APPENDIX P
MORE ON PGP

William Stallings

Supplement to
Cryptography and Network Security, Sixth Edition
Prentice Hall 2013
ISBN: 0133354695
http://williamstallings.com/Cryptography

This appendix supplements the material on PGP in Chapter 19.

P.1 CRYPTOGRAPHIC KEYS AND KEY RINGS

PGP makes use of four types of keys: one-time session symmetric keys, public keys, private keys, and passphrase-based symmetric keys (explained subsequently). Three separate requirements can be identified with respect to these keys.

1. A means of generating unpredictable session keys is needed.
2. We would like to allow a user to have multiple public-key/private-key pairs. One reason is that the user may wish to change his or her key pair from time to time. When this happens, any messages in the pipeline will be constructed with an obsolete key. Furthermore, recipients will know only the old public key until an update reaches them. In addition to the need to change keys over time, a user may wish to have multiple key pairs at a given time to interact with different groups of correspondents or simply to enhance security by limiting the amount of material encrypted with any one key. The upshot of all this is that there is not a one-to-one correspondence between users and their public keys. Thus, some means is needed for identifying particular keys.
3. Each PGP entity must maintain a file of its own public/private key pairs as well as a file of public keys of correspondents.

We examine each of these requirements in turn.

Session Key Generation

Each session key is associated with a single message and is used only for the purpose of encrypting and decrypting that message. Recall that message encryption/decryption is done with a symmetric encryption algorithm. CAST-128 and IDEA use 128-bit keys; 3DES uses a 168-bit key. For the following discussion, we assume CAST-128.

Random 128-bit numbers are generated using CAST-128 itself. The input to the random number generator consists of a 128-bit key and two 64-bit blocks that are treated as plaintext to be encrypted. Using cipher feedback mode, the CAST-128 encrypter produces two 64-bit cipher text blocks, which are concatenated to form the 128-bit session key. The algorithm that is used is based on the one specified in ANSI X12.17.

The "plaintext" input to the random number generator, consisting of two 64-bit blocks, is itself derived from a stream of 128-bit randomized numbers. These numbers are based on keystroke input from the user. Both the keystroke timing and the actual keys struck are used to generate the randomized stream. Thus, if the user hits arbitrary keys at his or her normal pace, a reasonably "random" input will be generated. This random input is also combined with previous session key output from CAST-128 to form the key input to the generator. The result, given the effective scrambling of CAST-128, is to produce a sequence of session keys that is effectively unpredictable.

Section P.3 discusses PGP random number generation techniques in more detail.

Key Identifiers

As we have discussed, an encrypted message is accompanied by an encrypted form of the session key that was used for message encryption. The session key itself is encrypted with the recipient's public key. Hence, only the recipient will be able to recover the session key and therefore

recover the message. If each user employed a single public/private key pair, then the recipient would automatically know which key to use to decrypt the session key: the recipient's unique private key. However, we have stated a requirement that any given user may have multiple public/private key pairs.

How, then, does the recipient know which of its public keys was used to encrypt the session key? One simple solution would be to transmit the public key with the message. The recipient could then verify that this is indeed one of its public keys, and proceed. This scheme would work, but it is unnecessarily wasteful of space. An RSA public key may be hundreds of decimal digits in length. Another solution would be to associate an identifier with each public key that is unique at least within one user. That is, the combination of user ID and key ID would be sufficient to identify a key uniquely. Then only the much shorter key ID would need to be transmitted. This solution, however, raises a management and overhead problem: Key IDs must be assigned and stored so that both sender and recipient could map from key ID to public key. This seems unnecessarily burdensome.

The solution adopted by PGP is to assign a key ID to each public key that is, with very high probability, unique within a user ID.[1] The key ID associated with each public key consists of its least significant 64 bits. That is, the key ID of public key PU_a is (PU_a mod 2^{64}). This is a sufficient length that the probability of duplicate key IDs is very small.

A key ID is also required for the PGP digital signature. Because a sender may use one of a number of private keys to encrypt the message digest, the recipient must know which public key is intended for use. Accordingly, the digital signature component of a message includes the 64-bit key ID of the required public key. When the message is received, the recipient verifies

[1] We have seen this introduction of probabilistic concepts before, in Section 8.3, for determining whether a number is prime. It is often the case in designing algorithms that the use of probabilistic techniques results in a less time-consuming, a less complex solution, or both.

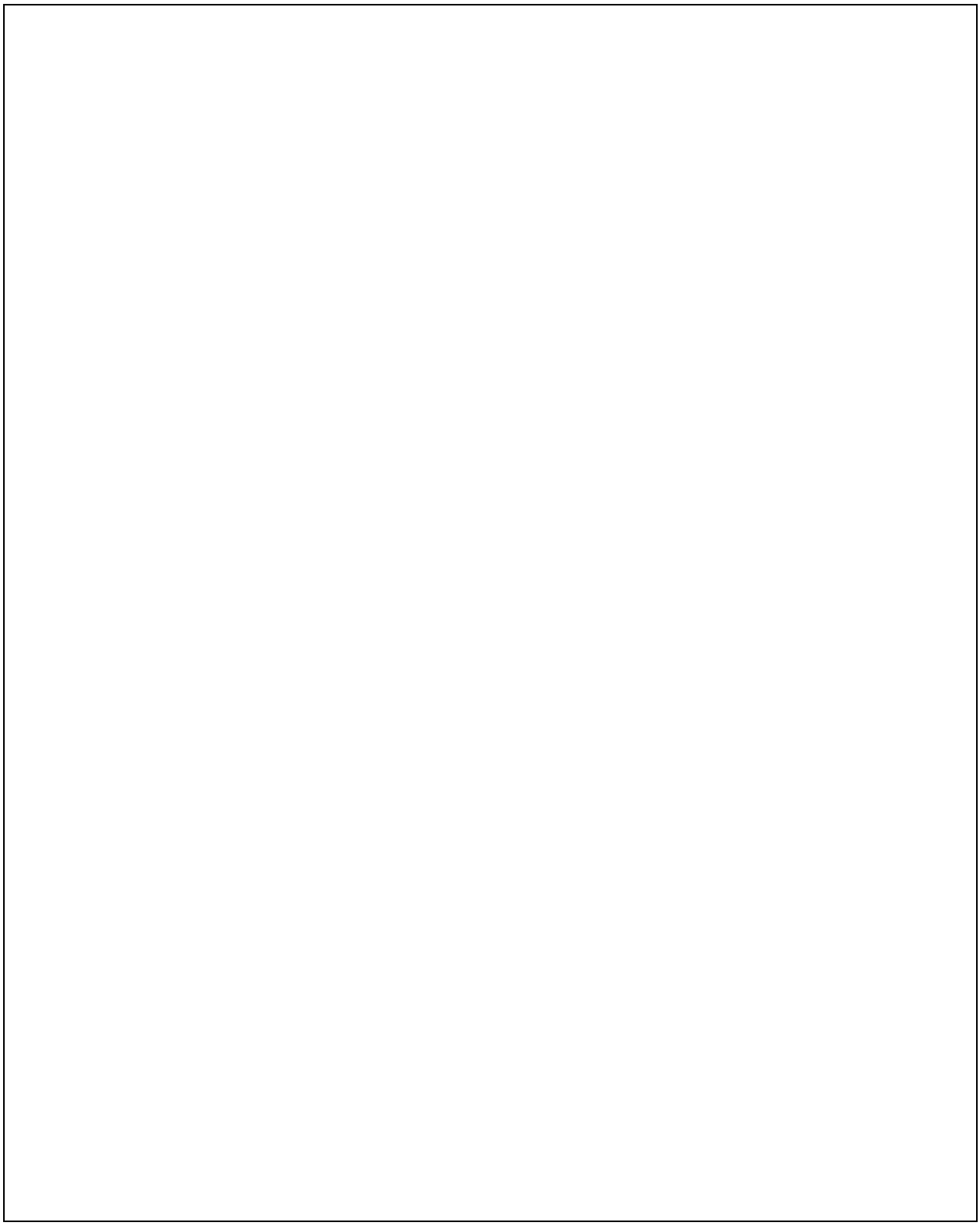
that the key ID is for a public key that it knows for that sender and then proceeds to verify the signature.

Now that the concept of key ID has been introduced, we can take a more detailed look at the format of a transmitted message, which is shown in Figure P.1. A message consists of three components: the message component, a signature (optional), and a session key component (optional).

The **message component** includes the actual data to be stored or transmitted, as well as a filename and a timestamp that specifies the time of creation.

The **signature component** includes the following.

- **Timestamp:** The time at which the signature was made.
- **Message digest:** The 160-bit SHA-1 digest encrypted with the sender's private signature key. The digest is calculated over the signature timestamp concatenated with the data portion of the message component. The inclusion of the signature timestamp in the digest insures against replay types of attacks. The exclusion of the filename and timestamp portions of the message component ensures that detached signatures are exactly the same as attached signatures prefixed to the message. Detached signatures are calculated on a separate file that has none of the message component header fields.
- **Leading two octets of message digest:** Enables the recipient to determine if the correct public key was used to decrypt the message digest for authentication by comparing this plaintext copy of the first two octets with the first two octets of the decrypted digest. These octets also serve as a 16-bit frame check sequence for the message.
- **Key ID of sender's public key:** Identifies the public key that should be used to decrypt the message digest and, hence, identifies the private key that was used to encrypt the message digest.



The message component and optional signature component may be compressed using ZIP and may be encrypted using a session key.

The **session key component** includes the session key and the identifier of the recipient's public key that was used by the sender to encrypt the session key.

The entire block is usually encoded with radix-64 encoding.

Key Rings

We have seen how key IDs are critical to the operation of PGP and that two key IDs are included in any PGP message that provides both confidentiality and authentication. These keys need to be stored and organized in a systematic way for efficient and effective use by all parties. The scheme used in PGP is to provide a pair of data structures at each node, one to store the public/private key pairs owned by that node and one to store the public keys of other users known at this node. These data structures are referred to, respectively, as the private-key ring and the public-key ring.

Figure P.2 shows the general structure of a **private-key ring**. We can view the ring as a table in which each row represents one of the public/private key pairs owned by this user. Each row contains the entries:

- **Timestamp:** The date/time when this key pair was generated.
- **Key ID:** The least significant 64 bits of the public key for this entry.
- **Public key:** The public-key portion of the pair.
- **Private key:** The private-key portion of the pair; this field is encrypted.
- **User ID:** Typically, this will be the user's e-mail address (e.g., stallings@acm.org). However, the user may choose to associate a different name with each pair (e.g., Stallings, WStallings, WilliamStallings, etc.) or to reuse the same User ID more than once.

The private-key ring can be indexed by either User ID or Key ID; later we will see the need for both means of indexing.

Private Key Ring

Timestamp	Key ID*	Public Key	Encrypted Private Key	User ID*
• • •	• • •	• • •	• • •	• • •
T_i	$PU_i \bmod 2^{64}$	PU_i	$E(H(P_i), PR_i)$	User i
• • •	• • •	• • •	• • •	• • •

Public Key Ring

Time-stamp	Key ID*	Public Key	Owner Trust	User ID*	Key Legitimacy	Signature(s)	Signature Trust(s)
• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •
T_i	$PU_i \bmod 2^{64}$	PU_i	$trust\_flag_i$	User i	$trust\_flag_i$		
• • •	• • •	• • •	• • •	• • •	• • •	• • •	• • •

* = field used to index table

Figure P.2 General Structure of Private and Public Key Rings

Although it is intended that the private-key ring be stored only on the machine of the user that created and owns the key pairs and that it be accessible only to that user, it makes sense to make the value of the private key as secure as possible. Accordingly, the private key itself is not stored in the key ring. Rather, this key is encrypted using CAST-128 (or IDEA or 3DES). The procedure is as follows:

1. The user selects a passphrase to be used for encrypting private keys.

2. When the system generates a new public/private key pair using RSA, it asks the user for the passphrase. Using SHA-1, a 160-bit hash code is generated from the passphrase, and the passphrase is discarded.
3. The system encrypts the private key using CAST-128 with the 128 bits of the hash code as the key. The hash code is then discarded, and the encrypted private key is stored in the private-key ring.

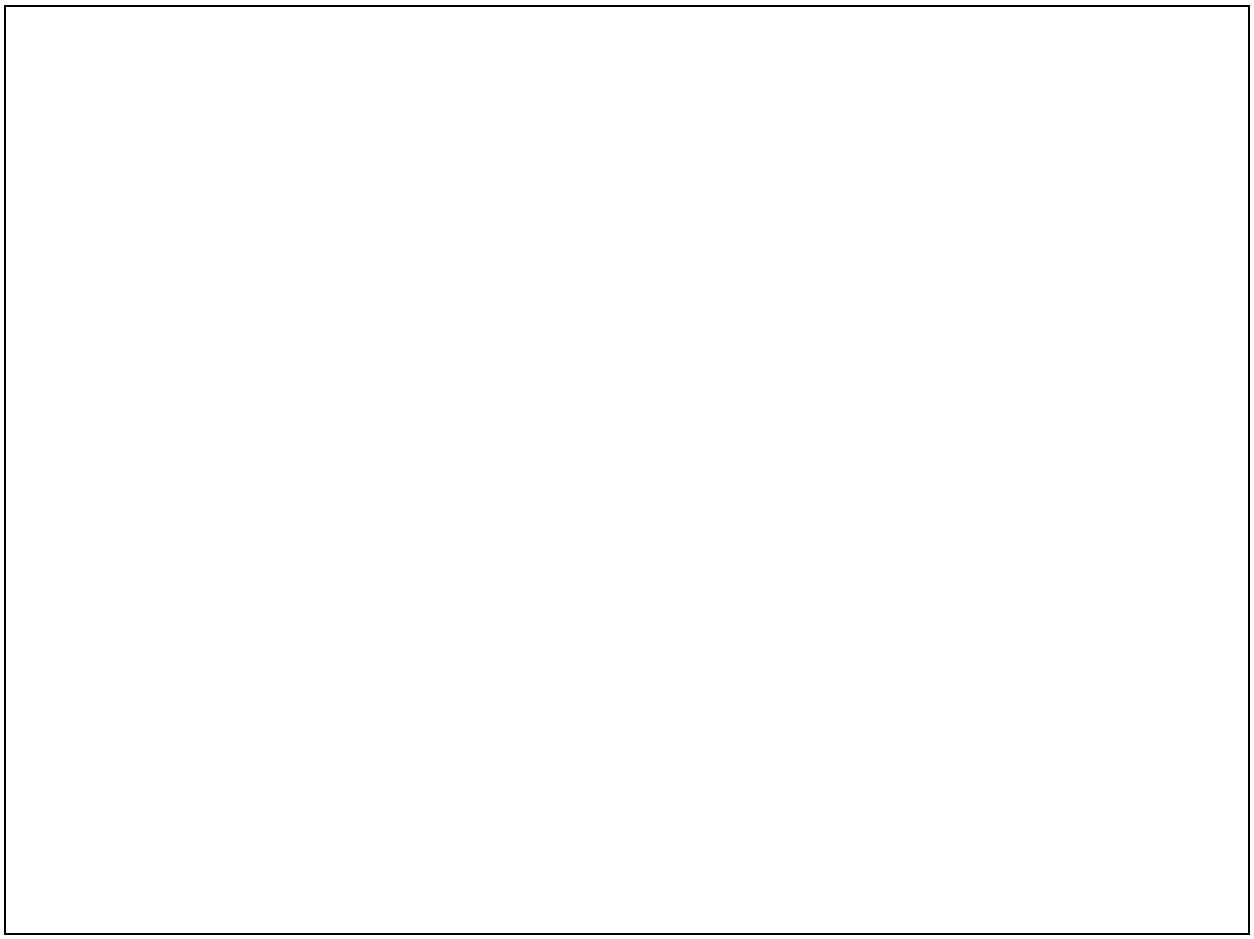
Subsequently, when a user accesses the private-key ring to retrieve a private key, he or she must supply the passphrase. PGP will retrieve the encrypted private key, generate the hash code of the passphrase, and decrypt the encrypted private key using CAST-128 with the hash code.

This is a very compact and effective scheme. As in any system based on passwords, the security of this system depends on the security of the password. To avoid the temptation to write it down, the user should use a passphrase that is not easily guessed but that is easily remembered.

Figure P.2 also shows the general structure of a **public-key ring**. This data structure is used to store public keys of other users that are known to this user. For the moment, let us ignore some fields shown in the figure and describe the following fields.

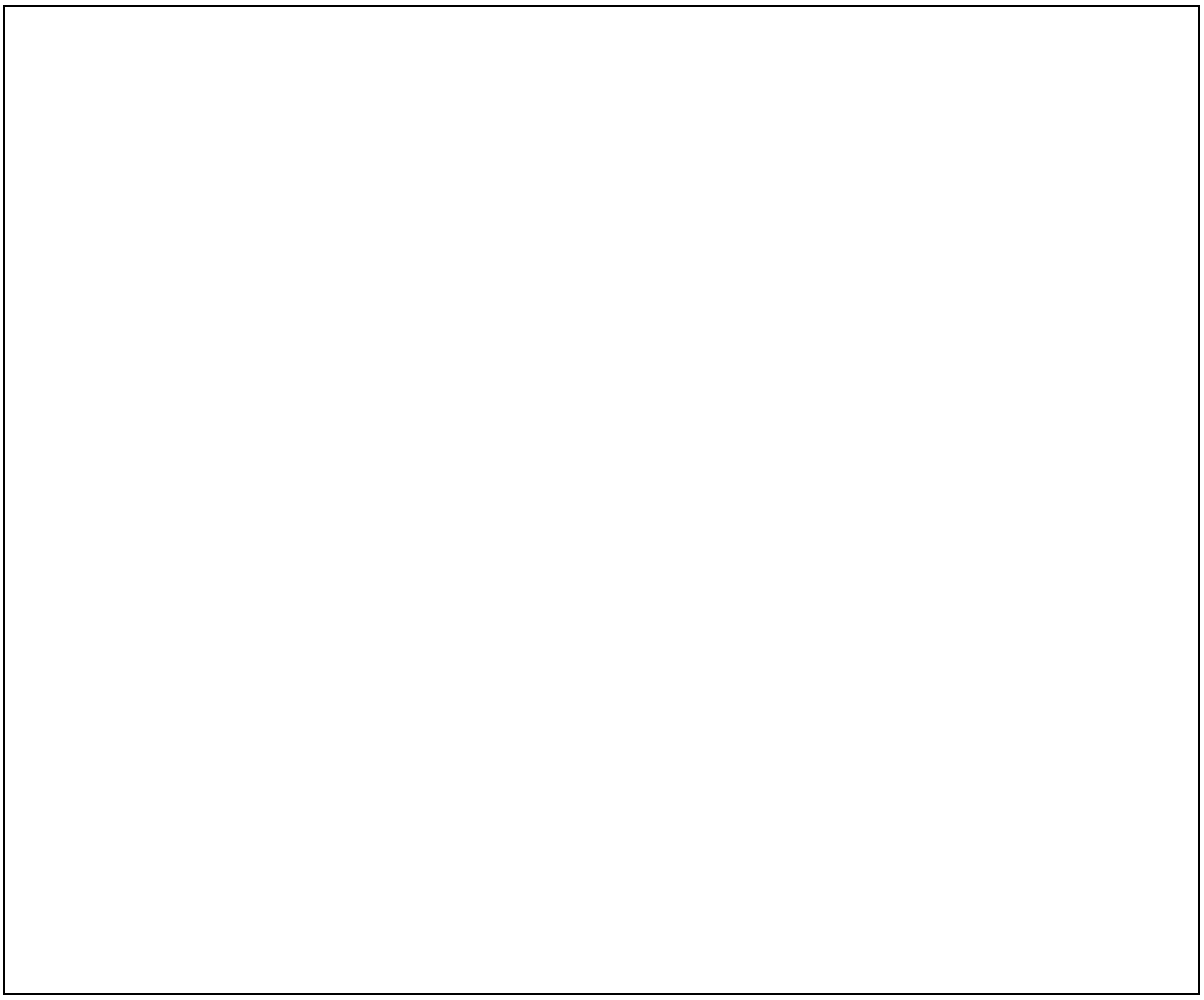
- **Timestamp:** The date/time when this entry was generated.
- **Key ID:** The least significant 64 bits of the public key for this entry.
- **Public Key:** The public key for this entry.
- **User ID:** Identifies the owner of this key. Multiple user IDs may be associated with a single public key.

The public-key ring can be indexed by either User ID or Key ID; we will see the need for both means of indexing later.



We are now in a position to show how these key rings are used in message transmission and reception. For simplicity, we ignore compression and radix-64 conversion in the following discussion. First consider message transmission (Figure P.3) and assume that the message is to be both signed and encrypted. The sending PGP entity performs the following steps.

1. Signing the message:
   a. PGP retrieves the sender's private key from the private-key ring using `your_userid` as an index. If `your_userid` was not provided in the command, the first private key on the ring is retrieved.
   b. PGP prompts the user for the passphrase to recover the unencrypted private key.



c. The signature component of the message is constructed.

2. Encrypting the message:

a. PGP generates a session key and encrypts the message.

b. PGP retrieves the recipient's public key from the public-key ring using her_userid as an index.

c. The session key component of the message is constructed.

The receiving PGP entity performs the following steps (Figure P.4).

1. Decrypting the message:

a. PGP retrieves the receiver's private key from the private-key ring using the Key ID field in the session key component of the message as an index.

b. PGP prompts the user for the passphrase to recover the unencrypted private key.

c. PGP then recovers the session key and decrypts the message.

2. Authenticating the message:

a. PGP retrieves the sender's public key from the public-key rin, using the Key ID field in the signature key component of the message as an index.

b. PGP recovers the transmitted message digest.

c. PGP computes the message digest for the received message and compares it to the transmitted message digest to authenticate.

P.2 PUBLIC-KEY MANAGEMENT

As can be seen from the discussion so far, PGP contains a clever, efficient, interlocking set of functions and formats to provide an effective confidentiality and authentication service. To complete the system, one final area needs to be addressed, that of public-key management. The PGP documentation captures the importance of this area:

> This whole business of protecting public keys from tampering is the single most difficult problem in practical public key applications. It is the "Achilles heel" of public key cryptography, and a lot of software complexity is tied up in solving this one problem.

PGP provides a structure for solving this problem with several suggested options that may be used. Because PGP is intended for use in a variety of

formal and informal environments, no rigid public-key management scheme is set up, such as we will see in our discussion of S/MIME later in this chapter.

Approaches to Public-Key Management

The essence of the problem is this: User A must build up a public-key ring containing the public keys of other users to interoperate with them using PGP. Suppose that A's key ring contains a public key attributed to B, but in fact the key is owned by C. This could happen, for example, if A got the key from a bulletin board system (BBS) that was used by B to post the public key but that has been compromised by C. The result is that two threats now exist. First, C can send messages to A and forge B's signature so that A will accept the message as coming from B. Second, any encrypted message from A to B can be read by C.

A number of approaches are possible for minimizing the risk that a user's public-key ring contains false public keys. Suppose that A wishes to obtain a reliable public key for B. The following are some approaches that could be used.

1. Physically get the key from B. B could store her public key (PU_b) on a floppy disk and hand it to A. A could then load the key into his system from the floppy disk. This is a very secure method but has obvious practical limitations.
2. Verify a key by telephone. If A can recognize B on the phone, A could call B and ask her to dictate the key, in radix-64 format, over the phone. As a more practical alternative, B could transmit her key in an e-mail message to A. A could have PGP generate a 160-bit SHA-1 digest of the key and display it in hexadecimal format; this is referred to as the "fingerprint" of the key. A could then call B and ask her to

dictate the fingerprint over the phone. If the two fingerprints match, the key is verified.

3. Obtain B's public key from a mutual trusted individual D. For this purpose, the introducer, D, creates a signed certificate. The certificate includes B's public key, the time of creation of the key, and a validity period for the key. D generates an SHA-1 digest of this certificate, encrypts it with her private key, and attaches the signature to the certificate. Because only D could have created the signature, no one else can create a false public key and pretend that it is signed by D. The signed certificate could be sent directly to A by B or D, or it could be posted on a bulletin board.
4. Obtain B's public key from a trusted certifying authority. Again, a public-key certificate is created and signed by the authority. A could then access the authority, providing a user name and receiving a signed certificate.

For cases 3 and 4, A already would have to have a copy of the introducer's public key and trust that this key is valid. Ultimately, it is up to A to assign a level of trust to anyone who is to act as an introducer.

The Use of Trust

Although PGP does not include any specification for establishing certifying authorities or for establishing trust, it does provide a convenient means of using trust, associating trust with public keys, and exploiting trust information.

The basic structure is as follows. Each entry in the public-key ring is a public-key certificate, as described in the preceding subsection. Associated with each such entry is a **key legitimacy field** that indicates the extent to which PGP will trust that this is a valid public key for this user; the higher

the level of trust, the stronger is the binding of this user ID to this key. This field is computed by PGP. Also associated with the entry are zero or more signatures that the key ring owner has collected that sign this certificate. In turn, each signature has associated with it a **signature trust field** that indicates the degree to which this PGP user trusts the signer to certify public keys. The key legitimacy field is derived from the collection of signature trust fields in the entry. Finally, each entry defines a public key associated with a particular owner, and an **owner trust field** is included that indicates the degree to which this public key is trusted to sign other public-key certificates; this level of trust is assigned by the user. We can think of the signature trust fields as cached copies of the owner trust field from another entry.

The three fields mentioned in the previous paragraph are each contained in a structure referred to as a trust flag byte. The content of this trust flag for each of these three uses is shown in Table P.1. Suppose that we are dealing with the public-key ring of user A. We can describe the operation of the trust processing as follows.

1. When A inserts a new public key on the public-key ring, PGP must assign a value to the trust flag that is associated with the owner of this public key. If the owner is A, and therefore this public key also appears in the private-key ring, then a value of *ultimate trust* is automatically assigned to the trust field. Otherwise, PGP asks A for his assessment of the trust to be assigned to the owner of this key, and A must enter the desired level. The user can specify that this owner is unknown, untrusted, marginally trusted, or completely trusted.

`

Table P.1 Contents of Trust Flag Byte

(a) Trust Assigned to Public-Key Owner (appears after key packet; user defined)	(b) Trust Assigned to Public Key/User ID Pair (appears after User ID packet; computed by PGP)	(c) Trust Assigned to Signature (appears after signature packet; cached copy of OWNERTRUST for this signator)
OWNERTRUST Field —undefined trust —unknown user —usually not trusted to sign other keys —usually trusted to sign other keys —always trusted to sign other keys —this key is present in secret key ring (ultimate trust) BUCKSTOP bit —set if this key appears in secret key ring	KEYLEGIT Field —unknown or undefined trust —key ownership not trusted —marginal trust in key ownership —complete trust in key ownership WARNONLY bit —set if user wants only to be warned when key that is not fully validated is used for encryption	SIGTRUST Field —undefined trust —unknown user —usually not trusted to sign other keys —usually trusted to sign other keys —always trusted to sign other keys —this key is present in secret key ring (ultimate trust) CONTIG bit —set if signature leads up a contiguous trusted certification path back to the ultimately trusted key ring owner

2. When the new public key is entered, one or more signatures may be attached to it. More signatures may be added later. When a signature is inserted into the entry, PGP searches the public-key ring to see if the author of this signature is among the known public-key owners. If so, the OWNERTRUST value for this owner is assigned to the SIGTRUST field for this signature. If not, an *unknown user* value is assigned.

3. The value of the key legitimacy field is calculated on the basis of the signature trust fields present in this entry. If at least one signature has a signature trust value of *ultimate*, then the key legitimacy value is set to complete. Otherwise, PGP computes a weighted sum of the trust

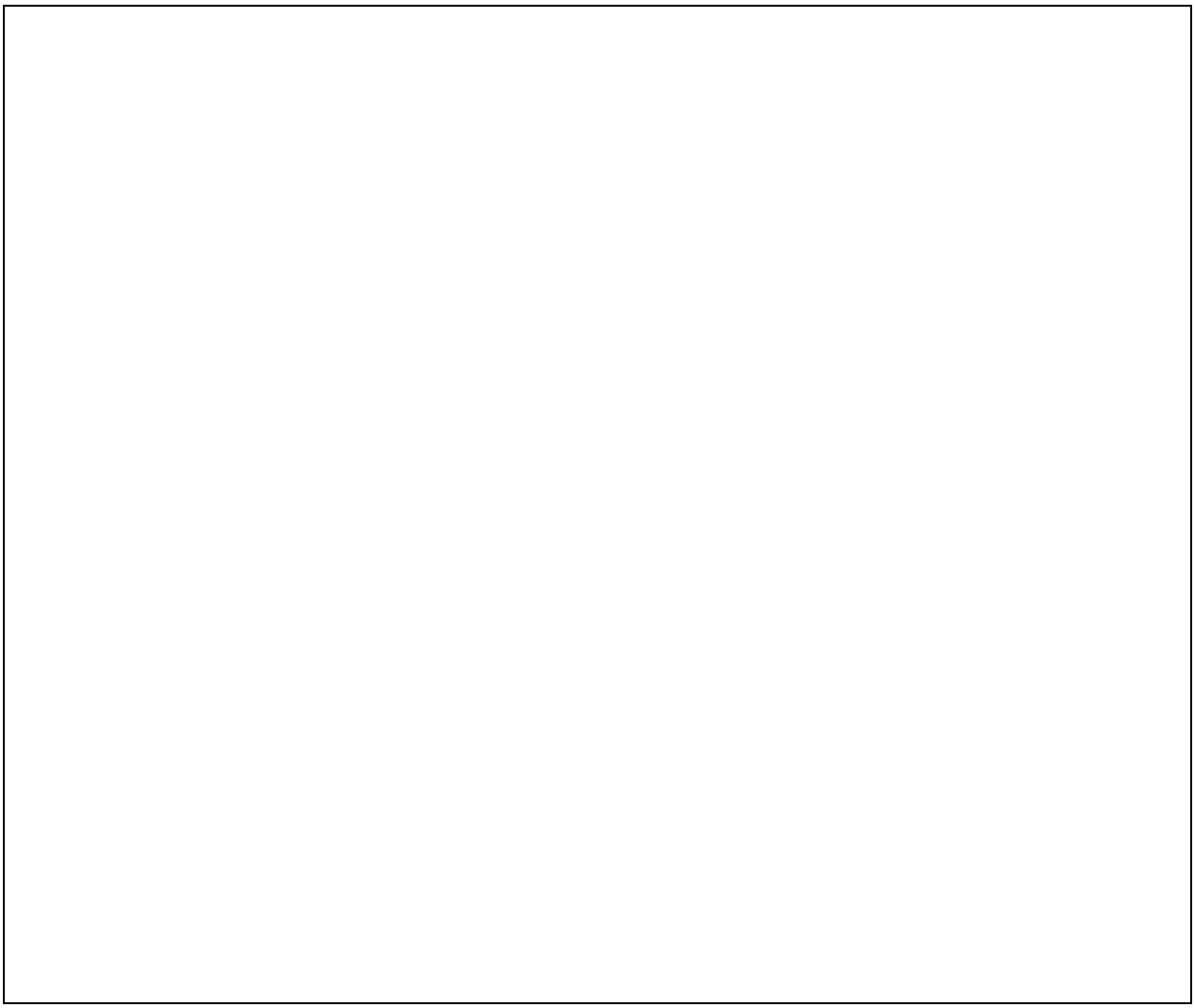
values. A weight of 1/*X* is given to signatures that are always trusted and 1/*Y* to signatures that are usually trusted, where *X* and *Y* are user-configurable parameters. When the total of weights of the introducers of a Key/UserID combination reaches 1, the binding is considered to be trustworthy, and the key legitimacy value is set to complete. Thus, in the absence of ultimate trust, at least *X* signatures that are always trusted, *Y* signatures that are usually trusted, or some combination is needed.

Periodically, PGP processes the public-key ring to achieve consistency. In essence, this is a top-down process. For each OWNERTRUST field, PGP scans the ring for all signatures authored by that owner and updates the SIGTRUST field to equal the OWNERTRUST field. This process starts with keys for which there is ultimate trust. Then all KEYLEGIT fields are computed on the basis of the attached signatures.

Figure P.5 provides an example of the way in which signature trust and key legitimacy are related.[2] The figure shows the structure of a public-key ring. The user has acquired a number of public keys — some directly from their owners and some from a third party such as a key server.

The node labeled "You" refers to the entry in the public-key ring corresponding to this user. This key is legitimate, and the OWNERTRUST value is ultimate trust. Each other node in the key ring has an OWNERTRUST value of undefined unless some other value is assigned by the user. In this example, this user has specified that it always trusts the following users to sign other keys: D, E, F, L. This user partially trusts users A and B to sign other keys.

[2] Figure provided to the author by Phil Zimmermann.



So the shading, or lack thereof, of the nodes in Figure P.5 indicates the level of trust assigned by this user. The tree structure indicates which keys have been signed by which other users. If a key is signed by a user whose key is also in this key ring, the arrow joins the signed key to the signatory. If the key is signed by a user whose key is not present in this key ring, the arrow joins the signed key to a question mark, indicating that the signatory is unknown to this user.

Several points are illustrated in Figure P.5.

1. Note that all keys whose owners are fully or partially trusted by this user have been signed by this user, with the exception of node L. Such a user signature is not always necessary, as the presence of node L indicates, but in practice, most users are likely to sign the keys for most owners that they trust. So, for example, even though E's key is already signed by trusted introducer F, the user chose to sign E's key directly.
2. We assume that two partially trusted signatures are sufficient to certify a key. Hence, the key for user H is deemed legitimate by PGP because it is signed by A and B, both of whom are partially trusted.
3. A key may be determined to be legitimate because it is signed by one fully trusted or two partially trusted signatories, but its user may not be trusted to sign other keys. For example, N's key is legitimate because it is signed by E, whom this user trusts, but N is not trusted to sign other keys because this user has not assigned N that trust value. Therefore, although R's key is signed by N, PGP does not consider R's key legitimate. This situation makes perfect sense. If you wish to send a private message to some individual, it is not necessary that you trust that individual in any respect. It is only necessary that you are sure that you have the correct public key for that individual.
4. Figure P.5 also shows an example of a detached "orphan" node S, with two unknown signatures. Such a key may have been acquired from a key server. PGP cannot assume that this key is legitimate simply because it came from a reputable server. The user must declare the key legitimate by signing it or by telling PGP that it is willing to trust fully one of the key's signatories.

A final point: Earlier it was mentioned that multiple user IDs may be associated with a single public key on the public-key ring. This could be

because a person has changed names or has been introduced via signature under multiple names, indicating different e-mail addresses for the same person, for example. So we can think of a public key as the root of a tree. A public key has a number of user IDs associating with it, with a number of signatures below each user ID. The binding of a particular user ID to a key depends on the signatures associated with that user ID and that key, whereas the level of trust in this key (for use in signing other keys) is a function of all the dependent signatures.

Revoking Public Keys

A user may wish to revoke his or her current public key either because compromise is suspected or simply to avoid the use of the same key for an extended period. Note that a compromise would require that an opponent somehow had obtained a copy of your unencrypted private key or that the opponent had obtained both the private key from your private-key ring and your passphrase.

The convention for revoking a public key is for the owner to issue a key revocation certificate, signed by the owner. This certificate has the same form as a normal signature certificate but includes an indicator that the purpose of this certificate is to revoke the use of this public key. Note that the corresponding private key must be used to sign a certificate that revokes a public key. The owner should then attempt to disseminate this certificate as widely and as quickly as possible to enable potential correspondents to update their public-key rings.

Note that an opponent who has compromised the private key of an owner can also issue such a certificate. However, this would deny the opponent as well as the legitimate owner the use of the public key, and therefore, it seems a much less likely threat than the malicious use of a stolen private key.

P.3 PGP RANDOM NUMBER GENERATION

PGP uses a complex and powerful scheme for generating random numbers and pseudorandom numbers, for a variety of purposes. PGP generates random numbers from the content and timing of user keystrokes, and pseudorandom numbers using an algorithm based on the one in ANSI X9.17. PGP uses these numbers for the following purposes:

- True random numbers:
  - —used to generate RSA key pairs
  - —provide the initial seed for the pseudorandom number generator
  - —provide additional input during pseudorandom number generation
- Pseudorandom numbers:
  - —used to generate session keys
  - —used to generate initialization vectors (IVs) for use with the session key in CFB mode encryption

True Random Numbers

PGP maintains a 256-byte buffer of random bits. Each time PGP expects a keystroke, it records the time, in 32-bit format, at which it starts waiting. When it receives the keystroke, it records the time the key was pressed and the 8-bit value of the keystroke. The time and keystroke information are used to generate a key, which is, in turn, used to encrypt the current value of the random-bit buffer.

Pseudorandom Numbers

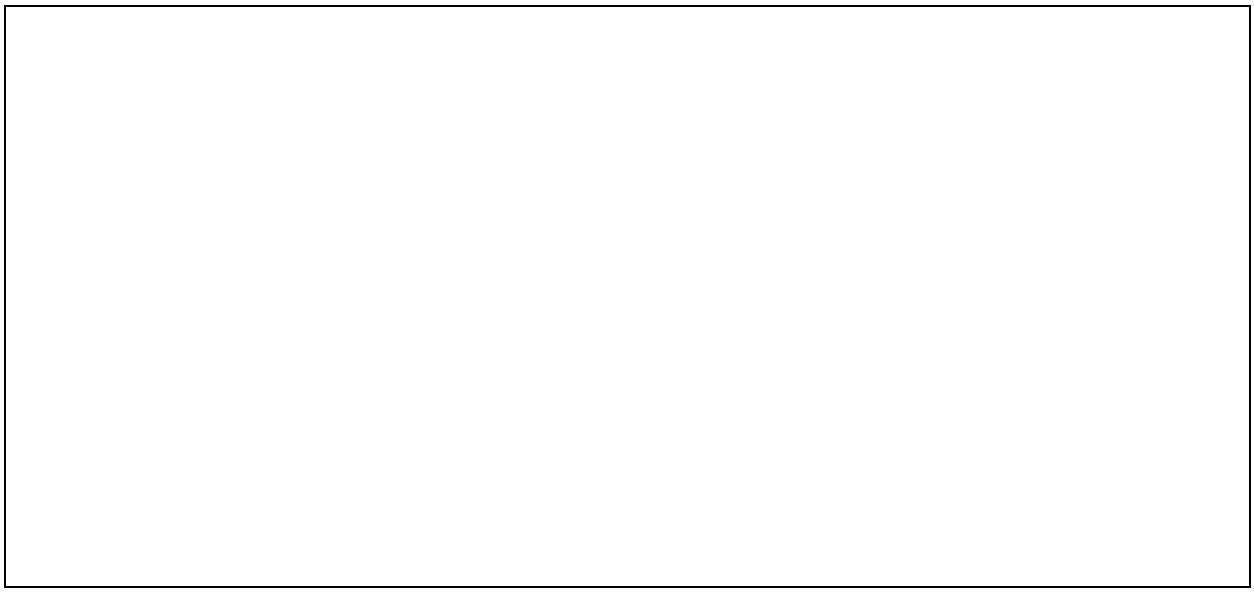
Pseudorandom number generation makes use of a 24-octet seed and produces a 16-octet session key, an 8-octet initialization vector, and a new

seed to be used for the next pseudorandom number generation. The algorithm is based on the X9.17 algorithm described in Chapter 7 (see Figure 7.5) but uses CAST-128 instead of triple DES for encryption. The algorithm uses the following data structures:

1. Input
   —randseed.bin (24 octets): If this file is empty, it is filled with 24 true random octets.
   —message: The session key and IV that will be used to encrypt a message are themselves a function of that message. This further contributes to the randomness of the key and IV, and if an opponent already knows the plaintext content of the message, there is no apparent need for capturing the one-time session key.

2. Output
   —K (24 octets): The first 16 octets, K[0..15], contain a session key, and the last eight octets, K[16..23], contain an IV.
   —randseed.bin (24 octets): A new seed value is placed in this file.

3. Internal data structures
   —dtbuf (8 octets): The first 4 octets, dtbuf[0..3], are initialized with the current date/time value. This buffer is equivalent to the DT variable in the X12.17 algorithm.
   —rkey (16 octets): CAST-128 encryption key used at all stages of the algorithm.
   —rseed (8 octets): Equivalent to the X12.17 V_i variable.
   —rbuf (8 octets): A pseudorandom number generated by the algorithm. This buffer is equivalent to the X12.17 R_i variable.
   —K' (24 octets): Temporary buffer for the new value of randseed.bin.



The algorithm consists of nine steps, G1 through G9. The first and last steps are obfuscation steps, intended to reduce the value of a captured randseed.bin file to an opponent. The remaining steps are essentially equivalent to three iterations of the X12.17 algorithm and are illustrated in Figure P.6 (compare Figure 7.5). To summarize,

G1. [Prewash previous seed]

a. Copy randseed.bin to K[0..23].

b. Take the hash of the message (this has already been generated if the message is being signed; otherwise the first 4K octets of the message are used). Use the result as a key, use a null IV, and encrypt K in CFB mode; store result back in K.

G2. [Set initial seed]

a. Set dtbuf[0..3] to the 32-bit local time. Set dtbuf[4..7] to all zeros. Copy rkey ← K[0..15]. Copy rseed ← K[16..23].

b. Encrypt the 64-bit dtbuf using the 128-bit rkey in ECB mode; store the result back in dtbuf.

G3. [Prepare to generate random octets] Set rcount ← 0 and k ← 23. The loop of steps G4-G7 will be executed 24 times (k = 23...0), once for each random octet produced and placed in K. The variable rcount is the number of unused random octets in rbuf. It will count down from 8 to 0 three times to generate the 24 octets.

G4. [Bytes available?] If rcount = 0 goto G5 else goto G7. Steps G5 and G6 perform one instance of the X12.17 algorithm to generate a new batch of eight random octets.

G5. [Generate new random octets]

a. rseed ← rseed ⊕ dtbuf

b. rbuf ← E_{rkey}[rseed] in ECB mode

G6. [Generate next seed]

a. rseed ← rbuf ⊕ dtbuf

b. rseed ← E_{rkey}[rseed] in ECB mode

c. Set rcount ← 8

G7. [Transfer one byte at a time from rbuf to K]

a. Set rcount ← rcount – 1

b. Generate a true random byte b, and set K[k] ← rbuf[rcount] ⊕ b

G8. [Done?] If k = 0 goto G9 else set k ← k – 1 and goto G4

G9. [Postwash seed and return result]

a. Generate 24 more bytes by the method of steps G4-G7, except do not XOR in a random byte in G7. Place the result in buffer K'

b. Encrypt K' with key K[0..15] and IV K[16..23] in CFB mode; store result in randseed.bin

c. Return K

It should not be possible to determine the session key from the 24 new octets generated in step G9.a. However, to make sure that the stored

randseed.bin file provides no information about the most recent session key, the 24 new octets are encrypted and the result is stored as the new seed.

This elaborate algorithm should provide cryptographically strong pseudorandom numbers.