Adding a new public key algorithm to the Gnu Privacy Guard

Hong Linh Thai, Christian Klos and Nina Bindel
Technische Universität Darmstadt, Germany
Email: {hong_linh.thai, christian.klos}@stud.tu-darmstat.de, nbindel@cdc.informatik.tu-darmstadt.de

Contents

I. Introduction

As part of our studies at the "Technische Universität Darmstadt", Germany, we added cryptographic primitives (signature and encryption scheme) to the *Gnu Privacy Guard* (GPG) software. More specifically, we integrated a lattice-based signature scheme [1] and (lateron) an encryption scheme to GPG. To this end, we worked our way through the documentation and code of GPG. In this document we summarize the steps we performed in order to integrate the new algorithms. We explain each step using code snippets from the original software in detail. To make our descriptions even more understandable, the depicted code examples contain at least one line above and below the modified parts which are marked by "+". Hence, it is easy to find them in the official code and reproduce the modifications. All code shown in this documentation is based on GPG and *Libgcrypt* cloned at 24^{th} November 2015 from git://git.gnupg.org/gnupg.git and git://git.gnupg.org/libgcrypt.git. A git repository with all performed code changes including the lattice-based signature scheme is available on github https://github.com/hlThai/gpg-TeslaRing. However, as the lattice-based encryption scheme is not officially published yet, the code for encryption scheme cannot be found on github. It will be added as soon the paper is published. Please note that the implementation's main focus was a proof of concept. Thus, we do not guarantee that the performed modifications are completely secure against side-channel attacks, buffer overflow attacks etc. We also do not recommend to modify your GPG on your own unless you are knowing what you are doing, since it could undermine your security. Comments, questions, and feedback is very welcomed.

II. Compiling GnuPG and all the necessary libraries

To compile the code, we use Debian 8.1. Nevertheless, the following description should hold for any other Linux distribution. Furthermore, we show a compile process where GPG is install in a seperate sub folder, as we do not want to to modify the officially installed GPG version that is used in productive environment. On a small side note the code for the signature and encryption scheme use special AVX2 processor instructions. Hence, it is only possible to compile and run the published code on a capable processor. Nevertheless the steps needed for adding a new algorithm to GPG are processor technology independent. The process is divided into several tasks: Installing the dependencies, compiling and installing the libraries, and finally compiling and installing GPG itself. We created a makefile which automatizes the compile steps but needs to be modified for the personal folder structure and preferences. The complete makefile can be found in our git repository.

A. Installing the dependencies

First, all dependencies for compiling have to be installed. This works as usual with Code 1 :

```
apt-get install autoconf automake libksba-dev libassuan-dev libpth-dev libgcrypt-dev pinentry-curses
    transfig gettext
```

Code 1. Bash command for installing the dependencies

The installation of pinentry-curses is optional as it is used to insert the secret key password while key generation. We install here the command line version of pinentry as we used a server where we had no X-Session to work with another pinentry version. As GPG is not able find the pinentry-curses executable by itself, we also need to add it to its configuration file (our compiled version and the productive version share the configuration files).Code 2 adds the pinentry program to the configuration file of GPG:

```
echo "pinentry-program $(which pinentry)" >> ~/.gnupg/gpg-agent.conf
```

Code 2. bash command for adding the pinentry version to the configuration file of GPG

B. Compiling the libraries

Before we can compile the libraries we need to create an output folder for the binaries. As we need the full path of the folder for the complete compile process, we recommend to store it in a variable, as shown in Code 3:

```
mkdir gpg2lattice
export P=$(pwd)/gpg2lattice
```

Code 3. bash command for storing the output folder in a local variable

GPG itself needs a lot of libraries to be compilable but most of them are sufficiently up-to-date. Hence, the only missing libraries to build are: *libgpgerror* and *libgcrypt*. *libgpgerror* is needed as the repository version is too old to work with GPG and *libgcrypt*. Because *libgcrypt* contains all the encrypt/decrypt/sign/verify algorithms, it is additionally modified to GPG in the scope of this project.

1) Compiling libgpgerror:

As *libgcrypt* depends on *libgpgerror* this is the first library that needs to be compiled and installed. We use version 1.21 which is available at https://www.gnupg.org/ftp/gcrypt/libgpg-error/libgpg-error-1.21.tar.bz2. Code 4 builds and makes *libgpgerror*:

```
./configure --prefix=$P
make
make install
```

Code 4. Bash command for building and making *libgpgerror*

2) Compiling libgcrypt:

Code 5 shows the compile process of *libgcrypt*,that is very similar to *libgpgerror*:

```
./autogen
./configure --prefix=$P --with-gpg-error-prefix=$P --enable-maintainer-mode --enable-static --disable-
    shared
make
make install
```

Code 5. Bash command for building and making *libgcrypt*

The parameter `--enable-maintainer-mode` is needed the first time to generate a *version.texi* file that contains the version infos and the information that the build version is not a release version. In our special case two additional flags were added to the configure command: `--enable static` and `--disable-shared`. These commands are needed, since the implementation for the lattice-based cryptography primitives is using special *AVX2* processor instructions. These *AVX2* processor instructions are compiled using qhasm [2] and they do not work in a shared dynamic library. Therefore, we enforce that *libgcrypt* is build as a static library.

3) Compiling Gnu Privacy Guard:

Compiling GPG is nearly the same as the compile process of the libraries but this time we need to define a system variable so that GPG is able to find all the earlier compiled libraries. This is showed in Code 6.

```
export LD_LIBRARY_PATH=$P/lib/
./autogen.sh
./configure --prefix=$P --with-gpg-error-prefix=$P --with-libgcrypt-prefix=$P --enable-maintainer-mode
make
make install
```

Code 6. Bash command for building and making GPG

C. Running Gnu Privacy Guard

Running the compiled GPG works similar as running a precompiled GPG. But before running GPG its necessary to start the correct gnupg agent in the folder `$P/bin/` with Code 7.

```
$P/bin/gpg-agent --daemon --write-env-file ./.gpg-agent-info --enable-ssh-support --debug-all --allow-
    preset-passphrase --verbose --log-file ./gpg-agent-verbose.lo
```

Code 7. Bash command for executing the gnupg agent

Afterwards run the *gpg2* executable in the folder `$P/bin/` as usual, e.g. `./gpg2 --gen-key`.

III. Basic workflow to add a new Algorithm to Gnu Privacy Guard (GPG)

In the following explanation, we divide the many necessary steps into semantic blocks. We use the algorithms of the lattice-based signature scheme as an example to describe the different tasks. We will also show additional changes that are needed when adding an encryption scheme. First, we start with explaining the changes in *libgcrypt*. Afterwards, we describe the modifications in GPG.

A. Changes in Libgcrypt

Libgcrypt is a dependency of GPG. It contains and manages the cryptographic functions and algorithms used, e.g., signature generation, key generation, encrypting and decrypting, hash functions, etc. In order to communicate with *libgcrypt*, GPG uses function calls, which are mapped to the internal ones of *libgcrypt*. The functions calls can be found in *libgcrypt/visibility.c*. Therefore, when adding a new cryptography algorithm in GPG it needs to be actually implemented in *libgcrypt*. More about *libgcrypt* can be found in the documentation [3].

1) Adding the algorithm definition to libgcrypt:

Libgcrypt and GPG both contain integer representations of their algorithms which are stored as global constants. Hence, when adding a new algorithm a new constant is needed. In *libgcrypt* they are stored in the file *src/gcrypt.h.in*. Code 8 shows the necessary changes.

```
    GCRY_PK_ECDH  = 302,    /* (only for external use).  */
    GCRY_PK_EDDSA = 303,     /* (only for external use).  */
+   GCRY_PK_LATTICE = 400    /* lattice-based signature scheme in developement */
  };
```

Code 8. Adding a new algorithm constant to *src/gcrypt.h.in*

In the next step, the new algorithm needs to be added to the list of available public-key ciphers in the *configure.ac* file in *libgcrypt*, as shown in Code 9.

```
# Definitions for public-key ciphers.
+available_pubkey_ciphers="dsa elgamal rsa ecc lattice_sig"
enabled_pubkey_ciphers=""
```

Code 9. Adding a new algorithm definition to *configure.ac*

In order to enable our new algorithm for usage, an additional check needs to be added to the same file. The Code 10 verifies that if the compiled file for the algorithm is available.

```
+LIST_MEMBER(lattice_sig, $enabled_pubkey_ciphers)
+if test "$found" = "1" ; then
+   GCRYPT_PUBKEY_CIPHERS="$GCRYPT_PUBKEY_CIPHERS \
+                           lattice.lo"
+   AC_DEFINE(USE_LATTICE, 1, [Defined if this module should be included])
+fi
```

Code 10. Adding an additional check to *configure.ac*

In our case the file of the lattice-based signature scheme is lattice.c. And lattice.lo is the output from the build process of *libgcrypt*.

Depending on the latter check the new added algorithm will be shown in the list of supported algorithms. Code 11 implements this functionality in the file *libgcrypt/cipher/pubkey.c*.

```
/* This is the list of the public-key algorithms included in Libgcrypt. */
static gcry_pk_spec_t *pubkey_list[] = {
  ...
#if USE_ELGAMAL
    &_gcry_pubkey_spec_elg,
#endif
+ #if USE_LATTICE
+      &_gcry_pubkey_spec_lattice,
+ #endif
    NULL
};
```

Code 11. Adding the new algorithm to the list of supported algorithms

The list of supported algorithms can be displayed by executing the *libgcrypt-config* executable in the *bin* folder of the build folder using the `--algorithms` command.

2) [Algorithm].c or the definition of the algorithm itself:

The complete cryptographic algorithm logic is usually defined in one file that needs to be created in the folder *libgcrypt/cipher*. Basically this file contains a definition of the algorithm which we will show in Code 12 here for the lattice-based signature scheme and the functions specified in the algorithm definition. The important point is that the new algorithm needs to implement the interface `gcry_pk_spec_t` for public keys.

```
+gcry_pk_spec_t _gcry_pubkey_spec_lattice =
+  {
+    GCRY_PK_LATTICE, { 0, 1 },
+    GCRY_PK_USAGE_SIGN,
+    "LATTICE", lattice_names,
+    "e", "ed", "ab", "s", "e",
+    lattice_generate,
+    lattice_check_secret_key,
+    NULL,
+    NULL,
+    lattice_sign,
+    lattice_verify,
+    lattice_get_nbits,
+    NULL,
+    NULL
+  };
```

Code 12. Implemented interface for the lattice-based signature scheme

Note that in case an encryption scheme is implemented that one needs to define the functions for encrypt and decrypt instead of sign and verify. For further explanation on all the parameters and the function headers of the interface we recommend reading the documentation which can be found at [4].

Additionally, a forward declaration to this interface is needed in *libgcrypt/src/cipher.h*, shown in Code 13

```
 extern gcry_pk_spec_t _gcry_pubkey_spec_dsa;
 extern gcry_pk_spec_t _gcry_pubkey_spec_ecc;
+extern gcry_pk_spec_t _gcry_pubkey_spec_lattice;
```

Code 13. Forward declaration in libgcrypt/src/cipher.h for the new algorithm

3) Putting all together in the makefile:

The only changes that added new files were done in *libgcrypt*. Thus, these new created files need to be included in the build process of *libgcrypt*. As *libgcrypt* generates its makefile and its configure file with *autogen* we need to modify the *libgcrypt/cipher/Makefile.am*. `libcipher_la_SOURCES` is build before `EXTRA_libcipher_la_SOURCES`, so when there are any dependencies it is possible to sort them by splitting the classes into this variables.

If there are any special GNU Compiler Collection (gcc) flags necessary like in our case `-mavx2 -msse2avx -march=corei7-avx -lm -fPIC -lmpfr -lgmp` they needed to be added into *libgcrypt/configure.ac* the corresponding line, as shown in Code 14.

```
if test "$GCC" = yes; then
+ CFLAGS="$CFLAGS -Wall -mavx2 -msse2avx -march=corei7-avx -O3 -fomit-frame-pointer -Wextra -g -lm -fPIC -
    lmpfr -lgmp"
...
```

Code 14. Adding cflags to libgcrypt in file *libgcrypt/configure.ac*

Depending if the flags `--enable static` and `--disable-shared` are used or not, one needs also to add additional cflags to GPG in the file *gnupg/configure.ac*.

B. Changes in Gnu Privacy Guard

In this section we show all changes that have to be made in GPG to add a new public key algorithm to GPG.

1) Adding the algorithm definition to GPG and the mapping functions to libgcrypt:

Code 15 shows the algorithm definitions for public key encryption in GPG, that are stored in *include/cipher.h*. Similar to our step in *libgcrypt* we need to define here a new constant for the lattice-based signature scheme.

```
 #define PUBKEY_ALGO_ECDSA           19
 #define PUBKEY_ALGO_ELGAMAL         20 /* Elgamal encr+sign */
+#define PUBKEY_ALGO_LATTICE         23 /* lattice-based sign in development */
 #define PUBKEY_USAGE_SIG     GCRY_PK_USAGE_SIGN  /* Good for signatures. */
```

Code 15. Definition of the public key algorithm constants in GPG in the file *include/cipher.h*.

As GPG needs to know the representation in *libgcrypt* if they differ from each other a mapping function is needed. Code 16 shows this modification in the file *g10/misc.c*:

```
     case PUBKEY_ALGO_ECDH:      return 302 /*GCRY_PK_ECDH*/;
     case PUBKEY_ALGO_ELGAMAL_E: return GCRY_PK_ELG;
+    case PUBKEY_ALGO_LATTICE:  return GCRY_PK_LATTICE;
     default: return algo;

openpgp_pk_algo_usage ( int algo ) {

+       case PUBKEY_ALGO_LATTICE:
+         use = PUBKEY_USAGE_SIG;
+         break;
        default:
          break;
}

pubkey_nbits( int algo, gcry_mpi_t *key ) {

+    else if (algo == PUBKEY_ALGO_LATTICE) {
+        rc = gcry_sexp_build ( &sexp, NULL,"(public-key(lattice-enc(e%m)))",key[0] );
+    }

}
```

Code 16. Changes in the file *g10/misc.c*: Extending the integer representation of public key algorithms. Extending the mapping of the function `pubkey_nbits` and adding internal information for lattice-based signature scheme.

Furthermore, Code 16 shows the necessary modification of the file *g10/misc.c* to add information about the usage of the algorithm, since GPG does not fetch the information from *libgcrypt*. In case a public-key encryption scheme should be implemented one needs to add `PUBKEY_USAGE_ENC` instead of `PUBKEY_USAGE_SIG`. Additionally, the function `pubkey_nbits` needs to be modified. This function returns the number of bits needed for the modulus of a given pubkey for a given public key algorithm by calling the respective function in *libgcrypt*. It is very important for the usage of padding, e.g. for encryption with a public key GPG creates a symmetric key and adds the correct padding to the symmetric key, which will be encrypted with the public key. Thus, if the function `get_nbits` of a public key cryptography algorithm is not implemented correctly, encryption will probably fail.

In the next step we need to add the mapping for the basic functionality of the algorithm such as, key generation, signing and verifying a message etc. This is done in GPG in the file *gnupg/g10/pkglue.c* at the corresponding places. Here in Code 17 we give an example for sign, verify, and check secret key. The encrypt and decrypt algorithm can be modified in the same pattern:

```

int
pk_sign (int algo, gcry_mpi_t * data, gcry_mpi_t hash, gcry_mpi_t * skey)
{

/* make a sexp from skey */
+   else if (algo == PUBKEY_ALGO_LATTICE) {
+     rc = gcry_sexp_build (&s_skey, NULL, "(private-key(lattice(e%m)(d%m)))",
        skey[0], skey[1]);
+   }
}

int
pk_verify (int algo, gcry_mpi_t hash, gcry_mpi_t * data, gcry_mpi_t * pkey)
{

/* make a sexp from pkey */
+  else if (algo == PUBKEY_ALGO_LATTICE) {
+   rc = gcry_sexp_build (&s_pkey, NULL, "(public-key(lattice(e%m)))", pkey[0]);
+  }

/* Put data into a S-Exp s_sig. */
+  else if (algo == PUBKEY_ALGO_LATTICE) {
+   if (!data[0])
+     rc = gpg_error (GPG_ERR_BAD_MPI);
+   else
+   rc = gcry_sexp_build (&s_sig, NULL, "(sig-val(lattice(s%m)))", data[0]);
+  }
}

int
pk_check_secret_key (int algo, gcry_mpi_t *skey)
{
+  else if(algo == PUBKEY_ALGO_LATTICE){
+   rc = gcry_sexp_build (&s_skey, NULL, "(private-key(lattice(e%m)(d%m)))",
      skey[0], skey[1]);
  }
}
```

Code 17. Extending the mapping for sign, verify and check secret key for lattice-based signatures in the file *gnupg/g10/pkglue.c*

In all functions a S-expression is build containing the needed information e.g. public key or the signature, which needs to be transferred from GPG to *libgcrypt* to perform the cryptographic function. S-expression are LISP like objects used by public key functions to pass complex data structure around. More information on them can be found under [5]. These S-expression are passed to the corresponding cryptographic function of *libgcrypt*, e.g. `pk_sign` will call the signing function of lattice based encryption `lattice_sign` in the file *libgcrypt/cipher/lattice.c* with the created S-expression as argument.

2) Key Generation Function:

For the key generation in GPG following changes have to be made in the file *g10/keygen.c* containing of multiple parts: The first change, shown in Code 18, is to modify the function `ask_algo`. This function is responsible for handling the user input of key generation and printing out the respective lines on the console. Therefore, we need to add text options, where the user can choose the new algorithm.

```
static int
ask_algo (int addmode, int *r_subkey_algo, unsigned int *r_usage)
{
if (opt.expert)
    {
      tty_printf (_("   (%d) DSA (set your own capabilities)\n"), 7 );
      tty_printf (_("   (%d) RSA (set your own capabilities)\n"), 8 );
    }
+ tty_printf (_("   (%d) Lattice-Based-Cryptography (sign only)\n"), 10);

```

```
...

+ else if ((algo == 10 || !strcmp (answer, "lattice/s")))
+    {
+     algo = PUBKEY_ALGO_LATTICE;
+     *r_usage = PUBKEY_USAGE_SIG;
+     break;
+ }
}
```

Code 18. Extending user input handling for a new algorithm in the file *g10/keygen.c*

The second change is the function `do_create`, which is responsible for the basic key generation and diverts the key generation to the actual function. This is shown in Code 19

```
static int
do_create (int algo, unsigned int nbits, KBNODE pub_root, KBNODE sec_root,
     DEK *dek, STRING2KEY *s2k, PKT_secret_key **sk,
           u32 timestamp, u32 expiredate, int is_subkey )
{

...
   else if( algo == PUBKEY_ALGO_RSA )
      rc = gen_rsa(algo, nbits, pub_root, sec_root, dek, s2k, sk, timestamp, expiredate, is_subkey);
+  else if( algo == PUBKEY_ALGO_LATTICE)
+     rc = gen_lattice(algo, nbits, pub_root, sec_root, dek, s2k, sk, timestamp, expiredate, is_subkey);

}
```

Code 19. Extending the basic key generation function for a new algorithm in the file *g10/keygen.c*

The third change is the creation of the actual function, which is responsible for the generation of the key. In our case this is `gen_lattice`, shown in Code 20, which is adapted from the other algorithms. In this function first a S-expression is created with the important parameters for the key generation. In our case the number of bits is not used in key generation as the implementation of lattice based cryptography uses only one parameter set, however the name of the algorithm is important since it has to fit with defined one in *libgcrypt*. This S-expression is given as an argument to the key generation function of lattice and afterwards the public and the secret key is retrieved from the returned S-expression. Afterwards the secret key is protected with the passphrase from the user. In the last step the key packets for the public and private key are build. More information about the key generation process of GPG can be found in appendix A.

```

/*
 * Generate a lattice-based signature key.
 */
static int
gen_lattice (int algo, unsigned nbits, KBNODE pub_root, KBNODE sec_root, DEK *dek,
         STRING2KEY *s2k, PKT_secret_key **ret_sk,
         u32 timestamp, u32 expireval, int is_subkey)
{
    int rc;
    PACKET *pkt;
    PKT_secret_key *sk;
    PKT_public_key *pk;
    gcry_sexp_t s_parms, s_key;

    assert (is_LATTICE(algo));

    if (!nbits)
      nbits = DEFAULT_STD_KEYSIZE;

    /* create the s-exp which is send to libgcrypt */
    rc = gcry_sexp_build (&s_parms, NULL,
                          "(genkey(lattice(nbits %d)))",
                          (int)nbits);
    if (rc)
      log_bug ("gcry_sexp_build failed: %s\n", gpg_strerror (rc));

    /* call the gen key function of lattice */
    rc = gcry_pk_genkey (&s_key, s_parms);
    gcry_sexp_release (s_parms);
    if (rc)
      {
        log_error ("gcry_pk_genkey failed: %s\n", gpg_strerror (rc) );
        return rc;
      }
```

```

    sk = xmalloc_clear( sizeof *sk );
    pk = xmalloc_clear( sizeof *pk );
    sk->timestamp = pk->timestamp = timestamp;
    sk->version = pk->version = 4;
    if (expireval)
       {
         sk->expiredate = pk->expiredate = sk->timestamp + expireval;
       }
    sk->pubkey_algo = pk->pubkey_algo = algo;

    /* extract the public key from the returned sexp */
    rc = key_from_sexp (pk->pkey, s_key, "public-key", "e");
    if (rc)
       {
         log_error ("key_from_sexp failed: %s\n", gpg_strerror (rc));
         gcry_sexp_release (s_key);
         free_public_key(pk);
         free_secret_key(sk);
         return rc;
       }

    /* extract the secret key from the returned sexp */
    rc = key_from_sexp (sk->skey, s_key, "private-key", "ed");
    if (rc)
      {
        log_error ("key_from_sexp failed: %s\n", gpg_strerror (rc) );
        gcry_sexp_release (s_key);
        free_public_key(pk);
        free_secret_key(sk);
        return rc;
      }
    gcry_sexp_release (s_key);

    sk->is_protected = 0;
    sk->protect.algo = 0;

    sk->csum  = checksum_mpi (sk->skey[1] );
    if( ret_sk ) /* return an unprotected version of the sk */
      *ret_sk = copy_secret_key( NULL, sk );

    /* protect the secret key */
    rc = genhelp_protect (dek, s2k, sk);
    if (rc)
      {
        free_public_key (pk);
        free_secret_key (sk);
        return rc;
      }

    /* build the key packets */
    pkt = xmalloc_clear(sizeof *pkt);
    pkt->pkttype = is_subkey ? PKT_PUBLIC_SUBKEY : PKT_PUBLIC_KEY;
    pkt->pkt.public_key = pk;
    add_kbnode(pub_root, new_kbnode( pkt ));

    pkt = xmalloc_clear(sizeof *pkt);
    pkt->pkttype = is_subkey ? PKT_SECRET_SUBKEY : PKT_SECRET_KEY;
    pkt->pkt.secret_key = sk;
    add_kbnode(sec_root, new_kbnode( pkt ));

  return 0;
}
```

Code 20. The actual key generation function for a lattice-based signature key in the file *g10/keygen.c*

3) Key information:

To support a correct display of lattice keys, when using the command `--list-keys` the file *g10/keyid.c* needs to be modified. Code 21 shows the necessary modifications.

```
pubkey_letter( int algo )
{
...
       case PUBKEY_ALGO_ECDSA:  return 'E' ;  /* ECC DSA (sign only)   */
       case PUBKEY_ALGO_ECDH: return 'e' ;  /* ECC DH (encrypt only) */
+      case PUBKEY_ALGO_LATTICE: return 'L' ;    /* lattice-based (sign only)   */
       default: return '?';
```

```
}
```

Code 21. Extending the key information display to support lattice-based signatures in the file *g10/keyid.c*.

4) Supporting public key encryption:

In case that a public key encryption algorithm should be implemented an additional change in the file *g10/mainproc.c* is necessary. Code 22 shows the modification of the function `proc_pubkey_enc`, which is responsible for the handling of public key encryption packets.

```
proc_pubkey_enc( CTX c, PACKET *pkt )
{
     else if( enc->pubkey_algo == PUBKEY_ALGO_ELGAMAL_E
              || enc->pubkey_algo == PUBKEY_ALGO_RSA
              || enc->pubkey_algo == PUBKEY_ALGO_RSA_E
              || enc->pubkey_algo == PUBKEY_ALGO_ELGAMAL
+             || enc->pubkey_algo == PUBKEY_ALGO_LATTICE_E) {
         ...
    }

}
```

Code 22. Modification of `proc_pubkey_enc` in the file *g10/mainproc.c*.

IV. Special modifications for lattice-based cryptography and handling huge Keys

The lattice-based signature scheme uses in the current parameter setting a public key of size 16 760 832 bits and a private key of 29 652 736 bits, i.e. 2 095 104 bytes and 3 706 592 bytes, respectively. However, GPG and libgcrypt are only supporting keys by default up to a keysize of 4K bits or with some special commands (`--enable-large-secmem` and `--enable-large-rsa`) keysizes up to around 15K bits. The reason for this choice is that according to NIST RSA with a public key of 4096 bits has a security level between AES-128 and AES-192 and that RSA with a public key of 15360 bits is comparable to AES-256 [6]. But as lattice-based keys are much larger than this limit, implementations in GPG and libgcrypt needs to be changed in order to support these huge keys.

A. Larger Secure Memory

GPG uses a secure memory to store important variables during internal computation such as private keys. This secure memory is locked by GPG so that it can not be accessed by any other process and such that it will not be written to swap. However, by default only 32 768 bytes are used(, 65 536 bytes when used with the command `--enable-large-secmem`). But for lattice-based cryptography this is still too small. Therefore, a `--enable-huge-secmem` command was added which uses a secure memory of 16 777 216 bytes, by modifying the file *gnupg/configure.ac*. This is shown in Code 23.

```
AC_MSG_CHECKING([whether to allocate extra secure memory])
AC_ARG_ENABLE(large-secmem,
              AC_HELP_STRING([--enable-large-secmem],
                             [allocate extra secure memory]),
              large_secmem=$enableval, large_secmem=no)
AC_MSG_RESULT($large_secmem)

+AC_MSG_CHECKING([whether to allocate huge secure memory which is needed for lattice based crypto])
+AC_ARG_ENABLE(huge-secmem,
+              AC_HELP_STRING([--enable-huge-secmem],
+                             [allocate extra huge secure memory]),
+              huge_secmem=$enableval, huge_secmem=no)
+AC_MSG_RESULT($huge_secmem)
+if test "$huge_secmem" = yes ; then
+   SECMEM_BUFFER_SIZE=16777216
+else
if test "$large_secmem" = yes ; then
   SECMEM_BUFFER_SIZE=65536
else
   SECMEM_BUFFER_SIZE=32768
fi
+fi
```

Code 23. Adding a huge secmem command to GPG *gnupg/configure.ac*.

B. Extended Multi-Precision Integers

GPG and libgcrypt use MPI, which are unsigned integers to hold large integers, especially the ones used in cryptographic calculations. However, when storing this MPIs to the disk in their corresponding packets, GPG uses the definition of the OpenPGP message format, which caps them at 65 535 bit. This OpenPGP message format can be found in the RFC 2440 [7] respectively in the RFC 4880 [8]. This format defines MPIs as a two piece: a two-octet scalar that is the length of the MPI in bits followed by a string of octets that contain the actual integer. However, the two-octet scalar is not large enough for our keys, since it can only encode a length up to 65 535 bits. Thus, we create an own extension, which we call extended Multi-Precision Integer. Similar to MPIs we define extended MPIs as a two piece, but we are using a four-octet scalar for the length instead. Doing so extended MPIs support upto a length of 536 870 911 bits and the size of an extended MPI is ((MPI.length + 7) / 8) + 4 octets.

Examples: (all numbers are in hexadecimal)

The string of octets [00 00 00 01 01] form an MPI with the value 1. The string [00 00 00 09 01 FF] form an MPI with the value 511.

To support these extended MPIs we need to change following parts of GPG: In the file *gnupg/g10/build-packet.c* we add a new function to write these extended MPIs to a buffer given an MPI in USG format, as shown in Code 24.

```
+static int
+mpi_write_extended (iobuf_t out, gcry_mpi_t a)
+{
+  char *buffer; /* 4 is for the mpi length. */
+  size_t nbytes;
+  int rc;
+
+  buffer = xmalloc((MAX_EXTERN_EXTENDED_MPI_BITS+7)/8+4);
+
+  nbytes = (MAX_EXTERN_EXTENDED_MPI_BITS+7)/8+4;
+
+  /* write first the length of the MPI in the buffer */
+  unsigned long lenMPIbytes;  // length of the mpi
+  unsigned long lenMPIbits;   // length of the mpi in bits
+  unsigned char byteArray[4]; // array where we store the length byte wise
+
+  // compute length of the MPI
+  rc = gcry_mpi_print (GCRYMPI_FMT_USG, NULL, 0, &lenMPIbytes, a );
+  lenMPIbits = lenMPIbytes*8;
+
+  // transform the length in to bytes so that we can write them into the buffer
+  byteArray[0] = (int)((lenMPIbits & 0xFF000000) >> 24 );
+  byteArray[1] = (int)((lenMPIbits & 0x00FF0000) >> 16 );
+  byteArray[2] = (int)((lenMPIbits & 0x0000FF00) >> 8 );
+  byteArray[3] = (int)((lenMPIbits & 0X000000FF));
+
+  // now write the bytes into the buffer using some pointer arithmetic
+  *(buffer) = byteArray[0];
+  *(buffer+1) = byteArray[1];
+  *(buffer+2) = byteArray[2];
+  *(buffer+3) = byteArray[3];
+
+  rc = gcry_mpi_print (GCRYMPI_FMT_USG, (buffer+4), nbytes, &nbytes, a );
+  if( !rc ) {
+    rc = iobuf_write( out, buffer, nbytes+4 ); // add aditional 4 bytes for the header
+  }
+  else if (gpg_err_code(rc) == GPG_ERR_TOO_SHORT )
+    {
+      log_info ("mpi too large (%ld bits)\n", gcry_mpi_get_nbits (a));
+      /* The buffer was too small. We better tell the user about the MPI. */
+      rc = gpg_error (GPG_ERR_TOO_LARGE);
+    }
+
+  return rc;
+}
```

Code 24. Adding a new function to the file *gnupg/g10/build-packet.c*, which handles the storage of extended MPIs to the disk

Furthermore we need to add a function for parsing the written extended MPIs in the file *gnupg/g10/parse-packet.c*, as shown in Code 25

```
+static gcry_mpi_t
+mpi_read_extended (iobuf_t inp, unsigned int *ret_nread, int secure)
```

```
+{
+  int c, c1, c2, c3, c4, i;
+  unsigned int nmax = *ret_nread;
+  unsigned long nbits, nbytes;
+  size_t nread = 0;
+  gcry_mpi_t a = NULL;
+  byte *buf = NULL;
+  byte *p;
+
+  if (!nmax)
+    goto overflow;
+
+  // read the first byte
+  if ( (c = c1 = iobuf_get (inp)) == -1 )
+    goto leave;
+  if (++nread == nmax)
+    goto overflow;
+  nbits = c << 24;
+  // read the second byte
+  if ( (c = c2 = iobuf_get (inp)) == -1 )
+    goto leave;
+  if (++nread == nmax)
+    goto overflow;
+  nbits |= c << 16;
+  // read the third byte
+  if ( (c = c3 = iobuf_get (inp)) == -1 )
+    goto leave;
+  if (++nread == nmax)
+    goto overflow;
+  nbits |= c << 8;
+  // read the fourth byte
+  if ( (c = c4 = iobuf_get (inp)) == -1 )
+    goto leave;
+  ++nread;
+  nbits |= c;
+
+  // convert the bytes into an long again
+
+  if ( nbits > MAX_EXTERN_EXTENDED_MPI_BITS )
+    {
+      log_error("mpi too large (%ld bits)\n", nbits);
+      goto leave;
+    }
+
+  nbytes = (nbits+7) / 8;
+  // allocating buffer
+  buf = secure ? gcry_xmalloc_secure (nbytes + 4) : gcry_xmalloc (nbytes + 4);
+  p = buf;
+  p[0] = c1;
+  p[1] = c2;
+  p[2] = c3;
+  p[3] = c4;
+
+  for ( i=0 ; i < nbytes; i++ ) // reading the mpi byte per byte into the buffer
+    {
+      p[i+4] = iobuf_get(inp) & 0xff;
+      if (nread == nmax)
+        goto overflow;
+      nread++;
+    }
+
+  if (nread >= 4 && !(buf[0] << 24 | buf[1] << 16 | buf[2] << 8 | buf[3]))
+    {
+      /* Libgcrypt < 1.5.0 accidently rejects zero-length (i.e. zero)
+         MPIs.  We fix this here.  */
+      a = gcry_mpi_new (0);
+    }
+  else
+    {
+      if ( gcry_mpi_scan( &a, GCRYMPI_FMT_USG, (buf+4), nbytes, &nread ) )
+        a = NULL;
+    }
+
+  *ret_nread = (nread+4); //add additional 4 bytes for the length header
+  gcry_free(buf);
+  return a;
+
```

```
+ overflow:
+  log_error ("mpi larger than indicated length (%ld bits)\n", 8*nmax);
+ leave:
+  *ret_nread = nread;
+  gcry_free(buf);
+  return a;
}
```

Code 25. Adding a new function to the file *gnupg/g10/parse-packet.c*, which is responsible for reading stored extended MPIs from the disk

Also we add a macro similar to MPI, which defines the limit of bits, that are accepted when reading or writing extended MPIs in *gnupg/g10/gpg.h*, shown in Code 26.

```
+ #define MAX_EXTERN_EXTENDED_MPI_BITS 1073741824
```

Code 26. Adding a macro for the limit when writing or reading extended MPIs in the file *gnupg/g10/gpg.h*

C. Additional Changes in GPG for extended MPIs

Since we are using extended MPIs respectively unsigned MPIs for all the integers used for lattice-based cryptography, more files needed to be adapted.

GPG uses packets, as defined in the OpenPGP Message Format in the RFC 4880 [8], for every type of message, e.g. to a secret or public key in the keyring or to write a signature on the disk. For this reason changes needs to be made at the corresponding functions to support extended MPIs. Code 27 shows this changes in the file *gnupg/g10/build-packet.c*.

```
static int
do_public_key( IOBUF out, int ctb, PKT_public_key *pk ) {

+  if( pk->pubkey_algo != PUBKEY_ALGO_LATTICE && pk->pubkey_algo != PUBKEY_ALGO_LATTICE_E)
    for (i=0; i < n && !rc ; i++ )
      rc = mpi_write(a, pk->pkey[i] );
+  else
+    // case lattice
+    for (i=0; i < n && !rc ; i++ )
+     rc = mpi_write_extended(a, pk->pkey[i]);

}

static int
do_secret_key( IOBUF out, int ctb, PKT_secret_key *sk ) {

+  if( sk->pubkey_algo != PUBKEY_ALGO_LATTICE && sk->pubkey_algo != PUBKEY_ALGO_LATTICE_E)
    for (i=0; i < npkey; i++ )
      if ((rc = mpi_write (a, sk->skey[i])))
        goto leave;
+  else
+   // except that for lattice we use our own function
+   for (i=0; i < npkey; i++ )
+     if ((rc = mpi_write_extended(a, sk->skey[i])))
+       goto leave;

}

static int
do_signature( IOBUF out, int ctb, PKT_signature *sig ) {

+  if (sig->pubkey_algo != PUBKEY_ALGO_LATTICE)
      for (i=0; i < n && !rc ; i++ )
            rc = mpi_write(a, sig->data[i] );
+  else
+     // use extended mpi write since lattice signatures could be large too
+     rc = mpi_write_extended(a, sig->data[0] ); // lattice has only one signature element

}

static int
do_pubkey_enc( IOBUF out, int ctb, PKT_pubkey_enc *enc )
{

  for (i=0; i < n && !rc ; i++ )
+   if(enc->pubkey_algo == PUBKEY_ALGO_LATTICE_E)
+     rc = mpi_write_extended(a, enc->data[i] );
+   else
```

```
      rc = mpi_write(a, enc->data[i] );

}
```

Code 27. Adding extended MPIs to the corresponding write packet functions in the file *gnupg/g10/build-packet.c*

In our case we use extended MPIs only for our newly implemented algorithms, since we do not want to change anything in the functionality of the existing algorithms. But if lattice-based cryptography should be included in the future in GPG, it is advised to completely change everything to extended MPIs.

Code 28 shows the changes in the file *gnupg/g10/parse-packet.c*, which are necessary for adding extended MPIs to the parse packet functionality.

```
+ #define MAX_KEY_PACKET_LENGTH (1048576 * 1024)

static int
parse_key (IOBUF inp, int pkttype, unsigned long pktlen,
           byte *hdr, int hdrlen, PACKET *pkt)
{

  for(i=0; i < npkey; i++ ) {
        n = pktlen;
+       if(algorithm != PUBKEY_ALGO_LATTICE && algorithm != PUBKEY_ALGO_LATTICE_E)
              sk->skey[i] = mpi_read(inp, &n, 0 );
+       else
+         sk->skey[i] = mpi_read_extended(inp, &n, 0);
          pktlen -=n;
          ...
  }

int
parse_signature( IOBUF inp, int pkttype, unsigned long pktlen,
            PKT_signature *sig )
{

- if (pktlen > (5 * MAX_EXTERN_MPI_BITS/8))
+ if (pktlen > (5 * MAX_EXTERN_EXTENDED_MPI_BITS/8))

  for( i=0; i < ndata; i++ ) {
      n = pktlen;
-     sig->data[i] = mpi_read(inp, &n, 0 );
+     if (sig->pubkey_algo != PUBKEY_ALGO_LATTICE)
+       sig->data[i] = mpi_read(inp, &n, 0 );
+     else
+       sig->data[i] = mpi_read_extended(inp, &n, 0 );
+     }
      pktlen -=n;
      ...
}

}
```

Code 28. Adding extended MPIs to the corresponding parse packet functions in the file *gnupg/g10/parse-packet.c*

Furthermore, the file *gnupg/g10/seckey-cert.c*, responsible for protecting and unprotecting the secret key with the passphrase given by the user needs to be changes as shown in Code 29.

```
static int
do_check( PKT_secret_key *sk, const char *tryagain_text, int mode,
          int *canceled )
{

  for( ; i < pubkey_get_nskey(sk->pubkey_algo); i++ ) {
+   if(sk->pubkey_algo == PUBKEY_ALGO_LATTICE || sk->pubkey_algo == PUBKEY_ALGO_LATTICE_E) {
+     if ( gcry_mpi_scan( &sk->skey[i], GCRYMPI_FMT_USG, p+2, ndata-22, &nbytes))
+     // subtract 22 manually since we use sha1 checksum (20) and don't use the pgp format (2) for lattice
      keys respectively add 2
+       {
+       /* Checksum was okay, but not correctly decrypted.  */
+       sk->csum = 0;
+       csum = 1;
+       break;
+       }
+   } else {
         ...
      }
```

```
  }

}
```

Code 29. Modifying the function responsible for protecting and unprotecting the secret key to work with extended MPIs in the file *gnupg/g10/seckey-cert.c*

The last change is Code 30 concerning the fingerprint of lattice-based signature public keys, which is computed in the file *gnupg/g10/keyid.c*

```
hash_public_key( gcry_md_hd_t md, PKT_public_key *pk )
{
     for(i=0; i < npkey; i++ )
       {
+      const enum gcry_mpi_format fmt = ((pk->pubkey_algo==PUBKEY_ALGO_LATTICE) ? GCRYMPI_FMT_USG : GCRYMPI_FMT_PGP);
-      if (gcry_mpi_print (GCRYMPI_FMT_PGP, NULL, 0, &nbytes, pk->pkey[i]))
+      if (gcry_mpi_print (fmt, NULL, 0, &nbytes, pk->pkey[i]))
           BUG ();
       pp[i] = xmalloc (nbytes);
       ...
      }
}
```

Code 30. Supporting fingerprints of lattice-based signature keys in the file *gnupg/g10/keyid.c*

V. Enigmail

Enigmail is not part of GPG, but it uses GPG and provides some kind of user interface, which helps to visualize the implemented functionality. Additionally it is useful for encrypting mails. Hence, we modified Enigmail to support the new lattice-based algorithm added to GPG.
Basically Enigmail works out of the box when there are new and unknown algorithms in GPG or when you receive a mail or import a key that uses this new algorithm, as long the new algorithm is supported by your GPG. However, key generation needs some small modifications and also by default appear only the algorithm number is displayed and not the name. We will show in the subsection V-B how this can be fixed.

A. Compiling and Installing

Enigmail contains a configure and make file so it is basically build by running Code 31.

```
./configure
make
```

Code 31. Bash commands to compile and build enigmail

The so compiled xpi can be found in the build folder and is installed in the normal add-on installation procedure to Thunderbird. However, after installing the add-on one needs to modify in the preferences the path of *GPG2*, as it usually points to the one in *usr/bin* and not to the modified version.

B. Algorithm names

First of all the algorithm numbers, which we defined in GPG (section III-B1), needs to be paired with names. This is done in Code 32 in the main language file *ui/locale/en-US/enigmail.properties*:

```
 keyAlgorithm_22=EDDSA
+keyAlgorithm_23=LATTICE
+keyAlgorithm_24=LATTICE-ENC
 keyUsageEncrypt=Encrypt
```

Code 32. Adding new algorithm names to Enigmail in the main language file *ui/locale/en-US/enigmail.properties*

The same modification can be applied to all the language files found in *lang/de/enigmail.properties*.
Enigmail contains another function that transforms algorithm numbers to algorithm names in the file *package/gpg.jsm* where we also need to add the corresponding names, shown in Code 33:

```
       case 22:
         return "EDDSA";
+      case 23:
+        return "LATTICE";
+      case 24:
+        return "LATTICE-ENC";
       default:
```

Code 33. Adding algorithm name mapping for lattice-based cryptography schemes in the file *package/gpg.jsm*

C. Entry at keyGeneration

To add an entry at the key generation, we need to modify the UI which is build in the *ui/content/enigmailKeygen.xul*. This is shown in Code 34, where an additional entry in xml style is added.

```
  <menupopup id="keyTypePopup">
+   <menuitem id="keyType_lattice" value="3" label="&enigmail.keyGen.keyType.lattice;"/>
    <menuitem id="keySize_rsa" value="2" label="&enigmail.keyGen.keyType.rsa;" selected="true"/>
```

Code 34. Adding a menu item for lattice-based signature key generation in *ui/content/enigmailKeygen.xul*

We see in Code 34 that all the menu entries have a value and a label. The label is defined in another language file: *ui/locale/en-US/enigmail.dtd* (and their equivalents in *lang*). Therefore, we need to change the language file to define the label, as shown in Code 35

```
 <!ENTITY enigmail.keyGen.keyType.rsa      "RSA">
+<!ENTITY enigmail.keyGen.keyType.lattice  "LATTICE">

 <!ENTITY enigmail.preferences.label       "Enigmail Preferences">
```

Code 35. Defining the label entry for the new added menu item in *ui/locale/en-US/enigmail.dtd*

The value in Code 34 defines which function is called when the list entry is selected and we start the key generation. To define the correct logic we have to add the correct mapping for the value in the file *package/keyRing.jsm*, as shown in Code 36.

```
 const KEYTYPE_RSA = 2;
+const KEYTYPE_LATTICE = 3;
```

Code 36. Adding a mapping for the function handle of the new added menu item in *package/keyRing.jsm*

Afterwards, the logic of the function call needs to be defined in the same file for the new defined constant. Code 37 shows, how we add the key generation logic for a lattice-based key with a signature key as primary key and a encryption key as the subkey.

```
          inputData += "\nSubkey-Type: RSA\nSubkey-Usage: encrypt\nSubkey-Length: ";
          break;
+       case KEYTYPE_LATTICE:
+         inputData += "23\nSubkey-Type: 24\nSubkey-Length: ";
+         break;
        default:
```

Code 37. Defining the logic of the new added menu item in *package/keyRing.jsm*

References

[1] S. Akleylek, N. Bindel, J. Buchmann, J. Krämer, and G. A. Marson, "An efficient lattice-based signature scheme with provably secure instantiation," in *International Conference on Cryptology – AFRICACRYPT 2016* (D. Pointcheval, T. Rachidi, and A. Nitaj, eds.), pp. 44–60, Springer, 2016.

[2] "qhasm: tools to help write high-speed software." https://cr.yp.to/qhasm.html. Accessed: 2016-04-11.

[3] "The Libgcrypt Reference Manual." https://gnupg.org/documentation/manuals/gcrypt/index.html#Top. Accessed: 2016-04-11.

[4] "GNU PG Public key modules." https://www.gnupg.org/documentation/manuals/gcrypt-devel/Public-key-modules.html. Accessed: 2016-04-09.

[5] "The Libgcrypt Reference Manual: Working with S-expressions." https://gnupg.org/documentation/manuals/gcrypt/Working-with-S_002dexpressions.html#Working-with-S_002dexpressions. Accessed: 2016-04-11.

[6] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management-part 1: General, revision 4," in *NIST special publication*, Citeseer, 2016.

[7] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer, "OpenPGP Message Format." RFC 2440 (Proposed Standard), Nov. 1998. Obsoleted by RFC 4880.

[8] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format." RFC 4880 (Proposed Standard), Nov. 2007. Updated by RFC 5581.

Appendix A
Important Call Hierarchies in GPG

In this section we show the most important call hierarchies of GPG, for key generation, encryption and decryption, which we extracted while working with GPG. They can be very helpful for debugging or understanding the workflow of GPG. To give a better understanding how they can be read, we give some small explanation. Line 2 of Code 38, shows the first function call, which is called when key generation command of GPG is run. In this case the function `main` in file `gpg.c` is called as entry point when the command `gpg2 --gen-key` entered. Line 3 of Code 38, shows the next function call and so on. Line 5 is a function call inside the function of line 4, we indicate this with a tabulator space. `->` can be seen as a function call while `<-` indicates a function return. In case the function returns important elements we explain the returned elements in a intuitive way.

```
Key Generation using Keyfiles:
-> main (gpg.c) main entry point of gpg, parses the command input
-> generate_keypair (keygen.c)
-> do_generate_keypair (keygen.c)
  -> do_create (keygen.c) chooses which algorithm is used for key generation
    -> gen_lattice (keygen.c) creates the lattice key
      -> gcry_sexp_build build the sexp which is send to libgcrypt
      -> gcry_pk_genkey send the generate key command with the keyparams to libgcrypt
        -> to libgcrypt
        <- returns the public and secret key in a sexp
      -> genhelp_protect (keygen.c) help function for protecting the secret key
        -> protect_secret_key (seckey-cert.c) protect the secret key
        <-
      <-
    <- returns KBNODE pub_root, KBNODE sec_root, PKT_secret_key** ret_sk, key nodes with public key, protected secret key and an unprotected version of the secret key
  <-
  ->write_selfsigs (keygen.c) sign the created key with the own public key
    -> make_keysig_packet (sign.c)
      -> hash_public_key
      <-
      -> keyid_from_sk (keyid.c)
      <-
      -> complete_sig  (sign.c)
        -> check_secret_key (seckey-cert.c)
          -> do_check (seckey-cert.c)
          <-
        <-
        -> do_sign (sign.c)
          -> encode_md_value (seskey.c)
          <-
          -> pk_sign (pkglue.c)
            -> to libgcrypt
          <-
        <-
      <-
    <-
  <-
  -> write_keyblock (keygen.c) function which should write the key files
    -> build_packet (build-packet.c) write the key packets
      -> do_public_key (build-packet.c)
        -> mpi_write_extented (build-packet.c) / mpi_write
        -> write_header2 (build-packet.c) write the header for the packet
        -> iobuf_write_temp (iobuf.c) write the packet to the real stream, because before that all data is written into a temp buffer
      <-
      -> do_secret_key (build-packet.c)
        -> mpi_write_extended (build-packet.c) / mpi_write write public key
        -> iobuf_put (iobuf.c) write a lot of things, such as protect algo and all important parameters for that
        -> gcry_mpi_get_opaque (to libgcrypt) write the secret key as it is if the secret key is protected (it is protected in our case)
        -> write_header2 (build-packet.c) write the header for the packet
        -> iobuf_write_temp (iobuf.c) write the packet to the real stream, because before that all data is written into a temp buffer
      -> do_signature (build-packet.c)
        -> mpi_write_extended (build-packet.c) / mpi_write write signature
        -> write_header2 (build-packet.c) write the header for the packet
        -> iobuf_write_temp (iobuf.c) write the packet to the real stream, because before that all data is written into a temp buffer
      <-
    <-
  <-
<-
```

Code 38. call hierarchy for key generation with key files

```
Encrypting Files:
-> main (gpg.c)
  -> encode_crypt (encode.c) creates a random sesson symmetric key and encrypts with this sesson key the
    message
    -> make_session_key (seskey.c) creates a session key
    -> write_pubkey_enc_from_list (encode.c) writes the packet with the symmetric session key encrypted by
    the asymmetric key
      -> encode_session_key (seskey.c) Encodes the session key so that it can send to encrypt function in
    libgcrypt, does some sort of padding
      -> pk_encrypt (pkglue.c) Encrypt the symmetric session key with the public key of the asymmetric
    encryption
        -> to libgcrypt
        <- returns the encrypted message
      -> build_packet (build-packet.c) builds a Public-Key Encrypted Session Key Packet as specified in RFC
     4880
    -> iobuf_push_filter is called with a cipher filter and the session key, which creates a Symmetrically
    Encrypted Data Packet
    <-
  <-
<-
```

Code 39. call hierarchy for encryption of a given file

```
Decrypting Files:
-> main (gpg.c)
  -> decrypt_message (decrypt.c) entry point for decryption, open the file, extract the packet
    -> proc_encryption_packets (mainproc.c) process the extracted encryption packet
      -> do_proc_packets (mainproc.c)
        -> proc_pubkey_enc (mainproc.c) main entry point to process a Public-Key Encrypted Session Key
    Packet
          -> get_session_key (pubkey-enc.c) Get the session key from a pubkey enc packet
            -> get_seckey (getkey.c) Get the secret key from the packet
            -> get_it (pubkey-enc.c) Get the session key from the pubkey enc packet, by decrypting the
    packet with the secret key and undoing the padding
          <-
        <-
        -> proc_encrypted (mainproc.c) decrypt the following Symmetrically Encrypted Data Packet with the
    extracted session key
      <-
    <-
  <-
<-
```

Code 40. call hierarchy for decryption of a given file