

Phil's Pretty Good Software

presents

===
PGP
===

Pretty Good Privacy
RSA Public Key Cryptography for the Masses

Version 1.0 - 5 Jun 91

PGP User's Guide

(c) Copyright 1990 Philip Zimmermann
Software and Documentation Written by Philip Zimmermann

For information on PGP licensing, distribution, copyrights, patents,
trademarks, liability limitations, and export controls, see the "Legal
Issues" section later in this document.

Contents

=====

Quick Overview

How it Works

Installing PGP

How to Use PGP

To See a Usage Summary

Encrypting a Message

Signing a Message

Signing and then Encrypting

Using Just Conventional Encryption

Decrypting and Checking Signatures

Managing Keys

RSA Key Generation

Adding a Key to Your Key Ring

Removing a Key from Your Key Ring

Viewing the Contents of Your Key Ring

Signed Public Key Certificates

Advanced Topics

Separating Signatures from Messages

Sending Ciphertext Through E-mail Channels: Uuencode Format

Leaving No Traces of Plaintext on the Disk

Environmental Variable for Path Name

A Peek Under the Hood

Random Numbers

PGP's Conventional Encryption Algorithm

Data Compression

Vulnerabilities

Compromised Pass Phrase and Secret Key

Public Key Tampering

"Not Quite Deleted" Files

Viruses and Trojan Horses

Physical Security Breach

Tempest Attacks

Traffic Analysis

Cryptanalysis

Trusting Snake Oil

Why Do You Need PGP?

PGP Quick Reference

Legal Issues

Trademarks, Copyrights, and Warranties

Patent Rights on the Algorithms

Licensing and Distribution

Export Controls

Acknowledgements

Recommended Readings

About the Author

Quick Overview

=====

Pretty Good(tm) Privacy (PGP), from Phil's Pretty Good Software, is a high security cryptographic software application for MSDOS. PGP allows people to exchange files or messages with privacy, authentication, and convenience. Privacy means only those intended to receive a message can read it. Authentication means 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 an powerful new technology called "public key" cryptography.

PGP combines the convenience of the Rivest-Shamir-Adleman (RSA) public key cryptosystem with the speed of fast conventional cryptographic algorithms, fast message digest algorithms, data compression, and sophisticated key management. And PGP performs the RSA functions faster than most other software implementations. PGP is RSA public key cryptography for the masses.

PGP does not provide any built-in modem communications capability. You must use a separate product such as TELIX or PROCOMM for that.

This document only explains how to use PGP without explaining the underlying technology details and data structures and cryptographic algorithms. It would help if you were already familiar with the concept of cryptography in general and RSA public key cryptography in particular. Nonetheless, here are a few introductory remarks about public key cryptography.

How it Works

In conventional cryptosystems, such as the Federal Data Encryption Standard (DES), a single key is used for both encryption and decryption. This means that keys must be initially transmitted via secure channels so that both parties can know them 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 complimentary keys, a publicly revealed key and a secret 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 origin 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 RSA public key encryption algorithm is so slow, encryption is better accomplished by using a high-quality fast conventional 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 RSA public key is used to encipher this temporary random conventional key. This RSA-enciphered conventional "session" key is sent along with the enciphered text (called "ciphertext") to the recipient. The recipient uses her own RSA secret key to recover this temporary session key, and then uses that key to run the fast conventional algorithm to decipher the large ciphertext message.

RSA keys are kept in "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. A key file, or "key ring" contains one or more of these key certificates. Public key certificates contain the public key material, while secret key certificates contain the secret key material. Public key rings contain only public keys, and secret key rings contain just secret keys. Secret keys are cryptographically protected by their own password.

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 RSA key). When this key ID is displayed, only the lower 24 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 RSA secret key to form a signature. The message digest algorithm used here is the MD4 Message Digest Algorithm, placed in the public domain by RSA Data Security, Inc.

Documents are signed by prefixing them with signature certificates, which contain the key ID of the key that was used to sign it, an RSA-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. An individual key file is no different from a key ring that contains only one key.

Installing PGP

=====

To install PGP on your MSDOS system, you just have to copy it into a suitable directory on your hard disk (like C:\PGP), and use the shareware PKUNZIP utility to decompress it from the compressed archive release file. For best results, you will 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.

For further details on installation, see the separate PGP Installation Guide, in the MSDOS file SETUP.DOC included with this release. It fully describes how to set up the PGP directory and how to use PKUNZIP to install it, and also describes how to detect virus infections of PGP releases.

How to Use PGP

=====

To See a Usage Summary

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

```
pgp
```

This will display a usage summary for the most essential commands only. The commands described in the Advanced Topics section are not displayed.

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.ctx. A specific example is:

```
pgp -e letter.txt Alice_S
```

This will search your public key ring file "keyring.pub" for any public key certificates that contain the string "Alice S" anywhere in the user ID field. The search is not case-sensitive. Note that underlines get changed to spaces. That's because you can't use real spaces in the user ID on the command line. If it finds a matching public key, it uses it to encrypt the plaintext file "letter.txt", producing a ciphertext file called "letter.ctx".

PGP will attempt 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.

Signing a Message

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

```
pgp -s textfile your_userid
```

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

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

This will search your secret key ring file "keyring.sec" for any secret key certificates that contain the string "Bob" anywhere in the user ID field. The search is not case-sensitive. Note that underlines get changed to spaces. If it finds a matching secret key, it uses it to sign the plaintext file "letter.txt", producing a signature file called "letter.ctx".

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 your_userid
```

This example produces a nested ciphertext file called textfile.ctx. 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 keyring via her_userid. If you leave these user ID fields off the command line, you will be prompted for them.

Note that PGP will attempt to compress the plaintext before encrypting it.

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.ctx, 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. Note that PGP will attempt to compress the plaintext before encrypting it.

Decrypting and Checking Signatures

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

```
pgp ciphertextfile [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 ".ctx". The optional plaintext 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 will then use 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 will recognize this and will prompt you for the pass phrase to decrypt it.

Managing Keys

=====

RSA Key Generation

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

```
pgp -k
```

The software will prompt you for a filename for the pair of keys, which will be written to filename.pub and filename.sec. It will also give you a menu of recommended key sizes (casual grade, commercial grade, or military grade) and prompt you for what size key you want, up to around a thousand bits.

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. Also, if you put your last name first it would facilitate producing lists of public keys sorted by user ID. e.g.: "Smith, Robert M."

It will also ask for a "pass phrase" to protect your RSA 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 RSA 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. If you don't want a pass phrase (ill-advised), just press return (or enter) at the pass phrase prompt.

The RSA key pair is derived from large truly random numbers derived from measuring the intervals between your keystrokes with a fast timer.

Note that RSA key generation is a VERY lengthy process. It may take a few seconds for a small key on a fast processor, or many minutes for a large key, or even hours for a large key on an old IBM PC/XT.

The public keyfile 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 you secret key to anyone else. For the same reason, don't make keys for your friends. Everyone should make their own key pair.

Adding a Key to Your Key Ring

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 -a keyfile [keyring]
```

The keyfile extension defaults to ".pub", implying a public key. The optional keyring file name is assumed to be literally "keyring.pub" or "keyring.sec", depending on whether the keyfile name had a ".pub" or ".sec" extension. You may specify a different key ring file name. The default key ring extension is ".pub".

If the key is already on your keyring, PGP will not add it again. All of the keys in the keyfile will be added to the keyring. Note that the keyfile should only contain one key, because PGP only checks the first key in the keyfile for duplicates on the keyring.

Removing a Key from Your Key Ring

To remove a key from your public key ring:

```
pgp -r userid [keyring]
```

This will search for the specified user ID in your keyring, and will remove 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 "keyring.pub". It can be omitted, or you can specify "keyring.sec" if you want to remove a secret key. You may specify a different key ring file name. The default key ring extension is ".pub".

Viewing the Contents of Your Key Ring

To view the contents of your public key ring:

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

This will list 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 will be listed. The optional keyring file name is assumed to be literally "keyring.pub". It can be omitted, or you can specify "keyring.sec" 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 ".pub".

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.sec
```

Using this approach requires that the key ring name be fully qualified with the extension of ".pub" or ".sec", because if you don't specify a file extension, ".ctx" is assumed.

Signed Public Key Certificates

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. 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 Charles 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 Charles instead of Alice's public key. All looks normal because this bogus key has Alice's user ID. Now Charles 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 in the usual way, 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 imposter 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 "guerilla 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 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.

You should add a new public key to your key ring 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. The user ID should have the full name of the key's owner, not just her first name.

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. 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 may want to keep your own public key around with signatures from a variety of "introducers" in the hopes that most people will trust at least one of the introducers to vouch for your own public key's integrity.

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 public keys that are already on your own public key ring. Keep a trusted backup copy of your public key ring on write-protected media, and check it once in a while against the working copy. Back up your secret key ring, too.

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.

Advanced Topics

=====

Separating Signatures from Messages

Normally, signature certificates are prepended (attached) to the text they sign. This makes it convenient in simple cases to check signatures. It is desirable in some circumstances to have signature certificates stored separately from the messages they sign. It is possible to generate signature certificates that are detached from the text they sign. To do this, combine the 'b' (break) option with the 's' (sign) option. For example:

```
pgp -sb letter.txt your_userid
```

This example produces an isolated signature certificate in a file called "letter.ctx". The contents of letter.txt are not appended to the signature certificate.

After creating the signature certificate file (letter.ctx in the above example), send it along with the original text file to the recipient. The recipient must have both files to check the signature integrity. When the recipient attempts to process the signature file, PGP will notice that there is no text in the same file with the signature and will prompt the user for the filename of the text. Only then will PGP be able to properly check the signature integrity. If the recipient knows in advance that the signature is detached from the text file, she can specify both filenames on the command line:

```
pgp letter.ctx letter.txt  
or: pgp letter letter.txt
```

PGP will not have to prompt for the text file name in this case.

A detached signature certificate is useful if you want to keep the signature certificate in a separate certificate log. A detached signature of an executable program is also useful for detecting a subsequent virus infection. It is also useful if more than one party must sign a document such as a legal contract, without nesting signatures. Each person's signature is independent.

Sending Ciphertext Through E-mail Channels: Uuencode Format

For all you Unix fans out there: PGP supports uuencode format for ciphertext messages. 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.

Uencode format converts the plaintext by expanding groups of 3 binary 8-bit bytes into 4 printable ASCII characters, so the file will grow by about 35%. But this expansion isn't so bad when you consider that the file probably was compressed more than that by PGP before it was encrypted.

To produce a ciphertext file in uencode format, just add the "u" option when encrypting or signing a message, like so:

```
pgp -esu message.txt her_userid your_userid
```

This example produces a ciphertext file called "message.ctx" that contains data in Unix uencode 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 uencode-formatted message is no different than a normal decrypt. For example:

```
pgp message
```

PGP will automatically recognize that the file "message.ctx" is in uencode format and will udecode it before processing as it normally does. The output file will be in normal plaintext form, just as it was in the original file "message.txt".

During decryption, after PGP udecodes the ".ctx" file, it leaves it in binary ciphertext form. In other words, the ".ctx" file is no longer in uencode format when PGP is done processing it. PGP produces a decrypted plaintext file, and also produces as a by-product a udecoded ciphertext file in binary form.

If you want to send a public key or key ring to someone else in uencode format, sign it with the "-su" options to create a ".ctx" file with the signed key in uencode format.

Leaving No Traces of Plaintext on the Disk

After PGP makes a ciphertext file for you, you can have PGP automatically overwrite the plaintext file and delete it, leaving no trace of plaintext on the disk so that no one can recover it later using a disk block scanning utility. This is useful if the plaintext file contains sensitive information that you don't want to keep around.

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

```
pgp -esw message.txt her_userid your_userid
```

This example will create the ciphertext file "message.ctx", and the plaintext file "message.txt" will be destroyed beyond recovery.

Obviously, you should be careful with this option. Also note that this will not wipe out any fragments of plaintext that your word processor might have created on the disk while you were editing the message before running PGP. Most word processors create backup files, scratch files, or both.

Environmental Variable for Path Name

The standard key ring files "keyring.pub" and "keyring.sec" can be kept in any directory, by setting the environmental variable "PGPPATH" to the desired pathname. For example, the MSDOS shell command:

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

will make PGP assume the key ring filenames "C:\PGP\keyring.pub" and "C:\PGP\keyring.sec". Assuming, of course, 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.

A Peek Under the Hood

=====

Let's take a look at a few internal features of PGP.

Random Numbers

PGP uses a cryptographically strong pseudorandom number generator for creating temporary conventional session keys. The seed file for this is called "randseed.pgp". It too can be kept in whatever directory is indicated by the PGPPATH environmental variable. If this random seed file does not exist, it will be automatically created and seeded with truly random numbers derived from timing your keystroke latencies.

This generator reseeds the disk file each time it is used with new key material partially derived with the time of day and other truly random sources. It uses the conventional encryption algorithm as an engine for the random number generator. The seed file contains both random seed material and random key material to key the conventional encryption engine for the random generator.

If you are a security fanatic and distrust any algorithmically derived random number source however strong, you can defeat this feature by creating an empty file named "randseed.pgp". This file must be empty or nearly empty to turn off this pseudorandom generator. In that case, every encryption session key will require a bothersome request to the user to type some text in at the keyboard to measure the keystroke intervals with a high speed timer. It would be more convenient and not that unsafe to use the strong pseudorandom generator.

PGP's Conventional Encryption Algorithm

PGP does not use the DES as its conventional single-key algorithm to encrypt messages. Instead it uses a custom conventional single-key block encryption algorithm. It "bootstraps" into this faster algorithm by using RSA to encipher the conventional session key.

For the cryptographically curious, PGP's conventional block cipher has a 256-byte block size for the plaintext and the ciphertext. It also uses a key size of up to 256 bytes. Permutation and substitution are used on all the bits throughout the block in each round, rapidly building intersymbol dependence between the ciphertext and both the plaintext and the key. It can be configured to run from 1 to 8 rounds. It compares well with software implementations of the DES in speed. Like the DES, it can be used in cipher feedback (CFB) and cipher block chaining (CBC) modes. PGP uses it in CFB mode.

PGP's conventional encryption algorithm is based in large part on cryptographer Charles Merritt's algorithms. Merritt's algorithm does

have something of a track record; derivatives of it have been used for secure U.S. military communications. Merritt's original designs were refined by Zhahai Stewart and myself to improve security and to improve performance in a portable C implementation. The algorithm has not yet (in 1991) been through a formal security review and has had only limited peer review. It has been carefully scrutinized for weaknesses. A full discussion of the architecture is beyond the scope of this preliminary draft of this document. Interested parties can get design details from me or from the published source code.

Data Compression

PGP normally compresses the plaintext before encrypting it. It's too late to compress it after it has been encrypted; encrypted data is incompressible. Data compression saves modem transmission time and disk space and more importantly strengthens cryptographic security. Most cryptanalysis techniques exploit redundancies found in the plaintext to crack the cipher. Data compression reduces this redundancy in the plaintext, thereby greatly enhancing resistance to cryptanalysis. It seems to take longer to compress the plaintext than to encrypt it, but from a security point of view it seems worth the extra time, at least in my cautious opinion.

Files that are too short to compress or just don't compress well are not compressed by PGP.

If you prefer, you can use PKZIP to compress the plaintext before encrypting it. PKZIP is a widely-available and effective MSDOS shareware compression utility from PKWare, Inc (9025 N Deerwood Dr, Brown Deer, WI 53223). Unlike PGP's built-in compression algorithm, PKZIP has the nice feature of compressing multiple files into a single compressed file, which is reconstituted again into separate files when decompressed. PKZIP also compresses faster than the internal compression algorithm used in PGP. PGP will not try to compress a plaintext file that has already been compressed by PKZIP. After decrypting, the recipient can decompress the plaintext with PKUNZIP. If the decrypted plaintext is a PKZIP compressed file, PGP will automatically recognize this and will advise the recipient that the decrypted plaintext appears to be a PKZIP file.

For the technically curious readers, PGP uses the public domain LZHuf compression routines written in Japan by Haruyasu Yoshizaki, based on the original LZSS compression routines by Haruhiko Okumura. The adaptive Huffman algorithm was added by Yoshizaki to increase speed and compression, and he used the LZHuf routines to develop the LHarc archiver. Allan Hoeltje added a run-length encoding layer for better speed. This compression software was selected for PGP because of its public domain portable C source code availability, and because it has a good compression ratio.

Vulnerabilities

=====

No data security system is impenetrable. PGP can be circumvented in a variety of ways. In any data security system, you have to ask yourself if the information you are trying to protect is valuable enough to your attacker that the cost of the attack is less than the value of the information. This should lead you to protecting yourself from the cheapest attacks, while not worrying about the more expensive attacks. Some of the discussion that follows may seem unduly paranoid, but such an attitude is appropriate for a reasonable discussion of vulnerability issues.

Compromised Pass Phrase and Secret Key

Probably the simplest attack is if you leave your pass phrase for your secret key written down somewhere. If someone gets it and gets your secret key file, they can read your messages and make signatures in your name. Also, don't use obvious passwords that can be easily guessed, such as the names of your kids or spouse.

Public Key Tampering

Another vulnerability exists if public keys are tampered with. This attack and appropriate hygienic countermeasures are detailed in this document in the section "Signed Public Key Certificates". When you use someone's public key, make certain it has not been tampered with. Also make sure no one else can tamper with your own public key ring.

"Not Quite Deleted" Files

Another potential security problem is caused by how most operating systems delete files. When you encrypt a file and then delete the original plaintext file, the operating system doesn't actually physically erase the data. It merely marks those disk blocks as deleted, allowing the space to be reused later. It's sort of like discarding sensitive paper documents in the paper recycling bin instead of the paper shredder. The disk blocks still contain the original sensitive data you wanted to erase, and will probably eventually be overwritten by new data at some point in the future. If an attacker reads these deleted disk blocks soon after they have been deallocated, he could recover your plaintext.

In fact this could even happen accidentally, if for some reason something went wrong with the disk and some files were accidentally deleted or corrupted. A disk recovery program may be run to recover the damaged files, but this often means some previously deleted files are resurrected along with everything else. Your confidential files that you thought were gone forever could then reappear and be

inspected by whomever is attempting to recover your damaged disk. Even while you are creating the original message with a word processor or text editor, the editor may be creating multiple temporary copies of your text on the disk, just because of its internal workings. These temporary copies of your text are deleted by the word processor when it's done, but these sensitive fragments are still on your disk somewhere. The only way to prevent all this from happening is to somehow cause the sensitive deleted plaintext files to be overwritten. There are disk utilities available that can overwrite all of the unused blocks on a disk. For example, I think the Norton Utilities for MSDOS can do this.

Viruses and Trojan Horses

Another attack could involve a specially-tailored hostile computer virus or worm that might infect PGP or your operating system. This hypothetical virus could be designed to capture your pass phrase or secret key or deciphered messages, and covertly write the captured information to a file or send it through a network to the virus's owner. Or it might alter PGP's behavior so that signatures are not properly checked. This attack is cheaper than cryptanalytic attacks.

Defending against this falls under the category of defending against viral infection generally. There are some moderately capable anti-viral products commercially available, and there are hygienic procedures to follow that can greatly reduce the chances of viral infection. A complete treatment of anti-viral and anti-worm countermeasures is beyond the scope of this document. PGP has no defenses against viruses, and assumes your own personal computer is a trustworthy execution environment. If such a virus or worm actually appeared, hopefully word would soon get around warning everyone.

Another similar attack involves someone creating a clever imitation of PGP that behaves like PGP in most respects, but doesn't work the way it's supposed to. For example, it might be deliberately crippled to not check signatures properly, allowing bogus key certificates to be accepted. This "Trojan horse" version of PGP is not hard for an attacker to create, because PGP source code is widely available, so anyone could modify the source code and produce a lobotomized zombie imitation PGP that looks real but does the bidding of its diabolical master. This Trojan horse version of PGP could then be widely circulated, claiming to be from me. How insidious.

To help protect against viral infection of PGP or later Trojan horse copies of PGP, I included a signature certificate file called PGP.CTX in the MSDOS release of PGP. It bears my signature for the MSDOS executable file PGP.EXE, to assure that PGP.EXE has not been subsequently infected with a virus. To run this self-test of PGP to check its own integrity with my signature certificate, type:

```
pgp pgp.ctx pgp.exe
```

PGP should report a good signature from Philip R. Zimmermann on the PGP.EXE executable program file, which, in theory, indicates your copy

of PGP software has no virus infection and has not been tampered with. This will not help at all if your operating system is infected, nor will it detect if your original copy of PGP.EXE has been maliciously altered in such a way as to compromise its own ability to check signatures.

You should try to get at least your first copy of PGP from a trusted reliable source, so that you can use it to check my signature on subsequent releases of PGP. You can keep the older trusted version of PGP around on a write-protected backup floppy, along with a trusted copy of my public key to check signatures on future PGP releases. You'd also have to somehow make sure that my public key (also included in the PGP release) actually belongs to me, so it can be trusted to verify my signature. Make sure that you use this trusted copy of my public key, and not rely on a public key included with a newer release of PGP that may be suspect.

Just for good measure, I also included a signature certificate for this document, called PGPGUIDE.CTX. I also included a signature certificate for the all PGP source files in the source release.

Physical Security Breach

A physical security breach may allow someone to physically acquire your plaintext files or printed messages. A determined opponent might accomplish this through burglary, trash-picking, unreasonable search and seizure, or coercion or infiltration of your staff. Some of these attacks may be especially feasible against grassroots political organizations that depend on a largely volunteer staff. It has been widely reported in the press that the FBI's COINTELPRO program used burglary, infiltration, and illegal bugging against antiwar and civil rights groups. And look what happened at the Watergate Hotel. Don't be lulled into a false sense of security just because you have a cryptographic tool. Cryptographic techniques protect data only while it's encrypted-- direct physical security violations can still compromise plaintext data or written or spoken information. This kind of attack is cheaper than cryptanalytic attacks.

Tempest Attacks

Another kind of attack that has been used by well-equipped opponents involves the remote detection of the electromagnetic signals from your computer. This expensive and somewhat labor-intensive attack is probably still cheaper than direct cryptanalytic attacks. An appropriately instrumented van can park near your office and remotely pick up all of your keystrokes and messages displayed on your computer video screen. This would compromise all of your passwords, messages, etc. This attack can be thwarted by properly shielding all of your computer equipment and network cabling so that it does not emit these signals. This shielding technology is known as "Tempest", and is used by some Government agencies and defense contractors.

There are hardware vendors who supply Tempest shielding commercially, although it may be subject to some kind of Government licensing.

Traffic Analysis

Even if the attacker cannot read the contents of your encrypted messages, he may be able to infer at least some useful information by observing where the messages come from and where they are going, the size of the messages, and the time of day the messages are sent. This is analogous to the attacker looking at your long distance phone bill to see who you called and when and for how long, even though the actual content of your calls is unknown to the attacker. This is called traffic analysis. PGP alone does not protect against traffic analysis. Solving this problem would require specialized communication protocols designed to reduce exposure to traffic analysis in your communication environment, possibly with some cryptographic assistance.

Cryptanalysis

An expensive and formidable cryptanalytic attack could possibly be mounted by someone with vast supercomputer resources, such as a Government intelligence agency. They might crack your RSA key by using some new secret factoring breakthrough. Perhaps so, but it is noteworthy that the US Government trusts the RSA algorithm enough in some cases to use it to protect its own nuclear weapons, according to Ron Rivest.

Perhaps the Government has some classified methods of cracking the conventional encryption algorithm used in PGP. This is every cryptographer's worst nightmare. There can be no absolute security guarantees in practical cryptographic implementations. Still, some optimism seems justified. Widely accepted cryptographic design principles were followed in the design of this algorithm. Since the source code and design are publicly available, other cryptographers will have a chance to review it. Even if this algorithm has some subtle unknown weaknesses, the data compression of the plaintext before encryption should greatly reduce those weaknesses.

If your situation justifies worrying about very formidable attacks of this caliber, then perhaps you should contact a data security consultant for some customized data security approaches tailored to your special needs. Boulder Software Engineering, whose address and phone are given at the end of this document, can provide such services.

Without good cryptographic protection of your data communications, it may have been practically effortless and perhaps even routine for an opponent to intercept your messages, especially those sent through a modem or E-mail system. If you use PGP and follow reasonable

precautions, the attacker will have to expend far more effort and expense to violate your privacy.

If you protect yourself against the simplest attacks, and you feel confident that your privacy is not going to be violated by a determined and highly resourceful attacker, then you'll probably be safe using PGP. PGP gives you Pretty Good Privacy.

Trusting 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. 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 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. 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 these products do not come from companies that specialize in cryptographic technology.

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 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.

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 principle job of the US Government's National Security Agency (NSA) is to gather intelligence, principally by covertly tapping into people's private communications (see James Bamford's book, "The Puzzle Palace"). They have 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 henhouse. NSA has been pushing a new 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?

I'm not as cock-sure 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 snake oil. Source code is available, so other cryptographers are welcome to review its design. It's reasonably well researched. It's based on the work of a number of reputable cryptographers. It's been years in the making. And I don't work for the NSA. I hope it doesn't require a large "leap of faith" to trust the security of PGP.

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, 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.

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 going to be routed through electronic channels. Electronic mail will gradually replace 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. Perhaps the Government will protect our E-mail with Government-designed encryption algorithms. Probably most people will trust that. But perhaps some people will prefer their own protective measures.

The 17 Apr 1991 New York Times reports on an unsettling US Senate proposal that is part of a counterterrorism bill. If this nonbinding resolution became real law, it would force 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."

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 do not have access to affordable "military grade" public-key cryptographic technology.

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

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 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 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 [plaintextfile]

To generate your own unique public/secret key pair:
pgp -k

To add a public or secret key file's contents to your public or secret key ring:
pgp -a keyfile [keyring]

To remove a key from your public key ring:
pgp -r userid [keyring]

To view the contents of your public key ring:
pgp -v [userid] [keyring]

To create a signature certificate that is detached from the document:
pgp -sb textfile your_userid

To produce a ciphertext file in Unix uuencode format, just add the "u" option when encrypting or signing a message:
pgp -esu textfile her_userid your_userid

To wipe out the plaintext file after producing the ciphertext file, just add the "w" (wipe) option when encrypting or signing a message:
pgp -esw message.txt her_userid your_userid

Legal Issues

=====

Trademarks, Copyrights, and Warranties

"Pretty Good Privacy", "Phil's Pretty Good Software", and the "Pretty Good" label for computer software and hardware products are all trademarks of Philip Zimmermann and Phil's Pretty Good Software. PGP is (c) Copyright Philip R. Zimmermann, 1990.

The author assumes no liability for damages resulting from the use of this software, even if the damage results from defects in this software, and makes no representations concerning the merchantability of this software or its suitability for any specific purpose. It is provided "as is" without express or implied warranty of any kind.

Patent Rights on the Algorithms

The RSA public key cryptosystem was developed at MIT with Federal funding from grants from the National Science Foundation and the Navy. It is patented by MIT (U.S. patent #4,405,829, issued 20 Sep 1983). A company called Public Key Partners (PKP) holds the exclusive commercial license to sell and sub-license the RSA public key cryptosystem. For licensing details on the RSA algorithm, you can contact Robert Fougner at PKP, at 408/735-6779. The author of this software implementation of the RSA algorithm is providing this implementation for educational use only. Licensing this algorithm from PKP is the responsibility of you, the user, not Philip Zimmermann, the author of this software implementation. The author assumes no liability for any breach of patent law resulting from the unlicensed use by the user of the underlying RSA algorithm used in this software.

The LZHuf compression routines in PGP come from public domain source code. I'm not aware of any patents on the LZHuf algorithm, but I've heard that a related compression algorithm, LZW, has some patent claims from Unisys Corporation. LZHuf is different from LZW, and might not be affected by this patent. If you're interested, you're welcome to look into this murky issue yourself. If there are any patent claims that apply to LZHuf, then well, sorry, you'll have to take care of the patent licensing, not me.

It seems like the patent office has been issuing patents on ideas to anyone who applies for one. A software engineer may create a software package and unknowingly infringe on any number of patents. Perhaps there is a patent somewhere on using a computer to do any kind of cryptography at all. I once saw a Peanuts cartoon in the newspaper where Lucy showed Charlie Brown a fallen autumn leaf and said "This is the first leaf to fall this year." Charlie Brown said, "How do you know that? Leaves have been falling for weeks." Lucy replied, "I had this one notarized."

Licensing and Distribution

PKP controls licensing of the underlying RSA algorithm, but not on the PGP software that uses their RSA algorithm. As far as I'm concerned, anyone may freely use or distribute PGP, without payment of fees to me (except as provided below). You must keep the copyright notices on PGP and keep this documentation with it. However, this may not satisfy any legal obligations you may have to PKP for using the RSA algorithm as mentioned above. You may choose to pay PKP a licensing fee on the RSA algorithm.

PGP is not shareware, it's freeware. Published as a community service. If I sold PGP for money, then I would have to pay a license fee to PKP for using their RSA algorithm. More importantly, giving PGP away for free will encourage far more people to use it, which hopefully will have a greater social impact. This could lead to widespread awareness and use of the RSA public key cryptosystem, which will probably make more money for PKP in the long run.

All the source code for PGP is available for free under the "Copyleft" General Public License from the Free Software Foundation (FSF). A copy of the FSF General Public License is included in the source release package of PGP.

Regardless of and perhaps contrary to some provisions of the FSF General Public License, the following terms apply:

- 1) Written discussions of PGP in magazines or books may include fragments of PGP source code and documentation, without restrictions.
- 2) If you are able and willing to pay PKP a license fee for the RSA algorithm, then I guess that sort of makes PGP not exactly free, doesn't it? If you decide to do that, then I'm asking for a \$50 donation from each user that pays PKP a license fee.
- 3) Although the FSF General Public License allows non-proprietary derivative products, it prohibits proprietary derivative products. Despite this, I may grant you a special license if you want to derive a proprietary commercial product from some of PGP's parts. There may or may not be a fee depending on what kind of a deal you plan to pursue with PKP. Retaining my copyright notice and attribution might suffice in some cases. Give me a call and we'll discuss it. I'm real easy to please.

Please disseminate the complete PGP release package as widely as possible. Give it to all your friends. If you have access to any electronic Bulletin Boards Systems, please upload the complete PGP executable object release package to as many BBS's as possible. You may disseminate the PGP source release package too, if you've got it. The PGP version 1.0 executable object release package for MSDOS contains the PGP executable software, documentation, sample keyrings including my own public key, and signatures for the software and this

manual, all in one PKZIP compressed file called PGP10.ZIP. The PGP source release package for MSDOS contains all the C source files in one PKZIP compressed file called PGP10SRC.ZIP.

You may obtain free updates to PGP from BBS's or other public sources. If you must have an update directly from me, send me a \$50 handling charge (made out to Philip Zimmermann). This fee is NOT a charge for PGP, which you can get for free anywhere else. This outrageous fee is just to get me to overcome my procrastination and interrupt my bread-and-butter work and prepare a release disk for you and maybe drive down to the post office to buy some stamps, since I don't have a secretary to handle these matters. If you want much faster service, include a stamped self-addressed floppy disk mailer and a blank floppy disk. If you want even faster service, include your Federal Express account number (or better yet, one of your own Fedex airbill forms already filled out addressed to you) and I will Fedex it to you overnight at your expense. I will send you a disk with my latest and greatest source and executable object release packages of PGP. Assuming that no one tampers with the disk before it reaches you, you can trust that my public key is good and that the software is free of viruses. There's no guarantee that my version of PGP is more up-to-date than the one you have already.

After all this work I have to admit I wouldn't mind getting some fan mail for PGP, to gauge its popularity. Let me know where you heard about it and what you think and how many of your friends use it. Bug reports and suggestions for enhancing PGP are welcome, too. Perhaps a future PGP release will reflect your suggestions. But please don't count on a reply, because this project has not been funded. Technical support is cheerfully provided for an hourly fee.

If anyone wants to volunteer to improve PGP, please let me know. It could certainly use some more work. Some features were deferred to get it out the door. Perhaps you can help port it to some new machine environments, such as the Apple Macintosh or MS Windows or X Windows or XVT.

This is the first release of PGP. Future versions of PGP may have to change the data formats for messages, signatures, keys and key rings, in order to provide important new features. This may cause backward compatibility problems with this version of PGP. Future releases may provide conversion utilities to convert old keys if this is practical, but you may have to generate new keys and dispose of old messages created with the old PGP. Such a conversion effort will probably only have to be done once, if at all.

Export Controls

The Government has made it illegal in many cases to export good cryptographic technology, and that may include PGP. This is determined by volatile State Department policies, not fixed laws. Many foreign governments impose serious penalties on anyone inside their country using encrypted communications. In some countries they might even shoot you for that. I will not export this software in

cases when it is illegal to do so under US State Department policies, and I assume no responsibility for other people exporting it without my permission.

Acknowledgements

=====

I'd like to thank the following people for their contributions to the creation of PGP. Charlie Merritt designed the prototypic conventional encryption algorithm and taught me how to do decent multiprecision arithmetic. Zhahai Stewart wrote some 8086 assembly primitives and gave many helpful suggestions on PGP file formats and on the conventional encryption algorithm improvements. Allan Hoeltje integrated the LZHuf compression routines into PGP. These were developed and placed in the public domain by Haruyasu Yoshizaki and Haruhiko Okumura. The MD4 routines were developed and placed in the public domain by Ron Rivest.

Charlie Merritt can be reached at PO Box 317, West Fork, AR 72774.
Zhahai Stewart can be reached at 6521 Old Stage Rd, Boulder, CO 80302.
Allan Hoeltje can be reached at PO Box 18045, Boulder, CO 80308.

Recommended Readings

=====

- 1) Dorothy Denning, "Cryptography and Data Security", Addison-Wesley, Reading, MA 1982
- 2) Dorothy Denning, "Protecting Public Keys and Signature Keys", IEEE Computer, Feb 1983
- 3) Philip Zimmermann, "A Proposed Standard Format for RSA Cryptosystems", IEEE Computer, Sep 1986
- 4) Ronald Rivest, "The MD4 Message Digest Algorithm", MIT Laboratory for Computer Science, 1990

About the Author

=====

Philip Zimmermann is a software engineer consultant with 17 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.

Faster versions of RSA implementations are available from Zimmermann, as well as other cryptography and authentication products and custom product development services.

His consulting firm's address is:

Boulder Software Engineering
3021 Eleventh Street
Boulder, Colorado 80304 USA
Phone 303-444-4541 (10:00am - 7:00pm Mountain Time)
FAX 303-444-4541 ext 10
Internet: prz@sage.cgd.ucar.edu

