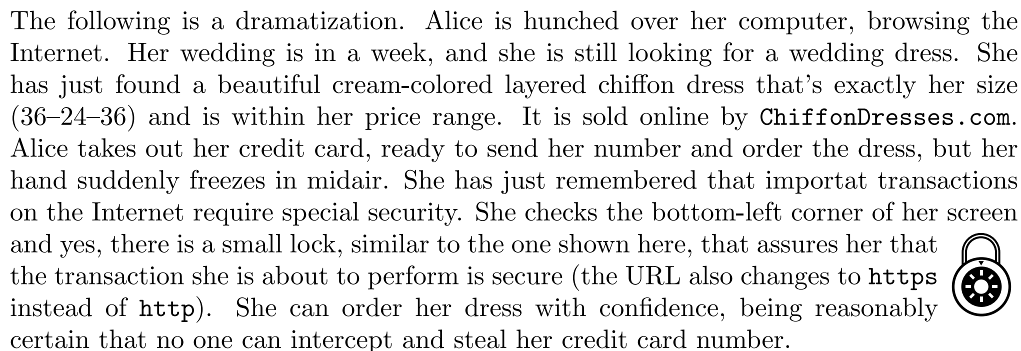
4.1 SSL: Secure Socket Layer

The following is a dramatization. Alice is hunched over her computer, browsing the Internet. Her wedding is in a week, and she is still looking for a wedding dress. She has just found a beautiful cream-colored layered chiffon dress that's exactly her size (36–24–36) and is within her price range. It is sold online by `ChiffonDresses.com`. Alice takes out her credit card, ready to send her number and order the dress, but her hand suddenly freezes in midair. She has just remembered that importat transactions on the Internet require special security. She checks the bottom-left corner of her screen and yes, there is a small lock, similar to the one shown here, that assures her that the transaction she is about to perform is secure (the URL also changes to `https` instead of `http`). She can order her dress with confidence, being reasonably certain that no one can intercept and steal her credit card number.



This scenario is common. Most of us perform sensitive transactions over the Internet, and we expect them to be private. Online purchasing is one example. Online banking, where a bank account can be reviewed by a customer after a PIN is sent, is another.

This section describes the SSL (secure socket layer) protocol employed by all major Web browsers, as well as by other software, to secure messages sent over the Internet. First, a disclaimer. SSL provides secure communications, but cannot guarantee total security. A credit card number or other sensitive information sent over the Internet by the SSL protocol is encrypted and cannot be compromised while in transit. When it arrives at its destination, however, the security provided by SSL ceases and the information may become vulnerable. A dishonest employee may steal it. An insecure data base may be taken over by a hacker and its content copied and misused. The conclusion is simple. Don't trust SSL all the way. Trust it only for communicating your sensitive data. If there is any reason to doubt the integrity of the receiver, don't send the data. The better business bureau [BBB 03] is one source that can be employed to evaluate the integrity of a commercial organization.

SSL was developed at Netscape Communications, Inc. in 1994 in response to users' demand for secure Internet communications. It has since evolved and strengthened considerably by several organizations. Today, the SSL protocol that's mostly used is the transport layer security (TLS) and there are other versions of SSL, such as an open version (openssl) and a version for wireless communications (WTLS).

Two recommended references are [Rescorla 00] and [Thomas 00].

As usual, we assume two protagonists, Alice and Bob. Alice plays the part of a consumer trying to purchase an item online. Bob is the seller. The SSL protocol proceeds in the following steps:

1. An authentication protocol is executed by Alice to make sure that Bob is really who he claims to be. Bob's public key is sent to Alice as part of the protocol. This protocol is based on the public-key concept and employs RSA encryption and also a trusted third party.

2. Alice selects a random key for encrypting her sensitive information. This key is encrypted with Bob's public key and is sent to Bob.

3. Alice uses this key to encrypt her sensitive data with DES or another strong

encryption algorithm. Bob uses the same key and algorithm to decrypt the data. Several messages can be exchanged this way between the two parties in complete privacy.

It is obvious that step 1 is the most important part of SSL. It provides secure communications over an insecure channel. This step is complex and slow, which is why it is used only for communicating a short (normally 128-bit) key. The sensitive data itself is encrypted with a fast cipher. This step depends on a basic property of the RSA encryption algorithm. Data encrypted with a public key can be decrypted only with the corresponding private key, but data can also be encrypted with the private key and decrypted only with the corresponding public key. With this in mind, we start with a simple authentication protocol. (We use the notation "`<message> key`" to indicate a message encrypted with a certain key.) If alice wants to authenticate Bob, she can him send a short message and have Bob encrypt it with his private key and return the result.

Alice → Bob: `Authenticate this.`
Bob → Alice: `<Authenticate this> Bob's private key.`

Alice now decrypts this with Bob's public key. If the result matches her original message, she has authenticated Bob. This simple protocol has two drawbacks as follows:

1. Alice must know Bob's public key. If Alice and Bob are members of a group, say, both are scientists and have been communicating by email for a while, then their public keys are known to all the group's members because they are included in each email message. However, if Bob is an organization, such as a new online store, Alice may not have its public key. Even if Bob sends his public key to Alice, she cannot be sure that it really came from Bob's store; it could have been sent by Eve pretending to be Bob and trying to steal Alice's card number.

2. Encrypting a message with your private key and sending it to Alice leads to weak security. Remember that Alice has the original message. If she also has its private-key encryption, she may use both to pretend to be Bob.

Our simple protocol needs improvements. The first one eliminates the need to encrypt Alice's message with Bob's private key. Instead, Bob selects a new message, computes its *message digest*, encrypts the digest with his private key, and sends the (plain) message and the encrypted digest to Alice.

Alice → Bob: `Looking for Bob.`
Bob → Alice: `I'm Bob, <Digest[I'm Bob]> Bob's private key.`

The digest of a message is a function of the message with the following useful properties: (1) It is practically infeasible to compute the original message from its digest and (2) the chance of finding another message that will produce the same digest is extremely small. In practice, a digest is a hash function that hashes text of any length to a small (typically 128-bit) number. The SHA-1 MD5 hash function is currently popular as a digest generator. It has replaced the (somewhat similar) MD5 function, which is described below.

With this protocol, Bob still has to send a message (`I'm Bob`) and the encrypted version of its digest (this is known as a digital signature), but now he can select the

message, which gives him more protection from an unscrupulous Alice. The protocol constitutes authentication because Alice has a plain message and the private-key encryption of its digest. She can decrypt the digest, digest the message, and compare the two digests. There is still the problem of having Bob's public key and being certain that it is Bob's, and no one else's public key. Here is what may happen if Eve pretends to be Bob.

Alice $\rightarrow$ Bob: `I'm looking for Bob.`
Eve $\rightarrow$ Alice: `I'm Bob, Eve's-public-key.`
Alice $\rightarrow$ Bob: `Are you?`
Eve $\rightarrow$ Alice: `Of course I am, <Digest[Of course I am]> Eve's private key.`

The solution to this dilemma involves a third, trusted party, an escrow, that issues *certificates*. When Bob opens his store, he applies for a certificate from an escrow. The escrow company sends an inspector to check Bob and his facilities, and look at their operations and identification. If all is satisfactory, a certificate is issued, but it has an expiration date and has to be renewed periodically. Admittedly, this solution is not elegant. In principle, we would like a protocol that involves just the two communicating parties, but in practice a third party is needed. A certificate contains the following fields (Figure 4.1):

1. The name of the certificate issuer (the escrow).
2. A digital signature of the certificate issuer.
3. The name of the subject, Bob (the entity for which the certificate is issued).
4. The subject's public key.
5. The certificate's expiration date.

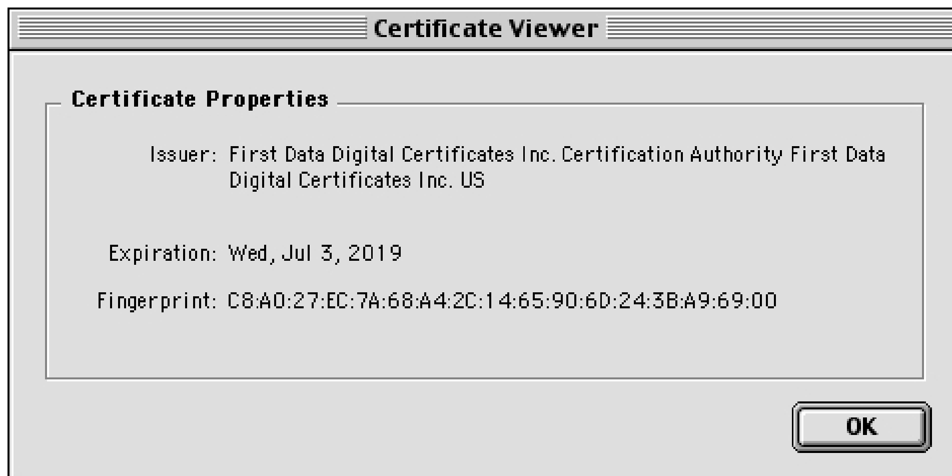


Figure 4.1: A Certificate

Figure 4.3 is a detailed listing of the fields of a typical certificate.

Many organizations apply for certificates, so there must be many certificate issuers. Alice doesn't know all of them. She can be expected to know only a few. There is therefore a need for root certificate issuers. Every leading Web browser comes with a list of root certificates preinstalled. A root certificate (also known as a CA or certificate authority) belongs to a trusted authority that can issue certificates to other, smaller certificate issuers after checking each to make sure it can be trusted. The encryption preferences (or security preferences) menu of the browser can display the list of CA certificates it knows. If the certificate has expired, the browser displays a dialog box similar to Figure 4.2 that gives the user a choice to continue the sensitive web session or terminate it.

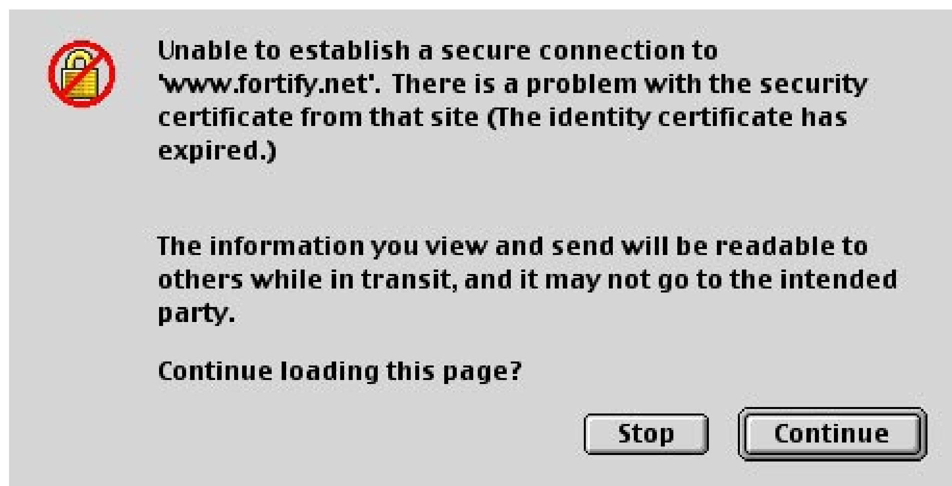


Figure 4.2: An Expired Certificate

With certificates, the SSL protocol proceeds as follows:

Alice → Bob: I'm looking for Bob.
Bob → Alice: I'm Bob, Bob's certificate.
Alice → Bob: Are you really?
Bob → Alice: Definitely, <Digest[Definitely]> Bob's private key.

It is the certificate that provides Alice (or, in practice, her Web browser) with Bob's public key. To verify that this is really Bob's certificate, Alice's browser reads the certificate's issuer name (say, Y) and signature from the certificate. The digital signature contains the issuer's own certificate, which has been issued by one of the root issuers (call it X). The browser has a list of the root certificate issuers and it communicates with root issuer X to verify the certificate of issuer Y. This is a slow, tedious process, so it is used as little as possible.

Eve may try to impersonate Bob in this protocol, so we have to keep her in mind. She can execute step 2 in the protocol, because she may have Bob's certificate from

a past transaction with him, but she cannot execute step 4 because she doesn't have Bob's private key.

The protocol above is the first and most important step in the complete, three-step SSL protocol. Once it has been executed, Alice is confident that she is dealing with Bob and that she has his public key. In the second step, Alice (or rather her browser) selects a random number to serve as a secret key and sends it to Bob, encrypted with his public key, as a short message. Only Bob can decrypt this message, but again we have to place ourselves in Eve's shoes. What can she do? She cannot decrypt this short message, but she can damage it on its way to Bob. This may be useful to Eve, so the SSL protocol must have a way for Bob to identify damaged messages. One way of verifying a message is to append to it a message authentication code (MAC) that consists of a digest of the message and of the secret key. Eve doesn't know the secret key, so she cannot generate the MAC. Here is the revised protocol

Alice → Bob: Is this Bob?
Bob → Alice: I'm Bob, Bob's certificate.
Alice → Bob: Are you really?
Bob → Alice: Definitely, <Digest[Definitely]> Bob's private key.
Alice → Bob: You have been authenticated.
Alice → Bob: Here is our new key <secret-key> Bob's public key.
Alice computes: MAC=Digest[My CC # is 12345, secret-key].
Alice → Bob: <My CC # is 12345, MAC> secret-key.

Bob knows to expect two-part messages, where the second part consists of the MAC of the first part. Any corrupted message can easily be identified by Bob. Once Bob has received the new secret key (normally, 40, 56, or 128 bits) from Alice, the two can exchange messages with confidence. The messages are encrypted with this key by a secure, fast algorithm, such as DES or RC4.

As noted earlier, SSL was developed at Netscape Communications in 1994. Just a year later, several hackers discovered a weakness in the first SSL version. It turned out that the secret key was selected by a pseudo-random number generator (PRNG) whose seed was a combination of the current time (just the seconds and microseconds) and the process id. Netscape programmers believed that such a combination was sufficiently random and would lead to pseudo-random numbers that could be used as secure keys. However, someone intercepting information packets sent by a browser can have a good idea of the time (in seconds) when the packets were generated. Also, someone with access to any account on the operating system where the Netscape browser is running can find the id of any process. The microseconds part of the seed can then be found by trying the million values between 0 and 999,999. It seems that Netscape has since improved the way the seed of the PRNG is computed.

Example of a Certificate.

Figure 4.3 is a detailed listing of the fields of a typical certificate. The issuer part and the subject part have the fields C (two-letter international country code), ST (state or province), L (locality), O (organization name), and OU (organizational unit).

```
Certificate:
Data:
Version: 1 (0x0)
Serial Number: 0 (0x0)
Signature Algorithm: md5WithRSAEncryption
Issuer:
  C=US,
  ST=NC,
  L=Cary,
  O=My New Outfit, Inc.,
  OU=Sales,
  CN=ntbox.somewhere.com/Email=me@somewhere.com

Validity
  Not Before: Oct 7 04:19:24 1999 GMT
  Not After : Oct 6 04:19:24 2000 GMT

Subject:
  C=US,
  ST=NC,
  L=Cary,
  O=My New Outfit, Inc.,
  OU=Sales,
  CN=ntbox.somewhere.com/Email=me@somewhere.com

Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (1024 bit)
Modulus (1024 bit):

00:c9:dd:68:31:ca:1c:ab:74:7c:21:a8:de:71:22:
25:ec:48:dd:54:34:b5:b8:be:ad:96:cf:56:ad:a2:
7d:9f:81:d5:62:3a:f1:c2:03:4d:8d:73:a3:cb:ac:
f8:f4:d7:95:0d:3f:9e:2c:8f:5f:d3:40:91:09:79:
21:c4:8b:f6:0a:3b:2c:c7:42:3d:2c:c3:5b:17:68:
58:2e:47:42:1e:24:41:1d:59:ba:57:0c:26:63:2e:
46:55:72:e5:1e:61:6c:6e:c2:73:ad:e0:68:ed:70:
a9:43:73:69:b5:c3:9f:64:54:d6:12:11:f3:10:38:
42:e8:54:82:23:f7:20:26:03

Exponent: 65537 (0x10001)

Signature Algorithm: md5WithRSAEncryption

4f:27:7b:c5:f1:52:33:bc:f8:50:19:b9:98:e6:3b:08:9b:4b:
7b:24:f8:80:10:18:a4:25:6a:39:b1:75:35:05:64:54:ec:5e:
e4:c1:88:fb:7f:72:d1:32:f4:8c:0d:08:28:7e:7e:a5:5f:61:
9c:cc:b4:5c:13:f0:71:a8:d0:56:58:11:e6:b8:35:0a:01:b7:
72:7f:e8:a7:b6:82:aa:52:5d:05:29:d8:48:ba:26:8e:ed:41:
38:86:b8:62:2e:9a:f1:be:99:3c:20:76:57:0f:70:4b:a6:18:
82:aa:90:0c:1f:18:05:c3:98:b8:20:9e:e5:64:02:0d:01:4e:
c4:4e
```

Figure 4.3: A Detailed Certificate

References

BBB (2003) is URL `www.bbbonline.com`.

Rescorla, Eric (2000) *SSL and TLS: Designing and Building Secure Systems*, Reading, MA, Addison Wesley.

Thomas, Steven A. (2000) *SSL and TLS Essentials: Securing the Web*, New York, John Wiley.

4.2 MD5 Hashing

MD5 (short for "message digest 5") is a hash function developed by Ronald Rivest in 1992 ([Rivest 92] and [rfc1321 03]) to serve as a fast and secure message digest for digital signature applications. MD5 inputs a message of any length and hashes it to a 128-bit number that can serve as a fingerprint of the message. MD5 was designed to be especially fast on 32-bit machines. MD5 is an extension of the similar MD4 algorithm [Rivest 91]. It is somewhat slower than MD4 but is deemed more secure. Based on the experience of the algorithm's developer it is conjectured (although not proved) that it is computationally infeasible to generate a message whose MD5 digest will equal a given 128-bit number or to find two nontrivial messages that have the same MD5 digest. Specifically, it is conjectured that the difficulty of coming up with two messages that will have the same MD5 output is on the order of 2^{64} operations, and the difficulty of finding a message that has a given MD5 digest is on the order of 2^{128} operations.

Computations in MD5 are based on 32-bit words. Thus, the symbol "+" indicates modulo 2^{32} addition of 32-bit words, and other symbols also correspond to operations on 32 bits.

We start with a b-bit message where b is an arbitrary nonnegative integer. It can be zero and doesn't have to be a multiple of 8 or of any other number. The bits are denoted by m_0 through m_{b-1}. Hashing the message is done by scrambling its bits in five steps as follows:

Step 1. Append padding bits. The message is extended by appending bits until its length becomes 64 bits less than the next multiple of 512. We say that the extended message size is congruent to 448 modulo 512. Padding is always done, even if the original length b of the message satisfies the above condition. This implies that at least one bit and at most 512 bits are appended. The first bit appended is a 1 and the remaining bits are zeros.

Step 2. Append length. The value of b (the original length of the message) is now appended to the extended message as a 64-bit number. If b is greater than $2^{64}-1$, only the 64 least-significant bits of b are appended (the message length cannot be deduced from its 64 least-significant bits, but there is no need to deduce it). The message length at this point is a multiple of 512, so it is also a multiple of 16×32. We denote the number of 32-bit words in the message by N (N is a multiple of 16) and the words themselves by $M(0)$ through $M(N-1)$.

Step 3. Initialize buffer. A 4-word buffer denoted by `A`, `B`, `C`, and `D` is allocated. They are initialized to the following values

$$A \leftarrow 01234567_{16}, \quad B \leftarrow 89abcdef_{16}, \quad C \leftarrow fedcba98_{16}, \quad D \leftarrow 76543210_{16}.$$

The final 128-bit result will be computed in these four 32-bit words.

Step 4. Process message. The hashing of the message involves four functions denoted by `F`, `G`, `H`, and `I`, a 16-word array `X`, a 64-word table `T`, four words denoted by `AA`, `BB`, `CC`, and `DD`, and shifts and logical operations. Each of the four functions receives three 32-bit words as input parameters and computes one word. The definitions of the four functions are

$$\mathtt{F}(X,Y,Z) = (X \times Y) \vee (\text{not}(X) \times Z), \quad \mathtt{G}(X,Y,Z) = (X \times Z) \vee (Y \times \text{not}(Z)),$$
$$\mathtt{H}(X,Y,Z) = X \text{ xor } Y \text{ xor } Z, \quad \mathtt{I}(X,Y,Z) = Y \text{ xor } (X \vee \text{ not}(Z)),$$

where $\times$, $\vee$, and, xor indicate logical AND, OR and XOR, respectively.

A table T of 64 constants is first computed, such that element `T[i]` (for `i` values 1 through 64) becomes the integer part of the product of `abs(sin(i))` (where `i` is in radians) and the constant 4294967296. This table serves to further scramble the bits of the message.

The computations consist of steps where each step processes a block of 16 nonconsecutive words from the message. Each step consists of four rounds. Each round starts by initializing array `X` to a block of 16 different words from the message, then scrambles the values of `A`, `B`, `C`, and `D` by means of `T`, `X`, and one of the four functions. Here are the details (the notation `X <<< s` denotes an `s` position, left-circular shift of `X`).

```
for i=0 to N/16-1 do {loop on blocks}
 for j=0 to 15 do
  X[j]:=M[i*16+j]
 endfor {j}
 AA:=A; BB:=B; CC:=C; DD:=D;
{Round 1. We denote by [abcd k s i] the operation
  a=b+((a+F(b,c,d)+X[k]+T[i])<<<s).
  Round 1 consists of the following 16 operations}
[ABCD  0  7  1]  [DABC  1 12  2] [CDAB  2 17  3] [BCDA  3 22  4]
[ABCD  4  7  5]  [DABC  5 12 16] [CDAB  6 17  7] [BCDA  7 22  8]
[ABCD  8  7  9]  [DABC  9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
[ABCD 12  7 13]  [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
{Round 2. We denote by [abcd k s i] the operation
  a=b+((a+G(b,c,d)+X[k]+T[i])<<<s).
  Round 2 consists of the following 16 operations}
[ABCD  1  5 17]  [DABC  6  9 18] [CDAB 11 14 19] [BCDA  0 20 20]
```

```
[ABCD  5  5 21]  [DABC 10  9 22] [CDAB 15 14 23] [BCDA  4 20 24]
[ABCD  9  5 25]  [DABC 14  9 26] [CDAB  3 14 27] [BCDA  8 20 28]
[ABCD 13  5 29]  [DABC  2  9 30] [CDAB  7 14 31] [BCDA 12 20 32]
{Round 3. We denote by [abcd k s i] the operation
  a=b+((a+H(b,c,d)+X[k]+T[i])<<<s).
  Round 3 consists of the following 16 operations}
[ABCD  5  4 33]  [DABC  8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
[ABCD  1  4 37]  [DABC  4 11 38] [CDAB  7 16 39] [BCDA 10 23 40]
[ABCD 13  4 41]  [DABC  0 11 42] [CDAB  3 16 43] [BCDA  6 23 44]
[ABCD  9  4 45]  [DABC 12 11 46] [CDAB 15 16 47] [BCDA  2 23 48]
{Round 4. We denote by [abcd k s i] the operation
  a=b+((a+I(b,c,d)+X[k]+T[i])<<<s).
  Round 4 consists of the following 16 operations}
[ABCD  0  6 49]  [DABC  7 10 50] [CDAB 14 15 51] [BCDA  5 21 52]
[ABCD 12  6 53]  [DABC  3 10 54] [CDAB 10 15 55] [BCDA  1 21 56]
[ABCD  8  6 57]  [DABC 15 10 58] [CDAB  6 15 59] [BCDA 13 21 60]
[ABCD  4  6 61]  [DABC 11 10 62] [CDAB  2 15 63] [BCDA  9 21 64]
{final operations}
A:=A+AA; B:=B+BB; C:=C+CC; D:=D+DD;
endfor {i}
```

Step 5. Output. The final output is the four words `A`, `B`, `C`, and `D`, each read as four bytes from right to left. Thus, the most-significant byte of the 128-bit output is the least-significant byte of `A`, and the output ends with the most-significant byte of `D`.

(End of the five steps.)

The MD5 algorithm is based on the long experience of its creator. It has been used extensively for since its inception about a decade ago and no weaknesses have been discovered.

References

rfc1321 (2003) is URL `http://www.ietf.org/rfc/rfc1321.txt`.

Rivest, R. (1991) "The MD4 message digest algorithm," in A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology: CRYPTO '90 Proceedings*, pages 303–311, New York, Springer-Verlag.

Rivest, R. (1992) "The MD4 Message Digest Algorithm," RFC 1320, MIT and RSA Data Security, Inc., April.