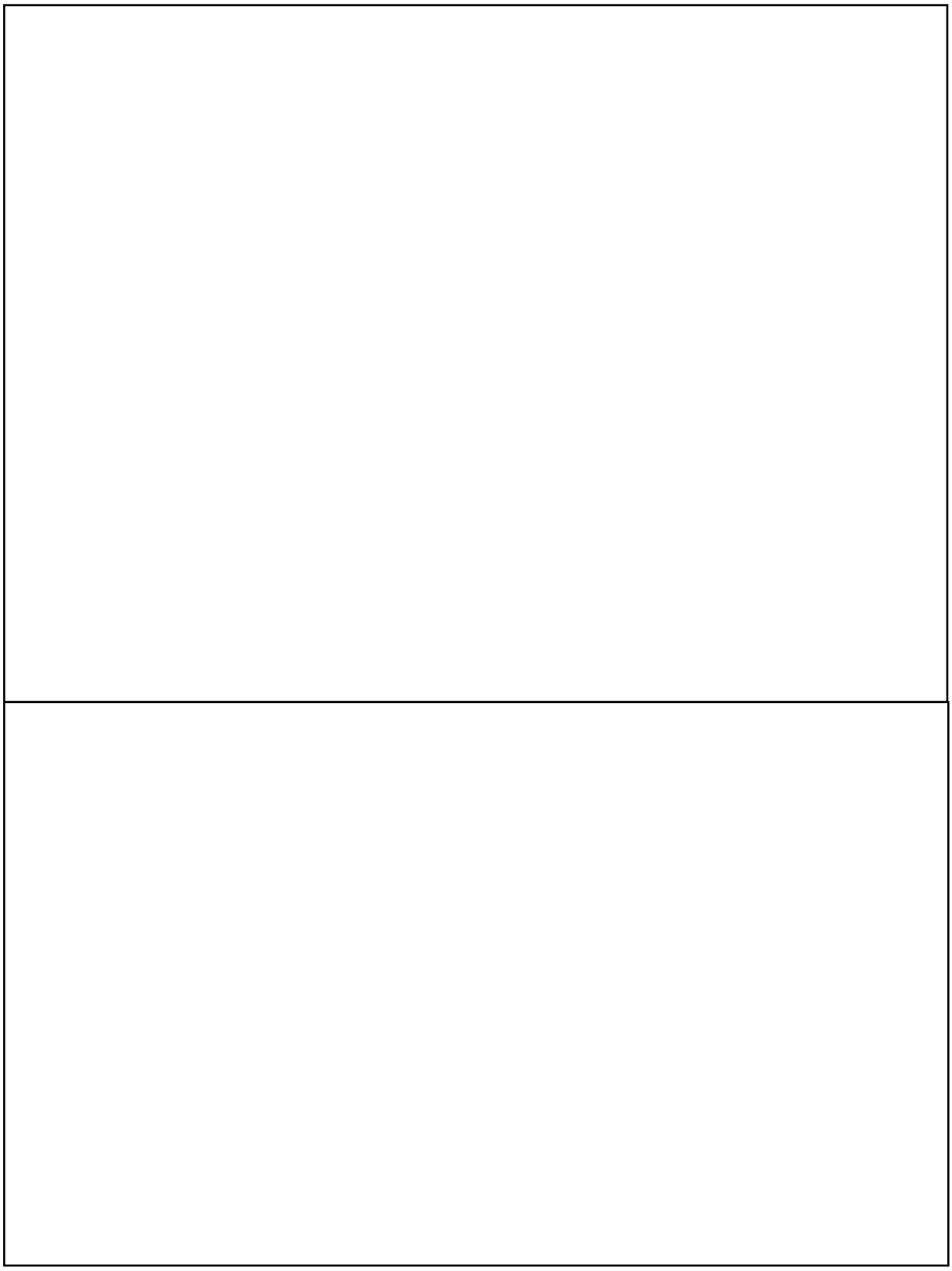

8.4.3 Message Digests

One criticism of signature methods is that they often couple two distinct functions: authentication and secrecy. Often, authentication is needed but secrecy is not always needed. Also, getting an export license is often easier if the system in

question provides only authentication but not secrecy. Below we will describe an authentication scheme that does not require encrypting the entire message.

This scheme is based on the idea of a one-way hash function that takes an arbitrarily long piece of plaintext and from it computes a fixed-length bit string. This hash function, *MD*, often called a **message digest**, has four important properties:

1. Given P, it is easy to compute $MD(P)$.
2. Given $MD(P)$, it is effectively impossible to find P.
3. Given P, no one can find P' such that $MD(P') = MD(P)$.
4. A change to the input of even 1 bit produces a very different output.

To meet criterion 3, the hash should be at least 128 bits long, preferably more. To meet criterion 4, the hash must mangle the bits very thoroughly, not unlike the symmetric-key encryption algorithms we have seen.

Computing a message digest from a piece of plaintext is much faster than encrypting that plaintext with a public-key algorithm, so message digests can be used to speed up digital signature algorithms. To see how this works, consider the signature protocol of Fig. 8-18 again. Instead, of signing P with $K_{BB}(A, t, P)$, *BB* now computes the message digest by applying *MD* to P, yielding $MD(P)$. *BB* then encloses $K_{BB}(A, t, MD(P))$ as the fifth item in the list encrypted with K_B that is sent to Bob, instead of $K_{BB}(A, t, P)$.

If a dispute arises, Bob can produce both P and $K_{BB}(A, t, MD(P))$. After Big Brother has decrypted it for the judge, Bob has $MD(P)$, which is guaranteed to be genuine, and the alleged P. However, since it is effectively impossible for Bob to find any other message that gives this hash, the judge will easily be convinced that Bob is telling the truth. Using message digests in this way saves both encryption time and message transport costs.

Message digests work in public-key cryptosystems, too, as shown in Fig. 8-20. Here, Alice first computes the message digest of her plaintext. She then signs the message digest and sends both the signed digest and the plaintext to Bob. If Trudy replaces P along the way, Bob will see this when he computes $MD(P)$.

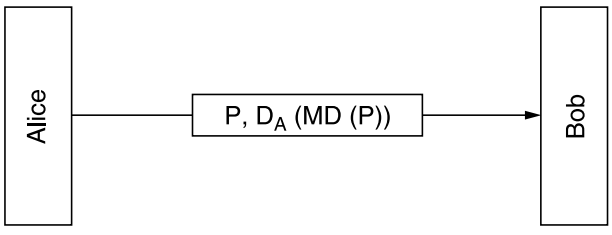

Figure 8-20. Digital signatures using message digests.

SHA-1 and SHA-2

A variety of message digest functions have been proposed. One of the most widely used functions is **SHA-1** (**Secure Hash Algorithm 1**) (NIST, 1993). Like all message digests, it operates by mangling bits in a sufficiently complicated way that every output bit is affected by every input bit. SHA-1 was developed by NSA and blessed by NIST in FIPS 180-1. It processes input data in 512-bit blocks, and it generates a 160-bit message digest. A typical way for Alice to send a nonsecret but signed message to Bob is illustrated in Fig. 8-21. Here, her plaintext message is fed into the SHA-1 algorithm to get a 160-bit SHA-1 hash. Alice then signs the hash with her RSA private key and sends both the plaintext message and the signed hash to Bob.

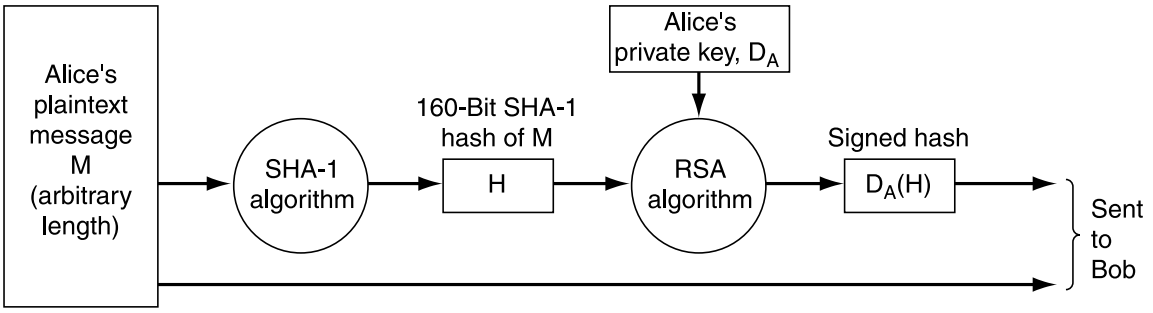

Figure 8-21. Use of SHA-1 and RSA for signing nonsecret messages.

After receiving the message, Bob computes the SHA-1 hash himself and also applies Alice's public key to the signed hash to get the original hash, *H*. If the two agree, the message is considered valid. Since there is no way for Trudy to modify the (plaintext) message while it is in transit and produce a new one that hashes to *H*, Bob can easily detect any changes Trudy has made to the message. For messages whose integrity is important but whose contents are not secret, the scheme of Fig. 8-21 is widely used. For a relatively small cost in computation, it guarantees that any modifications made to the plaintext message in transit can be detected with very high probability.

Now let us briefly see how SHA-1 works. It starts out by padding the message by adding a 1 bit to the end, followed by as many 0 bits as are necessary, but at least 64, to make the length a multiple of 512 bits. Then a 64-bit number containing the message length before padding is ORed into the low-order 64 bits. In Fig. 8-22, the message is shown with padding on the right because English text and figures go from left to right (i.e., the lower right is generally perceived as the end of the figure). With computers, this orientation corresponds to big-endian machines such as the SPARC and the IBM 360 and its successors, but SHA-1 always pads the end of the message, no matter which endian machine is used.

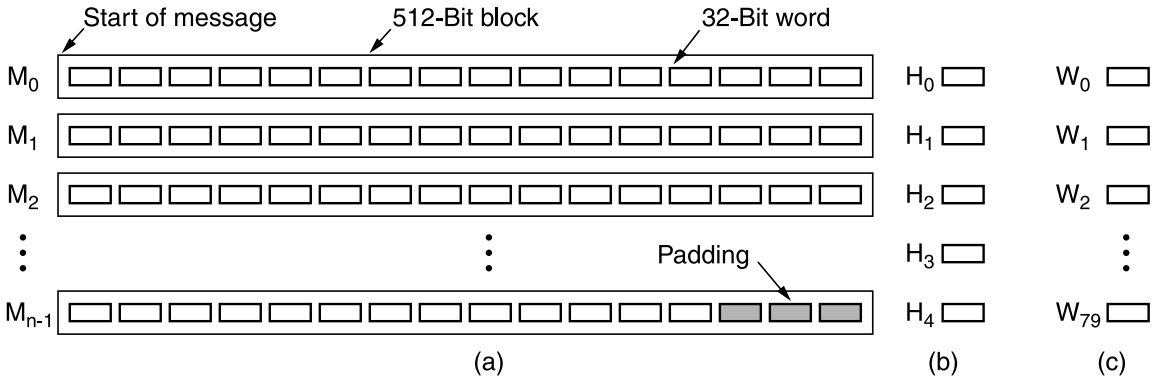

Figure 8-22. (a) A message padded out to a multiple of 512 bits. (b) The output variables. (c) The word array.

During the computation, SHA-1 maintains five 32-bit variables, H_0 through H_4, where the hash accumulates. These are shown in Fig. 8-22(b). They are initialized to constants specified in the standard.

Each of the blocks M_0 through M_{n-1} is now processed in turn. For the current block, the 16 words are first copied into the start of an auxiliary 80-word array, W, as shown in Fig. 8-22(c). Then the other 64 words in W are filled in using the formula

$$W_i = S^1(W_{i-3} \text{ XOR } W_{i-8} \text{ XOR } W_{i-14} \text{ XOR } W_{i-16}) \quad (16 \le i \le 79)$$

where $S^b(W)$ represents the left circular rotation of the 32-bit word, W, by b bits. Now five scratch variables, A through E, are initialized from H_0 through H_4, respectively.

The actual calculation can be expressed in pseudo-C as

```
for (i = 0; i < 80; i++) {
    temp = S^5(A) + f_i (B, C, D) + E + W_i + K_i;
    E = D;  D = C;  C = S^30(B);  B = A;  A = temp;
}
```

where the K_i constants are defined in the standard. The mixing functions f_i are defined as

$$f_i(B,C,D) = (B \text{ AND } C) \text{ OR } (\text{NOT } B \text{ AND } D) \quad (0 \le i \le 19)$$
$$f_i(B,C,D) = B \text{ XOR } C \text{ XOR } D \quad (20 \le i \le 39)$$
$$f_i(B,C,D) = (B \text{ AND } C) \text{ OR } (B \text{ AND } D) \text{ OR } (C \text{ AND } D) \quad (40 \le i \le 59)$$
$$f_i(B,C,D) = B \text{ XOR } C \text{ XOR } D \quad (60 \le i \le 79)$$

When all 80 iterations of the loop are completed, A through E are added to H_0 through H_4, respectively.

Now that the first 512-bit block has been processed, the next one is started. The W array is reinitialized from the new block, but H is left as it was. When this

block is finished, the next one is started, and so on, until all the 512-bit message blocks have been tossed into the soup. When the last block has been finished, the five 32-bit words in the *H* array are output as the 160-bit cryptographic hash. The complete C code for SHA-1 is given in RFC 3174.

New versions of SHA-1 have been developed that produce hashes of 224, 256, 384, and 512 bits. Collectively, these versions are called SHA-2. Not only are these hashes longer than SHA-1 hashes, but the digest function has been changed to combat some potential weaknesses of SHA-1. SHA-2 is not yet widely used, but it is likely to be in the future.

MD5

For completeness, we will mention another digest that is popular. **MD5** (Rivest, 1992) is the fifth in a series of message digests designed by Ronald Rivest. Very briefly, the message is padded to a length of 448 bits (modulo 512). Then the original length of the message is appended as a 64-bit integer to give a total input whose length is a multiple of 512 bits. Each round of the computation takes a 512-bit block of input and mixes it thoroughly with a running 128-bit buffer. For good measure, the mixing uses a table constructed from the sine function. The point of using a known function is to avoid any suspicion that the designer built in a clever back door through which only he can enter. This process continues until all the input blocks have been consumed. The contents of the 128-bit buffer form the message digest.

After more than a decade of solid use and study, weaknesses in MD5 have led to the ability to find collisions, or different messages with the same hash (Sotirov, et al., 2008). This is the death knell for a digest function because it means that the digest cannot safely be used to represent a message. Thus, the security community considers MD5 to be broken; it should be replaced where possible and no new systems should use it as part of their design. Nevertheless, you may still see MD5 used in existing systems.