A security analysis of Pretty Good Privacy

Sieuwert van Otterloo

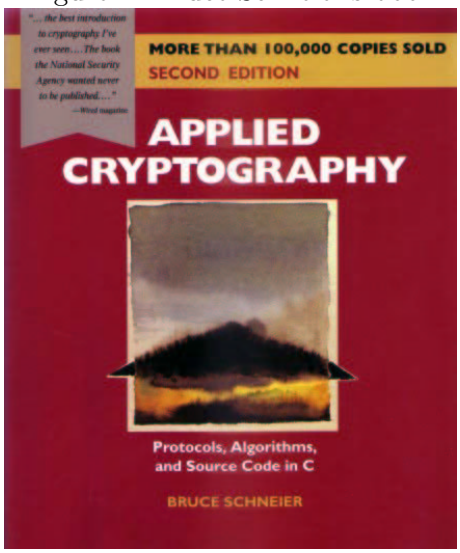
September 7, 2001

Contents

Preface

Congratulations on your acquisition of *A security analysis of PGP*. I hope you enjoy reading this text, and that it will encourage you to learn more about cryptography. To help you read more about cryptography, there is a CD-Rom accompanying this paper. It contains many of the sources that were used to make this text, the PGP program, and some otherwise useful programs. One of the highlights is the complete text of the Handbook of applied cryptography [13]. This book is a complete introduction to cryptography, and much of what is mentioned in this text can be found there, even if the handbook is not cited. If you want to buy a book, I recommend Applied Cryptography[18], which is not as good as the handbook but cheaper and nice to read. Those who live in the neighborhood can borrow it from me.

Figure 1: Bruce Schneier's book



To further enhance your reading pleasure, I have resisted starting all chapters with the crypto truisms that you will see often enough, like

> *It may roundly be asserted that human ingenuity cannot concoct a cipher which human ingenuity cannot resolve.* Edgar Allan Poe
> *Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.* John Von Neumann

I did include a few illustrations to enhance the glance-through-factor, but I am afraid it is still a very heavy text. I also added an index that will help you find definitions and explanations of important terms and acronyms. All words in `this font` are in the index.

0.1 The analysis

The following people have done the research for this 'thesis':

Sieuwert van Otterloo: A mathematics student at the university of Utrecht. Sieuwert has done most of the research, including writing this text.

Mario de Boer: Founder of the security company Njama. Mario did some of the analysing work, and gave many suggestions and assistance.

Gerard Tel: Lecturer at the Universiteit Utrecht. Gerard gave many helpful comments and suggestions.

The research was done from January to August 2001.

The main questions we want to answer are:

- What is PGP and how does it work?
- How secure is PGP? Are there any known or unknown weaknesses or backdoors?
- What are the differences between the different versions, editions and distributions of PGP? Which versions are secure, and which are not?

One of the best sources of information was the sourcecode : One of the main features of PGP was that this code was open for public inspection, so that you did not need to rely upon the skills and intentions of computer programmers you do not know. One of the goals was to do a close sourcecode analysis.

The discoveries of Ralph Senderek (see section 5.4) were one of the inspirations for me to do this research. When I began with cryptography I doubted that widely known and famous programs like PGP could contain serious or interesting bugs, but discoveries like Senderek's forced me to alter my thoughts. Senderek did not make his discoveries by sourcecode analysis, but by using an experimental aproach, analysing the input and output of PGP. Therefore the actual sourcecode analysis takes only takes one chapter, and other methods have been tried as well.

As an introduction, we wanted to describe the history of PGP, including all controversies. It is not hard to find information on this subject, since many people feel that the PGP story is worth telling, but putting it all together, and retrieving the right dates was harder. The full story does not say anything about the security of PGP, but we hope that the story contributes to a better understanding of the design decisions of PGP, and that it gives insight into the social aspects of cryptography.

As a further aid to understand what the program is doing, we ran the program with debugging aids like SoftIce, IDA PRO, and the utilities NT file monitor or registry monitor from *www.sysinternals.com*. We only studied PGP on Windows NT, because we reason that this is the way companies use PGP, and it was the platform at hand. We did not focus on studying the interaction of PGP and the Operating System(OS), because that takes lots of detailed knowledge of the OS (that we do not have nor want to have), and all errors discovered are most likely errors in the operating system. It is widely known that Windows is insecure, and we took no effort in further proving that point.

I would like to thank the people who helped me making this paper: Leon Kuunders, Beernd Noordkamp, Ernst van Rheenen, Phil Zimmermann and Werner Koch.

The official website of my PGP research is

www.bluering.nl/pgp

This thesis can be downloaded there. If you find any error or omission, or just want me to know what you think of it, please contact me:

Sieuwert van Otterloo
Rijnlaan 33bis
3522 BB Utrecht, The Netherlands
+31 6 155 24 227
sieuwert@bluering.nl

Figure 2: Sieuwert van Otterloo and Phil Zimmermann

Chapter 1

Introduction

1.1 What is PGP?

`Pretty Good Privacy` (`PGP`) is a computer program for personal privacy. It delivers privacy by making it impossible for other people to read your computer files and email, and by making it impossible for other people to impersonate you. To achieve this it uses cryptographic techniques: encryption and digital signatures. Most people are familiar with normal, `conventional encryption`, where the sender and receiver share a secret password (often called a key), that is used to encrypt and later decrypt messages. These systems are simple to understand but hard to use because you must meet someone in private to agree on a password first, and you must exchange many passwords: a group of 100 people communication with each other needs to exchange 4950 keys. To overcome these problems PGP implements a `Public Key Infrastructure` (`PKI`). PKI systems need far less keys (100 key pairs for a community of 100 people), and can also be used for unforgeable digital signatures. A drawback is that PKI systems are more complicated.

PGP is even harder to understand and use because it is a distributed system: There is no Central Server trusted by everyone to keep track of the infrastructure. All users need to make decisions about whom to trust and about the correctness of keys themselves. It would have been easier to leave those decisions to dedicated, trustable organisations.

PGP started as a small utility. I guess the original idea behind PGP was that PGP protected the communication between computers using the new RSA algorithm. The threat model was a government agency eavesdropping on individuals. During its lifetime the threat model changed, because computers became multitasking and internet-connected. Not only the communications needed to be protected, but also the computer itself had to be guarded against hostile take-overs. File erasing, hard disk encryption, a TEMPEST safe viewer and network encryption was added. A user-friendly windows version was created, and all kinds of features, like keys with photo's, integration into mail programs, and windows integration. The intended user group shifted from individual people to companies, because only companies actually paid for the program.

PGP is famous because it was one of the first publicly available programs with strong cryptography, cryptography that could stop major governments. It was free, people could read the source if they wanted to, and "it was so good that it was illegal". A large group of hackers supported and co-developed the program. Later on it wasn't illegal and owned by a large company, so rumors about backdoors rose ("If it is good why isn't it illegal?").

1.2 How PGP works

It is not possible to shortly explain how to use PGP. This section just gives a summary, to refreshen your previous knowledge. The three basic uses of the PGP programs are:

- to encrypt and sign files
- to decrypt and verify files and signatures
- to manage the collection of PGP keys you have: set properties, import or export, validate, etc.

After you installed PGP you create a key pair for yourself, and you import the public keys of your friends. You export the public part of your key. Now you can encrypt files: You select names of the keyring of people that are allowed to read the message, and PGP creates an encrypted file that can only be decrypted by the owners of the selected keys. You can also sign files: You select a private key, enter the pass phrase of that key, and PGP either creates a file with the signed data and the signature, or a separate signature. It is also possible to combine these two actions, and sign and encrypt a file at the same time. If this is the case, the signature can only be verified after the file is decrypted.

If you receive a signed file, you can doubleclick it and PGP will verify the signature. If you receive an encrypted file, you can open it and PGP will ask for your passphrase so that it can use your private key to decrypt the message. Further things to do are signing keys if they are valid and setting the trust values of the public keys you collected. This is described in the web of trust section below.

1.3 Mathematics

Cryptographic algorithms are often based on mathematical structures, especially on group theory and number theory. I do not intend to give a crash-course in number theory here, because good books on the subject already exist, but I would like to introduce the properties used and their notation here. Group theory is not explained here, because up til now PGP only uses algorithms based on groups containing integer numbers.

size(a) The size of a number is the number of bits needed to write down the number in binary form. size(a) $\simeq \log_2(a)$

$a + b$ This notation denotes addition of integer numbers.

$a - b$ This notation denotes subtraction of integer numbers.

$a * b$ This denotes normal multiplication. I prefer to use an explicit symbol instead of ab. The latter is more common in mathematical texts but I have a programming background, where the * sign is needed.

$a \mid b$ This reads "a divides b". It means that b is a multiple of a.

gcd(a, b) The greatest common divider of a and b. It denotes the largest number c with $c \mid a$ and $c \mid b$. It can be calculated using the recurrence $\gcd(a, b) = \gcd(b - a, a)$. This is called the Euclid algorithm. The Euclidean algorithm is one of the oldest algorithms (discovered in ancient Greece), and very important in number theory.

$a \bmod n$ is the number c, $0 \leq c < n$ for which there is a number k such that $a = c + k * n$.

$a + b \bmod n$ first calculate $a + b$ (this is approximately one bit larger than a or b), and then apply mod$n$. For each n, all numbers c , $0 \leq c < n$ form a group with $+ \bmod n$ as the group operation.

$a * b \bmod n$ first calculate $a * b$ (this has roughly size size(a) + size(b)), and then apply mod$n$.

$a^b \bmod n$ This is usually done with $b \in \mathbb{Z}_n$ [1] and then a^b has roughly size size$(a) *$ size(b). This is often too large to fit in memory, so this expression is calculated by writing b in binary form, like $b = 2^{i_0} + 2^{i_1} + \ldots + 2^{i_m}$,calculating $k_0 = 1, k_1 = a, k_2 = a^2 \bmod n, k_3 = {k_2}^2 \bmod n, \ldots$ and then calculating the answer with $a^b \bmod n = ((k_{i_0} * k_{i_1}) \bmod n * k_{i_1}) \bmod n * \ldots * k_{i_m} \bmod n$.

$a^{-1} \bmod n$ The above procedure is only applicable for $0 \leq b < n$. $a^{-1} \bmod n$ is defined to be the number such that $a^{-1} * a \bmod n = 1$. It exists if $\gcd(a, n) = 1$. It can be calculated using some intermediary values you get when calculating $\gcd(a, n)$ using the (extended) Euclidean algorithm.

$\mathbb{Z}_n$ The additive group with all numbers x , $0 \leq x < n$ as members and with addition modulo n as group operation: $c = a + b \bmod n$.

$\mathbb{Z}_n{}^*$ The multiplicative group with all numbers x, $0 < x < n, \gcd(x, n) = 1$ as members. The group operation is $c = a * b \bmod n$.

$\phi(n)$ The number of members of $\mathbb{Z}_n{}^*$. It is $n - 1$ if n is a prime number. Fermat's little theorem says that if $a * b \bmod \phi(n) = 1$ then for all $m \in \mathbb{Z}_n{}^*$, $(m^a)^b \bmod n = m$. Calculating $\phi(n)$ is hard if you do not know the factorisation of n.

$a \oplus b$ The bitwise exclusive OR of two integers. It is calculated by writing the inputs in binary form, and setting a bit of output 1 if the input bits are different, 0 if they are the same. For instance $12 \oplus 10 = 1100 \oplus 1010 = 0110 = 6$.

ord(g) The order of an element g in a group is the smallest number k such that $g^k = 1$.

1.4 Key size

One question keeps coming back in discussions between cryptographers: What is the right key size? This question has no general answer. There is no mathematical way to prove a certain key size to be sufficient. One can not look into the future and predict what computers will be able to do then.

The reason keysize is important is that the US government have long restricted the maximum key size allowed, because they wanted to be able to break all cryptographic products that were exported to other countries. They have spread false information, telling that 40 bits is enough for all but military purposes. PGP users and other cryptographic enthusiasts, paranoid as they are, counterreacted and used huge key sizes. Some products claim to be more secure than others because they have a larger key size. This is not true: once the key size is beyond a certain limit, defined below, the security depends on other factors.

The notion of key size has a few different meanings. I would like to define three types of key size here:

[1] or $b \in \phi(n)$, but $\phi(n)$ is not always known

raw key size $S_{raw}(K)$: For a key K, the raw key size $S_{raw}(K)$ is the number of bits needed to write down K. We can define, for an algorithm or crypto system A, $S_{raw}(A) = S_{raw}(K)$ if K is a key for A and all keys for A have the same raw size.

real key size $S_{real}(A)$: The real key size of a certain algorithm/crypto system A (with all parameters fixed) is the 2-based logarithm of the number of possible keys. Note that the different keys of A need not have the same raw key size. This key size relates to the hardness of a `brute force attack`: trying all possible keys.

effective key size $S_{eff}(A)$: The effective key size, or the strength of A, is the 2-based logarithm of the number of keys that must be tried to break the system, or of the number of actions an attacker must do to break the system. Note that the attacker might have a better attack than a brute force attack, so that the strength can be less than the real key size.

It is always true that

$$S_{raw}(A) \geq S_{real}(A) \geq S_{eff}(A) \tag{1.1}$$

In the end, the designers want that their system will never be broken in practice, at least not in the next decennia. One can estimate the amount of computing power currently available, and based on such estimates the following facts are widely accepted among security specialists

Fact 1 *A cryptosystem A is insecure (practically in 2001) if $S_{eff}(A) < 60$*

As some sort of proof of this fact: The Electronic Frontier Foundation built a specialised DES cracking machine, that could search trough 2^{56} keys in a few hours. The American government has long restricted the key size to 40 bits. That is a million times less secure than insecure.

Fact 2 *A cryptosystem A is secure according to current (2001) knowledge if $S_{eff}(A) \geq 80$*

This fact needs redefinition in the future because computers get faster, and the number of computers is also increasing. In fact, some people have already gone from 80 to 128, and some (paranoid) people are already thinking of even larger key sizes, because modern computers can handle it and having more key bits does certainly not decrease the security. However, increasing the key size is often improving one of the strong spots of PGP, while attackers usually focus on a weak spot.

For block ciphers, designers try to make $S_{raw}(A) = S_{real}(A) = S_{eff}(A)$, and they can choose $S_{raw}(A)$ equal to 128 bits to have some sort of safety margin. Since there are usually no constraints on the keys, $S_{raw}(A) = S_{real}(A)$ can be achieved easily. One notorious counterexample is DES: DES had an 8-byte key (64 bits), but the last bit of every byte was not used, so $S_{raw}(DES) = 64$ and $S_{real}(DES) = 56$. Triple DES suffers the same problem: $S_{raw}(3DES) = 192$ and $S_{real}(3DES) = 168$ (and $S_{eff}(3DES) \leq 112$ because of a meet in the middle attack).

In the distant future scientists might develop a `quantum computer`. This is a device based upon the principles of quantum mechanics that can do calculations on many numbers on the same time. Quantum computers can do no more than reducing the amount of work to do by a square root [2], so people worried about quantum computers should use double size keys: 160 bit or 256 bit keys instead of 80 or 128 bit keys. If quantum computers work the way we expect them to do, $S_{eff}(A) \leq S_{real}(A)/2$, but please take this with a grain of salt because quantum computers are distant future.

[2] according to Schneier, 10th Oct 1998 posting to soc.history.what-if USENET group. See the pgpdhrsafaq on the CD

For hash functions H there is no key, so the size of the output is taken as the 'key size' of the hash function. The `birthday attack` assures that $S_{eff}(H) \leq S_{real}(H)/2$. Therefore $S_{real}(H)$ must be at least 160. MD5 fails this requirement, but since $S_{real}(MD5) > 2 * 60$ it is not per definition insecure. Let's call it dubious.

For public key algorithms the matter is more complicated. For the 'key-size' of a PK algorithm one often takes the 'group size' as the size of the key. $S_{raw}(PK)$ is often larger, because you need to take all the numbers into count to get the raw key size. There is also a difference between the secret part of the key and the public part of the key. For $S_{eff}(PK)$ only the secret part is relevant. Furthermore it often holds that $S_{raw}(PK) > S_{real}(PK)$ because the numbers involved must have special properties like being prime.

For public key algorithms we also want that $S_{eff}(A) \geq 80$, and because of attacks similar to the birthday attack, it holds that $S_{real}(x) \geq 160$ with x a part of the secret key. These attacks are described in section 3.3.2

Public Key algorithms often work in a certain kind of group. The name of the group is known, but some things must be hard to do without the private key, so people should not find out the structure of this group. In practice, for the algorithms in PGP, it means that the group must have at least 2^{1024} members to offer an 80 bit strength against factoring algorithms. Very different groups, like Elliptic curves, could behave differently and need a different group size.

1.5 Web of Trust

Each person that wants to participate in PGP communications must retrieve the necessary public keys. This can be quite a task, because in PGP all keys are generated at different places. Often you need to send an email to the person you want to communicate with. You can also connect to a dedicated server, a so-called key server, to automatically retrieve keys.

If you have all the necessary keys, a bigger problem emerges: You must verify that the keys are valid: that they indeed belong to the person whose name is on it[3]. You cannot assume that email or any other internet service gave you the right information: If internet were secure we would not need PGP. Even if you have used a certain key successfully for a long time, you cannot be sure it is right. It could be the case that you and the other person are using keys belonging to an eavesdropper. The eavesdropper intercepts all your messages, reads them, modifies them and sends them through. This is called a `man in the middle attack`.

You can mark keys to be valid yourself. You should only do this if you got the key personally, or if you have verified the fingerprint of the key. You can do this by phone, if you can identify the voice of the key owner. Marking a key as valid is done by signing it. You can allow the signature you made exportable, and distribute the key with the signature. If you send it back to the owner, he can distribute his key with signatures by several people. This allows for indirect validation of keys: Suppose A wants to check E's key. E's key has a few signatures on it, one of them made by C. A retrieves C's key. It is signed by B, and A has met B in person, so A has a valid version of his key. If, and this is important, if A trust B to be a reliable person, she can assume that the key of C she has is valid. If she further assumes that C is a reliable person, who would not incorrectly sign E's key, she can conclude that the key of E that she has is valid.

If B would have trusted C, and signed the key of E based upon his faith in C, A could conclude that E's key is valid solely based upon the validity of B's key and the reliability of B as a person.

The indirect verification of PGP keys makes it possible to safely communicate with people you

[3]We assume for the moment that all the keys have just one name

have never met. This `web of trust` is one of the main features of PGP. Other PKI infrastructures rely on some central server to publish certificates, and ask each user to rely on those certificates. That system has two disadvantages. It costs a lot of money: those certificates will not be free, and they are often distributed with a limited lifetime, forcing you to pay a considerable sum of money each year. The second problem with a central certification authority is, and this is a debatable objection, that it is not wise to trust central authorities. You cannot check that they are careful, and it is very easy for the government to manipulate these authorities. If you trust the government, you might as well assume that people are friendly and the internet is safe, and stop bothering about encryption.

The PGP philosophy gives you the right to choose whom to trust. Everyone maintains his own database of trust, and create his own certificates. You can share the information for free with your friends, without any government control. The only drawback is that the PGP web of trust system is hard to grasp, and it is easy to draw wrong conclusions. The most common pitfall is to confuse the validity of someone's key with the trustworthiness of the owner. In the first variant of the above example, A relied on C's signature, and his key was signed by his friend B. B did not state that C was trustworthy. By signing some-ones key you only state that the key belongs to the person whose name is on it. The decision to trust C was made by A, and if A does not know C, it was a wrong decision.

The above explanation is the simple, basic web of trust. PGP uses a more advanced system using marginal or partial trust. The idea is that you should never trust people or keys totally: Each extra step in the chain of reasoning about validity slightly reduces the probability that a key is valid, and you should only use keys with a high enough probability of being valid. The chance that a key is valid increases if you can establish two independent paths from him to you.

Another feature of the PGP system is the concept of meta-introducers. You can trust people on different levels. For example

level 0 You can trust that you have a valid copy of B's key.

level 1 You can trust that you have a valid copy of B's key, and that B only signs valid keys. That is: B is a trustworthy person.

level 2 You can trust that you have a valid copy of B's key, and that B only signs valid keys of trustworthy persons.

...

level n You trust that B's key is only used to sign keys belonging to people of category $n-1$. (and that you have a valid copy of B's key).

If you trust B on or above level 1, B is a meta-introducer for you. His signature can validate whole families of keys. It seems not right to trust regular people on high levels, because few people are that careful. The concept is useful because many organisations have special keys that are stored extra securely, and only used to sign keys. The sysop of the organisation will verify all keys by phone and sign them with the keysigning key, and everyone else in the company can trust the keysigning key on level one.

Ueli Maurer[12] designed a probability model for reasoning with partial trust. This is the system used in PGP. It uses the basic reasoning rules sketched above to reason about validity of keys, based on a set of initials beliefs about validity and the signatures given. Each of the initial beliefs is given independently a confidence value that can be considered a probability value between 0 and 1. For

each statement that can be derived, the confidence is the probability that any set of initial beliefs from which it can be derived is completely true. This leads to a system where partially independent validity paths increase the confidence in a key validity in a plausible way.

The advanced model seems to be very well thought. In practice one does not always want to deal with such detailed estimates of trust. The PGP 7.0 does not allow arbitrary confidence settings, but only 4 steps: invalid, valid but not trusted, marginally trusted and fully trusted. When using PGP myself I tend to only use fully trusted and untrusted, because it is easier to understand, and I think most people do the same.

It is important to have your key signed by a lot of people if you want it to be easy for other people to verify your key. The early users of PGP, the 'hackers', went to key signing parties to get those signatures: All party people bring their public keys, and get their key signed by all other participants.

The information you give to PGP about trust is treated confidentially: Signatures are only exported if the user explicitly allowed it, and the trust information is never exported. This is important: You do not want your best friend to know you do not trust him because he is sloppy.

1.6 Vulnerabilities beyond PGP

PGP, and cryptography, have certain limitations. They cannot solve all problems related to computer security and privacy. The manual of PGP[14] warns against a few threats that PGP users must be aware of.

1.6.1 Easy passphrases

When people are asked to create a password or passphrase of some sort, they often come up with the name of a person, a date or phone number, or a common word. They choose something they can easily remember, but also something that can be guessed easily. Hackers have created complete dictionaries of common passwords, and in a `dictionary attack` they try all these words, often with success. PGP does not speak of *passwords* but of passphrases. Any private key is always stored encrypted, and the passphrase is needed to unlock it. Users are encouraged to choose something hard to guess: Multiple words, with numbers and funny characters in it. It is also important that each passphrase is unique. If you use the same passphrase for your PGP private key and your webmail account, the webmail sysop knows your PGP passphrase, as well as any hacker that can break that system.

1.6.2 Not quite deleted files

Deleting information from a computer disk is not as easy as it seems. Suppose you have a file containing sensitive data, and you remove the last line of the file because it is too compromising. The line is probably still there because the file system has just lowered a counter that says how much of that cluster belongs to the file. If that counter is raised again, the line is back.

Suppose you decide to delete the entire file. The system will remove the file from the index of the disk, but not remove the contents of the file. It is still there. Anyone can recover the file, for instance with common system utilities (Norton disk doctor was famous for this). What is needed is that the file is overwritten.

PGP contains subroutines for overwriting all free space on a disk: Both the free clusters and the unused part of partially used clusters. This makes sure that one cannot retrieve the data without opening the computer, and it is called `freespace wiping`.

If one is willing to open the computer, things become interesting again. The data is still not destroyed if overwritten once. `Destruction of data` or `DoD` is harder than it seems. By using the right electronics one can find out what the previous data on a disk was by measuring the strength of the signal. For instance: If the disk gives 0.98, it indicates a 1 overwritten by a one, 0.8 is a zero overwritten by a one, 0.2 indicates a 1 overwritten by a 0, and 0.02 indicates a 0 overwritten by a 0. If an attacker is even willing to open up the hard disk, he can measure the entire surface of the disk, and the off-centre parts of a track. Peter `Gutmann` [7] wrote an article about it that forms the basis of PGP's file wiping routines. It involves overwriting the data several times with different patterns, making it as hard as possible to retrieve previous values. This is really a task for an operating system, but because the major OSes are not written for security PGP will do it for you, since PGP 6.0.

The same paper [7] also tells how to destroy data from memory, but this theory is not brought into practice by PGP. I guess the PGP makers think these attacks are not likely.

1.6.3 Viruses and Trojan horses

An attacker that desperately wants your pass phrase but does not know how to break PGP, can infect the rest of your computer with a piece of hostile software. He can for instance write a virus that collects all keyboard presses and sends them through. PGP users should use antivirus and firewall software to protects against these attacks.

Another attack is that the attacker tries to slip you a malfunctioning copy of PGP. This copy could leak your secret key, or accept certain signatures it should not. To prevent this, NAI signs all releases of PGP, and users should verify their version of PGP using a previous PGP installation.

To research this last possibility, we downloaded many PGP installations from the internet and compared them. It turned out that there are thousands of copies of PGP floating around, but most of them are equal. Few people bother to compile their own version. A reason for this can be that we tested for the windows version of PGP 6, and it is a lot of work to compile this program, and you need programs not many people have to do it. Our conclusion is that if you randomly download a PGP version from the internet, it will not be sabotaged in any way. But if you have specific enemies you should really check the correctness of your PGP.

1.6.4 Physical security breach

If you have secured your computer using PGP and other tools, it might be worthwhile to secure your office equally well. Determined attackers would not mind burglary, trash-picking, blackmail, bribery or infiltration.

1.6.5 Bogus time stamps

This is not so much a weakness in PGP, but one of its natural limitations. All signatures have a timestamp in it: you can see when the signature was made. The time is whatever the signer wanted it to be. It has no absolute meaning, and does not prove anything. This is similar to dates companying ordinary signatures under paper contracts.

1.6.6 Traffic analysis

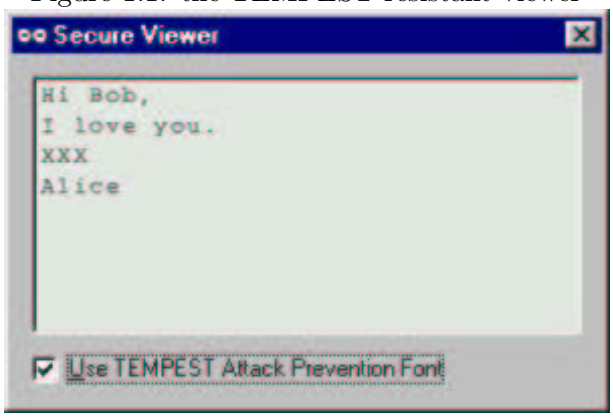
PGP prevents people from reading what you sent to whom, but does not prevent people from monitoring with whom you are communicating. This is called `traffic analysis`, and can leak

much information about you. PGP does not prevent against this. If you are afraid of this, you should send email indirectly (send the message encrypted to a friend and ask him to send it through with a random time delay), or set up a mail server to do this.

1.6.7 TEMPEST

TEMPEST (no acronym, but derived from Shakespeare – or maybe it means Transient Electromagnetic Pulse Emanation Standard) stands for collecting secret information by 'listening' to the radiation emitted by electronic equipment. It dates back to the 1960's and military equipment is often shielded against these attacks. TEMPEST does not happen often, since it takes expensive equipment, but is a possible threat nevertheless. See *www.tempest.org* for more information. The best results are obtained by listening for radiation emitted by cathode ray tube screens (ordinary glasstube monitors). PGP has the option to show messages in a special font with smooth edges and irregular characters that makes the TEMPEST signals harder to catch. Check *http://www.eskimo.com/˜ joelm/tempest.htm* for more information.

Figure 1.1: the TEMPEST resistant viewer



Chapter 2

The history of PGP

The first half of this chapter is mostly based on [6]. Other information comes from various internet resources, which can be found on the CD-ROM.

2.1 Before 1.0

The initial and main author of PGP is Philip (R.) `Zimmermann`. He was born in Camden, New Jersey, in 1954. He graduated in 1978 in Computer Science and moved to Boulder, Colorado. He became active in the anti-nuclear-arms movement in 1982, and learned much about military policy. He also started a small computer company, `Metamorphic Systems`. It is common to abbreviate his name as `PRZ`.

Figure 2.1: Phil Zimmermann

Charlie `Merritt`, founder of another small computer company, contacted Metamorphic Systems to interest them in his microcomputer cryptography package. Zimmermann was very enthusiastic, and called Merritt more than once a week for several years, trying to learn the large-number arithmetic needed for RSA. They decided to meet each other in the summer of 1986. Jim Bidzos, the president of the RSA Data Security Company, was also present for a business meeting with Merritt. They talked about encryption. Zimmermann and Bidzos did not get along very well, because they had very different personalities.

`RSA Data Security` (`RSADS`) is an American computer software company founded by the creators of the RSA algorithm: Ron Rivest, Adi Shamir and Len Adleman, in 1983. The company had a patent on the use of the RSA algorithm. In fact a company called `Public Key Partners` (`PKP`), formed by RSADS and `Cylink` had the license, but RSADS exploited it. It expired in September 2000. Of the two PKP partners, Cylink was the largest, but also the less visible. They did not make software but hardware. The partnership fell apart when they started arguing about how the income of PKP had to be split. Cylink owned the `Merkle-Hellman` patents,

which protect all public key cryptography, according to Cylink. Some people think those MH-patents are invalid, because the patented technology is not working: the knapsack system described in the patent is broken.

Phil Zimmermann needed a license to use the RSA algorithm in his program. The company still exists. The name has been changed to `RSA Security` (websize:*http://www.rsasecurity.com/*).

2.2 Guerrillaware

2.2.1 PGP 1.0

Zimmermann went on programming and in 1991 his program, PGP, was nearly finished. In April 1991 he wrote a letter to RSADS asking for 'the free license they had spoken about'. Jim Bidzos refused to give this license. Phil Zimmermann could have bought a license, but that would mean that PGP could not be distributed as shareware like he wanted to do. So there seemed to be a problem. At the same time something else happened: the American government tried to forbid cryptography in the 1991 Anti-Crime Bill S.266. The relevant paragraph was removed after an industry-wide offensive, but Zimmermann felt really attacked by this law: He had invested years of his life in PGP, was almost losing his house, was frustrated by RSADS, and now the government tried to illegalize his program. In June he quickly 'released' his program in the following way: He gave a copy to a friend, who put it on the internet. The software was released by the imaginary company Phil's Pretty Good Software (`PPGS`).

RSADS, in the person of Jim Bidzos, was not very happy about it. Although the company was not very threatened by the program, PGP set a bad example, and other people might also decide not to pay for the use of RSA. Jim Bidzos also had a personal dislike against Phil Zimmermann, so RSADS threatened to sue Zimmermann for spreading a program that illegally used a patented algorithm. Zimmermann defended himself by saying that not he but a friend distributed the program. They made an agreement that Zimmermann would not spread any more copies of his PGP program, and that RSADS would not sue him for the copies already distributed. But there was no way to stop the success of the program: people continued using PGP.

Version 1.0 of PGP used RSA as public key algorithm, Bass-o-Matic for conventional encryption, `MD4` as hash algorithm, and LZHUF as data compression algorithm. The Bass-o-Matic algorithm was created by Zimmermann himself, and at the Annual *Crypto* conference held in August 1991, he discovered that it was insecure. (He designed a cipher himself because the DES, the best known cipher, had a too small keysize and he knew no alternative).

Besides this security problem, this version of PGP also suffered from a few legal problems:

- It uses the patented RSA algorithm. The patent was only valid in the US, because only in the US it is possible to obtain a patent after publication.
- PGP contains cryptography, and cryptography falls under the *International Traffic in Arms Regulations*. It was illegal to export cryptographic software (except in book form, strangely enough).
- Other countries have other rules against importing and using cryptography. These however were not very relevant for the development op PGP in this stage.

The US government where tipped by Jim Bidzos that Zimmermann was infringing software patents. During the investigation, they changed subject: They tried to prosecute him for violating the export restrictions. PGP users started a campaign against the export rules, and to fund Zimmermann's legal expenses. The US government finally gave up the investigation in early 1996.

2.2.2 PGP 2.0-2.3a

In the fall of 1991 and in 1992 a team of programmers all over the world worked on PGP version 2.0. The most important change was the replacement of Bass-o-Matic with IDEA . The most important authors were Branko Lankester (Netherlands) and Peter Gutmann (New Zealand). It had a few bugfixes until a stable version emerged: PGP 2.3a.

IDEA is a very good block cipher, but it is not patent free: According to [18] it is free for non-commercial use, and that is probably why they decided to use it, but its inclusion meant more legal trouble for PGP.

2.2.3 PGP 2.4.x

The company ViaCrypt had obtained a license for the use of RSA. Zimmermann made a deal with this company, and in November 1993 they released a commercial version of PGP: PGP 2.4. It was not infringing any patents, but it also wasn't for free any more. The commercial version was not better than the free one, but could be bought by companies that do not wish to use free software.

2.2.4 PGP 2.5

The company RSADS was always opposed against freeware, but in 1993 they released RSAREF, a toolkit freely usable for non-commercial purposes. The goal of this release was to support the adoptation of PEM (Privacy Enhanced Mail) as an internet standard. At the end of the summer of 1993, some people of the Massachusetts Institute of Technology (MIT) wanted to make a new version of PGP using the RSAREF library. In January 1994 there was a meeting between Jim Bidzos, Jeffrey Schiller (MIT's network manager), Ronald Rivest (the R in RSA , professor at MIT) and John Preston (boss of the MIT technology licensing office). One month later PRZ talked with Schiller and James Bruce (professor and vice president for information systems at the MIT). These talks were not successful.

In March 1994 RSA released version 2.0 of RSAREF. This library contained new functions, and these new functions were very useful for the implementation of PGP. The MIT group contacted Zimmermann, and they created PGP 2.5. This is a free edition, but less free than the first: All previous versions of PGP appeared under the copyleft license agreement. This means that the program and the sourcecode are free, and that any program that uses the copyleft program must also be free (for details check *http://www.gnu.org/copyleft/*). Since RSAREF is only for non-commercial use, the agreement had to be changed: PGP version 2.5 can only be used freely for non-commercial purposes, which probably means: free for use for private email on home computers.

2.2.5 PGP 2.6

RSADS never intended the use of RSAREF 2.0 in PGP, and they were furious about it. Since 2.5 was compatible with earlier versions of PGP, Jim Bidzos reasoned that it would induce people to use illegal software. Two weeks after the release of 2.5 an agreement was made: RSADS would allow the distribution of a new version of PGP, PGP 2.6, with the same functionality of 2.5, but incompatible with the older, patent-infringing versions. It appeared May 22 1994. The incompatibility was that if you run PGP 2.6 after 1 September 1994 it will create incompatible signatures. This dateline was agreed on to give people some time to switch.

2.2.6 PGP 2.6.x

The MIT releases where targeted at the American market. The software could not be exported to European or other international users. The official MIT server was programmed not to send it to computers outside the US. Several people however risked a few years jail and exported the program. Once a few versions of the program had escaped the US these versions where modified, beautified and improved by enthusiast users. Also people in the US appreciated the improvements, so some of the 'international' versions were imported in the US.

The most important improvements where the use of other libraries than RSAREF, and larger key sizes. The improved versions have no consistent naming scheme. Most of them are called 2.6.xyy, wherein x stands for the subsubversion number, and y for various adjectives: i for international, g for guerrilla, u for unofficial. Just to name a few:

- PGP 2.6.3g (US Guerrilla version)
- PGP 2.6.3ig (International Guerrilla version)
- PGP 2.63uin
- PGP 2.6.3CKT
- PGP 2.6ui 2.62ui 2.63ui 2.64ui.

These versions are mentioned on the website *www.pgpi.org*.

2.2.7 What about PGP 3.0?

The version number 2.6 remained the newest number for quite some time. Many fixes and small adjustment where made, but large changes where postponed until the upcoming version 3. This version should present a complete code-cleanup, new features etcetera, and be produced by PRZ. The version is often announced, mentioned and discussed, but it does not exist: The development of version 3 took much time, partly because PRZ was occupied by the legal struggles. When it finally emerged, it was so different that they called it version 5.

2.2.8 PGP 4.0 and 4.5

Like 2.4, these versions were released as commercial products by ViaCrypt. Both of these can be used in two flavors: The Business version and the Personal version. The business version includes some sort of data recovery features, to facilitate business use. PRZ was not directly involved in the design of the new features, so these versions form a sideline.

2.2.9 PGPPhone

Phil Zimmermann also tried to make a program for making secure telephone calls. `PGPPhone` is an attempt to make encrypted phone calls over the internet. It did not work very well, mostly because internet telephony requires a very, very good internet connection. On march 21 1999 the sourcecode of the program was released because the new owner of anything PGP, `NAI`, did not want to spend more time on the program, and after that not much has happened.

2.3 PGP as a normal program

2.3.1 PGP 5.0

This version ends the underground history. All legal issues are settled, and PRZ can finally do what he wanted to do: make a living from selling cryptographic software. The cute company name PPGS is replaced by the simple `PGP Incorporated`.

The new program was written from scratch, and provided a graphical user interface. It ran under Windows 95, Windows NT and Apple Macintosh. A nongraphical version was written for Linux/Unix. It featured many new algorithms like the CAST and 3DES blockciphers, and SHA-1. More important was the introduction of Diffie-Hellman public keys. The patent on DH keys expired April 29 1997. Now a commercial company sold that PGP, the policy was that both a commercial and a freeware version would be released, as well as the sourcecode, so that anyone can inspect the program and check the security.

PGP Inc was very convinced of the superiority of their new DH keys, and suggested that everyone upgraded to the new system. RSA support was removed from many distributions, mainly from the freeware versions. Users objected: People did not want to upgrade. It was felt that RSA support was not available in the freeware edition to force people to buy the new version. Arguments in favour of a massive change are that the new keys are more secure than the old: Key ID's are harder to replicate and MD5 has been replaced by a better hash function. Maintaining RSA support was not impossible, but would mean paying RSADS for the commercial versions of PGP. This difference in opinion between users and makers of PGP took quite some time, and explains why there are so many versions of PGP 5 and beyond. Many websites try to sort out which versions exist and what PK algorithms they support. I have lost count: This document does not list all of them, and does not intend to predict the availability of RSA.

Other new features were plug-ins for mail programs like Eudora and Outlook.

The most accurate information I have about the 5.0 versions lists the following official variants, released June 16 1997 (the 5.0i appeared a few weeks later):

PGP 5.0 The full function business edition

PGP For Personal Privacy Freeware 5.0 A MIT release. It can use, but not generate RSA keys.

PGP 5.0i The international freeware version. It features RSA generation and use. This version was the first version exported out of the US without breaking the export rules. PGP Inc printed and published the program in book form (10 books, 5000 pages) and the books were exported to Europe (exporting cryptographic software on paper was not forbidden). A team of volunteers, lead by Stale `Schumacher` from Norway, converted the books back to bytes by means of scanners and OCR software(70 volunteers, 1000 hours of work). The website of these volunteers is: *www.pgpi.org*. The latest version exported this way is 6.5.1i, because in 1999 the US government lifted the export controls.

PGP 5.0ic This version is distributed by `PGP Europe`. This was an independent company, not affiliated with PGP Inc, but with a license to use their trademarks. It features RSA generation and use. I assume the company no longer exists, because I have never read about it again.

Version 5 also supported the use of key servers: The program could automatically connect to internet servers that maintained a database for public keys.

2.3.2 PGP 5.5

This version features improvements to the interface and file wiping, support for additional decryption keys and improved plug-ins. PGP Inc distributed both the freeware and the commercial edition. PGP 5.5.3 was released 4 December 1997. The international version appeared in January 1998 (scanning was completed in two weeks, because the books had a better font this time).

An `Additional Decryption Key` or `ADK` is a new and disputed feature in this version. One of the 'problems' of using cryptography is that once you lose a key or passphrase, you cannot access the valuable data. This problem seems inevitable because it is in fact exactly what you want from encryption, but several solutions have been thought of. One of these solutions is that anyone gives his private key to the government, which will safely store it. This solves another 'problem', namely that the government cannot read encrypted messages. This is called `key escrow`. One can also store each key in such a way that only a government owned private key can read it. This is called `key recovery`. One can imagine that not everyone is fond of this solution.

To facilitate business use of PGP the makers created an alternative, where the keys remain secret, but the messages can be retrieved in case of emergency: `data recovery`. This means that a certain company-owned public key is always included in the intended recipients of each message, so that the company can read any message sent to an employee. For outgoing email this is just a matter of policy: The company tells all employees they should do this. For incoming mail, an extra feature had to be added: The ADK. It means that the employees add the company key in their public key as an ADK, and that all people that send email to them also include the ADK in the recipients. All this is voluntary: If the sender does not want this, he can simply ignore the ADK.

This solution is not as bad as the first one, but still offended many PGP users. They saw it as a first step towards the end of their privacy. The feature however was not designed for them, but for company use: For companies data loss can be worse than privacy loss.

2.3.3 PGP 5.5.5

PGP Inc bought by NAI

On December 1 1997 the company Pretty Good Privacy Incorporated was taken over by `Network Associates Incorporated`. They took over the entire company, not just the product, so the same people kept working on PGP (including PRZ). NAI was a member of the `Key Recovery Alliance` (`KRA`), an industry group that was in favour of key recovery, mostly because it would allow export. As soon as NAI acquired PGP, they stepped out of the alliance.

PGP 5.5.5 is basically the same as 5.5.3 but with a different copyright message mentioning NAI.

NAI later became a member of the KRA again when they bought another company, `Trusted Information Systems` (`TIS`), but they dropped membership again very soon.

2.3.4 PGP 6.0

A new version with a few new features: It is possible to include a photo of the key owner in the public key and you can specify in your public key that you allow other people to revoke your key. This is useful if you need to revoke your key because you lost the pass phrase. Freespace wiping is also introduced (see section 1.6.2), and key splitting.

`Key splitting` allows one to split a private key into several parts, such that the key can only be used if enough parts are joined. This feature is intended to be combined with ADK: The additional decryption key can be split to make sure it is not used too often.

NAI released 5 variants of this version:

- PGP Desktop Security 6.0.0 DH
- PGP Desktop Security 6.0.0 RSA Add-on
- PGP Personal Privacy 6.0.0. DH
- PGP Personal Privacy 6.0.0 RSA Add-on
- PGP Freeware 6.0.0 DH

The RSA-marked versions can use and generate both kinds of keys, The DH marked versions just work with Diffie-Hellman keys.

A bug was discovered in the handling of RSA keys larger than 2048 bits. NAI fixed it in the next version:

- PGP Desktop Security 6.0.1 RSA Add-on
- PGP Personal Privacy 6.0.1 RSA Add-on

PGPDisk

The business version of PGP 6.0 includes `PGPDisk`: This is a separate program that can be used to create virtual disks that are encrypted entirely. This program can be used without the rest of PGP, but uses the same encryption techniques. (It uses no public key encryption.) PGPDisk is included in all commercial versions after this version, but not in all freeware versions.

2.3.5 PGP 6.0.2

This version features the `TEMPEST` safe viewer for sensitive messages (see section 1.6.7).

The versions officially released are

- PGP Desktop Security 6.0.2 DH
- PGP Desktop Security 6.0.2 RSA Add-on
- PGP Personal Privacy 6.0.2 DH
- PGP Personal Privacy 6.0.2 RSA Add-on
- PGP Freeware 6.0.2 DH

Freeware versions of these are:

- PGP 6.0.2i does not have RSA key generation
- PGP 6.0.2 CKT. Above build 03 it features PGPDisk.

2.3.6 PGP 6.5

This version was released on April 5, 1999. New features are PGPNet and Self-Decrypting Archives (PGP creates an executable that ask for a pass phrase and then decrypts itself, so that it can be sent to people without PGP) , and X.509 and CA authentication. The following variants were released:

- PGP Desktop Security 6.5.1
- PGP Personal Privacy 6.5.1 (30-day trial available)
- PGPfreeware 6.5.1 for Win 95/98/NT
- PGPfreeware 6.5.1 for MacOS
- PGP Command Line Freeware 6.5.1 for Linux/Solaris/NT

This version has had a few subversions that offered minor improvements:

- PGP 6.5.2 Support for Windows 2000, and Intel RNG support.
- PGP 6.5.3 First version legally exported in executable form. No new features
- PGP 6.5.8 Fixes the ADK problem discovered by Ralph Senderek.

2.3.7 PGP 7.0

Released 10 September 2000. The freeware version appeared later: 3 February 2001. Although the source code release had been announced, it was not released. This version offers many improvements in the PGPNet section, new format RSA keys, continuous entropy collection, Twofish support and an ICQ plug-in.

PGP 7.0.1

Most important new feature is the introduction of a new algorithm for conventional encryption: AES.

2.3.8 Cyber Knights of Templar

The CKT or Cyber Knights of Templar was a loose group of hackers and digital freedom-fighters that made their own variants of PGP. One of their members, Imad R Faiad, still maintains a website. They were certainly not the only group releasing PGP variants, but their versions are widely known, and I also list them as an example, to show that indeed many variants exist.

- PGP 5.5.3ckt
- PGP 6.0.2ckt
- PGP Freeware 6.5.1i
- PGP Freeware 6.5.1i - CLI
- PGP 6.5.8ckt

2.4 The future of PGP

PGP is still in development and new versions will emerge, but since it is no longer open source, I would like to introduce two alternative projects.

2.4.1 Gnu Privacy Guard

It is nice that all the legal problems are solved, and undoubtedly the world benefits from the good cryptographic software that NAI is selling. Unfortunately NAI seems to think only about money: new features are targeted at companies running Windows, and the sourcecode is no longer released.

A group of people lead by Werner Koch is now making a free alternative to PGP, fully compatible with the OpenPGP standard. It runs under windows and linux/unix. They are not making a graphical user interface, but provide a command line tool like the older versions of PGP. Full support of all new formats is given. No patented algorithms are supported, so it does not have IDEA. The products website is *www.gnupg.org*, and it all started in Germany. The group also received a large sum of money from German government.

GNU stands for *GNU is not UNIX* and is an attempt, supported by a large group of computer enthusiasts, to create a complete set of free software. Many excellent GNU programs exist, like the GIMP (GNU Image Manipulation Program) and the Gnu C Compiler. Although most GNU projects are based on UNIX, GNU is not limited to UNIX, hence the name.

GPG is a very promising alternative to PGP. It is open, free, and hopefully as good as PGP. GPG is less mature than PGP. New GPG versions appear more often than PGP versions, and I believe that is because they are still in a process of debugging, but if you are willing to upgrade from time to time GPG is as safe as PGP.

2.4.2 Hushmail secure email

Many people use a webmail service: Their email is stored on an internet server, and can be accessed through a website. The largest webmail service is *www.hotmail.com*. Webmail has some severe security problems, the most important of which is that the server owner can read your email and knows your password. It is possible to use PGP together with a webmail account, but it is not as easy as using PGP with an email program.

`Hushmail` is a webmail service that solves those security problems. It uses cryptography similar to PGP, and does all calculations on the client side computer, in a java applet. PRZ is a cryptographic consultant for Hushmail. They are making the next version of Hushmail, Hushmail 2.0, compatible with OpenPGP.

The sourcecode of Hushmail is freely available to be reviewed. A basic hushmail account is free, but they hope you want to pay for a 'premium account' with more space and less advertisements.

PGP never was the overwhelming success the makers hoped for. Most people use unencrypted email, presumably because encryption is too cumbersome for those people. Maybe hushmail-like services can change this.

2.5 Overview of PGP versions

The following table gives an overview of the different PGP versions and their release date. Important events are also included.

Several websites give similar overviews, mostly concerned with the availability of RSA and DH support. The information is included in the features column. The table does not list all different

packages ever distributed. It is simply too much to list, and the differences are not very shocking. Looking at the version number will give the approximate release, and the code after the version number gives a hint about the features (noRSA version do not support RSA for instance). Small version number changes indicate bug fixes.

name	**date**	**legal status**	**maker**	**features**
PGP 1.0	June 5 1991	illegal	PPGS	Bass-o-Matic
PGP 1.4	January 19 1992	illegal	PPGS	
PGP 1.5	February 12 1992	illegal	PPGS	
PGP 1.6	February 24 1992	illegal	PPGS	
PGP 1.8	March 29 1992	illegal	PPGS	
PGP 1.8a	May 23 1992	illegal	PPGS	
PGP 2.0	September 2 1992	illegal	volunteers	IDEA
PGP 2.1	December 6 1992	illegal	volunteers	
	February 17 1993	criminal investigation starts		
PGP 2.2	March 6 1993	illegal	volunteers	
PGP 2.3	June 13 1993	illegal	volunteers	
PGP 2.3a	July 1 1993	illegal	volunteers	
PGP 2.4.x	November 6 1993	commercial	ViaCrypt	
PGP 2.5	May 5 1994	free	MIT	uses RSAREF library
PGP 2.6	May 22 1994	free	MIT	incompatible with previous
PGP 2.6.x	1994-1995	free/illegal	many	
	January 11 1996	investigation stops		
PGP 4.0	March 12 1996	commercial	ViaCrypt	
	March 21 1996	PGPPhone 1.0beta released by MIT		
PGP 4.5	February 4 1997	commercial	ViaCrypt	corporate key
	April 29 1997	Merkle-Hellman patent expires		
PGP 5.0	June 16 1997	free/commercial	PGP Inc	3DES DH CAST noRSA
PGP 5.5.3	November 26 1997	free/commercial	PGP Inc	ADK introduced (+bug)
	December 1 1997	PGP is sold to NAI		
PGP 5.5.5	December 4 1997	free/commercial	NAI	
PGP 6.0	September 1 1998	free/commercial	NAI	
PGP 6.02	November 12 1998	free/commercial	NAI	RSA support in all editions
PGP 6.5.1	April 5 1999	free/commercial	NAI	
	January 2000	export restriction lifted		
PGP 6.5.3	January 31 2000	free/commercial	NAI	
	August 2000	Ralph Senderek discovers ADK bug		
PGP 6.5.8	August 25 2000	free/commercial	NAI	ADK-bug fixed
	September 20 2000	RSA patent expires		
PGP 7.0	September 10 2000	free/commercial	NAI	no source code
PGP 7.0.3	February 2 2001	free/commercial	NAI	minor bugfixes
	February 19 2001	Zimmermann leaves NAI		

Table Notes

noRSA This means that certain editions of this version did not have full RSA support.

Chapter 3

Algorithms in PGP

3.1 Kinds of algorithms

Each cryptographic algorithm solves a certain problem or task. For each of these task there is often a handful of equivalent algorithms, so a programmer can choose what algorithm to use. PGP needs algorithms for the four tasks described below. PRZ originally choose one algorithm for each task, but the newest version of PGP supports a wide variety of algorithms to choose from. This fits the distributed philosophy of PGP: each user can use the algorithm he thinks is safe. Another good reason for this variety is the fact that not all algorithms can be used freely in all countries. It is also useful in cases where the algorithms have different characteristics.

The right to choose may be convenient for encoders, for decoders it can be hard (make that impossible) to decode an unknown algorithm. Users are forced to always upgrade to the newest version. NAI, the current maker of PGP, certainly would not mind if that is the result. In the future it might be an idea to include a one-line algorithm identifier that is a PERL implementation of the algorithm, so that older versions can also decrypt newer algorithms:-). Or more seriously, maybe the makers of PGP should not try to support all algorithms, like they seem to intend. It does not increase the security, because one weak algorithm and a tiny bit of bad luck can compromise the whole system. One cannot expect that the average PGP user can make a good choice between algorithms.

Block ciphers Symmetric encryption algorithms do `normal encryption`, where the same password or key is needed for encryption and decryption. `Block ciphers` are symmetric ciphers that operate on blocks of say 8 or 16 bytes at the same time. The key also has a fixed size (typically 16 bytes or 128 bits). Block ciphers are the working horses of cryptography: they are fast, easy to program, storage-space efficient (they do not expand the data), so they can handle bulk data. Conceptually they replace a large secret (the data) with a small secret (the key). It is up to advanced protocols to do something spectacular with the small secrets.

PK algorithms `Public Key algorithms` do the tricks with public and private keys. They can be slow, and are also more vulnerable to special kinds of attacks. Therefore these keys are only used to encrypt small pieces of random data, like randomly generated session keys of block ciphers and the outcome of hash functions. Public key algorithms can normally be used for encryption and signing.

Hash functions A `hash function` takes a variable length file and produces a short fixed size hash or `fingerprint` of that file (often 16 or 20 bytes). They resemble block ciphers because

they are optimized for processing bulk data, and they replace a large doubt (whether the file is the same) with a small doubt (whether the hash is the same). Again it is up to other algorithms to do something with the fingerprint.

Fingerprints have two important properties: an attacker can never find, for a given fingerprint, a file with that fingerprint, and an attacker cannot find two files with same fingerprint (cannot find in the practical sense. Theoretically they do exist, but because there are so many hash values an attacker will not have the time to generate enough fingerprints to find a match). Because of these properties it is enough to sign the hash of a file, instead of the whole file. Hash functions are also used to convert pass phrases (which vary in length) to keys for block ciphers (which have a fixed length), and to convert data with some randomness to data looking truly random. PGP can calculate a fingerprint for a public key. You can check that instead of the whole key when verifying a public key.

Secret sharing algorithms These algorithms are not needed for the basic functions of PGP, but can be used in the later versions. Secret sharing algorithms can split a private key in any number of blocks m , such that if you bring any n of these blocks together you have the private key again, but you cannot find this key with less than n blocks ($n \leq m$). Typical use is in a company with 3 executive officers, where any two of these three are together allowed to sign a contract.

3.2 Symmetric block ciphers

3.2.1 Bass-o-Matic

This block cipher, listed first here because it was the first one present in PGP, is no longer part of PGP. It is insecure. Phil Zimmermann designed this cipher himself, and two months after he released PGP 1.0 he and `Biham` figured out the weaknesses of Bass-o-Matic. It is hard to find this cipher in the literature: the bible of blockciphers, Applied Cryptography[18], does not even mention it. Luckily the source of PGP 1.0 is still available as the definitive reference on this algorithm. The name stems from a saturday night show sketch where a fish is prepared in a blender.

3.2.2 IDEA

`IDEA` started its life in 1991, under the name `IPES`(Improved Proposed Encryption Standard). In 1992 the name was changed to International Data Encryption Algorithm. Its creators are Xuejia `Lai` and James `Massey`. It is a 64 bits block cipher with a 128 bits key. It's design is based on mixing bitwise exclusive OR ($\oplus$), Addition modulo 2^{16} and multiplication modulo $2^{16}+1$ (the famous prime 65537). It is fast in software (all personal computer processor chips can do multiplication in a single instruction), but less efficient in hardware (the small processors used in chip cards do not have such instructions). IDEA is patented, and the patent is held by Ascom-Tech AG in Switzerland.

There are no practical attacks known against the complete IDEA algorithm. If you intend to attack PGP, you can forget breaking any of the block ciphers. This is not the weakest point.

Gory details

This section gives a detailed description of IDEA, for those people who want to know how a block cipher works. Reading this section is not needed to understand the rest of this paper.

- The key is expanded to give many 16 bit subkeys. For each of the 8 rounds there are six subkeys S1, S2, S3, S4, S5 and S6, and for the final output transformation we get another four subkeys S1, S2, S3 S4. Then the following steps happen in each of the eight rounds:
  - The 64 bits input of this round is divided in 4 16 bit numbers: X_1, X_2, X_3 and X_4.
  - $A = X_1 + S1$. This is addition modulo 2^{16}
  - $B = X_2 + S2$
  - $C = X_3 + S3$
  - $D = X_4 * S4$.This is multiplication modulo 65537. 0 multiplies like -1.
  - $E = A \oplus C$
  - $F = B \oplus D$
  - $G = E * S5$
  - $H = F + G$
  - $I = H * S6$
  - $J = G + I$
  - $K = A \oplus I$
  - $L = C \oplus I$
  - $M = B \oplus J$
  - $N = D \oplus J$
- the output of each round is K L M N. After each round except the last the two inner blocks are swapped, so the next round starts with $(X_1X_2X_3X_4)_{i+1} := (KMLN)_i$.
- after the eighth round a final output transformation is applied (with new subkeys S1, S2, S3 and S4):
  - $W = X1 * S1$
  - $X = X2 * S2$
  - $Y = X2 * S3$
  - $Z = X3 * S4$

What the subkeys are depends on whether you are encrypting or decrypting. If you are encrypting you just take for the first 8 subkeys the bits of the key K, divided in 16 bit groups, then you shift the key 25 bits to the left, take again 8 subkeys, shift 25 bits left, ... until you have all the subkeys. Table 3.2.2 summarizes this. K_{0-16} means taking bits 0 to 16 (exclusive) from the key.

If you are decrypting the subkeys are inverses of the encryption subkeys, but in a backwards order. The idea is that you try to do exactly the inverse operation of what happened at the encryption. Let us define E(1,2) or shorter E12 as the second encryption subkey from round 1, and -X as the additive inverse of X, and X' as the multiplicative inverse of X. Then table 3.2 gives the decryption subkeys.

IDEA is similar to other block ciphers, except for one thing: It does not have S-boxes. A `S-box` is a lookup table that acts as a very random function: It must not be linear or otherwise systematic. The idea is that XOR and addition diffuse the information, while the S-boxes do the real obscuring. Often the S-boxes determine the strength of the cipher against cryptanalytic attacks. IDEA uses the multiplication modulo 65537 as S-box. This is a very chaotic function, so they have a very large S-box that they do not have to store.

Table 3.1: IDEA encryption subkeys

round 1	K_{0-16}	K_{16-32}	K_{32-48}	K_{48-64}	K_{64-80}	K_{80-96}
round 2	K_{96-112}	$K_{112-128}$	K_{25-41}	K_{41-57}	K_{57-73}	K_{73-89}
round 3	K_{89-105}	$K_{105-121}$	$K_{121-128}$,K_{0-9}	K_{9-25}	K_{50-66}	K_{66-82}
round 4	K_{82-98}	K_{98-114}	$K_{114-128}$,K_{0-2}	K_{2-18}	K_{18-34}	K_{34-50}
round 5	K_{75-91}	K_{91-107}	$K_{107-123}$	$K_{123-128}$,K_{0-11}	K_{11-27}	K_{27-43}
round 6	K_{43-59}	K_{59-75}	$K_{100-116}$	$K_{116-128}$,K_{0-4}	K_{4-20}	K_{20-36}
round 7	K_{36-52}	K_{52-68}	K_{68-74}	K_{84-100}	$K_{125-128}$,K_{0-13}	K_{13-29}
round 8	K_{29-45}	K_{45-61}	K_{61-77}	K_{77-93}	K_{93-109}	$K_{109-125}$
output transf	K_{22-38}	K_{38-54}	K_{54-70}	K_{70-86}		

Table 3.2: IDEA decryption subkeys

round 1	E91'	-E92	-E93	E94'	E85	E86
round 2	E81'	-E83	-E82	E84'	E75	E76
round 3	E71'	-E73	-E72	E74'	E65	E66
round 4	E61'	-E63	-E62	E64'	E55	E56
round 5	E51'	-E53	-E52	E54'	E45	E46
round 6	E41'	-E43	-E42	E44'	E35	E36
round 7	E31'	-E33	-E32	E34'	E25	E26
round 8	E21'	-E23	-E22	E24'	E15	E16
final tr.	E11'	-E12	-E13	E14'		

3.2.3 CAST

Carlisle Adams and Stafford Tavares created the CAST blockcipher. They claim that the name refers to the design procedure, but it seems more likely that they deliberately named it after their initials. The exact variant of CAST used in PGP is CAST5-128 as defined in RFC 2144[1]. It is a Feistel cipher with 128 bit key and 16 rounds, and it operates on 64 bit blocks.

The original CAST description did not specify what S-boxes should be used. The RFC defines them. The document gives a description of CAST-128: The 128 bit variant of the algorithm. This variant can also be used with reduced key sizes. To differentiate these variants, CAST-128 is also called CAST5, and the actual key size can be postfixed. PGP uses the largest keys possible: 128 bits. Hence CAST5-128.

3.2.4 3DES

DES stands for Data Encryption Standard. This encryption algorithm was created in 1977 by the US government (NIST and NSA) based on work done by IBM. DES is a 64 bit blockcipher with a 64 bit key. Unfortunately, the last bit of every byte in the key is a parity bit, so effectively there is only a 56 bit key. This is way too short, a brute force attack with a dedicated DES cracking machine would take less than a day. Triple DES or 3DES is doing DES 3 times, with different keys. So triple DES is a 64 bit block cipher with 168 key bits (plus 24 parity bits). The original DES algorithm has been studied for a very long time, and therefore experts consider 3DES to be very secure. The drawback is that it is significantly slower than al other algorithms: DES on itself is slow because it uses bit permutations that are easy in dedicated chips but slow on general purpose computers, and because you need to do three operations to get the security of two operations. The only reason for using Triple DES is that it is very well studied. It is a cipher for conservative people.

3DES is the most interesting block cipher in PGP. Multiple encryption makes ciphers stronger, but it is not problem free. An attack on double encryption is described below. DES itself is also not as good as the other ciphers. There are attacks known against the 16 round DES encryption, making 3DES in theory the most doubtful cipher in PGP.

Differential Cryptanalysis

Differential cryptanalysis can find a DES key using a chosen plaintext attack with 2^{43} chosen plaintexts. A chosen plaintext attack is an attack where the attacker can submit plaintexts of his choice and receive the encryption. This is a very theoretical attack, because it is not feasible to collect that much plaintext. The S-boxes in DES are designed to be very strong against differential cryptanalysis, so one can conclude that the NSA knew about this technique back in 1977. The rest of the world heard about it in 1990 from Eli Biham and Adi Shamir.

Linear Cryptanalysis

Linear cryptanalysis is even more powerful against DES. It is a similar technique, but different, and discovered by Mitsuru Matsui in 1993. It can do a known plaintext attack against DES using 2^{43} plaintexts. For a known plaintext attack, the attacker has access to a certain amount of plaintext and the corresponding ciphertext, but cannot choose what the plaintext is. This attack is almost practical, although 2^{43} is still a lot. This attack does not generalize to 3DES, so it has no practical value against PGP, but it does show that a blockcipher need not be secure, even if it

is designed by the best cryptographers at that time[1].

The DES S-boxes are not chosen to be extra strong against linear cryptanalysis, so it can be assumed that the NSA did not know about this technique in 1977.

Both techniques are described in [18] page 285-293.

Attacks on multiple encryption

The effective strength of 3DES is 112 bits, not 168 bits. One would expect that this strength could be achieved by encrypting twice. This is not the case, because one can do a `meet in the middle attack` ([18] page 358). To do a brute force known plaintext attack on DES you would have to do 2^{56} encryptions. If double encryption is used, one needs to do 2^{57} encryptions and also memory for 2^{56} blocks.

For a known plaintext attack the attacker has C_1 and P_1 such that

$$C_1 = E_{k_2}(E_{k_1}(P_1)) \tag{3.1}$$

For each possible key k, he encrypts and stores $E_k(P_1)$. When finished, he tries to compute for all keys l the expression $D_l(C_1)$ and looks the result up in memory: If it finds a match $E_k(P_1) = D_l(C_1)$ then kl is a likely candidate for the key. Assuming that the attacker has a few more known plaintexts he can check whether it is correct, and try more l values if it is not.

Although the extra memory requirement of 2^{56} blocks is not completely trivial, it is clear that double encryption offers not much extra security. For triple encryption there is a similar meet-in-the middle attack, and this attack takes 2^{2n} steps (and 2^n memory blocks). The first triple encryption implementations had to be backward compatible with singe encryption, so they used encryption-decryption-encryption:$C = E_{k_3}(D_{k_2}(E_{k_1}(P)))$. This does not affect the security, but if you take $k_1 = k_2 = k_3$ it is equivalent to single encryption.

To complicate things there is also a triple encryption scheme that needs only two keys: $C = E_{k_1}(D_{k_2}(E_{k_1}(P)))$. This is not a bad scheme at all if you are short on key storage space, but it is not as secure as triple encryption. See [18] page 359.

PGP uses 3DES with a 168 bits key(real key size) in EDE mode. This gives an effective strength of 112 bit.

3.2.5 Rijndael/AES

In 2000 the blockcipher `Rijndael` was elected to be the new `Advanced Encryption Standard`. It replaces the DES standard. PGP 6.5.8 does not have AES support, but versions after PGP 7.0 do. The `AES` is a very new cipher, and no attacks are known.

Rijndael has been designed by Joan `Daemen` and Vincent `Rijmen`. It is an unorthodox design: It uses operations on polynomials, and decryption is not as fast as encryption. It has a block size of 128 bit (twice the size of the other ciphers) and a key size of 128, 192 or 256 bit. For more information check *http://csrc.nist.gov/encryption/aes/rijndael/*

3.2.6 Encryption modes

A block cipher can be used in different ways, called `encryption modes`. The naive way, called `Electronic Codebook Mode`(`ECB`), is to divide the input in blocks, and encrypt each block. This mode has disadvantages: If the input has repeating patterns (long runs of the same character for

[1] It is possible, some people would say likely, that the designers tried to make DES as weak as it is.

instance) they will be visible in the output. Another problem is that the same input file will give the same output each time. And last but not least: an attacker can make predictable changes to the data by exchanging or repeating blocks.

To overcome these problems three other modes exist: CBC, CFB and OFB. PGP uses `CFB` mode, so I will discuss that one. CFB stands for `Cipher Feedback Mode`. The CFB-mode used by PGP is designed by PRZ, and differs slightly from the standard CFB-mode. It is described in [3].

CFB

The CFB algorithm uses some extra storage, named a *feedback register*, whose size is equal to the block size of the cipher used (called a block of data. Often 8 bytes). To encrypt a block, it is simply XOR-ed with the contents of the register. If the 'block' you want to encrypt is not a complete block, but is shorter than the length of the feedback shift register, you can take the first bytes of the register. After using (part of) the register it is filled with the encryption of the last block of the output. If only a part has been used and it is not the end of the input, this is called a *sync*. Let C_i be the ith block of encrypted text or ciphertext, R the register, and P_i the ith block of plaintext.

$$C_i = P_i \oplus R; \qquad R = E(C_i) \tag{3.2}$$

To start encrypting, the register is filled with the encryption of the allzero block. Then an initial value (IV) consisting of one block of random bytes is encrypted in CFB mode: the output is discarded, but the register is now set. Then the first two bytes of the random block are encrypted again. This forces a sync. Now the data can be encrypted.

The non-standard part of PGP's CFB is that the registers initial value is not simply set, but obtained by processing random data, and the fact that it is allowed to process shorter incomplete blocks at any time: When encrypting multiple big numbers, a sync is done after each number. These small peculiarities do not affect the security. The repeatal of the first two random bytes provide some sort of checksum: it allows you to detect that you are using the wrong key. A nice property of this mode is that you do not have to pad the last bytes to make a full block.

Problems with CFB

A severe weakness of CFB is that one can make predictable changes to the last block of bytes: If you flip bits in the last part of the encrypted data, the corresponding bits of the decrypted plaintext also have changed. These changes will often be detected by PGP if the encrypted data is somehow signed. Sometimes the data is not signed: If you used PGP to password encrypt a file, it is not completely protected against changes. A secret keyring is also encrypted, but not signed. This is a severe problem because all cipher modes are designed to protect against changes, and it is a pity that this protection fails for the last block. The Klima-Rosa attack on page 66 exploits this.

A more theoretical disadvantage is that CFB is vulnerable for a chosen ciphertext attack described in [11]. In a chosen plaintext attack the attacker can have the decryption of any block of data, except the one it is attacking. This is quite a strong assumption, but in certain implementations it could be the case. The attacker can send in the original message, with arbitrary bytes instead of the last block, and from that decryption he can determine what the last block must be Xored with. This attack is not a serious threat for PGP, but could be interesting in other uses of CFB.

CBC

Another common mode is `CBC` or `Cipher Block Chaining mode`. PGP does not use it for encryption, but it is used inside the random number generator (see page 77). It also uses a register R. R always contains the last output. A plaintext block is encrypted by Xoring it first with R, and then encrypting it.

$$C_i = E(P_i \oplus R); \qquad R = C_i \tag{3.3}$$

This mode does not allow using a partial block at the end of the message. Any change made in the ciphertext will corrupt an entire block.

3.3 Public key algorithms

A `public key algorithm` or `PK` algorithm is an algorithm where keys come in pairs: one of the keys is the public key, and can be given to anyone. The other part of the key is the secret key. These keys are each other's inverses: The public key can 'undo' the private key operation (verification of a signature) and the private key can undo the operation of the public key. One further property is that one cannot derive the private key from the public one.

There are not many good Public Key algorithms. This seems surprising because there are many (NP-)hard problems known in computer science, and each of these problems should be usable to define a PK algorithm. Unfortunately in practice most of these hard problems have many easy instances, and you cannot risk basing your private key on an easy instance.

There are two problems that survived all attacks until now: Factoring and the discrete log problem. The RSA algorithm is based on factoring. There are a few RSA variants known, but there is only one original, and that is what is used in PGP.

For the discrete log problem things are more interesting. The discrete log problem is the problem finding x such that $g^x = y \pmod p$ for given p, g and y. In discrete-log public key schemes, y is the public key, and x is the secret key, but there are a few different ways of doing the encryption/decryption with these numbers. The discrete log problem can also be stated for different groups than ${\mathbb{Z}_p}^*$. For instance on polynomials, or things like elliptic curves. It is hard to say whether all of these are different Public Key algorithms, or just variations of one algorithm. The first discrete log based system was an interactive protocol designed by Diffie and Hellman. ElGamal is the common name for the class of discrete log systems, but the PGP program prefers the name Diffie-Hellman. They chose this name because it was better known, because `Cylink` asked for it when they asked whether they could include DH in PGP, and because the difference between the system designed by Diffie and Hellman and the system by ElGamal is not that big.

3.3.1 RSA

`RSA` is the most famous public key algorithm. It was used in the first version of PGP. RSA is based on number theory: It uses large prime numbers. It was invented by Ronald `Rivest`, Adi `Shamir` and Leonard `Adleman`, who first published the algorithm in April, 1977. This algorithm became popular despite patent problems.

Key generation

One always makes a key of a certain size. A bigger key will be slower, but it will also be safer against a factoring attack. There is an ongoing improvement in factoring techniques, so the recommended

keysize is slightly increasing. At the current moment it is 1024 bits. The size of a key is the size of the number n expressed in bits. The user chooses at random

- p a large prime number, half the size of n (say 512 bits). p is chosen as one of the the first primes next to a randomly chosen number, and starting with the bits 11, to ensure the size of n. It chooses not always the first prime number, but shuffles the 256 elements above the starting point so that primes above a large primeless gap do not have a too big chance of being chosen.
- q another prime number (again 512 bits). In PGP q is chosen not too close to p. q is larger than p (if not swap the numbers). It also starts with the bits 11.
- e any convenient number relatively prime to $p-1$ and to $q-1$. Small numbers with low Hamming weight are faster to calculate with, so common values are 3, 17 and 65537. 3 is a very risky value, see the section on low exponent attacks, so the new versions use 65537. If the value of e you want is not relatively prime to $p-1$ and $q-1$, choose new values for p and q.

The user now calculates

- $n = p*q$. The size of n is the size of the key. It will be the size of p plus the size of q, because they both start with 11 in PGP.
- $\phi(n) = (p-1)*(q-1)$. This means that $m^{\phi(n)} \bmod n = m$, for every m. If you check the sourcecode of PGP, you will see that they instead of $(p-1)*(q-1)$ they use $\phi(n) = \text{lcm}(p-1, q-1)$. This is slightly faster, because it is a smaller number, but it will give the same d, so it does not matter.
- $d = e^{-1} \bmod \phi(n)$.
- $u = p^{-1} \pmod q$

The public key consists of (n, e). The private key is (n, e, d, p, q, u).

Usage

- encryption of a short message x: Convert x to a large number m, $1 < m < n$ and calculate the encrypted text $C = m^e \pmod n$.
- signing a hash value h: convert h to a large number m, $1 < m < n$ and calculate $S = m^d \pmod n$.
- decryption: calculate $m = C^d \bmod n$.
- signature verification: check that $m = S^e \pmod n$.

The calculations like $S = m^d \bmod n$ can be done faster, if you possess the private key, by first calculating $s_1 = m^d \bmod p$ and $s_2 = m^d \bmod q$. This is faster because p and q are smaller than n. Now we have $S = s_1 + p*u*(s_2 - s_1) \bmod n$. In the literature this is called an application of the `Chinese Remainder Theorem`.

This description of RSA is a little longer than the one in most textbooks, because it incorporates certain details from inside PGP (like the extra variables in the private key).

There are many ways to convert messages x to large numbers. Not all of these are equally good. The method used by PGP is described in section 3.3.4.

Problems with RSA

RSA is widely believed to be very secure. No-one has broken this algorithm completely like the other early cryptosystems based on knapsack problems. The recommended key size has been very much increased because factoring methods have improved. It is possible to factor 512 bit numbers with current technology, so keys smaller than 768 bits are on the edge, and 1024 bit or larger keys are recommended. There are also some attacks on special cases of RSA. The following list gives ways of attacking RSA.

Factoring the key

The most aggressive way of attacking RSA is to try to factor n. If you find one of the factors of n, you can calculate the private key. Several smart ways to do this have been developed. The `Quadratic Sieve` (QS), described in [20] chapter 14 is a simple method. To factor n, it searches for a pair of numbers with $x^2 = y^2 \pmod{n}$. If this is the case then $(x-y)(x+y) = k*n$ for some k and there is a good chance that $\gcd(x-y,n)$ or $\gcd(x+y,n)$ is a factor of n. To find such a pair, many relations $z = k^2 \bmod n$ are collected, where z can be factored into small prime factors. These relations can be combined: If $v = l^2 \bmod n$ and $w = m^2 \bmod n$ then $vw = (lm)^2 \bmod n$. Using linear algebra one can search for a combination of relations where each small prime factor appears an even number of times. If this is the case then $z = \prod_i {p_i}^{2*e_i} = (\prod_i {p_i}^{e_i})^2$ and we have a pair of squares.

The QS runs subexponential and a variation of it factored a 428 bit number in 1994. The `Number Field Sieve` is an even faster algorithm.

Predictable message attack

If you send a short message from a small set of messages with RSA, and you do not pad it in a special way, but you just add zeroes to the beginning (that is what can be considered the natural way of padding), an attacker can try to encrypt all possible messages until it finds a matching one. To prevent this attack always use a message-to-number conversion that adds at least 80 random bits.

Low exponent attack 1

Many people use the value $e = 3$, because this small value makes encryption and signature verification faster. If this is the case and the message is less than one third of the size of n, finding the message is easy. The encrypted text is $C = m^3 \bmod n = m^3$ because $m^e < n$. In this case one can find the value of C by using a binary search method. This attack is only possible with the most naive implementation of RSA. It is one of the motivations behind using sophisticated padding schemes.

Low exponent attack 2

If someone encrypts the same message m to different people (with different public keys) and they all use $e = 3$, an attacker can again find the value m^3 by using three different encryptions. He can use the Chinese Remainder Theorem to obtain m from the equations

$$c_1 = m^3 \bmod n_1 \tag{3.4}$$

$$c_2 = m^3 \bmod n_2 \tag{3.5}$$

$$c_3 = m^3 \bmod n_3 \tag{3.6}$$

To prevent this attack never encrypt the same number twice. Always include some fresh random bytes into every operation. This attack also works against other low values of e, but the attackers always needs e encryptions to e different people to apply this method, so this attack is harder to do if $e = 65537$.

Low exponent attack 3

The low exponent attack described above can be generalized: if one has encryptions that are related in a certain way, they form a set of equations that might be easily solvable. As an example of an attack like this, suppose $e = 3$ and you encrypt the same message to seven different people. To prevent the attack above, you take m_i is m followed by a timestamp unique for each of these people. Mathematically: $m_i = m * 2^{32} + t_i$. If the values t_i are known by an attacker he can solve the set of these seven equations. The method used is described in [8]. To prevent this attack, always use secret, random bits in the padding. One can also consider not using $e = 3$ because it seems much easier to attack than larger values.

3.3.2 ElGamal/Diffie Hellman

This algorithm, named DH in PGP but ElGamal in literature, is only used for encryption, and not for signing in PGP 5.0. This is because the signatures are quite large. ElGamal signing is possible and secure, and will most likely be available in the future, but the signature will consist of two numbers of at least 1024 bit, while RSA takes only one 1024 bit number, and DSA only takes two 160 bit numbers.

Key generation

- A prime p is chosen. The size of p determines the size of the key. A typical size is 1024 bits.
- Find a generator g of a large enough subgroup of prime order (g is said to generate the set of numbers $g^i \bmod p$). Large enough means at least 160 bit. You are looking for a g, $1 < g < p$, with $g^q \bmod p = 1$ for a prime number q. This is only possible for a q dividing $p - 1$.
- Find a secret number x. The size of x is chosen to be a small as it can be securely to speed up computations. The choice PGP makes is based on a paper from Michael Wiener. For a 2048 bit key, the size of x is 225 bits. The complete table of x sizes is listed in `pgpKeyMisc.c`. It should at least be 160 bits, according to the security assumptions in this paper.
- y is calculated: $y = g^x \bmod p$

The public key is (p, g, y). The private key is (p, g, y, x). There is more than one way to do the above. It is possible to search for $p = 2 * q + 1$ with q also prime. In this case $g = 2$ will do. p is called a `Sophie Germain prime`. This gives a very strong ElGamal key, but it takes a long time to find such a prime, because they are rare. It is safe to share p and g with a group of users, so PGP can use some precomputed values. This option is called *fastgen*. A third way is to take a random prime q, a few bits shorter than the requested length of p (say 10 to 20 bits shorter). A few values of k are tried to find a p such that $p = 2 * k * q + 1$. Now it is checked that $2^{\frac{p-1}{q}}$ is not 1. If this is the case, $g = 2$ is a good choice, otherwise another value for k is tried.

Encryption

m is the message padded in the PKCS way described in the section [10], to make it a positive number smaller than p. The sender selects randomly a number k, $0 < k < p-1$. He computes $a = g^k \bmod p$ and $z = m * y^k \bmod p$. The encrypted message is (a, z).

Decryption

$m = (a^x)^{-1} * z \bmod p$.

Security of ElGamal

This algorithm seems very secure. There are no 'low generator attacks', so the only way of attacking seems solving the Discrete Log problem. There is a close relation between solving the DL problem and factoring. In fact one can say it is equally hard to solve the DL problem in ${\mathbb{Z}_p}^*$ as factoring a composite number the same size as p.

DL-algorithms can be divided in group-theoretic algorithms, which work on all the ElGamal variants, even the ones on different groups, and the ones specific for ${\mathbb{Z}_p}^*$. A specific algorithm is the `index calculus`, which is the DL variant of the quadratic sieve. It is described in [20].

A group-theoretic algorithm is Shanks' `baby step -- giant step algorithm`. Let $m = \sqrt{\text{upperbound}(x)}$. You want to know x with $x = g^y$ in a certain group. Write $x = x_1 * t - x_2$ for $x_1, x_2 \leq m$. It holds that $y * g^{x_2} = (g^t)^{x_1}$. Create a table T that holds for all values of x_1, $(g^t)^{x_1}$. Next try calculating $y * g^{x_2}$ for all values of x_2 and look up the results in T. If you have a match, you know x_1 and x_2, and thus you know x.

The running time of this algorithm is $\sqrt{\text{upperbound}(x)}$. This is the best a group-theoretic algorithm can do, so PGP is safe from these attacks, because it would mean doing more than 2^{80} steps.

If you want to solve the DL problem in ${\mathbb{Z}_p}^*$, and you know something about the factorisation of the order of g, for instance $\text{ord}(g) = a * b * c$, you can solve the DL problem in a time depending on the largest factor of $\text{ord}(g)$. For this reason one should always be sure that $\text{ord}(g)$ contains at least one large prime factor. The algorithm to do this is called `Pohlig-Hellman`.

3.3.3 DSA

The Digital Signature Algorithm, part of the digital signature standard (DSS) , is a US government standard for signing documents. It can only be used for signing. The main advantage over RSA and ElGamal signature schemes is that it produces much smaller signatures. The DSS document, [15] describing the DSA was published in 1994.

The DSA is a special variant of ElGamal, called subgroup-ElGamal. This version of ElGamal needs a large group for doing the calculations in, 1024 bits, but the signature numbers are chosen from a smaller subgroup (2^{160} elements), and the designers have succeeded in making a protocol where only 160 bits are needed to denote these elements of the subgroup. This makes DSA a variant of ElGamal that generates quite small signatures: It takes two 160 bit numbers instead of two 1024 bit numbers.

The DSA has been created by the National Institute of Standards and Technolgy (`NIST`), but they got help from the NSA. The standard got much criticism, because people were getting used to RSA, and now the government tried them to make them switch to DSA. It was also criticised for just being a signature standard, not being an encryption standard. While it is probably true

that the NSA would not help people encrypting data, this does not make DSA a worse signature standard.

The official key size of DSA varies from 512 to 1024 bit. This is not much: 512 bit is certainly insufficient, and 1024 bit is the only reasonable value, so it will not last forever. The reasoning behind the small key size is that powerful government adversaries, like the NSA will try to break encryption, but not try to forge signatures: Once they start using this ability people will notice it and it will not remain a secret that they can do it. So you need not fear governments but only less powerful adversaries. Anyway, PGP always makes 1024 bits DSA keys, and this is secure enough.

Key generation

- Select the 160 bit prime number q
- Select the 1024 bit prime number p such that $q|(p-1)$.
- Find a and g such that $g = a^{\frac{p-1}{q}} \bmod p$ with $g \neq 1$.
- Select randomly a number x , $0 < x < q$.
- Compute $y = g^x \bmod p$

The public key is (y, p, q, g), the private key is x.

Signature generation

- Select randomly a number k, $0 < k < q$.
- Compute $k_2 = k^{-1} \bmod q$.
- Calculate $s = (k_2 * (H(m) + x * r)) \bmod q$. $H(m)$ is the hash of the message.

(r, s) is the signature.

Signature verification

- Verify that $0 < r < q$ and $0 < s < q$
- Calculate $t = s^{-1} \bmod q$
- Calculate $u_1 = t * H(m) \bmod q$
- Calculate $u_2 = t * r \bmod q$
- Verify that $r = (g^{u_1} * y^{u_2} \bmod p) \bmod q$

Note that unlike RSA signatures, you cannot recover the hash value from this signature. This is caused by the fact that this is subgroup ElGamal: the numbers given contain enough information to do a check, but not to reconstruct the input.

The signature does not contain information about the hash function used. The official standard specifies that SHA is used, but PGP specifies that any 160 bit hash function can be used. Theoretically this could be a problem, if a weak hash algorithm is included in PGP (see section 4.4.1).

3.3.4 Padding

Padding is the process of converting messages into the big numbers algorithms like RSA and ElGamal need. Often the data must be expanded, because the message is shorter than the number needed. This padding is vital to the security, because most public key algorithms are not immune to attacks with specially chosen numbers.

The padding described here is specially designed for use with RSA. It is also used for ElGamal. It is described in [10]. The document appears in the `PKCS` series, a series of Public Key Cryptography Standards maintained by RSA security. It is described in document # 1, and the version of this document used is version 1.5. This documnet is also a RFC, number 2313 [10]. There is a newer version, RFC 2437.

A string of the right number of bytes is formed (this is size(n)/8, rounded upwards) by concatenating the parts 00, BT, PS, 00, M. Hexadecimal notation is assumed, so 00 means a null byte. The parts have the following meaning:

- BT is a one byte block type. It is 01 in a signing operation, or 02 when encrypting. The value 00 is also mentioned is the RFC, but it is not recommended and not used in PGP.
- PS is the padding string. It has variable length, and consists of many FF bytes if BT is 01, or random nonzero bytes if BT is 02. In the latter case they must be really random not pseudorandom.
- M is the message you want to include. In PGP, it is the session key appended with a 2 byte checksum, or the hash with a certain prefix to identify the hash algorith(see page pagerefhashprefix). This checksum is calculated by adding all message bytes modulo 65536.

Applications should check the checksum, and the FF's in the padding string. The reason the encryption has random bytes and the signing fixed bytes, is that when encrypting, you want to prevent that attackers guess the number you encrypted, while this does not matter while signing. For signing, you want to prevent that people can verify random numbers until something comes out that looks like a signature: An attacker might want to do this to forge a signature. He cannot show the document in this case, but in some applications even this kind of forgery is bad.

One small potential weakness is that the signing operation contains no random padding. If something goes wrong during a signing operation, the attackers exactly know what the input was, and can compare the numbers to obtain information on the secret key. I recommend inserting 10 random bytes in the message when using this kind of padding in a PK algorithm like RSA, where the message can be recovered from the signature. It can help avoid the attack on page 37

3.4 Hash functions

3.4.1 MD5

MD5 is the fifth Message Digest algorithm from a series defined by Ron `Rivest`. MD5 is designed around a `compression function`: a compression function takes a fixed length input and returns a fixed length shorter output. Other members of the family are MD2 (designed to work on 8 bit processors) and `MD4` (a message digest algorithm designed on the edge of security. MD5 is just slightly slower but much more secure. MD4 is broken). MD5 is defined in RFC 1321 [17]. It produces a 128 bit hash.

To hash a certain input, the input is padded with zeroes to make it 64 bits short of a multiple of 512 bits. Then the total length is appended as a 64 bit number. Four 32 bit variables A, B, C and D are initialized with special numbers defined in the standard, and each 512-bit input block and the current values A B C and D are fed into the compression function, and the result, 128 bits large, is split into 4 32 bit numbers, and those numbers are added to A, B, C and D respectively. If all blocks are processed, A, B, C and D form the output.

In the key size section (page 12) it is explained that 128 bits is not enough for a hash function against the birthday attack described below. For MD5 things are even worse: Cryptanalysists have specific attacks against MD5 that show a weakness in the compression function.

Rivest intended to design a collision-free compression function for MD5. He did not succeed. Hans `Dobbertin` was able to construct a collision in the compression function (see [5]). It took 10 hours of calculating time on one Pentium 90 computer. A `collision` for the compressing function is a triple $(ABCD, x, y)$ such that

$$\text{compress}(ABCD, x) = \text{compress}(ABCD, y) \tag{3.7}$$

It is not a collision of MD5 itself because one does not have a document prefix that will bring $ABCD$ to the required value. This collision is therefore not an immediate threat for PGP.

The short report in which the collision is presented does not show how the collision has been calculated. I presume some MD5-specific techniques are used, in a very clever way. Nevertheless I would like to point out one can expect that finding a collision in the compression function is easier than finding a collision in the hash function.

If you want to find 1 document having a particular hash, you are looking for a 2^{-128} event , while you can choose one variable: the document. This will take 2^{127} time steps on average.

If you are looking for two documents x and y with the same hash, you have two variables (x and y), and by just starting to hash random documents you will find such pair in only 2^{64} steps, because you have two points floating around in the 128 bit search space instead of one. That is a lot faster.

For finding a compression function collision one has three variables: x, y and the state $ABCD$. If, and this is a big if, you know some way to make use of the extra freedom offered by $ABCD$, it seems that you only need $2^{128/3} = 2^{43}$ hashes. If a Pentium would do one million compressions per second, one can make 2^{38} steps in 10 hours. The difference between these two is just a factor 32.

The reasoning above is highly hypothetic, and does not tell you how to search. I just want to point out that it is not a complete surprise that some-one found such a collision as Dobbertin did. A 128 bit hash is simply too small, considering the speed of today's computers. MD5 is no longer recommended, and one of the weak points of PGP. Although no one has broken it in practice, one can expect it will happen.

3.4.2 SHA

The `Secure Hash Algorithm` is designed by NIST and the NSA. Its design is based on MD4, just like MD5. `SHA` produces a 160 bit hash. SHA is used in the DSS standard. It is defined in the Secure Hash Standard, FIPS PUB 180[16].

The SHA version that anyone uses is SHA-1. SHA-1 is a modified version of the original SHA. Just a few details were changed, presumably to make it stronger against certain attacks. PGP also uses the modified version SHA-1.

3.4.3 RIPEMD160

RIPEMD-160 is a 160-bit hash function, designed by Hans `Dobbertin`, Antoon `Bosselaers`, and Bart `Preneel`. It is intended to be used as a secure replacement for the 128-bit hash function RIPEMD. RIPE stands for an EU project named RACE Integrity Primitives Evaluation, 1988-1992. It was originally published in [4].

3.4.4 Security

Birthday attack

The best attack that works against any hash function is the `birthday attack`. This attack is based on the remarkable fact that it only takes a group of 23 people on average to find two people with the same birthday. In this attack Eve needs a signature of Bob under a document that Bob would never sign. To do this, she lets Bob sign the hash of an innocent document. Then she distributes the signature together with the evil document, and if it has the same hash, it will convince people that Bob signed the evil document.

Eve generates a whole series of almost identical innocent documents, and stores all the hashes. The documents differ in very small details: extra spaces, insignificant words changed, spelling errors, etc. She also creates a series of almost identical evil documents, and tests if the hashes match any of the innocent hashes stored. She makes both series $L = 2^{S_{real}(H)/2}$ hashes long. Now the birthday effect will take place. The probability that a certain evil message does not collide with any innocent one is $\frac{L^2-L}{L^2} = \frac{L-1}{L}$. Quite big. However, all evil messages are generated independently after another, so the probability that none of the evil messages collides with a good one is $\frac{L-1}{L}^L \sim 0.36$ [2]. It is thus fairly likely that you have found a collision. If not, try again. This attack shows that $S_{eff}(H) \leq S_{real}(H)/2$.

Based on the 80 bit security definition and the birthday attack, a fingerprint must be at least 160 bits long. Both RIPEMD160 and SHA-1 have this length. There are no specific attacks known that weaken their security, so both are considered safe. In fact they are both quite similar, so I cannot say which is better.

3.5 Secret sharing algorithms

Secret sharing algorithms can split information in shares, in such a way that if you do not have enough shares, you know knothing about the information, but if you do have enough shares, you can reconstruct it.

The secret sharing algorithm of PGP is called `LaGrange polynomial interpolation` (another name is `Blakely-Shamir splitting`), and it works byte by byte. It considers each byte to be in the finite field $\mathbb{F}_{256}$. For each byte y_0 to be shared over players 1 to n, it constructs a polynomial P that passes through the points (x_0, y_0),$(1, y_1)$,$(2, y_2)$, ..., (n, y_n). Let m be the number of people that should be able to reconstruct the secret. y_1 , ..., y_m are chosen at random, and the rest of the y follow from the fact that P must be a polynomial of degree m. The maximum number of shares is 255, but that seems to be enough.

[2] try a few values to see this, for instance L=100, 200,...

3.5.1 Problems

There is nothing wrong with the algorithm described above. Assuming that the random number generator is good, it gives `perfect secrecy`: Without enough shares, one can say nothing about the value of the secret: It can still become anything, depending on the missing shares. This is probably the reason that there is only one secret sharing algorithm in PGP.

The only problem with this kind of secret sharing is that it is used to split private keys. A key split that way must be reconstructed before use: If the shareholders want to sign something, they must get together, and put all their shares, and their pass phrases into one computer. This poses a serious security risk: They must carefully check that the information is later destroyed. If they decide to do the reconstruction on the computer of one of the shareholders, he can easily modify his PGP version to sneakily store the shares and pass phrases for later use.

Better schemes exist where a key can be split in parts, and used without having to reconstruct the key. For RSA and $m = n$, a simple scheme is to find shares $S_1, S_2, \ldots, S_n$ with $S_1+S_2+\ldots+S_n = e \bmod \phi(n)$. To calculate $X = M^e \bmod \phi(n)$ all players do $X_i = M^{S_i} \bmod \phi(n)$ and they all together reconstruct $X = X_1 * X_2 * \ldots * X_n \bmod \phi(n)$.

More advanced schemes exist to create threshold schemes, but the idea is the same: The algorithm used in PGP is only suited for secret data sharing. For secret key sharing one should use a PK-algorithm specific solution that allows one to use the shares without reconstructing the secret.

Chapter 4

The PGP file format

A `file format` is a set of rules that describe how to build files, what is allowed and what not, and if it is allowed, what it means. The format for all files used and produced by PGP is described in a document called *OpenPGP Message Format* [3]. The name OpenPGP refers to the file format standard that PGP conforms to, and that can also be used by other programs. The document is an internet `RFC`: `Request For Comments`. The RFC series is a series of public electronic documents giving information on internet related technology. They can be found at the site *www.ietf.org*. The documents have no official status, they are just request for other people to comment on, but in practice they can be considered the ultimate authority on the subject.

This file format is used for several kinds of files. For PGP-encrypted messages, (.pgp files), for standalone signatures (.sig files), for exported keys (.asc files[1]) and for the public keyring file(.pkr) and the private keyring (.skr). The last two files, the public keyring and the private keyring file, are two files that are created during the installation of PGP. Their default name is `pubring.pkr` and `secring.skr`. The secret keyring contains all private keys, the pubring the public keys and the trust information. These files are internal for PGP, so it is not immediately clear why they should conform to a standard. The reason they do is that one can upgrade the PGP installation without losing your old keys, and that OpenPGP was written afterwards: it was not an entirely new standard, but a formalization of the status quo, with some new features.

The file format is interesting for two reasons: First of all, it could be that the file format specification contains a design error that leaks information or makes them easy to manipulate. Secondly, by comparing what should be produced and accepted by what PGP actually creates and accepts, one can check whether PGP works.

4.1 Basics

A `file` is a finite length sequence of bytes. A `byte` is 8 bits of information and has a numerical value between 0(included) and 256(excluded). The OpenPGP standard uses the word octet instead of byte. Bits can have the value 0 or 1. The first bit of a byte, bit 0, is the most significant one, bit 7 the least significant. The default encoding in OpenPGP is big-Endian. That means that if a number is encoded using multiple bytes, the first byte is the most significant one.

The value of a byte is given as a decimal number, unless it appears grouped into two characters at a time: In that case it is a hexadecimal value. For instance, 70 on itself is ambiguous, 00 70 is

[1] .asc can also be used for other (ascii-armored) files

hexadecimal, 3 is decimal, 03 is hexadecimal. Occasionally I will prefix hexadecimal numbers with 0x to make it extra clear that they are hexadecimal.

The phrase `old format` that appears frequently in the standard refers to the details in OpenPGP that exist for compatibility with PGP versions older than 5.0 (that have there own RFC[2]). This format is also called version 3, because in the packets the old format is indicated with a 03 byte-value. This numbering has nothing to do with the official PGP versions numbering. The new format is called version 4. All versions above PGP 5.0 can handle both versions. The new version is recommended because it is safer, but the old version is more compatible.

A big integer is encoded in the following way: The first two bytes contain the size in bits of the number. Then the number follows, in bigendian order. If the size is not divisible by 8, the third byte will start with a few zero bits.

A time field is 4 bytes long, and contains the number of seconds since the beginning of January 1, 1970. A range *a*-*b* means greater than or equal to *a* and strictly less than *b*.

4.2 Packets

The data in a file is divided in packets. Each packet consists of the data it contains, prefixed with a header indicating what kind of packet it is, and the length of the packet. The packet kinds are:

1 Public-Key Encrypted Session Key Packet

2 Signature Packet

3 Symmetric-Key Encrypted Session Key Packet

4 One-Pass Signature Packet

5 Secret Key Packet

6 Public Key Packet

7 Secret Subkey Packet

8 Compressed Data Packet

9 Symmetrically Encrypted Data Packet

10 Marker Packet

11 Literal Data Packet

12 Trust Packet

13 User ID Packet

14 Public Subkey Packet

60-64 Private or Experimental Values

4.2.1 Packet header

The packet header describes the type and the length of the packet. All packets start with a packet header. An old packet header can be 1,2, 3 or 5 bytes long. The first bit is always 1. The second bit is 0 if the old format is used.

In the old format the next four bits contain the packet type (so the type is between 0 and 16), and the last two bits tell how the length is encoded. Possible values are:

0 The header is two bytes, and the next byte contains the length. This length type can be used for packets where the data is less than 256 bytes.

1 The header is 3 bytes long, and the next two bytes encode the length. This length type can be used for data with a size less than 65536 bytes.

2 The header is 5 bytes long, and the next four bytes encode the length. This length type can be used for data with a size less than 4 gigabytes.

3 The packet length cannot be determined from the header The header is exactly one byte. The length of the packet must be determined 'from the context'.

For new packets, the second bit is 1, and the remaining 6 bits of the first byte of the header indicate the packet type. The header can be 2, 3, 5 bytes long. Which length is appropriate depends on the value of the second byte of the header.

0-192 The data is as long as this byte indicates.

192-224 The data has length $(byte_1 - 192) * 2^8 + byte_2$. (Remember that $byte_1$ is the second byte). This can be used if the data size is in the range 192 to 8384.

224-255 This indicates that the length of the data is not completely known yet. The length of the first part of the data is $2^{byte_1 - 224}$. After this first part a new length indicator(header without first byte) must follow, and the rest of the file(possibly using this length type repeatedly). The last part must have a different length type. This length specification is called *partial body length*.

255 This indicates that the length is encoded in the next four bytes. This length type can be used for any data size less than 4 Gb.

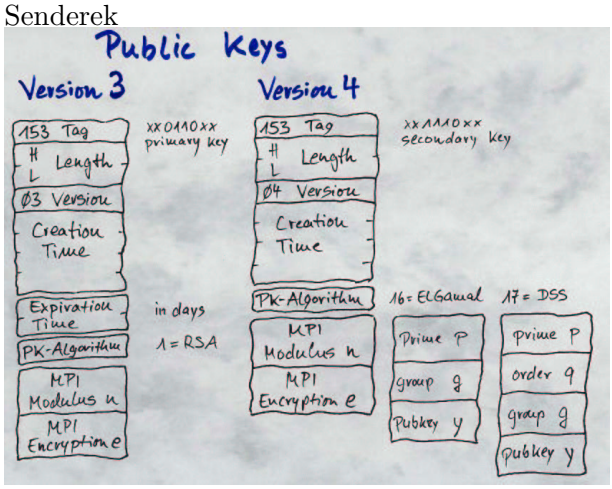
An important point of this design is that one can recognize the type of packets. This means that PGP does not need to know whether it is reading a key or a signature. I have tried to confuse PGP by feeding it key files that have the extension ".sig". It turns out that PGP completely relies on the packet types. All other information is ignored. Another good point of this encoding is that PGP knows in advance the length for the upcoming data (most of the times). This allows PGP to create buffers of exactly the right size. In cases where programs need to read until the end of the line, hackers often try to make huge lines so that their data is copied to a location that is beyond the end of the buffer.

4.3 Key packets

4.3.1 Public key packets

Public key packets contain the public information of a key. They do not contain information about the owner, so it is typically found combined with id packets and signatures.

Figure 4.1: PGP keys according to Ralph Senderek



4.3.2 Version 3

A version 3 public key packet consist of:

> 03 (=version indicator), [time] (4 bytes), [number of days it is valid, 2 bytes, 0 means always valid], [PK algorithm, 1 byte [2]], [algorithm dependent big numbers]

The algorithm specifier must be RSA, because RSA is the only supported PK algorithm in PGP versions before 5.0. I admire the designers' open eye for the future that made them include an algorithm specifier, but it is a futile feature since the file format is deprecated.

4.3.3 Version 4

> 04 (=version indicator), [time, 4 bytes] , [PK algorithm, 1 byte], [big numbers]

For RSA, the big numbers are (n, e). For DSA it is (p, q, g, y). For ElGamal it's (p, g, y). There is no information about the expiration of this key. That information must be stored in signatures attached to the key. The expiration date in the self signature can be seen as the key expiration date, but note that in this new format the signer can re-sign his key to extend its lifetime without losing the signatures by other people.

4.3.4 Secret key packets

OpenPGP secret key packets are quite identical to their public counterparts, except for the different type in the packet header, and the extra big numbers at the end.

A secret key packet consists of

> [All fields of the corresponding public key], [1 byte setting the string to key procedure], [optional: a symmetric encryption algorithm, 1 byte], [optional: a string to key specifier], [if encrypted: 8 byte initial value], [encrypted big numbers], [2 byte checksum]

The big numbers are (d, p, q, u) for RSA, and x for ElGamal or DSA. The [1 byte setting the string to key procedure] can be a symmetric key algorithm specifier. In that case the pass phrase must be converted to a key by using MD5, and the optional fields are not present. If the byte has the value 255, the optional fields are present. The given block cipher is used, and the given string to key specifier is applied. If the byte is 0, the secret fields are not encrypted, and there is no initial value present.

[2] see page 62 for the actual values of algorithm specifiers

For version 4 keys, the bytes of all secret numbers and the checksum are encrypted using CFB, with the given initial value. The checksum is the sum modulo 65536 of the bytes of the secret numbers. The checksum is used to determine whether the passphrase is correct.

For version 3 keys, the first two bytes of each secret number are not encrypted (the first two bytes contain the length of the number). The rest of the secret number data is encrypted using CFB (see page 35). A sync is done at the start of the next secret number. The checksum is also stored unencrypted, and is calculated the same way as in V4.

The reason version 3 keys do not have their length encrypted, is probably that it is convenient for programmers to know the length beforehand. The reason the V4 format does encrypt it, is that this information might help an attacker. The OpenPGP authors disrecommend using the old format unless you are forced to do for compatibility reasons.

4.3.5 Subkeys

In the first version of PGP all users had one key, a RSA key, that was used for both encryption and signing. Since PGP 5.0 things can be different: Every user has one signing key. This key has a couple of user ID's attached, and a few subkeys. A subkey is used for encryption, and can be revoked independently from his super key. It is signed by its superkey. If the police ever forces you to hand over your key because they read your email, you can give them the relevant subkey instead of a key they can also sign with.

4.4 Signature packets

In PGP a handful of different signatures are used. Certification signatures are made on public keys to assert that the key and the user ID belong together. A special case of this is a `self signature`, that is made on its own key. Signatures can also be made on a message or a file. The last possibility is that the signature does not depend on any data. This is for instance the case with revocation signatures.

The meaning of a signature is determined by its type. The following types exist:

00 : Signature on a binary document: The signer asserts that he owns, made or saw the file, or that it was not modified.

01 : Signature on a text document. The difference between this type and the first is that this signature is made on a file with the line endings converted to [carriage return][line feed] and trailing spaces removed. This is useful because it means that the signature remains valid if the document is sent to a different computer platform with different line ending rules.

02 : Standalone signature: this signature only signs its subpackets. This is useful for version 4 signatures because they can contain extra information.

10 : Generic certification of a public key and a user ID. It does not state how thorough the signer checked that it is correct.

11,12,13 : These stand respectively for *persona*, *casual* and *positive* verification of the user ID and key. They do state something about the effort the signer did to ensure it is right: persona verification means that the signer did not do any verification, casual stands for some casual verification and positive stands for substantial verification. This is all very vague. According to the OpenPGP standard, this vagueness is not a flaw, but a feature of the system, but

I do not see the use of fine grained claims with no precise meaning. All PGP certification signatures so far are of type 10.

18 : Subkey binding signature. It states that this subkey belongs to the topkey. It is calculated on the subkey itself, not on any user ID.

1F : Signature on a key. It is useful if the signature contains subpackets that give information on the key. For those to be valid, the signature must be made by the key itself. Other people can use this signature to make statements (using subpackets) about the key.

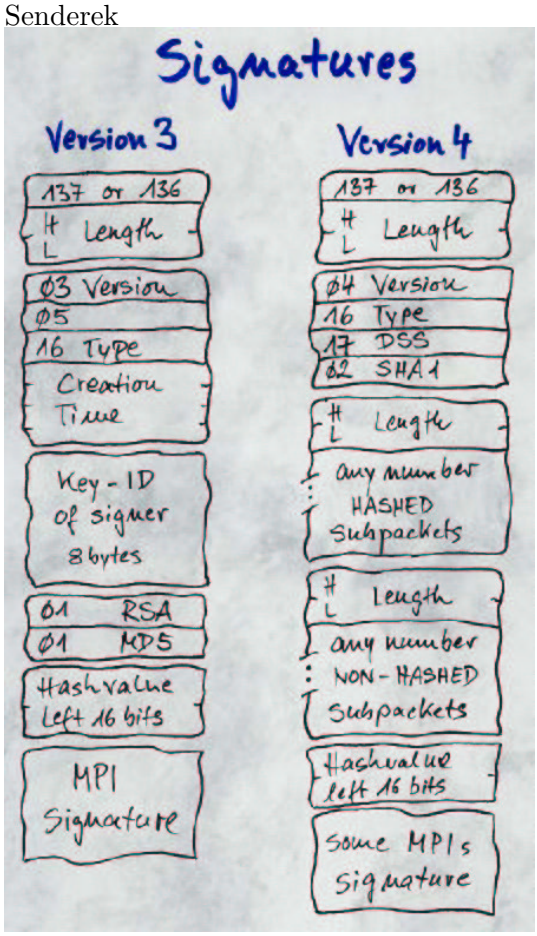
20 : Key revocation signature. The signature is calculated on a public key that is being revoked. It can be made by the key itself, or a key designated as a revocation key.

30 : Certification revocation signature. This signature can revoke a certification signature of type 10-13. It can be made by the key of the signature, or a revocation key of that key.

40 : A timestamp signature. This signature only states the time that is in it.

When a signature is made on data containing the signing key, it is called a `self signature`. They can be of type 10,11,12,13,18 and 1F. Self signatures are useful because they assure that the binding they make (between a user ID and a key for types 10,11,12,13, or between a key and a subkey in case 18) was made by the owner of a key.

Figure 4.2: signatures according to Senderek



4.4.1 version 3

A version 3 signature consists of

> 03 (=version indicator), 05 (=number of bytes to be hashed along with the data), [1 byte signature type], [time], [key id of signer], [PK algorithm], [hash algorithm], [first two bytes of hash], [big numbers of signature].

The time is encoded in four bytes that represent the number of seconds since the start of January 1 1970. The signature type is defined in a table. The key id of the signer is 8 bytes long. The PK algorithm and hash algorithm specifier are one byte each, and are in table 4.2. The input of the hash function is the signed data, the signature type and the time. The first two bytes of the hash are included so that PGP can see that it tries to verify a signature on the wrong document before doing an expensive public key operation. The key id is used to find the right key in the key ring.

Table 4.1: Hash prefixes

hashfunction	prefixes
MD2	30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 **02** 05 00 04 10
MD5	30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 **05** 05 00 04 10
RIPEMD-160	30 21 30 09 06 05 2b 24 03 02 01 05 00 04 14
SHA-1	03 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14

RSA

In this case the signature contains one big integer $m^d \bmod n$. The integer m is formed by prefixing the hash with a hashfunction specific prefix from table 4.1.

The result is padded according to subsection 3.3.4. The prefixes are based on some sort of formal encoding scheme, explained in [10].

Why use hash prefixes?

The reason that there is an algorithm-specific prefix inside the RSA encoding is to prevent that some-one lifts a signature to another document by changing the hash function. Suppose we have two different hash functions, one strong and one weak. You always use the strong one when signing your documents. I can find one of your signatures, change the hash function indicator to the weak algorithm, and find an evil document with the same hash (this should not be to hard because we are breaking a weak hash function). This is not possible if your signature contains information about which hash function should be used: To do this attack on a classic RSA signature one must also break RSA.

DSA

For DSA two big integers are added: r and s. The input of the signature algorithm is the output of a hash function with 160 bit output. There is no further processing. This means that DSA signatures are vulnerable for the hash-algorithm change attack described above. At the moment none of the 160 bit hash functions in PGP are considered weak, but once one is broken it makes all signatures by that algorithm invalid, even if they come from a trusted person.

4.4.2 Version 4

A version 4 signature consists of

> 04 (=version number), [signature type], [PK algorithm], [hash algorithm], [total length of the following subpackets, encoded in two bytes], [a few subpackets that will be hashed], [total length of the following subpackets, encoded in two bytes], [a few subpackets that will not be hashed], [first two bytes of hash value], [big integers of signature].

The signed data is being hashed, and everything from the version number to and including the hashed subpackets.

4.4.3 Version 4 subpackets

Signature subpackets were created to make a more flexible signature scheme: Many options and preferences can be specified this way. The header of a subpacket consists of a length specifier like a version 4 length specifier. The use of 'partial body length' is not allowed. After the length specifier is a one byte subpacket type specifier, with a value from the following list

2 signature creation time. 4 bytes. This must be present in the hashed subpacket session

3 signature expiration time. 4 bytes. If it is not present or has a value of zero, the signature does not expire.

4 exportable certification. 1 byte, 0 if not exportable, 1 if it is exportable. This subpacket can be found on certification signatures on keyrings, and indicates whether this signature must also be exported if the key is exported.

5 trust signature 1 byte indicates the level of trust in this key: 0 means valid, 1 means that the owner of this key is a trusted introducer, 2 means it is a trusted introducer of introducers, ...

6 regular expression ending in 00. This regular expression limits the trust signature. It states that the trust only applies to the user ID's matching the expression. The regular expression format is described fully in [3].

7 revocable 1 byte. 0 means not revocable, 1 means revocable. If not present, the signature is revocable.

9 key expiration time 4 bytes. The number of seconds after which the signature expires. If 0, the key does not expire. This can only be found on a self signature.

10 placeholder for backward compatibility. I do not know what it means, but I assume it can be ignored.

11 preferred symmetric algorithms (block ciphers). This is only found on a self signature, and is a sequence of magic numbers of block ciphers in decreasing order of preference.

12 revocation key. It consists a *class* (1 byte), algorithm id (1 byte) and a fingerprint. It means that the key with the given fingerprint may revoke this key. It is useful to assign this if you might forget the passphrase of the key, because then you cannot revoke it yourself. The class byte must have the first bit set. If the second bit is set, this information is sensitive, and should not be exported unless necessary. This subpacket is found on a self signature, and must be in the hashed section.

16 issuer key ID. 8 bytes.

20 notation data. Consists of 4 bytes of flags (first bit is set if it is humanreadable), length of name N (2 bytes), length of value V (2 bytes), name(N bytes), value (V bytes). This can be used to make any comment the signer wishes.

21 preferred hash algorithms. This is only found on a self signature, and is a sequence of magic numbers of hash functions in decreasing order of preference.

22 preferred compression algorithms.

23 key server preferences. This packet can only be found on a self signature, and consists of any number of flag bytes that tell a key server how to handle this key. Only the first bit has a meaning: if it is set it means: 'no modify', so that only the key owner can modify this key on the key server. I hope they add useful flags in the future, because at this moment I do not see the why of this subpacket.

24 preferred key server One string with the URL of the preferred key server. Note that each user ID may have its own preferred key server.

25 primary user id. One octet telling that this user ID is the primary user ID of this key.

26 policy URL. A string with a URL to a document that describes the policy that the signature was made under.

27 key flags. A string of flags. Currently defined are: bit 7: may be used to certify other keys, bit 6: this key can sign data, bit 5: this key can encrypt communications, bit 4: this key can be used to encrypt storage, bit 3: the private part of this key may be split, bit 1: the private part of this key can be owned by more than one person.

More flags can be added in the future. Superfluous flags must be set to zero, absent flags are considered zero. Some of these flags only make sense on a self signature.

28 signer's user id. Using this subpacket one can indicate which user ID made the signature, for instance to differentiate business- from personal signatures.

29 reason for revocation. Consists of revocation code (1 byte) and a string stating the reason of revocation. The first byte can be 00(no reason specified), 01 (key is superseded), 02 (key has been compromised) 03 key is no longer used, 20 (user ID is no longer valid). 20 may appear only on certification signatures, 01, 02 and 03 on key signatures and 00 on both kinds.

100-110 internal or user-defined

4.5 Other packets

PK encrypted session key packet(1)

This packet contains an encrypted session key. It consists of

> 03 (version indicator), key ID (8 bytes), PK algorithm (1 byte), big integers depending on the algorithm: $m^e \bmod n$ for RSA, or $g^k \bmod p$ and $m * y^k \bmod p$ for ElGamal.

The value m is created by appending a symmetric algorithm specifier (1 byte), the session key (#bytes depends on algorithm) sum of the session key bytes (2 bytes, mod 65536 if necessary).Then PKCS # 1 padding is applied.

It is not necessary to specify the real key ID, to prevent `traffic analysis`. One can also give the all-zero key id.

SK encrypted session key packet(3)

SK stands for symmetric encrypted session key packet. One can use this packet and a passphrase to find a session key. It consists of 04 (version number), sym. alg. specifier (1 byte), an S2K specifier (see page 57 for the length), optionally the encrypted session key. If the encrypted session

key is present, it can be decoded with the pass phrase (CFB, allzero IV). If not, the passphrase converted with the S2K gives the session key. In both cases, the result is a symmetric algorithm specifier followed by the session key.

One pass signature packet(4)

In a signed PGP file, the data comes first and then the signature. PGP would have to skip the data, find the signature, go back and hash the data with the indicated hash algorithm, and check the signature. A one pass signature packet can be placed at the beginning of such a file, and tells PGP what hash algorithm to use, so that it can process the file in one sweep. It consists of

> 03 (version number) [signature type] [hash algorithm] [PK algorithm] [key ID of signing key] [1 byte, zero if another one pass signature packet follows, not zero if otherwise]

The one pass and the actual signature packets must bracket the message: the first one pass packet must match the last signature.

Compressed data packet(8)

One byte gives the compression algorithm, the rest of the packet is compressed data. The compression is described in RFC 1950(ZIP) and 1951(ZLIB).

PGP contained `data compression` from the very first version. It is included because compressing data elimates obvious patterns in the data, making it harder to break the conventional encryption. Later people discovered that modern block cipher are so good that data compression is not needed for security. It still makes sense to do compression before encryption: you cannot compress ciphertext, so compression afterwards is not useful. It also compensates for the expansion caused by the ascii armoring.

Encrypted data packet(9)

This packet contains the output of the selected block cipher.

Marker packet(10)

This packet spells 'PGP': 50 47 50. It can be placed at the beginning of a message that cannot be read by PGP 2.6.x to make sure that that program does not try to handle what it cannot handle.

Literal data packet(11)

text/data specifier (one byte, 0x62 ('b') if it is binary data, 0x74 ('t') if it is textual data), length L of the filename (1 byte) , the filename (L bytes), modification time of the file (4 bytes), literal data that needs no further interpretation. A file name can be given if the data comes from a certain file. If the filename is "_CONSOLE", the data must be considered extra sensitive. PGP will not save the data, but only display it in a TEMPEST-safe font. The time can be 0, to indicate current time. Text data is stored with [Carriage Return][Line Feed] between the lines.

Trust packet(12)

The trust packet must only be used in keyrings. This packet contains the users specification of which key holders are trustworthy introducers. OpenPGP does not describe how this packet is formed.

User ID packet(13)

This packet contains one string. By convention, it contains a name followed by an email address enclosed in <>.

4.6 key ID and fingerprint

A key ID is an 8 byte value that can be used to aid looking up keys. It cannot be assumed that key ID's are different for different keys. One can speed up finding keys by first checking key ID's, but later on the right fingerprint must be checked. The original V3 format used the lower 8 bytes of n as key ID. In version 4 the key Id is the lower 8 bytes of the fingerprint.

A fingerprint is a 16 or 20 byte string uniquely identifying a key. To check the validity of a key, it is enough to check the fingerprint, for example by phone. The fingerprint of a V3 key can be found by hashing the numbers n and e, without the length bytes with MD5.

A V4 fingerprint is a SHA-1 hash of the entire public key packet, with the header. The header must use a twobyte length type.

4.6.1 key ID problems

Due to the small size of a key ID it is possible to find two keys with the same key ID. For V3 keys, it is also possible to find a key with a specific ID. The CTK group created a tool `AbattoiR` to do this. One of the key ID's that was generated was DE AD BE EF (hexadecimal notation). For V4 keys this cannot be done, because of the hash function. This is not a security breach because no-one made any security claim about key ID's. However, it might be possible to trick certain applications (and users), that call PGP using key ID's instead of using fingerprints or names, into using a different key than the caller intended.

The use of key ID's can best be avoided. The suggested use is dangerous, since it is hard to deal with the non-uniqueness. The old format at least offered the advantage that calculating the key ID is cheap, but since the 'fix' in the newer format this is no longer true. You need to calculate the fingerprint, so there is no point in not using the entire fingerprint.

4.7 String to Key specifiers

A (`S2K`) describes how a variable length pass phrase can be converted into a fixed length key. These pass phrases are used for conventional encryption and for decrypting secret keys.

Simple s2k This s2k consists of 00 [hash algorithm id, 1 byte]. The corresponding procedure is: Apply the hash function on the pass phrase.

A weakness in this procedure is that a dictionary attack can be mounted: An attacker can build a key database by hashing a large set of common and likely passphrases. Many people can use this database many times, to try to guess the pass phrase of a key. This attack is better than just trying passphrases because the hashing needs to be done only once.

Salted s2k A specifier for salted s2k is 01 [hash algorithm, 1 byte] [8 byte salt value]. This works the same as simple s2k, but instead of hashing the pass phrase, the concatenation of the salt and the pass phrase are used. This approach makes it impossible to do a dictionary attack: The attacker needs a database built with the right salt value. Because this value is chosen different all the time the attacker cannot use such database often.

iterated and salted s2k This specifier looks like 03 [hash algorithm] [8 byte salt value] [1 byte count value c]. The value c specifies a number of counts, using the formula $count = (16 + (c \bmod 16)) * 2^{c/16+6}$. Count specifies the total number of bytes that will be hashed. To get these bytes, the salt and pass phrase are hashed repeatedly. In the unlikely case that a low count is specified, such that the count is less than the salt+passphrase, the salt and pass phrase are hashed entirely.

This count can be used if you are encrypting and decrypting on a fast computer, to frustrate attackers. If you use a slow procedure, they are also forced to do the extra calculations when they do a passphrase guessing attack. The theoretical security does not improve, because you have no advantage over the attacker: both of you must do more work. The standard recommends that the iterated and salted s2k is used. In practice, it can really frustrate a brute force attacker. OpenPGP recommends using this string to key specifier. There is no number of counts specified. If you know that both the encrypting and the decrypting computer have plenty of time, and that the pass phrase might be guessable, use $c = 255$. If you are PGPing with a webserver that needs to do a lot of crypto, you can use the salted s2k instead, because otherwise you would seriously drag down server performance.

If the output of the hash function is too short to be used as a key, the procedure is repeated, but with the value 00 prefixed to the input of the hash function. If another output is needed, 00 00 is prefixed, ... In practice, a 128 bit key must be produced from a 160 bit hash value, so this is not relevant, but if you combine the AES with MD5 this applies.

4.7.1 Problems

The string to key conversions seems to be dealt with properly. I see no problems. The use of a hash function assures that the key contains uniformly distributed bytes, thus assuring that no weak key is selected. The salting and iterating are designed against a dictionary attack. The only protection against a dictionary attack is a good pass phrase (long and random). If the pass phrase is not good, the disclosure of your key is a matter of time, and the above measure will only buy you time. It does not give the user any advantage on the attacker.

4.8 How to combine packets

The packets are building blocks. They need to be combined to make something usable. Not only can they be put behind and in front of each other, they can also be put inside each other. The OpenPGP standard does not state clearly how the elements should be combined. They do not because they think it is obvious, and because they want to leave possibilities open for future extension. So the standard allows much more than PGP will generate, and most likely also more than PGP accepts.

4.8.1 Keys

To export a key, concatenate the following: A public key packet or a secret key packet, maybe a few revocation signatures, one or more user ID packets. Each user ID packet must have one or more signatures directly behind it. The first signature is the self signature: it is made by the key itself. It shows that the creator of the user ID indeed had access to the key, and in the new format it can state extra information about the key, like the expiration date. The other signatures are made by

other keys. After the user ID's subpackets the subkeys can follow. Each subkey must be directly followed by a (subkey binding) signature, and maybe a few revocation signatures.

4.8.2 Messages

To PGP a message, quite a few packets must be combined. I hope you understand the following grammar: a | b denotes a choice: a or b. a , b means a followed by b. a, b | c means (a,b)|c.

OpenPGP Message:- Encrypted Message | Signed Message | Compressed Message | Literal Message.

Compressed Message:- Compressed Data Packet.

Literal Message:- Literal Data Packet.

ESK:- Public Key Encrypted Session Key Packet | Symmetric-Key Encrypted Session Key Packet.

ESK Sequence:- ESK | ESK Sequence, ESK.

Encrypted Message:- Symmetrically Encrypted Data Packet | ESK Sequence, Symmetrically Encrypted Data Packet.

One-Pass Signed Message:- One-Pass Signature Packet, OpenPGP Message, Corresponding Signature Packet.

Signed Message:- Signature Packet, OpenPGP Message | One-Pass Signed Message.

4.9 ASCII Armor

PGP users often want to send the encrypted files over a computer networks. Some (older) computer networks cannot be used to send binary files: These networks assume that only displayable characters are used, and 'correct' any nondisplayable byte value. To deal with this PGP can ASCII-armor these files: the file is re-encoded using only normal characters.

ASCII is the old name for a standard that describes which bytevalue should be used to encode a character (It has become an ANSI standard). The highest bit is not used, so only bytevalues below 128 are possible. The idea was that the highest bit could be used as a parity bit, to detect transmission errors. All asciivalues below 32 (space) have a certain meaning like 'beep', 'end of line', 'line feed', so they are also not displayable.

Armoring happens in two steps: First the data is recoded using displayable characters, then headers are added to help people recognize the data as PGP-input.

The encoding is quite simple: A group of three bytes(thus 3 groups of 8 bits) is changed into 4 groups of 6 bits. These six bits encode a number between 0 and 64. For each of these groups a character is output, from the list: *A..Za..z0..9+/*.If the last group has two bytes instead of three, two zero bits are added, three character are output in the normal way, and a '=' character is added. If the last group contains just one byte, 4 zero bits are added, two characters are formed in the normal way, and 2 '=' characters are added. Optionally, after the data follows a '='character and a 24 bit CRC-sum (also encoded, so 4 printable bytes). This checksum is not cryptographically strong, but can be used to detect transmission errors. This encoding is called `radix-64`.

Any characters not used by the encoding, like spaces and returns, can be inserted in the data and will be ignored. For Header generation, the data is split in lines with at most 76 characters, and header and footer lines are added.

Possible header lines are.

```
-----BEGIN PGP MESSAGE-----
-----BEGIN PGP PUBLIC KEY BLOCK-----
-----BEGIN PGP PRIVATE KEY BLOCK-----
-----BEGIN PGP MESSAGE, PART X/Y-----
-----BEGIN PGP MESSAGE, PART X-----
-----BEGIN PGP SIGNATURE-----
```

After this line zero or more key value pairs may be printed, to give extra information. Currently defined keys are

Version: [pgp version number]

Comment: [anything the sender wants to say]

MessageID: [used to find all parts of a split message. A 32 character string of printable characters]

Hash: [a comma-separated list of the hash algorithms used. For clearsigned messages]

Charset: [the character set used. To indicate a nonstandard character set(like Japanese)]

After the key-value pairs an empty line follows, and then the encoded data, and an end line. An endline is formed by replacing the word BEGIN with the word END. One exception is that a —– END PGP SIGNED MESSAGE—– is not used if a message and a signature are in the same file. The message ends where the signature starts.

A file can be split into multiple files, in case the mail program poses a maximum to the permitted message length. The PART X or PART X/Y header must then be used, with X the number of the part, and Y the total number of parts. Parts of the same message must have the same message-ID.

If you only want to sign a message, and not encrypt, you do not need to re-encode the message. You can put the message in the clear, and then the signature ascii-armored, like this:

```
-----BEGIN PGP SIGNED MESSAGE-----

Hi Bob,
I love you.
XXX
Alice
-----BEGIN PGP SIGNATURE-----
Version: PGP 7.0

iQCVAwUBO2V7Pyp3hQindTDDAQEgOAP/RhGnALKArLCLDoBLJidkETkQQSFFO/t4
noDX7fNKpMoQGyDkRPI22bqhTp/NCQEPh5wmrmk/UzPOwNOPJXW/yUVIaUgZWDWN
OEeTDkS5/ifsir0mzSyR+HDzt2mo6OzPpPuStG3AGhVkHiEwGKjagEB37Gau/40e
H5fupJo0jg0=
=Oz2J
-----END PGP SIGNATURE-----
```

4.10 Constants

The following tables give the constant values that are used within OpenPGP.

Table 4.2: OpenPGP algorithm names and magic numbers

Block ciphers

0	Plaintext or unencrypted data
1	IDEA
2	Triple-DES (DES-EDE,168 bit key derived from 192)
3	CAST5 (128 bit key)
4	Blowfish (128 bit key, 16 rounds)
5	SAFER-SK128 (13 rounds)
6	Reserved for DES/SK
7	Reserved for AES with 128-bit key
8	Reserved for AES with 192-bit key
9	Reserved for AES with 256-bit key
100 to 111	Private/Experimental algorithm

Public key algorithms

1	RSA (Encrypt or Sign)
2	RSA Encrypt-Only
3	RSA Sign-Only
16	ElGamal Encrypt-Only
17	DSA (Digital Signature Standard)
18	Reserved for Elliptic Curve
19	Reserved for ECDSA
20	ElGamal (Encrypt or Sign)
21	Reserved for Diffie-Hellman (X9.42,as defined for IETF-S/MIME)
100 to 111	Private/Experimental algorithm

Hash algorithms

1	MD5 "MD5"
2	SHA-1 "SHA1"
3	RIPE-MD/160 "RIPEMD160"
4	Reserved for double-width SHA (experimental)
5	MD2 "MD2"
6	Reserved for TIGER/192 "TIGER192"
7	Reserved for HAVAL (5 pass, 160-bit) "HAVAL-5-160"
100 to 111	Private/Experimental algorithm

Chapter 5

Bugs and Attacks

5.1 About bugs and attacks

Bugs are errors in computer programs. All large computer programs contain lots of them, but most bugs are not very dangerous and live on undetected until they are replaced by new code with different bugs. Some bugs are found during tests done by the maker or vendor of the software. In this case, the maker fixes the bug in the next version, and if it is serious it writes a 'patch'(program part that can fix installed versions) for current users, and distributes the patch to the official users. Some software makers also publish the bug on the internet, others do not.

Sometimes people find bugs in programs they did not write. If it is a bug that affects computer security, the following procedure is often followed: The vendor is informed, and gets a reasonable time (2 to 4 weeks) to fix the bug and inform its customers. Then the discoverer publishes the bug on the internet. There are special newsgroups and mailing lists for discussing bugs. One website for computer security is *www.securityfocus.com*, and they maintain the *bugtraq* database of computer bugs and the bugtraq mailing list.

Often vendors do not react properly the first time they are informed. They do not feel like fixing the bug, deny the bug, or the message got lost because no one in the company feels it is his task answering the mail. The discoverer will wait a 'reasonable' time (some hackers feel that 20 minutes is reasonable) and still publish the bug. This will greatly help evil minded people in attacking systems.

After the publication of the bug it is often fixed in the next version and can be patched. This does not mean the bug is history: not everyone installs all patches and uses the newest version of all its software. The bug or similar bugs can also be present in other software products. In 1999 the *format string bug* was discovered: It turned out that a very common function in the language C, *sprintf*, had a very unusual feature: It can overwrite arbitrary memory locations with arbitrary values. This 'feature' was available more than 20 years, but not all programmers knew about it. Thousands of programs have this bug, and every now and then a new instance of this bug makes it to bugtraq. In 2001, GPG turned out to have a format string bug. Other bugs also tend to return, and therefore I have tried to give an overview of security bugs that were discovered in the past in PGP, and I have tried to check whether they have been fixed.

5.2 Linux random bug

In PGP for Linux there is a severe bug that makes the keys generated easily guessable. (BID:1251) (CVE-2000-0445). The exact vulnerable version is PGP 5.0i for Linux. The error made is that when generating a key with the pgpk command from the console, PGP tries to read from the random device available under Linux with the following line of code:

```
RandBuf = read(fd, &RandBuf, count);
```

The problem is that *read* does not return the random byte: it returns the number of bytes read, which is 1 because count is 1. the line should have been:

```
read(fd, &RandBuf, count);
```

This problem was found by Germano Caronni on May 23 2000. It had already been solved without being discovered in the newer versions.

This a major bug, because it makes the keys that were generated non-interactively with this version completely worthless (note that it is disrecommended in the manual to generate keys non-interactively). A large part of PGP is devoted to securely generating random numbers. They have a very sophisticated randomness administration, and they hash all random data at least twice. If you do this kind of work, and it turns out that it does not do anything, it is a devastating bug. At the same time this bug is not interesting from a cryptographic viewpoint.

5.3 PGPdisk key schedule

There is a bug in the key schedule of the symmetric cipher of certain versions of PGPdisk. Vulnerable versions are PGPdisk for Windows 1.0 and the PGPdisk for windows that shipped with PGP 6.0.1. The version that ships with PGP 6.0.2 is fixed. PGPdisk is a program that allows you to encrypt your harddisk, so that you cannot use the harddisk without having the passphrase. PGPdisk functions independently of PGP. The bug is in the keyschedule routine of the CAST encryptions. Just like IDEA ,CAST generates a larger set of subkeys from the key, with a total size of 1024 bits. Instead of following the specification, the key is only copied into the buffer. So the first 128 bits of the subkeys are initialized, the rest remains zero. This means that the encryption is only as good as two round CAST, and that is not very good. The key can be easily recovered with a known plaintext attack.

The bug was found by NAI itself. This particular bug could not have happened with PGP, because any error in the ciphers will result in messages other people cannot read. PGPdisk volumes are not meant to be portable, but I assume NAI included a cross-compatibility test in their testset after this bug.

5.4 ADK bug

In August 2000, Ralph `Senderek`, a German researcher discovered a bug related to Additional Decryption Keys. Senderek is opposed against this feature, and he proved himself right by doing some key experiments. He added an ADK subpacket to a public key packet outside the signature protected area: some subpackets must be signed by the owner of the key, some not. The OpenPGP specification demands that ADK keys are placed in the signature-protected area, but he tested whether PGP would accept them if they are placed in the unprotected area.

He discovered that PGP does not complain about encountering an ADK in this area, although PGP never puts them there and it is not allowed in OpenPGP. This is thus a bug in certain implementations, not a flaw in OpenPGP. An attacker could obtain a victim's public key, add his own key as ADK, and then distribute the key as often as possible. People using this modified key would normally (it is considered polite, because it seems the victim asked for it) encrypt the messages with the modified key in such a way that the owner of the ADK can also read the message. The vulnerable versions are 5.5.x to 6.5.3. It is fixed in version 6.5.8. This bug can be described in two ways: As a design specification error, because apparently the designers of PGP overlooked this unusual attack, or as a typo, because all that was needed to correct the bug was adding one if-statement. The bug can be found in the function *ringKeyAdditionalRecipientRequestKey* in the file `pgpsrc\libs\pgpcdk\priv\keys\keys\pgpRngPub.c`. In this file there is the statement

```
krpdata = ringKeyFindSubpacket (obj, set,
          SIGSUB_KEY_ADDITIONAL_RECIPIENT_REQUEST, nth, &krdatalen,
          &critical, &hashed, NULL, &matches, error);
if (!krpdata  ||  !hashed) {
   if (IsntPGPError(*error))
          *error = kPGPError_ItemNotFound;
   return NULL;
}
```

The essential variable is *hashed*. The function *ringKeyFindSubpacket* will set it to *true* if it found the ADK packet in the part of the key that is protected by the signatures. In 6.5.1 this variable was ignored:

```
krpdata = ringKeyFindSubpacket (obj, set,
            SIGSUB_KEY_ADDITIONAL_RECIPIENT_REQUEST, nth, &krdatalen,
            &critical, NULL, NULL, &matches, error);
if (!krpdata) {
   if (IsntPGPError(*error))
          *error = kPGPError_ItemNotFound;
   return NULL;
}
```

Besides the above fix, all PGP software, including the key server program, was modified to delete any ill-placed ADK packet. This makes mounting an ADK attack very hard, since the extra ADK packet will not be distributed by the key server or by users with new versions of PGP.

Ralph Senderek placed his report[19] on his website, together with a reply made by Zimmermann. Both are worth reading. Senderek has some very radical ideas about PGP: He believes version 2.6.x is the best version of PGP, and that the new versions are too complex to be secure. He also states that it is not possible to do a source code analysis for a program of that size, and that ADK is a bad idea. Instead of source code analysis he proposes experimenting like one would do in physics.

No one has ever found illplaced ADK packets in the wild. A similar attack, with less harmful results, can be done by adding a revocation certificate in the unhashed area. This could lead to a `denial of service attack`. By doing this attack the attackers do not gain anything, but the defenders do loose a lot. If the ADK-bug was not discovered, some-one could revoke keys, which

would lead to a lot of confusion. Experiments by Martijn Stam[1] prove that this can no longer be done.

5.5 Czech private key attack

On 20 March 2001 Vlastimil `Klima` and Tomas `Rosa`, two Czech scientists, published a new attack on OpenPGP and PGP. Their attack exploits weaknesses in OpenPGP and in the case of DSA, lack of testing in PGP itself. The goal of their attack is obtaining the private key of the opponent. This key is on the secret keyring that can be obtained by breaking into someone's computer. PGP defends itself against this by storing private keys in encrypted form: You need the passphrase to decrypt the key.

They use the known fact that errors made when calculating a signature can leak the secret information in the key. They introduce those errors by modifying the secret key file.

Their attack plan is this:

1. Break into the victim's office, start his computer, copy his private keyring to a floppy disk, and make certain clever changes to his private key(s).

2. Wait until the victim comes back and makes a signature with its private key. (You can facilitate this with some social engineering: "You have won our Britney Spears Lottery. Send us this filled in and signed coupon and we will send you a billboard size poster."). In normal circumstances it is completely safe to make and publish signatures, so there is no reason why the victim would not do this.

3. Compute the secret key using the signature. This requires a little math depending on the kind of key (RSA or Diffie-Hellman, new or old format).

4. Break in again and restore the original private keyring. This is necessary if you do not want the victim to find out his key is compromised. With the modified key you cannot decrypt properly.

The first attack in the paper is on RSA private keys. These keys have the following structure (described in detail on page 50):

- public data: n and e
- secretdata: $d, p, q, u,$ checksum

All big numbers (n, e, d, p, q and u) are stored in the following format: first the length as a 16bit number, than the content bytes to contain the data in most significant byte first order. In the version 3 format, compatible with PGP 2.6.x, only the content of the numbers is encrypted, not the length. In the version 4 format (supported since PGP 5.0)(see page 50), the entire secretdata section is encrypted as one large chunk. The twobyte checksum consists of the sum of all secretdata bytes modulo 65536. What the attacker tries to do is change the value of u. To attack the version 3 format, he lowers the length of u with 1, and also lowers the checksum. The checksum is still correct.

When the victim tries to do a calculation with the corrupted key, he will construct the final output of the RSA operation in the following way (described on page 37):

[1] source is personal correspondence, results were not published

$$S' = s1 + p * u' * (s2 - s1) \bmod n = S + k * p \tag{5.1}$$

for a certain nonzero value of k. So the difference of S and S' is a multiple of p. The attacker does not know S. However, if he knows the document that was signed, he can construct S^e, because this is the hash of the document converted to a biginteger in a certain (deterministic) way. He calculates

$$A = S'^e - S^e = S^e + p * [anotherunknownnumber] - S^e \tag{5.2}$$

A is a multiple of p, so the attacker calculates $p = \gcd(A, N)$ and factored the victim's key.

The attack on version 4 keys is a little harder, because the lengths of the bigintegers are encrypted. What the attackers does instead is changing one of the last bits in u, and changing a bit in the same byteposition in the checksum. These one-bit changes in the encrypted data result in one-bit changes in the decrypted data due to the CFB mode used by PGP. The checksum will 'check' if both the byte of u and the checksum decreased, or both of them increased. This is a fifty percent chance, because all that is needed is that the changed bits were equal. So you might have to repeat this attack if it does not work the first time. Now the signature will factor q in the same way as for the version 3 attack.

These attacks on RSA do not work in PGP 7.0 (and other versions) because they check all public keys they read in. As an attack against OpenPGP it can be called successful, because OpenPGP does not warn against this attack.

DSA keys

To reach a complete knock-out, the Czech researchers also have a similar attack on DH/DSA keys that actually works against PGP. The security of DSA is based upon the insolvability of the discrete logarithm problem, and what the attackers do is modifying the public parameters, to make the discrete logarithm problem easier. They change p to $p' = 167 * 2^{151} + 1$. There is nothing particular about 167, except that it is a small prime, and that p' is again prime and $p' < q$. They also select a new generator g' with $g'^{((p'-1)/2)} \bmod p \neq 1$ and $g'^{((p'-1)/167)} \neq 1$. This assures that g' is again a generator of the group $\mathbb{Z}_{p'}{}^*$. Now the attackers wait for a signature on a known message m consisting of r' and s'. It follows that

$$r' = g'^k \bmod p' \bmod q = g'^k \bmod p' \tag{5.3}$$

$$s' = (k^{-1} \bmod q) * (H(m) + x * r') \bmod q \tag{5.4}$$

which can also be written as:

$$x = (s' * k - H(m)) * (r'^{-1} \bmod q) \bmod q \tag{5.5}$$

What the attacker does is retrieve k from the first equation. This is solving the discrete logarithm problem, but this instance of the problem is solvable because of the special structure of p': Pohlig-Hellman can be applied (see page 40). Then they use the last equation to find the secret value x.

Several people, including Bruce Schneier and Phil Zimmermann, downplay this attack, because this attack assumes it is possible to break into some-ones office. If you can break into some-ones office, you have won the game: You can install all kinds of malicious software or install equipment to eavesdrop the keyboard. These two options are much better than doing the Czech attack: In case of DH, you do not get any decryption key, just a signing key, and this attack can be discovered because the signatures made with the modified key are invalid.

These people are right, but one can think of special circumstances in which the Czech attack is a good attack: If someone mails his private keyring to another computer, relying on the passphrase protection. The paper that describes the attack, gives some valuable hints how PGP can be modified to prevent this attack: add a good checksum, verify the public key parameters, and verify any signature made. I think these are valuable hints that should be implemented as soon as possible.

On the other hand, Zimmermann's reply is not entirely wrong. The PGP manual states that people should guard their private keyring. PGP assumes the user controls his computer, and the scenario for this attack violates this assumption. Therefore it is, in a strict sense, not PGP's problem. However, if they believe this, why do they store the secret key encrypted?

Zimmermann is also not very happy about the way the attack was announced: at a trade show (Cebit) without warning NAI beforehand.

One can also apply the same technique to encrypted files: they neither contain a checksum. This is important to know if you use PGP for conventional encryption: PGP only protects signed files against changes. Any other file can be changed, and the bits in the last block of the file can be changed in a predictable manner.

This attack shows what one can do with a little mathematics, knowledge of the file format, and a very wide definition of the word 'attack'. I think it is a very interesting and even spectacular result. The conclusions of the report are important for everyone writing cryptographic software: Make all possible checks when reading key material.

5.6 ASCII armor bug

This bug was discovered in April 2001 by Chris `Anley`, working at @stake, a security company *www.atstake.com*. His summary of this bug is this:

> Opening an ASCII armored file such as a public key or a detached signature can cause the creation of an arbitrary file on the target machine. On the Windows platform this can lead to the execution of arbitrary code on the target machine.

The problem is that the ascii-armor parser of PGP can create temporary files, and does not delete these files if something unexpected happened in the input. It is possible to design a (malformed) armored file that causes the armor parser to write any file to the hardisk if you doubleclick it. This file will remain on the system. Anley enhances this attack by leaving behind a DLL file. A DLL is a Dynamic Link Library that contains subroutines that can be called by programs. All windows programs consist of *.exe* files and DLL's. If the file that is left has a catchy name, matching the name of an existing DLL, for instance a DLL used by the Acrobat reader, the code in it could be executed any time the user reads a PDF document in the same directory. The vulnerable versions are 5.0 to 7.0.4. NAI created a patch for this bug that will make sure that PGP gives a warning when a file is remaining on the system. It also ensures that PGP uses the right DLL's and not DLL's with the same name in the local directory. For more info on this bug check *http://www.atstake.com/research/advisories/2001/a040901-1.txt*. The relevant source file is discussed on page 75.

The example file Anley created is not ascii armored at all. This title is `ascii armor bug` because it is the ascii armor parser that makes the mistake. His example file, called *notes.doc.sig* contains one literal packet. It starts like this:

> af 30 c4 70 67 70 73 63 2e 64 6c 6c 00 20 61 72 62 69 74 72 61 72 79 20 66 69 6c 65 6e
> 61 6d 65 00 a0 ee 01 3d de f1 05 4c a0 16 92 fa de cb 69 cf 8a 85 3f 84 0b 62 c9 c5 ed

0d 16 35 d7 e2 21 b3 bd 52 a7 dc 56 89 36 d0 d1 4f 87 39 c4 2c 0d 2f 2e 2e 2f 2e 2e 2f
2e 2e 2f 2e 2e 2f 2e 2e 2f 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
4d 5a 90 00 03 ...

In the first line, *af*, or *10 1011 11* states the packet type, (old format, type 11 decimally, two byte length). *30 c4* states the length, and *70–6c 6c* spells *pgpsc.dll*. The *00* ends this filename. The rest of the field is simply more room for a longer filename, and is ignored by PGP. Chris Anley filled it with "arbitrary filename" and more spaces. The data starts at the *4d*, and this byte and all the next bytes are simply copied into the newly created file with the given name. The file *pgpsc.dll* is a hostile replacement for a PGP file with the same name. It should verify all the signatures (even the invalid ones) but at my system it just forces PGP to state an unintelligible error message. Note that this file does not exactly conform to the OpenPGP standard for literal data packets, and that is what confuses PGP.

The fact that PGP uses DLL's that are in its current directory is also a bug in PGP, but not a cryptographic one. It is a Windows weakness that is known to Microsoft, and Microsoft gives a solution on its website. This attack shows that the security of PGP heavily depends on the security of the underlying operating system.

5.7 Multiple user ID attack

This attack has been discovered by Sieuwert van Otterloo, July 2001. The bug is based on confusion created by keys with multiple user ID's. The attack exploits a few small bugs in the graphical user interface of PGP 5.0 and above. A patch for this attack appeared September 4 2001.

In section 1.5 we spoke about key validity. In that section it was assumed that keys have only one name on it. This is not true in PGP: keys can have more than one user ID. If this is the case, there is a problem with determining key validity. One cannot speak about *the* validity of the key anymore: Both names on the key can be valid independently of another. One should speak about the validity of all key-name combinations separately, not about the validity of the key. Often one can ignore this issue: Suppose a key has two names on it: "Alice <alice@home.com>" and "Alice <alice@hotmail.com>". One could say that the key is valid if the Alice that created the key is the owner of both email addresses. If on the other hand the creator of the key owns neither email address, and is not even called "Alice", then the key is clearly invalid.

Things get interesting if the owner of the key owns one of the email addresses, but not the other, and sometimes calls herself Alice. One of the key-userID links is valid, and the other key-userID link is invalid. In this case the key is not completely invalid and not completely valid. It is then not meaningful to speak about the key validity.

PGP does not use the concept of key validity internally, and the OpenPGP standard also does not assume it. All signatures are valid for just one user ID. But the user interface tries to make things simpler than they are: There is a green/gray light indicating the 'key validity', and there is a slider in the key properties window giving the key validity. They use different heuristics to communicate validity to the user:

- The first strategy is to assume that the first user ID is the most important, and to use the properties of this key as the properties of the key. For instance PGP uses the first user ID as the name of the key.

- The second strategy is to use the most valid user ID. This makes sense: PGP simply ignores the user ID's it knows nothing about.

It turns out the PGP mixes these strategies in a very unlucky way, and one can confuse users with this property. To exploit the confusion, a key with two user ID's is needed. Below is a recipe for making such keys. It describes what Eve must do to cheat on Bob. She needs a little assistance from Carol. Carol is a close friend of Bob, and Carol's key is listed as trusted in Bob's keyring. It is a simple procedure that can be done without diving into file format details. The file name suggestions are there to make it easier to refer back to files. It works with any new PGP version: 5, 6 and 7.

- Eve generates a key with the name Alice on it, with Alice's email address, or an address that appears to be Alice's.

- Eve adds her own name and email address as a second user ID to the key. (click on the key, select keys/add/name)

- Eve exports the key including the private key somewhere on the computer, under the name *temp1.asc*.

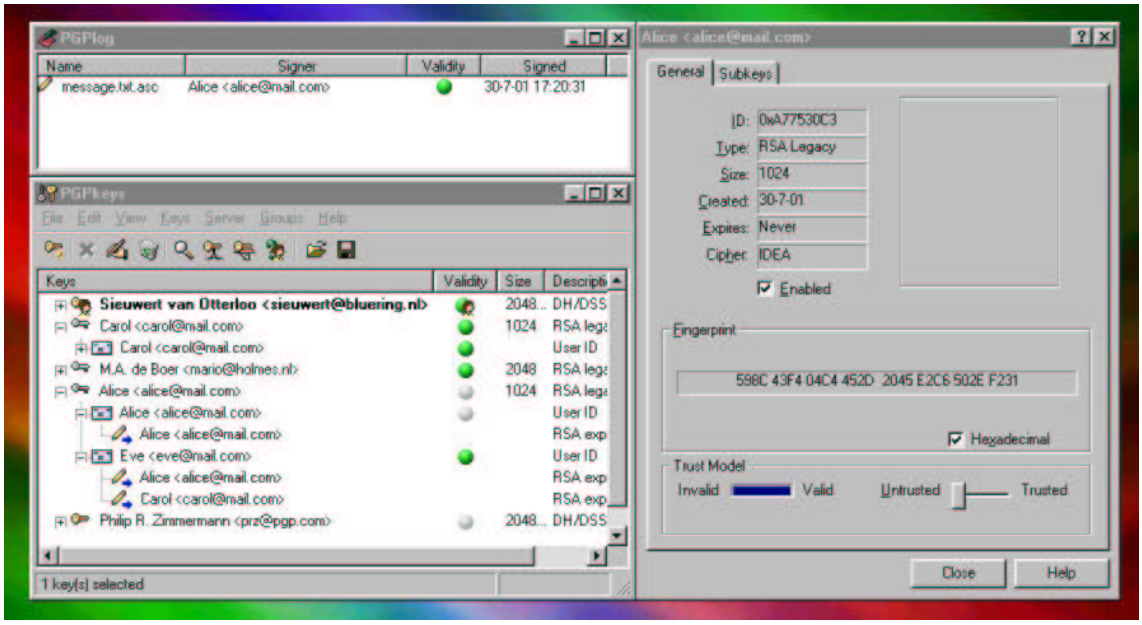
- Eve deletes the Alice user ID from the key, and exports the key, now having the name Eve as its only name, without the private key, in the file *tocarol.asc*. Then Eve deletes her key from her keyring.

- Eve personally goes to Carol with her public key file *tocarol.asc*. Carol sees that the key indeed belongs to Eve, so she signs it (,makes the signature exportable), and gives the public key with the signature back to Eve, so she can use the signature to convince other people she is Eve (filename *fromcarol.asc*). By setting the signature Carol did not declare she trusts Eve, she only declared that the key indeed belongs to Eve.

- Eve first imports *temp1.asc*. Then she imports *fromcarol.asc*. Now she again has one key, with two user ID's. The first one is Alice, so the key has Alice as primary name in the PGP keyring view. The second user ID, Eve, has Carol's signature.

- Eve exports this key under the name Alice.asc without the private key, and tries to give it to Bob when he is looking for Alice's key.

If Bob imports this key into PGP, and tries to verify a signature made with the key, he will see something weird. Please check the screenshot. The dark validity lights are green on a computer screen, and mean valid. The lighter ones are also gray on color screens and mean invalidity.



In the PGPLog window (upper left), you can see that the signature is considered valid, and the signature seems to be made by Alice. What happens is that the name field uses strategy 1, and gives the first name of the key. The validity light in this window however uses strategy 2, and reasons using the most valid user ID: the second. Bob now falsely believes that the file comes from Alice. We have successfully forged a signature!

The other windows also show interesting things. The PGPKeys window shows exactly what it should show: The first user ID is not valid, the second is, and if one takes Alice as the name of the key, then the key is not valid. The properties window however is confused and/or confusing. It shows the name Alice, but the Eve properties: It thinks the key is valid. It even allows raising the trust bar. If Bob raises the trust bar, because he trusts Alice, the first user ID becomes also valid (it is signed by a trusted key, namely itself) and Bob will start encrypting his secret messages to Alice with Eve's key. Eve can now read everything Bob sends to Alice! PGP normally does not allow setting the trust of invalid keys, but here it does. So both the validity and the trust slider are wrong (or one can say the properties window should show a different name. It is the mixing of the two strategies that creates the problem).

Experienced and careful PGP users can discover this attack easily. It is also a very daring attack: Eve identifies herself to Carol, and Bob knows her name immediately. For this reasons it is not very likely that someone will do this attack for real. Nevertheless, if someone decided to try it, he or she has a fair chance to get away with it. And once you have fooled Bob, he might sign the key and give it to his friends. If there is one gap in the web of trust, it can spread and infect the whole system (again, it is not likely to happen in real life, but it neither completely unlikely). I think it is an attack with a very good effort/effect ratio: the attackers needs no computing power at all, just a little bit of luck and carelessness.

Chapter 6

Sourcecode analysis

6.1 How the code was analysed

The latest source that was available at the beginning of the project was the source of PGP 6.5.8. With this source all variations (freeware/business) of PGP 6.5.8 for the windows platform(s) can be built by setting the appropriate switches. The source of the latest version at that time, 7.0, was not available, but these versions are still quite similar. At the start of the project it was expected that the sourcecode of 7.0 would be released, but now it seems that that will never happen.

The sourcecode is written in the C language. To build a windows version, one needs Microsoft Visual C 6.0, a handful of extra utilities (windows Software development kits and driver development kits) and a few 3rd party products (like installshield). With that source, one could make the complete product, which also consists of many components. The main component containing the crypto is `pgpsdk.dll`. It was verified that we could build that DLL without the 3rd party products we did not have. We disassembled the precompiled file and the one we compiled, and found many differences. The larger ones could be explained by different compiler settings, and the smaller one are maybe caused by some arbitrary compile-time decisions. We think it is quite likely that NAI distributes the real source of PGP, not 'a source without the backdoors'. Note that other people always could find the bugs found in the executables in the sourcecode,

We did not acquire the third party products because we did not intend learning to use them.

When using C, it is common to split the code into many smaller files. The files with suffix .c contain most of the code, and for each of these files there is a .h file with the same name, that contains declarations from the corresponding C code that need to be used elsewhere. Both of these files contain lots of comment: extra text that is not used by the compiler but explains the program. The comment in the .h file usually explains what the code is for, and the comment in the .c files explains what is happening at that piece of code. Most comments are just short remarks, but some files contain complete stories about the underlying ideas of PGP. The comment often mentions the articles the code is based on, like [12].

The initial idea was to read this sourcecode from the beginning to the end, checking everything and further investigate everything deviating from normal. This should result in finding many of the bugs.

However, reading the sourcecode was not as easy as expected. There are 1620 C source and header files, with a total size of 15 Mb. Most of these files are not about cryptography. Although the source is commented, it is not always fun to read it. The C language is not the most advanced language. It gives fast programs but does not necessarily give the same readability other programming languages can provide.

The method used instead of reading the source entirely was searching the code for interesting pieces, large comments, specific keywords, and checking methods called or callers of methods. The utility called *WinGrep* by Huw Millington was of great value by these searches.

6.2 Memory management

There is a lot of sensitive information inside the computer's memory while PGP is running: secret messages, secret keys, session keys and the random pool. PGP is designed not to write this information to your harddisk. It even tries to stop the operating system from writing sensitive data to a swap file, if the memory used by PGP is needed elsewhere. Under Windows, it accomplishes this by implementing a special device driver for nonswappable memory.

PGP tries to minimize its use of non-swappable memory. In the Public Key code all memory for large numbers is allocated using the statement

```
bnBegin(&bn, mgr, TRUE);
```

`&bn` is the memory address of the variable `bn`. `bnBegin` will store a reference to the new memory in this variable. `mgr` is the memory manager that knows how to get memory, and the TRUE indicates that the memory could contain sensitive information. It is hard to check whether windows will not store this information despite all PGP's efforts (Windows NT is a large program, the source is not available), but we did check that all FALSE indications are correct.

It is not hard to find all occurrences of the String "bnBegin" by using a program like Windows Grep. Most of these invocations are correct, but a few needed more attention: The last arguments are sometimes omitted, like

```
bnBegin(&a);
bnBegin(&e);
```

in the file `bndsaprime.c`. It was hard to find out what this means because there is no bnBegin function or macro with one argument. It turned out that the 1-argument bnBegin's occur only in old code that under normal settings is removed by the preprocessor. If the settings are changed, the compiler complains about the functions. These findings are not dangerous on itself, but it does make one suspicious: How often and thorough has this code been checked? Is this secure programming?

6.3 File Parsing

In the OpenPGP chapter, it was shown that the file format that PGP uses is quite difficult. The ascii armor bug has shown that things can go wrong while reading in data. For this reasons we have checked the sourcecode that reads the files. File parsing is always a good place to look for security bugs: Programmers tend to write parsers that assume the input is correct and just do what should be done in that case, while parsers are used on any input and their primary task is determining that the input is correct. This is especially true if the input rules are not very clear: The OpenPGP format allows many interesting packet combinations, while PGP always produces some very specific combinations.

The code is in the directory `/libs/pgpcdk/priv/keys/keys`

6.3.1 Web of Trust

The code concerning the web of trust is in `pgpRngMnt.c`. There is code for three trustmodels: Trustmodel 0 is the model from older versions of PGP. Trust model 1 is an approximation of the full model: The model is very expensive to calculate, and this code leaves out complex path, and thus returns a slightly lower trust value than the real value. Trustmodel 2 does the Maurer model exactly, with a depth limit (only paths of length shorter than 10), and other constraints if needed. The standard setting is trustmodel 2, but I can imagine that trustmodel 1 is used if there is a large keyring.

One of the comments in this file is

```
 * Since PGP supports multiple names on a key, the question arises
 * which validity value is used.  The solution here is to assign a trust
 * value to each of the names, take the validity*trust product for each
 * name, and take the maximum.
```

This sounded so strange to me that I started thinking what exactly was meant here, started experimenting and found the user ID attack in section 5.7. The bugs the attack is based on have nothing to do with the code in the file. However, this comment shows that the whole concept of multiple user ID's is hard to handle, and that at several places in the code choices have to be made between the two possible strategies. In this case studying the sourcecode has lead to the discovery of bugs, showing that sourcecode analysis is important.

6.3.2 Ooh, a binary PGP message

The file `/pgpsrc/libs/pgpcdk/priv/crypto/pipe/file/pgpprsasc.c` contains the bug Chris Anley found, and the errors happens around the line marked

```
/*
 * ooh, a binary PGP message fed into the
 * ascii armor parser!
*/
```

This file is huge (3500 lines), and it is hard to say from the source what is actually happening. It implements a finite state machine to parse the several parts a PGP armored message can consist of.

6.4 Public key code

One of the first sections that were checked was the code dealing with the public key cryptography. The reason is that this is a very localized subject (the relevant code is easily found, and the relevant code is short), secondly because there are many things that can go wrong here. There are all kinds of demands on the numbers involved (prime, relatively prime, random, nonzero, not too small, ...).

The PK code is in the directory `/libs/pgpcdk/priv/keys/pubkey`.

6.4.1 RSA

The sourcecode of RSA is complicated by the fact that the code can use different libraries.`pgpaltrsaglue.c`1 uses the RSAREF cryptographic library from RSA Data Security. This file is not used under default settings. The same is true for `pgpBRSAGlue.c`. This uses the BSAFE library also from RSA

Data Security. The next RSA file, `pgpMSRSAGlue.c` uses the MicroSoft Crypto API. It is also not used in default settings (sigh). The file `pgpRSAGlue` is the one truly used. It uses PGP's own big number code, which is the fastest.

I do not think anyone would want to use the alternative libraries. They were used because of legal problems. The legal problems are solved. The alternative libraries are slower than PGP's own, and some libraries have lower limits on the key size.

A very interesting file is `pgpRSAkey.c`. It contains the code to generate RSA keys. PGP generates random primes by generating a random number of the specified size, setting the lowest bit (to make it odd), and setting the highest two bits. The highest bit must be set to make the number the requested size. The use of setting the second highest bit is unclear: The comment states that it is forcing the number to be in the high range makes it harder to factor, but I think they mean that setting these bits makes sure that n is the requested size (it can happen that you multiply two 512 bit numbers, and get a 1023 bit number, and that is generally not what you want). A prime just above the starting random number is returned by the function bnPrimeGen (not always the first prime, because it shuffles the 256 numbers above the starting point).

It is funny to see that studying the exact details reveils some bits of n. Because $q, p > 3 * 2^{512}$ it must be that $n > 9 * 2^{1024}$ so n does not start with 1000.

For e a fixed value called RSA_DEFAULT_EXPONENT is taken. In the sourcecode it is 17, but the PGP version 7 I use uses 65537. The code contains a bug: after generating q, they should check that $gcd(e, q-1) == 1$. They do not however. The code (line 1454) says

```
pgpAssert(bnModQ(&sec->s.p, exp) != 1);
```

It rechecks p, instead of checking q. It should have been

```
pgpAssert(bnModQ(&sec->s.q, exp) != 1);
```

The test is redundant, because the bnPrimeGen function can have e as an extra argument, forcing it to search for a prime for which e is invertible: It has been assured that the properties holds at the moment the number was generated, so the test is superfluous. Therefore this bug is not exploitable, but it is worrying that there is a bug in such a crucial section of PGP.

6.4.2 DSA

The file `pgpDSAkey.c` contains the code for DSA signatures. PGP does the checks needed to verify a signature ($r, s < q$). Note that PGP cannot deal with negative big integers, so that positivity checks can be omitted. This is quite unusual, most big integer libraries are general-purpose so they admit negative numbers, but it seems to add to the safety of PGP.

The comment surrounding the DSA code also motivates the existence of a dummy pool: Because the public parameters are generated at the same time as the secret parameter x, they fear that it could in theory be possible to reconstruct the state of the random pool from the public parameters (by breaking the hashfunction, which is hard). To prevent this, they want to generate the public parameters from only a small part from the pool. In this way the public parameters represent only a small part of the pool. The dummy generator is seeded with 64 bits from the random pool. This is not unguessable (80 bit is secure according to section 2), but that is not what we need here: 64 bits assures that we have unique public parameters each time. Note that having the same public parameters is not even a security risk: The algorithm also works with common public parameters. It is just a more attractive target for attackers. (If PGP had common public parameters, rumors about a backdoor could rise.)

6.5 Random numbers

6.5.1 The use of random numbers

Many computer programs need numbers chosen at random. Computers have trouble making these numbers, because the electronics are designed to always give one outcome on each instruction. Often carefully constructed complicated formulas are used to generate a stream with a complicated and therefore hard to recognize pattern, and for many applications these `pseudo-random numbers` will do. For cryptography only the best quality of random numbers is good enough: truly random numbers. According to [13] chapter 5 a random bit generator is a device that outputs a sequence of independent and unbiased binary digits.

The component of PGP that delivers these numbers is called the (Truly) `Random Number Generator`, in contrast with the so-called Pseudo Random Number Generators, algorithms that generate deterministic stream of data appearing to be random. This `TRNG` uses a hybrid approach: it consults external sources to get indeterminacy, and uses a hash function to improve unbiasedness.

PGP maintains a pool with 5120 bits of data. This pool is kept in memory and is written to disk when PGP is not running so that the pool is not empty at a start-up of the program. The Pool's input consists of timings of user events such as mouse movements and key presses. It maintains a counter with the estimated entropy in the random pool. Using this administration the TRNG can try to collect extra randomness if it runs short on entropy. The administration is very precise (it even uses fractions of bits).

`Entropy` is a measure for uncertainty. It is measured in bits. If you no nothing about an 8-bit number except that it is an 8 bit number, it contains 8 bits of entropy. If you know it contains the number of children of a certain person, you know that low values are more likely than high values, so it contains less than 8 bits of entropy.

6.5.2 random buffer output

If PGP wants some random bytes, the random pool is encrypted in CBC mode (see section 3.2.6) and the contents of the register R is placed in the output buffer of the random pool (the pool is replaced by its encryption). Instead of a block cipher, the compression function of SHA is used, because compression-functions are one-way, while block ciphers do not have this property. The output buffer then can be read in small pieces. If someone adds a byte in the meantime, the buffer is cleared. This output routine is very defensive. Every output bit depends on every bit in the pool, and that knowing the output does not help you in reconstructing the pool, assuming that SHA is indeed a one-way function. Once the random pool has at least 80 bits of entropy, the output is practically random.

There is also a stir operation that feeds bytes from the output back into the pool. This shuffles the content of the pool. Methods can stir the pool if they want the random bytes before and after the shuffle to be 'extra unrelated'. The stirring operation is not needed, considering the amount of hashing already applied. PGP calls the stirring operation after generating a key, to make sure that if attackers obtain the state of the pool after the key generation, it will not help them in reconstructing the key.

6.5.3 random buffer input

Any input bytes are first placed in a key buffer. If this buffer is full enough (80 bits entropy or 32 bytes data) the to be replaced bytes from the pool are XORed into the key buffer, the pool is

CBC-hashed with the key buffer, and the result replaces one word of the pool (word means a block of bytes with exactly the length of the current hash output). The Xoring ensures that no entropy is lost: the randomness in the overwritten data is still present in the outcome of the Xor function.

The hash hides all patterns in the input. The truly random data gathered from various sources is often not uniformly distributed: certain values are more likely than others. Applying a hash function does not increase nor decrease the amount of entropy in the data[1], but it does hide all skewedness in the distribution of the data and returns uniformly distributed bits.

The most important question that remains is where the entropy in the input of the TRNG comes from. There are a few ways to collect entropy in a standard PC. PGP uses from keypress timings, mouse movement timings, key presses (the actual values) and various elements called OS data. What the OS data is depends on the operating system. Under Windows it consists of statistical data about processes, which can be found in the registry. PGP normally collects these data silently, but if a lot of entropy is needed, for instance when generating a new public/private key pair, it shows a dialog where the user is asked to act randomly. A further source of random data is a hardware random device. This is a bunch of electronics that can for instance be included in a processor that measures noise. Intel thought about including it in modern processors around 1999, but they decided otherwise. The device is not present in any processor but available on some motherboards (with certain Intel chipsets). The PGP code is ready to use it. In the sourcecode distribution it is switched off (a flag is defined in `pgpSDKbuildflags.h`), but in the executables it is turned on. These hardware random devices are very fast, and probably as safe as other randomness sources. Check [9] for a report on this random device.

PGP carefully estimates the entropy in each sample. For the key presses, it does not assume any entropy in a character that appeared more than once in the past four presses. It also tries to estimate the entropy based on previous samples using a formula for the entropy E in a sample X with an expected value A: $2^E = \text{abs}(X - A)$. This formula does not work on any data stream, but it seems safe for timings of random events.

It uses this formula with the expected value based on the previous sample, and using a first order derivative on last two examples, and with A based on a second order derivative on previous three examples, ..., as far back as requested. For mouse events, it looks two samples back, but for key presses there is a three step history. The minimum value of E found is used.

What makes the code hard to read is that the PGP source code wants to use fractional entropy without using floating-point numbers (not all processors have these instructions. especially the original 8088 chip that PGP 1.0 was designed for and current embedded chips). To store an amount of entropy it uses two integers: the first contains the integer number of bits, $\text{floor}(E)$ for the mathematicians, the other stores $(2^{E-\text{floor}(E)}) * 2^{32}$. It is multiplied by 2^{32} to make it integer. The 2^E can be estimated directly with the formula given above, and the most important test, $E > 1$, can be done by noticing $2^E > 2$.

6.5.4 dummy pool

Despite the careful design, the PGP designers did not think it is a wise idea to use the TRNG for the generation of the public numbers in for instance ElGamal: They do not want the outside world to learn too much about the random pool. They created a different random pool that produces all zeroes: the `dummy pool`. It is compatible with the real TRNG, it just does not give any random numbers. This not so useful device must not be used *unless you know what you're doing*, according to a comment in the code.

[1] unless the input contained more entropy than the size of the hash output

With the help of the dummy pool, a strong pseudorandom number generator can be created (a `pseudorandom number generator` is a deterministic function that produces randomly looking output). You need to random pool to create one, so that is properly initiated. If you use the dummy pool, it is of course not properly inititiated. so this generator must then be seeded again with output from the real pool, and is then used to generate the requested random data. This procedure minimises the drain from the random pool and is safe. It is also a good example of not so good programming: One author enforced the proper seeding of pseudorandom number generators by demanding a random pool to create a generator, another author bypassed this security measure by creating a dummy pool.

The files closely examined are:

- `pgpRndWin32.h`
- `pgprandompool.h`
- `pgpRndWin32.c`
- `pgprandompool.c`

6.5.5 PGP 1.0

As a sidestep of historical interest, I would like to quote the comment from the file `random.c` in the original PGP 1.0 distribution.

> This code generates truly random numbers derived from a counter that is incremented continuously while the keyboard is scanned for user input. Every time the user touches a key, the least significant bits of the counter are pushed on a stack. Later, this supply of random bytes can be popped off the stack by applications requiring stochastic numbers. Cryptographic applications require this kind of randomness.

This also sounds secure (although there should be some hashing before using the bytes) and a lot simpler because this code is targeted to a specific operating system and processor: MS-DOS, a singletasking operating system where the program can access the hardware directly. On a different system the code might not work properly because the operating system decided to process the key presses at predictable times.

6.6 The XXX comment

Besides looking at the above specific points, a quick walkthrough revealed that many sections of the sourcecode need further examination, according to the authors. These sections are marked with XXX, so that they can be easily found by a utility like GREP.

This is remarkable. One would expect that the makers of PGP would solve al these issues before releasing anything. Apparently they use a different policy: They test until they think it is time for a release, not until things are perfect. I cannot say this is wrong, but I would like to quote some of the comments, for fun and to show that it is naive to expect that a program does not contain errors just because professional programmers wrote it.

```
/* Experimental XXX@@@ */
/* XXX: should I do this?*/
XXX Beware that calling this routine could be a security flaw,
```

```
/* XXX Is this stuff still necessary? */
XXX This code is buggy!
/* XXX Experimental backwards compat code - put DSA keys at end */
XXX: We may want to rethink this heuristic
XXX AUTOMATIC ADK EXTRACTION DOES NOT YET WORK
XXX Need to fix these based on trust model and
    other aspects return kPGPError_LazyProgrammer;
XXX Need to recognize error of having another packet type there.
XXX Potential space leak if <closeFlag> is false
/* XXX do something here */
/* XXX this doesn't seem right */
```

I could not exploit the possible bugs mentioned above, and sometimes it is just a 'beauty error': For instance a LazyProgrammer error just indicates that they have no time to produce the right kind of error message, but they still give an error message. It is hard to start just at a marked point, and reason how to exploit it. If you start tracing how you got into the code, you quickly end up with 15 source code files calling each other and things seeming to be OK.

6.7 Conclusions

It is hard to find all bugs in the source code of a large program, especially if you are not the author. Simply reading the source code from start to end is not a good method. What we have done is scanning the code, reading all comments, and studying the parts we think were interesting. This results in two things:

- Finding implementation level details in certain algorithms. It shows how PGP exactly does certain things, for instance in the PK section. The textbook descriptions leave a lot of details to the programmer. It is important to know those details, because they might affect the security.
- Finding certain bugs, omissions or pecularities in the code. We have found a handful of these, but are convinced that there are more.

The general impression is that the PGP programmers knew what they were doing, because important things are checked and doublechecked, and that there is a lot of effort put into PGP to make it secure. This give some confidence in PGP.

Unfortunately it is also clear that PGP contains bugs, like any other program. This does not mean one should not use it, but it means that one must remain careful when using PGP. PGP security is not absolute.

Chapter 7

Conclusion

It was very interesting to dive deeply into a complete implementation of a crypto system, especially into such a famous system with such colorful history of PGP. Maybe the most important conclusion of our research is that building a cryptosystem involves more that combining a few algorithms. As for the questions we posed, these are our answers:

7.1 Is PGP broken?

There were and most likely are serious bugs inside PGP. Some of them discovered, some of them not. These bugs are PGP's weak spot, and are the entry point for attacks. The attacks people created do spectacular things: Obtain secret keys, forge signatures, read messages, but at the same time they do not break PGP in the naive sense of the word: PGP is still a program that gives you pretty good privacy, because the attacks require special circumstances or carelessness. Therefore PGP is not broken, and using PGP certainly gives you more privacy than not using PGP. I do recommend using PGP, because it gives you a fair amount of security for a reasonable price.

Some of my friends like to ask me

Did you break PGP?

On page 69 you will find my PGP attack plan. It allows one to forge signatures, and even read messages in the right circumstances. So the answer is yes :-)

If PGP will be broken in the future, it will most likely not be one of the algorithms in PGP that will be broken: Those algorithms are very strong and many researchers already have tried to attack them, without practical results. The bugs in PGP are the most worrying, and the security of the whole computer you use PGP on.

7.2 Does PGP contain a backdoor?

Neither in the algorithms, nor in the sourcecode could we find any backdoor. It is highly unlikely that someone can easily break PGP. I think all rumours about backdoors built in on behalf of for instance the US government are nothing but rumours.

7.3 Which version of PGP is best?

This question is hard to answer, partly because it is a matter of taste: the first versions have an entirely different user interface than the latter.

During its lifetime PGP have gained many nifty features, like harddisk encryption and file wiping. Using these features certainly enhances the security of their users computers. But the most important question is whether PGP does its primary task well: secure communications.

The makers claim the latest version of PGP is best, because of all the bugs and errors they corrected. It is impossible to check this claim: The sourcecode is not available. They most likely have a very good reason for not publishing the source code, so any version after 6.5.8 cannot be good. The newest version is maybe the most userfriendly or advanced version, it cannot be recommended for principal reasons.

All versions before 2.0 use Bass-o-Matic, and should never be used. From the old versions 2.6, in all its variations is the most advanced version. This is the best version according to experts like Ralph Senderek. This version is weaker than newer versions in some respects: it uses MD5 and the old file format is weaker than the new one.

Newer versions of PGP, from 5.0 to 6.5.8 have a better file format, but this is compensated by the more complex sourcecode, that certainly contains more bugs. If you want to use one of those versions, use 6.5.8, because this one does not contain the ADK bug.

The only product I really wish to recommend is the Gnu Privacy Guard, because this is an open source project. I think open source is very important for a computer security product, more important than ease of use. It is very sad that NAI thinks otherwise.

7.4 How can PGP be more secure?

The algorithms are excellent, the file format is good, all methods used are correct. What should be changed is that the code should contain fewer bugs. One of the best was to achieve this is to simplify the program and shorten the sourcecode. They should create a PGP version that only does the basic things and does that well, written as short as possible.

For the distant future, I also have a few personal opinions: The OpenPGP standard is growing over time, getting more complicated. This does not improve the security. It becomes more and more unclear what a signature means each time features are added, and this makes wrong judgements more likely. Having more and more algorithms is also not improving the security, because if one is broken attackers can try to influence people using that one. People should strive for a core standard that just contains the bare minimum. Average users that do not have the time to get to know the ins and outs of cryptography have no need for advanced features: If you do not know how it works you cannot use it safely.

I think there is room for a second program, intended for crypto experts and enthusiasts that can be used for experimenting with different algorithms. This program would be great for spreading cryptographic knowledge and demonstrations. This advanced version could then also analyse the messages of the core standard and maybe contain factoring algorithms as well. Many of the hardcore PGP users would enjoy having these options.

Bibliography

[1] Carlisle Adams. *The CAST-128 Encryption Algorithm*, RFC 2144 edition, 1997.

[2] Derek Atkins, W Stallings, and Philip Zimmermann. *PGP messages exchange format*, RFC 1991 edition, 1996.

[3] J Callas, L Donnerhacke, H Finney, and R Thayer. *openPGP Message Format*, rfc 2440 edition, 1998.

[4] H. Dobbertin, A. Bosselaers, and B. Preneel. Ripemd-160, a strengthened version of ripemd. *Fast Software Encryption*, pages 71–82, 1996.

[5] Hans Dobbertin. Cryptanalysis of MD5 compress, 1996.

[6] Simson Garfinkel. *PGP:Pretty Good Privacy*. O'Reilly and associates, 1995.

[7] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. *Sixth USENIX Security Symposium Proceedings*, 1996.

[8] Johan Hastad. Solving simutaneous modular equations of low degree. *SIAM Journal on Computing*, 17(2), 1988. http://www.nada.kth.se/ johanh/papers.html.

[9] Benjamin Jun and Paul Kocher. The intel random number generator, 1999.

[10] B Kaliski. *RSA encryption PKCS 1 version 1.5*, rfc 2313 edition, 1998.

[11] J Katz and B Schneier. A chosen ciphertext attack against several e-mail encryption protocols. *www.counterpane.com*, 2000.

[12] Ueli Maurer. Modelling a public key infrastructure. *proceedings of European Symposium on Research in Computer Security 1996*, (1146), 1996.

[13] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC press, 1996. available as pdf.

[14] NAI. *PGP desktop security administrators guide*, 2000.

[15] NIST. *Digital Signature Standard(DSS)*, fips publication 186 edition, 1994.

[16] NIST. *Secure Hash Standard (SHS)*, fips publication 180 edition, 1994.

[17] Ron Rivest. *The MD5 Message-Digest Algorithm*, RFC 1321 edition, 1992.

[18] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, 2 edition, 1996.

[19] Ralph Senderek. Key experiments. *http://senderek.de/security/key-experiments.html*, 2000.

[20] Gerard Tel. *Cryptografische beveiliging van de Digitale maatschappij*. 2002.

Index