INFORMATION SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

SECRECY, AUTHENTICATION, AND PUBLIC KEY SYSTEMS

By

Ralph Charles Merkle

June 1979

Technical Report No. 1979-1

This work was supported by the National Science Foundation under grant ENG-10173; the U.S. Air Force Office of Scientific Research under contract F49620-78-C-0086; and the U.S. Army Research Office under contract DAAG29-78-C-0036.

SEL 79-017

SECRECY, AUTHENTICATION, AND PUBLIC KEY SYSTEMS

By

Ralph Charles Merkle

June 1979

Technical Report No. 1979-1

This work was supported by the National Science Foundation under grant ENG-10173; the U.S. Air Force Office of Scientific Research under contract F49620-78-C-0086; and the U.S. Army Research Office under contract DAAG29-78-C-0036.

ACKNOWLEDGEMENTS

It is the author's great pleasure to acknowledge the aid, the help, the assistance, and the support of: my fellow graduate students, Steve Pohlig, Raynold Kahn, Dov Andleman, and Justin Reyneri; Bob Fabry and Jim Reeds on the faculty at U.C. Berkeley; the independent and imaginative Whit Diffie; the ever helpful Charlotte Coe; my fellow Berkeley students Peter Blatman, Bruce Englar, Frank Olken, and Loren Kohnfelder; the love and support of Carol Shaw; and my mother, who knew I could do it all along.

I would like to give my special thanks to Martin Hellman, whose support made it possible and who encouraged me when it counted most: when little was known and much was doubted.

Thanks are also due to the National Science Foundation for it's support of the work described in chapter's III, IV, VI, VII, VIII, IX and X under grant ENG-10173; and to the U.S. Air Force Office of Scientific Research and the U.S. Army Research Office for their support of the work described in chapters II, V and VII under contracts F49620-78-C-0086 and DAAG29-78-C-0036.

Table of Contents

APPENDICES

I. INTRODUCTION

1. Introduction

Cryptography is a fascinating subject, even more so today than in the past. The new and once unthinkable ideas of public key distribution and digital signatures have opened up new fields of research, and new possibilities for the marketplace. To be one of the first to venture into this virgin territory has been a great privilege.

This thesis presents the findings of work done between fall of 1974 and spring of 1979.

Chapters III and IV describing the "puzzle" methods are now primarily of pedagogical and historical interest: they were the first break in what at that time appeared to be a smooth and solid wall.

Chapter VI, on the trapdoor knapsack, describes the second real breakthrough, (the first was the key distribution method based on exponentiation developed by Hellman) and represents work done in the summer of 1976.

Chapter V, on a certified digital signature, was conceived in the summer of 1977. To some extent it represents frustration over the difficulties of extracting signatures in a clean and reliable way from the trapdoor knapsack.

Chapter VII is a follow up on chapter VI. It attempts to provide reasons for believing that the trapdoor knapsack is actually secure. The author is, of course, quite convinced it is secure; but inventors have traditionally been blind to the weaknesses and obvious faults of their cryptographic inventions. It has therefore been very encouraging to hear from others that they have failed miserably in attempts to analyze it.

Chapter IX on protocols provides insight into the problems and techniques of actually using public key systems.

Chapter VIII describes an essentially NP-complete conventional cryptosystem, a theoretical result which might provide a useful avenue for further research aimed at getting proofs of security.

Chapter X describes a cryptanalytic method for breaking an apparent improvement which has been suggested to the DES. Its description of potential weakness in a particular scheme for multiple encipherment carries with it a simple moral: simple extensions or modifications to a cryptographic algorithm can have unexpected weaknesses.

2. Conventional Cryptography

Conventional cryptographic systems provide secrecy and authentication to information which may be overheard or modified by unauthorized third parties. This is done by encrypting (or enciphering) the plaintext P with a key K to produce the ciphertext C: $S_K(P) = C$, where S_K denotes the enciphering function under key K. Only authorized users know K, and so only they can decipher C by computing $P = S_K^{-1}(C)$. Although unauthorized users know C and the set of functions $\{S_K\}$, this does not allow them either to determine P or to modify C to produce a C' which deciphers to a meaningful message.

The security of such systems resides entirely in the key K. All other components of the system are assumed to be public knowledge. To maintain security, the legitimate users of the system must learn K, while preventing others from learning it. To date, this has been done by sending K to the legitimate users of the system over special physically secure communication channels, e.g., registered mail or couriers. The flow of information in a conventional cryptographic system is shown in figure 1.

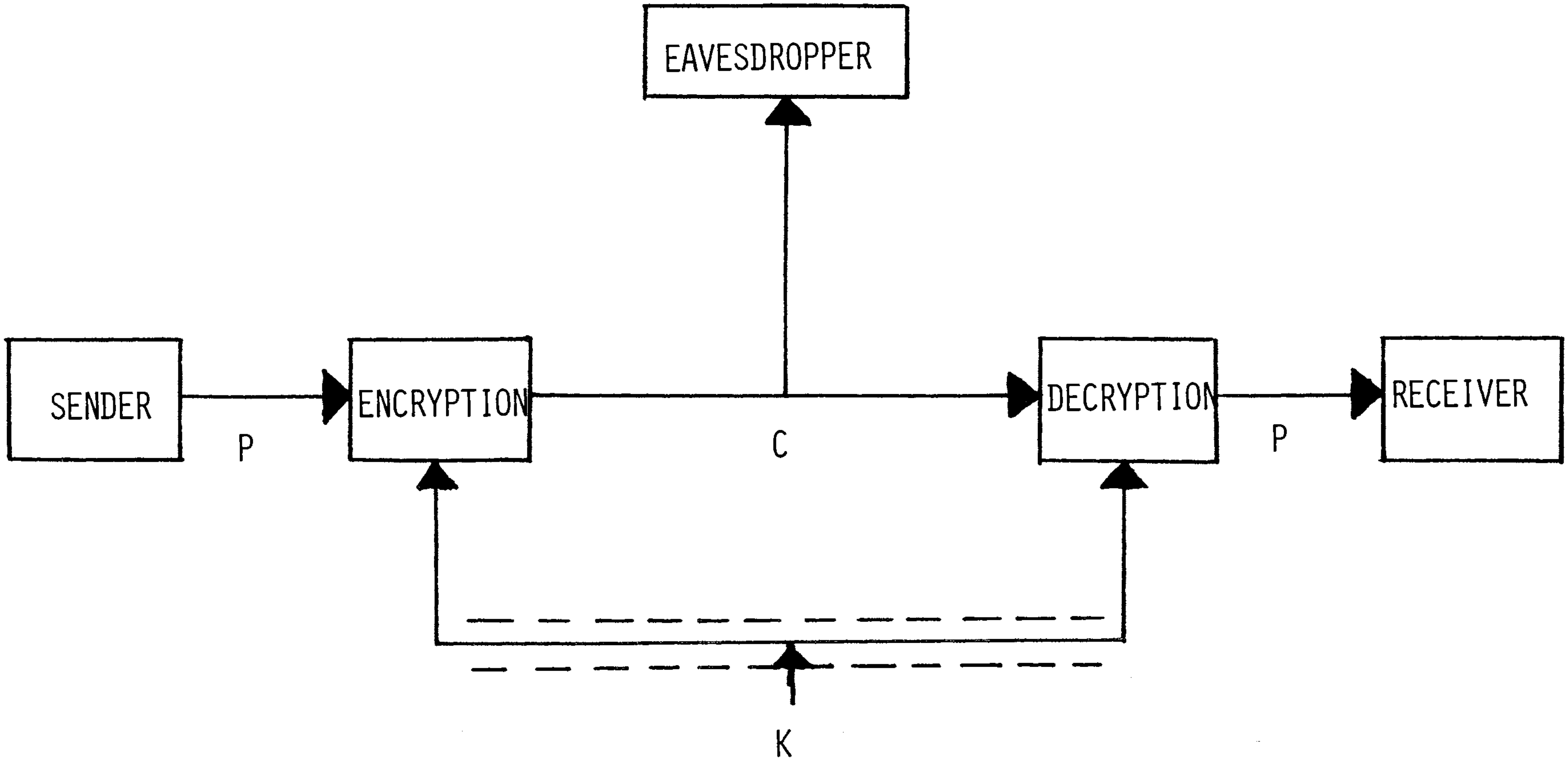

FIG 1

THE FLOW OF INFORMATION IN
A CRYPTOGRAPHIC PRIVACY SYSTEM.

3. Public Key Systems

The reader interested in public key cryptography is referred to [4] for an excellent tutorial overview. So that this thesis is self contained, two sections from that paper are reproduced below with only minor changes to introduce the concepts of public key systems and digital signatures.

The difficulty of distributing keys has been one of the major limitations on the use of conventional cryptographic technology. In order for the sender and receiver to make use of a physically secure channel such as registered mail for key distribution, they must be prepared to wait while the keys are sent, or have made prior preparation for cryptographic communication.

In the military, the chain of command helps to limit the number of user-pair connections, but even there, the key distribution problem has been a major impediment to the use of cryptography. This problem will be accentuated in large commercial communication networks where the number of possible connections is $(n^2-n)/2$ where n is the number of users. A system with one million users has almost 500 billion possible connections, and the cost of distributing this many keys is prohibitive.

At this point we introduce a new kind of cryptographic system which simplifies the problem of key distribution. It is possible to dispense with the secure key distribution channel of figure 1, and communicate over the insecure channel without any prearrangement. As indicated in figure 2, two way communication is allowed between the transmitter and receiver, but the eavesdropper is passive and only listens. Systems of this type are called public key systems, in contrast to conventional systems.

The reason that keys must be so carefully protected in conventional cryptographic systems is that the enciphering and deciphering functions are inseparable. Anyone who has access to the key in order to encipher messages can also decipher messages. If the enciphering and deciphering capabilities are separated, privacy can be achieved without keeping the enciphering key secret, because it can no longer be used for deciphering.

The new systems must be designed so that it is easy to generate a random pair of inverse keys E, for enciphering, and D, for deciphering, and easy to operate with E and D, but computationally infeasible to compute D from E.

A public key cryptosystem is a pair of families $\{E_K\}$ and $\{D_K\}$ for K in {K}, of algorithms representing an invertible transformation and its inverse defined such that:

1) For every K in {K}, D_K is the inverse of E_K. That is, $D_K(E_K(M)) = M$, for any K and any M.

2) For every K in {K} and M in {M}, the values $E_K(M)$ and

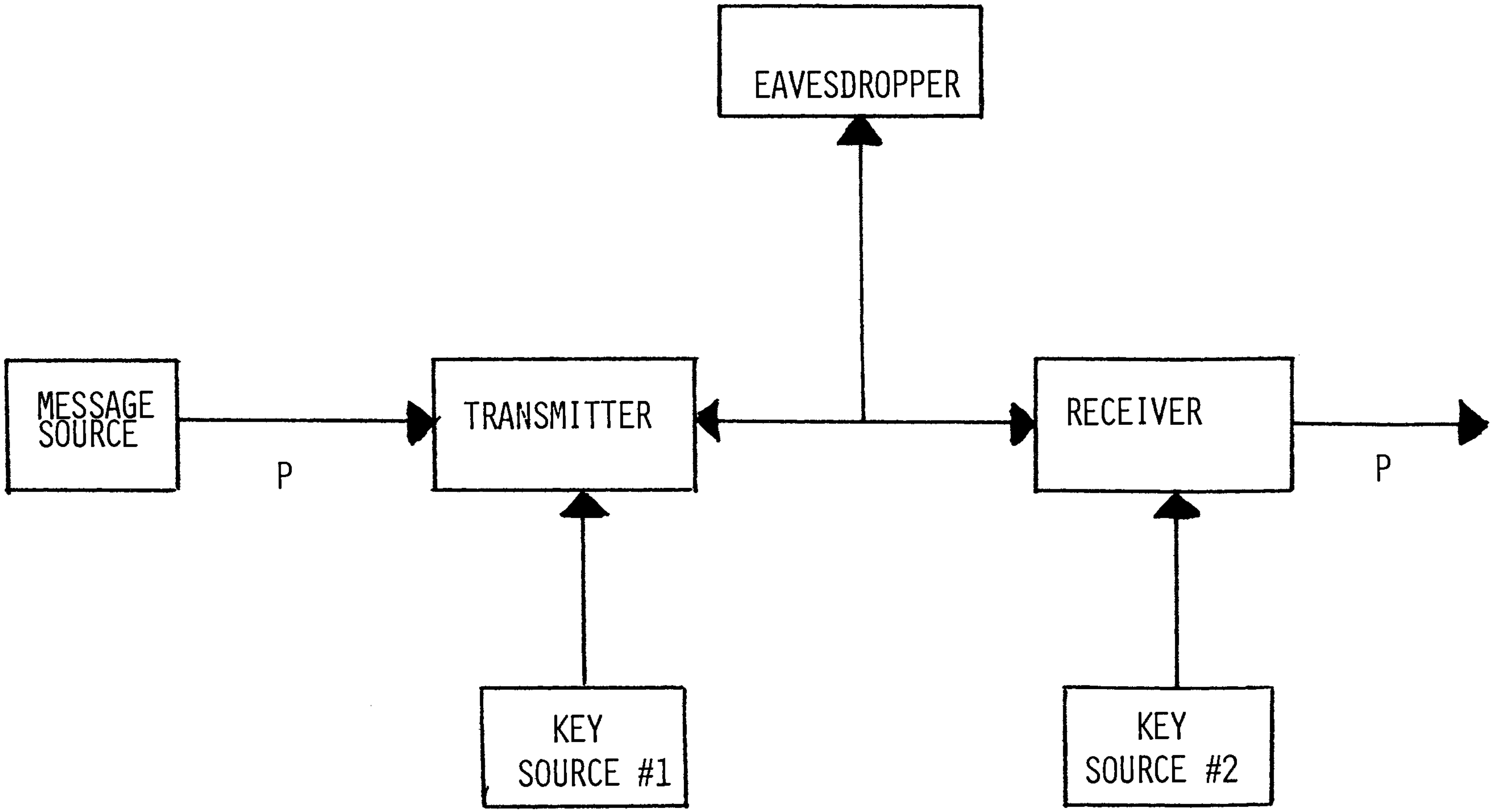

FIG 2
THE FLOW OF INFORMATION IN A
PUBLIC KEY SYSTEM.

$D_K(M)$ are easy to compute.

3) For nearly all K in {K}, any easily computed algorithm equivalent to D_K is computationally infeasible to derive from E_K.

4) For every K in {K}, it is feasible to generate the inverse pair E_K and D_K from K.

The third property allows a user's enciphering key E_K to be made public without compromising the security of his secret deciphering key D_K. The cryptographic system is therefore split into two parts, a family of enciphering transformations, and a family of deciphering transformations in such a way that given a member of one family it is infeasible to find the corresponding member of the other.

The fourth property guarantees that there is a feasible way of computing corresponding pairs of inverse transformations when no constraint is placed on what either the enciphering or deciphering transformation is to be. In practice, the cryptoequipment must contain a true random number generator (e.g., a noisy diode) for generating K, together with an algorithm for generating the E_K-D_k pair from K.

A system of this kind greatly simplifies the problem of key distribution. Each user generates a pair of inverse transformations, E and D. He keeps the deciphering transformation D secret, and makes the enciphering transformation E public by, for example, placing it in a public directory similar to a phone book. Anyone can now encrypt messages and send them to the user, but no one else can decipher messages intended for him.

If in addition to conditions 1) - 4) above, the set of transformations satisfy

1') For every K in {K}, E_K is the inverse of D_K. That is for any K and any M, $E_K D_K(M) = M$.

It is possible, and often desirable, to encipher with D and decipher with E. For this reason, E_K is sometimes called the public key, and D_K the secret (or signing) key.

4. Digital Signatures

A second difficulty which has limited the application of conventional cryptography is its inability to deal with the problem of dispute. Conventional authentication systems can prevent third party forgeries, but cannot settle disputes between the sender and receiver as to what message, if any, was sent.

In current commercial practice, the validity of contracts and agreements is guaranteed by handwritten signatures. A signed contract serves as proof of an agreement which the holder can present in court if necessary, but the use of signatures requires the transmission and storage of written documents: a major barrier to more widespread use of electronic communications in business.

The essence of a signature is that although only one person can produce it, anybody can recognize it. If there is to be a purely digital replacement for this paper instrument, each user must be able to produce messages whose authenticity can be checked by anyone, but which could not have been produced by anyone else, especially the intended recipient. In a conventional system the receiver authenticates any message he receives from the sender by deciphering it in a key which the two hold in common. Because this key is held in common, however, the receiver has the ability to produce any cryptogram that could have been produced by the sender and so cannot prove that the sender actually sent a disputed message.

Public key cryptosystems provide a direct solution to the signature problem, if they satisfy condition 1'). Systems which almost satisfy 1') are also usable (see chapter VI).

If user A wishes to send a signed message M to user B, he operates on it with his private key D_A to produce the signed message $S = D_A(M)$. D_A was used as A's deciphering key when privacy was desired, but is now used as his "enciphering" or "signing" key. When user B receives S he can recover M by operating on S with A's public key E_A.

B saves S as proof that user A sent him the particular message M. If A later disclaims having sent this message, B can take S to a judge who obtains E_A and checks that $E_A(S) = M$ is a meaningful message with A's name at the end, the proper date and time, etc. Only user A could have generated S because only he knows D_A, so A will be held responsible for having sent M.

This technique provides unforgeable, message dependent, digital signatures, but allows any eavesdropper to determine M because only the public information E_A is needed to recover M from S. To obtain privacy of communication as well, A can encrypt S with B's public key and send $E_B(S)$ instead of S. Only B knows D_B, so only he can recover S and thence M. B still saves S as proof that user A sent him M.

Other methods of generating digital signatures which do not depend on public key cryptosystems have been suggested [6], [19] and Chapter V.

[Note: This concludes the two sections taken largely from [4].]

II. ONE WAY HASH FUNCTIONS

There are many instances in which a large data field (e.g. 10,000 bits) needs to be authenticated, but only a small data field (e.g. 100 bits) can be stored or authenticated. (See, for example, chapter V). It is often required that it be infeasible to compute other large data fields with the same image under the hash function, giving rise to the need for a _one way hash function_.

Intuitively, a one way hash function F is one which is easy to compute but difficult to invert and can map arbitrarily large data fields onto much smaller ones. If y = F(x), then given x and F, it is easy to compute y, but given y and F it is effectively impossible to compute x. More precisely:

1) F can be applied to any argument of any size. F applied to more than one argument (e.g. $F(x_1,x_2)$) is equivalent to F applied to the concatenation of the arguments, i.e. $F(<x_1,x_2>)$.
2) F always produces a fixed size output, which, for the sake of concreteness, we take to be 100 bits.
3) Given F and x it is easy to compute F(x).
4) Given F and F(x), it is computationally infeasible to determine x.
5) Given F and x, it is computationally infeasible to find an $x' \neq x$ such that $F(x) = F(x')$.

The major use of one way functions is for authentication.

If a value y can be authenticated, we can authenticate x by computing

$$F(x) = y$$

and authenticating y.

No other input x' can be found (although they probably exist) which will generate y. A 100 bit y can authenticate an arbitrarily large x. This property is crucial for the convenient authentication of large amounts of information. Although a 100 bit y is plausible, selection of the size in a real system involves tradeoffs between the reduced cost and improved efficiency of a smaller size, and the improved security of a larger size.

Because y is used to authenticate the corresponding x, it would be intolerable if someone could compute an x' such that y = F(x) = F(x'). The fraudulent x' could be substituted for the legitimate x and would be authenticated by the same information. If y is 100 bits long, an interloper must try about 2^{100} different values of x' before getting a value such that F(x') = y. In an actual system, F will be applied to many different values of x, producing many different values of y. As a consequence, trying fewer than 2^{100} different values of x will probably yield an x' such that F(x') = y for some already authenticated y. To take a concrete example, assume F has been applied to 2^{40} different values of x, and produced 2^{40} corresponding values of y, each of which has been authenticated. If the y's are 100 bits, then a random search over 2^{60} values of x would probably yield an x' such that y = F(x) = F(x') for some value of y. While this search is still difficult, it is easier than searching over 2^{100} different values of x. This demonstrates that y might have to be longer than expected in a heavily used system. Forcing an opponent to search over all 2^{100} different values of x would be more desirable. This can usually be done by using many different functions, F_1, F_2, ... The effect of using many different one way functions is to prevent analysis of F by exhaustive techniques, because each value of x is authenticated with a distinct F_i. This will significantly increase security, yet requires only minor changes in implementation.

Functions such as F can be defined in terms of conventional cryptographic functions [6]. Assume we have a conventional encryption function C(key,plaintext) which has a 200 bit key size and encrypts 100 bit blocks of plaintext into 100 bit blocks of ciphertext. (It is a common misconception that the key can be no larger than the plaintext blocksize, but as an example the DES can be regarded as having a 768 bit key and a 64 bit block size).

We first define F_0, which is simpler than F and which satisfies properties 2, 3, 4, and 5; but whose input x is restricted to be 200 bits. We define

$$F_0(x) = y = C(x,\underline{0})$$

F_0 accepts a 200 bit input x and produces a 100 bit output y, as desired. Furthermore, given y, the problem of finding an x' such that $F(x') = y$ is equivalent to finding a key x' such that $y = C(x',\underline{0})$. If C is a good encryption function, this is computationally infeasible.

If the input x to F is fewer than 200 bits, then we can "pad" x by adding 0's until it is exactly 200 bits, and then define $F = F_0$. If the input is more than 200 bits, we will break it into 100 bit pieces. Assume that

$$\underline{x} = x_1, x_2, \ldots x_k$$

and that each x_i is 100 bits long. Then F is defined in terms of repeated applications of F_0. F_0 is first applied to x_1 and x_2 to obtain $y_1 = F_0(x_1,x_2)$. Then $y_2 = F_0(y_1,x_3)$, $y_3 = F_0(y_2,x_4)$, $y_4 = F_0(y_3,x_5)$, ... $y_i = F_0(y_{i-1},x_{i+1})$, ... $y_{k-1} = F_0(y_{k-2},x_k)$. $F(\underline{x})$ is defined to be y_{k-1}; the final y in the series. If $\underline{x}$ is not an exact multiple of 100 bits, then it is padded with 0's, as above.

It is obvious that F can accept arbitrarily large values for x. Although complexity theory has not progressed to the point where it is possible to prove that it will be computationally infeasible to find any vector $\underline{x}'$ not equal to $\underline{x}$ such that $F(\underline{x}) = F(\underline{x}')$, a plausibility argument can be made inductively that this is the case. As a basis, when $k = 2$, the property holds because $F(\underline{x}) = F_0(x_1,x_2)$, and the property holds for F_0 by assumption. We establish the case for $k = 3$ by contradiction. We assume that $F(x_1,x_2,x_3) = F(x_1',x_2',x_3')$ and that $x_i \neq x_i'$ for some i in {1,2,3}. We first note that $F(x_1,x_2,x_3) = F_0(y_1,x_3)$ by definition. If either $y_1 \neq y_1'$ or

$x_3 \neq x_3'$, and $F_0(y_1,x_3) = F_0(y_1',x_3')$, then we have violated assumption 5 made about F_0. If $y_1 = y_1'$ and $x_3 = x_3'$, then either $x_1 \neq x_1'$ or $x_2 \neq x_2'$. By definition $F_0(x_1,x_2) = y_1$, and $F_0(x_1',x_2') = y_1'$, so $F_0(x_1,x_2) = F_0(x_1',x_2')$ and again we contradict assumption 5 made about F_0. This line of logic can be extended to the cases k = 3, 4, 5,

This argument cannot be made fully rigorous until the properties of F_0 are made rigorous. This must await further advances in complexity theory.

III. A PUBLIC KEY CRYPTOSYSTEM BASED ON PUZZLES

This chapter describes the first public key distribution system and the first public key cryptosystem, which were both based on the concept of a "puzzle".

Puzzles can be used in a variety of ways to make public key distribution systems where the effort involved to break the system grows as the square of the effort to use the system. The method described in chapter IV is not the simplest, but was selected because it required the least memory and was deterministic. (As an interesting aside, Ron Rivest sent the author a letter mentioning this fact, and described a simpler system which was, in fact, the system the author originally devised.)

As originally conceived, the method was more closely linked to the concept of a one way function and was probabilistic in nature. Basically, it centered on the observation that if two people randomly select n numbers from a space of n^2 numbers, there is a significant probability that both will have selected at least one number in common. This is closely related to the "birthday problem" [41]. Given n people in a room, what is the probability that at least two of them were born on the same day? The probability becomes surprisingly high when there are more than square root(365) $\cong$ 20 people in the room.

This surprising statistical result can be used for public key distribution rather easily. If A and B wish to agree on a common key, then each selects n numbers from a space of n^2 numbers. The probability that they selected a common number is significant and if A and B can determine this common number it can be used as a cryptographic key in further communications. Note that the number of possible keys is n^2, so anyone who tries to break this method must search through all n^2, rather than the O(n) keys that A and B know.

A and B determine the common number as follows: A applies a one way function F to hide each of his n randomly chosen numbers, and sends them to B. B applies F to hide his randomly chosen numbers and sends the result to A. A and B can now sort both lists of hidden numbers, and look for a match. If A and B chanced upon the same number, then the hidden version of this number computed by A and B will be the same. All that E knows is the hidden versions of the numbers. E can easily look through A's and B's hidden numbers and find the match, but then E must search through all n^2 possible values to find the original value, which is used as the key. A and B, on the other hand, already KNOW the original values, because they generated the hidden values by applying F to numbers they knew.

The next phase in the evolutionary development of the method was to create a deterministic version of the original method. This is done fairly easily. Instead of selecting n numbers from n^2 possibilities completely at random, select one number randomly in the range from 1 to n, the second number randomly in the range from n+1 to $2 \cdot n$, the third number randomly in the range from $2 \cdot n + 1$ to $3 \cdot n$, the ith number from the range $(i-1) \cdot n + 1$ to $i \cdot n$, and the nth number from the range $(n-1) \cdot n + 1$ to $n \cdot n$. This guarantees that there must be one number in each of the n possible ranges. By picking a random range, and then searching sequentially through all possible numbers in that range, a "collision" must occur. In other words, A picks a single number randomly from each range, hides them, randomly permutes their order and transmits the permuted hidden values to B. B picks a random range, hides all n numbers in that range, but does not send the result to A. Instead, B looks for a match between the hidden numbers A sent, and the hidden numbers B just generated. When B finds the match, B sends the hidden value back to A, who compares this single hidden number against the n hidden numbers A generated. The method now deterministically achieves an n^2:n ratio of effort (work factor). However, it still requires a great deal of memory.

The method described in chapter IV was the next evolutionary step beyond this.

The "puzzles" method also evolved into the first public key cryptosystem. Basically, the enciphering and deciphering keys are just explicit tabular representations of randomly chosen enciphering and deciphering functions. The only modification is to the enciphering key. It cannot be represented in a simple tabular format, because this would allow it to be inverted too easily, i.e., the public enciphering key must be hard to invert, and a tabular format is not hard to invert, so the tabular format must be extended somewhat. This is done by enpuzzling the elements of the range.

First, we define the enpuzzlement of an argument by the function:

P(x,n)

where x is the value to be enpuzzled, and n represents the difficulty of breaking the resulting puzzle. P(345,45) means that the number "345" can be recovered by putting in 45 units of effort (on average).

A small enciphering key; which maps plaintext 1 into ciphertext 7, plaintext 2 into ciphertext 3, ... and plaintext 8 into ciphertext 5; is shown in figure 1, with n = 8.

The Enciphering Key

1	P(7,8)
2	P(3,8)
3	P(6,8)
4	P(8,8)
5	P(1,8)
6	P(4,8)
7	P(2,8)
8	P(5,8)

Figure 1

Note that it requires O(n) units of effort to compute

E(plaintext) from this tabular representation, but that computing D(ciphertext) requires $O(n^2)$.

The corresponding secret deciphering key is shown in figure 2.

The Deciphering Key

1	5
2	7
3	2
4	6
5	8
6	3
7	1
8	4

Figure 2

The secret deciphering key is not enpuzzled, and so it is easy to compute D(ciphertext) from it, (or to compute E(plaintext), but this is largely beside the point). Making the enciphering and deciphering keys can be done in O(n) time, enciphering requires O(n) time, deciphering can be done in unit time, but breaking the system requires $O(n^2)$ time.

IV. PUBLIC KEY DISTRIBUTION USING PUZZLES

1. INTRODUCTION

This chapter describes the first public key system ever developed. Until this system, it had been assumed that a necessary precondition for cryptographically secure communications was the transmission of a key, by secret means, prior to an attempt to communicate securely. The system described below, however, allows two communicants to select a key publicly, but in such a fashion that no one else can easily determine what it is.

The body of the chapter will begin with a description of a conventional cryptographic system, in which secure transmission of the key is required. It will then develop the new concept of a public key system. The implications of public key systems will then be explored in more detail, with the aid of some examples.

2. REVIEW

We introduce three protagonists into our paradigm: A and B, the two communicants, and E, the enemy, who wishes to find out what A and B are communicating. A and B have available a conventional cryptographic system for encrypting and decrypting messages that they send to each other. A, B, and E all know the general method of encryption. A and B also have available a normal communications channel, over which they send the bulk of their messages. To allow A and B to communicate securely, they must load a key, which is unknown to E, into their cryptographic devices. The general method uses this key as a parameter, and will perform a particular transformation on messages for a particular key. Because E does not know this key, he cannot perform the particular transformation, and thus cannot encrypt or decrypt messages.

A and B must both know what the key is, and must insure that E does not know what it is. In the traditional paradigm for cryptography, this situation comes about by the transmission of the key from A to B over some special and secure communications channel which we shall refer to as the key channel. E cannot intercept messages sent on this channel, and the key is therefore safe.

The key channel is not used for normal communications because of its expense and inconvenience.

In view of the central position that the key channel will occupy in this chapter, it would be wise to state, somewhat more clearly, the conditions which it must satisfy. There are two such conditions.

1) E cannot modify or alter messages on the key channel, nor can he inject false or spurious messages.

2) E is unable to determine the content of any message sent over the key channel, i.e., E cannot intercept the messages.

Systems of this type are referred to as conventional cryptographic systems, and their study dates back to antiquity (See Shannon [35] for a good overview). We now make a modification which had not previously been considered.

3. THE NEW APPROACH

We modify the traditional paradigm by dropping the second restriction on the key channel, but not the first. We no longer demand that E be unable to learn what is sent on the key channel, rather, we assume that E has perfect knowledge of everything that is sent over this channel [footnote page 31]. Although in some instances the key and normal channels may be one and the same, we shall treat them as logically distinct.

It is the thesis of this chapter that secure communications between A and B can take place under the conditions we have just described.

The reader should clearly understand that no key lurks in the background. There is no method by which A and B can communicate other than the normal channel and the key channel. They have made no secret preparations prior to the time that they wish to communicate securely.

We must carefully consider what constitutes a solution. If A and B eventually agree upon a key, and if the work required of E to determine the key is much higher than the work put in by either A or B to select the key, then we have a solution. Note that E can determine the key used in most conventional cryptographic systems (with the exception of the one time pad) simply by trying all possible keys and seeing which one produces a legible message. However, the amount of work required grows exponentially compared to the amount of work put in by A or B. The public key solution described is not exponential, but the amount of work required of E to determine the key increases as the square of the amount of work put in by A and B to select the key. Methods which appear to force E to put in an amount of work which grows exponentially with the amount of work A and B put in have been discovered since the conception of this method. While this method is therefore not as practical, its simplicity makes it the most nearly provably secure system and the best for pedagogical purposes. It relies on little more than the existence of one way functions.

4. THE METHOD

The method is based on the concept of a puzzle, that is, a cryptogram which is meant to be broken. To solve the puzzle, we must cryptanalyze the cryptogram. Having done this, we learn the information that was "enpuzzled", the plaintext of the cryptogram. Just as we can encrypt plaintext to produce a cryptogram, so we can enpuzzle information to produce a puzzle. A puzzle, though, is meant to be solved, while ideally, a cryptogram cannot be cryptanalyzed. To solve a puzzle, all you need do is put in the required amount of effort.

To sharpen our definition, we will consider the following method of creating puzzles. First, select a strong encryption function. We are not interested in the details of how this encryption function works: our only interest is that it does work. The reader can select any encryption function that he feels is particularly strong and effective. A concrete example might be the DES encryption function [24], which with a longer key is currently felt to be quite strong.

After selecting an encryption function, we create our puzzle by encrypting some piece of information with that function with a key chosen at random from a specified subset of the keyspace. We artificially restrict the size of the key space used with the encryption function to make the puzzle solvable. If the key is normally 128 bits, we might use only 30 bits and set the remaining 98 bits to 0. While searching through 2^{128}

possible keys is completely infeasible, searching through 2^{30} is tedious, but quite possible. We can control the difficulty of solving a puzzle, simply by changing the restriction on the size of the key space used. To make the puzzle harder to solve, we might select a 40 bit key, while to make it easier, we might select a 20 bit key. We assume the strength of the underlying encryption function is adequate to insure that our puzzle can only be solved by exhaustive search through the restricted key space, and we can then adjust the size of the key space to precisely control the difficulty of solving the puzzle.

There is still one more point that must be brought out. In cryptanalyzing an encrypted message, the cryptanalyst relies on redundancy in the message to indicate when the proper key is tried. If the information we enpuzzle is random, there will be no redundancy, and thus no way of solving the puzzle. We must deliberately introduce redundancy into our puzzle, so that it can be solved. This can be done easily enough by encrypting, along with the information, a constant that is publicly stated. When we try to decrypt the puzzle with a particular key, the recovery of this constant can be taken as evidence that we have selected the right key, and thus have solved the puzzle. The absence of the constant part in the decrypted puzzle guarantees that we have used the wrong key, and should try again. While an incorrect key can produce a false alarm, if the constant field is larger than the number of bits in the restricted key space, then unicity distance arguments indicate that false alarms should be rare.

With the concept of puzzle in hand, we can proceed. We let A and B agree upon the value of N which they wish to use. A then generates N puzzles, and transmits these N puzzles to B over the key channel. A chooses the size of the key space so that these puzzles require O(N) effort to break. (That is, A selects a key space of size K·N, for a constant, K.) Each puzzle contains, within itself, two pieces of information. Neither piece of information is readily available to anyone examining the puzzle. By devoting O(N) effort to solving the puzzle, it is possible to determine both these pieces of information. One piece of information is a puzzle id, which uniquely identifies each of the N puzzles. The ids were assigned by A at random. The other piece of information in the puzzle is a random bit string which is the proper size for use as a true (unrestricted) key, i.e., one of the possible keys to be used in subsequent encrypted communications. To distinguish the true keys, one for each puzzle, from the keys randomly selected from the restricted key space to create the puzzles, we will call the former "true keys", and the latter, "restricted keys". Thus, N true keys are enpuzzled, and in the process of enpuzzling each true key, a restricted key is used.

When B is presented with this menu of N puzzles, he selects a puzzle at random and spends O(n) effort to solve the puzzle. B then transmits the id back to A over the key channel, and uses the true key found in the puzzle as the key for further encrypted communications over the normal channel.

A, B, and E all know the N puzzles. They also know the id, because B transmitted the id over the key channel. B knows the corresponding true key, because B selected the puzzle to be solved. A knows the corresponding true key, because A knows which true key is associated with the id that B sent. E knows only the id, but does not know the true key. E does not know which puzzle contains the true key that B selected, and which A and B are using, even though he knows the id. To determine which puzzle is the correct one, he must break puzzles at random until he encounters the one with the correct id.

If E is to determine the key which A and B are using, then, on an average, E will have to solve 1/2 N puzzles before reaching the puzzle that B solved. Each puzzle has been constructed so that it requires O(N) effort to break, so E must spend, on an average, $O(N^2)$ effort to determine the key. B, on the other hand, need only spend O(N) effort to break the one puzzle he selected, while A need only spend O(N) effort to manufacture the N puzzles. Thus, both A and B will only put in O(N) effort. A detailed description appears in [21].

In summary: the method allows the use of channels satisfying assumption 1, and not satisfying assumption 2, for the transmission of key information. We need only guarantee that messages are unmodified, and we no longer require that they be unread. If the two communicants, A and B, put in O(N) effort, then the enemy, E, must put in O(N^2) effort to determine the key.

Generating the n puzzles is orders of magnitude less costly than transmitting them. Creating a method in which the ratio of efforts was still n^2:n, but which did not require transmitting O(n) bits would be a substantial and practical improvement. There seems no reason in principal why such an improvement should not be possible.

5. FOOTNOTE

Wyner [39] introduced a different (information theoretic) approach to secure communication over an insecure channel without prearrangement. Wyner assumes that the wiretapper E has inferior reception of the messages being transmitted. By taking advantage of this inferior reception, Wyner shows how the wiretapper can be completely confused. Our approach is different and assumes that both the legitimate receiver and the wiretapper perfectly receive whatever the transmitter sends.

V. A CERTIFIED DIGITAL SIGNATURE

1. Introduction

Digital signatures promise to revolutionize business by phone or other telecommunication devices [6] but use of the currently known public key cryptosystems [18], [20], [21], [31] is risky until they have been carefully certified. A signature system whose security rested solely on the security of a conventional cryptographic function would be "pre-certified" to the extent that the underlying encryption function had been certified. The delays and cost of a new certification effort would be avoided. Lamport and Diffie [6] suggested such a system, but it has severe performance drawbacks. Lipton and Matyas [17] nonetheless suggested its use as the only near term solution to a pressing problem.

This chapter describes a digital signature system which is "pre-certified" in the above sense, generates signatures of about 15 kilobits (2 kilobytes), requires a few thousand applications of the underlying encryption function per signature, and only a few kilobytes of memory. If the underlying encryption function takes 10 microseconds to encrypt a block, generating a signature takes approximately 20 milliseconds.

A CERTIFIED DIGITAL SIGNATURE

The following major points are covered:

1.) A description of the Lamport-Diffie one time signature.

2.) An improved version of the Lamport-Diffie one time signature.

3.) A method of converting any one time signature into a convenient signature system.

2. The Lamport-Diffie One Time Signature

The Lamport-Diffie one time signature [6] is based on the concept of a one way function [7] ,[38]. If y = F(x) is the result of applying the one way function F to input x, then the key observation is:

> The person who computed y = F(x) is the only person who knows x. If y is publicly revealed, only the originator of y knows x, and can choose to reveal or conceal x at his whim.

This is best clarified by an example. Suppose a person A has some stock, which he can sell at any time. A might wish to sell the stock on short notice, which means that A would like to tell his broker over the phone. The broker, B, does not wish to sell with only a phone call as authorization. To solve this problem, A computes y = F(x) and gives y to B. They agree that when A wants to sell his stock he will reveal x to B. (This agreement could be formalized as a written contract [17] which includes the value of y and a description of F but not the value of x.) B will then be able to prove that A wanted to sell his stock, because B will be able to exhibit x, and demonstrate that F(x) = y.

If A later denies having ordered B to sell the stock, B can show the contract and x to a judge as proof that A, contrary to his statement, did order the stock sold. Both F and y are given in the original (written) contract, so the judge can compute F(x) and verify that it equals y. The only person who knew x was A, and the only way B could have learned x would be if A had revealed x. Therefore, A must have revealed x; an action which by prior agreement meant that A wanted to sell his stock.

This example illustrates a signature system which "signs" a single bit of information. Either A sold the stock, or he did not. If A wanted to tell his broker to sell 10 shares of stock, then A must be able to sign a several bit message. In the general Lamport-Diffie scheme, if A wanted to sign a message m whose size was s bits, then he would precompute $F(x_1) = y_1$, $F(x_2) = y_2$, $F(x_3) = y_3$,... $F(x_s) = y_s$. A and B would agree on the vector $Y = y_1, y_2 \ldots y_s$. If the jth bit of m was a 1, A would reveal x_j. If the jth bit of m was a 0, A would not reveal x_j. In essence, each bit of m would be individually signed. Arbitrary messages can be signed, one bit at a time.

In practice, long messages (greater than 100 bits) can be mapped into short messages (100 bits) by a one way function and only the short message signed. We can therefore assume, without loss of generality, that all messages are a fixed length, e.g., 100 bits.

The method as described thus far suffers from the defect that B can alter m by changing bits that are 1's into 0's. B simply denies he ever received x_j, (in spite of the fact he did). However, 0's cannot be changed to 1's. Lamport and Diffie overcame this problem by signing a new message m', which is exactly twice as long as m and is computed by concatenating m with the bitwise complement of m. That is, each bit m_j in the original message is represented by two bits, m_j and the complement of m_j in the new message m'. Clearly, one or the other bit must be a 0. To alter the message, B would have to turn a 0 into a 1, something he cannot do.

It should now be clear why this method is a "one time" signature: Each Y = y_1, y_2, ... $y_{2\ s}$ can only be used to sign one message. If more than one message is to be signed, then new values Y_1, Y_2, Y_3, ... are needed, a new Y_i for each message.

One time signatures are practical between a single pair of users who are willing to exchange the large amount of data necessary but they are not practical for most applications without further refinements. If each y_i is 100 bits long and a 100 bit one way hash function of each message is signed, each Y_i must be 20,000 bits. If 1000 messages are to be signed before new public authentication data is needed, over 20,000,000 bits or 2.5 megabytes must be stored as public information. Even if this is not overly burdensome when only two users, A and B, are involved in the signature system, if B had to keep 2.5 megabytes of data for 1000 other users, B would have to store 2.5 gigabytes of data. While possible, this hardly seems economical. With further increases in the number of users, or in the number of messages each user wants to be able to sign, the system becomes completely unwieldy.

How to eliminate the huge storage requirements is a major subject of this chapter.

3. An Improved One Time Signature

This section explains how to reduce the size of signed messages in the Lamport-Diffie method by almost a factor of 2.

As previously mentioned, the Lamport-Diffie method solves the problem that 1's in the original message can be altered to 0's by doubling the length of the message, and signing each bit and its complement independently. In this way, changing a 1 to a 0 in the new message, m', would result in an incorrectly formatted message, which would be rejected. In essence, this represents a solution to the problem:

> Create a coding scheme in which accidental or intentional conversion of 1's to 0's will produce an illegal codeword.

An alternative coding method which accomplishes the same result is to append a count of the number of 0 bits in m before signing. The new message, m', would be only $\log_2$ s bits longer than the original s bit message, m. If any 1's in m' were changed to 0's (0's cannot be changed to 1's), it would falsify the count of 0's.

Notice that while it is possible to reduce the count by changing 1's to 0's in the count field, and while it is possible to increase the number of 0's by changing 1's to 0's in the message, these two "errors" cannot be made to compensate for each other.

A small example is in order. Assume that our messages are 8 bits long, and that our count is $\log_2 8 = 3$ bits long. If our message m is

m = 11010110

Then m' would be

m' = 11010110,011

(Where a comma is used to clarify the division of m' into m and its 0 count.)

If the codeword 11010110,011 were changed to 01010110,011 by changing the first 1 to a 0, then the count 011 would have to be changed to 100 because we now have 4 0's, not 3. Put this requires changing a 0 to a 1, something we cannot do. If the codeword were changed to 11010110,010 by altering the 0 count then the message would have to be changed so that it had only 2 0's instead of 3. Again, this change is illegal because it requires changing 0's to 1's.

This improved method is easy to implement and cuts the size of the signed message almost in half, although this is still too large for most applications; e.g., it reduces 2.5 gigabytes to 1.25 gigabytes.

4. Tree Authentication

A new protocol would eliminate the large storage requirements and the need for prior arrangements. If A transmitted Y_i to B just before signing a message, then B would not previously have had to get and keep copies of the Y_i from A. Unfortunately, such a protocol would not work because anyone could claim to be A, send a false Y_i, and trick B into thinking he had received a properly authorized signature when he had received nothing of the kind. B must somehow be able to confirm that he was sent the correct Y_i and not a forgery.

The problem is to authenticate A's Y_i. The simplest (but unsatisfactory) method is, as suggested above, to keep a copy of A's Y_i. In this section, we describe a method called "tree authentication" which can be used to authenticate any Y_i of any user quickly and easily, but which requires minimal storage.

Problem Definition: Given a vector of data items $\underline{Y} = Y_1$, Y_2, ... Y_n design an algorithm which can quickly authenticate a randomly chosen Y_i but which has modest memory requirements, i.e., does not have a table of Y_1, Y_2, ... Y_n.

We authenticate the Y_i by "divide and conquer". As illustrated in figure 1, define the function $H(i,j,\underline{Y})$ by

1.) $H(i,i,\underline{Y}) = F(Y_i)$

2.) $H(i,j,\underline{Y}) = F(\ H(i,k,\underline{Y}),\ H(k+1,j,\underline{Y})\ )$

where $k = (i+j)/2$

for the <i,j> pairs of figure 1 and its extensions to larger trees described below.

$H(i,j,\underline{Y})$ is a function of Y_i, Y_{i+1}, ... Y_j and can be used to authenticate all of them. $H(1,n,\underline{Y})$ can be used to authenticate Y_1 through Y_n and is only 100 bits, so it can be conveniently stored. We restrict n to powers of 2 to simplify the explanation.

This method lets us selectively authenticate any "leaf," Y_i, that we wish. To see this, we use an example where n = 8. To authenticate Y_5, we proceed in the manner illustrated in figure 2:

1.) $H(1,8,\underline{Y})$ is already known and authenticated.

2.) $H(1,8,\underline{Y}) = F(\ H(1,4,\underline{Y}),\ H(5,8,\underline{Y})\ )$. Send $H(1,4,\underline{Y})$ and $H(5,8,\underline{Y})$ and let the receiver compute $H(1,8,\underline{Y}) = F(\ H(1,4,\underline{Y}),\ H(5,8,\underline{Y})\ )$ to confirm that they are correct.

3.) The receiver has authenticated $H(5,8,\underline{Y})$. Send $H(5,6,\underline{Y})$ and $H(7,8,\underline{Y})$ and let the receiver compute

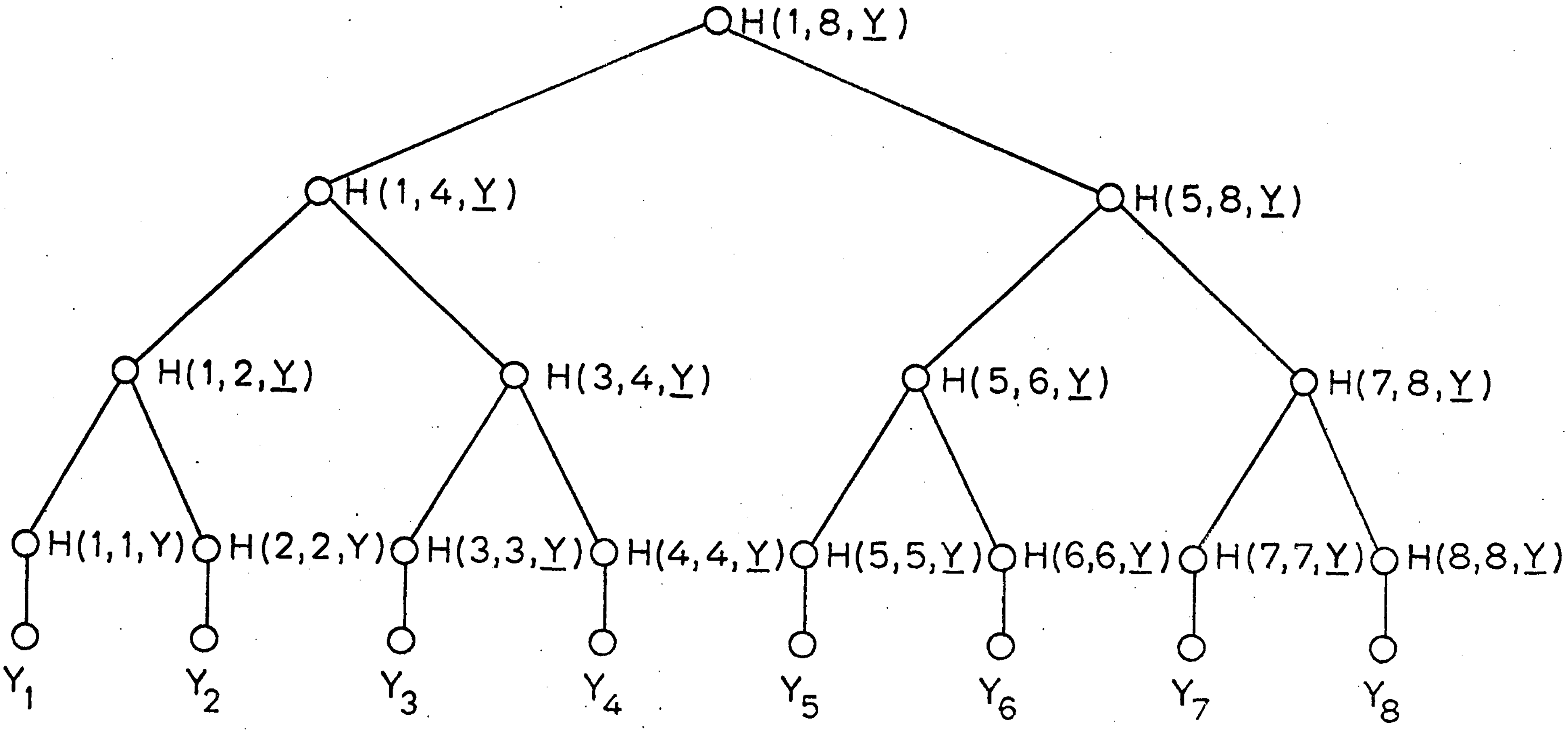

FIG 1

AN AUTHENTICATION TREE WITH N = 8.

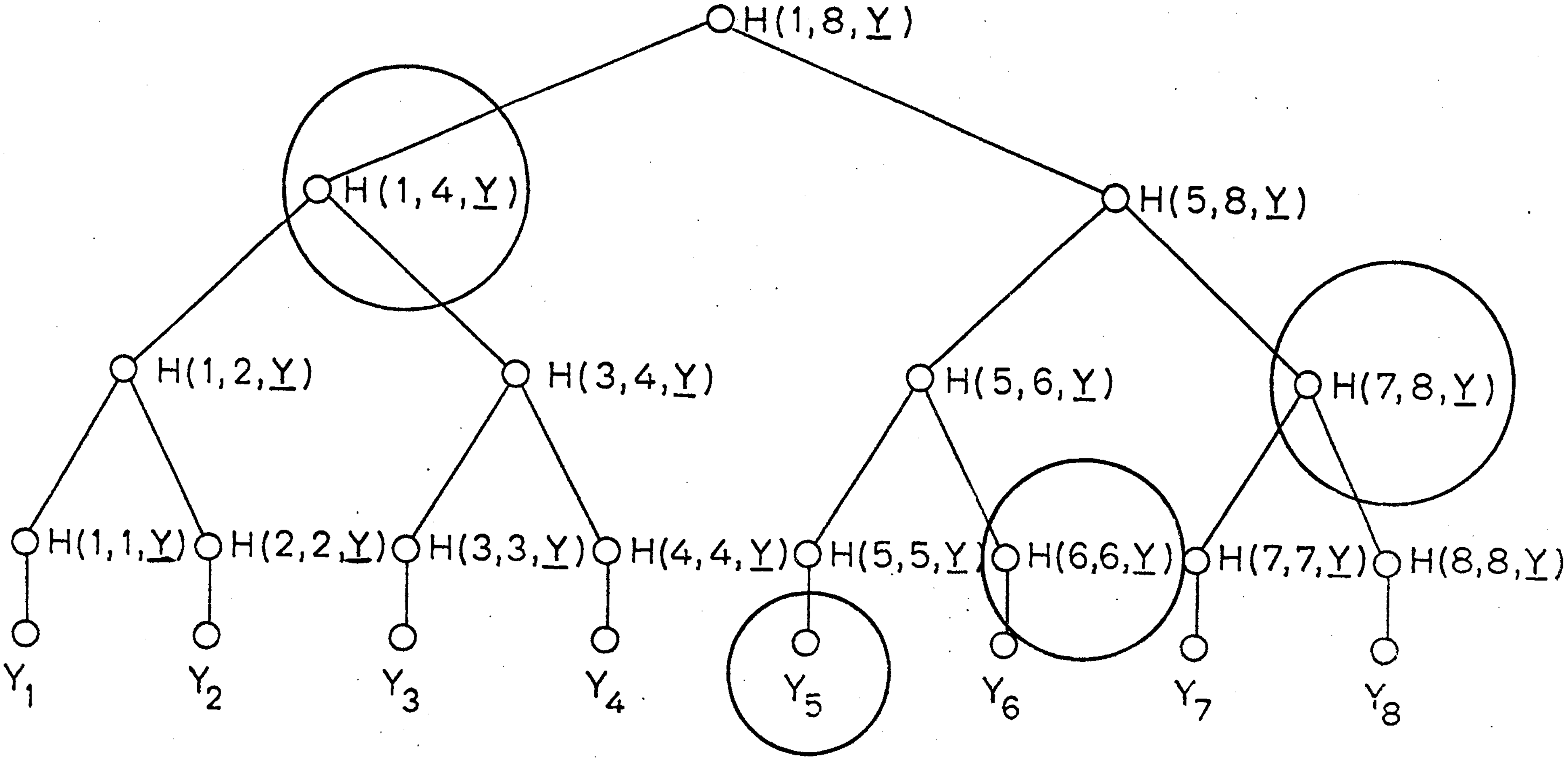

FIG 2

CIRCLED ENTRIES SHOW THE AUTHENTICATION PATH FOR Y_5.

$H(5,8,\underline{Y}) = F(\ H(5,6,\underline{Y}),\ H(7,8,\underline{Y})\ )$ to confirm that they are correct.

4.) The receiver has authenticated $H(5,6,\underline{Y})$. Send $H(5,5,\underline{Y})$ and $H(6,6,\underline{Y})$ and let the receiver compute $H(5,6,\underline{Y}) = F(\ H(5,5,\underline{Y}),\ H(6,6,\underline{Y})\ )$ to confirm that they are correct.

5.) The receiver has authenticated $H(5,5,\underline{Y})$. Send Y_5 and let the receiver compute $H(5,5,\underline{Y}) = F(\ Y_5\ )$ to confirm that it is correct.

6.) The receiver has authenticated Y_5.

Using this method, only $\log_2 n$ transmissions are required, each of about 200 bits. Close examination of the algorithm will reveal that half the transmissions are redundant. For example, $H(5,6,\underline{Y})$ can be computed from $H(5,5,\underline{Y})$ and $H(6,6,\underline{Y})$, so there is really no need to send $H(5,6,\underline{Y})$. Similarly, $H(5,8,\underline{Y})$ can be computed from $H(5,6,\underline{Y})$ and $H(7,8,\underline{Y})$, so $H(5,8,\underline{Y})$ need never be transmitted, either. (The receiver must compute these quantities anyway for proper authentication.) Therefore, to authenticate Y_5 required only that we have previously authenticated $H(1,8,\underline{Y})$, and that we transmit Y_5, $H(6,6,\underline{Y})$, $H(7,8,\underline{Y})$, and $H(1,4,\underline{Y})$. That is, we require $100 \log_2 n$ bits of information to authenticate an arbitrary Y_i.

The method is called tree authentication because the computation of $H(1,n,\underline{Y})$ forms a binary tree of recursive calls.

Authenticating a particular leaf Y_i in the tree requires only those values of H() starting from the leaf and progressing to the root, i.e., from $H(i,i,\underline{Y})$ to $H(1,n,\underline{Y})$. $H(1,n,\underline{Y})$ will be referred to as the root of the authentication tree, or R. The information near the path from R to $H(i,i,\underline{Y})$ required to authenticate Y_i will be called the authentication path for Y_i.

A "proof" that the authentication path actually authenticates the chosen leaf is similar to the "proof" that F defined in chapter II correctly authenticates its input. Again, more rigorous proofs must await advances in complexity theory.

Although H() produces a 100 bit output, unless additional precautions (outlined in chapter II) are taken, only 2^{60} or 2^{70} operations would suffice to break the system. To force the cryptanalysis to use 2^{100} operations it is necessary to make each application of F unique, i.e., to use a family of one way functions F_1, F_2, ... each one of which is used only once.

The use of tree authentication is now fairly clear. A transmits Y_i to B. A then transmits the authentication path for Y_i. B knows R, the root of the authentication tree, by prior arrangement. B can then authenticate Y_i, and can accept the <u>ith</u> signed message from A as genuine.

The prior arrangements include the computation of R by A. If A wishes to be able to sign 1,000,000 messages, this precomputation will require about an hour, assuming a single encryption takes 10 microseconds. (Fairchild is now (1979) producing a 4-chip set which costs about $100 and which encrypts faster than this.) The time required for the pre-computation is linear in n, so if A desires to be able to sign 1,000,000,000 messages, his pre-computation will be about 1000 hours.

The major distinction between this method and digital signatures generated using public key cryptosystems is the requirement that R be changed periodically because only n messages can be signed. With a public key cryptosystem, it is possible to sign an almost indefinite number of messages, and for a user to retain D_A for his lifetime if he so desires. In practice, this restriction does not appear to be significant.

If the <u>jth</u> user has a distinct authentication tree with root R_j, then tree authentication can be used to authenticate R_j just as easily as it can be used to authenticate Y_i. It is not necessary for each user to remember all the R_j in order to authenticate them. A central clearinghouse could accept the R_j from all u users, and compute $H(1,u,\underline{R})$. This single 100 bit quantity could then be distributed and would serve to authenticate all the R_j, which would in turn be used to authenticate the Y_i. In practice, A would remember R_A and the authentication path for R_A and send them to B along with Y_i and the authentication path for Y_i. (A different method of authentication would be for the clearinghouse to digitally sign "letters of reference" for new users of the system using a one time signature. Kohnfelder [14] has suggested this method for use with public key cryptosystems; see chapter 9.)

Tree authentication and authentication using one time signatures can be intermixed to produce systems with all the flexibility of public key based systems.

5. The Path Regeneration Algorithm

A must know the authentication path for Y_i before transmitting it to B. Unfortunately this requires the computation of $H(i,j,\underline{Y})$ for many different values of i and j. In the example, it was necessary to compute $H(6,6,\underline{Y})$, $H(7,8,\underline{Y})$, and $H(1,4,\underline{Y})$ and send them to B along with Y_5. This is simple for the small tree used in our example, but computing $H(4194304,8388608,\underline{Y})$ just prior to sending it would be an intolerable burden. One obvious solution would be to precompute $H(1,n,\underline{Y})$ and to save all the intermediate computations: i.e., precompute all authentication paths. This would certainly allow the quick regeneration of the authentication path for Y_i, but would require a large memory.

A more satisfactory solution is to note that we wish to authenticate Y_1, Y_2, Y_3, Y_4, ... in that order. Most of the computations used in reconstructing the authentication path for Y_i can be used in computing the authentication path for Y_{i+1}. Only the incremental computations need be performed, and these are quite modest.

In addition, although the Y_i must appear to be random, they can actually be generated (safely) in a pseudo-random fashion from a small truly random seed. It is not necessary to keep the Y_i in memory, but only the small truly random seed used to generate them.

The result of these observations is an algorithm which can recompute each Y_i and its authentication path quickly and with modest memory requirements. Before describing it we review the problem:

Problem Definition: Sequentially generate the authentication paths for Y_1, Y_2, Y_3, ... Y_n with modest time and space bounds.

The simplest way to understand how an algorithm can efficiently generate all authentication paths is to generate all the authentication paths for a small example.

An example of all authentication paths for n = 8 is:

leaf	authentication path
Y_1	$H(1,8,\underline{Y})$ $H(5,8,\underline{Y})$ $H(3,4,\underline{Y})$ $H(2,2,\underline{Y})$
Y_2	$H(1,8,\underline{Y})$ $H(5,8,\underline{Y})$ $H(3,4,\underline{Y})$ $H(1,1,\underline{Y})$
Y_3	$H(1,8,\underline{Y})$ $H(5,8,\underline{Y})$ $H(1,2,\underline{Y})$ $H(4,4,\underline{Y})$
Y_4	$H(1,8,\underline{Y})$ $H(5,8,\underline{Y})$ $H(1,2,\underline{Y})$ $H(3,3,\underline{Y})$
Y_5	$H(1,8,\underline{Y})$ $H(1,4,\underline{Y})$ $H(7,8,\underline{Y})$ $H(6,6,\underline{Y})$
Y_6	$H(1,8,\underline{Y})$ $H(1,4,\underline{Y})$ $H(7,8,\underline{Y})$ $H(5,5,\underline{Y})$
Y_7	$H(1,8,\underline{Y})$ $H(1,4,\underline{Y})$ $H(5,6,\underline{Y})$ $H(8,8,\underline{Y})$
Y_8	$H(1,8,\underline{Y})$ $H(1,4,\underline{Y})$ $H(5,6,\underline{Y})$ $H(7,7,\underline{Y})$

TABLE 1

If we had to separately compute each entry in table 1, then it would be impossible to efficiently generate the authentication paths. Fortunately, there is a great deal of duplication. If we eliminate all duplicate entries, then table 1 becomes table 2:

leaf	authentication path			
Y_1	H(1,8,$\underline{Y}$)	H(5,8,$\underline{Y}$)	H(3,4,$\underline{Y}$)	H(2,2,$\underline{Y}$)
Y_2				H(1,1,$\underline{Y}$)
Y_3			H(1,2,$\underline{Y}$)	H(4,4,$\underline{Y}$)
Y_4				H(3,3,$\underline{Y}$)
Y_5		H(1,4,$\underline{Y}$)	H(7,8,$\underline{Y}$)	H(6,6,$\underline{Y}$)
Y_6				H(5,5,$\underline{Y}$)
Y_7			H(5,6,$\underline{Y}$)	H(8,8,$\underline{Y}$)
Y_8				H(7,7,$\underline{Y}$)

TABLE 2

Clearly we can generate all authentication paths by separately computing each of the 2 n-1 entries in table 2, but this would require too much memory, and it is not clear what the execution time would be. We first consider the execution time, the memory requirement will be considered later. Because all computations must eventually be defined in terms of the underlying encryption function C(key,plaintext), it seems appropriate to define execution time requirements in terms of the number of applications of C. One application of C counts as one "unit" of computation. We shall call this "unit" the "et," (pronounced eetee) which stands for "encryption time."

Computing F requires a number of ets proportional to the length of its input. In particular, if the input is composed of 100 k bits, then F requires k-1 ets (see chapter 2).

First, we must determine the cost of computing the individual entries. The algorithm for computing $H(i,j,\underline{Y})$ from $\underline{Y}$ does a tree traversal of the subtree whose leaves are Y_i, Y_{i+1}, Y_{i+2}, ... Y_j. At each non-leaf node in this traversal it does 1 et of computation (one application of F to a 200-bit argument). A tree with j-i+1 leaves has j-i non-leaf nodes, i.e., j-i nodes internal to the tree. For example, a tree with 8 leaves has 4 + 3 + 2 + 1 = 10 internal nodes. Because there are j-i non-leaf nodes, computing $H(i,j,\underline{Y})$ requires j-i ets, excluding the leaves. The computations required to regenerate a leaf (using a truly random seed in a pseudo random number generator) will be fixed and finite. Let r be the (fixed) number of ets required to regenerate a leaf. There are (j-i+1) leaves, so the overall cost of computing $H(i,j,\underline{Y})$ is $(j-i) + (j-i+1) \cdot r$ ets. In practice r will be a few hundred, so we can approximate this by $(j-i+1) \cdot r$ ets.

We can now approximate the cost of computing each entry in table 2. There are n entries which require about r ets, n/2 entries which require about 2 r ets, n/4 entries which require about 4 r ets, and n/8 entries which require about 8 r ets. This means that the total cost of computing all entries in a single column is about 8 r ets. There are 4 columns, so the total computational effort is about $4 \cdot 8\ r = 32\ r$ ets. In general, the computational effort required to compute table 2 will be $n \cdot (1 + \log_2 n) \cdot r$ ets. This is because computing all the entries in each column will require $n \cdot r$ ets, and there are $1 + \log_2 n$ columns.

This result implies that an algorithm which sequentially generated the authentication paths would require an average of about

$$r \cdot \log_2 n \qquad (5.1)$$

ets per path, where r is a constant representing the number of ets required to regenerate a leaf. This is quite reasonable.

Although the time required to generate each authentication path is small, we must also insure that the space required is small and that the computational burden is smoothly distributed as a function of time. We can do this by again looking at table 2. As we sequentially generate the authentication paths, we will sequentially go through the entries in a column. This implies that at any point in time there are only two entries in a column of any interest to us: the entry needed in the current authentication path, and the entry immediately following it. We must know the entry in the current authentication path, for without it, we could not generate that path. At some point, we will need the next entry in the column to generate the next authentication path. Because it might require a great deal of effort to compute the next entry (e.g. to compute $H(1,4,\underline{Y})$), we need to compute it incrementally, and to begin computing it well in advance of the time we will actually require it to generate an authentication path.

As an example, $H(5,8,\underline{Y})$ is required in the authentication paths for Y_1, Y_2, Y_3, and Y_4 while $H(1,4,\underline{Y})$ is required in the paths for Y_5, Y_6, Y_7, and Y_8. The values of H() for the first authentication path must be precomputed with some delay (discussed below). Once this precomputation is complete, the succeeding values of H() required in succeeding authentication paths must be incrementally computed. As we generate the first 4 authentication paths, we must be continuously computing $H(1,4,\underline{Y})$ even though it is not needed until we reach Y_5. If we waited until time 5 to start computing it, it would take about 4 r ets to compute and entail some delay. By computing it during times 1 through 4, a processor capable of only r ets/unit time is needed. In general, if the tree is of depth k it will take $2^{k-1} \cdot r$ ets to compute the second element in the second column, but there are 2^{k-1} time units in which to compute it, again requiring a processor capable of only r ets/unit time.

Similarly, we must start computing the second element in the third column, $H(1,2,\underline{Y})$, when we generate the first authentication path. It takes about 2 r ets to compute this element (2^{k-2} in general), but there are 2 time units (2^{k-2} in general), in which to do this, so the processor for computing entries in the third column also needs to operate at only r ets/unit time. It is seen that it takes $2^{k-i-1} \cdot r$ ets to compute the next entry in the ith column and that there are 2^{k-i-1} time units in which to do this. Thus, only one processor is needed per column ($\log_2 n$ in all), and each processor need operate at only r ets/unit time.

If we assume a convenient block size (of 100 bits) and if we ignore constant factors, then the memory required by this method can be computed. We can first determine the memory required by the computations in each column, and then sum over all $\log_2 n$ columns. We must have one block to store the current entry in the column. We must also have enough memory to compute the next entry in the column. The memory required while computing $H(i,j,\underline{Y})$ is $1 + \log_2 (j-i+1)$ blocks. This assumes a straightforward recursive algorithm whose maximum stack depth will be $1 + \log_2 (j-i+1)$. The memory required to recompute a leaf (to recompute $H(i,i,\underline{Y})$) is ignored because it is small (a few blocks), constant, and the same memory can be shared by all the columns. Representing the memory requirements of H() in a new table in the same format as table 2 gives table 3:

leaf	memory required to compute entries in authentication path (in blocks)			
Y_1	4	3	2	1
Y_2				1
Y_3			2	1
Y_4				1
Y_5		3	2	1
Y_6				1
Y_7			2	1
Y_8				1

TABLE 3

Table 3 shows the memory required to compute each entry in table 2. Clearly the memory required to compute $H(i,i,\underline{Y})$ is 1. The memory required to compute $H(1,2,\underline{Y}) = 1 +$ the memory required to compute $H(1,1,\underline{Y})$ since we first compute $H(1,1,\underline{Y})$ and must remember it to compute $H(1,2,\underline{Y})$. Similarly, to compute $H(1,2^t,\underline{Y})$ requires one more memory location than was needed for $H(1,2^{t-1},\underline{Y})$. The memory required for each column will be about the memory required during the computation of a single entry in the column because once an entry is computed, the memory is available to compute the next entry and the old entry is discarded after use. This means the total memory required will be about: $3 + 2 + 1 = 6$ blocks. (This assumes we do not recompute $H(1,8,\underline{Y})$).

For n in general, there are $\log_2 n$ columns and each column requires, on an average, $(\log_2 n)/2$ blocks so the total memory required will be on the order of:

$$(\log_2 n)^2/2 \text{ blocks}$$

This means that the memory required when $n = 2^{20}$ (1,048,576) is about $20 \cdot 20/2 = 200$ blocks. For 100 bit blocks, this means 20 kilobits, or 2.5 kilobytes. Other overhead might amount to 2 or 3 kilobytes, giving an algorithm which requires 5 or 6 kilobytes of memory, in total.

Readers interested in implementing this technique can use the following program, written in a Pascal-like language with two multiprocessing primitives added:

1.) While <condition> wait

2.) Fork <statement>

In addition, the function "MakeY(i)" will regenerate the value of Y_i from the truly random seed.

```
Declare flag:  array[0..$\log_2$(n)-1] of integer;
        AP:    array[0..$\log_2$(n)-1] of block;
               (* AP -- Authentication Path *)
Procedure Gen(i);
Begin
  For j:= 1 to n step $2^{i+1}$ Do
```

```
   Begin
     Emit(i,H(j+$2^i$,j+$2^{i+1}$-1));
     Emit(i,H(j,j+$2^i$-1));
   End;
End;



Procedure Emit(i,value);
Begin
   While flag[i] ≠ 0 wait;
   AP[i]:= value;
   flag[i]:= $2^i$;
End;



Procedure H(a,b);
Begin
   If a = b Return( F(MakeY(a)))
   Else
   Return(F(H(a,(a+b-1)/2),H((a+b+1)/2,b));
    (*  Note that F should be parameterized by
        the user's name and by a and b.  If
        this is not done, Y must be made larger
        to assure security (see chapter II).    *)
End;
```

```
(*  The main program  *)
Begin
   For i := 0 to $\log_2(n)-1$ Do
    Begin
     flag[i]:= 0;
     Fork Gen(i);
    End;
   For j:= 1 to n Do
   Begin
    Print("Authentication Path ", j, " is:");
    For k := 0 to $\log_2(n)-1$ Do
     Begin
      While flag[k] = 0 wait;
      Print(AP[k]);
      flag[k]:= flag[k]-1;
     End;
   End;
End;
```

The general structure of this program is simple: the main routine forks off $\log_2 n$ processes to deal with the $\log_2 n$ columns. Then it prints each authentication path by sequentially printing an output from each process. The major omission in this program is the rate at which each process does its computations. It should be clear from the previous discussion that each process has adequate time to compute its next output.

There are three major ways of improving this algorithm. First, each process is completely independent of the other processes. However, separate processes often require the same intermediate values of H(), and could compute these values once and share the result.

Second, values of H() are discarded after use, and must be recomputed later when needed. While saving all values of H() takes too much memory, saving some values can reduce the computation time *and also* reduce memory requirements. The reduction in memory is because of the savings in memory when the saved value is not recomputed. Recomputing a value requires memory for the computation, while saving the value requires only a single block.

Finally, the memory requirements can be reduced by carefully scheduling the processes. While it is true that each process requires about $\log_2 n$ blocks of memory, this is a maximum requirement, not a typical requirement. By speeding up the execution of a process when it is using a lot of memory, and then slowing it down when it is using little memory, the average memory requirement of a process (measured in block-seconds) can be greatly reduced. By scheduling the processes so that the peak memory requirements of one process coincide with the minimum memory requirements of other processes, the total memory required can be reduced.

All three approaches deserve more careful study because the potential savings in time and space might be large. Even without such improvements the technique is completely practical.

Before the time requirements of the algorithm can be fully analyzed, a description of MakeY is needed: i.e., we must determine r in equation (5.1). If we assume that the improved version of the Lamport-Diffie algorithm is used, then MakeY must generate pseudo-random X_i vectors, from which Y_i vectors can then be generated. If the one way hashed messages are all 100 bits long, then the X_i vectors will have $100 + \log_2 100 = 107$ elements.

The X_i vectors can be generated using a conventional cipher, C(key,plaintext). A single 200 bit secret key is required as the "seed" of the pseudo-random process which generates the X_i vectors. The output of C is always 100 bits, and the input must be 100 bits or fewer, (if fewer, 0's are appended). We can now define $x_{i,j}$ as

$$x_{i,j} = C(\text{seedkey},\langle i,j\rangle)$$

where "seedkey" is the 200 bit secret and truly random key used as the "seed" of this somewhat unconventional pseudo-random number generator. The subscript i is in the range 1 to n, while the subscript j is in the range 1 to 107. There are n possible messages, each 100 bits in length. Each X_i is a vector $x_{i,1}$, $x_{i,2}$, ... $x_{i,107}$.

Determining any $x_{i,j}$ knowing some of the other $x_{i,j}$'s is equivalent to the problem of cryptanalyzing C under a known plaintext attack. If C is a good encryption function, it will not be possible to determine any of the $x_{i,j}$ without already knowing the key. The secret vectors X_i are therefore safe.

We know that $y_{i,j} = F(x_{i,j})$, and that $H(i,i,\underline{Y}) = F(Y_i) = F(y_{i,1}, y_{i,2}, y_{i,3}, \ldots y_{i,107})$. The cost of computing $F(Y_i)$ is 106 ets, because Y_i is 107·100 bits long. The total effort to compute $H(i,i,\underline{Y})$ is the effort to generate the elements of the X_i vector, plus the effort to compute $F(x_{i,1})$, $F(x_{i,2})$, ... $F(x_{i,n})$, plus the effort to compute $F(Y_i)$. This is 107 ets to compute the X_i vector, 107 ets to compute the Y_i vector, and 106 ets to compute $F(Y_i) = H(i,i,\underline{Y})$. This is a total of 320 ets to regenerate each leaf in the authentication tree.

Using equation (5.1), we know that the cost per authentication path is 320 $\log_2 n$ ets. For $n = 2^{20}$, this is 6400 ets. To generate authentication paths at the rate of one per second implies 1 et is about 160 microseconds. While easily done in hardware, this speed is difficult to attain in software on current computers. Reducing the number of ets per authentication path is a worthwhile goal. This can effectively be done by reducing either the cost of computing $H(i,i,\underline{Y})$, or by reducing the number of times that $H(i,i,\underline{Y})$ has to be computed.

As mentioned earlier, keeping previously computed values of H() rather than discarding them and sharing commonly used values of H() among the $\log_2 n$ processes reduces the cost of computing each authentication path. In fact, a reduction from over 6000 ets to about 1300 ets (for $n = 2^{20}$) can be attained. (To put this in perspective, MakeY *requires* 320 ets and *must* be executed at least once per authentication path. Therefore, 320 ets is the absolute minimum that can be attained without modifying MakeY.) This means the path regeneration algorithm can run in reasonable time (a few seconds) even when the underlying encryption function, C, is implemented in software.

6. Conclusion

Digital signature systems not requiring public key cryptosystems are possible and desirable because they are easy to certify. Such a system was described which had modest space and time requirements and a signature size of about 15 kilobits. The method described can be implemented at once, with no delay due to certification.

VI. THE TRAPDOOR KNAPSACK

1. Introduction

This chapter describes a public key cryptosystem based on the knapsack problem. Given a one-dimensional knapsack of length S and n rods of lengths $a_1, a_2, \ldots, a_n$, the "knapsack problem" is to find a subset of the rods which exactly fills the knapsack, if such a subset exists. Equivalently, find a binary n-vector $\underline{x}$ such that $S = \underline{a} * \underline{x}$, if such an $\underline{x}$ exists, (* applied to vectors denotes dot product.)

A supposed solution $\underline{x}$ is easily checked in at most n additions, but finding a solution is believed to require a number of operations which grows exponentially in n. Exhaustive, trial and error search over all 2^n possible $\underline{x}$'s is computationally infeasible if n is larger than one or two hundred. The best published method for solving knapsacks of the form considered here requires $2^{n/2}$ complexity both in time and memory [10]. In addition, Schroeppel [33] has devised an algorithm which takes $O(2^{n/2})$ time and $O(2^{n/4})$ space. Theory supports the belief that the knapsack problem is hard because it is an NP-complete problem [footnote page 81], and is therefore one of the most difficult computational problems of a cryptographic nature [1 pp 363-404] [6]. Its degree of difficulty, however, is crucially dependent on the choice of $\underline{a}$. If $\underline{a} = (1,2,4,\ldots 2^{n-1})$, then solving for $\underline{x}$ is equivalent to finding the binary representation of S. Somewhat less trivially, if for all i,

$$a_i > \sum_{j=1}^{i-1} a_j \qquad (6.1)$$

then $\underline{x}$ is also easily found: $x_n = 1$ if and only if $S \geq a_n$ and, for $i = n-1, n-2, \ldots 1$, $x_i = 1$ if and only if

$$S - \sum_{j=i+1}^{n} x_j \cdot a_j \geq a_i \qquad (6.2)$$

While the theory of NP-complete problems and these examples demonstrate that the knapsack problem is only difficult from a worst case point of view, it is probably true that choosing the a_i independently and uniformly from the integers between 1 and 2^n generates a difficult problem with probability tending to one as n tends to infinity. While several efficient algorithms exist for solving the knapsack problem under special conditions [10], [11], [16], none of these special conditions is applicable to trap door knapsacks generated as suggested in this chapter.

A trap door knapsack [6] is one in which careful choice of $\underline{a}$ allows the designer to easily solve for any $\underline{x}$, but which prevents anyone else from finding the solution. We will describe one method for constructing a trap door knapsack, and another (multiplicative) method due to Hellman is described in [21]. We first indicate how knapsacks can be used to hide information. Each user I in a system generates a trap door knapsack vector, $\underline{a}(I)$, and places it in a public file with his name and address. When someone wishes to send the binary information vector $\underline{x}$ to the Ith user, he sends $S = \underline{x} * \underline{a}(I)$. The intended recipient can recover $\underline{x}$ from S but no one else can. Section 5 shows how trap door knapsacks can be used to generate electronic signatures and receipts [6].

Before proceeding, a word of caution is in order. First, as is always the case in computational cryptography, we cannot yet prove that the systems described in this chapter are secure. For brevity, however, we will not continue to repeat this. Second, the trap door knapsacks described in this chapter form a proper subset of all possible knapsacks and their solutions are therefore not necessarily as difficult as for the hardest knapsacks, and it is the hardest knapsacks with which NP theory is concerned.

2. A Method for Constructing Trap Door Knapsacks

The designer chooses two large numbers, m and w, such that w is invertible modulo m (equivalently GCD(w,m)=1). He selects a knapsack vector, $\underline{a}'$ which satisfies (6.1) and therefore allows easy solution of $S' = \underline{a}' * \underline{x}$. He then transforms the easily solved knapsack vector $\underline{a}'$ into a trap door knapsack vector $\underline{a}$ via the relation

$$a_i = w \cdot a'_i \mod m \qquad (6.3)$$

The a_i are pseudo-randomly distributed and it therefore appears that anyone who knows $\underline{a}$, but not w and m, would have great difficulty in solving a knapsack problem involving $\underline{a}$. The designer, on the other hand, can easily compute

$$\begin{aligned} S' &= w^{-1} \cdot S \mod m && (6.4) \\ &= w^{-1} \cdot \sum x_i \cdot a_i \mod m && (6.5) \\ &= w^{-1} \cdot \sum x_i \cdot w \cdot a'_i \mod m && (6.6) \\ &= \sum x_i \cdot a'_i \mod m && (6.7) \end{aligned}$$

If m is chosen so that

$$m > \sum a'_i \qquad (6.8)$$

then (6.7) implies that S' is equal to $\sum x_i \cdot a'_i$ in integer arithmetic as well as mod m. This knapsack is easily solved for $\underline{x}$, which is also the solution to the apparently difficult, but trap door, knapsack problem $S = \underline{a} * \underline{x}$.

To help make these ideas clearer, we give a small example with n = 5. Taking m = 8443, $\underline{a}'$ = (171,196,457,1191,2410), and w = 2550 (so w^{-1} = 3950), then $\underline{a}$ = (5457,1663,216,6013,7439).

Given S = 1663 + 6013 + 7439 = 15115, the designer computes

$$S' = w^{-1} \cdot S \mod m \qquad (6.9)$$

$$= 3950 \cdot 15115 \mod 8443 \qquad (6.10)$$

$$= 3797 \qquad (6.11)$$

Because $S' > a'_5$, he determines that $x_5 = 1$. Then, using (6.2) for the $\underline{a}'$ vector, he determines that $x_4 = 1$, $x_3 = 0$, $x_2 = 1$, $x_1 = 0$, which is also the correct solution to $S = \underline{a} * \underline{x}$.

Anyone else who does not know m, $\underline{a}'$, and w has great difficulty in solving for $\underline{x}$ in $S = \underline{a} * \underline{x}$ even though the general method used for generating the trap door knapsack vector $\underline{a}$ must be public. His task can be further complicated by scrambling the order of the a_i, and by adding different random multiples of m to each of the a_i.

The example given was extremely small in size and only intended to illustrate the technique. Using n = 100, which is the bottom end of the usable range for secure systems, m can be chosen uniformly from the numbers between $2^{201} + 1$ and $2^{202} - 1$; a'_1 can be chosen uniformly from the range $[1, 2^{100}]$; a'_2 can be chosen uniformly from $[2^{100} + 1, 2 \cdot 2^{100}]$; a'_3 can be chosen uniformly from $[3 \cdot 2^{100}+1, 4 \cdot 2^{100}]$; ... a'_i can be chosen uniformly from $[(2^{i-1}-1) \cdot 2^{100}+1, 2^{i-1} \cdot 2^{100})]$; ... a'_{100} can be chosen uniformly from $[(2^{99}-1) \cdot 2^{100} +1, 2^{99} \cdot 2^{100}]$; and w can be chosen uniformly from [2, m-2] and then repeatedly divided by the greatest common divisor of w and m, to yield the value of w that is actually used.

These choices ensure that (6.8) is met and that an opponent has at least 2^{100} possibilities for each parameter and hence cannot search over even one of them. Note that each a_i will be pseudo-randomly distributed between 1 and m-1 and hence will require a 202 bit representation. S will require a 209 bit representation, so there is a 2.09:1 data expansion from $\underline{x}$ to S.

3. An Iterative Method

This section discusses techniques for improving the security and utility of the basic methods.

In the first method we transformed a hard and apparently very difficult knapsack problem, $\underline{a}$, into a very simple and easily solved knapsack problem, $\underline{a}'$, by means of the transformation:

$$a'_i = w^{-1} \cdot a_i \mod m \tag{6.15}$$

We could solve a knapsack involving $\underline{a}$ because we could solve a knapsack involving $\underline{a}'$. Notice, though, that it does not matter _why_ we are able to solve knapsacks involving $\underline{a}'$, all that matters is that we _can_ solve them. Rather than requiring that $\underline{a}'$ satisfy (6.1), we could require that $\underline{a}'$ be transformable into a new problem, $\underline{a}''$, by the transformation:

$$a''_i = w'^{-1} \cdot a'_i \mod m' \tag{6.16}$$

where the new problem, $\underline{a}''$, satisfies (6.1), or is otherwise easy to solve. Having done the transformation twice, there is no problem in doing it a third time. That is, we select an $\underline{a}''$ which is easy to solve, not because it satisfies (6.1), but because it can be transformed into $\underline{a}'''$, which _is_ easy to solve, by:

$$\underline{a}'''_i = w''^{-1} \cdot \underline{a}''_i \mod m'' \tag{6.17}$$

It is clear that we can repeat this process as often as we wish.

With each successive transformation, the structure in the publicly known vector, $\underline{a}$, becomes more and more obscure. In essence, we are encrypting the simple knapsack vector by the repeated application of a transformation which preserves the basic structure of the problem. The final result $\underline{a}$ appears to be a collection of random numbers. The fact that the problem can be easily solved has been totally obscured.

The effect of repeating the process several times is very different from that obtained with certain ciphers, such as a simple substitution. A simple substitution cipher is not strengthened by repetition because the composition of two substitution ciphers is yet another substitution cipher. The (w,m) transformations do not have this closure property. The following example shows that the repetition of two (w,m) transforms is not in general equivalent to a single (w,m) transform.

If $w = 3$, $m = 89$, $w' = 17$, $m' = 47$, and $\underline{a}'' = (5,10,20)$ then $\underline{a}' = (38,29,11)$ and $\underline{a} = (25,87,33)$. Assume there exists $\hat{w}$ and $\hat{m}$ such that

$$\underline{a} = \hat{w} \cdot \underline{a}'' \bmod \hat{m} \qquad (6.18)$$

Then $a_1 = 25$ and $a''_1 = 5$ implies that

$$25 = \hat{w} \cdot 5 \bmod \hat{m} \qquad (6.19)$$

From this we have

$$2 \cdot 25 = \hat{w} \cdot 2 \cdot 5 \bmod \hat{m} \qquad (6.20)$$

or

$$50 = \hat{w} \cdot 10 \bmod \hat{m} \qquad (6.21)$$

But now the relation $a_2 = 87$ and $a''_2 = 10$ implies that

$$87 = \hat{w} \cdot 10 \bmod \hat{m} \qquad (6.22)$$

so $87 = 50 \bmod \hat{m}$, or $37 = 0 \bmod \hat{m}$, which implies that $\hat{m} = 37$. Equation (6.19) then becomes

$$25 = \hat{m} \cdot 5 \bmod 37 \qquad (6.23)$$

so $\hat{w} = 5$. But if $\hat{w} = 5$, and $\hat{m} = 37$, then equation (6.18) for $a_3 = 33$ and $a''_3 = 20$ becomes

$$33 = 5 \cdot 20 \bmod 37 \qquad (6.24)$$

or $33 = 26 \bmod 37$, a contradiction. We conclude that no such $\hat{w}$ and $\hat{m}$ can exist.

The original, easy to solve knapsack vector can meet any condition, such as (6.1) which guarantees that it is easy to solve.

It is important to consider the rate of growth of $\underline{a}$, because this rate determines the data expansion involved in transmitting the n bit vector $\underline{x}$ as the larger quantity S. The rate of growth depends on the method of selecting the numbers but, with n = 100, each a_i need be at most 7 bits larger than the corresponding a'_i, each a'_i need be at most 7 bits larger than a''_i, etc. etc. Each successive stage of the transformation need increase the size of the problem by only a small, fixed amount. Repeating the transformation 20 times will add at most 140 bits to each a_i. If each a_i is 200 bits long to begin with, then they need only be 340 bits long after 20 stages, and S is representable in 347 bits. The data expansion is then only 3.47:1.

4. Compressing The Public File

As described above, the Ith user must place his trap door knapsack vector, a(I), in a public file. The Jth user can then look up a(I), and send a message x to I, hidden as S = a(I) * x. To avoid storing the rather large vector a(I), J could ask I to transmit a(I) to him. But, unless J has some method for testing a(I), user K might fool J by sending him a(K), and saying it was a(I). J would then mistakenly tell all his secrets to K. A method is needed for J to convince himself that he was really sent a(I). With a public file, each user can make one personal appearance when depositing his vector and, after so identifying himself to the system, he could identify (authenticate) himself to any user by his ability to decipher messages hidden with his vector. The file itself must be protected, but this is relatively easy because only write protection is needed.

To preserve this authentication benefit of the public file, but to reduce its size(20 or more kilobits per user) we suggest storing a 100 bit one-way hash total, h[a(I)], instead of a(I) itself. When J receives a(I) from I he computes h[a(I)] and checks this against I's value stored in the public file. The hash function, h, must be a one-way function [6], [38], [7], [28] so that K cannot generate a new vector a(K) such that h[a(K)] = h[(a(I)], without having to perform a computationally impossible feat.

Allowing 200 bits for storing the user's name and address, (or "phone number"), the public file now contains 300 bits, instead of over 20 kilobits, per user. A system with a million users requires a 300 million bit, instead of a 20 billion bit, public file. Transmission costs are comparable for both implementations.

A 100 bit number can be coded as 20 alphanumeric characters, which is small enough to fit in a phone book. A typical entry would look like this:

Joe Smith......497-1573

KSDJR E6K65 3GFVM OMK4K

The extra entry, KSDJR E6K65 3GFVM OMK4K, is the one-way hash total of Smith's trap door knapsack vector, $\underline{a}$(Smith). With this information, we can call up Smith, and hold a secure conversation with him, which no one else can understand. We do not need to have met Smith previously to know we are talking with him or for him to know he is talking with us.

Transmitting 20 kilobits on a high speed, 50 kbps link, takes 0.4 second, but on a low speed, 300 bps link, it takes over a minute. The transmission time can be reduced by a factor of 5, to about 4 kilobits, which takes less than 15 seconds to transmit at 300 bps, by cutting the number of a_i to n=20. The vector $\underline{x}$, however, now has only 20 binary elements, which is small enough to allow solution by exhaustive search. To maintain security, the information in the $\underline{x}$ vector must be increased to about 100 bits, while keeping n = 20. This can be done by allowing each element, x_i, to take on values in the set {0,1,2,3,...,31}, instead of just in {0,1}. Specifying each x_i takes 5 bits, and specifying the whole vector $\underline{x}$ takes 100 bits. Equation (6.1) must now be modified to

$$a_i > 31 \cdot \sum_{j=1}^{i-1} a_j \qquad (6.25)$$

If n is reduced to 1 and the single element of the $\underline{x}$ vector assumes a value in $\{0,1,2,\ldots\ 2^{100}-1\}$, then the system is easily broken because

$$x = S/a \qquad (6.26)$$

When n=2, the system can also be broken easily, by an algorithm similar in spirit to the greatest common divisor algorithm. It seems that small values of n weaken the system, and further research is needed to determine how small n can be, while still preserving security. The value n = 20 suggested above must be treated with suspicion until an adequate certificational study is conducted.

5. Signatures

As discussed in [6], the need for a digital equivalent of a written signature is a major barrier to the replacement of physical mail by teleprocessing systems. Usual digital authenticators protect against third party forgeries, but cannot be used to settle disputes between the transmitter and receiver as to what message, if any, was sent. A true digital signature allows the recipient to prove that a particular message was sent to him by a particular person. Obviously, it must be impossible for the recipient to alter the contents of the message and generate the corresponding signature, but it must be easy for him to check the validity of a signature for any message from any user. A digital signature can also be used to generate receipts. The recipient signs a message saying, "I have received the following message: TEXT." This section describes how trap door knapsacks might be used to generate such signatures and receipts.

If every S in some large fixed range had an inverse image $\underline{x}$ then it could be used to provide signatures. When the Ith user wanted to send the message m he would compute and transmit $\underline{x}$ such that $\underline{a}(I) * \underline{x} = m$. The recipient could easily compute m from $\underline{x}$ and, by checking a date/time field (or some other redundancy in m), determine that the message was authentic. Because the recipient could not generate such an $\underline{x}$ he saves $\underline{x}$ as proof that the Ith user sent him the message m.

This method of generating signatures can be modified to work when the density of solutions (the fraction of S between 0 and $\sum a_i$ which have solutions to $\underline{x} * \underline{a} = S$) is less than 1, provided it is not too small. The message m is sent in plaintext form, or encrypted if eavesdropping is a threat, and a sequence of one-way functions [6], [38], [7], [28] $y_1 = F_1(m)$, $y_2 = F_2(m)$,... are computed. The transmitter then seeks inverse images for $y_1, y_2, \ldots$ until one is found and appends the corresponding $\underline{x}$ to m as a signature. The receiver computes $y = \underline{a} * \underline{x}$ and checks that y is equal to y_k with k not too large, for example at most 10 times the expected value of k.

The sequence of functions $F_i(\cdot)$ can be as simple as;

$$F_i(m) = F(m) + i \qquad (6.20a)$$

or

$$F_i(m) = F(m+i) \qquad (6.20b)$$

where $F(\cdot)$ is a one-way function. It is necessary that the range of $F(\cdot)$ have at least 2^{100} values to foil trial and error attempts at forgery. If the message is much longer than 100 bits, the expansion caused by the addition of a 100 bit authentication field is unimportant.

If the trap door knapsack vector were generated as suggested at the end of section 2, the solution density would be less than $1/(2^{100})$, and over 2^{100} y_k would have to be tried, on the average, before one with a solution was found. It is possible, however, to use the iterative method of section 3 to obtain a solution density of approximately $1/(10^4)$ with two iterations or $1/(10^6)$ with three iterations, when $n = 100$. First, a knapsack vector $\underline{a}''$ with a solution density near 1 is selected. If $\underline{a}'' = (1,2,4,8,\ldots,2^{99})$ then the solution density is 1, but increasing some of the larger a''_i need not greatly reduce the solution density. For example, (1,2,4,8,17,35,68,142) has a solution density of .92 and still satisfies (6.1). Such choices may not be necessary, but they provide an additional margin of safety at almost no additional cost.

After selecting $\underline{a}''$, parameters m' and w' are chosen such that $m' > \quad a''_i$ and $w^{-1}{}'$ exists modulo m'. The weak trap door knapsack vector

$$\underline{a}' = w' \cdot \underline{a}'' \qquad \text{mod } m' \qquad (6.27)$$

is then computed. New parameters $m > \sum \underline{a}'_i$ and w (with w^{-1} existing mod m) are chosen, and the more secure trap door knapsack vector

$$\underline{a} = w \cdot \underline{a}' \qquad \text{mod } m \qquad (6.28)$$

is computed. The process can be iterated more than twice to obtain the final vector, $\underline{a}$, but the solution density typically decreases by a factor of $n/2$ with each iteration. When used for hiding information this decrease is of little importance, but when used for signatures, several iterations are all that can be afforded because of the need for a high solution density. With so few iterations, it is possible for two adjacent a_i to be in the same ratio (usually 2:1) as they were in the $\underline{a}$ vector. This weakness can be overcome by adding multiples of m' (or m) to a subset of the a'_i (or a_i) which suffer from this problem. This decreases the solution density somewhat, and accounts for our $1/(10^4)$ and $1/(10^6)$ estimates for two and three iterations when n = 100.

A small example is again helpful in illustrating the method. Starting with

$$\underline{a}'' = (1,2,4,8,17,35,68,142) \qquad (6.29)$$

whose components sum to 277, we choose m' = 291 and w' = 176 (w'^{-1} = 167), resulting in

$$\underline{a}' = (176,61,122,244,82,49,37,257) \qquad (6.30)$$

The second, third and fourth components are in the ratio of 2:1 which can be hidden by adding m' to the third component to obtain the new vector

$$\underline{a}' = (176,61,413,244,82,49,37,257) \qquad (6.31)$$

whose components sum to 1319. Choosing m = 1343, w = 498 (w^{-1} = 925) yields

$$\underline{a} = (353,832,195,642,546,228,967,401) \qquad (6.32)$$

whose components sum to 4164. The density of solutions using $\underline{a}$ is 256/4164 = .061 so approximately 16 attempts are needed, on the average, to obtain a signature. This agrees well with the estimated range of $n^2/4$ = 16 to n^2 = 64.

The density of solutions can be increased by restricting the y_k to lie near the middle of the range $(0, \sum \underline{a}_i)$, say between 1000 and 3000 in this example. The law of large numbers indicates that for most $\underline{x}$, the sum $\underline{a} * \underline{x}$ will lie in this range.

THE TRAPDOOR KNAPSACK

Shamir [43] has developed a different method of using the knapsack problem to obtain signatures.

6. Discussion

We have shown that it is possible to construct trap door knapsack problems and that information and signatures can be hidden in them for transmission over an insecure channel. Conventional cryptographic systems also can hide information and authenticators during transmission over an insecure channel, but have the disadvantage that first a "key" must be exchanged via courier service or some other secure means. Also, in conventional cryptography, the authenticator only prevents third party forgeries and cannot be used to settle disputes between the transmitter and receiver as to whether a message was actually sent.

We have not proved that it is computationally difficult for an opponent who does not know the trap information to solve the problem. Indeed, proofs of security are not yet available for normal cryptographic systems, and even the general knapsack problem has not been proved difficult to solve. The theory of computational complexity has not yet reached the level of development where such proofs are feasible. The best published algorithm for solving the knapsack problem is exponential, taking $O(2^{n/2})$ time and space [10]. Schroeppel [33,unpublished] has devised an algorithm which takes $O(2^{n/2})$ time and $O(2^{n/4})$ space.

Faith in the security of these systems must therefore rest on intuition and on the failure of concerted attempts to break them.

Attempts to break the system can start with simplified problems (e.g. assuming m is known) (see chapter VII section 5.) If even the most favored of certificational attacks is unsuccessful, then there is a margin of safety against cleverer, wealthier, or luckier opponents. Or, if the favored attack is successful, it helps establish where the security really must reside.

As noted, the techniques suggested in this chapter generalize to x_i in the set {0,1,2,3,...,N}. The advantages and weaknesses of such systems deserve further study.

Recently, Rivest, Shamir, and Adleman [31] have proposed another public key cryptosystem, which yields signatures more directly because the density of solutions in their problem is 1. Their system also requires a smaller key (apparently 600 bits versus 20 kilobits); but is significantly slower. Neither system's security has been adequately established but, when iterated, the trap door knapsack appears less likely to possess a chink in its armor. When used for obtaining signatures, the trap door knapsack appears to be the weaker of the two. Both public key systems clearly need further certification and study.

7. Footnote

Other definitions of the knapsack problem exist in the literature [10], [11], [31]. The definition used here is adapted from Karp [13]. To be precise, Karp's knapsack problem is to determine whether or not a solution $\underline{x}$ exists, while the corresponding cryptographic problem is to determine what $\underline{x}$ is, given that it exists. The cryptographic problem is not NP-complete, but is just as hard as the corresponding NP-complete problem. If there is an algorithm for solving the cryptographic problem in time T(n), i.e., for determining $\underline{x}$ given that it exists, then we can determine whether or not an $\underline{x}$ exists in time T(n), i.e., solve the corresponding NP-complete problem. If the algorithm determines $\underline{x}$ in time T(n), then some $\underline{x}$ exists. If the algorithm does not determine $\underline{x}$ in time T(n), or determines an incorrect $\underline{x}$ --which is easily checked-- then no such $\underline{x}$ exists.

VII. HOW SECURE IS THE TRAPDOOR KNAPSACK?

1. Introduction

Although the knapsack problem can be used as the basis of a public key cryptosystem, a closer investigation of its security is needed in order to select the size and exact type of trapdoor knapsack to use. This chapter is a study of the knapsack problem from several perspectives, in an attempt to determine how secure it is, and what type of trapdoor knapsack is most suitable for actual implementation. Besides discussing the best known algorithms for solving the knapsack problem, and for solving special cases of the trapdoor knapsack problem, it also gives direct reductions of Boolean circuits to the knapsack problem. (The problem of Boolean circuits has already been shown to be NP-complete. As a byproduct of this reduction we show that if DES [24] is secure, so are general knapsacks with n = 10,000.) The proofs presented are accessible to anyone with a modest knowledge of Boolean circuit theory, and require no theoretical background, (in particular, no knowledge of NP complete problems is assumed.) They can serve to demystify complexity theory as applied to the knapsack problem.

There are two aspects to the complexity of the knapsack problem. First, does there exist a method of solving the trapdoor knapsack which cannot solve the general knapsack problem? That is, is the trapdoor knapsack problem easy to solve even though the general knapsack problem is NP-complete and therefore presumably hard to solve? Secondly, what is the complexity of the general knapsack problem? Do there exist unusually efficient algorithms for solving the knapsack problem? Do there exist algorithms which will rapidly solve some knapsack problems? This chapter first attempts to give a better idea of the complexity of the general knapsack problem, both by considering the most efficient algorithms known for its solution, and by examining direct reductions of Boolean circuits to the knapsack problem. It will then examine some algorithms which can efficiently break certain specialized cases of the trapdoor knapsack problem.

2. The Knapsack Problem

The knapsack problem is: given an integer S and an integer vector $\underline{a} = a_1, a_2, a_3, \ldots a_n$ find a vector $\underline{x} = x_1, x_2, \ldots x_n$ where x_i is in {0,1} such that $S = \underline{x} * \underline{a}$, (where "*" denotes dot product.)

Karp [13] showed that this problem is NP-complete. The best published algorithm for solving this problem requires $2^{n/2}$ operations and $2^{n/2}$ memory [10]. This algorithm is simple in nature: all possible sums involving $a_1, a_2, \ldots a_{n/2}$ are generated, and the list of $2^{n/2}$ possible sums is sorted. Then, all possible sums involving $a_{n/2+1}, a_{n/2+2}, \ldots a_n$ are generated, each sum is subtracted from S, and the resulting list of numbers is sorted. If a number in the first list matches a number in the second list, then a solution exists and can be readily computed. (An unpublished algorithm which runs in time $2^{n/2}$ and in space $2^{n/4}$ has been discovered by Schroeppel [33]).

The proof by Karp that this problem is NP-complete is elegant and concise, but gives little hint as to whether the best known algorithms (mentioned above) are very close to the best possible algorithms or whether much better algorithms are possible. A more fundamental problem with the use of the theory of NP complete problems is that the nondeterministic polynomial time Turing machine was reduced to the knapsack problem, but the actual complexity of a nondeterministic Turing machine has not been investigated closely enough for cryptographic purposes: in cryptography, constant factors are VERY important.

Rather than reduce a non-deterministic Turing machine to the knapsack problem, it is more appropriate to reduce Boolean circuits to the knapsack problem. The rationale for this is simple: modern cryptographic systems are actually built out of Boolean circuits, i.e., "and," "or" and "not" gates. Cryptographic systems built out of arbitrary combinations of such logical building blocks have already been certified. As a consequence, reducing Boolean circuits to the knapsack problem directly will not only let us infer that the knapsack problem is NP-complete, it will also let us draw some inferences about a lower bound on n needed for a secure system. If it is possible to embed the problem of cryptanalyzing an already certified cryptographic system into a knapsack problem with n = 10,000, then we can safely infer that there is no uniformly fast algorithm which can solve knapsack problems with n = 10,000. Furthermore, the details of the actual reduction might give us some feel for the security of the knapsack problem.

We first extend the problem of Boolean circuits by including a multi-input multi-output "gate," which is intended to model the ROM-based S-boxes (substitution boxes) found in many modern cryptographic functions. We define a "(k,m)S-box" as a device with k Boolean inputs, and m Boolean outputs, where the function which determines the output from the input is arbitrary. As an example, figure 1 shows the truth table for a (3,2)S-box.

input	output
000	01
001	11
010	00
011	10
100	11
101	11
110	01
111	00

Figure 1

The reader should notice that "and" and "or" gates are just (2,1)S-boxes, and a "not" gate is just a (1,1)S-box. These devices will therefore not be considered separately.

In order to embed a modern cryptographic function into the knapsack problem, it is sufficient if we can embed arbitrary (k,m)S-boxes and their interconnections. This can be done easily, and the (3,2)S-box of figure 1 is shown in figure 2 embedded in a knapsack problem with $n = 13$. The $\{a_i\}$ and S have been chosen so that exactly 8 $\underline{x}$ vectors satisfy $S = \underline{a} * \underline{x}$ and these 8 vectors specify the data of figure 1 under the following interpretation.

	a_1	= 000--01---------1
	a_2	= 001--11---------1
	a_3	= 010--00---------1
	a_4	= 011--10---------1
	a_5	= 100--11---------1
	a_6	= 101--11---------1
	a_7	= 110--01---------1
	a_8	= 111--00---------1
input 1	a_9	= 100--00---------0
input 2	a_{10}	= 010--00---------0
input 3	a_{11}	= 001--00---------0
output 1	a_{12}	= 000--10---------0
output 2	a_{13}	= 000--01---------0
sum	s	= 111--11---------1

Figure 2

The a_i are shown in binary, and "-" is used for zero's which are not structurally important, but which are used only for spacing and to prevent carries between the three structural fields during addition of the a_i. x_9, x_{10}, and x_{11} represent the 3 input values to the S-box in complementary notation. If

x_9 is 0, then input 1 to the S-box is a 1. If x_9 is a 1, then input 1 is a 0. Similarly for x_{10} and x_{11}. The outputs are represented, again in complementary logic, by x_{12} and x_{13}. The variables x_1 through x_8 are not interpreted, they are part of the "internal workings" of this S-box representation. The final "1" in a_1 through a_8 coupled with the requirement that S end in 1 guarantees that only one of x_1 through x_8 can be a 1, all the rest must be 0's.

Perhaps the best explanation of this S-box is to work through an example of what happens when a given 3-bit input is specified as part of the $\underline{x}$ vector, and how it eventually produces the correct 2-bit output, also as part of the $\underline{x}$ vector.

Figure 1 shows that on input 110 this S-box must produce output 01. Input 110 corresponds to $x_9 = 0$, $x_{10} = 0$, and $x_{11} = 1$, and output 01 corresponds to $x_{12} = 1$ and $x_{13} = 0$. To force a_7, which corresponds to this input-output pair, to be included in the sum we set the first three bits of S to 111. Only one of x_1 through x_8 can be a 1, and the x_i selected must make the first 3 bits of the sum equal 111. Because carries are prevented between the three structural fields, the only possible choice is $x_7 = 1$, and the rest of x_1 through x_8 are equal to 0. Now, x_7 has a two-bit "output" section, which is the correct output (01) for the original input 110. By choosing the second field in S to be 11 we force x_{12} and x_{13} to assume the correct output in complementary notation (10) in a manner analogous, but precisely reversed, from the manner in which the input selected x_7 in the first place.

The reader should note that a_1 through a_8 in figure 2 are nearly exact copies of the S-box shown in figure 1, the only change being the addition of structurally unimportant 0's, indicated with "-". The last structural field is the constant, 1. Each of a_9 through a_{11} has a single 1 bit, surrounded by 0 bits to form an "identity matrix" in the first structural field. The same is true for a_{12} through a_{13} in the second structural field.

The knapsack problem in figure 2 models the (3,2)S-box in figure 1, but the details of how to interconnect two or more S-boxes into a single circuit must still be developed. For simplicity, we do this with an example using two identical S-boxes. These can be represented by a single knapsack with $n = 26$, and of twice the "width" as the original knapsacks with six structural fields. The first three fields are used only for a_1 to a_{13} to represent the first S-box. The second three fields are only used for a_{14} through a_{26} to represent the second S-box. By including a non-structural buffer of 0's between fields three and four, any $\underline{x}$ vector which solves the knapsack problem is consistent with the input/output relation of both S-boxes. Figure 3 shows the result of applying this procedure to generate two non-interconnected S-boxes identical to the S-box of figures 1 and 2.

To interconnect an output from one S-box to an input of the other S-box, we must force the associated x_i and x_j to be equal to each other. To connect input 1 of the first S-box (a_9 in figure 3) to output 2 of the second S-box (a_{26} in figure 3) we must force x_9 to equal x_{26}. This is done by making x_9 identical to x_{26}, i.e., a_{26} is removed, and a new a'_9 is defined to be the sum of a_9 and a_{26}. This is illustrated in figure 4. In general, to connect a_i with a_j, create $a'_j = a_j + a_i$, and delete the old a_i.

As an example, if we wished to have two (3,2)S-boxes exactly like the one shown in figures 1 and 2, and to interconnect the second output of one with the first input of the other, the result would look like figure 3.

	a_{1}	=	000--01---------1¦------------------
	a_{2}	=	001--11---------1¦------------------
	a_{3}	=	010--00---------1¦------------------
	a_{4}	=	011--10---------1¦------------------
	a_{5}	=	100--11---------1¦------------------
	a_{6}	=	101--11---------1¦------------------
	a_{7}	=	110--01---------1¦------------------
	a_{8}	=	111--00---------1¦------------------
input 1	a_{9}	=	100--00---------0¦------------------
input 2	a_{10}	=	010--00---------0¦------------------
input 3	a_{11}	=	001--00---------0¦------------------
output 1	a_{12}	=	000--10---------0¦------------------
output 2	a_{13}	=	000--01---------0¦------------------
	a_{14}	=	------------------¦-000--01---------1
	a_{15}	=	------------------¦-001--11---------1
	a_{16}	=	------------------¦-010--00---------1
	a_{17}	=	------------------¦-011--10---------1
	a_{18}	=	------------------¦-100--11---------1
	a_{19}	=	------------------¦-101--11---------1
	a_{20}	=	------------------¦-110--01---------1
	a_{21}	=	------------------¦-111--00---------1
input 1	a_{22}	=	------------------¦-100--00---------0
input 2	a_{23}	=	------------------¦-010--00---------0
input 3	a_{24}	=	------------------¦-001--00---------0
output 1	a_{25}	=	------------------¦-000--10---------0
output 2	a_{26}	=	------------------¦-000--01---------0
sum	s	=	111--11----------¦-111--11---------1

Figure 3

	a_1	=	000--01---------1¦------------------
	a_2	=	001--11---------1¦------------------
	a_3	=	010--00---------1¦------------------
	a_4	=	011--10---------1¦------------------
	a_5	=	100--11---------1¦------------------
	a_6	=	101--11---------1¦------------------
	a_7	=	110--01---------1¦------------------
	a_8	=	111--00---------1¦------------------
input 1 (and also output 2)	a'_9	=	100--00---------0¦------01----------
input 2	a_{10}	=	010--00---------0¦------------------
input 3	a_{11}	=	001--00---------0¦------------------
output 1	a_{12}	=	000--10---------0¦------------------
output 2	a_{13}	=	000--01---------0¦------------------
	a_{14}	=	-----------------¦-000--01---------1
	a_{15}	=	-----------------¦-001--11---------1
	a_{16}	=	-----------------¦-010--00---------1
	a_{17}	=	-----------------¦-011--10---------1
	a_{18}	=	-----------------¦-100--11---------1
	a_{19}	=	-----------------¦-101--11---------1
	a_{20}	=	-----------------¦-110--01---------1
	a_{21}	=	-----------------¦-111--00---------1
input 1	a_{22}	=	-----------------¦-100--00---------0
input 2	a_{23}	=	-----------------¦-010--00---------0
input 3	a_{24}	=	-----------------¦-001--00---------0
output 1	a_{25}	=	-----------------¦-000--10---------0
sum	s	=	111--11----------¦-111--11---------1

Figure 4

From figure 2 it is seen that a (k,m)S-box can be imbedded in a knapsack of size 2^k+k+m. Outputs of one S-box will usually serve as inputs to other S-boxes, so the size of the knapsack which represents the interconnection of several S-boxes will be somewhat reduced from the sum of the sizes of the knapsacks representing the isolated S-boxes.

To model cryptanalysis, note that in a known plaintext attack, the key is the only free input to the circuit because the plaintext and ciphertext are fixed. (The known plaintext and ciphertext values can be obtained in the circuit by several methods, the easiest to explain is to use (1,1) S-boxes whose outputs are independent of their inputs.) Any $\underline{x}$ vector which solves the resultant knapsack problem specifies the key in complementary notation. Solving the knapsack problem is thus at least as hard as cryptanalysis of the system modeled by the Boolean circuitry.

It is now possible to embed the problem of cryptanalysis of a modern encryption function into the knapsack directly. As an example we consider embedding the National Bureau of Standards Data Encryption Standard (DES) [24]. Although there is controversy about the security of the DES [5], [15], there is little doubt that an encryption function with an equivalent complexity could be made secure. Because the embedding we describe could be used to embed any encryption function of comparable complexity into the same size knapsack problem, the security of the DES is not an issue.

We shall not consider the DES algorithm in any great detail, except to estimate the number and type of S-boxes it contains. It has 8 S-boxes, but requires 16 "rounds" or iterations. That is, the DES is a clocked sequential circuit which requires 16 clock periods to compute a result. Because we can model only combinatorial circuits, and not sequential circuits, we must "unroll" the 16 rounds into a single combinatorial circuit 16 times longer than the circuitry required for a single round. We shall therefore compute how large a knapsack would be required to model a single round of the DES algorithm, and then multiply by 16.

There are 8 (6,4)S-boxes involved in one round. In addition, there are 80 (2,1)S-boxes used for exclusive-oring. Of these, 48 are used to ex-or the key with the data, and 32 to ex-or the two halves of the data. The result is $8 \cdot (2^6 + 6 + 4) + 80 \cdot (2^2 + 2 + 1) = 1152$. To continue this for 16 rounds would require a knapsack with $n = 18432$.

This estimate is conservative (too large) for two major reasons: First, the (2,1)S-boxes are used for exclusive-or gates, and exclusive-or gates can in fact be implemented in a knapsack of size $n = 4$, rather than $n = 7$. Second, this counts the input of one S-box which is also the output of a preceding S-box twice, again inflating the size of knapsack needed. If both of these factors are taken into account, the size of the knapsack required shrinks to $n = 11{,}264$. A few more improvements can yield an n of about 10,000, but further improvements become more difficult at this point. We shall use the estimate n = 10,000 for the rest of the chapter.

The interpretation given to this fact (that the DES can be imbedded in a knapsack of size n = 10,000) is quite straightforward: should an algorithm exist which can efficiently and uniformly solve knapsack problems with n = 10,000, then the DES --and all other encryption functions of similar complexity-- can be broken. Because the second statement is highly implausible, it is also highly implausible that any fast algorithm will be found for solving arbitrary knapsacks with n=10,000.

The best known algorithm for solving the knapsack problem, coupled with the largest computer that can be imagined will be totally unable to solve knapsack problems with n = 1000. Even solution of knapsacks with n = 100 will be prohibitively expensive with today's technology.

Even if we accept n = 100 as a lower bound and n = 10,000 as an upper bound on the size of knapsack problem necessary for a secure system, there is still a great deal of difference between the two. Fortunately, the upper bound seems very loose. A knapsack with n = 10,000 can actually embed any DES-like encryption function with 16 · 8 = 128 (6,4)S-boxes, as well as some auxiliary logic.

3. Double Sum is Easy to Solve

An apparently effective method for reducing the value of n in the trapdoor knapsack (thus reducing the size of the public enciphering key a) is to allow the x_i to take on values in the range {0,1,2, ... B} instead of the range {0,1}. Starting with n = 100 and B = 1 (the usual case) and pushing this to its limit it produces n = 1 and $B = 2^{100}-1$ with

$$S = a_1 \cdot x_1$$

as the enciphering operation. Cryptanalysis is easily done by one division. The case n = 2 results in the double sum problem defined as: given integers S, a_1, a_2, and B, find integers x_1 and x_2 such that

$$S = a_1 \cdot x_1 + a_2 \cdot x_2 \qquad (7.1)$$

$$\text{and} \quad 0 \leq x_1, x_2 < B \qquad (7.2)$$

We may assume that $GCD(a_1,a_2) = 1$ since otherwise we can compute their GCD and divide a_1, a_2, and S by it to get a new problem with $GCD(a_1,a_2) = 1$. Then we can use the GCD algorithm to generate numbers y_1 and y_2 such that

$$1 = a_1 \cdot y_1 + a_2 \cdot y_2$$

Multiplying this equation by S gives

$$S = a_1 \cdot (S \cdot y_1) + a_2 \cdot (S \cdot y_2)$$

We also have

$$0 = a_1 \cdot a_2 + a_2 \cdot (-a_1)$$

If we add multiples of the second equation to the first, we will still have an equation which satisfies (7.1), and because a_1 and a_2 are relatively prime, it generates all possible solutions to (7.1).

The solution must satisfy (7.1), and x_1 can be expressed in the form

$$x_1 = k \cdot a_2 + r \qquad (7.3)$$

which gives

$$S = a_1 \cdot (k \cdot a_2 + r) + a_2 \cdot x_2 \qquad (7.4)$$

Furthermore, the value of r can be computed from

$$S \cdot y_1 = x_1 = k \cdot a_2 + r \mod a_2$$

or

$$S \cdot y_1 = r \mod a_2$$

We must satisfy $x_1 < B$. We must also satisfy (7.1) and (7.2), which implies $x_1 < (S+1)/a_1$. If $x_1 \geq (S+1)/a_1$, then $x_2 < 0$, contradicting (7.2). If we let $\min = \min(B, (S+1)/a_1)$ then these two conditions reduce to $x_1 < \min$. If we now select the largest allowable value of x_1, which will generate the smallest possible value of x_2, either it is a solution, or no solution exists. Using equation (7.3), we have

$$k \cdot a_2 + r = x_1 < \min$$

or

$$k \cdot a_2 < \min - r$$

or

$$k < (\min - r)/a_2$$

so the largest allowable value of k will be

$$k = (\min - r - 1)/a_2$$

By truncating k to an integer, we obtain an exact solution to (7.3), and generate a value for x_1.

We now compute x_2, using the already obtained value of x_1, and determine if $x_2 < B$. If $x_2 < B$, we have computed the solution. Otherwise, no solution exists.

To summarize:

```
$1 = a_1 \cdot y_1 + a_2 \cdot y_2$
compute $r = S \cdot y_1 \bmod a_2$
compute $\min = \min(b, (S+1)/a_1)$
compute $k = (\min - r - 1)/a_2$
if $k < 0$, there is no solution
truncate k
compute $x_1 = k \cdot a_2 + r$
compute $x_2 = (S - a_1 \cdot x_1) / a_2$
if $x_2 < B$ then $\underline{x}$ is the solution
   otherwise no solution exists
```

4. What Value of n is Safe?

It is clear that n = 2 is not safe. At the present time, although we do not have a fully tested general algorithm for solving problems with n = 3, the author believes that this is also not safe. It is not clear whether n = 4 is safe, but small values of n, i.e., less than 10, should certainly be avoided at the present time. The author wishes to emphasize that estimates about a "safe" value of n have a large subjective component. The only method of establishing that a particular selection of parameters for the trapdoor knapsack can be relied on to provide a high level of security is to have a certificational attack on the system by individuals skilled in cryptanalysis and the particular problem area. Closer investigation of the parameter n in the generalized knapsack seems justified before adopting a value for a particular system. (Certificational attacks should optimally include the creators of the system as one group. Those interested in initiating a serious certification of the trapdoor knapsack should contact the author.)

5. How Many Iterations of the Knapsack is Safe?

Although even the single iteration knapsack has not yet been broken, it is the author's belief that at least two or three iterations of the (w,m) transform are needed to produce a margin of safety. The author would presently feel comfortable with ten iterations, although, as mentioned before, such "feelings" should be viewed with caution. A full certificational attack by several experts would be preferable.

Work by the author, by Martin Hellman, and by Adi Shamir [unpublished] on the security of the single iteration trapdoor knapsack indicates that revealing any one of the parameters allows solution of the problem.

Taking the three cases in order, let us assume that in addition to the public vector $\underline{a}$, our opponent learns $\underline{a}'$in a single iteration trapdoor knapsack. In this case, the following equations hold:

$$a_1 \cdot w = a_1' \mod m$$
$$a_2 \cdot w = a_2' \mod m$$

which in turn implies that

$$a_1 \cdot w \cdot a_2' = a_2 \cdot w \cdot a_1' \mod m$$

which together with GCD(w,m) = 1 implies

$$a_1 \cdot a'_2 - a_2 \cdot a'_1 = k \cdot m$$

where k is an unknown integer.

The essential point of this computation is that we can easily compute a multiple of m. By repeating this trick a few times with other numbers from the trapdoor knapsack, we can compute several different multiples of m. Taking their gcd will then give us m, which can in turn be used to recover w.

If all we know is m, we can often recover w using a method devised by Shamir [42]. The basic equations behind this method are:

$$a_1 \cdot w = a'_1 \bmod m$$
$$a_2 \cdot w = a'_2 \bmod m$$

which implies

$$a_1/a_2 = a'_1/a'_2 \quad \bmod m$$

But we can compute a_1/a_2 mod m, which lets us rewrite the equation as:

$$\text{known} = a'_1/a'_2$$

or again as

$$a'_2 \cdot \text{known} = a'_1 \mod m$$

Furthermore, we know that a'_1 and a'_2 are very small compared with m. We therefore seek two numbers satisfying

$$n_1 \cdot \text{known} = n_2 \mod m$$
$$n_1, n_2 \text{ are small}$$

and hope that these two numbers are a'_1 and a'_2 (Shamir [42] has the details). In many cases, our hopes will be satisfied. In particular, random function arguments imply that the solution will be unique if length(a'_1) + length(a'_2) < length(m). Although it is fairly easy to pick m small enough to foil this particular attack, it still indicates that m should be kept secret to maintain security. Although m might not satisfy the constraint given, it might still be possible to solve for a'_1 and a'_2 using a generalized attack with the first three elements of the trapdoor knapsack.

Finally, if w^{-1} is known, this allows ready computation of $w^{-1} \cdot a_1$, $w^{-1} \cdot a_2$ and $w^{-1} \cdot a_3$. These numbers are all equal to a multiple of m plus the small numbers (relative to m) a'_1, a'_2 and a'_3. All we need do is compute a number which satisfies these conditions, and we have recovered m.

Since there are three secret parameters in the single iteration trapdoor knapsack and making any one public destroys security, conservatism dictates using the iterated trapdoor knapsack even though all three parameters are secret. Much as product ciphers can build a strong encryption function by iterating weak simple ciphers, so the iterated trapdoor knapsack builds strength by iterating the (w,m) transform.

6. Conclusion

The purpose of this chapter has been to give the reader a better idea of the security of particular trapdoor knapsack problems. It seems clear at this time that a trapdoor knapsack with n = 1000, with 100 iterations, and with each number in the a vector 5000 bits long should be totally secure. Reducing these numbers to more practical values is essential before the trapdoor knapsack is used in a real system.

VIII. AN NP-COMPLETE CONVENTIONAL CIPHER

1. Introduction

It is possible to make a conventional cipher based on the knapsack problem which is essentially NP-complete. Those knowledgeable about the theory of NP-completeness might object to the use of the term "NP-complete" in these circumstances, so the qualification "essentially" has been added. The precise distinctions are more definitional than substantive and are explained below.

It appears that the proof technique used here can be generalized to other ciphers.

2. The Basic Idea

It is possible to create a one way function based on the knapsack problem by defining $\underline{x}$ to be the input to the function, S to be the output from the function, and $\underline{a} = a_1, a_2, \ldots a_n$ as the function specification. This allows definition of the function by

$$F(\underline{x}) = S = \underline{x} * \underline{a}$$

(where * denotes dot product).

We can define a stream cipher [4] based on this:

$$C_t = P_t \oplus F(\underline{x})_t$$

and the deciphering process is just:

$$P_t = C_t \oplus F(\underline{x})_t$$

where $\oplus$ represents modulo two addition (exclusive or), $\underline{x}$ is defined to be the key and P_t, C_t, and $F(\underline{x})_t$ are all one bit quantities. C_t is the single bit of ciphertext transmitted at time t; P_t is the single bit of plaintext generated by A at time t; and $F(\underline{x})_t$ is the single bit of S needed to encrypt P_t at time t.

The advantage of this definition is that the key, $\underline{x}$, is the argument of a one way function, and should therefore be difficult to determine.

To be useful, it must be possible to send indefinite amounts of plaintext: P must be infinite. This implies S must be infinite. This in turn implies that the a_i must be infinite. (Notice that it does *not* imply $\underline{x}$ is infinite.) Infinite a_i's are trivial: they need only be generated as indefinite streams, least significant bit first. For each index i in {1,2, ... n}, A will transmit $a_{i,1}$, $a_{i,2}$, ... $a_{i,t}$ .... At time t, A will transmit n + 1 bits: $a_{1,t}$, $a_{2,t}$, $a_{3,t}$ ... $a_{n,t}$, and C_t.

A uses key $\underline{x}$ to select a subset of the a_i. A adds up the subset of the a_i to compute the sum, S. B knows $\underline{x}$, hence B can also compute S. (Everyone knows $\underline{a}$, which is public knowledge.)

Because the a_i are infinite in length, S will also be infinite in length. As the a_i are transmitted, least significant bit first, it will become possible for B to compute S, also least significant bit first. That is, S will be a bit stream, computable from the n bit streams a_1, a_2, ... a_n, and the key $\underline{x}$.

This is illustrated in figure 1.

```
time t:                ... 9 8 7 6 5 4 3 2 1

    a_1 = .... 0 1 0 0 0 1 0 1 1 0 1 1 1 0 1

    a_2 = .... 1 0 1 1 0 1 1 0 0 1 1 1 1 0 1

    a_3 = .... 1 0 1 0 1 0 1 1 1 0 0 1 0 0 0

     .

     .

     .

    a_n = .... 0 0 1 0 0 1 1 1 0 1 0 0 0 1 1

    S   = .... 0 1 0 1 0 0 1 0 1 0 1 1 0 0 1

    P   = .... 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0

    C   = .... 1 0 1 0 0 0 1 0 0 1 0 0 0 0 1
```

Figure 1

E cannot learn $\underline{x}$ without solving the knapsack problem but it seems conceivable that E might be able to deduce some portion of S without ever knowing $\underline{x}$. In the following paragraphs, this is shown to be impossible.

If we assume E attacks this cipher under a known plaintext attack, then E knows S_t for $t \leq$ the present time. This is because

$$S_t = P_t \oplus C_t$$

In spite of this, E will not be able to compute S_t for any t > the present time, even given the full values of the a_i (which extend indefinitely into the future) without also solving for $\underline{x}$, and thus solving the knapsack problem. This can be proven by contradiction.

Assume E can predict S_t for some t > the present time and for any value of the a_i. To determine if x_i is a 0 or a 1, E makes two predictions. E first predicts S_t, and then makes another prediction $\hat{S}_t$ after complementing the single bit, $a_{i,t}$. If $\hat{S}_t \neq S_t$, then x_i must be a 1, otherwise it must be a 0.

This proves that solving this cipher for even one bit of unknown plaintext allows E to recover one of the x_i. Repeating this n times allows recovery of the $\underline{x}$ vector, thus solving the knapsack problem.

Why is this cipher only "essentially" NP-complete? In essence, the question is the distinction between the following two problems:

Find $\underline{x}$, given that we know it exists.

Determine whether or not $\underline{x}$ exists.

The latter is the knapsack problem, and is NP-complete. The former is not quite the knapsack problem, and I have been calling it "essentially" NP-complete.

From a cryptographer's point of view there is not much difference between these two problems. If an algorithm exists that will find _x_ in time T(n), then we can determine whether or not _x_ exists by running the algorithm for time T(n) and noting whether it produces the correct _x_ at the end of that time. If it does produce the correct _x_, then _x_ exists. If it did not find any _x_ in that time, no _x_ exists.

It can be argued that this "proof" is inadequate because we have not taken into account the time required to compute T itself. It is not the purpose of this chapter to provide air-tight theoretical definitions, simply to point out that such theoretical considerations exist, but do not appear to be important for cryptographic applications.

3. Conclusion

A conventional cipher which is essentially NP-complete was given. This cipher, based on the knapsack problem, is the first with this property known to the author.

IX. PROTOCOLS FOR PUBLIC KEY CRYPTOSYSTEMS

1. Introduction

This chapter examines the ways in which public key systems can be used and the special strengths they offer, by giving a series of example protocols. Beyond providing recipes for solving some specific problems, these examples are intended to improve the reader's ability to judge other protocols and, when faced with new problems, to synthesize new protocols.

The reader is assumed to be familiar with the general ideas behind public key cryptosystems, as described in [6], [4].

For many of the following examples, we shall need the services of two communicants, called A and B, and an opponent E. A and B will attempt to send secret messages and sign contracts, while E will attempt to discover the keys, learn the secrets, and forge contracts. Sometimes, A will attempt to evade a contract he signed with B, or B will attempt to forge A's signature to a new contract.

A and B will need to apply one way functions to various arguments of various sizes, so we define the one way function F with the properties that:

1) F can be applied to any argument of any size. F applied to more than one argument is defined to be the same as F applied to the concatenation of the arguments.

2) F will produce an output of fixed size (perhaps 100 bits)

3) Given F and x it is easy to compute F(x).

4) Given F and F(x) it must be impossible to determine x.

5) Given F and x, it must be impossible to determine x' $\neq$ x such that F(x) = F(x').

For a more complete discussion of one way functions, see [7], [38], [19] and chapter II.

2. Centralized Key Distribution

Centralized key distribution using conventional encryption functions was the only reasonable method of handling key distribution in a multi-user environment before the discovery of public key distribution methods. Only conventional encryption functions need be used, which presently offers a performance advantage. (The currently known public key systems are less efficient than conventional cryptographic systems. Whether or not this will continue is not now known. Discovery of new public key systems seems almost inevitable, and discovery of more efficient ones probable.)

In centralized key distribution, A, B, and all other system users somehow deposit a conventional cryptographic key with a central key distribution center. Call X's key k_X, and let C(key,plaintext) be the ciphertext resulting from the conventional encryption function. If A wishes to communicate with B, then A picks a random key k' and computes $y = C(k_A,$<k',"send this key to B">) and sends it to the center along with his name. The center computes $C^{-1}(k_A,y)$ = <k',"send this key to B"> and then computes $z = C(k_B,$<k',"this key is from A">) and sends this to B. B computes $C^{-1}(k_B,z)$ = <k',"this key is from A"> and uses k' in further encrypted communications with A.

This protocol is simple and requires only conventional encryption functions. Needham and Schroeder [25] and Popek and

Kline [27] have defended its use.

The major vulnerability of this protocol is to both centralized loss of security and centralized loss of function. All of the eggs are in a central basket. Theft of the central keys, or bribery of personnel at the central site will compromise all users of the system. Similarly, destruction of the central keys destroys the key distribution mechanism for all users. In addition, even though A and B can communicate with each other, if either of them is unable to communicate with the key distribution center they will not be able to establish a secure key. In contrast, public key distribution will be seen to continue to function when only two users are left, and only the single communication path between them is functional. Public key systems are much more robust.

The security and reliability of centralized key distribution can be increased by using two or more centers, each with its own keys [6]. Destruction or compromise of a single center will not affect the other center. If users always use several keys --one from each center-- both to encrypt and decrypt messages, then compromise of a single key (or a single center) has no effect on user security. Only if all centers are compromised is the users' security compromised. In general, any number of centers can be established; although practical considerations will usually dictate a small number, e.g., two to five.

A system with multiple centers forces each user to establish a key with each center. This increases cost, but also increases security. There are two ways of modeling this increase in security. In the first, we argue that the probability of compromising one center is p, so the probability of compromising k centers is p^k. If p is reasonably small, this model predicts a rapid and dramatic increase in security as the number of centers is increased. In the second model we argue that if the cost of compromising one center is d dollars, then the cost of compromising k centers is only k·d dollars. This model predicts only a small increase in security as new centers are added. The truth probably lies somewhere in between.

The centralized key distribution protocol does not fully solve the key distribution problem. Some sort of key distribution method must be used between each user X and the center to establish each k_X. This problem is nontrivial because no electronic communications can be used for the transmission of k_X, and inexpensive physical methods, e.g., registered mail, offer only moderate security. The use of couriers is reasonably secure, although more expensive. Some implementations of public key distribution protocols do not require a secure channel for transmitting individual keys, rather they only require authentication of one (system) key or the root node in a tree authentication system (see sections 5 and 6).

Centralized key distribution is more vulnerable to both loss of security and loss of function than well designed public key distribution systems. At the present time, it does provide improved performance because conventional encryption functions are more efficient (faster or require less memory) than public key functions. In addition, certified conventional encryption functions are widely available, but this is not true of public key systems. The latter two situations can be expected to change.

3. Simple Public Key Distribution

This is the most basic application of public key systems [6], [18], [20], [21], [31]. Its purpose is to allow A and B to agree on a common key k without any prior secret arrangements, even though E overhears all messages. While public key distribution systems which are not based on public key cryptosystems exist [6], [20], we describe the protocol in terms of a public key cryptosystem. A randomly computes enciphering and deciphering keys E_A and D_A, and sends E_A to B (and inadvertently E). B picks a random key, k, and transmits $E_A(k)$ to A (and E). A computes $D_A(E_A(k)) = k$. A then discards both E_A and D_A, and B discards E_A. The key in future communications is k. It is used to encrypt all further messages using a conventional encryption function. Once A and B have finished talking, they both discard k. If they later resume the conversation the process is repeated to agree on a new key k'.

This protocol is very simple, and has a great deal to recommend it. First, no keys and no secret materials exist before A and B start communicating, and nothing is retained after they have finished. It is impossible for E to compromise any keys either before the conversation takes place, or after it is over, for the keys exist only during the conversation. Furthermore, if E is passive and does not actively interfere with the messages being sent, then E will understand nothing and the conversation will be secure.

The disadvantage of this protocol is that E might actively interfere with the exchange of keys. Worse yet, if E has control of the channel, he can force a known k on both A and B. All further messages encrypted with k can then be read by E. All E need do is pretend to B that he is A, and pretend to A that he is B. To do this, E blocks transmission of E_A to B, and substitutes E_E. B will compute $E_E(k)$ and transmit it to A. E will block this transmission, learn k by computing $D_E(E_E(k))$ = k and then send $F_A(k)$ to A. A will compute $D_A(E_A(k))$ = k as usual. E knows k, and both A and B are none the wiser.

In spite of this disadvantage, the protocol is very useful for two reasons. Passive eavesdropping, by itself, is a major problem. In "The Codebreakers," the authoritative 1164 page history of cryptography by David Kahn [12], the threat was from passive eavesdropping in the vast majority of cases. Use of a simple public key distribution protocol provides protection from this attack, and also provides a positive guarantee against lost or stolen codebooks, bribery or blackmail of code-clerks, and "practical cryptanalysis" by theft of keys. For example, the major vulnerability of the U.S. telephone network today is from technically sophisticated passive eavesdropping. The Russians use their embassies and consulates in the U.S. to house microwave receivers which listen to conversations carried between telephone company microwave towers [36], [3]. They are not jamming or altering phone calls; just listening.

Secondly, if the reader has a preference for any other key distribution protocol which does not provide these blanket guarantees against lost or stolen keys, then it is simple to combine the readers preferred key distribution protocol with the simple public key distribution protocol to obtain a hybrid which offers the strengths of both. The problem of carelessly lost keys, poor key security, theft of keys, and bribery of clerks or janitors who have access to the key are not minor, as history shows [12]. A blanket guarantee against all passive attacks is extremely comforting.

When guarantees of authenticity are also required, the simple public key distribution protocol can be used together with other methods because of the remarkably strong guarantees it provides against the passive eavesdropper. Even though a "better" method is being used to provide authenticity, its security might have been compromised by theft of keys, in which case it is impossible to guarantee authenticity, but the simple key distribution protocol at least still guarantees secrecy.

4. Authenticated Public Key Distribution

There is a now classic protocol [6] which provides secure and authenticated communications between A and B: A and B generate F_A and E_B and make them public, while keeping D_A and D_B secret. The public enciphering keys of all users are entered in a public file, allowing easy and authenticated access to E_X for any user, X. E_X can be authenticated upon entry in the file by X making a personal appearance.

If A and B wish to agree on a common key k, then

1) A looks up E_B in the public file.

2) A generates k_1 randomly and transmits $E_B(k_1)$ to B.

3) B looks up E_A in the public file.

4) B generates k_2 randomly and transmits $E_A(k_2)$ to A.

5) A computes $k = \langle k_1,k_2 \rangle$, where $k_2 = D_A(E_A(k_2))$.

6) B computes $k = \langle k_1,k_2 \rangle$, where $k_1 = D_B(E_B(k_1))$.

At the end of this protocol, A and B have agreed on a common key, k, which is both secret and authenticated. A is assured he is talking to B, for only B can decipher $E_B(k_1)$, while B is assured he is talking to A because only A can decipher $E_A(k_2)$.

This protocol suffers from two weaknesses. First, entries in the public file might be altered. E might create a false entry in A's public file which read:

$$B\ldots\ldots\ldots\ldots E_E$$

This false entry would let E pretend to A that he was B, to the disadvantage of both A and B.

False entries in the public file can be dealt with both by good physical security, or by using new protocols (see sections 5 and 6) for authenticating the entries in the public file.

Second, secret deciphering keys can be lost. If E should learn D_B, then E could masquerade as B to A without altering the public file. Unless additional precautions are taken, A and B might never find out about the loss. Note that if D_B is compromised but D_A is still secure, then A can no longer be sure he is talking to B, but he can be sure he is talking secretly to some (unauthenticated) person claiming to be B.

5. Public Key Distribution with Certificates

Kohnfelder [14] suggested that entries in the public file can be authenticated by having a Central Authority (CA) sign them with D_{CA}. He called such signed entries certificates. The certificate for A, called C_A, is computed by the central authority as:

$$C_A = D_{CA}(<\text{"user A"},E_A>)$$

while similarly C_B is computed as:

$$C_B = D_{CA}(<\text{"user B"},E_B>)$$

The protocol with certificates is the same as the authenticated protocol, except steps 1 and 3, which involve looking up E_A and E_B, are replaced by the steps of obtaining and checking the certificates for A and B. The modified protocol is:

1) A obtains B's certificate (either from a public file, or by requesting it from B) and confirms it by computing

$$E_{CA}(C_B) = \text{"user B"},E_B$$

2) A generates k_1 randomly and transmits $E_B(k_1)$ to B.

3) B obtains A's certificate and confirms it by computing

$$E_{CA}(C_A) = \text{"user A"},E_A$$

4) B generates k_2 randomly and transmits $E_A(k_2)$ to A.

5) A computes $k = \langle k_1,k_2 \rangle$, where $k_2 = D_A(E_A(k_2))$.

6) B computes $k = \langle k_1,k_2 \rangle$, where $k_1 = D_B(E_B(k_1))$.

This protocol assures A and B that each has the other's public enciphering key, and not the public enciphering key of some imposter.

The security of this protocol rests on the assumptions that D_A, D_B, and D_{CA} have not been compromised, that A and B have correct copies of E_{CA}, and that the central authority has not issued a bad certificate, either deliberately because it was untrustworthy, or accidentally because it was tricked.

E_{CA} can be published in newspapers and magazines, and sent over all available communication channels. Blocking its correct reception would be very difficult.

Security can be improved by having several "Central Authorities," each with its own secret deciphering key. Each user would be given a certificate from each authority, all authenticating the same public enciphering key. Compromise of a single authority will no longer result in compromise of the system.

If only a single "Central Authority" exists, and D_{CA} is compromised, then it is no longer possible to authenticate the users of the system and their public enciphering keys. The certificates are now worthless because the (unauthorized) person who has learned D_{CA} can produce false certificates at will.

This problem can be greatly reduced by destroying D_{CA} after certificates for all users have been created. If D_{CA} no longer exists, it cannot be compromised. The central authority would create E_{CA} and D_{CA}, sign all the certificates, then immediately destroy D_{CA}. D_{CA} would be vulnerable only during the short time that it was being used to sign certificates.

While it is now impossible for anyone to falsely add new users to the system by creating false certificates, it is also impossible to add legitimate users to the system as well. This is unacceptable. The simplest way of dealing with this problem is for the central authority to issue new certificates with a new (different) secret deciphering key. For example, each month the central authority could create new certificates for that month's new users using a newly created D_{CA}. The new E_{CA} would be published, and the new users would be accepted. The new D_{CA} would be destroyed after use.

Although this method sharply reduces the risk that D_{CA} might be compromised, it still leaves open the possibility that the central authority might issue bad certificates either by intent, or because of some trickery during the short period when new certificates are actually being signed. These possibilities can be effectively eliminated by the next protocol.

6. Public Key Distribution with Tree Authentication

Key distribution with certificates was vulnerable to the criticism that D_{CA} can be compromised, resulting in system wide loss of authentication (although not necessarily loss of secrecy). This problem can be solved by using tree authentication, as described in Chapter V.

Again, this protocol attempts to authenticate entries in the public file. However, instead of signing each entry in the public file, this protocol applies a one way hash function, H, to the entire public file. Even though H is applied to the entire public file, the output of H is only 100 or 200 bits long. The (small) output of H will be called the root, R, of the public file. If all users of the system know R, then all users can authenticate the correctness of the (whole) public file by computing R = H(public file). Any attempt to introduce changes into the public file will imply with probability near one that R ≠ H(altered public file), an easily detected fact.

This method effectively eliminates the possibility of compromising D_{CA} because no secret deciphering key exists. Anyone can compute R = H(public file), and so confirm that the copy of the public file that they have is correct. R, (like E_{CA} in the protocol of section 5) can be widely distributed.

Because correct copies of the public file are widely distributed, it is very easy for a user of the system to discover that someone else is attempting to masquerade as him. If E has put the false entry

$$A.............E_E$$

into the public file, then A will discover this fact when he looks at his own entry. A cannot be given a specially "printed" public file with his entry correct because then H(public file) would not equal R. If new public files are issued before they go into use, then all users of the system will have time to assure that they have been correctly entered into the public file. Because the public file will be subjected to the harsh glare of public scrutiny, and because making alterations in the public file is effectively impossible, a high degree of assurance that the public file is correct can be attained.

While this concept is very comforting, forcing each user to keep a complete copy of the public file might not be practical. Fortunately, it is possible to selectively authenticate individual entries in the public file, without having to know the whole public file. This is done by using "tree authentication," described in chapter V.

The essence of tree authentication is to authenticate the entire public file by "divide and conquer." If we define $\underline{Y}$ = public file = Y_1, Y_2, ... Y_n, (so the $i\underline{th}$ entry in the public file is denoted Y_i, and B's entry is Y_B); we can define H(public file) = $H(\underline{Y})$ as:

$$H(\underline{Y}) = F(\ H(\text{first half of } \underline{Y}),\ H(\text{second half of } \underline{Y})\ )$$

Where F is a one way function defined in section 1.

If A wishes to confirm B's public enciphering key, then A need only know the first half of the public file, (which is where Y_B appears) and H(second half of public file) which is only 100 bits long. A can compute H(public file) knowing only this information, and yet A only knows half the entries in the public file.

In a similar fashion, A does not really need to know all of the first half of the public file, for

H(first half of public file) =
  F(H(first quarter of public file),
        H(second quarter of public file)   )

All A needs to know is the first quarter of the public file (which has Y_B), and H(second quarter of public file).

By applying this concept recursively, A can confirm Y_B in the public file knowing only R, $\log_2 n$ intermediate H values, and Y_B itself. The information needed to authenticate Y_B, given that R has already been authenticated, lies along the path from R to Y_B. This information will be called the authentication path.

These definitions are illustrated in figure 1, which shows the authentication path for Y_5.

This brief sketch of tree authentication should serve to convey the idea. For a more detailed discussion the reader is

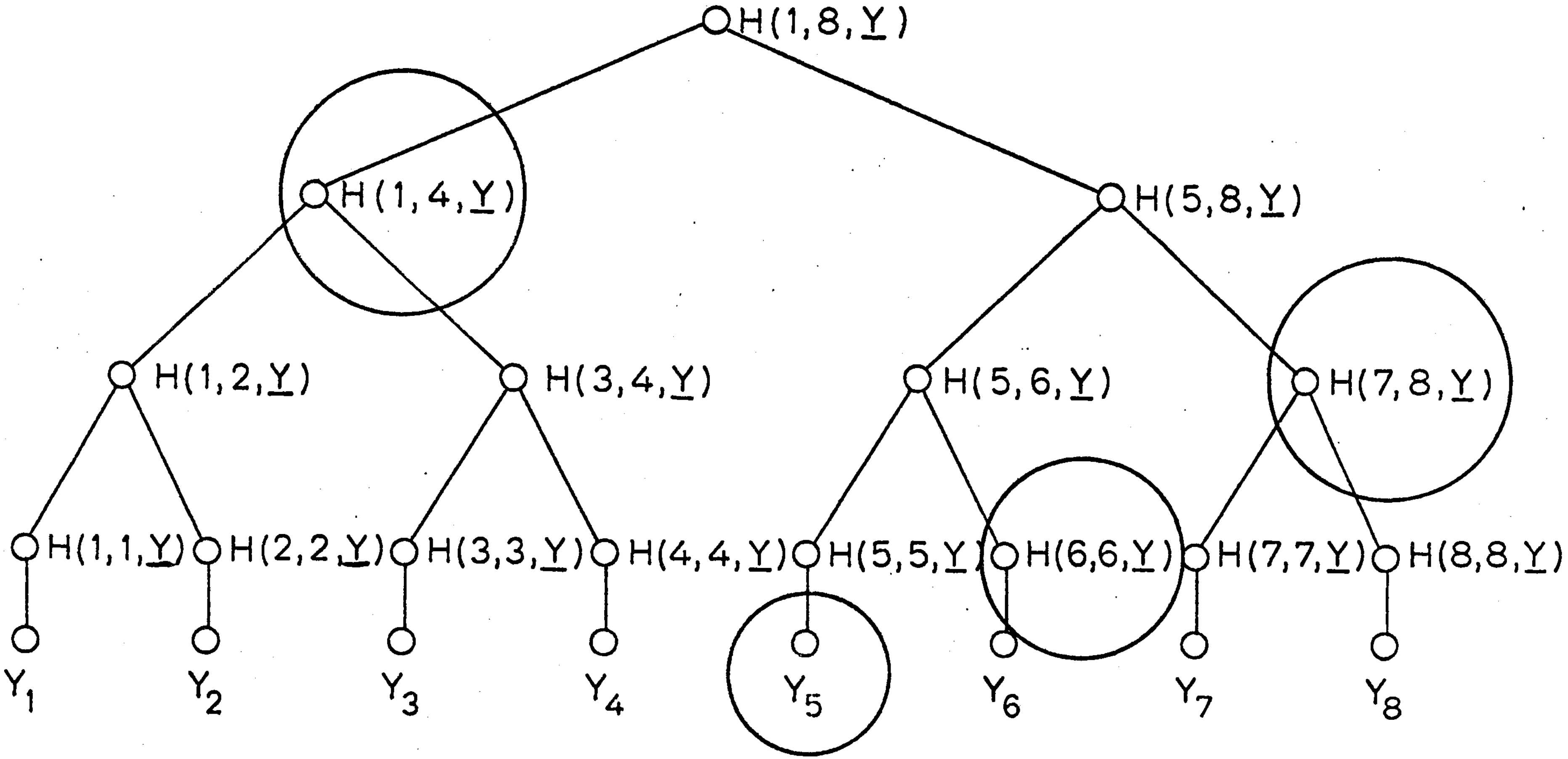

FIG 1

CIRCLED ENTRIES SHOW THE AUTHENTICATION PATH FOR Y_5.

referred to chapter V.

Using tree authentication, user A has an authentication path which can be used to authenticate user A's public enciphering key, provided that R has already been authenticated. An "authentication path" is a new form of certificate, with E_{CA} replaced by R.

The advantage of tree authentication over certificates is that no secret deciphering key D_{CA} exists, so D_{CA} cannot be compromised. It is impossible to create false certificates after R is computed.

With tree authentication, it is impossible to have a centralized loss of authentication, but it is also impossible to add new users without issuing a new tree. The tree, once computed, is fixed and unchanging. Therefore, the public file (which is just the leaves of the tree) is also fixed and unchanging. For this reason, it can be carefully and publicly checked for errors. For the same reason, it is impossible to update. A new tree must be issued periodically.

In summary: If tree authentication is used to authenticate each entry in the public file, the protocol for public key distribution proceeds as follows:

1) A obtains B's entry in the public file and B's authentication path (either from B or from some convenient storage device) and confirms their correctness.
2) A generates k_1 randomly and transmits $E_B(k_1)$ to B.

3) B obtains A's entry in the public file and A's authentication path and confirms their correctness.

4) B generates k_2 randomly and transmits $E_A(k_2)$ to A.

5) A computes $k = \langle k_1,k_2 \rangle$, where $k_2 = D_A(E_A(k_2))$.

6) B computes $k = \langle k_1,k_2 \rangle$, where $k_1 = D_B(E_B(k_1))$.

This protocol can only be compromised if: D_A or D_B is compromised, or if R is not correctly known by A or B, or if there is a false and misleading entry in the public file. The latter two are easily detectable. If either A or B has the wrong R, they will be unable to complete the protocol with any other legitimate user who has the correct R. Complete failure of the protocol is easily detected, and will lead to some sort of corrective action. Implicitly, the correct value of R is agreed on by A and B every time each confirms the correctness of the other's authentication path. The correct value of R is therefore being constantly transmitted between pairs of users as they establish keys. This is in addition to other means of confirming R, such as publication.

Because the public file is both open to public scrutiny and unalterable, false or misleading entries can be rapidly detected. In practice, a few users concerned with correctness can verify that the public file satisfies some simple global properties, i.e., each user name appears once and only once in the entire public file; individual users can then verify that their own entry is correct, and need not bother examining the rest of the public file.

The only practical method of compromising either A's or B's security is to compromise D_A or D_B. A user's security is thus dependent on himself and no one else.

It is still possible for A to claim to be the non-existent C. Because C does not exist, he will never object that A is masquerading as him. A can effectively establish pseudonyms. If it is essential to establish a one to one correspondence between named users of the system and real people, some form of physical authentication is necessary. In many applications there is no need to know that user C is really a pseudonym for user A. As long as C pays his bills, his real identity is irrelevant. The identifier "C" is relative, not absolute, and serves simply to tie together a sequence of transactions.

7. Digital Signatures

Diffie and Hellman [6] suggested the use of public key cryptosystems to provide digital signatures, and Rivest, Shamir and Adleman [31] have suggested an attractive implementation. Signature techniques based on methods other than public key cryptosystems have been suggested by Lamport and Diffie [6], Rabin [29], and Merkle [19].

Digital signatures, whether based on conventional encryption functions, on public key cryptosystems, on probabilistic computations, or on other techniques, share several important properties in common. These common properties are best illustrated by explaining the general concept of a digital signature.

The now classic example of a digital signature is that of a person A who wishes to place a purchase order with his stock broker P. A has just received word that the stock will go up in value, and wishes to purchase it within a few hours. A, on the Riviera, cannot send a written order to B in New York in time. All that A can quickly send to P is information, i.e., a sequence of bits, but B is concerned that A may later disclaim the order. A must somehow generate a sequence of bits (a digital signature) which will convince P (and if need be, a judge) that A authorized the order. It must be easy for P to validate the digital signature, but impossible for him (or anyone other than A) to generate it (to prevent charges that B was dabbling in the market illegally with A's money).

The signature must be a function of both the message and the signer, for it must convince B (and a judge) that the particular person, A, has signed the particular message, m. There is basically one situation which the digital signature must resolve: B claims that A signed a message, and A claims he did not. If in fact A signed the message, then he is guilty of disavowal; but if he did not, B is guilty of forgery. To summarize: a digital signature should be message dependent, signer dependent, easy for the sender to generate, easy for the user to validate, but impossible to forge or disavow.

There are digital signature schemes which do not involve public key cryptosystems, and some which involve elaborate interactions between A and B, as well as the clever use of random information [29], but it will be convenient notationally to let A sign message m by computing the signature, $D_A(m)$. Checking a signature will then be done by checking that $m = E_A(D_A(m))$. If $E_A(D_A(m))$ produces an illegible message (random bits) then the signature is rejected as invalid. This notation is somewhat misleading because the actual method of generating and validating signatures can be very different from this model. This notation is retained because it is widely known and because we will not discuss the differences among different digital signature methods, only their common properties. The conclusions reached in the following paragraphs apply to both public key and non-public key based signature systems.

8. A Simple Digital Signature Protocol

The first digital signature protocol, proposed in [6], proceeded as follows:

A and B agree on message m that A is to sign. A computes $D_A(m)$ (where D_A is known only to A) and transmits it to B. B looks up E_A in the public file. B can now check $D_A(m)$ by computing $E_A(D_A(m))$ and confirming that it equals m. B retains $D_A(m)$ as proof that A signed message m.

If A later denies having signed message m, B can give $D_A(m)$ to a judge, who can easily compute $E_A(D_A(m)) = m$, proving that A signed the message.

This protocol has been criticized [32], [27] on two grounds.

First, the public file might have been tampered with. When B looks up E_A in the public file, E might have altered the public file so that E_E appears next to A's name. B will then "check" a signature with the wrong public enciphering key, making B's "check" useless. Methods of authenticating the public file, discussed previously under key distribution protocols, minimize this problem.

A second criticism, raised by Saltzer [32] and by Popek and Kline [27], is that A can disavow the signed message, explaining "E has stolen D_A, and has posted it in a public place. Clearly, anyone could have computed $D_A(m)$, so it proves nothing." In fact, E did not steal anything; A posted D_A in a public place himself. A can now disavow any message he ever signed.

Although this simple protocol is flawed, it should be compared with current practice. At the moment, it is possible to order goods and services simply by giving a valid credit card number and nothing else.

Further, A cannot disavow a signed message without generating suspicion. Repeated disavowals would be especially questioned. In an actual system, the incidence of disavowal will be low, which implies that careful investigation of those cases that do occur is possible.

A simple solution to the disavowal problem is to adopt very good physical security for D_A, and then refuse to accept A's claim that D_A was compromised. Several factors combine to allow extraordinarily good physical security for D_A. First, destruction of D_A is merely inconvenient. A can always generate a new D'_A and E'_A. If theft of D_A is imminent, A can destroy D_A. Contracts signed with D_A are still valid and E_A still exists to authenticate them, even though D_A has been destroyed.

Second, only a single copy of D_A need exist. Because destruction of D_A is only inconvenient, backup copies of D_A need not be kept. D_A could be kept in a small strong box or on a single chip of silicon in a "signet ring" worn by A. Attempts to open the ring would cause destruction of D_A.

To summarize, the simple signature protocol is a great improvement over the current situation (no signature protocol), but it suffers from three problems: the public file that B checks must be accurate; A might disavow the resulting signature, explaining (falsely) that D_A was stolen; and D_A might actually be stolen by E who can then impersonate A.

Authenticating the entries in the public file was considered under public key distribution protocols.

Physical security of D_A is A's responsibility.

Disavowal is considered further in section 9.

9. Dealing with Disavowal

If it is necessary to assume that D_A can be compromised, then several protocols which reduce or mitigate the problem of disavowal suggest themselves.

One solution is to have a witness testify to the time of the order only. Essentially, this reduces the role of the witness to that of a reliable time stamp. As mentioned before, if a message was signed prior to the time the key was compromised, then it must be valid. The witness signs a statement of the format "The time is now 12:04:23 on the 17th of March, 1979, and I have been presented with the following bit pattern xxxxxxxxxxxxxxxxxxxxx." If the witness' signature is still valid, and the witness signed the statement prior to the time D_A was compromised, then the witnessed signature must also be valid. The witness need not physically authenticate A's agreement; the witness does not care where the bit pattern comes from nor what it means.

If A claims he lost D_A yesterday, then a message signed three months ago and "time stamped" by a witness two months ago is still valid. Only A's recently signed messages are open to question.

A can still disavow a message signed at 2:00 explaining that his signature was compromised at 1:00 but that he didn't notice this fact until 3:00. If A's argument that D_A was compromised and he failed to notice this for two hours is accepted, then he can disavow a signed message. Since the "benefit" of most disavowals (e.g. stock orders) is not realized until a significant time after the message is sent, time stamping is more valuable than might first appear.

If there is a witness who is trusted not to disavow his signature, why not rely on him entirely, and eliminate A's digital signature completely? The witness' testimony that A agreed to the contract would BE the "signature." If the witness is fully trusted, he need not even use digital signatures. He could simply remember the contracts. In the event of dispute, the witness would simply look up the appropriate contract, and all parties to the dispute would abide by that version. (Popek and Kline [27] advocate the use of such methods.)

The primary disadvantage to this alternative is that A loses control over his signing ability. The witness can now forge A's "signature" on a contract, either because the witness is malicious, or because the witness made a mistake. If the witness' word is accepted as binding, A would have no recourse. Even though A swore that he had not seen the contract, had not agreed to the contract, and would never have agreed to the contract, if the "signature" provided by the witness is to be useful, A's pleas must be ignored. In contrast, if the witness only countersigns A's digital signature, then A is guaranteed that forgery is impossible so long as D_A is secure.

In addition, if A is in San Francisco, B is in New York, and the witness is in Philadelphia, then some form of secure communications between the various sites is required. This adds additional points of vulnerability to the system.

Finally, if the witness is responsible for many contracts worth many millions of dollars, it is an attractive target for system penetrators and vandals. By contrast, digital signatures are distributed; there is no central site whose destruction or compromise would invalidate all signatures for all users.

Disavowal is an inherent property of any signature technique, including written signatures, stamps, seals, etc. For any signature system, the signer can try to disavow his signature by creating a fanciful but not impossible scenario which would have allowed someone else to have forged the signature. The essential question is the plausibility of these scenarios. As their plausibility is reduced, the risk of disavowal is also reduced. To be practical, a signature system must reduce the risk of disavowal to a level which is tolerable for the particular application. Complete elimination of all risk does not appear to be attainable in practice.

10. Conclusions

The primary purpose of this chapter has been to increase the readers insight into the strengths and weaknesses not only of the particular protocols described, but also of cryptographic protocols in general. Certainly, these are not the only cryptographic protocols possible. However, these protocols are valuable tools to the system designer: they illustrate what can be achieved and provide feasible solutions to some problems of recurring interest.

X. ON THE SECURITY OF MULTIPLE ENCRYPTION

Diffie and Hellman [5] have argued that the 56-bit key used in the Federal Data Encryption Standard (DES) [24] is too small and that current technology allows an exhaustive search of the 2^{56} keys. Although there is controversy surrounding this issue [9,15,34,23,40,2], there is almost universal agreement [37,5] that multiple encryption using independent keys can increase the strength of DES. But, as noted in [5], the increase in security can be far less than might first appear. This chapter shows that a recently proposed scheme [37] for multiple encryption suffers from such a weakness.

The simplest approach to increasing the key size is to encrypt twice, with two independent keys K1 and K2. Letting P be a 64 bit plaintext, C a 64 bit ciphertext, and K a 56 bit key, the basic DES encryption operation can be represented as

$$C = S_K(P) \qquad (10.1)$$

and simple double encryption is obtained as

$$C = S_{K2}[S_{K1}(P)] \qquad (10.2)$$

While exhaustive search over all 2^{112} keys ((K1,K2) pairs) requires 2^{112} operations and is clearly infeasible, this cipher can be broken under a known plaintext attack (where corresponding plaintext and ciphertext are both known) with 2^{56} operations [5], and 2^{56} words of memory. The complexity, measured as time plus memory is therefore no greater than is needed to cryptanalyze a single 56 bit key exhaustively. (Though the cost is somewhat higher since memory is "more expensive" than time.) If P and C represent a known plaintext-ciphertext pair, then the algorithm for accomplishing this [5] encrypts P under all 2^{56} possible values of K1, decrypts C under all 2^{56} values of K2 and looks for a match. For obvious reasons this is called a "meet in the middle" attack and is given in detail by the following algorithm (Where n is the number of keys in the key space. For DES, n = 2^{56}):

```
1.) For i = 1 to n Do
        a.) Table[i] = <$S_i(P)$,i,"encrypt">
        b.) Table[n+i] = <$S_i^{-1}(C)$,i,"decrypt">
2.) Sort the table on the first field.
3.) Search the table for adjacent entries of the form
                <value,$\hat{K1}$,"encrypt">
                <value,$\hat{K2}$,"decrypt">
    and test to see if $\hat{K1}$ and $\hat{K2}$ are the correct keys by
    encrypting further plaintext-ciphertext pairs. Based
    on unicity distance arguments [30,35], there will be
    about $2^{48}$ "false alarms" for $\hat{K1}$ and $\hat{K2}$ if a single
    plaintext-ciphertext pair is used. Testing these
    takes less than $2^{56}$ operations and therefore contri-
    butes an unimportant overhead to the computation.
```

While the algorithm given runs in time n log n, it could be rewritten using hash tables to run in essentially linear time. In any event, the present analysis will neglect logarithmic factors.

The use of double encryption provides some increase in security because the algorithm for cryptanalysis requires 2^{56} words of memory, as well as 2^{56} operations. The cost of a machine to perform 2^{56} operations in approximately a day has been estimated by Diffie and Hellman [5] to be about 20 million dollars. The cost of 2^{56} 64-bit words of memory on 6250 cpi reels of magnetic tape, assuming 2400 foot reels that cost $15 dollars each, is about 60 billion dollars.

While the cost of implementing this search is high enough to discourage its use today, the danger of cheaper technology or shortcuts [9] in the future prompted Diffie and Hellman to suggest triple encryption with three independent keys K1, K2 and K3. A generalized meet in the middle attack would then require 2^{112} operations and be well beyond the foreseeable technology for at least 50 years, and possibly forever.

At the 1978 National Computer Conference, Tuchman [37] proposed a triple encryption method which uses only two keys K1 and K2. The plaintext is encrypted with K1, decrypted with K2, then again encrypted with K1 so that:

$$C = S_{K1}\{S_{K2}^{-1}[S_{K1}(P)]\} \qquad (10.3)$$

This method seems to avoid the "meet in the middle" attack outlined above and is upwardly compatible with a single encryption by setting K1 = K2 to produce:

$$C = S_{K1}\{S_{K1}^{-1}[S_{K1}(P)]\} = S_{K1}(P) \qquad (10.4)$$

This allows users of the new (two key) system to decrypt data encrypted by users of the old (single key) system.

Although the encryption technique (10.3) provides more security than simple double encryption as in (10.2), it is shown below that the new method can be cryptanalyzed using a chosen plaintext attack [6] in about 2^{56} operations. We therefore recommend that if triple encryption is used, there be three independent keys. If compatibility with single encryption is desired, the operation can be taken to be:

$$C = S_{K1}\{S_{K2}^{-1}[S_{K3}(P)]\} \qquad (10.5)$$

Then when K1 = K2 = K3 = K, C = $S_K(P)$. Users could also be compatible with Tuchman's suggested two key method by taking K1 = K3.

Although chosen plaintext attacks can sometimes be mounted on real systems, the following should be viewed as a "certificational attack" which is only indicative of a weakness. History, littered with the broken remains of "unbreakable" ciphers, teaches extreme caution in certifying a new one [12], so that today even an indication of weakness is regarded as dangerous. In many cases, ciphers which have yielded to chosen plaintext attacks have later proven vulnerable to known plaintext or ciphertext only attacks as well.

We define some useful notation before describing the method of cryptanalysis:

$$C = \mathrm{Enc}(P) = S_{K1}\{S^{-1}_{K2}[S_{K1}(P)]\} \qquad (10.6)$$

$$M1 = S_{K1}(P) \qquad (10.7)$$

$$M2 = S^{-1}_{K2}(M1) \qquad (10.8)$$

$$= S^{-1}_{K2}(S_{K1}(P)) \qquad (10.9)$$

$$= S^{-1}_{K1}(C) \qquad (10.10)$$

M1 and M2 are intermediate values in the computation of C from P.

We can motivate the method of cryptanalysis with the following observations:

If we knew K1 and a (P,C) pair, then it would be possible to compute the intermediate values M1 and M2 from (10.7) and (10.10). This would let us mount a known plaintext attack on K2 using (10.8). There are 2^{56} values of K1, so if we could quickly determine the right K2 once we found the right K1, then cryptanalysis would only take 2^{56} operations. Determining K2 using a known plaintext attack requires 2^{56} operations, which is too long.

The trick is to somehow change the known plaintext attack on K2 to a chosen plaintext attack (that is, M1 is chosen in (10.8); for example, M1 = $\underline{0}$), so we can quickly solve for K2 with a table lookup. This increases the memory needed to 2^{56} words, the same as is needed by the meet in the middle attack for simple double encryption.

For this attack to work, we must find the value of P such that $M1 = S_{K1}(P) = \underline{0}$. If we knew the right K1, then we could easily compute $P = S_{K1}^{-1}(\underline{0})$ from(10.7), and cryptanalyze the system in one step, because we could then request Enc(P) = C, (by the chosen plaintext assumption); compute $S_{K1}^{-1}(C) = M2$; and compute K2 in one step from M2 using the precomputed table.

Since we do not know K1, we repeat this process for each of its 2^{56} possible values and test any resulting (K1,K2) pairs to see which one is correct. Again using unicity distance arguments, we expect 2^{48} false alarms, which is small compared with 2^{56}.

Because $P = S_{K1}^{-1}(\underline{0})$ from (10.7) and $M2 = S_{K1}^{-1}(\underline{0})$ from (10.8) the algorithm can proceed as follows:

```
1.)  For i = 1 to n Do
     a.) $\hat{M}2 = S_i^{-1}(\underline{0})$
     b.) Table[i] = <$\hat{M}2$,i,"2">
     c.) $\hat{M}2 = S_i^{-1}(Enc(S_i^{-1}(\underline{0})))$
     d.) Table[n+i] = <$\hat{M}2$ ,i,"1">
2.)  Sort the table on the first field.
3.)  Search the table for adjacent entries of the form
                    <value,$\hat{K}2$,"2">
                    <value,$\hat{K}1$,"1">
```

and test to see if $\hat{K}1$ and $\hat{K}2$ are the correct keys by checking further plaintext-ciphertext pairs.

Step 3 is guaranteed to find the correct (K1,K2) pair.

Conclusion

Two methods of multiple encryption have been shown to be less secure than they first appeared. The weakness in both cases came from an ability to separate the key into two halves which did not interact. We conclude that all bits of the key should come into play repeatedly in a complex fashion as they do in the 56-bit DES, and that multiple encryption with any cryptographic system is liable to be much less secure than a system designed originally for the longer key.

XI. CONCLUSION

The use of cryptography is growing because of the increasing demand for privacy as reflected in new legislation, the need to protect vulnerable electronic funds transfer systems, the increasing quantity and value of information sent over vulnerable communication channels, the dropping price of eavesdropping and analyzing information, and the dropping price of protecting information via encryption.

Although it is difficult to predict the curves and swerves of a new and rapidly developing technology, it appears that many of the techniques described in this thesis will be used in telecommunication systems that span the globe to protect the privacy and integrity of communications of all kinds.

XII. BIBLIOGRAPHY

1. A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Reading, Ma.: Addison-Wesley, 1974.

2. D.K. Branstad, J. Gait, and S. Katzke, Report of the workshop on cryptography in support of computer security, National Bureau of Standards Rep. NBSIR 77-1291, 21-22 Sept. 1976.

3. R. Davis, Remedies sought to defeat Soviet eavesdropping on microwave links, Microwave Syst., vol. 8, no. 6, pp. 17-20, June 1978.

4. Diffie, W., and Hellman, M.E., Privacy and authentication: an introduction to cryptography, Proceedings of the IEEE Vol. 67, No. 3, Mar. 1979 pp. 397-427.

5. Diffie, W. and Hellman, M. Exhaustive cryptanalysis of the NBS data encryption standard, Computer June 1977, pp. 74-84.

6. Diffie, W., and Hellman, M. New directions in cryptography. IEEE Trans. on Inform. IT-22, 6(Nov. 1976), 644-654.

7. Evans, A., Kantrowitz, W., and Weiss, E. A user authentication system not requiring secrecy in the computer. Comm. ACM

17, 8(Aug. 1974), 437-442.

8. Feistel, H. Cryptography and computer privacy. Sci. Am. 228,5 (May 1973), 15-23.

9. Hellman, M. Merkle, R. Schroeppel, R. Washington, L., Diffie, W., Pohlig, S., Schweitzer, P. Results of an initial attempt to cryptanalyze the NBS data encryption standard, Information Systems Laboratory SEL 76-042, Sept. 9, 1976.

10. E. Horowitz and S. Sahni, Computing partitions with applications to the knapsack problem, JACM, Vol 21, No. 2, April 1974, pp. 277-292.

11. O.H. Ibarra and C. E. Kim, Fast approximation algorithms for the knapsack and sum of subset problems, JACM Vol. 22, No. 4, October 1975, pp. 463-468.

12. Kahn, D. *The Codebreakers*. Macmillan, New York, 1976.

13. Karp, R. M. Reducibility among combinatorial problems, in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York (1972), pp. 85-104.

14. Kohnfelder, L.M. Using certificates for key distribution in a public-key cryptosystem. Private communication.

15. Kolata, G.B. Computer encryption and the National Security Agency. Science, Vol. 197, July 29, 1977.

16. E. L. Lawler, Fast approximation algorithms for knapsack problems, Electronics Research Laboratory, College of Eng. U.C. Berkeley Memorandum No. UCB/ERL M77/45, 21 June 1977.

17. Lipton, S.M., and Matyas, S.M. Making the digital signature legal--and safeguarded. Data Communications (Feb. 1978), 41-52.

18. McEliece, R.J. A public-key cryptosystem based on algebraic coding theory. DSN Progress Report, JPL, (Jan. and Feb. 1978), 42-44.

19. Merkle, R. A certified digital signature, submitted to CACM.

20. Merkle, R. Secure communications over insecure channels. Comm. ACM 21, 4(Apr. 1978), 294-299.

21. Merkle, R., and Hellman, M. Hiding information and signatures in trapdoor knapsacks. IEEE Trans. on Inform. IT-24, 5(Sept. 1978), 525-530.

22. R. Merkle, Secure communications over insecure channels,

CACM Vol. 4, No. 21, April 1978 pp 294-299.

23. Morris, R., Sloane, N.J.A., and Wyner, A.D. Assessment of the National Bureau of Standards proposed federal data encryption standard. Cryptologia, vol. 1, pp. 281-291, July 1977.

24. National Bureau of Standards, Federal Information Processing Standards Publication No. 46.

25. R.M. Needham and M.D. Schroeder, Using encryption for authentication in large networks of computers. CACM 21,12 Dec. 1978 pp. 993-999.

26. S.C. Pohlig and M.E. Hellman, An improved algorithm for computing logarithms over GF(P) and its cryptographic significance, IEEE Trans. on Info Theory, vol. IT-24, pp. 106-111, Jan. 1978.

27. G.J. Popek and C.S. Kline, Encryption protocols, public key algorithms, and digital signatures in computer networks; in _Foundations of Secure Computation_, pp. 133-153.

28. George B. Purdy, A high security log-in procedure, Comm. of the ACM, Vol. 17 No. 8, pp. 442-445, Aug. 1974.

29. Rabin, M.O., Digitalized signatures, in Foundations of Secure Computation, ed. Demillo, R.A., et. al. pp. 155-166.

30. Hellman, M.E. An extension of the Shannon theory approach to cryptography, IEEE Trans. Inform. Theory Vol. IT-23, May 1977 pp. 289-294.

31. Rivest, R.L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. Comm. ACM 21, 2(Feb. 1978), 120-126.

32. J. Saltzer, On Digital Signatures, private communication.

33. R. Schroeppel, unpublished work.

34. Senate Select Committee on Intelligence, Involvement of the NSA in the development of the data encryption standard. News release, April 12, 1978.

35. Shannon, C.E. Communication theory of secrecy systems. Bell Sys. Tech. Jour. 28 (1949) 654-715.

36. J. Squires, Russ monitor of U.S. phones, Chicago Tribune pp. 123, June 25, 1975.

37. W.L. Tuchman, Talk presented at the Nat. Computer Conf., Anaheim, Ca. June 1978.

38. Wilkes, M.V., Time-Sharing Computer Systems, Elsevier, New York, 1972.

39. Wyner, A. D. The wire tap channel. Bell Sys. Tech. Jour. 54,8 (Oct. 1975), 1355-1387.

40. E.K. Yasaki, Encryption algorithm: key size is the thing, Datamation, Vol. 22, No. 3, pp. 164,166, Mar. 1976.

41. Feller, W. An Introduction to Probability Theory and its Applications, Vol. I, third edition, New York, Wiley 1968, p. 33.

42. Shamir, A. On the cryptocomplexity of knapsack systems, Symposium on the Theory of Complexity, Atlanta, Georgia, Apr. 1979.

43. Shamir, A. A fast signature scheme, MIT Laboratory for Computer Science Report TM-107 July 1978.

XIII. ROUTINES TO GENERATE TRAPDOOR KNAPSACKS

```
/*
     General Comments:

     The following routines have been written in BC, a
simplified version of C that does indefinite precision
integer arithmetic.  BC runs on a PDP-11 under the Unix
operating system.
     BC has a few peculiar conventions as follows:

     %      The infix mod operator.  w%m computes w modulo m.
     auto   A local variable declaration.
     ++     The unary increment-by-one operator.  As a
            postfix operator, it first returns the value
            of a variable, then increments it.  As a prefix
            operator, it increments a variable and returns
            the incremented value.
     --     Same as ++, only it decrements.
     !=     The relational not-equal operator.  0!=1 is true,
            but 0!=0 is false.
     ==     The relational equal operator: 0==0 is true.
     "..."  The indicated string is printed, no print or output
            statement is needed.
     exp    A naked expression is printed.  The statement
                   " i = ";i
            when i has value 845 will print
                   i = 845
            To prevent unwanted printout of integer values
            from function calls, the construct
                   f=f(a,b,c)
```

```
              will often appear, where f is a dummy variable
              which is discarded
       []     Left and right braces cannot appear in comments or
              in quotes (a bug) so all comments and quotes use
              ( and ) for array subscripts.
       ;      The semicolon is optional at the end of a line.
       if(0==0) statement; The statement is executed.  This
              redundant construct is required because of
              bugs in BC.
*/
/*
      This file contains all routines necessary to
      generate trapdoor knapsacks,  EXCEPT the
      routine r(l,h), used to generate a random number
      in the range from l (low) to h (high).  That
      is, r(l,h) is a random number satisfying
         l ≤ r(l,h) ≤h

       The main routine is m:make.  Once called, it calls upon
       other routines, as needed, to generate the trapdoor knapsack.
*/
define m(m,n,r,g,b){
auto i,e,t
/*    m:make makes the enciphering and deciphering
      keys for the iterated generalized knapsack method.
      The input parameters have the following meanings:

        m:  The size in bits of m(1), the first modulus.
        n:  The number of integers in the generalized
```

```
           trapdoor knapsack.
       r:  the repetition count, i.e., how many iterations.
       g:  The growth rate, in bits/iteration.
       b:  The bound on the x(i): 0 ≤ x(i) < b.

 Print out the arguments:  */
if(1==1)   "



 The arguments m,n,r,g,b to m:make were:
"
m
n
r
g
b
/*    We compare the "natural" growth rate
      with the growth rate g.  If the natural growth rate
      exceeds the growth rate g, then we assume that the
      natural growth rate is desired.
*/
      g=2^g
      t=(b-1)*n
      if(g<t) g=t
/*   e determines how many multiples of m(i) can be
     added to the a(i)  */
      e=g/t-1
if(0==0)      "
```

```
 The value of e is: ";e
/*  now we generate m(1) and w(1)  */
  m[1] = r(2^(m-1),2^m)
  w[1] = r(1,m[1])
/*  Now to make sure gcd(w(1),m(1)) = 1   */
  t = g(w[1],m[1])
  while(t!=1) { w[1] = w[1]/t; t=g(w[1],m[1])}
  i[1] = i(w[1],m[1])
/*  now we generate the a' vector
  (the value of c is discarded)  */
  c=c(w[1],m[1],b,n)
/*  copy it for safekeeping  */
  for(i=1;i<=n;i++) p[i]=a[i]
" basis generated
"
/*  now to generate the rest of the w and m vectors  */
   for(i=2;i<=r;i++) {
i
    m[i] = r(g*m[i-1],2*g*m[i-1])
    w[i] = r(1,m[i])
/*  Make sure gcd(w(i),m(i))=1  */
    t = g(w[i],m[i])
    while(t!=1) { w[i]=w[i]/t; t=g(w[i],m[i])}
/*  and compute the inverse of w modulo m  */
    i[i] = i(w[i],m[i])
   }
/*  Now we can generate the public enciphering vector  */
  for(i=1;i<=n;i++)  for(j=1;j<=r;j++)  {
       a[i]=a[i]*i[j]%m[j]+r(0,e)*m[j]
      " Mark! "
i
j
      }
/*  And print the public enciphering vector  */
 if(0==0)      "  The public enciphering vector is:
"
    p(a[],n)
/*  print the simple knapsack vector  */
  if(0==0)     "

  The simple (secret) knapsack vector is:
"
    p(p[],n)
  if(0==0)   "  The w vector is:
"
    p(w[],r)
if (0==0)    "  The m vector is:
"
    p(m[],r)
if(0==0)   "  The i (inverse of w) vector is:
"
   p(i[],r)
    return(0)
}
"  just passed m:make
"
define i(a,b){
```

```
auto j,i,m,t
/* This routine computes a inverse mod b */
     i=1
     j=0
     m=b
     while(0==0){
      t=b/a
      j=j-t*i
      b=b-t*a
      if(b==0) if (a!=1) " in i: invert.  gcd is not 1
"
      if(b==0) return(i)
      t=a/b
      i=i-t*j
      a=a-t*b
      if(a==0) if (b!=1) "in i: invert.  gcd is not 1
"
      if(a==0) return(j+m)
     }
}
"  just passed i:invert
"
define g(a,b){
auto i,j,k
/*  computes the gcd of a and b  */
while(0==0){
   a=a%b
   if(a==0) return(b)
   b=b%a
   if(b==0) return(a)
  }
}
"  just passed g:gcd
"
define c(w,m,b,n){
auto k,j,x,y
/*  c:create creates the n numbers involved in the simple
      generalized knapsack vector.  0≤ x(i) < b.
      Note that the array a really represents
      the a' vector.

     Note that summation over n of (b-1)*a(i) is bounded by m,
     and that the a(i) satisfy a(i) > (b-1) * summation a(j)
     for j<i.
*/
  k = m / b^n
   for(j=1;j<=n;j++){
  a[j] = r((b^(j-1)-1)*k+1,b^(j-1)*k)
          }
  return(0)
}
"  just passed c:create
"
define p(a[],n){
auto i
/*  p:print prints out an array  */
    for(i=1;i<=n;i++) a[i]
    return()
```

```
}
"  just passed p:print
"
/*   The calling sequence:  global parameters are initialized,
     and the main routine, m, is called.
     The "?" is an input statement    */
m=?
n=?
r=?
g=?
b=?
m(m,n,r,g,b)
```

XIV. EXAMPLES OF TRAPDOOR KNAPSACKS

1. Introduction

This appendix gives some example trapdoor knapsacks. The author has retained the secret deciphering keys, and the reader is challenged to break any of them. They are of marginal strength to encourage attempts to break them. Full documentation, including listings of all relevant programs and the enciphering key, are given.

2. Description

The reader is assumed to be familiar with [6]. Most of the notation and all of the concepts that follow are described there.

The program which generates trapdoor knapsack vectors has five input parameters, as well as a source of random integers. The random numbers are provided by a subroutine. This subroutine, r(l,h), accepts two arguments: a lower limit and an upper limit. It returns a random integer in the range from the lower limit to the upper limit (inclusive).

The five parameters describe:

1) n: The number of integers in the knapsack.

2) b: The range of the x[i]. $0 \leq x[i] < b$.

3) r: The number of iterations.

4) g: The "growth" of the $\underline{m}$ vector per iteration in bits.

(That is, m[i] is about $2^g \cdot$m[i-1]).

5) m: The size of m[1], the first modulus. (Note that m, as used in this context, is NOT the same as m described in [1]. The m vector is the natural generalization of m in [1].)

The reader can check his understanding of the parameters n, b, and r by examining the following program segment. If we let the a' vector be the easy to solve (secret) knapsack vector, and a be the publicly known knapsack vector, then the routine for decoding s, the weighted sum of the integers in the a vector (s = x dot a) is:

```
For j = r downto 1 do s = s*w[j] mod m[j];
For j = n downto 1 do
 Begin
    x[j]=s/a'[j];
    if(x[j]>=b)  print(" Error:  x[j] larger than b");
    s=s-x[j]*a'[j];
 End
```

Note that the first For statement converts s from the difficult to solve knapsack problem to the easy to solve knapsack problem. The vectors w and m are just the generalizations of the integers w and m used in the single iteration knapsack.

The second For statement decodes s into a weighted sum of the a'[i].

The parameters g and m are used to define the size of the integers in the knapsack problem. In particular, m gives the size in bits of m[1], while g gives the "growth rate", i.e., the increase in the size between m[i] and m[i+1]. These two parameters, taken together, define the size of m[i] for all i.

Knapsacks with the following parameters have been generated:

m	n	r	g	b
300	20	6	30	2^10
300	6	2	30	2^30
300	20	1	-	2^10
550	4	1	-	2^100

3. THE PUBLIC ENCIPHERING KEYS

The following section has the output of the trapdoor knapsack generating function. The output has been edited both for clarity and to delete the secret deciphering keys. The digits in the secret deciphering keys have been replaced with X's, thus clearly showing the size of these numbers but concealing their exact values.

Numbers which require more than one line are extended using "\" at the end of the line.

EXAMPLES OF TRAPDOOR KNAPSACKS

```
  The arguments m,n,r,g,b to m:make were:

300

20

1

0

1024


  The value of e is: 0

  The public enciphering vector is:

1512960899555444826518329480125917370077769374250424584929351 7\
00743775970020807652616728474
17524932983557645722760284190473882583927059421015819376353932\
5799065041428006760782725691
17225937970747338687231551661572161286297722869710199263697080\
78209586559761332794877506 20
91488866379290109557739402894698211130838391427430023781525692\
3811073379658376192168379165
94262820794253890320443933071875869795625142774573849428509608\
650456231106385706727836631
11388384681036095320536124150646919167647675890919002134429991\
634930205818464324471736784 82
17164561490371563702262183773738261525487190864260252517798872\
7721339534749700672462349976 5
86517801731770532543741885456541883058148324212404989101077399\
873554581967770138044930143 3
```

```
119186701029456503403777607676503715238328713833999282550625 58\
025145100295251847488517518 94
101865118719125902445514802661125264230819120354302338375835 68\
2819952836679609697874758247 4
881329005115302109324377988408380384237134207921046871400798 93\
159600161042843953468257972 1
172774912480941142404286391668965378528709166942539076767887 23\
040154755675011297758534895 61
120853773743181502434243635436062318377999079871140626702220 09\
756693178957630020942892869 61
144854309225013644359922631123635652921513059579450418353887 46\
697717157065788067754452366 80
188812617110724278852263300739136000561419647547508106848809 76\
692082944713556277244611615 91
188406892047558975311386301874185926551002733898833670282676 23\
832373192194067494407151195 9
537969098608352034327152725397920347633033706014409441139701 49\
613435517970966975698524536 4
175089600797591909054527969105623157028743406266295859213877 36\
535203120524999797539473521 19
751102425183240200500717966637499286788243371031586211328085 28\
834253104532129465889505610
186330477962209830431586145943167104325412533096500755399345 35\
521199838875662815487500104 61
```

The simple (secret) knapsack vector is:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXX

  The w vector is:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXXXX

  The m vector is:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXXXX

  The i (inverse of w) vector is:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
  The arguments m,n,r,g,b to m:make were:

550

4

1

30

1267650600228229401496703205376



  The value of e is: 0

  The public enciphering vector is:

66548921643195336226466738477434229959192804189375188792173517\

98204581021406969103971543806570426505592910710830964580119256\

57074465022305714593614695666457981294232

11684898605722586318894226889837168088364682651279852730837157\

59301391908196763856942352126002186651785657829322167674672214\

87936298749063963446686191605473239307527

28673911921164002303992927072837051623067378006874600205947505\

33516592506537471315024349010350090619979216651185755427174124\

60170897669083174687480053177867989446487 7

10698579696822998161535367373067222285227832403890470260459712\

75683530572527154813286992913241349008324990670208801873643214\

62009225444854852995289708076692908866 9071


  The simple (secret) knapsack vector is:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXX

  The w vector is:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

  The m vector is:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

  The i (inverse of w) vector is:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
  The arguments m,n,r,g,b to m:make were:

300

6

2

30

1073741824


  The value of e is: 0

  The public enciphering vector is:

13132235906108717392663385023635308161076002659005223324995682\
257575999365008083392546946106350451 46

1252848819573365692846615267688252390018551687632428729861 9131\
38058630994652495400985651831 4444714159

98959536571058166688252421388712784824420035528572633604543631\
48601662744714607478456499807853916200

12178246401570249356105744244631115771305946255369054130689551\
72592567269671309679926232524090205 4193

13359484996590370029702383299514807005137667883850037475621696\
08420698777630026065126291976167628565 1

10605380625130962048597032199209554360229746160482647791377347\
41511373942928122586296363286818933 1960

  The simple (secret) knapsack vector is:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXX
```

The w vector is:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

The m vector is:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

The i (inverse of w) vector is:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
 The arguments m,n,r,g,b to m:make were:

300

20

6

30

1024


  The value of e is: 52479

  The public enciphering vector is:

387157309925544861212724057073023089001902543623810982488540 37\
020390891477746421823026786340592727484603216705226541857890 46\
69317062601758874

459634814130622708973369804711317878034798168375637048716349 29\
644125458244000344701412074819255145580649558145015813906498 68\
25526460022708092

473793058029246774055599849164244270317397791297070904056824 03\
719100704697981711088692790888316769244408037435044189760095 29\
4154941724990513

841474941791722954283750317580128987404428093713059371210657 06\
845777510608239142779226296475222046846177115338954155162151 44\
0356446225485422

355476045019644943705720102306139628278033720397186564977385 49\
705116394324082348574374159048564674262986058285691255716586 31\
23689858301771567

263278740021517231922217948690644977564245079198723931253737 81\
23867772453949577638673204672488027113486979841148992157446034\
95311577477033299
44137427640458270099948143106280924336347013911498245284167859\
96607761771476233994006770106080475456661015482830880038635956\
18128469849886865
82869734506728408857670646187367840701450363010442084975363677\
76927489127818427124735230413757953097918761358229919317923335\
3546850589724563
14435783033057224190224067526217500711045449689908289174307459\
73420352036464461078659313878916939486093236699367612785917679\
78264370706748596
76219518479176804330617483693702169070570461167513995951771603\
15437650600544017420217970789135573545530048922944206333030398\
0826452486752343
39057907333091628756443751313956792336454871430361636888411903\
66734535303428276644443984266324863268312425515094793334960194\
96299472093280176
33713269934573232136020826742991836655590472158848681940828486\
25686782702318796654172703957782207550398421357982260463217571\
00121265799943710
69921134220631872208059007887490035018668902960747850777141543\
19049543017365942287891801006275473440066900746652591567349236\
8805266202636565
51196238094281378251652093753017210817286419746759610742659928\
94616199507515513250496229763102416668715378488669390981450426\
56875350837400915
```

```
4472488362092822209078451956282183748459067235805042732293476\
97494151350493273373028039004676025506580949966000913729267884\
31749339300260941
852063443300675810868373309306672573839680380592478707260768 69\
078807278019728562949289721645927677025348383640992751935293 11\
43754226678945
389700251656853222930701545077307441532726944411273565460883 01\
190439230711040221386257003931029136742851532933839172263636 19\
263555816888896977
357350161416278353785968771967288687851454573555641206065953 59\
939933944066224911447462369393218564300458899919275829146966 31\
91555560301521980
237506060010165882813699912251331898155401833121651503559187 05\
143607626789753497526703466748523630301359635481923176452141 76\
45809506052808766
128113166611040851026735729027686735374923048082756165273426 62\
606929738846031590298150087260450889711279567242231433198367 01\
11913499359574243
0
```

The simple (secret) knapsack vector is:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXX
```

The w vector is:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXX
```

The m vector is:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXX
```

The i (inverse of w) vector is:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX\
XXXXXXXXX
```