Post-Quantum Cryptography with Python and Linux

A beginner's guide

Photo by Jean-Louis Paulin on Unsplash

If we believe Edward Snowden, encryption is "the only true protection against surveillance" [1]. However, advances in quantum technology might endanger this safeguard. Our article discusses why quantum computing poses a threat to data security and what to do about it. Instead of a purely theoretical analysis, we build on code examples using Python, C, and Linux.

Quantum basics

When Google scientists reported the first case of quantum supremacy in 2019, they caused great excitement. One area where quantum computing could have significant impact is encryption. To understand this issue, we need to discuss some basics.

In contrast to classical computers, algorithms for quantum computers do not rely on bits, but on *qubits*. A bit can either take the state 0 or 1. When we measure a bit several times, we invariably get the same result. With qubits, things are different. As strange as it sounds, a qubit can take the value 0 and 1 at the same time. When we measure it repeatedly, there is only a certain probability for getting the result 0 or 1. In the initial state of a qubit, the probability for measuring 0 is commonly one hundred percent. Through superposition, however, different probability distributions can be generated. Causes lie in quantum mechanics, following other laws than "normal" life.

The main advantage of quantum computers is their probabilistic nature. Classical computers excel at problems where we reliably need a single result. Quantum machines, on the other hand, are superb at dealing with probabilities and combinatorics. When we perform an operation on a qubit in a superposed state, it is simultaneously applied to both values 0 and 1. As the number of qubits increases, so does the advantage over a classical computer. A quantum machine with three qubits can process up to eight values (2^3) simultaneously: namely, the binary numbers 000, 001, 010, 011, 100, 101, 110, and 111.

Scientific literature agrees that quantum computers will help solving problems that previously seemed intractable. Yet there are no optimal quantum machines available. The current generation of quantum computers is referred to as noisy intermediate-scale quantum (NISQ). Such machines have limited processing power and are sensitive to errors. Modern devices offer up to several hundred qubits. One example is the 433-qubit Osprey chip that IBM introduced in 2022. Now, the company plans to develop a machine with 100,000 qubits by 2033.

Our article explains why this evolution poses a threat to data security. Using code examples, we show why quantum computers could break certain encryption and discuss workarounds. The source code is available on GitHub. It was developed under Kali Linux 2023.2 using Anaconda with Python 3.10.

Encryption and prime factors

When encrypting a message, a relatively simple way is to apply a symmetric algorithm. Such a method uses the same key for both the encryption of the plaintext and the decryption of the ciphertext. A major challenge with this approach is the secure exchange of the key between sender and recipient. Once the private key becomes known to a third party, they have the chance to intercept and decrypt the message.

Asymmetric cryptography seemed to be the solution to this problem. Methods like RSA use different keys for encryption and decryption. Encryption is performed here with one or more public keys that the recipient makes accessible to everyone. For decryption, the recipient uses a private key, which is known only to them. This way, the sender can obtain the public key without risk, since it is not secret anyway. Only the recipient's private key must be protected. But how can such a procedure be hardened, when potential attackers know the public key? To this end, asymmetric algorithms rely on mathematical problems like prime factorization.

Prime factorization is best understood by example. In Python, we can use the function `factorint` of the library SymPy to determine the prime factors of a certain integer.

```
>>> import sympy
>>> sympy.factorint(10)
{2: 1, 5: 1}
>>> 2**1 * 5**1
10
>>> sympy.factorint(1000)
{2: 3, 5: 3}
>>> 2**3 * 5**3
1000
>>> sympy.factorint(55557)
{3: 2, 6173: 1}
>>> 3**2 * 6173**1
55557
>>>
```

The above console output illustrates that every natural number can be expressed as a product of prime numbers. These are called prime factors. Thinking back to school days, a prime number is divisible only by 1 and itself. For example, the number 10 can be expressed by the term $10=2^1 * 5^1$. Thus, the prime factors of 10 are 2 and 5. Analogously, the number 55557 can be expressed by the equation $55557=3^2 * 6173^1$. So, the prime factors of 55557 equal 3 and 6173. The process of finding prime factors for a given integer is called prime factorization.

With classical computers, prime factorization is simple for small numbers, but becomes increasingly hard for big integers. Each additional number drastically increases the sum of possible combinations. Beyond a certain point, it becomes virtually impossible for a classical computer to determine prime factors. For example, consider the following number (RSA-260) from the RSA Factoring Challenge, ended in 2007. At the time of writing, it has not yet been factorized.

```
#!/usr/bin/env python
import sympy

rsa_260 =
2211282552952966643528108525502623092761208950247001539441374831912882294140200198651272972656974659908590033003140005117074220456085927635795375718595429883895870922923849100670303412462054578456641366454068421436129301769402084639106587591479425143514445819 9

print("Start factoring...")
factors = sympy.factorint(rsa_260)

# Will probably not be reached
print(factors)
```

Asymmetric algorithms like RSA utilize the computational hardness of prime factorization and similar problems to secure encryption. Unfortunately, the quantum world follows its own laws.

Quantum algorithms

Regarding cryptography, two quantum algorithms are of particular concern. Shor's algorithm provides an efficient way of prime factorization. When performed on a large quantum device, it can theoretically break asymmetric methods like RSA. From a practical perspective, this scenario lies in the future. A Nature article from 2023 mentions the number of at least 1,000,000 qubits required. Hardware aside, it is also difficult to find implementations of the algorithm that reliably scale on large quantum machines. IBM's framework Qiskit had tried to implement a function, but deprecated it with version 0.22.0. However, experimental implementations of Shor's algorithm can be found online.

Grover's algorithm poses a threat to symmetric encryption. Also known as quantum search algorithm, it offers a speed-up for unstructured search of the input for a given function. Quantum computers can use it to accelerate brute-force attacks on symmetrically encrypted information. Yet, unlike Shor's algorithm, the offered speedup is not exponential. In simple terms, this means that increasing the length of the key used for encryption makes the search excessively more expensive. For example, performing a brute-force attack on a 128-bit key requires a maximum of 2^{128} iterations. Assuming that Grover's search reduces this number to 2^{64}, doubling the key length to 256 bits increases it again to 2^{128} iterations. This opens the door to possible workarounds.

Symmetric workaround

Under certain conditions, symmetric encryption is a ready-to-use and simple way to counter quantum algorithms. Reason is that Grover's search does not scale exponentially and Shor's algorithm only threatens asymmetric approaches. To best of current knowledge, symmetric algorithms with a high degree of hardness can be considered quantum-resistant. At present, both the American NIST as well as the German BSI include AES-256 in this category [2][3]. The acronym AES stands for Advanced Encryption Standard and the number 256 represents the bit length of the key. Under Linux, AES-256 is implemented by the GNU Privacy Guard (GnuPG). The following shell script shows how a file can be encrypted and then decrypted again using AES-256.

```
# Encrypt
gpg --output encrypted.gpg --symmetric --cipher-algo AES256 plain.txt

# Decrypt
gpg --output decrypted.txt --decrypt encrypted.gpg
```

The above script encrypts the content of the file "plain.txt", writes the ciphertext to the document "encrypted.gpg", decrypts it again and finally saves the output to the file "decrypted.txt". Before encryption, GnuPG asks for a passphrase to generate the private key. For security reasons, it is vital to choose a strong passphrase and keep it secret. GnuPG might cache the passphrase and not ask again when decrypting. To clear the cache, the following shell command can be executed.

```
gpg-connect-agent reloadagent /bye
```

Integrating GnuPG into Python is relatively simple with the `subprocess` module. A prototype implementation of the encryption with AES-256 is shown in the code fragment below.

```
#!/usr/bin/env python
import subprocess
import getpass

# Read passphrase
passphrase = getpass.getpass("Passphrase:")
passphrase2 = getpass.getpass("Passphrase:")

if passphrase != passphrase2:
  raise ValueError("Passphrases not identical!")

# Perform encryption
print("Encrypting...")

args = [
  "gpg",
  "--batch",
  "--passphrase-fd", "0",
  "--output", "encrypted.gpg",
  "--symmetric",
  "--yes",
  "--cipher-algo", "AES256",
  "plain.txt",
]

result = subprocess.run(
  args, input=passphrase.encode(),
```

```
  capture_output=True)

if result.returncode != 0:
  raise ValueError(result.stderr)
```

For getting a passphrase, the above script uses the `getpass` module. After confirmation, the passphrase is transferred to GnuPG via standard input. This is indicated by the argument `passphrase-fd 0`. Alternatively, passphrases can be sent to GnuPG as a string or by file with command line arguments. However, as these arguments are visible to other users, both options were rejected for the prototype. Another, more secure way would be to use the GPG-Agent. Which option to take depends on the desired security level. A proof-of-concept including encryption and decryption is provided here. As an alternative to GnuPG, there are other AES-256 implementations. Choosing a trustworthy source is vital here.

Asymmetric workaround

In search of an asymmetric solution, the NIST Post-Quantum Cryptography Standardization program is a good starting point. Since 2016, it has evaluated multiple candidates for quantum-resistant algorithms. One of the winners is Kyber. The system implements a so-called secure key encapsulation mechanism. Similar to other algorithms, Kyber relies on a hard-to-solve problem to protect key exchange between two parties. Instead of prime factorization, it is based on a problem called “learning with errors.” What level of protection Kyber offers, depends on the key length. For example, Kyber-1024 aims at a security level “roughly equivalent to AES-256” [4].

A reference implementation of Kyber, written in C, is available on GitHub. Under Linux, we can clone and build the framework by executing the shell commands below. Some prerequisites are required for installation, which are documented in the project’s README.

```
git clone https://github.com/pq-crystals/kyber.git
cd kyber/ref && make
```

There are several ways to integrate the reference implementation into Python. One of them is to write a C program and call it. The C function below uses Kyber to perform a key exchange between two fictional parties, Alice and Bob. For the full source code, see here.

```
#include <stddef.h>
#include <stdio.h>
#include <string.h>
#include "kem.h"
#include "randombytes.h"

void round_trip(void) {
    uint8_t pk[CRYPTO_PUBLICKEYBYTES];
    uint8_t sk[CRYPTO_SECRETKEYBYTES];
    uint8_t ct[CRYPTO_CIPHERTEXTBYTES];
    uint8_t key_a[CRYPTO_BYTES];
    uint8_t key_b[CRYPTO_BYTES];

    //Alice generates a public key
    crypto_kem_keypair(pk, sk);
    print_key("Alice' public key", pk);

    //Bob derives a secret key and creates a response
    crypto_kem_enc(ct, key_b, pk);
    print_key("Bob's shared key", key_b);
```

```
    print_key("Bob's response key", ct);

    //Alice uses Bobs response to get her shared key
    crypto_kem_dec(key_a, ct, sk);
    print_key("Alice' shared key", key_a);
}
```

Without going into details, one can see that Kyber uses multiple public and private keys. In the above example, Alice generates a public key (pk) and a private key (sk). Next, Bob uses the public key (pk) to derive a shared key (key_b) and a response key (ct). Latter is returned to Alice. Finally, Alice uses the response key (ct) and her private key (sk) to generate an instance of the shared key (key_a). As long as both parties keep their private and shared keys secret, the algorithm offers protection. When running the program, we obtain an output similar to the text below.

```
Alice' public key: F0476B9B5867DD226588..
Bob's shared key: ADC41F30B665B1487A51..
Bob's response key: 9329C7951AF80028F42E..
Alice' shared key: ADC41F30B665B1487A51..
```

In order to call the C function in Python, we can use the `subprocess` module. Alternatively, it is possible to build a shared library and invoke it with the `ctypes` module. Second option is implemented in the Python script below. After loading the shared library, generated from the Kyber C code, the procedure `round_trip` is called like any other Python function.

```
#!/usr/bin/env python
import os
import ctypes

# Load shared library
libname = f"{os.getcwd()}/execute_round_trip1024.so"
clib = ctypes.CDLL(libname, mode=1)
print("Shared lib loaded successfully:")
print(clib)

# Call round trip function
print("Executing round trip:")
clib.round_trip()
```

In addition to Kyber's reference implementation, other providers have implemented the algorithm. Examples are the open-source projects Botan and Open Quantum Safe.

Conclusion

As our analysis reveals, quantum technology is still in its early stages. But we should not underestimate the threat it poses to encryption and other cryptographic methods such as signatures. Disruptive innovation can boost development at any time. Attackers can store messages now and decrypt later. Therefore, security measures should be taken immediately. Especially since there are workarounds available. When used properly, symmetric algorithms like AES-256 are considered quantum-resistant. In addition, asymmetric solutions like Kyber are progressing. Which alternatives to use depends on the application. Following a Zero Trust model, a combination of approaches gives the best protection. This way, the quantum threat could end up like the Y2K problem — as a self-defeating prophecy.

About the authors

Christian Koch is an Enterprise Architect at BWI GmbH and Lecturer at the Nuremberg Institute of Technology Georg Simon Ohm.

Lucie Kogelheide is the Technology Lead Post-Quantum Cryptography at BWI GmbH and responsible for initiating the company's migration process to quantum safe cryptography.

Raphael Lorenz is the Founder and CISO of Lorenz Systems and specializes in holistic security solutions.

References

1. Snowden, Edward: *Permanent Record.* Macmillan, 2019.
2. National Institute of Standards and Technology: *NIST Post-Quantum Cryptography: FAQS*. 29 June 2023. Accessed: 02 August 2023.
3. Federal Office for Information Security (BSI): *Quantum-safe cryptography — fundamentals, current developments and recommendations (PDF).* October 2021. Accessed: 02 August 2023.
4. CRYSTALS — Cryptographic Suite for Algebraic Lattices: *Kyber Home*. December 2020. Accessed: 02 August 2023.

Disclaimer

Please note that information security is a critical topic and that there is no warranty by the authors for the published content.