A CERTIFIED DIGITAL SIGNATURE

Ralph C. Merkle
Xerox PARC
3333 Coyote Hill Road,
Palo Alto, Ca. 94304
merkle@xerox.com
(Subtitle: That Antique Paper from 1979)

Abstract

A practical digital signature system based on a conventional encryption function which is as secure as the conventional encryption function is described. Since certified conventional systems are available it can be implemented quickly, without the several years delay required for certification of an untested system.

Key Words and Phrases: Public Key Cryptosystem, Digital Signatures, Cryptography, Electronic Signatures, Receipts, Authentication, Electronic Funds Transfer.

CR categories: 3.56, 3.57, 4.9

1. *Introduction*

Digital signatures promise to revolutionize business by phone (or other telecommunication devices)[1] but currently known digital signature methods [5,6,7,8,10,13] either have not been certified, or have other drawbacks. A signature system whose security rested 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[1][10] suggested such a system, but it has severe performance drawbacks. Lipton and Matyas[4] nonetheless suggested its use as the only near term solution to a pressing problem.

This paper describes a digital signature system which is "pre-certified," generates signatures of about 1 to 3 kilobytes (depending on the exact security requirements), requires a few thousand applications of the underlying encryption function per signature, and only a few kilobytes of

---

This work was partially supported under contracts F49620-78-C-0086 from the U.S. Air Force Office of Scientific Research and DAAG29-78-C-0036 from the U.S. Army Research Office. Much of this work was done when the author was at Stanford University in the Electrical Engineering Department, and some was done when the author was at BNR in Palo Alto.

Copyright (c) 1998, Springer-Verlag

memory. If the underlying encryption function takes 10 microseconds to encrypt a block, generating a signature might take 20 milliseconds.

The new signature method is called a "tree signature." The following major points are covered:

1.) A discussion of one way functions.
2.) A description of the Lamport-Diffie one time signature.
3.) An improvement to the Lamport-Diffie one time signature.
4.) The Winternitz one time signature.
5.) A description of tree signatures.

2. *One Way Functions*

One way functions[2,9] are basic to this paper. Intuitively, a one way function F is one which is easy to compute but difficult to invert. 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.

Readers interested only in getting the gist of this paper are advised to skip this section and continue with section 3.

We will parameterize F, i.e., create a family of one way functions F_1, F_2, F_3 ... F_i ..., to improve security. It is easier to analyze a single function which is used repeatedly than it is to analyze all the different F_i. Often it is desirable for F_i to also compress a large input (e.g. 10,000 bits) into a smaller output (e.g. 100 bits). This will be referred to as a one way hash function and it is required that, for all i:

1.) F_i can be applied to any argument of any size.
2.) F_i always produces a fixed size output, which, for the sake of concreteness, we can assume is 100 bits.
3.) Given x it is easy to compute $F_i(x)$.
4.) It is computationally infeasible to find $x' \neq x$ such that $F_i(x) = F_i(x')$.
5.) Given $F_i(x)$ it is computationally infeasible to determine x.

An important point of notation: when we wish to concatenate two arguments x_1 and x_2, we will write $<x_1,x_2>$. Thus, if x_1 and x_2 are both 100 bits long, $<x_1,x_2>$ will be their 200 bit concatenation.

The major use of one way functions is for authentication. If a value y can be authenticated, we can authenticate x by computing:

$F_i(x) = y$

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

Copyright (c) 1998, Springer-Verlag

system involves tradeoffs between the reduced cost and improved efficiency of a smaller size, and the improved security of a larger size.)

Functions such as F_i can be defined in terms of conventional cryptographic functions[6]. We therefore assume we have a conventional encryption function C(key,plaintext) which has a 300 bit key size and encrypts 100 bit blocks of plaintext into 100 bit blocks of ciphertext.

In order to prove that F_i is a good one way function, we must make some assumptions about the conventional cryptographic function on which it is based (Rabin has also considered this problem[13]). In particular, we require that it possess certain properties.

A "certified" encryption function C(k,p) = c, in which length(p) = length(c) $\leq$ length(k), must have the following properties:

1.) The average computational effort required to find any four values k, k', p, and c such that C(k,p) = c = C(k',p) and $k \neq k'$ is greater than $2^{\text{length}(p)/2}$.

2.) The average computational effort required to find four values k, k', p, and c such that C(k,p) = c = C(k',p) and $k \neq k'$ is $2^{\text{length}(p)-1}$ if the following conditions hold:

   a.) The plaintext, p, is known and fixed.
   b.) The key space is divided into mutually disjoint subsets S_1, S_2, ...
   c.) k is an element of the set $\{k_1, k_2, \ldots\}$
   d.) Each k_i is randomly chosen from S_i.
   e.) Each S_i must have at least $2^{\text{length}(p)}$ elements.
   f.) both k and k' must be elements of the same subset S_i.

For the rest of this paper, these will be referred to as "property 1" and "property 2."

Property 1 is rather clear: finding two keys k and k' for the same plaintext-ciphertext pair requires a certain minimum computational effort under all circumstances.

Property 2 requires more explanation. It states that finding two keys k and k' for the same plaintext ciphertext pair requires a full exhaustive search IF certain conditions are satisfied. (Notice that property 1, which applies unconditionally, states that the required effort to find k and k' is proportional to the square root of a simple exhaustive search.)

The most important condition is 2d: k must be randomly chosen. If k is chosen randomly, then c = C(k,p) should also be random. Given a random c, the problem of finding a k' such that C(k',p) = c should require a full

Copyright (c) 1998, Springer-Verlag

exhaustive search.

The additional conditions can be interpreted as meaning that encryption of two plaintexts with two keys from two disjoint key spaces is effectively equivalent to encryption with two unrelated ciphers: knowledge of how to cryptanalyze messages enciphered with keys from one space will be of no help in cryptanalyzing messages enciphered with keys from the other key space. The main reason that F is parameterized is to take advantage of this aspect of property 2. If $i \neq j$, then F_i and F_j are separate one way functions: breaking F_i and breaking F_j are two independent problems. If F were not parameterized, then the many applications of F by many different people to different arguments would constitute a single interrelated problem. The problem of reversing some application of F to one of many possible arguments would be much easier to solve than the problem of reversing a particular application of F to a particular argument. This entire issue can be avoided by parameterization.

Both properties 1 and 2 will be satisfied if C is a "random cipher," a concept described by Shannon [12]. The strength of modern encryption functions is based on their resemblance to random ciphers: to quote Feistel's [11] description of Lucifer, "As the input moves through successive layers the pattern of 1's generated is amplified and results in an unpredictable avalanche. In the end the final output will have, on the average, half 0's and half 1's,..."

Should ciphers that do not satisfy properties 1 and 2 be called "certified?" This is largely a question of the appropriate definition of the term. It seems prudent to demand that a cipher not be considered certified if it fails to satisfy either property 1 or 2: the author would certainly be reluctant to use such a cipher for any purpose.

The reader should note that property 1 is much more robust than property 2: designing systems which depend on property 2 requires special care.

We will define F_i in stages: first we define the one way function $G_{<i,j>}$, which satisfies properties 2, 3, 4, and 5; but whose input is restricted to 200 bits or less. We define

$$G_{<i,j>}(x) = y = C(<x,i,j>, \underline{0})$$

$G_{<i,j>}$ accepts up to a 200 bit input x, 50 bit parameters i and j, and produces a 100 bit output y, as desired. Furthermore, given y the problem of finding an x' such that $G_{<i,j>}(x') = y$ is equivalent to finding a key x' such that $y = C(<x',i,j>, \underline{0})$.

Copyright (c) 1998, Springer-Verlag

If C satisfies properties 1 and 2 this is computationally infeasible.

We can now define F_i in terms of $G_{<i,j>}$. If the input x to F_i is 100 bits or less, then we can "pad" x by adding 0's until it is exactly 100 bits, and define $F_i(x) = G_{<i,1>}(<\underline{0},x>)$. (Where $\underline{0}$ is 100 bits of 0).

If the input is more than 100 bits, we will break it into 100 bit pieces. Assume that

$$\underline{x} = <x_1, x_2, \ldots x_n>$$

and that each x_k is 100 bits long. Then F_i is defined in terms of repeated applications of $G_{<i,j>}$. $G_{<i,1>}$ is first applied to x_1 to obtain $y_1 = G_{<i,1>}(<\underline{0},x_1>)$. Then $y_2 = G_{<i,2>}(<y_1,x_2>)$, $y_3 = G_{<i,3>}(<y_2,x_3>)$, $y_4 = G_{<i,4>}(<y_3,x_4>)$, ... $y_j = G_{<i,j>}(<y_{j-1},x_j>)$, ... $y_n = G_{<i,n>}(<y_{n-1},x_n>)$. $F_i(\underline{x})$ is defined to be y_n; the final y in the series.

It is obvious that F_i can accept arbitrarily large values for x. It is less obvious (though true) that it is computationally infeasible to find any vector $\underline{x}'$ not equal to $\underline{x}$ such that $F_i(\underline{x}) = F_i(\underline{x}')$. We shall call finding such an $\underline{x}'$ as "breaking" F_i.

If we assume that C is a certified encryption function, i.e., that property 1 or 2 holds, we can prove inductively that breaking F_i is computationally infeasible. If we utilize assumption 1 we can prove that the average effort required to compute $\underline{x}'$ will be at least $2^{length(p)/2}$; while if we use assumption 2 we can prove that the average effort required to compute $\underline{x}'$ will be at least $2^{length(p)-1}$, although we require that x' be random.

As a basis, when n = 1 the property holds because, by definition, $F_i(\underline{x}) = G_{<i,1>}(<\underline{0},x_1>) = C(\ <\underline{0},x_1,i,1>,\underline{0}\ )$ and the property holds for C by assumption. To show that the property must hold for n+1 if it holds for n, we need only note that if $F_i(\underline{x}) = F_i(\underline{x}')$, then one of the following two conditions must hold:

Copyright (c) 1998, Springer-Verlag

A.) $x_k = x'_k$ for all $k \leq n$

B.) $x_k \neq x'_k$ for some $k \leq n$

If (B) holds, then by the induction hypothesis we have already spent the required effort to compute $x_k \neq x'_k$, for some $k \leq n$.

If (A) holds and $\underline{x} \neq \underline{x}'$, then $x_{n+1} \neq x'_{n+1}$. The effort required to compute x'_{n+1} not equal to x_{n+1}, but with $G_{<i,n+1>}(<y_n,x'_{n+1}>)$ equal to $G_{<i,n+1>}(<y_n,x_{n+1}>)$ must be $2^{length(p)/2}$ (if we use property 1), or $2^{length(p)-1}$ (if we use property 2), by definition of $G_{<i,n+1>}$ and properties 1 and 2.

In those cases where the conditions of property 2 do not hold, property 1 will.

It is important in practice to distinguish between those cases where property 2 can be used, and those which can use only property 1. The use of property 2 allows the size of the block cipher to be reduced by a factor of two, while still maintaining the same level of security. This will lead to a factor of two reduction in most storage and transmission costs in the following algorithms.

To clarify further explanations we will omit the subscript from F in the rest of the paper, but the reader should remember that parameterizing F is essential to take advantage of property 2. If property 1 is used, it is still advisable to parameterize F.

3. *The Lamport-Diffie One Time Signature*

The Lamport-Diffie one time signature[1] is based on the concept of a one way function[2,9]. 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 can know x, and can choose to reveal or conceal x at his whim.

Copyright (c) 1998, Springer-Verlag

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[4] 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 sold the stock, B can show the contract and x to a judge as proof that A, contrary to his statement, did sell the stock. 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 could possibly know x would be 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 compute $F(x_1) = y_1$, $F(x_2) = y_2$, $F(x_3) = y_3$,... $F(x_s) = y_s$. 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 be mapped into short messages (100 bits) by a one way function and only the short message signed. It is always possible to use property 2 (described in section 2). F can be parameterized as F_i (also described in section 2), the message can be encrypted with a newly generated random key by the signer before it is signed, and the random key appended to the message. The signed message will therefore be random (assuming that encryption with a random key will effectively randomize the message, a fact that is generally conceded for modern encryption functions [11]). These steps will satisfy the conditions for property 2. 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,

Copyright (c) 1998, Springer-Verlag

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 $Y = y_1, y_2, \cdots y_{2*s}$ can only be used to sign one message. If more than one message is to be signed, then new values $Y_1, Y_2, Y_3, \ldots$ 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. (Rabin [13] has described a different one time signature method).

Between two people, A and his broker B for example, a signature system for n possible messages might be designed as follows. A would compute

$$
\begin{aligned}
Y_1 &= y_{1,1}, y_{1,2} \cdots y_{1,2*s} \\
Y_2 &= y_{2,1}, y_{2,2} \cdots y_{2,2*s} \\
Y_3 &= y_{3,1}, y_{3,2} \cdots y_{3,2*s} \\
&\vdots \\
Y_n &= y_{n,1}, y_{n,2} \cdots y_{n,2*s}
\end{aligned}
$$

(where $y_{i,j} = F(x_{i,j})$, and the $x_{i,j}$ are chosen randomly). However, prior to using this method, A and B would have to agree that $\underline{Y} = Y_1, Y_2 \ldots Y_n$ was to be used for signatures, and B would have to have a copy of $\underline{Y}$. ($\underline{Y}$ would have to be authenticated in some fashion so it could be shown to a judge in the event of a dispute, and proven to be the $\underline{Y}$ that both A and B agreed on.) If each $y_{i,j}$ is 100 bits long, if s = 100, and if n = 1000 (i.e., 1000 possible messages can be signed, each 100 bits in length) then $\underline{Y}$ will be n * 2 * s * 100 = 1000 * 2 * 100 * 100 = 20,000,000 bits or 2.5 megabytes. While this might not be 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 paper.

Copyright (c) 1998, Springer-Verlag

4. *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. It can be skipped without loss of continuity.

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 would accomplish the same result would be to append a count of the 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, it would produce an illegal codeword by either increasing the number of 0's in m, and thus make the count of 0's too small, or it would alter the count of 0's. If the count of 0's is in standard binary, changing a bit in this count from 1 to 0 must reduce the count, and hence result in an illegal codeword. 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. But 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

Copyright (c) 1998, Springer-Verlag

message almost in half.

5. *The Winternitz Improvement*

Shortly before publication[e.g., in 1979], Robert Winternitz of the Stanford Mathematics Department suggested a further substantial improvement which reduces the size of the signed message by an additional factor of about 4 to 8. Winternitz's method trades time for space: the reduced size is purchased with an increased computational effort.

In the Lamport-Diffie method, given that $y = F(x)$ and that y is public and x is secret, a user signs a single bit of information by either making x public or keeping it secret.

In the Winternitz method we still use y and x, and make y public and keep x secret, but we compute y from x by applying F repeatedly, for example, $y = F^{16}(x)$. This allows us to sign 4 bits of information (instead of just 1) with the single y value. To sign the 4 bit message 1001 (9 in decimal), the signer makes $F^9(x)$ public. Anyone can check that $F^7(F^9(x)) = y$, thus confirming that $F^9(x)$ was made public, but no one can generate that value.

Because $F^9(x)$ is public, $F^{10}(x)$ can be easily computed by anyone. Someone could then (falsely) claim that the signed four bit message was 1010 (10 in decimal) rather than 1001. Overcoming this problem requires a slight extension of the method described in section 4, and adds only log n additional bits.

6. *Tree Authentication*

A new protocol would eliminate the large storage requirements. 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. 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 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.

Tree authentication can also be used to solve authentication problems which do not involve digital signatures: that it is being used to generate tree signatures in this paper should not prejudice the reader into thinking that that is its only application.

Copyright (c) 1998, Springer-Verlag

Problem Definition: Given a vector of data items $\underline{Y} = Y_1, Y_2, \ldots Y_n$ 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 $Y_1, Y_2, \ldots Y_n$.

To authenticate the Y_i we apply the "divide and conquer" technique. Define the function $H(i,j,\underline{Y})$ as follows:

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

2.) $H(i,j,\underline{Y}) = F(< H(i,(i+j-1)/2,\underline{Y}), H((i+j+1)/2,j,\underline{Y}) >)$

$H(i,j,\underline{Y})$ is a function of $Y_i, Y_{i+1}, \ldots Y_j$. $H(i,j,\underline{Y})$ can be used to authenticate Y_i through Y_j. $H(1,n,\underline{Y})$ can be used to authenticate Y_1 through Y_n. $H(1,n,\underline{Y})$ is only 100 bits, so it can be conveniently stored. This method lets us selectively authenticate any "leaf," Y_i, that we wish. To see this, we use an example where $n = 8$. The sequence of recursive calls required to compute $H(1,8,\underline{Y})$ is illustrated in Figure 1. To authenticate Y_5, we can proceed in the following manner:

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 $H(1,8,\underline{Y}) = F(< H(1,4,\underline{Y}), H(5,8,\underline{Y}) >)$ and confirm 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 $H(5,8,\underline{Y}) = F(< H(5,6,\underline{Y}), H(7,8,\underline{Y}) >)$ and confirm 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 $H(5,6,\underline{Y}) = F(< H(5,5,\underline{Y}), H(6,6,\underline{Y}) >)$ and confirm they are correct.

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

Copyright (c) 1998, Springer-Verlag

be transmitted, either. (The receiver *must* compute these quantities anyway for proper authentication.) Therefore, to authenticate Y_5 only required 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.

The proof that the authentication path actually authenticates the chosen leaf is similar to the proof in section 2 that F(x) correctly authenticates x, and will not be repeated. It is important to decide whether property 1 or property 2 should be used: if property 1 is used the size of the authentication path must be doubled to preserve the same level of security. This choice depends on whether we trust the person who first computed the authentication tree. If we do, then property 2 can be used. If we don't, then property 1 must be used. This is because property 1 is independent of the method of computation. Property 2 requires random selection, and can be subverted by non-random choices.

The use of tree authentication to create tree signatures 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 a signed message from A as genuine.

If the jth 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 1-200 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.

Because it is impossible to add new leaves (representing new users) to the "user tree" once it has been computed, it is necessary to compute and issue new user trees periodically. It is precisely this "inflexibility" which makes it unnecessary to trust the central clearinghouse. If it is impossible to add new users, it is impossible to add imposters. On the other hand, any system which allows new users to be added quickly, easily, and conveniently can be subverted by quickly, easily, and conveniently adding an imposter.

Copyright (c) 1998, Springer-Verlag

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. This has the virtue of convenience, but requires that the clearinghouse be trusted not to (secretly) sign false letters of reference. Kohnfelder[3] has suggested this method for use with other public key cryptosystems.

A full discussion of the protocols for using tree authentication, digital signatures and one time signatures is well beyond the scope of this paper.

7. *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 can be made quite modest.

In addition, although the X_i (from which the Y_i are generated) 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 X_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.

Copyright (c) 1998, Springer-Verlag

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 is this "efficient?" Before we can answer this question and determine the cost of computing these entries, we must decide on the units to be used in measuring this "cost." Because all computations must eventually be defined in terms of the underlying encryption function C(key,plaintext), it seems appropriate to define computational cost 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 k * 100 bits, then F requires k-1 ets.

Copyright (c) 1998, Springer-Verlag

First, we must determine the cost of computing the individual entries. The algorithm for H(i,j,$\underline{Y}$) 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). There are j-i non-leaf nodes, so the computation requires j-i ets, excluding the leaves. The computations required to regenerate a leaf 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) * r ets. If r is large, we can approximate this by (j-i+1) * 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 * 8 * r = 32 * r ets. In general, the computational effort required to compute table 2 will be n * (1 + $\log_2$ n) * r ets. This is because computing all the entries in each column will require n * r ets, and there are 1 + $\log_2$ n columns.

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

$$\log_2 n * r \qquad (1)$$

ets per path, where r is a constant representing the number of ets required to regenerate a leaf. This is quite reasonable. (The peak computational load is also reasonable, as will be seen in the next two paragraphs).

Although the time required to generate each authentication path is small, we must also insure that the space required is small. 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 all at once -producing a high peak load- 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. 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. Once this precomputation is complete, the succeeding values of H() required in succeeding authentication paths must be incrementally computed. As we

Copyright (c) 1998, Springer-Verlag

generate the first 4 authentication paths, we must be continuously and incrementally computing H(1,4,$\underline{Y}$) so that it will be available when we reach Y_5. In addition, we must start computing H(1,2,$\underline{Y}$) when we generate the first authentication path; we must start computing H(7,8,$\underline{Y}$) when we reach Y_3; we must start computing H(5,6,$\underline{Y}$) when we reach Y_5; and so on.

By incrementally computing the H() values required in the authentication paths, we insure that the peak computational effort is low ($O(\log_2 n)$ per authentication path) as well as the average computational effort.

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. The memory required for each column will be about the memory required during the computation of the next entry. This means the total memory required will be about: $3 + 2 + 1 = 9$ blocks. (This assumes we do not recompute H(1,8,$\underline{Y}$)).

There are $\log_2 n$ columns and each column requires, on an average, $(\log_2 n)/2$ blocks. The total memory required will be about:

Copyright (c) 1998, Springer-Verlag

$(\log_2 n)^2/2$ blocks

This means that the memory required when $n = 2^{20}$ (1,048,576) is about 20*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.

This algorithm can be described by 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. Note that n must be a power of 2.

```
Declare flag: array[0..log2(n)-1] of integer;
      AP:   array[0..log2(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
  (* Note that in a real implementation F must be
     parameterized as described in section 2 *)
  If a = b Return(F(MakeY(a)))
  Else
  Return( F(< H(a,(a+b-1)/2),H((a+b+1)/2,b) >) );
End;


(* The main program *)
Begin
```

Copyright (c) 1998, Springer-Verlag

```
  For i := 0 to log2(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 log2(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, though, that each process has adequate time to compute its next output. This follows from the observation that a single call to "Emit" will generate enough output for 2^i authentication paths, while the time required to compute the next entry is approximately 2^i.

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,

Copyright (c) 1998, Springer-Verlag

the total memory required can be reduced.

All three approaches deserve more careful study: the potential savings in time and space might be large.

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 (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 messages are all 100 bits, then the X_i vectors will have $100 + \log_2 100 = 107$ elements. (Longer messages can be mapped into a 100 bit message space using one way functions as described in section 2.)

The X_i vectors can be generated using a conventional cipher, C(key,plaintext). A single 300 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 less. We can now define $x_{i,j}$ as

$$x_{i,j} = C(\text{seedkey}, <i,j>)$$

(Where "seedkey" is the 300 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 $x_{i,1}, x_{i,2}, \ldots x_{i,107}$.

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 certified 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 (1), we know that the cost per authentication path is $\log_2 n$ * 320 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

Copyright (c) 1998, Springer-Verlag

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 (due to the complexity of the improvement, however, it will not be described). (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.

8. *CONCLUSION*

Digital signature systems not requiring public key cryptosystems are not only possible, they can be easier to certify. Such a system was described which had modest space and time requirements and a signature size of from 1 to 3 kilobytes. The method described can be implemented quickly, without the long delays due to certification.

9. *ACKNOWLEDGEMENTS*

It is a great pleasure for the author to acknowledge the pleasant and informative conversations he had with Dov Andelman, Whitfield Diffie, John Gill, Martin Hellman, Raynold Kahn, Loren Kohnfelder, Leslie Lamport, and Steve Pohlig.

10. *BIBLIOGRAPHY*

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

2. 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.

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

4. Lipton, S.M., and Matyas, S.M. Making the digital signature legal--and

Copyright (c) 1998, Springer-Verlag

safeguarded. Data Communications (Feb. 1978), 41-52.

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

6. Merkle, R. Secure Communications over Insecure Channels. Comm. ACM 21, 4(Apr. 1978), 294-299.

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

8. 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.

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

10. Lamport, L., Constructing digital signatures from a one way function. SRI Intl. CSL - 98

11. Feistel, H., Cryptography and computer security. Scientific American, 228(May 1973), 15-23.

12. Shannon, C.E., Communication theory of secrecy systems. Bell Sys. Tech. Jour. 28(Oct. 1949) 656-715.

13. Rabin, M.O., Digitalized signatures. In *Foundations of Secure Computation*, R. Lipton and R. DeMillo, Eds., Academic Press, New York, 1978, pp. 155-166.

ADDENDUM

This article was originally submitted to Ron Rivest, then editor of the Communications of the ACM, in 1979. It was accepted subject to revisions, and was revised and resubmitted in November of 1979. Unfortunately, Ron Rivest passed over the editorship to someone else, the author became involved in a startup, and the referees reportedly never responded to the revised draft. The version printed here is the final revised version submitted to CACM in 1979. The only change (besides formatting) is the author's affiliation. Then, he was at BNR in Palo Alto, CA. Now, he is at Xerox PARC in Palo Alto, CA.

Copyright (c) 1998, Springer-Verlag