PGP User's Guide, Volume I: Essential Topics

Philip Zimmermann

11 October 94

1 Introduction

Phil's Pretty Good Software presents PGP(tm) Pretty Good(tm) Privacy. Public key encryption for the masses. Document revised 11 October 94 for PGP version 2.6.2 – 11 Oct 94. Software by Philip Zimmermann, and many others.

Synopsis: PGP(tm) uses public-key encryption to protect E-mail and data files. Communicate securely with people you've never met, with no secure channels needed for prior exchange of keys. PGP is well featured and fast, with sophisticated key management, digital signatures, data compression, and good ergonomic design.

 For information on PGP licensing, distribution, copyrights, patents, trademarks, liability limitations, and export controls, see the "Legal Issues" section in the "PGP User's Guide, Volume II: Special Topics". Distributed by the Massachusetts Institute of Technology.

Contents

2 Quick overview

Pretty Good(tm) Privacy (PGP), from Phil's Pretty Good Software, is a high security cryptographic software application for MSDOS, UNIX, VAX/VMS, and other computers. PGP allows people to exchange files or messages with privacy, authentication, and convenience. Privacy means that only those intended to receive a message can read it. Authentication means that messages that appear to be from a particular person can only have originated from that person. Convenience means that privacy and authentication are provided without the hassles of managing keys associated with conventional cryptographic software. No secure channels are needed to exchange keys between users, which makes PGP much easier to use. This is because PGP is based on a powerful new technology called *public key* cryptography.

PGP combines the convenience of the Rivest-Shamir-Adleman (RSA) public key cryptosystem with the speed of conventional cryptography, message digests for digital signatures, data compression before encryption, good ergonomic design, and sophisticated key management. And PGP performs the public-key functions faster than most other software implementations. PGP is public key cryptography for the masses.

PGP does not provide any built-in modem communications capability. You must use a separate software product for that.

This document, *Volume I: Essential Topics*, only explains the essential concepts for using PGP, and should be read by all PGP users. *Volume II: Special Topics* covers the advanced features of PGP and other special topics, and may be read by more serious PGP users. Neither volume explains the underlying technology details of cryptographic algorithms and data structures.

3 Why do you need PGP?

It's personal. It's private. And it's no one's business but yours. You may be planning a political campaign, discussing your taxes, or having an illicit affair. Or you may be doing something that you feel shouldn't be illegal, but is. Whatever it is, you don't want your private electronic mail (E-mail) or confidential documents read by anyone else. There's nothing wrong with asserting your privacy. Privacy is as apple-pie as the Constitution.

Perhaps you think your E-mail is legitimate enough that encryption is unwarranted. If you really are a law-abiding citizen with nothing to hide, then why don't you always send your paper mail on postcards? Why not submit to drug testing on demand? Why require a warrant for police searches of your house? Are you trying to hide something? You must be a subversive or a drug dealer if you hide your mail inside envelopes. Or maybe a paranoid nut. Do law-abiding citizens have any need to encrypt their E-mail?

What if everyone believed that law-abiding citizens should use postcards for their mail? If some brave soul tried to assert his privacy by using an envelope for his mail, it would draw suspicion. Perhaps the authorities would open his mail to see what he's hiding. Fortunately, we don't live in that kind of world, because everyone protects most of their mail with envelopes. So no one draws suspicion by asserting their privacy with an envelope. There's safety in numbers. Analogously, it would be nice if everyone routinely used encryption for all their E-mail, innocent or not, so that no one drew suspicion by asserting their E-mail privacy with encryption. Think of it as a form of solidarity.

Today, if the Government wants to violate the privacy of ordinary citizens, it has to expend a certain amount of expense and labor to intercept and steam open and read paper mail, and listen to and possibly transcribe spoken telephone conversation. This kind of labor-intensive monitoring is not practical on a large scale. This is only done in important cases when it seems worthwhile.

More and more of our private communications are being routed through electronic channels. Electronic mail is gradually replacing conventional paper mail. E-mail messages are just too easy to intercept and scan for interesting keywords. This can be done easily, routinely, automatically, and undetectably on a grand scale. International cablegrams are already scanned this way on a large scale by the NSA.

We are moving toward a future when the nation will be crisscrossed with high capacity fiber optic data networks linking together all our increasingly ubiquitous personal computers. E-mail will be the norm for everyone, not the novelty it is today. The Government will protect our E-mail with Government-designed encryption protocols. Probably most people will acquiesce to that. But perhaps some people will prefer their own protective measures.

Senate Bill 266, a 1991 omnibus anti-crime bill, had an unsettling measure buried in it. If this non-binding resolution had become real law, it would have forced manufacturers of secure communications equipment to insert special *trap doors* in their products, so that the Government can read anyone's encrypted messages. It reads:

> *"It is the sense of Congress that providers of electronic communications services and manufacturers of electronic communications service equipment shall insure that communications*

> *systems permit the Government to obtain the plain text contents of voice, data, and other communications when appropriately authorized by law."*

This measure was defeated after rigorous protest from civil libertarians and industry groups.

In 1992, the FBI Digital Telephony wiretap proposal was introduced to Congress. It would require all manufacturers of communications equipment to build in special remote wiretap ports that would enable the FBI to remotely wiretap all forms of electronic communication from FBI offices. Although it never attracted any sponsors in Congress in 1992 because of citizen opposition, it was reintroduced in 1994.

Most alarming of all is the White House's bold new encryption policy initiative, under development at NSA since the start of the Bush administration, and unveiled April 16th, 1993. The centerpiece of this initiative is a Government-built encryption device, called the *Clipper* chip, containing a new classified NSA encryption algorithm. The Government is encouraging private industry to design it into all their secure communication products, like secure phones, secure FAX, etc. AT&T is now putting the Clipper into their secure voice products. The catch: At the time of manufacture, each Clipper chip will be loaded with its own unique key, and the Government gets to keep a copy, placed in escrow. Not to worry, though– the Government promises that they will use these keys to read your traffic only when duly authorized by law. Of course, to make Clipper completely effective, the next logical step would be to outlaw other forms of cryptography.

If privacy is outlawed, only outlaws will have privacy. Intelligence agencies have access to good cryptographic technology. So do the big arms and drug traffickers. So do defense contractors, oil companies, and other corporate giants. But ordinary people and grassroots political organizations mostly have not had access to affordable *military grade* public-key cryptographic technology. Until now.

PGP empowers people to take their privacy into their own hands. There's a growing social need for it. That's why I wrote it.

4 How it works

It would help if you were already familiar with the concept of cryptography in general and public key cryptography in particular. Nonetheless, here are a few introductory remarks about public key cryptography.

First, some elementary terminology. Suppose I want to send you a message, but I don't want anyone but you to be able to read it. I can *encrypt*, or *encipher* the message, which means I scramble it up in a hopelessly complicated way, rendering it unreadable to anyone except you, the intended recipient of the message. I supply a cryptographic *key* to encrypt the message, and you have to use the same key to decipher or *decrypt* it. At least that's how it works in conventional *single-key* cryptosystems.

In conventional cryptosystems, such as the US Federal Data Encryption Standard (DES), a single key is used for both encryption and decryption. This means that a key must be initially transmitted via secure channels so that both parties can know it before encrypted messages can be sent over insecure channels. This may be inconvenient. If you have a secure channel for exchanging keys, then why do you need cryptography in the first place?

In public key cryptosystems, everyone has two related complementary keys, a publicly revealed key and a secret key (also frequently called a private key). Each key unlocks the code that the other key makes. Knowing the public key does not help you deduce the corresponding secret key. The public key can be published and widely disseminated across a communications network. This protocol provides privacy without the need for the same kind of secure channels that a conventional cryptosystem requires.

Anyone can use a recipient's public key to encrypt a message to that person, and that recipient uses her own corresponding secret key to decrypt that message. No one but the recipient can decrypt it, because no one else has access to that secret key. Not even the person who encrypted the message can decrypt it.

Message authentication is also provided. The sender's own secret key can be used to encrypt a message, thereby *signing* it. This creates a digital signature of a message, which the recipient (or anyone else) can check by using the sender's public key to decrypt it. This proves that the sender was the true originator of the message, and that the message has not been subsequently altered by anyone else, because the sender alone possesses the secret key that made that signature. Forgery of a signed message is infeasible, and the sender cannot later disavow his signature.

These two processes can be combined to provide both privacy and authentication by first signing a message with your own secret key, then encrypting the signed message with the recipient's public key. The recipient reverses these steps by first decrypting the message with her own secret key, then checking the enclosed signature with your public key. These steps are done automatically by the recipient's software.

Because the public key encryption algorithm is much slower than conventional single-key encryption, encryption is better accomplished by using a high-quality fast conventional single-key encryption algorithm to encipher the message. This original unenciphered message is called *plaintext*. In a process invisible to the user, a temporary random key, created just for this one *session*, is used to conventionally encipher the plaintext file. Then the recipient's public key is used to encipher this temporary random conventional key. This public-key-enciphered conventional *session* key is sent along with the enciphered text (called *ciphertext*) to the recipient. The recipient uses her own secret key to recover this temporary session key, and then uses that key to run the fast conventional single-key algorithm to decipher the large ciphertext message.

Public keys are kept in individual *key certificates* that include the key owner's user ID (which is that person's name), a timestamp of when the key pair was generated, and the actual key material. Public key certificates contain the public key material, while secret key certificates contain the secret key material. Each secret key is also encrypted with its own password, in case it gets stolen. A key file, or *key ring* contains one or more of these key certificates. Public key rings contain public key certificates, and secret key rings contain secret key certificates.

The keys are also internally referenced by a *key ID*, which is an *abbreviation* of the public key (the least significant 64 bits of the large public key). When this key ID is displayed, only the lower 32 bits are shown for further brevity. While many keys may share the same user ID, for all practical purposes no two keys share the same key ID.

PGP uses *message digests* to form signatures. A message digest is a 128-bit cryptographically strong one-way hash function of the message. It is somewhat analogous to a *checksum* or CRC error checking code, in that it compactly *represents* the message and is used to detect changes in the message. Unlike a CRC, however, it is computationally infeasible for an attacker to devise a substitute message that would produce an identical message digest. The message digest gets encrypted by the secret key to form a signature.

Documents are signed by prefixing them with signature certificates, which contain the key ID of the key that was used to sign it, a secret-key-signed message digest of the document, and a timestamp of when the signature was made. The key ID is used by the receiver to look up the sender's public key to check the signature. The receiver's software automatically looks up the sender's public key and user ID in the receiver's public key ring.

Encrypted files are prefixed by the key ID of the public key used to encrypt them. The receiver uses this key ID message prefix to look up the secret key needed to decrypt the message. The receiver's software automatically looks up the necessary secret decryption key in the receiver's secret key ring.

These two types of key rings are the principal method of storing and managing public and secret keys. Rather than keep individual keys in separate key files, they are collected in key rings to facilitate the automatic lookup of keys either by key ID or by user ID. Each user keeps his own pair of key rings. An individual public key is temporarily kept in a separate file long enough to send to your friend who will then add it to her key ring.

5 Installing PGP

The MSDOS PGP release package comes in a compressed archive file with a file named in this form: *PGPxx.ZIP* (each release version has a different number for the *xx* in the filename). For example, the release package for version 2.6 is called PGP26.ZIP. The archive can be decompressed with the MSDOS shareware decompression utility *PKUNZIP*, or the UNIX utility *unzip*. When the PGP release package is decompressed, several files emerge from it. One such file, called README.DOC, should always be read before installing PGP. This file contains late-breaking news on what's new in this release of PGP, as well as information on what's in all the other files included in the release.

If you already have an earlier version of PGP, you should rename it or delete it, to avoid name conflicts with the new PGP.

For full details on how to install PGP, see the separate PGP Installation Guide, in the file SETUP.DOC included with this release package. It fully describes how to set up the PGP directory and your AUTOEXEC.BAT file and how to use PKUNZIP to install it. We will just briefly summarize the installation instructions here, in case you are too impatient to read the more detailed installation manual right now.

To install PGP on your MSDOS system, you have to copy the compressed archive PGPxx.ZIP file into a suitable directory on your hard disk (like C:\PGP), and decompress it with PKUNZIP. For best results, you should also modify your AUTOEXEC.BAT file, as described elsewhere in this manual, but you can do that later, after you've played with PGP a bit and read more of this manual. If you haven't run PGP before, the first step after installation (and reading this manual) is to make a pair of keys for yourself by running the PGP key generation command `pgp -kg`. Read the "RSA Key Generation" section of the manual first.

Installing on UNIX and VAX/VMS is generally similar to installing on MSDOS, but you may have to compile the source code first. A UNIX makefile is provided with the source release for this purpose.

6 How to use PGP

6.1 To see a usage summary

To see a quick command usage summary for PGP, just type:

```
pgp -h
```

6.2 Encrypting a message

To encrypt a plaintext file with the recipient's public key, type:

```
pgp -e textfile her_userid
```

This command produces a ciphertext file called textfile.pgp. A specific example is:

```
pgp -e letter.txt Alice
```

or:

```
pgp -e letter.txt "Alice S"
```

The first example searches your public key ring file *pubring.pgp* for any public key certificates that contain the string *Alice* anywhere in the user ID field. The second example would find any user IDs that contain *Alice S*. You can't use spaces in the string on the command line unless you enclose the whole string in quotes. The search is not case-sensitive. If it finds a matching public key, it uses it to encrypt the plaintext file *letter.txt*, producing a ciphertext file called *letter.pgp*.

PGP attempts to compress the plaintext before encrypting it, thereby greatly enhancing resistance to cryptanalysis. Thus the ciphertext file will likely be smaller than the plaintext file.

If you want to send this encrypted message through E-mail channels, convert it into printable ASCII *radix-64* format by adding the `-a` option, as described later.

6.3 Encrypting a message to multiple recipients

If you want to send the same message to more than one person, you may specify encryption for several recipients, any of whom may decrypt the same ciphertext file. To specify multiple recipients, just add more user IDs to the command line, like so:

```
pgp -e letter.txt Alice Bob Carol
```

This would create a ciphertext file called letter.pgp that could be decrypted by Alice or Bob or Carol. Any number of recipients may be specified.

6.4 Signing a message

To sign a plaintext file with your secret key, type:

```
pgp -s textfile [-u your_userid]
```

Note that [brackets] denote an optional field, so don't actually type real brackets.

This command produces a signed file called textfile.pgp. A specific example is:

```
pgp -s letter.txt -u Bob
```

This searches your secret key ring file *secring.pgp* for any secret key certificates that contain the string *Bob* anywhere in the user ID field. Your name is Bob, isn't it? The search is not case-sensitive. If it finds a matching secret key, it uses it to sign the plaintext file *letter.txt*, producing a signature file called *letter.pgp*

If you leave off the user ID field, the first key on your secret key ring is used as the default secret key for your signature.

PGP attempts to compress the message after signing it. Thus the signed file will likely be smaller than the original file, which is useful for archival applications. However, this renders the file unreadable to the casual human observer, even if the original message was ordinary ASCII text. It would be nice if you could make a signed file that was still directly readable to a human. This would be particularly useful if you want to send a signed message as E-mail.

For signing E-mail messages, where you most likely do want the result to be human-readable, it is probably most convenient to use the CLEARSIG feature, explained later. This allows the signature to be applied in printable form at the end of the text, and also disables compression of the text. This means the text is still human-readable by the recipient even if the recipient doesn't use PGP to check the signature. This is explained in detail in the section entitled "CLEARSIG - Enable Signed Messages to be Encapsulated as Clear Text", in the Special Topics volume. If you can't wait to read that section of the manual, you can see how an E-mail message signed this way would look, with this example:

```
pgp -sta message.txt
```

This would create a signed message in file *message.asc*, comprised of the original text, still human-readable, appended with a printable ASCII signature certificate, ready to send through an E-mail system. This example assumes that you are using the normal settings for enabling the CLEARSIG flag in the config file.

6.5 Signing and then encrypting

To sign a plaintext file with your secret key, and then encrypt it with the recipient's public key:

```
pgp -es textfile her_userid [-u your_userid]
```

Note that [brackets] denote an optional field, so don't actually type real brackets.

This example produces a nested ciphertext file called textfile.pgp. Your secret key to create the signature is automatically looked up in your secret key ring via your_userid. Her public encryption key is automatically looked up in your public key ring via her_userid. If you leave off her user ID field from the command line, you will be prompted for it.

If you leave off your own user ID field, the first key on your secret key ring is be used as the default secret key for your signature.

Note that PGP attempts to compress the plaintext before encrypting it.

If you want to send this encrypted message through E-mail channels, convert it into printable ASCII *radix-64* format by adding the `-a` option, as described later.

Multiple recipients may be specified by adding more user IDs to the command line.

6.6 Using just conventional encryption

Sometimes you just need to encrypt a file the old-fashioned way, with conventional single-key cryptography. This approach is useful for protecting archive files that will be stored but will not be sent to anyone else. Since the same person that encrypted the file will also decrypt the file, public key cryptography is not really necessary.

To encrypt a plaintext file with just conventional cryptography, type:

```
pgp -c textfile
```

This example encrypts the plaintext file called textfile, producing a ciphertext file called textfile.pgp, without using public key cryptography, key rings, user IDs, or any of that stuff. It prompts you for a pass phrase to use as a conventional key to encipher the file. This pass phrase need not be (and, indeed, SHOULD not be) the same pass phrase that you use to protect your own secret key. Note that PGP attempts to compress the plaintext before encrypting it.

PGP will not encrypt the same plaintext the same way twice, even if you used the same pass phrase every time.

6.7 Decrypting and checking signatures

To decrypt an encrypted file, or to check the signature integrity of a signed file:

```
pgp ciphertextfile [-o plaintextfile]
```

Note that [brackets] denote an optional field, so don't actually type real brackets.

The ciphertext file name is assumed to have a default extension of *.pgp*. The optional plaintext output file name specifies where to put processed plaintext output. If no name is specified, the ciphertext filename is used, with no extension. If a signature is nested inside of an encrypted file, it is automatically decrypted and the signature integrity is checked. The full user ID of the signer is displayed.

Note that the *unwrapping* of the ciphertext file is completely automatic, regardless of whether the ciphertext file is just signed, just encrypted, or both. PGP uses the key ID prefix in the ciphertext file to automatically find the appropriate secret decryption key on your secret key ring. If there is a nested signature, PGP then uses the key ID prefix in the nested signature to automatically find the appropriate public key on your public key ring to check the signature. If all the right keys are already present on your key rings, no user intervention is required, except that you will be prompted for your password for your secret key if necessary. If the ciphertext file was conventionally encrypted without public key cryptography, PGP recognizes this and prompts you for the pass phrase to conventionally decrypt it.

6.8 Managing keys

Since the time of Julius Caesar, key management has always been the hardest part of cryptography. One of the principal distinguishing features of PGP is its sophisticated key management.

6.8.1 RSA key generation

To generate your own unique public/secret key pair of a specified size, type:

```
pgp -kg
```

PGP shows you a menu of recommended key sizes (low commercial grade, high commercial grade, or "military" grade) and prompts you for what size key you want, up to more than a thousand bits. The bigger the key, the more security you get, but you pay a price in speed.

It also asks for a user ID, which means your name. It's a good idea to use your full name as your user ID, because then there is less risk of other people using the wrong public key to encrypt messages to you. Spaces and punctuation are allowed in the user ID. It would help if you put your E-mail address in ¡angle brackets¿ after your name, like so:

```
Robert M. Smith <rms@xyzcorp.com>
```

If you don't have an E-mail address, use your phone number or some other unique information that would help ensure that your user ID is unique.

PGP also asks for a *pass phrase* to protect your secret key in case it falls into the wrong hands. Nobody can use your secret key file without this pass phrase. The pass phrase is like a password, except that it can be a whole phrase or sentence with many words, spaces, punctuation, or anything else you want in it. Don't lose this pass phrase—there's no way to recover it if you do lose it. This pass phrase will be needed later every time you use your secret key. The pass phrase is case-sensitive, and should not be too short or easy to guess. It is never displayed on the screen. Don't leave it written down anywhere where someone else can see it, and don't store it on your computer. If you don't want a pass phrase (You fool!), just press return (or enter) at the pass phrase prompt.

The public/secret key pair is derived from large truly random numbers derived mainly from measuring the intervals between your keystrokes with a fast timer. The software will ask you to enter some random text to help it accumulate some random bits for the keys. When asked, you should provide some keystrokes that are reasonably random in their timing, and it wouldn't hurt to make the actual characters that you type irregular in content as well. Some of the randomness is derived from the unpredictability of the content of what you type. So don't just type repeated sequences of characters.

Note that RSA key generation is a lengthy process. It may take a few seconds for a small key on a fast processor, or quite a few minutes for a large key on an old IBM PC/XT. PGP will visually indicate its progress during key generation.

The generated key pair will be placed on your public and secret key rings. You can later use the `-kx` command option to extract (copy) your new public key from your public key ring and place it in a separate public key file suitable for distribution to your friends. The public key file can be sent to your friends for inclusion in their public key rings. Naturally, you keep your secret key file to yourself, and you should include it on your secret key ring. Each secret key on a key ring is individually protected with its own pass phrase.

Never give your secret key to anyone else. For the same reason, don't make key pairs for your friends. Everyone should make their own key pair. Always keep physical control of your secret key, and don't risk exposing it by storing it on a remote timesharing computer. Keep it on your own personal computer.

If PGP complains about not being able to find the PGP User's Guide on your computer, and refuses to generate a key pair without it, don't panic. Just read the explanation of the NOMANUAL parameter in the section "Setting Configuration Parameters" in the Special Topics volume of the PGP User's Guide.

6.8.2 Adding a key to your key ring

Sometimes you will want to add to your keyring a key provided to you by someone else, in the form of a keyfile.

To add a public or secret key file's contents to your public or secret key ring (note that [brackets] denote an optional field):

```
pgp -ka keyfile [keyring]
```

The keyfile extension defaults to *.pgp*. The optional keyring file name defaults to *pubring.pgp* or *secring.pgp*, depending on whether the keyfile contains a public or a secret key. You may specify a different key ring file name, with the extension defaulting to *.pgp*.

If the key is already on your key ring, PGP will not add it again. All of the keys in the keyfile are added to the keyring, except for duplicates.

Later in the manual, we will explain the concept of certifying keys with signatures. If the key being added has attached signatures certifying it, the signatures are added with the key. If the key is already on your key ring, PGP just merges in any new certifying signatures for that key that you don't already have on your key ring.

PGP was originally designed for handling small personal keyrings. If you want to handle really big keyrings, see the section on "Handling Large Public Keyrings" in the Special Topics volume.

6.8.3 Removing a key or user ID from your key ring

To remove a key or a user ID from your public key ring:

```
pgp -kr userid [keyring]
```

This searches for the specified user ID in your key ring, and removes it if it finds a match. Remember that any fragment of the user ID will suffice for a match. The optional keyring file name is assumed to be literally *pubring.pgp*. It can be omitted, or you can specify *secring.pgp* if you want to remove a secret key. You may specify a different key ring file name. The default key ring extension is *.pgp*.

If more than one user ID exists for this key, you will be asked if you want to remove only the user ID you specified, while leaving the key and its other user IDs intact.

6.8.4 Extracting (copying) a key from your key ring

To extract (copy) a key from your public or secret key ring:

```
pgp -kx userid keyfile [keyring]
```

This non-destructively copies the key specified by the user ID from your public or secret key ring to the specified key file. This is particularly useful if you want to give a copy of your public key to someone else.

If the key has any certifying signatures attached to it on your key ring, they are copied off along with the key.

If you want the extracted key represented in printable ASCII characters suitable for email purposes, use the `-kxa` options.

6.8.5 Viewing the contents of your key ring

To view the contents of your public key ring:

```
pgp -kv[v] [userid] [keyring]
```

This lists any keys in the key ring that match the specified user ID substring. If you omit the user ID, all of the keys in the key ring are listed. The optional keyring file name is assumed to be *pubring.pgp*. It can be omitted, or you can specify *secring.pgp* if you want to list secret keys. If you want to specify a different key ring file name, you can. The default key ring extension is *.pgp*.

Later in the manual, we will explain the concept of certifying keys with signatures. To see all the certifying signatures attached to each key, use the `-kvv` option:

```
pgp -kvv [userid] [keyring]
```

If you want to specify a particular key ring file name, but want to see all the keys in it, try this alternative approach:

```
pgp keyfile
```

With no command options specified, PGP lists all the keys in keyfile.pgp, and also attempts to add them to your key ring if they are not already on your key ring.

6.8.6 How to protect public keys from tampering

In a public key cryptosystem, you don't have to protect public keys from exposure. In fact, it's better if they are widely disseminated. But it is important to protect public keys from tampering, to make sure that a public key really belongs to whom it appears to belong to. This may be the most important vulnerability of a public-key cryptosystem. Let's first look at a potential disaster, then at how to safely avoid it with PGP.

Suppose you wanted to send a private message to Alice. You download Alice's public key certificate from an electronic bulletin board system (BBS). You encrypt your letter to Alice with this public key and send it to her through the BBS's E-mail facility.

Unfortunately, unbeknownst to you or Alice, another user named Charlie has infiltrated the BBS and generated a public key of his own with Alice's user ID attached to it. He covertly substitutes his bogus key in place of Alice's real public key. You unwittingly use this bogus key belonging to Charlie instead of Alice's public key. All looks normal because this bogus key has Alice's user ID. Now Charlie can decipher the message intended for Alice because he has the matching secret key. He may even re-encrypt the deciphered message with Alice's real public key and send it on to her so that no one suspects any wrongdoing. Furthermore, he can even make apparently good signatures from Alice with this secret key because everyone will use the bogus public key to check Alice's signatures.

The only way to prevent this disaster is to prevent anyone from tampering with public keys. If you got Alice's public key directly from Alice, this is no problem. But that may be difficult if Alice is a thousand miles away, or is currently unreachable.

Perhaps you could get Alice's public key from a mutual trusted friend David who knows he has a good copy of Alice's public key. David could sign Alice's public key, vouching for the integrity of Alice's public key. David would create this signature with his own secret key.

This would create a signed public key certificate, and would show that Alice's key had not been tampered with. This requires you have a known good copy of David's public key to check his signature. Perhaps David could provide Alice with a signed copy of your public key also. David is thus serving as an *introducer* between you and Alice.

This signed public key certificate for Alice could be uploaded by David or Alice to the BBS, and you could download it later. You could then check the signature via David's public key and thus be assured that this is really Alice's public key. No impostor can fool you into accepting his own bogus key as Alice's because no one else can forge signatures made by David.

A widely trusted person could even specialize in providing this service of *introducing* users to each other by providing signatures for their public key certificates. This trusted person could be regarded as a *key server*, or as a *Certifying Authority*. Any public key certificates bearing the key server's signature could be trusted as truly belonging to whom they appear to belong to. All users who wanted to participate would need a known good copy of just the key server's public key, so that the key server's signatures could be verified.

A trusted centralized key server or Certifying Authority is especially appropriate for large impersonal centrally-controlled corporate or government institutions. Some institutional environments use hierarchies of Certifying Authorities.

For more decentralized grassroots *guerrilla style* environments, allowing all users to act as a trusted introducers for their friends would probably work better than a centralized key server. PGP tends to emphasize this organic decentralized non-institutional approach. It better reflects the natural way humans interact on a personal social level, and allows people to better choose who they can trust for key management.

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.

You should use a public key only after you are sure that it is a good public key that has not been tampered with, and actually belongs to the person it claims to. You can be sure of this if you got this public key certificate directly from its owner, or if it bears the signature of someone else that you trust, from whom you already have a good public key. Also, the user ID should have the full name of the key's owner, not just her first name.

No matter how tempted you are—and you will be tempted—never, NEVER give in to expediency and trust a public key you downloaded from a bulletin board, unless it is signed by someone you trust. That uncertified public key could have been tampered with by anyone, maybe even by the system administrator of the bulletin board.

If you are asked to sign someone else's public key certificate, make certain that it really belongs to that person named in the user ID of that public key certificate. This is because your signature on her public key certificate is a promise by you that this public key really belongs to her. Other people who trust you will accept her public key because it bears your signature. It may be ill-advised to rely on hearsay—don't sign her public key unless you have independent firsthand knowledge that it really belongs to her. Preferably, you should sign it only if you got it directly from her.

In order to sign a public key, you must be far more certain of that key's ownership than if you merely want to use that key to encrypt a message. To be convinced of a key's validity enough to use it, certifying signatures from trusted introducers should suffice. But to sign a key yourself, you should require your own independent firsthand knowledge of who owns that key. Perhaps you could call the key's owner on the phone and read the key file to her to get her to confirm that the key you have really is her key—and make sure you really are talking to the right person. See the section called "Verifying a Public Key Over the Phone" in the Special Topics volume for further details.

Bear in mind that your signature on a public key certificate does not vouch for the integrity of that person, but only vouches for the integrity (the ownership) of that person's public key. You aren't risking your credibility by signing the public key of a sociopath, if you were completely confident that the key really belonged to him. Other people would accept that key as belonging to him because you signed it (assuming they trust you), but they wouldn't trust that key's owner. Trusting a key is not the same as trusting the key's owner.

Trust is not necessarily transferable; I have a friend who I trust not to lie. He's a gullible person who trusts the President not to lie. That doesn't mean I trust the President not to lie. This is just common sense. If I trust Alice's signature on a key, and Alice trusts Charlie's signature on a key, that does not imply that I have to trust Charlie's signature on a key.

It would be a good idea to keep your own public key on hand with a collection of certifying signatures attached from a variety of *introducers*, in the hopes that most people will trust at least one of the introducers who vouch for your own public key's validity. You could post your key with its attached collection of certifying signatures on various electronic bulletin boards. If you sign someone else's public key, return it to them with your signature so that they can add it to their own collection of credentials for their own public key.

PGP keeps track of which keys on your public key ring are properly certified with signatures from introducers that you trust. All you have to do is tell PGP which people you trust as introducers, and certify their keys yourself with your own ultimately trusted key. PGP can take it from there, automatically validating any other keys that have been signed by your designated introducers. And of course you may directly sign more keys yourself. More on this later.

Make sure no one else can tamper with your own public key ring. Checking a new signed public key certificate must ultimately depend on the integrity of the trusted public keys that are already on your own public key ring. Maintain physical control of your public key ring, preferably on your own personal computer rather than on a remote timesharing system, just as you would do for your secret key. This is to protect it from tampering, not from disclosure. Keep a trusted backup copy of your public key ring and your secret key ring on write-protected media.

Since your own trusted public key is used as a final authority to directly or indirectly certify all the other keys on your key ring, it is the most important key to protect from tampering. To detect any tampering of your own ultimately-trusted public key, PGP can be set up to automatically compare your public key against a backup copy on write-protected media. For details, see the description of the `-kc` key ring check command in the Special Topics volume.

PGP generally assumes you will maintain physical security over your system and your key rings, as well as your copy of PGP itself. If an intruder can tamper with your disk, then in theory he can tamper with PGP itself, rendering moot the safeguards PGP may have to detect tampering with keys.

One somewhat complicated way to protect your own whole public key ring from tampering is to sign the whole ring with your own secret key. You could do this by making a detached signature certificate of the public key ring, by signing the ring with the `-sb` options (see the section called "Separating Signatures from Messages" in the PGP User's Guide, Special Topics volume). Unfortunately, you would still have to keep a separate trusted copy of your own public key around to check the signature you made. You couldn't rely on your own public key stored on your public key ring to check the signature you made for the whole ring, because that is part of what you're trying to check.

6.8.7 How does PGP keep track of which keys are valid?

Before you read this section, be sure to read the above section on "How to Protect Public Keys from Tampering".

PGP keeps track of which keys on your public key ring are properly certified with signatures from introducers that you trust. All you have to do is tell PGP which people you trust as introducers, and certify their keys yourself with your own ultimately trusted key. PGP can take it from there, automatically validating any other keys that have been signed by your designated introducers. And of course you may directly sign more keys yourself.

There are two entirely separate criteria PGP uses to judge a public key's usefulness—don't get them confused:

1. Does the key actually belong to whom it appears to belong? In other words, has it been certified with a trusted signature?
2. Does it belong to someone you can trust to certify other keys?

PGP can calculate the answer to the first question. To answer the second question, PGP must be explicitly told by you, the user. When you supply the answer to question 2, PGP can then calculate the answer to question 1 for other keys signed by the introducer you designated as trusted.

Keys that have been certified by a trusted introducer are deemed valid by PGP. The keys belonging to trusted introducers must themselves be certified either by you or by other trusted introducers.

PGP also allows for the possibility of you having several shades of trust for people to act as introducers. Your trust for a key's owner to act as an introducer does not just reflect your estimation of their personal integrity—it should also reflect how competent you think they are at understanding key management and using good judgment in signing keys. You can designate a person to PGP as unknown, untrusted, marginally trusted, or completely trusted to certify other public keys. This trust information is stored on your key ring with their key, but when you tell PGP to copy a key off your key ring, PGP will not copy the trust information along with the key, because your private opinions on trust are regarded as confidential.

When PGP is calculating the validity of a public key, it examines the trust level of all the attached certifying signatures. It computes a weighted score of validity—two marginally trusted signatures are deemed as credible as one fully trusted signature. PGP's skepticism is adjustable—for example, you may tune PGP to require two fully trusted signatures or three marginally trusted signatures to judge a key as valid.

Your own key is *"axiomatically"* valid to PGP, needing no introducer's signature to prove its validity. PGP knows which public keys are yours, by looking for the corresponding secret keys on the secret key ring. PGP also assumes you ultimately trust yourself to certify other keys.

As time goes on, you will accumulate keys from other people that you may want to designate as trusted introducers. Everyone else will each choose their own trusted introducers. And everyone will gradually accumulate and distribute with their key a collection of certifying signatures from other people, with the expectation that anyone receiving it will trust at least one or two of the signatures. This will cause the emergence of a decentralized fault-tolerant web of confidence for all public keys.

This unique grass-roots approach contrasts sharply with Government standard public key management schemes, such as Internet Privacy Enhanced Mail (PEM), which are based on centralized control and mandatory centralized trust. The standard schemes rely on a hierarchy of Certifying Authorities who dictate who you must trust. PGP's decentralized probabilistic method for determining public key legitimacy is the centerpiece of its key management architecture. PGP lets you alone choose who you trust, putting you at the top of your own private certification pyramid. PGP is for people who prefer to pack their own parachutes.

6.8.8 How to protect secret keys from disclosure

Protect your own secret key and your pass phrase carefully. Really, really carefully. If your secret key is ever compromised, you'd better get the word out quickly to all interested parties (good luck) before someone else uses it to make signatures in your name. For example, they could use it to sign bogus public key certificates, which could create problems for many people, especially if your signature is widely trusted. And of course, a compromise of your own secret key could expose all messages sent to you.

To protect your secret key, you can start by always keeping physical control of your secret key. Keeping it on your personal computer at home is OK, or keep it in your notebook computer that you can carry with you. If you must use an office computer that you don't always have physical control of, then keep your public and secret key rings on a write-protected removable floppy disk, and don't leave it behind when you leave the office. It wouldn't be a good idea to allow your secret key to reside on a remote timesharing computer, such as a remote dial-in UNIX system. Someone could eavesdrop on your modem line and capture your pass phrase, and then obtain your actual secret key from the remote system. You should only use your secret key on a machine that you have physical control over.

Don't store your pass phrase anywhere on the computer that has your secret key file. Storing both the secret key and the pass phrase on the same computer is as dangerous as keeping your PIN in the same wallet as your Automatic Teller Machine bank card. You don't want somebody to get their hands on your disk containing both the pass phrase and the secret key file. It would be most secure if you just memorize your pass phrase and don't store it anywhere but your brain. If you feel you must write down your pass phrase, keep it well protected, perhaps even more well protected than the secret key file.

And keep backup copies of your secret key ring—remember, you have the only copy of your secret key, and losing it will render useless all the copies of your public key that you have spread throughout the world.

The decentralized non-institutional approach PGP uses to manage public keys has its benefits, but unfortunately this also means we can't rely on a single centralized list of which keys have been compromised. This makes it a bit harder to contain the damage of a secret key compromise. You just have to spread the word and hope everyone hears about it.

If the worst case happens—your secret key and pass phrase are both compromised (hopefully you will find this out somehow)—you will have to issue a *key compromise* certificate. This kind of certificate is used to warn other people to stop using your public key. You can use PGP to create such a certificate by using the `-kd` command. Then you must somehow send this compromise certificate to everyone else on the planet, or at least to all your friends and their friends, et cetera. Their own PGP software will install this key compromise certificate on their public key rings and will automatically prevent them from accidentally using your public key ever again. You can then generate a new secret/public key pair and publish the new public key. You could send out one package containing both your new public key and the key compromise certificate for your old key.

6.8.9 Revoking a public key

Suppose your secret key and your pass phrase are somehow both compromised. You have to get the word out to the rest of the world, so that they will all stop using your public key. To do this, you will have to issue a *key compromise*, or *key revocation* certificate to revoke your public key.

To generate a certificate to revoke your own key, use the `-kd` command:

```
pgp -kd your_userid
```

This certificate bears your signature, made with the same key you are revoking. You should widely disseminate this key revocation certificate as soon as possible. Other people who receive it can add it to their public key rings, and their PGP software then automatically prevents them from accidentally using your old public key ever again. You can then generate a new secret/public key pair and publish the new public key.

You may choose to revoke your key for some other reason than the compromise of a secret key. If so, you may still use the same mechanism to revoke it.

6.8.10 What if you lose your secret key?

Normally, if you want to revoke your own secret key, you can use the `-kd` command to issue a revocation certificate, signed with your own secret key (see "Revoking a Public Key").

But what can you do if you lose your secret key, or if your secret key is destroyed? You can't revoke it yourself, because you must use your own secret key to revoke it, and you don't have it anymore. A future version of PGP will offer a more secure means of revoking keys in these circumstances, allowing trusted introducers to certify that a public key has been revoked. But for now, you will have to get the word out through whatever informal means you can, asking users to *disable* your public key on their own individual public key rings.

Other users may disable your public key on their own public key rings by using the `-kd` command. If a user ID is specified that does not correspond to a secret key on the secret key ring, the `-kd` command will look for that user ID on the public key ring, and mark that public key as disabled. A disabled key may not be used to encrypt any messages, and may not be extracted from the key ring with the `-kx` command. It can still be used to check signatures, but a warning is displayed. And if the user tries to add the same key again to his key ring, it will not work because the disabled key is already on the key ring. These combined features will help curtail the further spread of a disabled key.

If the specified public key is already disabled, the `-kd` command will ask if you want the key reenabled.

7 Advanced topics

Most of the "Advanced Topics" are covered in the "PGP User's Guide, Volume II: Special Topics". But here are a few topics that bear mentioning here.

7.1 Sending ciphertext through E-mail channels: Radix-64 format

Many electronic mail systems only allow messages made of ASCII text, not the 8-bit raw binary data that ciphertext is made of. To get around this problem, PGP supports ASCII radix-64 format for ciphertext messages, similar to the Internet Privacy-Enhanced Mail (PEM) format, as well as the Internet MIME format. This special format represents binary data by using only printable ASCII characters, so it is useful for transmitting binary encrypted data through 7-bit channels or for sending binary encrypted data as normal E-mail text. This format acts as a form of *transport armor*, protecting it against corruption as it travels through intersystem gateways on Internet. PGP also appends a CRC to detect transmission errors.

Radix-64 format converts the plaintext by expanding groups of 3 binary 8-bit bytes into 4 printable ASCII characters, so the file grows by about 33consider that the file probably was compressed more than that by PGP before it was encrypted.

To produce a ciphertext file in ASCII radix-64 format, just add the `a` option when encrypting or signing a message, like so:

```
pgp -esa message.txt her_userid
```

This example produces a ciphertext file called *message.asc* that contains data in a MIME-like ASCII radix-64 format. This file can be easily uploaded into a text editor through 7-bit channels for transmission as normal E-mail on Internet or any other E-mail network.

Decrypting the radix-64 transport-armored message is no different than a normal decrypt. For example:

```
pgp message
```

PGP automatically looks for the ASCII file *message.asc* before it looks for the binary file *message.pgp*. It recognizes that the file is in radix-64 format and converts it back to binary before processing as it normally does, producing as a by-product a *.pgp* ciphertext file in binary form. The final output file is in normal plaintext form, just as it was in the original file *message.txt*.

Most Internet E-mail facilities prohibit sending messages that are more than 50000 or 65000 bytes long. Longer messages must be broken into smaller chunks that can be mailed separately. If your encrypted message is very large, and you requested radix-64 format, PGP automatically breaks it up into chunks that are each small enough to send via E-mail. The chunks are put into files named with extensions *.as1*, *.as2*, *.as3*, etc. The recipient must concatenate these separate files back together in their proper order into one big file before decrypting it. While decrypting, PGP ignores any extraneous text in mail headers that are not enclosed in the radix-64 message blocks.

If you want to send a public key to someone else in radix-64 format, just add the `-a` option while extracting the key from your keyring.

If you forgot to use the `-a` option when you made a ciphertext file or extracted a key, you may still directly convert the binary file into radix-64 format by simply using the `-a` option alone, without any encryption specified. PGP converts it to a *.asc* file.

If you sign a plaintext file without encrypting it, PGP will normally compress it after signing it, rendering it unreadable to the casual human observer. This is a suitable way of storing signed files in archival applications. But if you want to send the signed message as E-mail, and the the original plaintext message is in text (not binary) form, there is a way to send it through an E-mail channel in such a way that the plaintext does not get compressed, and the ASCII armor is applied only to the binary signature certificate, but not to the plaintext message. This makes it possible for the recipient to read the signed message with human eyes, without the aid of PGP. Of course, PGP is still needed to actually check the signature. For further information on this feature, see the explanation of the CLEARSIG parameter in the section "Setting Configuration Parameters" in the Special Topics volume.

Sometimes you may want to send a binary data file through an E-mail channel without encrypting or signing it with PGP. Some people use the Unix uuencode utility for that purpose. PGP can also be used for that purpose, by simply using the `-a` option alone, and it does a better job than the uuencode utility. For further details, see the section on "Using PGP as a Better Uuencode" in the Special Topics volume.

7.2 Environmental variable for path name

PGP uses several special files for its purposes, such as your standard key ring files *pubring.pgp* and *secring.pgp*, the random number seed file *randseed.bin*, the PGP configuration file *config.txt* (or *pgp.ini*, or *.pgprc*), and the foreign language string translation file *language.txt*. These special files can be kept in any directory, by setting the environmental variable *PGPPATH* to the desired pathname. For example, on MSDOS, the shell command:

```
SET PGPPATH=C:\PGP
```

makes PGP assume that your public key ring filename is *C:\PGP\pubring.pgp*. Assuming, of course, that this directory exists. Use your favorite text editor to modify your MSDOS AUTOEXEC.BAT file to automatically set up this variable whenever you start up your system. If PGPPATH remains undefined, these special files are assumed to be in the current directory.

7.3 Setting parameters in the PGP configuration file

PGP has a number of user-settable parameters that can be defined in a special PGP configuration text file called *config.txt*, in the directory pointed to by the shell environmental variable *PGPPATH*. Having a configuration file enables the user to define various flags and parameters for PGP without the burden of having to always define these parameters in the PGP command line.

In the interest of complying with local operating system file naming conventions, for Unix systems this configuration file may also be named *.pgprc*, and on MSDOS systems it may also be named *pgp.ini*.

With these configuration parameters, for example, you can control where PGP stores its temporary scratch files, or you can select what foreign language PGP will use to display its diagnostics messages and user prompts, or you can adjust PGP's level of skepticism in determining a key's validity based on the number of certifying signatures it has.

For more details on setting these configuration parameters, see the appropriate section of the PGP User's Guide, Special Topics volume.

8 Vulnerabilities

No data security system is impenetrable. PGP can be circumvented in a variety of ways. Potential vulnerabilities you should be aware of include compromising your pass phrase or secret key, public key tampering, files that you deleted but are still somewhere on the disk, viruses and Trojan horses, breaches in your physical security, electromagnetic emissions, exposure on multi-user systems, traffic analysis, and perhaps even direct cryptanalysis.

For a detailed discussion of these issues, see the "Vulnerabilities" section in the PGP User's Guide, Special Topics volume.

9 Beware of snake oil

When examining a cryptographic software package, the question always remains, why should you trust this product? Even if you examined the source code yourself, not everyone has the cryptographic experience to judge the security. Even if you are an experienced cryptographer, subtle weaknesses in the algorithms could still elude you.

When I was in college in the early seventies, I devised what I believed was a brilliant encryption scheme. A simple pseudorandom number stream was added to the plaintext stream to create ciphertext. This would seemingly thwart any frequency analysis of the ciphertext, and would be uncrackable even to the most resourceful Government intelligence agencies. I felt so smug about my achievement. So cock-sure.

Years later, I discovered this same scheme in several introductory cryptography texts and tutorial papers. How nice. Other cryptographers had thought of the same scheme. Unfortunately, the scheme was presented as a simple homework assignment on how to use elementary cryptanalytic techniques to trivially crack it. So much for my brilliant scheme.

From this humbling experience I learned how easy it is to fall into a false sense of security when devising an encryption algorithm. Most people don't realize how fiendishly difficult it is to devise an encryption algorithm that can withstand a prolonged and determined attack by a resourceful opponent. Many mainstream software engineers have developed equally naive encryption schemes (often even the very same encryption scheme), and some of them have been incorporated into commercial encryption software packages and sold for good money to thousands of unsuspecting users.

This is like selling automotive seat belts that look good and feel good, but snap open in even the slowest crash test. Depending on them may be worse than not wearing seat belts at all. No one suspects they are bad until a real crash. Depending on weak cryptographic software may cause you to unknowingly place sensitive information at risk. You might not otherwise have done so if you had no cryptographic software at all. Perhaps you may never even discover your data has been compromised.

Sometimes commercial packages use the Federal Data Encryption Standard (DES), a fairly good conventional algorithm recommended by the Government for commercial use (but not for classified information, oddly enough– hmmm). There are several *modes of operation* the DES can use, some of them better than others. The Government specifically recommends not using the weakest simplest mode for messages, the Electronic Codebook (ECB) mode. But they do recommend the stronger and more complex Cipher Feedback (CFB) or Cipher Block Chaining (CBC) modes.

Unfortunately, most of the commercial encryption packages I've looked at use ECB mode. When I've talked to the authors of a number of these implementations, they say they've never heard of CBC or CFB modes, and didn't know anything about the weaknesses of ECB mode. The very fact that they haven't even learned enough cryptography to know these elementary concepts is not reassuring. And they sometimes manage their DES keys in inappropriate or insecure ways. Also, these same software packages often include a second faster encryption algorithm that can be used instead of the slower DES. The author of the package often thinks his proprietary faster algorithm is as secure as the DES, but after questioning him I usually discover that it's just a variation of my own brilliant scheme from college days. Or maybe he won't even reveal how his proprietary encryption scheme works, but assures me it's a brilliant scheme and I should trust it. I'm sure he believes that his algorithm is brilliant, but how can I know that without seeing it?

In all fairness I must point out that in most cases these terribly weak products do not come from companies that specialize in cryptographic technology.

Even the really good software packages, that use the DES in the correct modes of operation, still have problems. Standard DES uses a 56-bit key, which is too small by today's standards, and may now be easily broken by exhaustive key searches on special high-speed machines. The DES has reached the end of its useful life, and so has any software package that relies on it.

There is a company called AccessData (87 East 600 South, Orem, Utah 84058, phone 1-800-658-5199) that sells a package for $185 that cracks the built-in encryption schemes used by WordPerfect, Lotus 1-2-3, MS Excel, Symphony, Quattro Pro, Paradox, and MS Word 2.0. It doesn't simply guess passwords—it does real cryptanalysis. Some people buy it when they forget their password for their own files. Law enforcement agencies buy it too, so they can read files they seize. I talked to Eric Thompson, the author, and he said his program only takes a split second to crack them, but he put in some delay loops to slow it down so it doesn't look so easy to the customer. He also told me that the password encryption feature of PKZIP files can often be easily broken, and that his law enforcement customers already have that service regularly provided to them from another vendor.

In some ways, cryptography is like pharmaceuticals. Its integrity may be absolutely crucial. Bad penicillin looks the same as good penicillin. You can tell if your spreadsheet software is wrong, but how do you tell if your cryptography package is weak? The ciphertext produced by a weak encryption algorithm looks as good as ciphertext produced by a strong encryption algorithm. There's a lot of snake oil out there. A lot of quack cures. Unlike the patent medicine hucksters of old, these software implementors usually don't even know their stuff is snake oil. They may be good software engineers, but they usually haven't even read any of the academic literature in cryptography. But they think they can write good cryptographic software. And why not? After all, it seems intuitively easy to do so. And their software seems to work okay.

Anyone who thinks they have devised an unbreakable encryption scheme either is an incredibly rare genius or is naive and inexperienced. Unfortunately, I sometimes have to deal with would-be cryptographers who want to make "improvements" to PGP by adding encryption algorithms of their own design.

I remember a conversation with Brian Snow, a highly placed senior cryptographer with the NSA. He said he would never trust an encryption algorithm designed by someone who had not *earned their bones* by first spending a lot of time cracking codes. That did make a lot of sense. I observed that practically no one in the commercial world of cryptography qualified under this criterion. *"Yes"*, he said with a self assured smile, *"And that makes our job at NSA so much easier."* A chilling thought. I didn't qualify either.

The Government has peddled snake oil too. After World War II, the US sold German Enigma ciphering machines to third world governments. But they didn't tell them that the Allies cracked the Enigma code during the war, a fact that remained classified for many years. Even today many UNIX systems worldwide use the Enigma cipher for file encryption, in part because the Government has created legal obstacles against using better algorithms. They even tried to prevent the initial publication of the RSA algorithm in 1977. And they have squashed essentially all commercial efforts to develop effective secure telephones for the general public.

The principal job of the US Government's National Security Agency is to gather intelligence, principally by covertly tapping into people's private communications (see James Bamford's book, *The Puzzle Palace*). The NSA has amassed considerable skill and resources for cracking codes. When people can't get good cryptography to protect themselves, it makes NSA's job much easier. NSA also has the responsibility of approving and recommending encryption algorithms. Some critics charge that this is a conflict of interest, like putting the fox in charge of guarding the hen house. NSA has been pushing a conventional encryption algorithm that they designed, and they won't tell anybody how it works because that's classified. They want others to trust it and use it. But any cryptographer can tell you that a well-designed encryption algorithm does not have to be classified to remain secure. Only the keys should need protection. How does anyone else really know if NSA's classified algorithm is secure? It's not that hard for NSA to design an encryption algorithm that only they can crack, if no one else can review the algorithm. Are they deliberately selling snake oil?

There are three main factors that have undermined the quality of commercial cryptographic software in the US. The first is the virtually universal lack of competence of implementors of commercial encryption software (although this is starting to change since the publication of PGP). Every software engineer fancies himself a cryptographer, which has led to the proliferation of really bad crypto software. The second is the NSA deliberately and systematically suppressing all the good commercial encryption technology, by legal intimidation and economic pressure. Part of this pressure is brought to bear by stringent export controls on encryption software which, by the economics of software marketing, has the net effect of suppressing domestic encryption software. The other principle method of suppression comes from the granting all the software patents for all the public key encryption algorithms to a single company, affording a single choke point to suppress the spread of this technology. The net effect of all this is that before PGP was published, there was almost no highly secure general purpose encryption software available in the US.

I'm not as certain about the security of PGP as I once was about my brilliant encryption software from college. If I were, that would be a bad sign. But I'm pretty sure that PGP does not contain any glaring weaknesses (although it may contain bugs). The crypto algorithms were developed by people at high levels of civilian cryptographic academia, and have been individually subject to extensive peer review. Source code is available to facilitate peer review of PGP and to help dispel the fears of some users. It's reasonably well researched, and has been years in the making. And I don't work for the NSA. I hope it doesn't require too large a "leap of faith" to trust the security of PGP.

10 Notice to Macintosh users

PGP was originally developed for MSDOS and Unix machines. There is also an Apple Macintosh version of PGP. This manual is written for the MSDOS/Unix versions of PGP, which use a command-line interface for all the PGP functions. On the Mac, all the PGP functions are accessed through pull-down menus and dialog boxes. There is also on-line help on the Mac for how to use MacPGP, and there should be some Mac-specific user documentation included in the MacPGP release package, in addition to this manual.

Almost all good Mac software applications are written from scratch for the Mac, not simply ported there from other operating systems. Unfortunately, the current Mac version of PGP was not designed for the Mac from scratch. It was ported from the MSDOS/Unix version to the Mac by Zbigniew Fiedorwicz. Since the MSDOS/Unix version of PGP was not designed for a GUI (graphical user interface), porting to the Mac was not an easy task, and many bugs still remain. An all-new version of PGP is under development, designed for easy adaptation to a GUI. A new Mac version will be developed from this new PGP source code. It will be more Mac-like, and more reliable. Despite the bugs in the current version of MacPGP, it is important to note that if Zbigniew had waited for this all-new version of PGP to be developed before he ported PGP to the Mac, the world would have been deprived of a Mac version of PGP for far too long.

11 PGP quick reference

Here's a quick summary of PGP commands.

To encrypt a plaintext file with the recipient's public key:

```
pgp -e textfile her_userid
```

To sign a plaintext file with your secret key:

```
pgp -s textfile [-u your_userid]
```

To sign a plaintext ASCII text file with your secret key, producing a signed plaintext message suitable for sending via E-mail:

```
pgp -sta textfile [-u your_userid]
```

To sign a plaintext file with your secret key, and then encrypt it with the recipient's public key:

```
pgp -es textfile her_userid [-u your_userid]
```

To encrypt a plaintext file with just conventional cryptography, type:

```
pgp -c textfile
```

To decrypt an encrypted file, or to check the signature integrity of a signed file:

```
pgp ciphertextfile [-o plaintextfile]
```

To encrypt a message for any number of multiple recipients:

```
pgp -e textfile userid1 userid2 userid3
```

11.1 Key management commands:

To generate your own unique public/secret key pair:

```
pgp -kg
```

To add a public or secret key file's contents to your public or secret key ring:

```
pgp -ka keyfile [keyring]
```

To extract (copy) a key from your public or secret key ring:

```
pgp -kx userid keyfile [keyring]
```

or:

```
pgp -kxa userid keyfile [keyring]
```

To view the contents of your public key ring:

```
pgp -kv[v] [userid] [keyring]
```

To view the *fingerprint* of a public key, to help verify it over the telephone with its owner:

```
pgp -kvc [userid] [keyring]
```

To view the contents and check the certifying signatures of your public key ring:

```
pgp -kc [userid] [keyring]
```

To edit the userid or pass phrase for your secret key:

```
pgp -ke userid [keyring]
```

To edit the trust parameters for a public key:

```
pgp -ke userid [keyring]
```

To remove a key or just a userid from your public key ring:

```
pgp -kr userid [keyring]
```

To sign and certify someone else's public key on your public key ring:

```
pgp -ks her_userid [-u your_userid] [keyring]
```

To remove selected signatures from a userid on a keyring:

```
pgp -krs userid [keyring]
```

To permanently revoke your own key, issuing a key compromise certificate:

```
pgp -kd your_userid
```

To disable or reenable a public key on your own public key ring:

```
pgp -kd userid
```

11.2 Esoteric commands:

To decrypt a message and leave the signature on it intact:

```
pgp -d ciphertextfile
```

To create a signature certificate that is detached from the document:

```
pgp -sb textfile [-u your_userid]
```

To detach a signature certificate from a signed message:

```
pgp -b ciphertextfile
```

11.3 Command options that can be used in combination with other command options (sometimes even spelling interesting words!):

To produce a ciphertext file in ASCII radix-64 format, just add the `-a` option when encrypting or signing a message or extracting a key:

```
pgp -sea textfile her_userid
```

or:

```
pgp -kxa userid keyfile [keyring]
```

To wipe out the plaintext file after producing the ciphertext file, just add the `-w` (wipe) option when encrypting or signing a message:

```
pgp -sew message.txt her_userid
```

To specify that a plaintext file contains ASCII text, not binary, and should be converted to recipient's local text line conventions, add the `-t` (text) option to other options:

```
pgp -seat message.txt her_userid
```

To view the decrypted plaintext output on your screen (like the UNIX-style *more* command), without writing it to a file, use the `-m` (more) option while decrypting:

```
pgp -m ciphertextfile
```

To specify that the recipient's decrypted plaintext will be shown ONLY on her screen and cannot be saved to disk, add the `-m` option:

```
pgp -steam message.txt her_userid
```

To recover the original plaintext filename while decrypting, add the `-p` option:

```
pgp -p ciphertextfile
```

To use a UNIX-style filter mode, reading from standard input and writing to standard output, add the `-f` option:

```
pgp -feast her_userid <inputfile >outputfile
```

12 Legal issues

For detailed information on PGP(tm) licensing, distribution, copyrights, patents, trademarks, liability limitations, and export controls, see the "Legal Issues" section in the "PGP User's Guide, Volume II: Special Topics".

PGP uses a public key algorithm claimed by U.S. patent #4,405,829. The exclusive licensing rights to this patent are held by a company called Public Key Partners (PKP), and you may be infringing the patent if you use PGP in the USA without a license. These issues are detailed in the Volume II manual, and in the RSAREF license that comes with the freeware version of PGP. PKP has licensed others to practice the patent, including a company known as ViaCrypt, in Phoenix, Arizona. ViaCrypt sells a fully licensed version of PGP. ViaCrypt may be reached at 602-944-0773.

PGP is "guerrilla" freeware, and I don't mind if you distribute it widely. Just don't ask me to send you a copy. Instead, you can look for it yourself on many BBS systems and a number of Internet FTP sites. But before you distribute PGP, it is essential that you understand the U.S. export controls on encryption software.

13 Acknowledgments

Formidable obstacles and powerful forces have been arrayed to stop PGP. Dedicated people are helping to overcome these obstacles. PGP has achieved notoriety as "underground software", and bringing PGP "above ground" as fully licensed freeware has required patience and persistence. I'd especially like to thank Hal Abelson, Jeff Schiller, Brian LaMacchia, and Derek Atkins at MIT for their determined efforts. I'd also like to thank Jim Bruce and David Litster in the MIT administration and Bob Prior and Terry Ehling at the MIT Press. And I'd like to thank my entire legal defense team, whose job is not over yet. I used to tell a lot of lawyer jokes, before I encountered so many positive examples of lawyers in my legal defense team, most of whom work pro bono.

The development of PGP has turned into a remarkable social phenomenon, whose unique political appeal has inspired the collective efforts of an ever-growing number of volunteer programmers. Remember that children's story called "Stone Soup"?

I'd like to thank the following people for their contributions to the creation of Pretty Good Privacy. Although I was the author of PGP version 1.0, major parts of later versions of PGP were implemented by an international collaborative effort involving a large number of contributors, under my design guidance.

Branko Lankester, Hal Finney and Peter Gutmann all contributed a huge amount of time in adding features for PGP 2.0, and ported it to UNIX variants.

Hugh Kennedy ported it to VAX/VMS, Lutz Frank ported it to the Atari ST, and Cor Bosman and Colin Plumb ported it to the Commodore Amiga.

Translation of PGP into foreign languages was done by Jean-loup Gailly in France, Armando Ramos in Spain, Felipe Rodriquez Svensson and Branko Lankester in The Netherlands, Miguel Angel Gallardo in Spain, Hugh Kennedy and Lutz Frank in Germany, David Vincenzetti in Italy, Harry Bush and Maris Gabalins in Latvia, Zygimantas Cepaitis in Lithuania, Peter Suchkow and Andrew Chernov in Russia, and Alexander Smishlajev in Esperantujo. Peter Gutmann offered to translate it into New Zealand English, but we finally decided PGP could get by with US English.

Jean-loup Gailly, Mark Adler, and Richard B. Wales published the ZIP compression code, and granted permission for inclusion into PGP. The MD5 routines were developed and placed in the public domain by Ron Rivest. The IDEA(tm) cipher was developed by Xuejia Lai and James L. Massey at ETH in Zurich, and is used in PGP with permission from Ascom-Tech AG.

Charlie Merritt originally taught me how to do decent multiprecision arithmetic for public key cryptography, and Jimmy Upton contributed a faster multiply/modulo algorithm. Thad Smith implemented an even faster modmult algorithm. Zhahai Stewart contributed a lot of useful ideas on PGP file formats and other stuff, including having more than one user ID for a key. I heard the idea of introducers from Whit Diffie. Kelly Goen did most of the work for the initial electronic publication of PGP 1.0.

Various contributions of coding effort also came from Colin Plumb, Derek Atkins, and Castor Fu. Other contributions of effort, coding or otherwise, have come from Hugh Miller, Eric Hughes, Tim May, Stephan Neuhaus, and too many others for me to remember right now. Zbigniew Fiedorwicz did the first Macintosh port.

Since the release of PGP 2.0, many other programmers have sent in patches and bug fixes and porting adjustments for other computers. There are too many to individually thank here.

Just as in the "Stone Soup" story, it is getting harder to peer through the thick soup to see the stone at the bottom of the pot that I dropped in to start it all off.

14 About the Author

Philip Zimmermann is a software engineer consultant with 19 years experience, specializing in embedded real-time systems, cryptography, authentication, and data communications. Experience includes design and implementation of authentication systems for financial information networks, network data security, key management protocols, embedded real-time multitasking executives, operating systems, and local area networks.

Custom versions of cryptography and authentication products and public key implementations such as the NIST DSS are available from Zimmermann, as well as custom product development services. His consulting firm's address is:


```
Boulder Software Engineering
3021 Eleventh Street
Boulder, Colorado 80304  USA
Phone: 303-541-0140 (10:00am - 7:00pm Mountain Time)
Fax: arrange by phone
Internet:  prz@acm.org
```