Comments in 2012 about the 1979 paper: A Certified Digital Signature

This paper was originally submitted to Ron Rivest, then editor at 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. This version is the final revised version submitted to the CACM in 1979.

It was finally published in Crypto 89, page 218. The slides following the paper are from the author’s conference presentation.

An implementation that uses most of the ideas described here, along with additional ideas developed later, is available at ftp://ftp.parc.xerox.com/pub/hashsig/hashSig1.0/. The reader is referred in particular to ftp://ftp.parc.xerox.com/pub/hashsig/hashSig1.0/signCheck1.0.c and the comments therein. The reader is also referred to patents #4,881,264, Digital Signature System and Method Based on a Conventional Encryption Function, and #4,309,569, Method of Providing Digital Signatures.

The following papers are also relevant:
"One Way Hash Functions and DES" by Ralph C. Merkle, Crypto '89.
"A Digital Signature Based On A Conventional Encryption Function" by Ralph C. Merkle, Crypto '87.
"Design Principles for Hash Functions" by Ivan Bjerre Damgard, Crypto '89.
"Constructing Digital Signatures from a One Way Function" by Leslie Lamport, Computer Science Laboratory Technical Report October 18, 1979, CSL-98

Subsequent excellent work by Michael Szydlo, then at RSA Labs, reduced memory requirements from log² n to log n, where n is the number of messages that can be signed. See M. Szydlo, "Merkle Tree Traversal in Log Space and Time," Eurocrypt '04.

It now appears that quantum computers are feasible, and will be implemented at some point in the future. While it is at present unclear how long we have before they will be able to attack problems of cryptographic relevance, it seems clear that we should begin developing and deploying quantum resistant systems as soon as feasible. The lead times involved in developing, certifying, standardizing, debugging, adopting, and distributing new systems is so long, and the time before we can reasonably expect to retire most old systems is so much longer, that even if we begin this process today it will be twenty years before the last vulnerable systems are retired from service.

Given these time frames, research aimed at developing deployable implementations of tree signatures should begin now, as their performance is acceptable for many applications and the available alternatives, assuming that quantum computers become available, are very limited.
A CERTIFIED DIGITAL SIGNATURE

Ralph C. Merkle
BNR Inc.,
Palo Alto, Ca.
94304

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

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.

12/14/79 Page No. 1
A CERTIFIED DIGITAL SIGNATURE

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 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."
A CERTIFIED DIGITAL 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 $F_1, F_2, F_3 \ldots F_i \ldots$, 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
A CERTIFIED DIGITAL SIGNATURE

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 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
A CERTIFIED DIGITAL SIGNATURE

function on which it is based. In particular, we require that it possess certain properties.

A "certified" encryption function \( C(k,p) = c \), in which \( \text{length}(p) = \text{length}(c) \leq \text{length}(k) \), 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 \( \{k_1, k_2, ... \} \)
   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 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
A CERTIFIED DIGITAL SIGNATURE

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

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

\( G_{<i,j>} \) accepts up to a 200 bit input \( x \), 50 bit parameters \( i \) and
A CERTIFIED DIGITAL SIGNATURE

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 \( y = C(x',i,j,0) \).

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 \( F_i(x) = G_{i,1}(x) \).

(Where 0 is 100 bits of 0).

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

\[ x = \langle x_1, x_2, \ldots, x_n \rangle \]

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 \( y_1 = G_{i,1}(0,x_1) \). Then \( y_2 = G_{i,2}(y_1,x_2) \), \( y_3 = G_{i,3}(y_2,x_3) \), \( y_4 = G_{i,4}(y_3,x_4) \), \ldots \( y_j = G_{i,j}(y_{j-1},x_j) \), \ldots \( y_n = G_{i,n}(y_{n-1},x_n) \). \( F_i(x) \) 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 \( x' \) not equal to x such that \( F_i(x) = F_i(x') \). We shall call finding such an \( x' \) 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 as-
A CERTIFIED DIGITAL SIGNATURE

Assumption 1 we can prove that the average effort required to compute \( x' \) will be at least \( 2^{\text{length}(p)/2} \); while if we use assumption 2 we can prove that the average effort required to compute \( x' \) will be at least \( 2^{\text{length}(p)-1} \), although we require that \( x' \) be random.

As a basis, when \( n = 1 \) the property holds because, by definition, \( F_i(x) = G_{i,1}(<0,x_1>) = C(<0,x_1,i,1>,0) \) 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 \( F_i(x) = F_i(x') \), then one of the following two conditions must hold:

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 \( x \neq x' \), 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 \( 2^{\text{length}(p)/2} \) (if we use property 1), or \( 2^{\text{length}(p)-1} \) (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
A CERTIFIED DIGITAL SIGNATURE

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.

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
A CERTIFIED DIGITAL SIGNATURE

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(x₁) = y₁, F(x₂) = y₂, F(x₃) = y₃, ... F(xₛ) = yₛ. A and B would agree on the vector Y = y₁, y₂ ... yₛ. If the jth bit of m was a 1, A would reveal xⱼ. If the jth bit of m was a 0, A would not reveal xⱼ. 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 F₁ (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.
A CERTIFIED DIGITAL SIGNATURE

The signed message will therefore be random (assuming that en-
cryption with a random key will effectively randomize the mes-
sage, 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 generali-
ty, 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 Dif-
fie 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 comple-
ment 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 \( Y = y_1, y_2, \ldots, y_{2^s} \) 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 \( Y_1 \) for each mes-
sage.

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 dif-
ferent 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

\[ Y_1 = y_{1,1}, y_{1,2} \ldots y_{1,2s} \]
\[ Y_2 = y_{2,1}, y_{2,2} \ldots y_{2,2s} \]
\[ Y_3 = y_{3,1}, y_{3,2} \ldots y_{3,2s} \]
\[ \cdots \]
\[ Y_n = y_{n,1}, y_{n,2} \ldots y_{n,2s} \]

(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 \( Y = Y_1, Y_2 \ldots Y_n \) was to be used for signatures, and B would have to have a copy of \( Y \). (\( Y \) 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 \( Y \) 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 \( Y \) will be \( n \times 2 \times s \times 100 = 1000 \times 2 \times 100 \times 100 = 20,000,000 \) 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
A CERTIFIED DIGITAL SIGNATURE

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.
A CERTIFIED DIGITAL SIGNATURE

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
A CERTIFIED DIGITAL SIGNATURE

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 message almost in half.
5. The Winternitz Improvement

Shortly before publication, 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.
Problem Definition: Given a vector of data items $Y = Y_1, Y_2, \ldots Y_n$ 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 $H(i,j,Y)$ as follows:

1.) $H(i,i,Y) = F(Y_i)$

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

$H(i,j,Y)$ is a function of $Y_i$, $Y_{i+1}$, $\ldots Y_j$. $H(i,j,Y)$ can be used to authenticate $Y_i$ through $Y_j$. $H(1,n,Y)$ can be used to authenticate $Y_1$ through $Y_n$. $H(1,n,Y)$ 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 $H(1,8,Y)$ is illustrated in Figure 1. To authenticate $Y_5$, we can proceed in the following manner:

1.) $H(1,8,Y)$ is already known and authenticated.

2.) $H(1,8,Y) = F(<H(1,4,Y), H(5,8,Y)>$. Send $H(1,4,Y)$ and $H(5,8,Y)$ and let the receiver compute $H(1,8,Y) = F(<H(1,4,Y), H(5,8,Y)>)$ and confirm they are correct.
A CERTIFIED DIGITAL SIGNATURE

3.) The receiver has authenticated $H(5,8,Y)$. Send $H(5,6,Y)$ and $H(7,8,Y)$ and let the receiver compute $H(5,8,Y) = F(<H(5,6,Y), H(7,8,Y)>)$ and confirm they are correct.

4.) The receiver has authenticated $H(5,6,Y)$. Send $H(5,5,Y)$ and $H(6,6,Y)$ and let the receiver compute $H(5,6,Y) = F(<H(5,5,Y), H(6,6,Y)>)$ and confirm they are correct.

5.) The receiver has authenticated $H(5,5,Y)$. Send $Y_5$ and let the receiver compute $H(5,5,Y) = F(Y_5)$ 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, $H(5,6,Y)$ can be computed from $H(5,5,Y)$ and $H(6,6,Y)$, so there is really no need to send $H(5,6,Y)$. Similarly, $H(5,8,Y)$ can be computed from $H(5,6,Y)$ and $H(7,8,Y)$, so $H(5,8,Y)$ need never 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 $H(1,8,Y)$, and that we transmit $Y_5$, $H(6,6,Y)$, $H(7,8,Y)$, and $H(1,4,Y)$. That is, we require $100 \ast \log_2 n$ bits of information to authenticate an arbitrary $Y_1$. 

12/14/79 Page No. 22
A CERTIFIED DIGITAL SIGNATURE

The method is called tree authentication because the computation of $H(1,n,Y)$ 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 $H(i,i,Y)$ to $H(1,n,Y)$. $H(1,n,Y)$ will be referred to as the root of the authentication tree, or $R$. The information near the path from $R$ to $H(i,i,Y)$ 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 $j$th 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 users, and compute $H(1,u,R)$. 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.

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 CERTIFIED DIGITAL SIGNATURE

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 $H(i,j,Y)$ for many different values of $i$ and $j$. In the example, it was necessary to compute $H(6,6,Y)$, $H(7,8,Y)$, and $H(1,4,Y)$ and send them to B along with $Y_5$. This is simple for the small tree used in our example, but computing $H(4194304,8388608,Y)$ just prior to sending it would be an intolerable burden. One obvious solution would be to precompute $H(1,n,Y)$ 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.

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

<table>
<thead>
<tr>
<th>leaf</th>
<th>authentication path</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_1$</td>
<td>$H(1,8,Y)$ $H(5,8,Y)$ $H(3,4,Y)$ $H(2,2,Y)$</td>
</tr>
<tr>
<td>$Y_2$</td>
<td>$H(1,8,Y)$ $H(5,8,Y)$ $H(3,4,Y)$ $H(1,1,Y)$</td>
</tr>
<tr>
<td>$Y_3$</td>
<td>$H(1,8,Y)$ $H(5,8,Y)$ $H(1,2,Y)$ $H(4,4,Y)$</td>
</tr>
<tr>
<td>$Y_4$</td>
<td>$H(1,8,Y)$ $H(5,8,Y)$ $H(1,2,Y)$ $H(3,3,Y)$</td>
</tr>
<tr>
<td>$Y_5$</td>
<td>$H(1,8,Y)$ $H(1,4,Y)$ $H(7,8,Y)$ $H(6,6,Y)$</td>
</tr>
<tr>
<td>$Y_6$</td>
<td>$H(1,8,Y)$ $H(1,4,Y)$ $H(7,8,Y)$ $H(5,5,Y)$</td>
</tr>
<tr>
<td>$Y_7$</td>
<td>$H(1,8,Y)$ $H(1,4,Y)$ $H(5,6,Y)$ $H(8,8,Y)$</td>
</tr>
<tr>
<td>$Y_8$</td>
<td>$H(1,8,Y)$ $H(1,4,Y)$ $H(5,6,Y)$ $H(7,7,Y)$</td>
</tr>
</tbody>
</table>

TABLE 1
A CERTIFIED DIGITAL SIGNATURE

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:

<table>
<thead>
<tr>
<th>leaf</th>
<th>authentication path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y_1</td>
<td>H(1, 8, Y) H(5, 8, Y) H(3, 4, Y) H(2, 2, Y)</td>
</tr>
<tr>
<td>Y_2</td>
<td>H(1, 1, Y)</td>
</tr>
<tr>
<td>Y_3</td>
<td>H(1, 2, Y) H(4, 4, Y)</td>
</tr>
<tr>
<td>Y_4</td>
<td>H(3, 3, Y)</td>
</tr>
<tr>
<td>Y_5</td>
<td>H(1, 4, Y) H(7, 8, Y) H(6, 6, Y)</td>
</tr>
<tr>
<td>Y_6</td>
<td>H(5, 5, Y)</td>
</tr>
<tr>
<td>Y_7</td>
<td>H(5, 6, Y) H(8, 8, Y)</td>
</tr>
<tr>
<td>Y_8</td>
<td>H(7, 7, Y)</td>
</tr>
</tbody>
</table>

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 under-
A CERTIFIED DIGITAL SIGNATURE

lying 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 etee) 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.

First, we must determine the cost of computing the individual entries. The algorithm for H(i,j,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,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 \( 4 \times 8 \times r = 32 \times r \) ets.
In general, the computational effort required to compute table 2 will be \( n \times (1 + \log_2 n) \times r \) ets. This is because computing all the entries in each column will require \( n \times r \) ets, and there are \( 1 + \log_2 n \) columns.

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

\[
\log_2 n \times r \quad (1)
\]

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 comput-
ing it well in advance of the time we will actually require it to generate an authentication path.

As an example, \( H(5,8,Y) \) is required in the authentication paths for \( Y_1, Y_2, Y_3, \) and \( Y_4 \). \( H(1,4,Y) \) 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 generate the first 4 authentication paths, we must be continuously and incrementally computing \( H(1,4,Y) \) so that it will be available when we reach \( Y_5 \). In addition, we must start computing \( H(1,2,Y) \) when we generate the first authentication path; we must start computing \( H(7,8,Y) \) when we reach \( Y_3 \); we must start computing \( H(5,6,Y) \) 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 \( H(i,j,Y) \) is \( 1 + \log_2 (j-i+1) \) blocks. This as-
A CERTIFIED DIGITAL SIGNATURE

sumes 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 $H(i,i,Y)$) 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:

<table>
<thead>
<tr>
<th>leaf</th>
<th>memory required to compute entries in authentication path (in blocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y_1</td>
<td>4 3 2 1</td>
</tr>
<tr>
<td>Y_2</td>
<td>1</td>
</tr>
<tr>
<td>Y_3</td>
<td>2 1</td>
</tr>
<tr>
<td>Y_4</td>
<td>1</td>
</tr>
<tr>
<td>Y_5</td>
<td>3 2 1</td>
</tr>
<tr>
<td>Y_6</td>
<td>1</td>
</tr>
<tr>
<td>Y_7</td>
<td>2 1</td>
</tr>
<tr>
<td>Y_8</td>
<td>1</td>
</tr>
</tbody>
</table>

**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: $3 + 2 + 1 = 9$
A CERTIFIED DIGITAL SIGNATURE

blocks. (This assumes we do not recompute H(1,8,Y)).

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

\((\log_2 n)^2/2\) 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..\log_2(n)-1] of integer;

AP: array[0..\log_2(n)-1] of block;

(* AP -- Authentication Path *)

Procedure Gen(i);

Begin

12/14/79 Page No. 33
A CERTIFIED DIGITAL SIGNATURE

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\([i]\) \( \neq 0 \) 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 *)

12/14/79

Page No. 34
A CERTIFIED DIGITAL SIGNATURE

Begin

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 log₂ n processes to deal with the log₂ n
columns. Then it prints each authentication path by sequen­
tially printing an output from each process. The major omis­
sion in this program is the rate at which each process does its
computations. It should be clear, though, that each process
A CERTIFIED DIGITAL SIGNATURE

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
A CERTIFIED DIGITAL SIGNATURE

with the minimum memory requirements of other processes, 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, \( C(\text{key},\text{plaintext}) \). 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 \( x_{i,1}, x_{i,2}, \ldots, x_{i,107} \).
A CERTIFIED DIGITAL SIGNATURE

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 $H(i,i,Y) = F(Y_1) = F(<y_{1,1}, y_{1,2}, y_{1,3}, \ldots y_{1,107}>)$. The cost of computing $F(Y_1)$ is 106 ets, because $Y_1$ is 107 * 100 bits long. The total effort to compute $H(i,i,Y)$ is the effort to generate the elements of the $X_i$ vector, plus the effort to compute $F(x_{i,1}), F(x_{i,2}), \ldots F(x_{i,n})$, plus the effort to compute $F(Y_1)$. This is 107 ets to compute the $X_i$ vector, 107 ets to compute the $Y_i$ vector, and 106 ets to compute $F(Y_1) = H(i,i,Y)$. 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 worthwhile goal. This can effectively be done by reducing either the cost of computing $H(i,i,Y)$, or by reducing the number of times that $H(i,i,Y)$ 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.
A CERTIFIED DIGITAL SIGNATURE

10. BIBLIOGRAPHY


A CERTIFIED DIGITAL SIGNATURE


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


The following slides were used in the author’s presentation at Crypto ’89
A CERTIFIED DIGITAL SIGNATURE

by

Ralph C. Merkle

Xerox PARC

Palo Alto, CA

(Work done in 1979)
SUBTITLE:

HOW MUCH MILEAGE CAN WE GET FROM ONE-WAY HASH FUNCTIONS?
WHAT'S A ONE WAY HASH FUNCTION?

There are many definitions, but for now we adopt the following:

F is a One Way Hash Function if:

1.) F accepts an input of arbitrary size
2.) F produces a fixed size output
3.) F is easy to compute
4.) Finding any pair of inputs X and X' such that

\[ F(X) = F(X') \]

is hard
WE CAN BUILD $F$ FROM A SIMPLER $F_0$

$F_0$ SATISFIES THE SAME PROPERTIES AS $F$

EXCEPT ITS INPUT SIZE IS FIXED
ASSUME THE INPUT, X, IS AN ARRAY OF 3 ELEMENTS

X[1], X[2], X[3]

DEFINITION BY PICTURE:

\[ F_0(X[i]) = \]

\[ F(X) = \]

\[ F(X) \]
EASY TO PROVE THAT BREAKING F

IMPLIED BREAKING F₀ (BY INDUCTION)

F(X) = F(X')

F₀( F(X[1..n-1]), X[n] ) = F₀( F(X'[1..n-1]), X'[n] )

Either the two inputs to F₀ are the same
 or
 they are different.

If they are the same, no problem!
[ inductively consider F(X[1..n-1]) = F(X'[1..n-1]) ]

If they are different, F₀ has been broken
(two different inputs to F₀ that produce the same output)
THE LAMPORT ONE-TIME SIGNATURE:

(WARNING! NOTATIONAL CHANGE!)

SECRET INFORMATION
(each X[i] is randomly chosen and perhaps 100 bits)

PUBLIC INFORMATION
compute Y[i] = F(X[i])
F must be hard to invert

TO SIGN THE 64-bit message M = M[1] ... M[64]
if M[i] = 0, reveal X[2*i - 1]
if M[i] = 1, reveal X[2*i]
NOTATION: $Y_i = Y_i[1] Y_i[2] \ldots Y_i[127] Y_i[128]$

THE LAMPORT SIGNATURE IS A ONE–TIME SIGNATURE

WE NEED A NEW VALUE OF $Y_i$ EACH TIME WE SIGN A MESSAGE

THIS MEANS WE NEED A LOT OF $Y_i$'s!
NOTATION: \( Y = Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7 Y_8 \)
\[ \| \text{ is bit-wise concatenation} \]

DEFINE \( H(i,j,Y) \)

IF \( i = j \) THEN \( H(i,j,Y) = F(Y_i) \)

IF \( i > j \) THEN \( H(i,j,Y) = \)
\[ F_0( H(i, (i+j-1)/2, Y) \| H((i+j+1)/2, j, Y) ) \]

NOW COMPUTE \( H(1,8,Y) \)
THE SAME INDUCTIVE PROOF CAN BE USED TO SHOW THAT BREAKING H IMPLIES A BREAK IN F₀
WHY IS A TREE USEFUL?

BECAUSE ANY "LEAF" CAN BE AUTHENTICATED WITH ONLY LOG(n) INFORMATION:

THE AUTHENTICATION PATH
A STICKY PROBLEM:

HOW TO GENERATE

THE AUTHENTICATION PATH........

(IT'S IN THE PAPER)
A TREE SIGNATURE:
SECRET INFORMATION: THE $X_{ij}$'s
PUBLIC INFORMATION: $H(1,n,Y)$

TO SIGN A MESSAGE:
REVEAL THE CORRECT SET OF $X_{ij}$'s
AND THE AUTHENTICATION PATH

TO CHECK A SIGNATURE
CHECK THE AUTHENTICATION PATH
AND CHECK THE REVEALED $X_{ij}$'s
AGAINST THE $Y_{ij}$'s
CONCLUSION:

DIGITAL SIGNATURES CAN BE BUILT FROM
ONE-WAY HASH FUNCTIONS

MORE GENERALLY
MANY AUTHENTICATION PROBLEMS CAN BE SOLVED WITH ONE-WAY HASH FUNCTIONS

F₀ IS A USEFUL CRYPTOGRAPHIC PRIMITIVE