Cryptographic Methods in Data Analysis 2024, Group 2

Implementation of the SABER Key Encapsulation Algorithm

Przemysław Spyra
przspyra@student.agh.edu.pl

Wojciech Ciężobka
wciezobka@student.agh.edu.pl

June 2024

1 Introduction: Key Encapsulation Mechanisms (KEMs) and SABER

Public-key cryptography (PKC) plays a vital role in securing communication in the digital age. A central concept within PKC is public-key encryption (PKE), which allows anyone to encrypt messages using a public key. However, only the authorized recipient with the corresponding private key can decrypt them. Although PKE offers robust security, it might not be the most efficient choice to encrypt large messages.

This report investigates an alternative approach: Key Encapsulation Mechanisms (KEMs). KEMs address the limitations of PKE for bulk encryption. Unlike PKE, which directly encrypts messages using public keys, a KEM focuses on securely transmitting a symmetric key. This symmetric key is then used for efficient encryption of the actual message content.

This distinction is crucial. KEMs generate a random, secret symmetric key and leverage the recipient's public key to create an encapsulation containing this key. Only the recipient's private key can decrypt the encapsulation and recover the symmetric key. This approach offers several advantages:

- **Efficiency:** Symmetric encryption algorithms are significantly faster than PKE for large messages.
- **Security:** KEMs benefit from the security guarantees of PKE for key exchange while relying on efficient symmetric algorithms for bulk encryption.
- **Simplification:** KEM simplifies the key exchange process and offers provable security for the key derivation process.

To enhance the efficiency and security of the hybrid encryption process, KEMs can be integrated into the PKE with symmetric encryption pipeline. This integration leverages the strengths of both PKE and KEM:

- **PKE:** Ensures the secure delivery of the encapsulated symmetric key exclusively to the authorized recipient.
- **Symmetric Encryption:** Provides efficient encryption for bulk data, optimizing performance.

In general, KEM acts as a bridge between public-key infrastructure and symmetric-key encryption, enabling secure and efficient communication.

This report delves into the details of the SABER algorithm [1], a specific KEM based on the Module Learning With Rounding problem (MLWR). We will explore the inner workings of SABER, its advantages, and its role in secure communication protocols.

2 Implementation and Use Case

The implementation of the SABER algorithm includes both the Public Key Encryption (PKE) and the Key Encapsulation Mechanism (KEM). To access the code written in Python, please visit our GitHub repository: `https://github.com/Przemyslaw11/Cryptography_project_2024`. We first present the use case scenario, and then, in the appendix A, we provide the implementation of this scenario with our code.

2.1 Use Case Scenario

Alice and Bob are two IoT devices that want to exchange sensitive data securely leveraging the KEM provided by the SABER algorithm.

- **Key Exchange:**
    1. Bob generates his SABER key pair (public key and secret key) by calling `Saber.KEM.KeyGen`:
        - $(pk_B, sk_B) = \texttt{Saber.KEM.KeyGen}()$
    2. Bob sends his public key pk_B to Alice
    3. Alice encapsulates the public key from Bob to obtain a session key and a ciphertext:
        - $(k_{ses}, c) = \texttt{Saber.KEM.Encaps}(pk_B)$
    4. Alice sends the ciphertext c to Bob
    5. Bob decapsulates the recived ciphertext c with his secret key sk_B to get the session key k_{ses}
        - $k_{ses} = \texttt{Saber.KEM.Decaps}(c, sk_B)$
- **Secure Communication:**
    - Now both Alice and Bob share the same session key, which can be used for symmetric encryption (e.g., AES) to communicate securely.
        1. Alice encrypts her data D with the AES algorithm:
            * $c_{AES} = \texttt{AES.Encrypt}(D)$
        2. Alice sends the encrypted data c_{AES} to Bob.
        3. Bob decrypts the recived data with the AES algorithm:
            * $D = \texttt{AES.Decrypt}(c_{AES})$
        4. The schema repeats for symmetric transsmision.

3 Comments and Conclusions

Implementing the SABER algorithm in Python was challenging. Although we started with pseudocode, we encountered several common hurdles along the way.

The first major challange was understanding the notation of the provided SABER specification file[1]. Although the Section 2 of this specification introduces the notation quite toroughly, there were still minor missunderstandings. One of these is the interpretation of a central binomial distribution β_μ used in the PKE scheme and the specific value of the μ parameter. The document describes the parameter: *a higher value for μ will result in a higher security, but a lower correctness of the scheme*; and provides only the upper bound as $\mu < p$. We decided to set the value of μ to 4, to ensure the correctness.

Another misunderstanding that resulted in decryption errors is the meaning of the subscript in the variables denoting polynomials. It turned out that the subscript denotes the modulus of the polynomial. To give an example, let us invoke line 8 of the `Saber.PKE.KeyGen`[2] algorithm:

$$\mathbf{b}_p[i] = \mathrm{SHIFTRIGHT}(\mathbf{b}[i], \mathrm{EQ} - \mathrm{EP})$$

Here it is important to note that the polynomials $\mathbf{b}[i]$ change their moduli from default q to p in $\mathbf{b}_p[i]$, which is denoted by the subscript $_p$. There are a lot more such examples along the code, but once spotted it was easy to correct.

Despite these obstacles, working on the SABER algorithm in Python was a rewarding experience. Once all the supporting functions and the PKE functions were working well, the remaining implementation of the KEM functions was straightforward. Overall, exploring post-quantum cryptography was enjoyable and developing our own secure data transmission and encryption methods felt like a significant achievement.

Bibliography

[1] J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren, "Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem," in *Progress in Cryptology–AFRICACRYPT 2018: 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, Proceedings 10.* Springer, 2018, pp. 282–305.

A Use case implementation

Here we provide a printed ipynb notebook page containing the use case (Section 2.1) implementation with our code[3].

[1]`https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf`

[2]`https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf#page=32.20`

[3]`https://github.com/Przemyslaw11/SABER_Python_Implementation/blob/main/saber/usecases.ipynb`

Use case 3: Key encapsulation mechanism (KEM) example

This use case is similar to the previous one, but thanks to KEM, Alice gets the symetric key along the ciphertext in one step, from Bob's public key. She does not need to generate the symmetric key by herself. Bob can decrypt the recived ciphertext to get the same symmetric key as Alice has and they can communicate securely and efficiently with symmetric encryption, e.g. AES.

In [ ]:
```
from kem import KEM
from Crypto.Cipher import AES
from utils.algorithms import randombytes
from utils.constants import CONSTANTS_LIGHT_SABER
```

Definition of the cryptosystem

In [ ]:
```
kem = KEM(**CONSTANTS_LIGHT_SABER)
```

Bob generates a public/secret key pair

He sends the public key to Alice, and keeps the secret key to himself.

In [ ]:
```
pk, sk = kem.KeyGen()
```

Alice encapsulates the Bob's public key

She encapsulates the Bob's public key to get the session key along the ciphertext. She keeps the session key and sends the ciphertext to Bob.

In [ ]:
```
session_key_alice, ciphertext = kem.Encaps(pk)
```

Eve intercepts the public key

Eve intercepts the public key and tries to get the session key. But her attempt is unsuccessfull because each encapsulation is random.

In [ ]:
```
session_key_eve = kem.Encaps(pk)

assert session_key_alice != session_key_eve, "Eavesdropping Eve has compromised the session key!"
print("Session key is secure!")
```

```
Session key is secure!
```

Bob decapsulates the ciphertext

He decapsulates the ciphertext with his secret key to get the session key. Now both Alice and Bob have the same symmetric key.

In [ ]:
```
session_key_bob = kem.Decaps(ciphertext, sk)
```

Symmetric key encryption for efficient communication

Now both Alice and Bob have the same session key (symmetric), and they can use it to communicate efficiently using the symmetric key encryption scheme, for example, AES.

In [ ]:
```
assert session_key_alice == session_key_bob, "Wrong session key!"

iv = randombytes(16)
```

1. Alice sends data to Bob

In [ ]:
```
aes_A = AES.new(session_key_alice, AES.MODE_CBC, iv)

data = 'Hi Bob! Our communication is now resilient for quantum atacks!'.encode('utf-8')
data = data + b"\x00" * (16 - len(data) % 16)  # Padding (if needed)
aes_encrypted= aes_A.encrypt(data)
```

2. Bob decrypts the data

In [ ]:
```
aes_B = AES.new(session_key_bob, AES.MODE_CBC, iv)

decrypted_data = aes_B.decrypt(aes_encrypted)
```

3. Test

In [ ]:
```
print(f"Original data:  {data}")
print(f"Decrypted data: {decrypted_data}")

assert data == decrypted_data, "Decryption failed!"
print("Decryption successful!")
```

```
Original data:  b'Hi Bob! Our communication is now resilient for quantum atacks!\x00\x00'
Decrypted data: b'Hi Bob! Our communication is now resilient for quantum atacks!\x00\x00'
Decryption successful!
```