Global Information Assurance Certification Paper

PGP: HOW IT WORKS AND THE MATHEMATICS BEHIND IT

by

Karen t. Coe

Submitted in partial fulfillment of the
requirements for the certification

GSEC Security Essentials version 1.4 b

SANS Institue

2002

SANS

Abstract

PGP: HOW IT WORKS AND THE MATHEMATICS BEHIND IT

by Karen Coe

PGP is a free, downloadable email encryption program. The newest versions use a combination of the CAST, ElGamal, DSA, and SHA-1 algorithms. Public key encryption (ElGamal) is used to encrypt a symmetric key (CAST) used for encryption of the encryption of the actual message; thus providing a mechanism for secure key exchange. The digital signature is a signed hash (DSA, SHA-1) ensuring authenticity and integrity. For extra assurance the owner of the key may be verified with a 'fingerprint' – a randomly generated string of numbers or words associated with the key. The public key algorithms use properties of the multiplicative group, Z*p, in their creation. This group satisfies the following two conditions: efficiency (its group operation is easy to apply) and security (the discrete logarithm problem in Z*p is computationally infeasible – given p is large enough). The symmetric key, (CAST-128) is a block cipher, of the Feistel Ciphers. The DSS (Digital Signature Algorithm), for DSA, is specified in FIPS 186, and the SHS (Secure Hash Standard), for SHA-1, is specified in FIPS PUB 180.

Part 1

How PGP Works

PGP (Pretty Good Privacy) is a freeware, downloadable program used to encrypt and digitally sign email, created by Phil Zimmerman. It is simple to use via plug-ins for Outlook and Eudora. PGP combines public key and symmetric key cryptography, providing both efficiency and security.

The process begins with the compression of the message ("...the algorithms are functionally equivalent to those used by PKWare's PKSIP 2.X." [1]). It is then encrypted with a one time session key - generated by random movements of the user's mouse, and keyboard strokes. The session key is then encrypted with the recipient's public key. Signing is done with the sender's private key, and a hash of the message.

Since symmetric key encryption is many times faster than public key encryption, this method yields a fast, yet secure encryption scheme. Decryption is via the decrypted (with the recipient's private key) session key. Authentication is done with the sender's public key. Below is a simple demonstration of how the process works.

All of this is transparent to the user. The plug-in for Outlook provides a popup menu, which includes check boxes for encryption and signing. After the initial setup, the program may even be set to encrypt and sign messages automatically.

During set up, the user generates two sets of public/private keys, and chooses one of CAST (default), AES, IDEA, Triple-DES, or Two-Fish (for the session key). One set of public/private keys (DH) is used for the encryption of the session key; the other (DSA) is used for signing. This way, even if the signing key is hacked, the message cannot be decrypted and, if the encryption key is hacked, the signing key is still secure. PGP versions prior to PGP 5.0 used the same RSA set for both signing and encryption. This meant that, in the event of key compromise, messages could be both read and signed by the compromiser.

The DH part of the key (actually a variant, ElGamal, is the algorithm used) may be of length 1024-4096 bits. The DSS part is always 1024 bits (this is the length specified in the standard). CAST is a 128-bit block cipher. Part 2 of this paper describes the algorithms in detail.

Part 2

THE ALGORITHMS

As stated in Part 1, PGP uses two public key algorithms, one symmetric key algorithm, a one-way hashing algorithm, and a compression algorithm. These are seamlessly and transparently integrated into the program. The user initially selects the algorithm and key sizes when the program is installed. The public key options are DH/DSS, RSA, and legacy RSA. The symmetric key options are CAST AES TripleDES, IDEA, and Twofish. The SHA-1 algorithm is used for the Hash to create the digital signature. PGP uses its own compression algorithm, but will not try to compress a message which has already been compressed with another. This paper will discuss only DH/DSS, CAST, and SHA-1, the algorithms most often used in PGP today.

DH/DSS

DH stands for 'Diffie Hellman', after its inventors, and DSS stands for 'Digital Signature Standard'. Actually, the algorithm used for the DH part, is a derivation of DH, called 'ElGamal', also after its inventor. The ElGamal algorithm is as follows:

> **"SUMMARY**: each entity creates a public key and a corresponding private key.
>
> Each entity A should do the following:
>
> 1. Generate a large random prime *p* and a generator *g* of the multiplicative group Z*p of the integers (modulo p)
> 2. Select a random integer *a*, 1<=a<=(p – 2), and compute g^a mod p
> 3. A's public key is (p; g; g^a); A's private key is 'a'.
>
> SUMMARY: B encrypts a message m for A, which A decrypts.
>
> 1. *Encryption.* B should do the following:
> (a) Obtain A's authentic public key (p; g; g^a).
> (b) Represent the message as an integer m in the range {1, 2,…, p – 1}.
> (c) Select a random integer k, 1<=k<=(p – 2).
> (d) Compute s=g^k mod p and t=m (g^a)^k mod p.
> (e) Send the ciphertext c= (s,t) to A.
>
> 2. *Decryption.* To recover plaintext m from c, A should do the following:
> (a) Use the private key a to compute s^a and then compute s^{-a}.
> (b) Recover m by computing (s^{-a})(t) mod p." [1]

Let us see an example (with numbers too small, by any standard, to actually be used):

1. Generate *p* and a generator *g* of the multiplicative group Z*p.
   P=5 g=2 Note: <2> = {2,4,3,1}=Z^*_5.

2. Select a random integer, *a*, 1<=a<=(p – 2)
   a = 1 1<=a<=(5- 2)

3. The public key is (p, g, g^a)=(5, 2, 1); the private key is 1.

To encrypt the message 'm=d=4' (note: 'm' is represented as an integer in {1,2,….,p-1}), using the public key:

1. Select a random integer 'k' so that 1<= k<=(p-2)
   k=3 1<=3<=3

2. Compute $s=g^k$ and $t=m(g^a)^k$ mod p
   $s= 2^3 \text{mod} 5 = 3$ and $t=4(2^1)^3$ mod 5=2

3. Send the ciphertext c=(s,t)
   c=(3,2)

To decrypt c=(3,2):

Use the private key, 'a' to calculate $s^{(p-1-a)}$
$3^{5-1-1}=3^3 \text{mod} 5=2$
then $m= s^{(p-1-a)} * t \pmod p$
$m= 2 * 2 \pmod 5=4$ m='d', as desired.

DSS

The Digit*al Signature Standard* (DSS) is a standard specified by NIST (National Institute of Standards and Technology) in as part of the FIPS (Federal Information Processing Standards). It specifies a Digital Signature Algorithm (DSA) that is appropriate when a digital signature is required.

> *"Explanation: This Standard specifies a Digital Signature Algorithm (DSA) appropriate for applications requiring a digital rather than written signature. The DSA digital signature is a pair of large numbers represented in a computer as strings of binary digits. The digital signature is computed using a set of rules (i.e., the DSA) and a set of parameters such that the identity of the signatory and integrity of the data can be verified. The DSA provides the capability to generate and verify signatures. Signature generation makes use of a private key to generate a digital signature. Signature verification makes use of a public key which corresponds to, but is not the same as, the private key. Each user possesses a private and public key pair. Public keys are assumed to be known to the public in general. Private keys are never shared. Anyone can verify the signature of a user by employing that user's public key. Signature generation can be performed only by the possessor of the user's private key.*
>
> *A hash function is used in the signature generation process to obtain a condensed version of data, called a message digest (see Figure 1). The message digest is then input to the DSA to generate the digital signature. The digital signature is sent to the intended verifier along with the signed data (often called the message). The verifier of the message and signature verifies the signature by using the sender's public key. The same hash function must also be used in the verification process. The hash function is specified in a separate standard, the Secure Hash Standard (SHS), FIPS 180. Similar procedures may be used to generate and verify signatures for stored as well as transmitted data."* [3]

First, we will examine the parameters used in the algorithm. As with the ElGamal algorithm, DSA makes use of the properties of Z*p as a multiplicative group.

"1. p = a prime modulus, where $2^{L-1} < p < 2^{L}$ for $512 <= L <= 1024$ and L a multiple of 64
2. q = a prime divisor of $p - 1$, where $2^{159} < q < 2^{160}$
3. $g = h^{(p-1)/q} \bmod p$, where h is any integer with $1 < h < p - 1$ such that $h^{(p-1)/q} \bmod p > 1$ (g has order $q \bmod p$)
4. x = a randomly or pseudorandomly generated integer with $0 < x < q$
5. $y = g^{x} \bmod p$
6. k = a randomly or pseudorandomly generated integer with $0 < k < q$ " [3]

The parameters (p, q, g) are made public. **The public key is 'y'; the private key is 'x'**. Parameters 'x' and 'k' are kept secret for signature generation (a new 'k' must be generated for each signature).

Let us generate a set of (again artificially small) set of parameters:

1. **p=11** (prime modulus of size specified in the algorithm – we are ignoring the size specifications for this example).
2. **q=5** (a prime divisor of p-1 (in our case,*10*), also of specified size).
3. **g=4** (chosen so that order(g)=q. Another way to say this is that <g>=Z*q. In our case <g>={4,5,9,3,1})
4. **x=2** (randomly generated with 0<x<q)
5. **y=5** ($y=g^x \bmod p$)
6. **k=3** (randomly generated for each signature, with 0<k<q)

Our public parameters are (11, 5, 4) – recall this is (p, q, g)
Our **Public Key is** 5 (this is 'y') and our **Private Key is** 2 (this is 'x')

Notice that the public key $y=g^x \bmod p$ could be used to solve for *x*, the private key, if the parameters were such that the discrete logarithm problem was 'easy'.

That is, $x = \log_g y \pmod p$. In the case of this example, $x=\log_4 5 \pmod{11}$. This can be solved very easily, by simply raising *4* to progressive powers. We see $4^2=5 \pmod{11}$ so x=2. This illustrates why the parameters must be chosen as specified.

At this point, make note of the fact that $y= g^x \bmod p$ and *g* and *p* are public. As illustrated above, if we could manage to take the discrete log of y(modp), we would know *x* – the secret key. The numbers *p*,*q* are chosen, and *g* calculated, in such a way as to make this too difficult to do, even with a lifetime of super computers available. What if it were possible to replace the parameters with easy values (say *p'*,*q'*,*g'*) in such a way as to make this possible? Black hat has already asked (and answered) this question as we will soon see!

We now examine the creation of a Digital Signature (again with small parameters for the example).

When creating the signature of the message *m* (or its hash value *h(m)*) the user uses the private key *x* and public parameters according to the following procedure:

1. Select the random secret number *k*, $0 < k < q$.
   $0 < k < 5$ $0 < 3 < 5$
   k = 3

2. Calculate $r = (g^k \bmod p) \bmod q$.
   $r = (4^3 \bmod 5) \bmod 11$
   r = 4

3. Calculate $kInv = k^{-1} \bmod q$.
   $k * k^{-1} = 1 \pmod 5$ so $3 * ? = 1 \pmod 5$
   Using Euclid's Algorithm or guess and check or a modular calculator we arrive at:
   $k^{-1} = 2$

4. Calculate $s = [kInv * (h(m) + x*r)] \bmod q$.
   $s = [2 * (h(m) + 2*4)] \bmod 5$ Let us suppose that m hashes to h(m) = 123
   $s = [2 * 123 + 2*4)] \bmod 5$
   s = 2

5. Digital signature of the message m is the pair (r, s).
   DIGITAL SIGNATURE: (4,2)

Let us note that r, s, q are generally 160bit numbers, whereas p, g, y are 1024bit numbers.

To verify the digital signature of the message *m* we use signer´s public key *y* and public parameters (*p*, *q*, *g*) according to the following procedure.

1. Verify that 0< r, s< q. In the opposite case the signature is invalid.
   0 < 4,2 < 5

2. Calculate sInv = s^{-1} mod q and hash value h(m).
   2^{-1}(mod 5) = 3 (by same method as above) Assume the message hashes to h(m) = 123 as above
   s^{-1} = 3 h(m) = 123

3. Calculate u_1 = sInv * h(m) mod q, u_2 = sInv * r mod q.
   u_1 = 3 * 123(mod5) u_2 = 3 * 4 (mod 5)
   u_1 = 4 u_2 = 2

4. Calculate v = (g^{u1} * y^{u2} mod p) mod q.
   v = (4^4 * 5^2mod11)mod5
   v = 4

5. The signature is valid iff v = r.
   v = 4; r = 4 so the signature is valid

Recall, the parameters have been chosen in such as way as to make the signature secure. However, Black Hat is not stopped quite so easily! Suppose we HAD been able to somehow substitute the values *p, g, q* into secring.skr (the secret key file) *and* obtained a signed clear text message. There would now be a new pair *r',s'* based on our spoofed values (*p'* and *g'*)and the randomly chosen *k*. And, we have a message with the spoofed signature, so we can see *r', s'*.

Now, from the definition of the signature value (r´, s´) it results that

(1) $r' = (g')^k \bmod p' \bmod q$
(1a) $r' = (g')^k \bmod p'$ because $p' < q$ (this was the reason for that choice)

Because of our spoofed values,it is possible to solve (1a) for the random number 'k'.

Next, we use

(2) $s' = \{[k^{-1} \bmod q] * [h(m) + x*r'] \} \bmod q$, thus

(2a) $x = \{ [s'* k^{-1} h(m)] * [(r')^{-1} \bmod q] \} \bmod q$. (multiple both sides by k and subtract h(m))

The key issue is now that we are able to calculate the unknown randomly chosen number thanks to the choice of p' and g'. We can do this because

The prime number p´ was selected in such a way, that the equation (1a), i.e. the task of the discrete logarithm in Z_p* be easy to solve. [7]

"Let us notice that the integrity of the "Public Key Packet" field is not visibly secured anywhere in the format of OpenPGP, and as it became apparent by effecting a practical attack, not even in PGPTM programs. Nevertheless, when creating the digital signature it is public parameters of this field that are just utilized (in the event of PGPTM program, the Secret Key Packet is stored specifically in secring.skr file). These parameters could be read from the record of the public key (the file pubring.pkr), but it is logical that if the record of the private key is open, they will be read from here. In the record of Secret Key Packet the value of the private signature key is protected, but the mistake is that here the value of public parameters or public key is not protected anyhow. Specifically in the event of DSA values p, q, g, y are at issue, of which we will use only p, g for specific attack.

The main idea of the attack on DSA consists in the following steps. The attacker:

1. will prepare special numbers (constants) of PGPrime and PGGenerator

2. will obtain the structure of Secret Key Packet of the given user and replace p, g values stored in the structure "Public Key Packet" inside Secret Key Packet by values p´= PGPrime and g´= PGGenerator

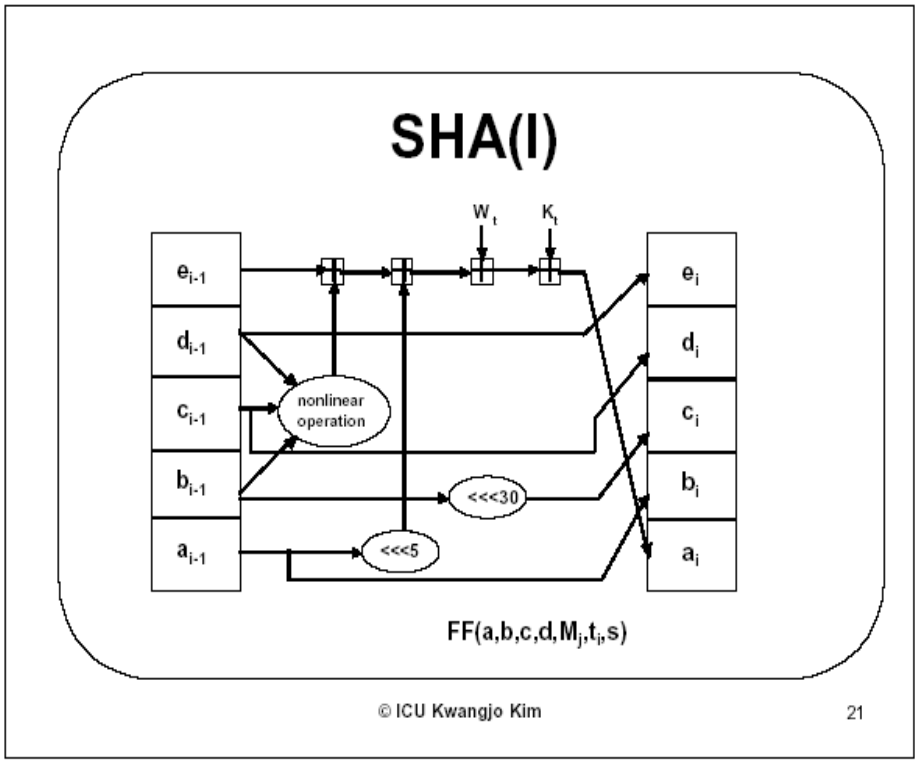
3. will capture the first not enciphered message or the file which the user signed with such false parameters and will keep its signature

4. on the basis of the obtained message and its signature it will calculate the private key of the user (x value)

5. will return the p, g values to the original condition

…The key issue is now that we are able to calculate the unknown randomly chosen number thanks to the choice of PGPrime and PGGenerator. The prime number p´= PGPrime was selected in such a way, that the equation (1a), i.e. the task of the discrete logarithm in Zp´ be easy to solve. On the basis of this procedure we then calculated value k from the equation (1a) and additionally computed value x from the equation (2a). We checked the correctness of x according to the relationship y = gx mod p with original values y, g, p. The x value is therefore calculated and its validity is verified against the value of the public key."* [7]

The *SHA-1 (Secure Hash Algorithm)* is the compression algorithm used in the signing. Below is a diagram of how it is used in the digital signature:



The following description of SHA-1 is taken from [8]:

"(The SHA-1 algorithm is an) iterative, one-way hash functions that can process a message to produce a condensed representation called a *message digest*. (The) algorithm enables the determination of a message's integrity: any change to the message will, with a very high probability, result in a different message digest. This property is useful in the generation and verification of digital signatures and message authentication codes, and in the generation of random numbers (bits).

(The) algorithm can be described in two stages: preprocessing and hash computation. Preprocessing involves padding a message, parsing the padded message into *m*-bit blocks, and setting initialization values to be used in the hash computation. The hash computation generates a *message schedule* from the padded message and uses that schedule, along with functions, constants, and word operations to iteratively generate a series of hash values. The final hash value generated by the hash computation is used to determine the message digest.

SHA-1 may be used to hash a message, *M*, having a length of *l* bits, where $0 < l <= 2^{64}$. The algorithm uses

1) a message schedule of eighty 32-bit words,
2) five working variables of 32 bits each, and
3) a hash value of five 32-bit words. The final result of SHA-1 is a 160-bit message digest.

The words of the message schedule are labeled W_0, W_1,…, W_{79}. The five working variables are labeled ***a***, ***b***, ***c***, ***d***, and ***e***. The words of the hash value are labeled$H_0^{(i)}$, $H_1^{(i)}$, …,$H_4^{(i)}$, which will hold the initial hash value, $H_{(0)}$, replaced by each successive intermediate hash value (after each message block is processed), $H_{(i)}$, and ending with the final hash value, $H_{(N)}$. SHA-1 also uses a single temporary word, *T*."

Recall, the hash function is one-way. That is, no plain text will ever be recovered from it. Also, note that, by its design, it will produce a unique hash for any given plaintext; and any change in the plaintext will produce a drastic change in the hash. Below is an example of implementation (also from [8]).

"Let the message, *M*, be the 24-bit (l = 24) ASCII string **"abc"**, which is equivalent to the following binary string:
01100001 01100010 01100011.

The message is padded by appending a "1" bit, followed by 423 "0" bits, and ending with the hex value 00000000 00000018 (the two 32-bit word representation of the length, 24). Thus, the final padded message consists of one block ($N = 1$).

For SHA-1, the initial hash value, $H(0)$, is
$H_0^{(0)}$ = 67452301
$H_1^{(0)}$= efcdab89
$H_2^{(0)}$= 98badcfe
$H_3^{(0)}$= 10325476
$H_4^{(0)}$= c3d2e1f0.

The words of the padded message block are then assigned to the words $W_0,\ldots,W_{15}$ of the message
schedule:
W_0 = 61626380
W_1 = 00000000
W_2 = 00000000
W_3 = 00000000
W_4 = 00000000
W_5 = 00000000
W_6 = 00000000
W_7 = 00000000
W_8 = 00000000
W_9 = 00000000
W_{10} = 00000000
W_{11} = 00000000
W_{12} = 00000000
W_{13} = 00000000
W_{14} = 00000000
W_{15} = 00000018.

The following schedule shows the hex values for ***a***, ***b***, ***c***, ***d***, and ***e*** *after* pass *t* of the "for $t = 0$ to 79" loop described in Sec. 6.1.2, step 4.

a	***b***	***c***	***d***	***e***

$t = 0$: 0116fc33 67452301 7bf36ae2 98badcfe 10325476
$t = 1$: 8990536d 0116fc33 59d148c0 7bf36ae2 98badcfe
$t = 2$: a1390f08 8990536d c045bf0c 59d148c0 7bf36ae2"

Note that the a,b,c,d,e values are the results of swaps, rotations, and functions defined in the algorithm. The functions make use of the $ROTL^{30}(x)$ function. $ROTL^{30}(b)$ is illustrated below. Note that $ROTL^{30}(b)$ is equivalent to $ROTR^{32-30}(b)$ where '32' is the size of the word, 'b'.

Original 'b'	1110 1111 1100 1101 1010 1011 1000 1001
Round t_0 'c'	0111 1011 1111 0011 0110 1010 1110 0010

The hex number is written in binary, then the rotation is applied. (Compare values with the hex values given in the table). The functions are detailed in [8].

CAST

The *CAST Algorithm* is a block cipher, based on framework of the Feistel cipher. CAST uses the Feistel structure because it has been well studied and is thought to

be free of structural weaknesses that would leave it open to certain types of attacks.[10] The Feistel structure is a Substitution and Permutation Network (SPN). This algorithm is illustrated as follows (the example used has nothing to do with the actual keys and functions of the CAST cipher; it is merely a simplistic illustration of the process):

Input a message block of '2n' bits and split it into a left half, 'L_1', and a right half, 'R_1'.

M="abcdefgh"="12345678; L_1=1234 R_1=5678

The right half and a subkey K_1 are input to a "round function", f_1.

f_1(R1, K1)=f1(5678, 34)= 5678 + 34 = 5712

Now the modified R_1 is XOR ed with L_1.

10011010010 + 1011001010000 = 1001010000010

This becomes R_2, and R_1 becomes L_2.

R_2=1001010000010 L_2=5678=1011000101110

This completes round one.
This continues for many rounds, depending on the cipher. For the very last round, the left and right halves are not swapped; rather they are concatenated. If the above example had only one round, the cipher-text would be:

10110001011101001010000010

The decryption is simply the reverse:

Split the cipher text.

R_2=1001010000010 L_2=5678=1011000101110

Assign L_2 to be R_1.

R_1=L_2=1011000101110=5678

Now 'R_2 XOR $f_1(K_1,R_1)$' is assigned to L_1

1001010000010 XOR (1011001010000) = 10011010010 = 1234 =L_1

Put the Left and Right sides back together.

M= 12345678

The actual CAST encryption algorithm may be described as follows:

Divide the plaintext block into a left half and a right half. In each of 8 rounds, the right half is combined with a key using a function 'f' and then XORed with the left to make the new right half. The original right half becomes the new left half. After the eight rounds are done (the left and right halves do not switch after the eighth round), the two halves are concatenated. This forms the cipher text.

The function 'f' is where the 'S-boxes' come in. The following explains the S-box process:

"An m-to-n S-Box gets a block of m bits as input and returns a block of n bits as output with n• m. The result is obtained by indexing a lookup table with the input.

Example. The S-Box S_1 *below is used by the DES algorithm. It gets 6-bit blocks as input and returns 4-bit blocks. [10]*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

If the input block is $x_1x_2x_3x_4x_5x_6$ *we take* x_1x_6 *as row index,* $x_2x_3x_4x_5$ *as column index, and return the corresponding table entry as output. For example, for input 011011 the row is* $(01)_2 = (1)_{10}$ *and the column is* $(1101)_2 = (13)_{10}$*; hence the output is* $(5)_{10} = (0101)_2$*. "*[11]

The CAST S-boxes differ from the DES S-box (described above) in that the input is 8 bits and the output is 32 bits. That is, the output is larger than the input. Yet the SPN still has equal input and output block sizes because of the round function. Within it, the input data half is modified by the subkey for that round. Then, it is split into pieces, and each piece is input to a different S-box. The outputs are combined using binary functions (such as XOR); the result is the output for that round. This use of the S-boxes does not result in expansion of the data. The S1,S2, S3, and S4 boxes are used for the round function, and the S5,S6,S7, and S8 boxes are used for key-schedules.

The function 'f' is defined differently for different rounds :

"Three different round functions are used in CAST-128. The rounds are as follows (where "D" is the data input to the f function and "Ia" - "Id" are the most significant byte through least significant byte of I, respectively). Note that "+" and "-" are addition and subtraction modulo 2**32, "^" is bitwise XOR, and "<<<" is the circular left- shift operation.

Type 1: I = ((Kmi + D) <<< Kri)
f = ((S1[Ia] ^ S2[Ib]) - S3[Ic]) + S4[Id]

Type 2: I = ((Kmi ^ D) <<< Kri)
f = ((S1[Ia] - S2[Ib]) + S3[Ic]) ^ S4[Id]

Type 3: I = ((Kmi - D) <<< Kri)
f = ((S1[Ia] + S2[Ib]) ^ S3[Ic]) - S4[Id]

Rounds 1, 4, 7, 10, 13, and 16 use f function Type 1.
Rounds 2, 5, 8, 11, and 14 use f function Type 2.
Rounds 3, 6, 9, 12, and 15 use f function Type 3." [12]

CAST-128 uses two subkeys per round: a 32-bit 'masking key', and a 5-bit 'rotation key'. The purpose of the masking key is to thwart differential and linear cryptanalytic attacks against the cipher. This done with a nonlinear, key-dependent operation before the S-box lookup. In this way, the input to the set of s-boxes is effectively masked. The Key schedule and s-boxes are given in [12] The default key size is 128 bits in PGP.

The usual choice of algorithms for PGP is ElGamal, DSA, and CAST. These algorithms provide adequate security for most non-classified data, if implemented properly. Probably, the biggest security hole in PGP is not in the keys, or their lengths. As is so often the case, it is the user's passphrase that is the weak link in the chain. Each time the program is used for encryption, decryption, or signing, the user is prompted to enter a passphrase. If Black Hat can gain access to the user's machine, why bother with an attack on the keys? The passphrase is far easier to crack! Users should be cautioned to choose a passphrase with an adequate length and complexity.

Part 3

THE MATHEMATICS

The following table ,excerpted from [1], outlines many of the mathematical problems associated with cryptography:

Problem	Description
FACTORING	*Integer factorization problem*: given a positive integer n, find its prime factorization; that is, write $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$ where the p_i are pairwise distinct primes and each $e_i \geq 1$.
RSAP	*RSA problem* (also known as *RSA inversion*): given a positive integer n that is a product of two distinct odd primes p and q, a positive integer e such that $\gcd(e, (p-1)(q-1)) = 1$, and an integer c, find an integer m such that $m^e \equiv c \pmod{n}$.
QRP	*Quadratic residuosity problem*: given an odd composite integer n and an integer a having Jacobi symbol $\left(\frac{a}{n}\right) = 1$, decide whether or not a is a quadratic residue modulo n.
SQROOT	*Square roots modulo n*: given a composite integer n and $a \in Q_n$ (the set of quadratic residues modulo n), find a square root of a modulo n; that is, an integer x such that $x^2 \equiv a \pmod{n}$.
DLP	*Discrete logarithm problem*: given a prime p, a generator α of $\mathbb{Z}_p^*$, and an element $\beta \in \mathbb{Z}_p^*$, find the integer x, $0 \leq x \leq p-2$, such that $\alpha^x \equiv \beta \pmod{p}$.
GDLP	*Generalized discrete logarithm problem*: given a finite cyclic group G of order n, a generator α of G, and an element $\beta \in G$, find the integer x, $0 \leq x \leq n-1$, such that $\alpha^x = \beta$.
DHP	*Diffie-Hellman problem*: given a prime p, a generator α of $\mathbb{Z}_p^*$, and elements $\alpha^a \bmod p$ and $\alpha^b \bmod p$, find $\alpha^{ab} \bmod p$.
GDHP	*Generalized Diffie-Hellman problem*: given a finite cyclic group G, a generator α of G, and group elements α^a and α^b, find α^{ab}.
SUBSET-SUM	*Subset sum problem*: given a set of positive integers $\{a_1, a_2, \dots, a_n\}$ and a positive integer s, determine whether or not there is a subset of the a_j that sums to s.

This paper will focus on the Discrete Logarithm Problem (DLP); the security of the ElGamal encryption algorithm is dependent upon the intractability of this problem. Following, is a brief discussion of the mathematics.

First, some conventions:

Z refers to the integers (…-2,-1,0,1,2,…).

p is a prime number.

'n(mod)p' refers to the remainder when n is divided by *p*.

Z*p = {1,2,3,…,p-1}, the set of equivalence classes modulo p, with 0 omitted. (Equivalence classes are sets of integers that have the same remainder when divided by p. For example, the equivalence class of 2 modulo 5 is {2,7,12,17,…})

If *a* is in Z*p, then **the inverse of a(modp)**, is the number *b*, such that ab=1(mod p). For example, in Z*5, *3* is the inverse of *2* because 3*2=1(mod5).

The **gcd(m,n)** is the greatest number that divides both m and n.

Now, for some elementary facts from number theory and algebra:

A number *m* has **an inverse (mod p)** if and only if gcd(m,p)=1.

A **group** is a set of objects that is associative, closed under its operation, contains an identity, and has an inverse for each element. That is, if *a,b,c* are in the set, then: (a*b)*c=a*(b*c); a*b is in the set; there is an element *r* such that a*r=r*a=e, and there is an element *e* in the set such that e*a=a*e=a. Note: '*' is a an operation that combines two elements, as in ordinary multiplication.

As an example, consider Z*5={1,2,3,4} with *e*=1 and multiplication (modp) - this means multiply two elements in the normal way, and then take the remainder when divided by *5*. Association holds by virtue of the properties of integer multiplication. Any two numbers multiplied together produce an element that is still in the set. i.e. 3*4=8=3(mod5) - still in the set. All of the elements have an inverse. For example 2*3=6=1(modp). It is easy to see that *1* has the desired properties for *e*, the identity. Thus, Z*5 is a group.

We will be considering the set Z*p as a group. Since it is a group, the elements of Z*p have inverses, but how can we find them? One way would be to try each element in the group, and see if it works. For example, to find the inverse of 2 in Z*p, we could try 2*1, 2*2, 2*3, and 2*4 and check to see which product was equal to 1(mod 5). Since 2*3=1(mod5), we see that 3 is the inverse of 2.

This is fine for small numbers, but what if p is 100 digits long? Typically, cryptography uses very large numbers for p, so this method is not practical. Of course, there is another way!

The **Euclidean Algorithm** is as follows:

To find the gcd(m,n), assuming m>n:
First calculate q_1 so that: $m=q_1n+r_1$
Next calculate q_2 so that: $n=q_2r_1+r_2$
Then calculate q_3 so that $r_1=q_3r_2+r_3$
Continue in this manner until $r_n=0$. r_{n-1} will be the gcd(m,n).

By backtracking, the gcd(m,n)=d may be written as xm + yn =d.

Getting back to inverses, recall we are looking for *r* such that ra=1(modp). This is the same as ra –sp=1 or ra+tp=1. But, since gcd(a,p)=1 if *a* has an inverse at all, we can use the Euclidean algorithm to find *r*. We can find multiplicative inverses (modp) in this way. However, we can also find the inverse of the generator, *g*, in another way. Fermat's Little Theorem tells us that $a^{-1}=a^{(p-2)}$(mod p). Hence, we need only raise *a* to the power *(p-2)* to find its inverse. To see why this is so, we need the concept of a generator.

A generator of Z*p is an element *g*, such that powers of *g* will produce all of the elements of Z*p. For example, <2> is {2, 4, 3, 1}=Z*5, so 2 is a generator of Z*5 (*<2>* denotes the set of all powers of *2*). There may be more than one generator, or a group may be generated by more than one element. In the case of Z*p, there will always be at least one single generator (this is what it means for a group to be called *cyclic*). Of course, there must be some number *l*, such that $g^l=1$ as $g^u<p$, by virtue of being a member of Z*p. This number *l*, is called the **order(g)**. If *g* is a generator, this number is *p-1*. Why? Because *g* , as a generator, must produce *p-1* elements; no more and no less. Recall: that's how many are in Z*p.

Returning to the ElGamal algorithm, recall that the first step is to choose a random prime *p*, and a generator *g* of Z*p. The number *p* will be generated with a pseudo prime generator, but how can we find *g*? Will there always be a generator? As it turns out, there *will* always be a generator. We find this generator though a sophisticated version of the famed mathematical method, G*uess and Check*. First, we make a *guess* for *g*, and then we *check* to see if it generates the entire group. Actually, it's not as haphazard as that. As it turns out, *g* is a generator if and only if $g^{(p-1)/q}$ is not equal to *1*(modp) - where *q* is a prime divisor of p-1. For example, in Z*5, *p-1=4*. The only prime divisor of *4* is *2*, so we look for elements that are not equal to *1* when raised t the power *2*. We see that 2^2 is not equal to *1*, but 4^4 does equal *1*. Hence, *2* is a generator, and *4* is not. Also, note that *g* will be prime. If it were *not*, then *g*=ab so $g^n=(ab)^n=a^nb^n$. But Z*p is cyclic, so there must be a *g* such that $a=g^r$ and $b=g^s$. Then, $(ab)^n=g^{(rsn)}$. This cuts down on the numbers that must be tested; we need only test primes that satisfy the above criteria. In fact,

there will be exactly J(p-1) generators, where **J(n)** is the number of elements relatively prime to *n*.

So, having found *g*, we select an 1<=*a*<=(p-2) and compute g^a. How, exactly is g^a computed? Remember that *p*,*g*, and *a* are all very large integers. To simply do g*g*…g* for *a* times, would take far too long. Instead, we use the *Square and Multiply Algorithm*. The gist of this algorithm is to repeatedly square (modp). i.e. 2^7(mod5) is (2^2(mod5)(2^2(mod5)(2^2(mod5))(2)(mod5) The bit complexity (bound on number of bit operations) for this is ((lg(p))^3).

To encipher, the sender chooses 1<=*k*<=(p-2) and calculates g^k and $m*g^{ak}$, both *(modp)*. These are sent to the recipient. The decryption is via the inverse of g^{ak}. This is computed as above. Recall, all of these numbers are in Z*p, and *p* is prime, so they all have inverses. To see why this has to be, recall the idea of a generator. It generates Z*p by its powers. Thus, all the elements of Z*p may be written as some power of *g*. Suppose $a=g^n$. Then $g^n g^{((l-n)}=1$, where *order(g)=l*. Thus, $g^{(l-n)}$ is the inverse of g^n.

We must have *g* be a generator in this encryption scheme. Why? Let us see an example where *g* is not chosen as a generator:
If the group is Z*5, suppose we had chosen *4* as *g* and tried to encrypt *m=3*, with $(4^2)^3$ - (k=2, a=3). Then, we get *c=3*1=3(mod)5*. Oops! Our cipher text is the clear text. This is the reason; if *g* is a generator, g^k will never be equal to *1* for *k<=(p-2)*. Restrictions on the choice of *a* are for similar reasons.

The group, Z*p is used because of its properties and the fact that there are algorithms available to do the computations in a reasonable amount of time. Additionally, the map, *a*-->g^a, is what is known as a one-way function. That is, while feasible to compute in one direction, going the other way is not. That is, given *y= g^a(modp)*, it is not feasible to compute *a=$\log_g(y)$(modp)*. This is what is known as the *Discrete Logarithm Problem* (DLP). This problem is thought to be intractable, although algorithms exist in special cases. The following describes their complexity. [13]

Name	Complexity
Baby-Step-Giant-Step	$O(\sqrt{p})$
Silver-Pohlig-Hellman	polynomial in q, the greatest prime factor of $p-1$.
Index-Calculus	$O(e^{(1+o(1))\sqrt{\ln(p)\ln(\ln(p))}})$

To understand why the discrete logarithm problem is so difficult, review the simple ElGamal example in this document. The encryption was (*3,2*). This means $2^k=3$ and $m*(2^a)^k=2$ (where $g=2$ and a is the private key). In this example, Black hat could easily compute k and then a. Using this, he could find $(g^{(ak)})^{-1}$ and so compute m. However, the numbers used are very large in practice, and this is becomes infeasible very quickly as the size of the numbers increase.

The mathematics used in the construction of public key algorithms is very sophisticated. Among others, the ideas in group theory are used; particularly the properties of the group Z*p. There are continued efforts to prove the intractability of the DLP in this group, given current processor power - this remains an open problem. The difficulty is considered t be equivalent to the difficulty of factoring large integers into primes, another intractable problem. However, cryptographers must be ever-vigilant against Black Hat, as new developments in mathematics (and increased processor power), could render this problem tractable in the future. Interestingly, the mathematics in use for creating cryptosystems today was originally developed for a completely different purpose. Much of it came about from attempts to factor polynomials of degree larger than three into their prime factorizations. In fact,cryptography in general makes an excellent argument for scientific research for its own sake. As the unexpected use of abstract algebra in cryptosystems suggests, 'pure' mathematics, often thought of as its own reward, can be very 'practical' indeed!

BIBLIOGRAPHY

1. A.Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
http://www.cacr.math.uwaterloo.ca/hac/about/chap3.pdf

2. *An Introduction to Cryptography*. Network Associates, Inc., 1990-1999.

3. FIPS PUB 186
http://www.itl.nist.gov/fipspubs/186m

4. http://www.netcore.co.in/pgp.html

5. http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci214292,00.html

6. H.X. Mel, Doris Baker. *Cryptography Decrypted*. Addison-Wesley, 2001

7. Vlastimi Klima, Toas Rosa. *Attack on Private Signature Keys of the OPenPGP format, PGP programs and other applications compatible with OPenPGP*
http://www.i.cz/en/pdf/openPGP_attack_ENGvktr.pdf

8. Secure Hash Signature Standard
(SHS)(FIPS PUB 180-22)
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

9. Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 1996.

10. Carlisle M. Adams. Constructing Symmetric Ciphers
http://adonis.ee.queensu.ca:8000/

11. http://stud3.tuwien.ac.at/~e9825530/computerscience/aes/2_4.html

12. C. Adams. The CAST-128 Encryption Algorithm (RFC2144)
http://www2.hunter.com/docs/rfc/rfc2144.html

13. Bernhard Esslinger, Bartol Filipovic, Henric Kay, Roger Oyono and Jorg Conelius Schneider. Mathematics and Cryptography, CrypTool.com, 2002.
http://www.cryptool.com