequences (averaged over repeated random choices of x_i) will be different. If a is a square, both sequences will be randomly chosen encodings of the message consisting of all 0's, so the algorithm's response on average to the two sequences will be identical.

7.8 Semantic Security

Theorem 5.2 of the paper shows that there is no property of the plaintext message which can be efficiently estimated by looking at the ciphertext. Typical properties might be "the last bit of the plaintext is 0" or "the number of

[9] The argument we give is a simplification of the one in the paper, in that we do not use the "sampling walk." The more complicated argument seems to be necessary to analyze encryption systems in general, as opposed to those based on squares and pseudo-squares.

1's is twice as much as the number of 0's." In general, a property is defined in the paper as the value of a function $f(m)$ which takes a message as input and gives a number as output. If $f(m)$ is constant for all m, prediction of $f(m)$ is trivial. Similarly, if $f(m)$ is almost constant for almost all m, there is a simple algorithm which will be close to right with high prob

We wish to show that, except in the special cases we've mentioned, there is no efficient algorithm which will predict $f(m)$ from the ciphertext for m. If there were, we could run our algorithm to estimate $f(m)$ on the ciphertext from randomly generated m until we found m_1, m_2 on which the algorithm behaved differently. But this would contradict the result of the previous section.

[The paper points out that it is not assumed that $f(m)$ is an easily computable function. I think this is a minor issue. The theorem really discusses the capabilities of a an easily computable program for estimating f.]

7.9 How to play poker over the telephone

We will not analyze an entire game of poker, but just the task of each player [we will assume only two players] getting dealt cards so that (i) each player gets his cards at random, with all cards equally likely (ii) neither player knows what his opponent has (iii) the players cannot get the same cards. You will probably appreciate the procedure more if you first try to devise a way of doing this yourself.

Several previous attempts to use cryptographic devices for this purpose were flawed[10]. The elaborate procedure we describe is based on some number-theory tools developed in section **??** and earlier in this section:

1. If $n = pq$ and a is a square mod n, it has four square roots. If we know roots r_1, r_2 with $r_1 \not\equiv \pm r_2$, we can find p, q.

2. If $p \equiv 3 \bmod 4$, a is a square mod p if and only if $-a$ is not a square (Lemma **??**). If we also have $q \equiv 3 \bmod 4$, then $a \in Z_n^1$ if and only if $-a \in Z_n^1$.

[10] R. Lipton, "How to cheat at mental poker," *Proceedings of AMS Short Course on Cryptography*

3. We can test whether or not $a \in Z_n^1$ without knowing p, q.

Two techniques are used repeatedly. They are also of interest in other applications.

Theorem 21 (random numbers) *B can generate a random number so that A does not know its value now, but can verify it later.*

A "first try" might be for B to generate a random number and give an encryption of it to A, with the key revealed for verification later. This does not work, since A cannot be sure that B chose his number at random.

To insure randomness, A gives B a second number (which A is supposed to choose at random) after receiving B's encryption, and the number used by B is the "exclusive or" of the two:

$$\begin{array}{r} \text{A chooses } 0110001 \\ \text{B chooses } 1011011 \\ \hline \text{B uses } 1101010 \end{array}$$

Even if one of the players does not choose his number at random, the result will be random as long as the other player does.

Theorem 22 *B can ask A a question related to n. The answer to this question may or may not allow B to factor n. At the time the question is asked, A cannot tell whether the answer he gives B is useful or useless, but this can be verified later.*

Proof: A chooses primes $p, q \equiv 3 \bmod 4$, and announces $n = pq$. Using the technique of Theorem **??**, B generates a random x, and will ask A for a square root of $a \equiv x^2$. At the time the question is asked, A will know a but not x. B is allowed to specify whether the square root A gives him is or is not in Z_n^1.

If $x \in Z_n^1$ and B specifies that the square root is in Z_n^1, A will give B $\pm x$, which is useless. B can get useful information by specifying that the square root is not in Z_n^1. If $x \notin Z_n^1$, the square root in Z_n^1 will be useful, and the other will be useless.

Since x is randomly chosen, and half the possible x are in Z_n^1 and half are not, A will not be able to guess right more than half the time whether he is being asked for useful or useless information.

The procedure

1. A announces $n_1, \dots n_{52}$, each of which is a product of two large primes $\equiv 3 \bmod 4$. He encodes the names of the different cards using different n_i and also announces these. [if B finds the factors of one of the n_i, it does not help him identify the other cards] B does the same thing using $m_1, \dots m_{52}$.

2. To get a card, B asks A one question for each n_i, using the procedure of Theorem **??**. 51 of the questions will be useless. The useful question allows B to decode the name of the card he receives. [it is crucial that A will be able to verify the uselessness of the other 51 questions after the game.]

3. B deletes the m_i corresponding to the card he received (this ensures A will not get this card).

4. A gets a card by asking 51 questions about the remaining m_i, of which 50 are useless. He deletes the n_i corresponding to this card.

5. If B gets a second card, he asks 51 questions. He avoids getting the same card twice by not asking a useful question about the same n_i as the first time.

This procedure is too cumbersome to be practical, but it is a good example of the kinds of things that can be done using cryptographic procedures. Current research focusses on other tasks involving exchanges of encrypted and partially encrypted information between two players.

8 Pseudo-random number generators

[This section is based on Blum, Blum, & Shub, "A simple unpredictable pseudo-random number generator," *SIAM J. Computing* 15, 364–383.]

Many programs (e. g., simulations, one-time-pads) make use of numbers that are supposed to be random. A genuine source of randomness might be a subroutine that made calls on something like a built-in Geiger counter. We will be concerned with algorithms that produce a sequence of numbers (usually 0's and 1's) which appears random (precise definition will be given later).

A typical example of such an algorithm is the function `rand()` in the C programming language. Each call updates an internally maintained N using the formula

$$N = N * 1103515245 + 12345 \mod 4294967296 = 2^{32}$$

with the output given by $2^{-16}N \bmod 2^{15}$.

I recently wrote a program to roll dice which involved using `rand()` mod 6. In over 100 calls, it never happened that the same number occurred on two consecutive rolls, even though this should have happened about 1/6(100) times! This suggests this particular generator has some problems.[11]

In this section, we will present random number generators for which it can be proved (given assumptions like (QRA)) that such problems will not occur.

8.1 The Quadratic Generator

Let $n = pq$, where p, q are primes $\equiv 3 \bmod 4$. For each prime, a is a square if and only if $-a$ is not a square (Lemma ??). This implies that, if $x \equiv \pm a_1 \bmod p$ and $x \equiv \pm a_2 \bmod q$, there will be exactly one choice which makes x a square mod n. Hence, if b is a square mod n, exactly one of its four square roots will also be a square. This *principal* square root will be denoted by $\sqrt{b}$.

The quadratic generator uses a randomly chosen square x (called the *seed*) not divisible by p or q to generate a sequence of 0's and 1's (*bits*). The sequence is a_i mod 2, where $a_0 = x$ and $a_{i+1} \equiv \sqrt{a_i} \bmod n$:

$$x \bmod 2 \quad \sqrt{x} \bmod 2 \quad \sqrt{\sqrt{x}} \bmod 2 \quad \ldots$$

(from a practical point of view, it is simpler to generate the sequence starting with the last number and squaring)

[11] Knuth suggests that a better way to obtain a random number between 0 and $k-1$ is to use k `rand()`$/M$, where M is the maximum value of `rand()`.

As a small example with $n = 589 = 19(31)$ and $x = 81$, the sequence of a_i is

$$81 \quad 9 \quad 586 \quad 175 \quad 112 \quad 443 \quad 214 \quad 237\ldots$$

(note that $\sqrt{9} = -3$, not 3) which gives the sequence of bits 11010101.

8.2 The Next Bit Theorem

It would certainly be undesirable if there were an efficient algorithm which took as input the first k bits of the sequence from the generator and guessed the $(k+1)$-st bit with probability much greater than $1/2$. We say a generator satisfies the *Next Bit Condition* if there is no such algorithm.

Theorem 23 *If (QRA) is true, the quadratic generator satisfies the Next Bit Condition.*

Proof: We will show that an algorithm that could predict the $(k+1)$-st bit could be used to distinguish squares from pseudo-squares mod n.

Let $b \in Z_n^1$. The sequence of length k

$$b^{2^k} \quad b^{2^{k-1}} \ldots b^4 \quad b^2$$

can be considered as coming from the quadratic generator with seed the first term of the sequence. If we take this sequence mod 2 and give it to our predictor, we would get a guess as to whether

$$\sqrt{b^2} \equiv 0 \text{ or } 1 \bmod 2$$

which has probability $> 1/2$ of being right. The principle square root of b^2 is b if b is a square, $n - b$ if b is a pseudo-square. Since $b \not\equiv n - b \bmod 2$, the information from the predictor gives us a guess as to whether b is a square.

8.3 The Efficient Test Theorem

When we are given a sequence of bits from a pseudo-random number generator, we often test the quality of the generator by doing things like counting the fraction of 0's, the fraction of subsequences of the form 111, etc.

A *test* is defined to be an efficiently computable function T which takes as input a sequence of bits of length m and gives as output a number between 0 and 1. Define

$$\begin{aligned} A_r &= \text{Average over all sequences } s \ \{T(s)\} \\ A_g &= \text{Average over } s \text{ from the generator } \{T(s)\} \end{aligned}$$

These averages both involve finite operations— A_r involves adding up $T(s)$ over the 2^m possible s and dividing. Similarly A_g deals with an average over all possible seeds (presumably the number of possible seeds is much less than 2^m).

It would take too much time to calculate A_r, A_g exactly, but they can be estimated with high probability using the sampling ideas in section **??**.

A generator is said to *satisfy* the test T if A_g is close to A_r, i. e., T cannot tell the difference between sequences from the generator and genuinely random sequences. [we are being deliberately vague about the precise definition of "close."]

Theorem 24 *If a generator satisfies the Next Bit Condition, it satisfies all efficiently computable tests T.*

Proof: We will show that, if we had T with A_r significantly different from A_g, then for some k, T could be used to predict the $(k+1)$-st bit from the first k bits with probability somewhat larger than 1/2. This would contradict the Next Bit Condition.

If s is a sequence of i bits, let f_s be the fraction of all possible seeds whose first i bits are s. For some s, we may have $f_s = 0$. Note that

$$A_g = \sum_s f_s T(s)$$

where the sum is over all s of length m.

The proof involves two steps:

1. Identify a $0 \leq k \leq m-1$ such that the behavior of $T(s)$ depends in a significant way on the $(k+1)$-st bit of s.

2. Use T to make a prediction for the $(k+1)$-st bit.

The proof of step 1 uses ideas similar to the "sampling walk" used to prove Theorem 5.1 in the Goldwasser-Micali paper. Define

$$A_i = \sum_{s,t} f_s 2^{i-m} T(s \circ t)$$

where the sum is over all s of length i and t of length $m - i$, with $\circ$ meaning to combine s and t to create a sequence of length m. A_i is the expected value of T applied to a sequence in which the first i bits come from the generator (using a randomly chosen seed), with the remaining bits coming from a genuinely random source.

Note that $A_0 = A_r$, $A_m = A_g$, and that all A_i can be estimated with high probability using sampling. Since

$$|A_r - A_g| \leq \sum_1^m |A_i - A_{i-1}| \text{ there is } k \text{ with } |A_{k+1} - A_k| \geq |A_r - A_g|/m \quad (1)$$

This completes step 1.[12]

In step 2, we are concentrating on a specific sequence s of length k, where k satisfies (**??**). We wish to use the behavior of T to predict whether the $(k+1)$-st bit should be 0 or 1. Intuitively, we ask T which of the two possibilities would make the sequence look more random.

We will need to look at the analogues of the averages A_k and A_{k+1}, restricting attention to those sequences which begin with s:

$$\begin{aligned} A_k(s) &= \sum_t 2^{k-m} T(s \circ t) \\ A_{k+1}(s) &= \sum_t (f_{s\circ 0}/f_s) 2^{k+1-m} T(s \circ 0 \circ t) + \\ &\quad \sum_t (f_{s\circ 1}/f_s) 2^{k+1-m} T(s \circ 1 \circ t) \end{aligned}$$

The definition of $A_{k+1}(s)$ is based on the idea that $s \circ 0$ and $s \circ 1$ are the only sequences of length $k+1$ which begin with s. Note that, for $i = k$ or $k+1$,

[12] Instead of estimating all the A_i, we could begin by estimating $A_{.5m}$. We would next estimate either $A_{.75m}$ or $A_{.25m}$, depending on whether $A_{.5m}$ was closer to A_0 or A_m.

$A_i = \sum_s f_s A_i(s)$, where the sum is taken over all s of length k.

$$\begin{aligned} \text{Define} \quad A_{s,0} &= \sum_t 2^{k+1-m} T(s \circ 0 \circ t) \\ A_{s,1} &= \sum_t 2^{k+1-m} T(s \circ 1 \circ t) \end{aligned}$$

These are the expected values of T for a sequence which begins with s, has either 0 or 1 as its $(k+1)$-st term, and continues randomly. They can be estimated by sampling. Let p_s be the fraction of the seeds which give s as the first k bits which give 0 as the $(k+1)-st$ bit (thus $p_s = f_{s\circ 0}/f_s$). Then

$$A_k(s) = \frac{1}{2}A_{s,0} + \frac{1}{2}A_{s,1} \tag{2}$$

$$A_{k+1}(s) = p_s A_{s,0} + (1-p_s)A_{s,1} \tag{3}$$

If we could estimate p_s from (**??**), it would be simple to predict the next generated bit after s. Unfortunately, we cannot efficiently estimate $A_{k+1}(s)$. The problem is that we would have to sample among the seeds which generate s, and there is no easy way to find such seeds. Instead, we must find a way to use the information that the average of $A_{k+1}(s)$ is A_{k+1}, which we can estimate.

The (far from obvious) idea will be to have the prediction of the $(k+1)$-st bit itself be random. As we will see below, the probabilities can be assigned to the two possible predictions can be chosen so that the expected number of correct guesses looks like the right-hand side of (**??**).

We will assume $A_{k+1} > A_k$ [remember, we chose k so that the difference between the two is significant]. The other case can be handled similarly. If $A_{s,0} > A_k(s) > A_{s,1}$, we would expect sequences beginning with $s \circ 0$ to look more like things from the generator than sequences beginning with $s \circ 1$. Our prediction for the next bit following s will be random, given by

$$\text{Predict} \begin{cases} 0 \text{ with probability } \frac{1}{2} + A_{s,0} - A_k(s) \\ 1 \text{ with probability } \frac{1}{2} + A_{s,1} - A_k(s) \end{cases}$$

[The probabilities add to 1 by equation (**??**).]

The probability that the prediction for input s is correct is

$$\begin{aligned} p_s\left(\frac{1}{2}+A_{s,0}-A_k(s)\right)+(1-p_s)\left(\frac{1}{2}+A_{s,1}-A_k(s)\right) &= \\ \frac{1}{2}+p_sA_{s,0}+(1-p_s)A_{s,1}-A_k(s) &= \frac{1}{2}+A_{k+1}(s)-A_k(s) \end{aligned}$$

When we average over all seeds resulting in all possible s, we get a correct prediction with probability $1/2+A_{k+1}-A_k$, which, by (**??**), is significantly greater than 1/2. [13]

8.3.1 A consequence involving symmetry

The Next Bit Condition was stated in a way that clearly distinguished the beginning of the pseudo-random sequence from the end. By contrast, the Efficient Test Theorem treats a pseudo-random sequence in a completely symmetrical way. From that point of view, it does not matter which end of the sequence is used to start the construction. This leads to

Corollary 25 *Let $n=pq$. Start with a random $1\le x\le n-1$ not divisible by p or q. Let $a_0=x$, $a_{i+1}\equiv a_i^2 \bmod n$. The sequence of bits given by a_i mod 2 satisfies all efficient tests.*

[13]Thanks to R. Sengupta for pointing out the importance of the expression for $A_{k+1}(s)-A_k(s)$.