PyGPG

PyGPG combines post-quantum asymmetric encryption with certain aspects and features of GPG in order to give a practical, useable, but most important, secure way of communicating over unsecure channels. To realize this PyGPG uses the NTRU lattice-based public cryptosystem. I chose to use NTRU because it consumes minimal CPU and battery resources, decryption is more than 92 times faster than RSA decryption at an equivalent security level, and NTRU is nearly 60% faster than RSA at encryption and TLS with a 370 times improvement in key generation time[1].

PyGPG's main functionality is split up in two main parts:

- Encryption and decryption
- TUI/user input

The encryption and decryption part concerns itself with the generation of keys, and encrypting and decryption arbitrary byte arrays. The signing and verifying part concerns itself with key/basis generation for the signatures, and signing/verifying arbitrary byte arrays. The TUI part concerns itself with providing a text user interface (TUI), handling files/messages, and managing the stored keys.

At first the plan was to also implement NTRUSign, a lattice based signing operation, but later I changed my mind about that decision. There are multiple reasons for this, but the most important reason is that the NTRUSign algorithm was broken in 2012 by Ducas and Nguyen[2]. In this paper they show that it takes as little as 8000 signatures to break NTRUSign-251 with one perturbation. Other reasons are that the implementation document for NTRUSign is quite vague, and I could not get certain parts, like basis generation, working properly. Therefore I will mostly be talking about the theory behind NTRUSign, instead of the practical implementation details.

Encryption and Decryption

For the implementation of the Encryption and decryption part I followed the IEEE specification paper that was submitted in 2009[3]. This paper specifies the key generation primitive, encryption primitive and decryption primitive that you need to use for NTRUEncrypt. Next to these primitives the paper also specifies all the parameter sets and necessary extra algorithms, for example for polynomial multiplication in a ring, polynomial inversion in a ring, converting polynomials to bit/octet strings and back, and mask and index generation functions.

Polynomial Functions

In order to represent polynomials I made a class that can be instantiated with a list of coefficients. The list with coefficient looks like [a_0, a_1, …, a_k], from which the following polynomial will be formed: $a_0 + a_1 * x + a_2 * x^2 + … + a_k * x^k$. This polynomial class also contains functions related to polynomials which we will discuss next. It is important to know that all operations discussed happen in the polynomial ring $\mathbb{Z}_p[x]$ where p is a prime number.

Multiplication

The result of multiplication between two ring elements should be another ring element, so the paper specified a special kind of multiplication, also known as star multiplication. This product is given by: $c_k = \sum_{i+j \equiv k \ (mod\ N)} a_i \, b_j$ where N is the degree of the polynomial ring.

Division

Next to multiplication, it is also necessary to be able to divide one polynomial by another polynomial. Because it is not possible to have fractions in a polynomial ring, the output of the division algorithm for input *a* and *b* will be two polynomials *q* and *r* such that $a = b \times q + r$.

Inverse

For the encryption and decryption operations it is necessary to be able to invert polynomials in $\mathbb{Z}_p[x]/(x^N - 1)$. In order to calculate the inverse this paper specifies a version of the Extended Euclidean Algorithm(EEA) that calculates the greatest common divisor *d* of two polynomials. For input polynomials *a* and *b* the algorithm also calculates polynomials *u* and *v* such that $a \times u + b \times v = d$. In order to find the inverse of a polynomial the EEA should be run with as inputs the polynomial you want to invert and x^N*-1.* If GCD *d* has a degree of 0, i.e. the greatest common divisor is a constant, then the inverse of the polynomial is, assuming the polynomial is *a*, the multiplicative inverse of *d* multiplied by *u*. If GCD d has a degree larger than zero it means the polynomial cannot be inverted in $\mathbb{Z}_p[x]/(x^N - 1)$.

In a later paper[4] Joseph H. Silverman, one of the researchers of NTRU, proposed other efficient algorithms for inverse calculation. PyGPG uses this way of inverse calculation as it performs a lot better than the EEA way of calculating an inverse.

Key generation

For the key generation two polynomials need to be generated; a private key polynomial *f* and a public key polynomial *h*. *f* is generated by randomly choosing a polynomial *F* with *dF* 1 and -1 coefficients, and calculating $f = 1 + p \times F$, where p is a parameter from the chosen parameter set. If *f* is not invertible in $\mathbb{Z}_q[x]/(x^N - 1)$ then another *F* needs to be chosen. *f* is chosen this way because *f,* next to being invertible modulo q, also needs to be invertible in in $\mathbb{Z}_p[x]/(x^N - 1)$, and this way *f* is per definition invertible modulo p. *A* polynomial *g* is chosen in the same way *F* is chosen, and if *g* is not invertible in $\mathbb{Z}_q[x]/(x^N - 1)$ then another *g* needs to be chosen. The public key *h* is calculated as follows: $h = f^{-1} \times g \times p$. As random number generator you can use the index generating function specified in the paper, or any other cryptographically secure random number generator.

Encryption operation

For the encryption operation a byte array representing the message and the public key *h* need to be provided. The plaintext that will be encrypted consists of the following parts:

- A random byte string of length bLen (specified in the parameters)
- A byte indicating the length of the message in bytes
- The message itself
- A byte string consisting of 0 bytes, with as length the maximum message length + 1 – the length of the message.

Each three bits of the plain text will be converted to two ternary coefficients of a polynomial which will be the message polynomial *MTrin*. The next step is to convert the public key *h* to a bit string, and create a seed for the blinding polynomial generator, consisting of:

- the OID (identifier of the parameter set)
- the message bytes
- the random byte string of length bLen
- the bit string of the public key *h*

This seed will be used to generate blinding polynomial *r,* and this *r* multiplied by public key *h* will give us polynomial *R*. *R,* in bit string form, is used as seed to create a mask polynomial *mask.* The final cyphertext *e* is computed as follows: $e = r \times h + MTrin \times mask = R + MTrin \times mask$.

Index generating function

For several steps of the encryption operation securely random generated numbers are needed. The paper proposes a index generating function that securely generates random numbers from a seed. In this algorithm the random bits are generated by taking the hash of the seed concatenated with a counter, using an approved hash function like SHA-256. The counter is incremented after it is used. These random bits are stored in a byte array *buf*. If there are more random bits needed another hash of the seed concatenated with a counter will be added to *buf*.

Mask generation function

The mask generation functions are similar to hash functions, but they differ in the fact that they can produce an output of arbitrary length, instead of the fixed length of a hash function. Mask generation functions can be seen as single-use sponge functions. The random bits in the mask generation function are generated in the same way as they are generated in the index generating function, but instead of returning a random number it will use these random bits to construct a polynomial with N ternary coefficients, where N is specified in the encryption parameters.

Decryption operation

For the decryption operation a byte array representing the cipher text *e* and the private key *f* are needed. The first step is to recover the candidate decrypted polynomial c_i using the private key as follows:

$$
\begin{aligned}
a &= f \times e \ (mod\ q) \\
a &= f \times (r \times h + MTrin \times mask) \ (mod\ q) \\
a &= f \times \left(r \times p \times f_q^{-1} \times g + MTrin \times mask\right) \ (mod\ q) \\
a &= pr \times g + f \times MTrin \times mask \ (mod\ q) \\
c_i &= a \ (mod\ p) \\
c_i &= f \times MTrin \times mask \ (mod\ p) \\
c_i &= MTrin \times mask \ (mod\ p)
\end{aligned}
$$

In the last equation the private key *f* can be removed from the equation because we constructed *f* as $1 + F \times p$ during key generation. The next step is to reconstruct $r \times h$ from *e* using c_i . Using this $r \times h$ the seed for the mask generation function can be recovered, and the masking polynomial can be generated in the same way it was generated in the encryption step. Using this mask polynomial the candidate *MTrin* polynomial can be recovered (by subtracting the mask from c_i). This candidate *MTrin* is a ternary polynomial will be converted to a bit string in the opposite way it was constructed as ternary polynomial in the encryption step. With this candidate message byte string the seed for the blinding polynomial can be recovered, and with this seed a candidate blinding polynomial *cr* will be generated. This candidate *cr* will be multiplied with public key *h* and the result will be compared with $r \times h$ that was reconstructed from *e*. If *cr* and the recovered $r \times h$ are equal the candidate message will be returned as decrypted message.

TUI/user input

With only the “raw” encryption and decryption operation we do not get a very usable application. That is why I build a Textual User Interface (TUI) which handles the commands the user issues. The TUI has three main functionalities:

- Generation of keys
- Encryption of files
- Decryption of files

Generation of keys

One of the most important parts of things we need for the encryption and decryption of files is a keypair, consisting of a private and a public part. Because a key/the hash of a key is quite hard to remember I chose to identify keys by email addresses. At this moment the program is not checking if an identifier really is an email address, but this could be added at a later stage because this is not a priority item. During key generation you will be asked to enter two things: security level (how many bits of security you would like) and a password to protect your private key. There are four different security levels, they are listed below together with the parameter set that guarantees that security level.

- 112 bits – ees659ep1
- 128 bits – ees791ep1
- 192 bits – ees1087ep1
- 256 bits – ees1499ep1

The command for generating a new key pair is:

```
pygpg(.exe) -g [key id]
```

Encryption/Decryption of files

At this moment the program only supports the encryption and decryption of files. The respective commands are:

```
pygpg(.exe) -e [key id] [file path]
pygpg(.exe) -d [file path]
```

The key id specifies which public key to use for the encryption of the file, and the file path point to the file to encrypt. The encrypted file will have the form filename.extension.pygpg, where filename and extension are the same filename and extension of the input file. The decryption operation will extract the used key id from the .pygpg file, and check if it has the private key for that key id. If it does it will prompt the user for the password and decrypt the file.

The encryption and decryption work in a similar way to PGP. That means that the file will be symmetrically encrypted, and the symmetric key will be asymmetrically encrypted using the NTRU cryptosystem. For the symmetric encryption I used the python package ‘cryptography’, which symmetrically encrypts files using AES128 in cbc mode.

1. onBoardSecurity. (n.d.). NTRU Post Quantum Cryptography. Retrieved from https://www.onboardsecurity.com/products/ntru-crypto

2. Ducas L., Nguyen P.Q. (2012) Learning a Zonotope and More: Cryptanalysis of NTRUSign Countermeasures. In: Wang X., Sako K. (eds) Advances in Cryptology – ASIACRYPT 2012. ASIACRYPT 2012. Lecture Notes in Computer Science, vol 7658. Springer, Berlin, Heidelberg
3. IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices," in IEEE Std 1363.1-2008 , vol., no., pp.C1-69, March 10 2009
4. Joseph H. Silverman, Almost Inverses and Fast NTRU Key Creation, NTRU Cryptosystems Technical Report 14, available at http://www.ntru.com