A DIGITAL SIGNATURE BASED ON A CONVENTIONAL ENCRYPTION FUNCTION

by Ralph C. Merkle
Elxsi
2334 Lundy Place
San Jose, CA 95131

ABSTRACT

A new digital signature based only on a conventional encryption function (such as DES) is described which is as secure as the underlying encryption function -- the security does not depend on the difficulty of factoring and the high computational costs of modular arithmetic are avoided. The signature system can sign an unlimited number of messages, and the signature size increases logarithmically as a function of the number of messages signed. Signature size in a 'typical' system might range from a few hundred bytes to a few kilobytes, and generation of a signature might require a few hundred to a few thousand computations of the underlying conventional encryption function.

INTRODUCTION

While digital signature systems have been proposed[1,3,5,10] that rely only on conventional encryption functions (or on one-way functions) none has quite succeeded in providing the convenience of systems based on more complex mathematical problems such as factoring[2,4]. A significant reason for interest in systems whose security is based only on one-way functions is that the existence of such functions seems assured, while the complexity of factoring and the most efficient factoring algorithm are still open questions of great interest. This is not an issue of purely academic interest, especially in light of the large number of 'unbreakable' cryptographic systems that were subsequently broken.

A second advantage is the reduced computational cost as compared with systems that require modular arithmetic: a software implementation of DES (the Data Encryption Standard) runs much faster than exponentiation modulo N, so a digital signature system based on use of DES would likewise benefit. This savings becomes more significant in a hardware implementation because DES chips are already available at low cost from many manufacturers, and are already present in many existing systems. The new digital signature system is very fast indeed when retro-fitted to a system

C. Pomerance (Ed.): Advances in Cryptology - CRYPTO '87, LNCS 293, pp. 369-378, 1988.

that already has a DES chip (or a hardware implementation of any conventional encryption function).

To make this article self-contained we first briefly review some previously known results on one-way functions and one-time signatures, and then show how a one-time signature system can be used in a new way to provide a digital signature system that overcomes the limitations and drawbacks that have hampered the acceptance and use of this approach.

ONE WAY FUNCTIONS

A one-way function F is a function that is easy to compute, but difficult to invert. Given x, computing y=F(x) is easy. Given y, determining any x such that F(x)=y is hard.

One way functions can be based on conventional encryption functions by observing that deducing the key given the plain text and cipher text is very hard. If we define a conventional encryption function as: $S_{key}(plaintext) = ciphertext$, then we can define a one way function F(x)=y by computing $S_x(0)=y$. That is, we encrypt a constant using x as the key. The resulting ciphertext is the output of the one way function. Deducing x given that we know y is now equivalent to determining the key given that we know the plaintext is 0 and the ciphertext is y.

One way hash functions -- e.g., a one way function which accepts an arbitrarily large input (say, a few kilobytes) and produces a small fixed size output (say, 64 bits) -- can be based on repeated applications of a conventional encryption function in a similar way. The design of one way hash functions should be approached with caution: the most obvious approaches are sometimes vulnerable to 'square root' attacks. For example, if we wish to reduce 112 bits to 64 bits using DES, the obvious technique is to break the 112 bits into two 56-bit blocks and then double encrypt a fixed constant. That is, if the two 56-bit blocks are designated K1 and K2, then compute: $S_{K2}(S_{K1}(0))$. Unfortunately, it is possible to determine a new K1' and K2' in about 2^{28} operations that will produce the same result using a 'meet in the middle' or 'square root' attack. While such attacks can be avoided, it is important to know that they exist and must be guarded against.

We assume that a secure one way hash function is available, possibly based on some conventional encryption function. We shall denote this function F.

SIGNING A ONE-BIT MESSAGE

This section and the next section provide a brief introduction to one-time signatures for those readers unfamiliar with them. These two sections can be skipped without loss of continuity.

Person A can sign a one-bit message for B by using the following protocol: first, in a pre-computation A uses F to one-way encrypt two values of x: x[1] and x[2] -- producing two values of y: y[1] and y[2]. Second, A makes y[1] and y[2] public while keeping x[1] and x[2] secret. Finally, if the one-bit message is a '1', A signs it by giving x[1] to B. If the one-bit message is a '0', A signs it by giving x[2] to B.

If the one-bit message was '1', B can prove that A signed it by presenting x[1] and showing that F(x[1])=y[1]. Because F and the y's are public, anyone can verify the results of this computation. Because only A could know x[1] and x[2], B's knowledge of x[1] implies that A gave x[1] to B -- an act which, by prior agreement, means that A signed the message '1'.

SIGNING A SEVERAL BIT MESSAGE

If A generated many x's and many y's, then A could sign a message with many bits in it. This is the Lamport-Diffie one-time signature[1]. To sign an n-bit message requires 2n x's and 2n y's.

Merkle[3] proposed an improvement to this method which reduces the signature size by almost two-fold. Instead of generating two x's and two y's for each bit of the message, A generates only one x and one y for each bit of the message to be signed. When one of the bits in the message to be signed is a '1', A releases the corresponding value of x -- but when the bit to be signed is a '0', A releases nothing. Because this allows B to pretend that he did not receive some of the x's, and therefore to pretend that some of the '1' bits in the signed message were '0', A must also sign a count of the '0' bits in the message. Now, when B pretends that a '1' bit was actually a '0' bit, B must also increase the value of the count field -- which can't be done. Because the count field has only $\log_2 n$ bits in it, the signature size is decreased by almost a factor of two -- from 2n to $n+\log_2 n$.

As an example, if we wished to sign the 8-bit message '0100 1110' we would first count the number of '0' bits (there are 4) and then append a 3-bit count field (with the value 4) to the original

8-bit message producing the 11-bit message '0100 1110 100' which we would sign by releasing x[2], x[5], x[6], x[7] and x[9]. B cannot pretend that he did not receive x[2], because the resulting erroneous message -- '0000 1110 100' would have 5 0's in it, not 4. Similarly, pretending he did not recieve x[9] would produce the erroneous message '0100 1110 000' in which the count field indicates that there should be no 0's at all. There is no combination of x's that B could pretend not to have received that would let B concoct a legitimate message.

Winternitz[6] proposed an improvement which reduces the signature size by several fold. Instead of being able to sign a one-bit message by pre-computing y[1]=F(x[1]) and y[2]=F(x[2]), A would be able to sign a 2-bit message by pre-computing y[1]=F(F(F(F(x[1])))) and y[2]=F(F(F(F(x[2])))). Notationally, we will show repeated applications of the function F with a superscript -- $F^3(x)$ is F(F(F(x))), $F^2(x)$ is F(F(x)), $F^1(x)$ is F(x), and $F^0(x)$ is x. If A wishes to sign message m -- which must be one of the messages '0', '1', '2', or '3' -- then A reveals $F^m(x[1])$ and $F^{3-m}(x[2])$. B can easily verify the power of F that A used by counting how many more applications of F must be used to reach y. Computing complimentary powers of both x[1] and x[2] is necessary because B might pretend to have received a higher power than A actually sent him. That is, if A sent $F^2(x[1])$ to B, B could compute $F^3(x[1])$ and pretend that A had sent THIS value instead. However, if B does this then B must compute $F^0(x[2])$ as well -- which A would have computed and sent to B if A had actually signed the message '3'. Because A actually sent $F^1(x[2])$, this means B must compute x[2] given only F(x[2]) -- which he can't do. Sending the complimentary powers of x[1] and x[2] in this technique is directly analogous to releasing either x[1] or x[2] in the Lamport-Diffie method.

Though this example shows how to sign one of four messages, the system can be generalized to sign one of n messages by pre-computing $y[1]=F^{n-1}(x[1])$ and $y[2]=F^{n-1}(x[2])$.

The almost two-fold improvement proposed by Merkle for the 1-bit one-time signature generalizes to the Winternitz one-time signature.

Thus, the original one-time signature system proposed by Lamport and Diffie, and improved by Winternitz and Merkle, can be used to sign arbitrary messages and has excellent security. The storage and computational requirements for signing a single message are quite reasonable. Unfortunately, signing more messages requires many more x's and y's and therefore a very large entry in the public file (which holds the y's). To allow A to sign 1000 messages might require roughly 10,000 y's -- and if there were 1000 different users of the system, each of whom wanted to sign 1000 messages, this would increase the storage requirement for the public file to hundreds of megabytes -- which is unwieldy and has effectively prevented use of these systems.

AN INFINITE TREE OF ONE-TIME SIGNATURES

The general idea in the new system is to use an infinite tree of one-time signatures. For simplicity, we assume that the tree is binary. The root of the infinite tree is authenticated simply by placing it in the public file. Each node of the tree performs three functions: (1) it authenticates the left sub-node (2) it authenticates the right sub-node and (3) it signs a single message. Because there are an infinite number of nodes in the tree, an infinite number of messages can be signed. To perform these three functions, each node must have three signatures -- a 'left' signature, a 'right' signature, and a 'message' signature. The 'left' signature is used to 'sign off' on the left sub-node, the 'right' signature is used to 'sign off' on the right sub-node, while the 'message' signature is available to sign a user message.

Notationally, it is convenient to number the nodes in the tree in the following fashion:

The root node is designated '1'.
The left sub-node of node i is designated 2i.
The right sub-node of node i is designated 2i+1.

This assignment of numbers has many convenient properties. It uniquely numbers every node in the infinite tree; the left and right sub-nodes are easily computed from a parent node; and the parent node can be computed from the sub-node by a simple integer division by 2. Note that if we start from node 1 and follow the left sub-node at each node, the node numbers are: 1,2,4,8,16,32,64, ...

We adopt some further notational conventions to distinguish between the x's and y's used to sign different messages at different nodes in the tree -- in particular, we shall use a three-dimensional array of x's and y's: array x[<node number>, <left,right, or message>, <index within the one-time signature>]. If we use the original Lamport-Diffie method (involving 128 x's per signature) then all the x's at node i would be:

x[i,left,1], x[i,left,2] ... x[i,left,128]
x[i,right,1], x[i,right,2] ... x[i,right,128]
x[i,message,1], x[i,message,2] ... x[i,message,128]

We will designate all the x's for the 'left' signature at node i by x[i,left,*]. Similarly, we shall designate all the y's associated with the message signature at node i by y[i,right,*]. We shall designate all the x's at node i (left, right, and message) by x[i,*,*].

We shall often wish to apply a one way hash function to all the y's for a given signature, so we define the notation F(y[i,right,*]) to mean use of the one way hash function F applied to all the y's for the right signature of node i.

Thus, our fundamental data structures will be two infinite three-dimensional arrays x and y, where each y is computed from the corresponding x by applying F.

We shall often wish to compute a 'hash total' for all the y's at a given node. We do this by first applying F to each signature individually, and then applying F to the three resultant values. We define the function HASH(i) as:

HASH(i) = F(F(y[i,left,*]), F(y[i,right,*]), F(y[i,message,*]))

This is the one way hash total for node i. It has the important property that if we already know HASH(i) and someone sends us what they claim are the y[i,*,*] values we can confirm that they sent us the correct values (or show that they sent the wrong values) by re-computing the function. If the value of HASH(i) computed from the values sent to us matches the value that we already know, then we know we have received the correct y[i,*,*] values.

Prior to the signature protocol, A enters HASH(1) into the public file. This value authenticates the root node of A's authentication tree, and it is assumed that it is publicly known to everyone.

We can now describe the algorithm that A uses to sign message m with signature i, and that B uses to check the signature.

THE NEW SIGNATURE ALGORITHM

Both A and B agree in advance on the message M to be signed. A selects the node i that will be used to sign it.

1. A sends i and y[i,message,*] to B.
2. A signs message M by sending B the appropriate subset of x's in x[i,message,*].
3. B checks that the released subset of the x[i,message,*] correctly sign message M when checked against the y[i,message,*].

4. A sends F(y[i,left,*]), F(y[i,right,*]) and F(y[i,message,*]) to B.
5. A computes HASH(i). By definition, this is:
   F(F(y[i,left,*]), F(y[i,right,*]), F(y[i,message,*]))
6. If the value of i is 1, then B checks that the value of HASH(1)
   computed from the values A transmitted matches the entry HASH(1)
   in the public file, and the algorithm terminates.
7. If i is even,
   A sends y[i/2,left,*] to B.
   A signs HASH(i) by sending the correct subset of x[i/2,left,*].
   B computes HASH(i) and verifies that it was properly signed
   by checking the x's against the y[i/2,left,*].
8. If i is odd,
   A sends y[i/2,right,*] to B.
   A signs HASH(i) by sending the correct subset of x[i/2,right,*].
   B computes HASH(i) and verifies that it was properly signed
   by checking the x's against the y's in y[i/2,right,*].
9. Both A and B replace i by i/2 and proceed to step 4.

When the algorithm terminates, B has $\log_2$ i one-time signatures, one of which is the one-time signature for message M that B actually wanted, while each of the others verifies the correctness and validity of the next signature -- and the validity of the 'root' signature is attested to by the entry in the public file. Thus, this 'audit trail' of one-time signatures starts with HASH(1), proceeds to HASH(i), and finally terminates with the one-time signature for message M.

It should be clear that this 'meta-system' can utilize any one-time signature system, and that improvements in the one-time signature system will produce corresponding improvements in the meta-systems performance. There is no particular reason to believe that current one-time signature systems have reached a plateau of perfection, and so further research into one-time signature systems might well produce worthwhile performance improvements.

It should also be clear that the use of a binary tree is arbitrary -- it could just as easily be a K-ary tree, and probably will be in practice. A binary tree requires $\log_2$ i one-time signatures, while a K-ary tree requires only $\log_K$ i one-time signatures -- which generally results in a smaller over-all signature size for larger values of K. However, in a K-ary tree the computation of HASH(i) becomes:

```
F(
 F(y[i,first-sub-node,*]),
```

```
F(y[i,second-sub-node,*]),
F(y[i,third-sub-node,*]),
.
.
.
F(y[i,Kth-sub-node,*])
F(y[i,message,*])
)
```

The computation at each node takes longer because all the y's for all the K sub-nodes must be re-computed, and each node in the resulting signature requires more memory. Thus, the optimal value of K can't be too large -- or it will run afoul of these limitations.

The problem of minimizing the additional authentication information required within each node as the value of K increases is actually interesting in its own right. As described above, the information required as part of the signature at each node will increase linearly with K. This has been reduced to $\log_2 K$ in the author's previous 'tree-signature' method[3,7 page 170]. Combining the two systems into a single hybrid seems quite appropriate and would allow rather large values of K to be used efficiently. The author's previous 'tree-signature' method is quite different in concept from the current method, though both use a tree. It is interesting to note that Goldwasser, Micali, and Rivest[8,9] also use a tree-like structure to provide desirable theoretical properties in a digital signature. Their signature system is '...based on the existence of a "claw-free" pair of [trapdoor] permutations' which they build by multiplying together two large primes (as in the RSA system).

Finally, some readers might object that the infinite three dimensional arrays x and y might be awkward for user A to store -- and so a compaction scheme seems appropriate. The array of y's is computed from the array of x's, and so the y's need not actually be stored. The array of x's is randomly chosen by A in any fashion that A desires. A might just as well generate the x's in a secure pseudo-random fashion. In particular, A can compute x[i,j,k] by concatenating i, j, and k and then encrypting this bit pattern with a conventional encryption function using a secret key: $x[i,j,k] = S_{\text{A's secret key}}(<i,j,k>)$. If we were to use DES, A's secret key need only be 56 bits. Even after many of the x's had been made public (in the course of signing various messages) it would be impossible to determine A's secret key. The pairs (<i,j,k>, x[i,j,k]) are plaintext-ciphertext pairs and by definition the key of a conventional encryption function cannot be determined even if many such pairs are known.

In practice, all A need remember is a single secret key (of perhaps 56 bits) and a simple integer count (of perhaps 20 or 30 bits) to keep track of which node in the tree was used to sign the last

message. If the computations are ordered correctly, generating a signature can be done in a very small memory (128 bytes of RAM is more than enough). Such small memories (and even smaller) often occur in low-cost high-volume applications, such as 'smart-cards' (credit cards with a built-in computer).

CONCLUSION

A digital signature system has been presented which is based solely on a conventional encryption function. The algorithms to sign and check signatures are rapid and require only a very small amount of memory. The size of the signatures grows as the logarithm of the number of messages signed. Signature size and memory requirements can be traded off against computational requirements.

REFERENCES

1. 'New Directions in Cryptography', IEEE Trans. on Information Theory, IT-22, 6(Nov. 1976),644-654

2. 'A method for obtaining digital signatures and public-key cryptosystems.' CACM 21,2, Feb. 1978 120-126

3. 'Secrecy, Authentication, and Public Key Systems', Ralph C. Merkle, UMI Research Press 1982.

4. 'How to Prove Yourself: Practical Solutions to Identification and Signature Problems', Amos Fiat and Adi Shamir, 1986.

5. 'Making the Digital Signature Legal -- and Safeguarded', S.M. Lipton, S.M. Matyas, Data Communications, Feb. 1978 41-52.

6. Private Communication, Robert Winternitz, 1980.

7. 'Cryptography and Data Security', by Dorothy E.R. Denning, Addison Wesley 1982.

8. 'A "Paradoxical" solution to the Signature Problem', by Shafi Goldwasser, Silvio Micali and Ronald L. Rivest, from the Symposium on the Foundations of Computer Science, 1984, page 441-448.

9. 'A Digital Signature Scheme Secure Against Adaptive Chosen Message Attack', by Shafi Goldwasser, Silvio Micali and Ronald L. Rivest, extended abstract, 1986.

10. 'Cryptography: a New Dimension in Computer Data Security', by Carl H. Meyer and Stephen M. Matyas, Wiley 1982.