An almost key-homomorphic post-quantum block cipher with key rotation and security update for long-term secret storage

Thomas Prévost[1][0009−0000−2224−8574], Bruno Martin[1][0000−0002−0048−5197], and Olivier Alibart[2][0000−0003−4404−4067]

[1] Université Côte d'Azur, CNRS, I3S, France
{thomas.prevost,bruno.martin}@univ-cotedazur.fr
[2] Université Côte d'Azur, CNRS, InPhyNi, France
olivier.alibart@univ-cotedazur.fr

Abstract. We propose a new block cipher primitive, based on ring-LWE, which allows key rotation with a possible security update. This makes it possible to double the key length and thus the security of the ciphertext with each key rotation. Our scheme could therefore be used for long-term secret storage, allowing the security of the ciphertext to be adapted to the attacker's computing power, without the need for decryption prior its reencryption.
We propose an implementation of our cryptographic scheme and prove its security.

Keywords: Block cipher · Long-term secret storage · Key rotation · Security update · Ring-LWE.

1 Introduction

Long-term secret storage requires a cryptographic primitive that is resistant to increases in the attacker's computing power. The increase in adversaries computing power and the emergence of quantum computing pose a long-term threat to currently used symmetric cryptography schemes.

On the quantum side, Grover's search algorithm [16] allows a quadratic reduction in the number of operations required to break a ciphertext. Thus, while AES-256 encryption offers 256 bits of security in the classical world, it only offers 128 bits of security against an attacker equipped with a sufficiently powerful quantum computer. Even assuming that the adversary will only ever have classical computers, their power increases year after year, rendering older ciphertexts obsolete. While 80 bits of security were considered sufficient in the 2000s [4], today 128 bits are considered the bare minimum, and 256 bits of security are considered the norm. Indeed, with the Bitcoin network exceeding 2^{94} hashes/year [3], it is possible, for an attacker with equivalent computing power, to break a ciphertext with 80 bits of security in a reasonable amount of time.

[3] https://www.blockchain.com/explorer/charts/hash-rate

Another way to look at this is to estimate the monetary cost required to break a ciphertext using an exhaustive attack [8]. Very simple calculations show that an attack against a ciphertext with 80 bits of security would cost less than a million dollars today [4], making such a ciphertext vulnerable to even a moderately powerful attacker.

Periodic updating of encryption keys is an essential mechanism for protecting secrets, especially on the Cloud. Indeed, an organization that uses the same cryptographic keys over the long term could have them revealed by a former employee, for example. Periodic key updating has therefore become an industry standard, and even a requirement for compliance with the PCI-DSS (Payment Card Industry Data Security Standard) [24]. Most organizations, however, adopt a "hybrid" encryption method to update their encryption key: the secret m is encrypted by a key k_1 that remains constant over time, and this same key is encrypted by a key k_2. The final ciphertext is therefore the pair $(\text{Enc}(m, k_1), \text{Enc}(k_1, k_2))$, and the final encryption key is k_2. The key is updated by decrypting and then re-encrypting k_1. However, such an approach has many flaws [14], and will not guarantee the confidentiality of secret m when the security of key k_1 becomes too weak compared to attacker's computing power.

A better approach for performing a key update is to use a key rotation scheme. These schemes, often based on a homomorphic encryption primitive, allow the ciphertext to be updated, i.e., to convert $\text{Enc}(m, k)$ to $\text{Enc}(m, k')$ without having to decrypt, with k the old key and k' the new key. [6] proposes an interesting approach for this, based on the Ring-Learning-With-Errors (RLWE) primitive [18], which is supposedly resistant to quantum computer attacks. Our scheme is based on this primitive.

In this paper, we propose a new block cipher scheme that allows key rotation, as well as security level updating without decryption. Our scheme would therefore make it possible to adapt a ciphertext to the evolution of an attacker's computing power, without needing to decrypt it. With each key update, it is thus possible to double the security of the key. Our scheme therefore allows to go from 128 bits to 256 bits of security without decryption. This security level can then be doubled up to 512 bits, etc. This property is particularly interesting if the ciphertext is located on a remote server or a Cloud environment that is not trusted. It is thus possible to update the ciphertext without the remote server being able to discover the plaintext. As in [6], the number of key rotations is however bounded, but it is possible to perform a significant number of rotations before the ciphertext is no longer decryptable. Since the LWE assumption, on which our cipher is based, is relatively recent in the history of mathematical study, we still recommend using a double cipher for use in critical contexts: AES-256 together with our cipher. Indeed, although LWE is considered secure today, it is not impossible that the assumption will be proven false in the more or less distant future.

[4] Someone on crypto stackexchange performed the calculation: `https://docs.google.com/spreadsheets/d/1hsvO2RBXWYxMMMCaDx5CASPy2lh22q5ApkUg1EE577g/`

This paper is organized as follow: first, we will recall the RLWE assumption in section 2. Next, in section 3, we will briefly explain how the block cipher with key rotation described in [6] works. We will then develop in section 4 the operation of our encryption scheme, with key rotation and security level update. We will give a detailed analysis of the security of our scheme in section 5. We will propose a proof of concept in section 6, and we will analyze its performance in section 7. Finally, we will conclude with ideas for future work.

Related work Several approaches have been proposed for key rotation or proxy reencryption [20,11,17,3]. [21] also proposes an interesting approach, based on block permutations. Compared to proxy reencryption, key rotation adds an indistinguishability constraint between the old and new ciphertexts.

To our knowledge, however, there is no block cipher scheme that allows key rotation with increased security level. The only similar approach was proposed by [2], which introduced the concept of "key-rotatable and security-updatable homomorphic encryption" (KR-SU-HE) for asymmetric encryption.

2 Ring-LWE

Ring-Learning-With-Errors (RLWE) [18] is a computational problem that has been proven equivalent to the Learning-With-Errors (LWE) problem, selected by NIST as a standard for post-quantum cryptography [1].

The RLWE assumption is formulated as follows. Let q be a prime number, and $\mathbb{F}_q[x]$ represents the ring of univariate polynomials whose integer coefficients lie in $[\![-\frac{q-1}{2}, \frac{q-1}{2}]\!]$. We then consider the quotient ring $\mathbb{F}_q[x]/\Phi(x)$, with $\Phi(x)$ being an irreducible polynomial in $\mathbb{Z}$ but not necessarily in $\mathbb{Z}_q$. For example, the cyclotomic polynomial $\Phi(x) = x^n + 1$ is a good choice of polynomial. The degree n of the polynomial $\Phi(x)$ is the security level of our cryptographic primitive.

Definition 1. *Let $\mathcal{D}_e$ be a uniform distribution on $[\![-b, b]\!]$, with $b \ll \frac{q}{2}$. Let $a(x) \in \mathbb{F}_q[x]/\Phi(x)$ be a* known *random polynomial, $b(x) \in \mathbb{F}_q[x]/\Phi(x)$ be an* unknown *random polynomial, and $e(x) \in \mathbb{F}_q[x]/\Phi(x)$ be an* unknown *random polynomial such that the coefficients of $e(x)$ are given by the distribution $\mathcal{D}_e$. Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary. Then, if the adversary $\mathcal{A}$ is given the pair $(a(x), c(x) = a(x) \cdot b(x) + e(x))$, the RLWE assumption states that:*

- *the adversary $\mathcal{A}$ will not be able to distinguish the polynomial $c(x)$ from a random polynomial in $\mathbb{F}_q[x]/\Phi(x)$ (decision problem)*
- *the adversary $\mathcal{A}$ will not be able to extract the polynomials $b(x)$ and $e(x)$ (search problem).*

The RLWE assumption is supposed to hold even against an adversary $\mathcal{A}$ equipped with a quantum computer, provided that the prime q and the error range $[\![-b, b]\!]$ are well chosen [10,13].

Remark 1. It is possible to replace the $\mathcal{D}_e$ distribution with a Gaussian distribution on $[\![-\frac{q-1}{2}, \frac{q-1}{2}]\!]$, centered at 0.

3 Block cipher with key rotation

In this section, we briefly present the scheme described in [6], which explains how to construct an RLWE-based block cipher scheme that allows key rotation, using Almost Key-Homomorphic pseudorandom functions (PRF).

We define the scheme $\Pi = (\text{KeyGen}, \text{Encrypt}, \text{Decrypt}, \text{ReKeyGen}, \text{ReEncrypt})$.

Let q be a large prime, $\Phi(x) = x^n + 1 \in \mathbb{Z}_q[x]$ the cyclotomic polynomial, with n the security level of our scheme. We define the quotient ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\Phi(x)$.

Let $l, r \in \mathbb{N}$, $\mathcal{D}_e$ be a uniform distribution on $[\![-b, b]\!]$, with $b \ll \frac{q}{2}$, and $\text{Samp}_{\mathcal{D}_e}$ be a sampler of polynomials in $\mathcal{R}_q$ according to the distribution $\mathcal{D}_e$, that takes a random string in $\{0,1\}^r$ as seed.

We define two pseudorandom hash functions defined as random oracles:

- $H_0 : \{0,1\}^l \longrightarrow \mathcal{R}_q$
- $H_1 : \mathcal{R}_q \times \{0,1\}^l \longrightarrow \{0,1\}^r$

We define the pseudo-random function $F : \mathcal{R}_q \times \{0,1\}^l \longrightarrow \mathcal{R}_q$ as follows:

$$
\begin{aligned}
&F(s, x) : \\
&\qquad a \longleftarrow H_0(x) && (1)\\
&\qquad \rho \longleftarrow H_1(s, x) && (2)\\
&\qquad e \xleftarrow{\$} \text{Samp}_{\mathcal{D}_e}(\rho) && (3)\\
&\qquad \text{output } y = a \cdot s + e && (4)
\end{aligned}
$$

As shown by [7], the function F is pseudo-random under the RLWE assumption.

Our scheme is defined as follows:

KeyGen(1^λ) : On input a security parameter λ, the key generation algorithm returns a random secret key $k \in \mathcal{R}_q$.

Encrypt(k, m) : On input a key $k \in \mathcal{R}_q$ and a message $m \in \{0,1\}^*$, we start by splitting the message into blocks m_i of size n, $i = 0, 1, \ldots$.

We then represent each block m_i as a polynomial p_i in $\mathcal{R}_q$ using the polynomial representation φ:

$$
\begin{aligned}
\varphi : \{0,1\}^n &\longrightarrow \mathcal{R}_q \\
m_i = (m_{i,0}, m_{i,1}, \ldots, m_{i,n-1}) &\longmapsto \lfloor \frac{q}{2} \rfloor \cdot m_{i,0} + \lfloor \frac{q}{2} \rfloor \cdot m_{i,1} \cdot x + \ldots \\
&\quad + \lfloor \frac{q}{2} \rfloor \cdot m_{i,n-1} \cdot x^{n-1}
\end{aligned} \tag{5}
$$

That is, the coefficients of p_i are obtained by multiplying each bit by $\lfloor \frac{q}{2} \rfloor$. With such a representation, there is no need for padding since the last coefficients will be set to 0.

We then obtain the encrypted block $c_i = p_i + F(k, i)$, and we return the ciphertext $c = [c_0|c_1|\dots|c_{n-1}]$. Such ciphertext is indistinguishable from uniform sampling [15].

Decrypt(k, c) : Given a key $k \in \mathcal{R}_q$ and a ciphertext $c = [c_0|c_1|\dots|c_{n-1}]$, with $c_i \in \mathcal{R}_q$, and $[c_i|c_{i+1}]$ being the concatenation of c_i and c_{i+1}. we calculate, for each c_i the polynomial $\omega_i = c_i - F(k, i)$.

We then find the clear block m_i using the inverse representation κ^{-1}:

$$\begin{gathered} \kappa^{-1} : \mathcal{R}_q \longrightarrow \{0,1\}^n \\ \omega_i = \omega_{i,0} + \omega_{i,1} \cdot x + \dots + \omega_{i,n-1} \cdot x^{n-1} \longmapsto m_i = (m_{i,0}, m_{i,1}, \dots, m_{i,n-1}) \end{gathered} \tag{6}$$

with

$$m_{i,j} := \begin{cases} 0 \text{ if } |\omega_{i,j}| < \lfloor \frac{q}{4} \rfloor \\ 1 \text{ if } |\omega_{i,j}| \geq \lfloor \frac{q}{4} \rfloor \end{cases} \tag{7}$$

where $|\omega_{i,j}|$ denotes the absolute value of $\omega_{i,j}$.

We return the plaintext $m = [m_0|m_1|\dots|m_{n-1}]$.

ReKeyGen(k_1, k_2) : On input the old key $k_1 \in \mathcal{R}_q$ and the new key $k_2 \in \mathcal{R}_q$, we return the rekeying token $\Delta = k_2 - k_1$.

It is possible to send Δ to a remote server, where the ciphertext is stored, without the server being able to learn k_1, k_2, or discover the plaintext.

ReEncrypt(Δ, c) : Given a rekeying token $\Delta \in \mathcal{R}_q$ and a ciphertext $c = [c_0|c_1|\dots|c_{n-1}]$, with $c_i \in \mathcal{R}_q$, we compute, for each c_i, the new encrypted block c'_i by:

$$c'_i = c_i + F(\Delta, i) \tag{8}$$

We then return the new ciphertext $c' = [c'_0|c'_1|\dots|c'_{n-1}]$.

The number of key rotations is, however, limited by the accumulation of errors e at the time of the ReEncrypt operation. The minimum bound on the number of rotations that can be performed is $\frac{q}{4b}$.

4 Our key rotation scheme with security update

Inspired by the block cipher scheme with key rotation described in section 3, we propose a new scheme that allows, during key rotation, to double the security of the key. Let λ be the initial security level (for example $\lambda = 128$), we can increase the security level to 2λ in the first round of key rotation, then 4λ in the next round etc. To double the security level, our scheme merges contiguous blocks c_{2i}

and c_{2i+1}. Our scheme also allows key rotation without updating the security level.

We define our scheme $\Pi = (\text{KeyGen}, \text{Encrypt}, \text{Decrypt}, \text{ReKeyGen}, \text{ReEncrypt}, \text{PrepareReEncryptSecUpd}, \text{ReKeyGenSecUpd}, \text{ReEncryptSecUpd})$.

Let's start by defining $\lambda_{\max}$, the upper bound on the security level for our scheme. That is, the security level for which we can reasonably assume that there will be no attack while the secret is being stored. For example, we take $\lambda_{\max} = 8192$ bits of security. We define $\Lambda = \{\lambda, 2\lambda, \ldots, \lambda_{\max}\}$ as the set of security levels in our scheme.

We choose a prime integer q large enough, such that $\lambda_{\max}$ divides $q-1$, for example $q = 945586177$. $\lambda_{\max}$ must divide $q-1$, which allows us to use the Number-Theoretic-Transform (NTT) [23] to speed up the multiplication of polynomials over the successive polynomial rings. We provide a simple Python script to generate a suitable prime [5].

We choose the polynomials $\Phi^{(\lambda)}$, $\Phi^{(2\lambda)}$, $\Phi^{(4\lambda)}$ etc. of degrees respectively λ, 2λ, 4λ etc. to be irreducible over $\mathbb{Z}$ and $\mathbb{Z}_q$.

We define the successive quotient rings $\mathcal{R}_q^{(\lambda)} = \mathbb{Z}_q[x]/\Phi^{(\lambda)}(x)$, $\mathcal{R}_q^{(2\lambda)} = \mathbb{Z}_q[x]/\Phi^{(2\lambda)}(x)$ etc.

For each key level $\zeta \in \Lambda$, let $\mathcal{D}_e$ be a uniform distribution on $[\![-b, b]\!]$, with $b \ll \frac{q}{2}$, and $\text{Samp}_{\mathcal{D}_e}$ be a sampler of polynomials in $\mathcal{R}_q^{(\zeta)}$ according to the distribution $\mathcal{D}_e$, that takes a random string in $\{0,1\}^r$ as seed, with $r \in \mathbb{N}$.

We define the pseudo-random function $F^{(\zeta)} : \mathcal{R}_q^{(\zeta)} \times \mathcal{R}_q^{(\lambda)} \times \mathbb{N} \longrightarrow \mathcal{R}_q^{(\zeta)}$ as follows:

$$
\begin{aligned}
&F^{(\zeta)}(k, a, i) : \\
&\qquad g \longleftarrow \lceil \log_2(\zeta) \rfloor - \lceil \log_2(\lambda) \rfloor && (9)\\
&\qquad \rho \longleftarrow H(k, i) && (10)\\
&\qquad e \xleftarrow{\$} \text{Samp}_{\mathcal{D}_e}(\rho) && (11)\\
&\qquad \text{output } y = a^{2^g \cdot i + 1} \cdot k + e && (12)
\end{aligned}
$$

With $H : \mathcal{R}_q \times \mathbb{N} \longrightarrow \{0,1\}^r$ a pseudorandom hash functions defined as a random oracle.

KeyGen(1^λ) : On input a security parameter λ, we output a key $k \in \mathcal{R}_q^{(\lambda)}$, and an Initialization Vector (IV) $a \in \mathcal{R}_q^{(\lambda)}$, of maximum multiplicative order on $\mathcal{R}_q^{(\lambda)}$, $\mathcal{R}_q^{(2\lambda)}$ etc.

Definition 2. *Let $\mathcal{R}$ be a ring, $e \in \mathcal{R}$. We say that e is of* maximum multiplicative order *on $\mathcal{R}$ if the smallest integer k such that $e^k = 1_{\mathcal{R}}$ is $k = |\mathcal{R}| - 1$.*

[5] `https://github.com/thomasarmel/block_cipher_updatable_security/blob/master/good_prime_generator.py`

Remark 2. In practice, ensuring that the IV a is of maximum multiplicative order on $\mathcal{R}_q^{(\lambda)}$, $\mathcal{R}_q^{(2\lambda)}$ etc. can be very complex. Indeed, for this it is necessary to test that $a^\alpha \neq 1$, for any α divisor of $|\mathcal{R}_q^{(\gamma)}| - 1 = q^\gamma - 1$, for all $\gamma \in \Lambda$. The number of divisors being gigantic, it is possible to restrict to only α lower than a limit l. We will then obtain an IV a such that $\mathrm{ord}_{\mathcal{R}_q^{(\gamma)}}(a) > l$. We explore this algorithm in Appendix A. In such case, this pair (IV, key) allows to encrypt at most l blocks $= l \cdot \lambda$ bits to guarantee the IND-CPA property.

The key k is kept secret, the IV a is public.

Encrypt(k, a, m) : Given a key $k \in \mathcal{R}_q^{(\lambda)}$, an IV $a \in \mathcal{R}_q^{(\lambda)}$ and a message $m \in \{0,1\}^*$, we start by splitting the message into blocks m_i of size n, $i = 0, 1, \ldots$.

We then represent each block m_i as a polynomial p_i in $\mathcal{R}_q^{(\lambda)}$ using the polynomial representation φ, as given in eq. 5.

Each block m_i is then encrypted to c_i with the operation:

$$c_i = m_i + F^{(\lambda)}(k, a, i) \tag{13}$$

The ciphertext c is obtained by concatenating the blocks c_i: $c = [c_0|c_1|\ldots|c_{n-1}]$.

The output is then given by the pair $(c, \text{key_increase_gen} = 0)$: there has not yet been an increase in the key size, so key_increase_gen $= 0$.

Decrypt(k, a, c) : Given a key $k \in \mathcal{R}_q^{(\zeta)}$, an IV $a \in \mathcal{R}_q^{(\lambda)}$ and a ciphertext $c = [c_0|c_1|\ldots|c_{n-1}]$, with $c_i \in \mathcal{R}_q^{(\zeta)}$ $(0 \leq i < n)$, we calculate, for each c_i the polynomial $\omega_i = c_i - F^{(\zeta)}(k, i)$.

We then find the clear block m_i using the inverse representation κ^{-1}, as described in eq. 6.

We return the plaintext $m = [m_0|m_1|\ldots|m_{n-1}]$.

ReKeyGen(k_1, k_2) : On input the old key $k_1 \in \mathcal{R}_q^{(\zeta)}$ and the new key $k_2 \in \mathcal{R}_q^{(\zeta)}$, we will proceed as in section 3 and output the key rotation token $\Delta = k_2 - k_1$.

ReEncrypt(Δ, a, c) : Given a rekeying token $\Delta \in \mathcal{R}_q^{(\zeta)}$, an IV $a \in \mathcal{R}_q^{(\lambda)}$ and a ciphertext $c = [c_0|c_1|\ldots|c_{n-1}]$, with $c_i \in \mathcal{R}_q^{(\zeta)}$, we compute, for each c_i, the new encrypted block c_i' by:

$$c_i' = c_i + F^{(\zeta)}(\Delta, a, i) \tag{14}$$

We then return the new ciphertext $c' = [c_0'|c_1'|\ldots|c_{n-1}']$, and increment the key_increase_gen metadata field.

PrepareReEncryptSecUpd(k, a, ζ) : On input a key $k \in \mathcal{R}_q^{(\zeta)}$, an IV $a \in \mathcal{R}_q^{(\lambda)}$ and the current security level $\zeta \in \mathbb{N}$, we compute the current key increasing generation $g = \lceil \log_2(\zeta) \rfloor - \lceil \log_2(\lambda) \rfloor$.

We then prepare the sequence of blocks $z = [z_0|z_1|...|z_{n-1}]$, with each $z_i \in \mathcal{R}_q^{(2\zeta)}$ $(0 \leq i < n)$ given by:

$$z_i = [a^{g \cdot i+1} \cdot k - (a^{g \cdot i+1} \cdot k \mod \Phi^{(\zeta)})] + e \mod \Phi^{(2\zeta)} \tag{15}$$

with $e \in \mathcal{R}_q^{(2\zeta)}$ a polynomial with coefficients randomly sampled from $\mathcal{D}_e$.

We finally output z, which can be sent in advance to the remote server.

In the case where the number of blocks is odd, we add a last block of zero bits encrypted with the key k.

ReKeyGenSecUpd($k, k_{\text{incr}}, a, \zeta$) : On input the old key $k \in \mathcal{R}_q^{(\zeta)}$, the new key $k_{\text{incr}} \in \mathcal{R}_q^{(2\zeta)}$, an IV $a \in \mathcal{R}_q^{(\lambda)}$ and the current security level $\zeta \in \mathbb{N}$, we compute the current key increasing generation $g = \log_2(\zeta) - \log_2(\lambda)$.

We then calculate the temporary key of merged blocks $k_{\text{tmp}} \in \mathcal{R}_q^{(2\zeta)}$ by

$$k_{\text{tmp}} = k \cdot (x^{\zeta} \cdot a^p + 1) \mod \Phi^{(2\zeta)} \tag{16}$$

with

$$p = 2^g \tag{17}$$

We finally calculate

$$\Delta_{\text{incr}} = k_{\text{incr}} - k_{\text{tmp}} \tag{18}$$

We output Δ_{incr}, and we keep k_{incr} secret.

ReEncryptSecUpd($z, \Delta_{\text{incr}}, c$) : Given $z \in \mathcal{R}_q^{(2\zeta)}$ the modular difference blocks generated by **PrepareReEncryptSecUpd**, $\Delta_{\text{incr}} \in \mathcal{R}_q^{(2\zeta)}$ the key increase token and the ciphertext $c = [c_0|c_1|\ldots|c_{n-1}]$, with $c_i \in \mathcal{R}_q^{(\zeta)}$, we start by merging the blocks of c two by two to obtain $c_{\text{tmp}} = [c_{\text{tmp},0}|c_{\text{tmp},1}|\ldots|c_{\text{tmp},n-1}]$, with $c_{\text{tmp},i} \in \mathcal{R}_q^{(2\zeta)}$ given by

$$c_{\text{tmp},i} = c_{2i} + z_{2i} + x^{\zeta} \cdot (c_{2i+1} + z_{2i+1}) \tag{19}$$

We then calculate the new cipher $c' = [c'_0|c_1|\ldots|c_{\frac{n}{2}-1}]$, with $c'_i \in \mathcal{R}_q^{(2\zeta)}$ by:

$$c'_i = c_{\text{tmp},i} + \Delta_{\text{incr}} \tag{20}$$

We finally output c', and increment the key_increase_gen metadata field.

5 Security analysis

Our scheme is designed to allow a user to entrust re-encryption and security update operations to a potentially malicious server (e.g., on an untrusted cloud). In the remainder of this section, we will consider this server as the attacker. The attacker therefore has at his disposal:

- the public parameters of the scheme: the prime q and the irreducible polynomials $\Phi^{(\gamma)}$, $\gamma \in \Lambda$
- the IV a
- the ciphertexts c after an arbitrary number of re-encryption and key update operations
- an arbitrary number of key rotation tokens Δ
- an arbitrary number of modular difference blocks z
- an arbitrary number of security update tokens Δ_{incr}.

We formalize the knowledge that the adversary can gain for the different security levels IND-CPA and IND-CCA2. We show that our scheme can easily be modified to ensure indistinguishability between a ciphertext before and after re-encryption with same security level, by simply removing "generation" (key_increase_gen) of the ciphertext next to it. Finally, we explain how it would be possible to modify our implementation to make it constant-time, at the cost of performance degradation.

Let Π = (KeyGen, Encrypt, Decrypt, ReKeyGen, ReEncrypt, PrepareReEncryptSecUpd, ReKeyGenSecUpd, ReEncryptSecUpd), be our block cipher with key rotation and security update scheme, for a message space $\mathcal{M} = \{0,1\}^*$. Let λ be the initial security level, and $h \in \mathbb{N}$ the maximum generation (ie the maximum key rotation count), as defined in section 3. We will assume that the pair (key, IV) is used to encrypt a number of blocks of size λ less than the limit l as defined in remark 2.

In fig. 1, we define the oracles $\mathcal{O} = (\mathcal{O}_{\text{Encryt}}, \mathcal{O}_{\text{ReKeyToken}}, \mathcal{O}_{\text{ReEncryt}}, \mathcal{O}_{\text{ModDiffBlocks}}, \mathcal{O}_{\text{SecUpdToken}}, \mathcal{O}_{\text{TmpUpdateBlocks}}, \mathcal{O}_{\text{SecUpdate}})$ corresponding to our primitives. These oracles maintain a lookup table $\mathbb{N} \longrightarrow \mathcal{R}_q^{(\zeta)}$, $\zeta \in \Lambda$, that maps key indexes to keys. The different keys k_i in the table are designed for potential increases in their security over time. That is, for all $i \in [\![0, h-1]\!]$, either $\dim(k_{i+1}) = \dim(k_i)$ or $\dim(k_{i+1}) = 2\dim(k_i)$.

5.1 IND-CPA

Indistinguishability under chosen-plaintext attack (IND-CPA) is a property of ciphertext indistinguishability in encryption scheme. It means that an adversary should not be able to distinguish a pair of ciphertext based on the plaintext they encrypt.

Given a PPT adversary $\mathcal{A}$, we define the challenge oracle $\mathcal{O}_{\text{challengeCPA}}$ and the IND-CPA experiment $\text{Expt}_{\Pi}^{\text{CPA}}$ in fig. 2. The adversary $\mathcal{A}$ may make

function $\mathcal{O}_{\textsc{Encrypt}}(i, a, m)$
  Output $\text{Encrypt}(k_i, a, m)$
end function

function $\mathcal{O}_{\textsc{ReKeyToken}}(i, j, a)$
  if $j \leq i$ or $j > h$ or $\text{security_level}(k_j) \neq \text{security_level}(k_i)$ **then**
    Output $\perp$
  end if
  $\Delta \leftarrow \text{ReKeyGen}(k_i, k_j)$
  Output Δ
end function

function $\mathcal{O}_{\textsc{ReEncryt}}(i, j, a, c)$
  if $j \leq i$ or $j > h$ or $\text{security_level}(k_j) \neq \text{security_level}(k_i)$ **then**
    Output $\perp$
  end if
  $\Delta \leftarrow \text{ReKeyGen}(k_i, k_j)$
  $c' \leftarrow \text{ReEncrypt}(\Delta, a, c)$
  Output c'
end function

function $\mathcal{O}_{\textsc{ModDiffBlocks}}(i, a)$
  $z \leftarrow \text{PrepareReEncryptSecUpd}(k_i, a, \text{security_level}(k_i))$
  Output z
end function

function $\mathcal{O}_{\textsc{SecUpdToken}}(i, j, a)$
  if $j \leq i$ or $j > h$ or $\text{security_level}(k_j) \neq 2 \cdot \text{security_level}(k_i)$ **then**
    Output $\perp$
  end if
  $\Delta_{\text{incr}} \leftarrow \text{ReKeyGenSecUpd}(k_i, k_j, a, \text{security_level}(k_i))$
  Output Δ_{incr}
end function

function $\mathcal{O}_{\textsc{TmpUpdateBlocks}}(z, c)$
  if $j \leq i$ or $j > h$ or $\text{security_level}(k_j) \neq 2 \cdot \text{security_level}(k_i)$ **then**
    Output $\perp$
  end if
  $z \leftarrow \text{PrepareReEncryptSecUpd}(k_i, a, \text{security_level}(k_i))$
  $\zeta \leftarrow \text{security_level}(k_i)$
  $c_{\text{tmp},i} \leftarrow c_{2i} + z_{2i} + x^{\zeta} \cdot (c_{2i+1} + z_{2i+1})$
  Output $c_{\text{tmp}} = [c_{\text{tmp},0} | c_{\text{tmp},1} | \dots]$
end function

function $\mathcal{O}_{\textsc{SecUpdate}}(i, j, a, c)$
  if $j \leq i$ or $j > h$ or $\text{security_level}(k_j) \neq 2 \cdot \text{security_level}(k_i)$ **then**
    Output $\perp$
  end if
  $\Delta_{\text{incr}} \leftarrow \text{ReKeyGenSecUpd}(k_i, k_j, a, \text{security_level}(k_i))$
  $z \leftarrow \text{PrepareReEncryptSecUpd}(k_i, a, \text{security_level}(k_i))$
  $c' \leftarrow \text{ReEncryptSecUpd}(z, \Delta_{\text{incr}}, c)$
  Output c'
end function

Fig. 1. Security experiments Oracles for our scheme's primitives

function $\text{EXPT}_{\Pi}^{\text{CPA}}(\lambda, \mathcal{A})$
    $(k, a) \leftarrow \text{KeyGen}(1^{\lambda})$
    $b' \leftarrow \mathcal{A}^{\mathcal{O}}(a)$
    Output b'
end function

function $\mathcal{O}_{\text{CHALLENGECPA}}(i, a, m_0, m_1)$
    Choose $b \xleftarrow{\$} \{0, 1\}$
    $c_b \leftarrow \text{Encrypt}(k_i, a, m_b)$
    Output c_b
end function

Fig. 2. IND-CPA challenge experiment and oracle

any polynomial number of queries to the primitive oracle $\mathcal{O}$ and the challenge oracle $\mathcal{O}_{\text{challengeCPA}}$.

Definition 3. *A block cipher with key rotation and security update scheme Π has* indistinguishable encryptions under a chosen-plaintext attack *(IND-CPA) if for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$ there exists a negligible function negl$(\cdot)$ such that:*

$$\Pr\left[Expt_{\Pi}^{CPA}(\lambda, \mathcal{A}) = 1\right] \leq \frac{1}{2} + negl(\lambda) \tag{21}$$

Definition 3 must hold even if the adversary $\mathcal{A}$ is equipped with a sufficiently powerful quantum computer.

We now show that our scheme Π is IND-CPA secure, assuming the hardness of RLWE.

Theorem 1. *The scheme Π = (KeyGen, Encrypt, Decrypt, ReKeyGen, ReEncrypt, PrepareReEncryptSecUpd, ReKeyGenSecUpd, ReEncryptSecUpd) has indistinguishbale encryptions under a chosen-plaintext attack (IND-CPA), assuming the RLWE assumption.*

Proof. We prove that if there exists a Probabilistic Polynomial Time (PPT) adversary $\mathcal{A}$ that breaks the IND-CPA security of our scheme Π with non-negligible advantage ϵ, then there exists a PPT algorithm $\mathcal{B}$ that solves the RLWE decision problem with non-negligible advantage.

Suppose $\mathcal{A}$ is an IND-CPA adversary. We construct a RLWE distinguisher $\mathcal{B}$ that takes a pair $(a(x), c(x)) \in (\mathcal{R}_q^{(\zeta)})^2$ and distinguishes whether $c(x) = a(x) \cdot k(x) + e(x)$ or $c(x)$ is random, for random secret $k(x), e(x) \in \mathcal{R}_q^{(\zeta)}$, with $e(x)$ having its coefficients generated by the distribution $\mathcal{D}_e$.

$\mathcal{B}$ interacts with $\mathcal{A}$ using polynomial-time algorithms as follows:

$\mathcal{A}$ sends two chosen messages m_0 and m_1 to the oracle $\mathcal{O}_{\text{challengeCPA}}$, as well as an IV a and any key index. $\mathcal{A}$ receives a ciphertext c_b and calculates, for each block $c_{b,i}$ of c_b, $f_0 = [f_{0,0}|f_{0,1}|\dots]$ such that:

$$f_{0,i} = c_{b,i} - m_{0,i} \tag{22}$$

and $f_1 = [f_{1,0}|f_{1,1}|\dots]$ such that:

$$f_{1,i} = c_{b,i} - m_{1,i} \tag{23}$$

$\mathcal{A}$ then sends f_0 and f_1 to $\mathcal{B}$. Then, for each block $f_{0,i}$ of f_0, $\mathcal{B}$ tries to decide whether

$$f_{0,i} = a^{i+1} \cdot k + e,\ k, e \in \mathcal{R}_q^{(\zeta)} \tag{24}$$

If so, then it returns $b = 0$. Otherwise, if for each block $f_{1,i}$ of f_1, $\mathcal{B}$ finds

$$f_{1,i} = a^{i+1} \cdot k + e,\ k, e \in \mathcal{R}_q^{(\zeta)} \tag{25}$$

Then it returns $b = 1$.

Let's see what happens if $\mathcal{A}$ calls the other oracles.

If $\mathcal{A}$ calls $\mathcal{O}_{\text{ReEncrypt}}$, $\mathcal{O}_{\text{TmpUpdateBlocks}}$ or $\mathcal{O}_{\text{SecUpdate}}$, he will receive a new ciphertext c' with a new key k' that he does not know. We then return to the situation described above.

If $\mathcal{A}$ calls $\mathcal{O}_{\text{ReKeyToken}}$ or $\mathcal{O}_{\text{SecUpdToken}}$, he will receive the tokens Δ or Δ_{incr}. Since these tokens are defined as the difference of two keys k_1 and k_2 unknown to $\mathcal{A}$ over a finite ring $\mathcal{R}_q$, then the tokens do not provide any information about the keys k_1 and k_2. Therefore, the adversary cannot use the Δ or Δ_{incr} tokens to distinguish whether $b = 0$ or $b = 1$.

If $\mathcal{A}$ calls $\mathcal{O}_{\text{ModDiffBlocks}}$, then he will receive a $z = [z_0|z_1|\dots]$, with each z_i modular quotient over $\Phi^{(\zeta)}$, such that

$$\begin{aligned} z_i &= [a^{i+1} \cdot k - (a^{i+1} \cdot k \mod \Phi^{(\zeta)})] + e \mod \Phi^{(2\zeta)} \\ &= Q_i \cdot \Phi^{(\zeta)} + e \mod \Phi^{(2\zeta)} \end{aligned} \tag{26}$$

with $Q_i, e \in \mathcal{R}_q^{(2\zeta)}$, e random error with coefficients sampled from $\mathcal{D}_e$.

Then the attacker can call the $\mathcal{O}_{\text{TmpUpdateBlocks}}$ oracle which will return a series of blocks $c_{\text{tmp}} = [c_{\text{tmp},0}|c_{\text{tmp},1}|\dots]$, with

$$c_{\text{tmp},i} = c_{2i} + z_{2i} + x^{\zeta} \cdot (c_{2i+1} + z_{2i+1}) \mod \Phi^{(2\zeta)} \tag{27}$$

The attacker then subtracts m_0 and m_1 from c_{tmp} to obtain f_0 and f_1, which he then sends to Algorithm $\mathcal{B}$. Algorithm $\mathcal{B}$ must then decide whether, for each block of $f_{b'}$ (with $b' \in \{0, 1\}$):

$$\begin{aligned} f_{b',i} &= a^{2i+1} \cdot k + e_1 + (Q_1 \cdot \Phi^{(\zeta)} + e_2) + x^{\zeta} \cdot (a^{2i+2} \cdot k + e_3 + (Q_2 \cdot \Phi^{(\zeta)} + e_4)) \\ &= a^{2i+1} \cdot k \cdot (x^{\zeta} \cdot a + 1) + \Phi^{(\zeta)} \cdot (Q_1 + x^{\zeta} \cdot Q_2) + e \end{aligned} \tag{28}$$

With a and $\Phi^{(\zeta)}$ known, and k, Q_1, Q_2 and $e = e_1 + e_2 + x^{\zeta} \cdot (e_3 + e_4)$ unknown.

If algorithm $\mathcal{B}$ succeeds in verifying equation 28 for $b' = 0$, then it outputs $b = 0$, otherwise it outputs $b = 1$.

As the existence of such a $\mathcal{B}$ algorithm contradicts the RLWE assumption, then the advantage ϵ of the adversary $\mathcal{A}$ is negl(λ), so our scheme Π is IND-CPA secure under RLWE.

□

5.2 IND-CCA

Indistinguishability under chosen-ciphertext attack (IND-CCA) is a property of ciphertext indistinguishability in an encryption scheme. It ensures that an adversary cannot distinguish between the encryptions of two chosen plaintexts, even when given access to a decryption oracle that decrypts arbitrary ciphertexts of their choice (excluding the challenge ciphertext). There are two main variants: IND-CCA1 and IND-CCA2. In IND-CCA1, the adversary is allowed decryption queries only before receiving the challenge ciphertext, representing a limited form of attack. In contrast, IND-CCA2 allows the adversary to make decryption queries both before and after the challenge, excluding the challenge ciphertext itself.

Given an adversary $\mathcal{A}$, we define the challenge oracles $\mathcal{O}_{\text{Decrypt}}$, $\mathcal{O}_{\text{challengeCCA}}$ and the IND-CPA experiment $\text{Expt}_{\Pi}^{\text{CPA}}$ in fig. 3. The adversary $\mathcal{A}$ may make any numbers of queries to the primitive oracle $\mathcal{O}$, the challenge oracle $\mathcal{O}_{\text{challengeCPA}}$ and the decryption oracle $\mathcal{O}_{\text{Decrypt}}$ (except for the challenge ciphertext obviously).

Definition 4. *A block cipher with key rotation and security update scheme Π has* indistinguishable encryptions under an adaptative chosen-ciphertext attack *(IND-CCA2) if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl($\cdot$) such that:*

$$\Pr\left[Expt_{\Pi}^{CCA}(\lambda, \mathcal{A}) = 1\right] \leq \frac{1}{2} + negl(\lambda) \tag{29}$$

where the probability is taken over the random coins used by $\mathcal{A}$, as well as the random coins used in the experiment.

Definition 4 must hold even if the adversary $\mathcal{A}$ is equipped with a sufficiently powerful quantum computer.

We now show that our scheme Π is IND-CCA secure (and even IND-CCA2 secure), assuming the hardness of RLWE.

We showed in Theorem 1 that our scheme Π is IND-CPA secure assuming RLWE difficulty. Building on this theorem, we show that our scheme is IND-CCA2 secure, assuming RLWE difficulty. To do this, we allow the adversary to call the oracle $\mathcal{O}_{\text{Decrypt}}$ on all ciphertexts c, except the challenge ciphertext c^*.

function $\mathcal{O}_{\text{DECRYPT}}(i, a, c, c^*)$
  if $c = c^*$ **then**
    Output $\perp$ ▷ Challenge ciphertext is not allowed
  end if
  $m \leftarrow \text{Decrypt}(k_i, a, c)$
  Output m
end function

function $\text{EXPT}_{\Pi}^{\text{CCA}}(\lambda, \mathcal{A})$
  $(k, a) \leftarrow \text{KeyGen}(1^{\lambda})$
  $b' \leftarrow \mathcal{A}^{\mathcal{O}}(a)$
  Output b'
end function

function $\mathcal{O}_{\text{CHALLENGECCA}}(i, a, m_0, m_1)$
  Choose $b \xleftarrow{\$} \{0, 1\}$
  $c^* \leftarrow \text{Encrypt}(k_i, a, m_b)$
  Output c^*
end function

Fig. 3. Decryption oracle, IND-CCA challenge experiment and IND-CCA challenge oracle

Theorem 2. *The scheme Π = (KeyGen, Encrypt, Decrypt, ReKeyGen, ReEncrypt, PrepareReEncryptSecUpd, ReKeyGenSecUpd, ReEncryptSecUpd) has indistinguishbale encryptions under an adaptative chosen-ciphertext attack (IND-CCA2), assuming the RLWE assumption.*

Proof. We prove that if there exists a Probabilistic Polynomial Time (PPT) adversary $\mathcal{A}$ that breaks the IND-CCA2 security of our scheme Π with non-negligible advantage ϵ, then there exists a PPT algorithm $\mathcal{B}$ that solves the RLWE decision problem with non-negligible advantage.

Suppose $\mathcal{A}$ is an IND-CCA adversary. We construct a RLWE distinguisher $\mathcal{B}$ that takes a tuple $(a(x), c(x)) \in (\mathcal{R}_q^{(\zeta)})^2$ and distinguishes whether $c(x) = a(x) \cdot k(x) + e(x)$ or $c(x)$ is random, for random secret $k(x), e(x) \in \mathcal{R}_q^{(\zeta)}$, with $e(x)$ having its coefficients generated by the distribution $\mathcal{D}_e$.

$\mathcal{B}$ interacts with $\mathcal{A}$ using polynomial-time algorithms as follows:

$\mathcal{A}$ sends two chosen messages m_0 and m_1 to the oracle $\mathcal{O}_{\text{challengeCCA}}$, as well as an IV a^* and any key index. $\mathcal{A}$ receives a ciphertext c^* and calculates, for each block c_i^* of c^*, $f_0 = [f_{0,0}|f_{0,1}|\dots]$ such that:

$$f_{0,i} = c_i^* - m_{0,i} \tag{30}$$

and $f_1 = [f_{1,0}|f_{1,1}|\dots]$ such that:

$$f_{1,i} = c_i^* - m_{1,i} \tag{31}$$

$\mathcal{A}$ then calls $\mathcal{O}_{\text{Decrypt}}$ to decrypt the chosen ciphertexts c_0 and c_1 such that for each block $c_{b,i}$, $b \in \{0,1\}$:

$$c_{b,i} = m_{b,i} + a_b^{i+1} \cdot k + e_{b,i} \tag{32}$$

with a_0, a_1 IVs such that $a_0 \neq a_1 \neq a^*$ and e_b random error with coefficients sampled from $\mathcal{D}_e$.

For each block $c_{b,i}$ of c_0 and c_1, $\mathcal{A}$ is now able to distinguish $a_b^{i+1} \cdot k + e_{b,i}$ from random polynomials over $\mathcal{R}_q^{(\zeta)}$, but cannot learn the value of k directly.

Since $\mathcal{R}_q^{(\zeta)}$ is a Galois field and q is a prime bounded by poly(ζ), then [18,12] show that the search-RLWE and decision-RLWE problems are equivalent. $\mathcal{A}$ therefore calls algorithm $\mathcal{B}$ to discover key k from $a_b^{i+1} \cdot k + e_{b,i}$.

For each f_b, $b \in \{0,1\}$, $\mathcal{A}$, now knowing the key k, can therefore find in polynomial time whether for each block $f_{b,i}$:

$$f_b, i = (a^*)^{i+1} \cdot k + e,\ e \in \mathcal{R}_q^{(\zeta)} \tag{33}$$

If $\mathcal{A}$ manages to verify the equation for $b = 0$, then it outputs $b' = 0$. Otherwise, it outputs $b' = 1$.

As the existence of such a $\mathcal{B}$ algorithm contradicts the RLWE assumption, then the advantage ϵ of the adversary $\mathcal{A}$ is negl(λ), so our scheme Π is IND-CCA2 secure under RLWE.

□

5.3 Indistinguishability after re-encryption and security update

We have deliberately chosen not to support indistinguishability between two ciphertexts before and after key rotation. Indeed, we indicate the generation of the ciphertext in plaintext (key_increase_gen), so the adversary can know the number of key rotations that have already taken place.

If we did not indicate the generation in plaintext, then our scheme Π would satisfy the indistinguishability after re-encryption and security update property when the old key k_1 and the new key k_2 are of the same size, as stated by [6] (see def 3.4).

When the new key k_2 is of higher security than the old key k_1, we cannot guarantee indistinguishability after re-encryption and security update. Indeed, since the ciphertext size must be a multiple of the key size, it could increase. For example, suppose a ciphertext c of 3 blocks size, protected by a key k_1 with a security of 128 bits. Then $\text{len}(c) \approx 384 \cdot \lceil \log_2(q) \rceil$. If we rotate the key to a 256-bit security key k_2, then we must first pad with encrypted 128 null bits, so that the size is a multiple of 256. We obtain a new ciphertext c', and $\text{len}(c') \approx 512 \cdot \lceil \log_2(q) \rceil$. The adversary is then able to distinguish between the old ciphertext c and new ciphertext c'.

5.4 Constant time

The implementation we propose in section 6 is not constant-time. Indeed, we favored a high-performance implementation over a constant-time one because we are targeting use for long-time secret storage, not direct transmission. However, we can consider a method to make our implementation constant-time.

The authors of [19] developed a Go software library, saferith[6], which provides a constant-time implementation of the main basic arithmetic operations. Their paper details the performance impact of constant-time. For example, their implementation slows down execution of modular multiplication by 21.66 times compared to a classic implementation.

Here, we detail the arithmetic operations in our cryptographic scheme that require attention when implementing them in constant time.

KeyGen(1^λ) :

Here we generate a polynomial $k \in \mathcal{R}_q^{(\lambda)}$, that is to say of degree λ, whose coefficients lie in the interval $[\![-\frac{q-1}{2}, \frac{q-1}{2}]\!]$. For each coefficient, the reduction modulo q and the subtraction to fall in the interval $[\![-\frac{q-1}{2}, \frac{q-1}{2}]\!]$ must therefore be implemented constant-time.

Encrypt(k, a, m) :

Each block c_i of the ciphertext c is generated using the operation

$$c_i = m_i + a^{i+1} \cdot k + e_i \mod \mathcal{R}_q^{(\zeta)} \tag{34}$$

with m_i being the ith block of the message m, a being the public IV, k being the secret key, and e_i being a secret random error polynomial. An implementation will therefore seek to be constant-time regarding to m_i, k and e_i.

For the multiplication $a^{i+1} \cdot k \mod \Phi^{(\zeta)}$, one can start with a constant-time multiplication of a^{i+1} and k over $\mathbb{Z}_q[x]$. [9] proposes a polynomial multiplication algorithm with a predictable number of integer multiplications depending on q, $\deg(a^i+1)$ and $\deg(k)$. The reduction modulo $\Phi^{(\zeta)}$ can be done in constant-time, using the modular integer inversion method presented by [5].

Decrypt(k, a, c) :

Here, each block m_i of the plaintext m is retrieved using the equation:

$$m_i = \kappa^{-1}(c_i - F^{(\zeta)}(k, a, i)) \tag{35}$$

with $F^{(\zeta)}$ being the pseudo-random function described in Section 4, and κ^{-1} being the inverse representation described in equation 6.

To implement the $F^{(\zeta)}$ function in constant time, the same method as described previously for the **Encrypt**(k, a, m) function will be used. The modulo q subtraction of the coefficients from those of c_i can be easily implemented in

[6] `https://pkg.go.dev/github.com/cronokirby/saferith`

constant time. Finally, to implement the κ^{-1} inverse representation in constant time, a simple branchless programming method can be used.

ReKeyGen(k_1, k_2) :
The modulo q subtraction of the coefficients of k_1 from those of k_2 should be implemented constant-time.

ReEncrypt(Δ, a, c) :
Since this operation does not depend on the secret key k or the plaintext m, there is no need to implement it constant-time.

PrepareReEncryptSecUpd(k, a, ζ) :
This operation returns a sequence of modular difference blocks $z = [z_0|z_1|\dots]$, such that:

$$z_i = [a^{g \cdot i+1} \cdot k - (a^{g \cdot i+1} \cdot k \mod \Phi^{(\zeta)})] + e \mod \Phi^{(2\zeta)} \tag{36}$$

Similar to **Encrypt**(k, a, m), the modular multiplication, addition and subtraction of polynomials must be implemented in constant time.

ReKeyGenSecUpd($k, k_{\text{incr}}, a, \zeta$) :
This operation calculates the Δ_{incr} token for updating the security level, with

$$\Delta_{\text{incr}} = k_{\text{incr}} - k \cdot (x^{\zeta} \cdot a^{p} + 1) \mod \Phi^{(2\zeta)} \tag{37}$$

Similar to **Encrypt**(k, a, m), modular multiplication and subtraction should be implemented in constant time.

ReEncryptSecUpd($z, \Delta_{\text{incr}}, c$) :
Since this operation does not depend on the secret key k or the plaintext m, there is no need to implement it constant-time.

6 Proof of concept

We developed a proof of concept for our cryptographic scheme using the Rust programming language. Our code is available at `https://github.com/thomasarmel/block_cipher_updatable_security/`.

We chose the Rust language primarily because the compiler is capable of detecting *Single Instruction on Multiple Data* (SIMD) optimizations, like with the AVX-512 instruction set extension on the x86 architecture. To do this, we compiled our executable with the "`target-cpu=native`" option.

To speed up calculations, we implemented some algorithmic optimizations:

- Since the exponent a^i of the IV a is incremental, we cache the successive exponentiations, so that $a^{i+1} = a \cdot \text{cache}[i]$. This optimization reduces the complexity of calculating the exponent relative to exponent i to $\mathcal{O}(1)$, instead of $\mathcal{O}(\log(i))$ if we had used the square-and-multiply algorithm. However, this optimization prevents encrypting blocks in parallel and increases memory consumption.
- We also optimized the multiplication of polynomials over the ring $\mathcal{R}_q^{(\zeta)}$ using the *number-theoretic transform* [22], which is a specialization of the discrete Fourier transform over $\mathbb{Z}_p$ rings, p being a prime.

7 Performance

We measured the performance of our cryptographic scheme on one core of our Intel Core i7-13700H. Here is the throughput for different operations, depending on the block size:

Encryption, 128 bits of security: 22.8 kB/s
Decryption, 128 bits of security: 193.9 kB/s
Security update from 128 bits to 256 bits: 14.5 kB/s
Decryption, 256 bits of security: 131.8 kB/s
Security update from 256 bits to 512 bits: 8.2 KB/s
Decryption, 512 bits of security: 82.3 kB/s

Our cryptographic scheme also uses more storage space than the raw plaintext: let l be the length of the plaintext in bits, then the length L of the ciphertext c is given by:

$$L = O(l \cdot \log_2(q)) \tag{38}$$

We must also add the size L' of the sequence of modular difference blocks z:

$$L' = O(2l \cdot \log_2(q)) \tag{39}$$

The remote server will therefore have to store a maximum of $L + L'$ bits, and compression algorithms are not very effective at reducing the size of the ciphertext.

8 Discussion

We developed a symmetric block cipher scheme that allows key rotation with or without security level update, without decryption. Our scheme could be used for long-term secret storage, increasing the security level of an encrypted file stored on an untrusted remote server.

Future work would be interested in reducing the size of the encrypted file, as well as improving encryption and security update performance.

The formulation of the LWE assumption, on which the security of our scheme is based, is relatively recent in the history of mathematical study. We therefore suggest applying double AES encryption + our scheme to protect sensitive data. Thus, if the LWE assumption were to be compromised, the data would still be protected with the initial security level.

9 Conclusion

In this paper, we propose a new cryptographic scheme for symmetric block ciphers. Our scheme allows for simple key rotation and key rotation with security level updates. It is therefore possible, at each rotation, to re-encrypt with a key of the same security level, or to double the security level, without decrypting. Our scheme is therefore attractive for long-term secret storage on an untrusted remote server: it is possible to update the ciphertext on the remote server without the server being able to decrypt the message.

The security of our scheme is based on the RLWE assumption. We prove the security of our scheme at security levels IND-CPA and IND-CCA2, as long as the RLWE assumption stands. We also propose an implementation written in Rust. This implementation is not constant-time, as we believe the performance loss incurred would not be worthwhile given the benefits of constant-time in our use case. However, we show how to modify our implementation to make it constant-time.

Finally, we analyze the performance of our implementation, as well as the storage space used by our ciphertext.

Acknowledgements We would like to thank Tom Niget[7] for his theoretical support.

References

1. Alagic, G., Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Liu, Y.K., Miller, C., et al.: Status report on the third round of the NIST post-quantum cryptography standardization process (2022). https://doi.org/10.6028/NIST.IR.8413-upd1
2. Aono, Y., Hayashi, T., Phong, L.T., Wang, L.: Efficient homomorphic encryption with key rotation and security update. IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences (2018). https://doi.org/10.1145/3055259.3055260
3. Ateniese, G., Benson, K., Hohenberger, S.: Key-private proxy re-encryption. In: Topics in Cryptology–CT-RSA 2009: The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings. Springer (2009). https://doi.org/10.1007/978-3-642-00862-7_19

[7] https://niget.fr/

4. Berbain, C., Gilbert, H., Patarin, J.: QUAD: A practical stream cipher with provable security. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer (2006). https://doi.org/10.1007/11761679_8
5. Bernstein, D.J., Yang, B.Y.: Fast constant-time gcd computation and modular inversion. IACR transactions on cryptographic hardware and embedded systems (2019). https://doi.org/10.13154/tches.v2019.i3.340-398
6. Boneh, D., Eskandarian, S., Kim, S., Shih, M.: Improving speed and security in updatable encryption schemes. In: Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea. Springer (2020). https://doi.org/10.1007/978-3-030-64840-4_19
7. Boneh, D., Lewi, K., Montgomery, H., Raghunathan, A.: Key homomorphic PRFs and their applications. In: Annual Cryptology Conference. Springer (2013). https://doi.org/10.1007/978-3-642-40041-4_23
8. Bouillaguet, C.: Another Look at the Cost of Cryptographic Attacks (2019), https://hal.science/hal-02306912
9. Cenk, M., Koç, C.K., Ozbudak, F.: Polynomial multiplication over finite fields using field extensions and interpolation. In: 2009 19th IEEE Symposium on Computer Arithmetic. IEEE (2009). https://doi.org/10.1109/ARITH.2009.11
10. Chen, H., Lauter, K., Stange, K.E.: Attacks on the search RLWE problem with small errors. SIAM Journal on Applied Algebra and Geometry (2017). https://doi.org/10.1137/16M1096566
11. Chow, S.S., Weng, J., Yang, Y., Deng, R.H.: Efficient unidirectional proxy re-encryption. In: International Conference on Cryptology in Africa. Springer (2010). https://doi.org/10.1007/978-3-642-12678-9_19
12. Eisenträger, K., Hallgren, S., Lauter, K.: Weak instances of PLWE. In: International Conference on Selected Areas in Cryptography. Springer (2014). https://doi.org/10.1007/978-3-319-13051-4_11
13. Elias, Y., Lauter, K.E., Ozman, E., Stange, K.E.: Provably weak instances of Ring-LWE. In: Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA. Springer (2015). https://doi.org/10.1007/978-3-662-49890-3_6
14. Everspaugh, A., Paterson, K., Ristenpart, T., Scott, S.: Key rotation for authenticated encryption. In: Annual international cryptology conference. Springer (2017). https://doi.org/10.1007/978-3-319-63697-9_4
15. Fouque, P.A., Hadjibeyli, B., Kirchner, P.: Homomorphic evaluation of lattice-based symmetric encryption schemes. In: International Computing and Combinatorics Conference. Springer (2016). https://doi.org/10.1007/978-3-319-42634-1_22
16. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (1996). https://doi.org/10.1145/237814.237866
17. Kirshanova, E.: Proxy re-encryption from lattices. In: Public-Key Cryptography–PKC 2014: 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina. Springer (2014). https://doi.org/10.1007/978-3-642-54631-0_5
18. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera. Springer (2010). https://doi.org/10.1145/2535925

19. Meier, L.C., Colombo, S., Thiercelin, M., Ford, B.: Constant-time arithmetic for safer cryptography. Cryptology ePrint Archive (2021)
20. Nuñez, D., Agudo, I., Lopez, J.: Proxy re-encryption: Analysis of constructions and its application to secure access delegation. Journal of Network and Computer Applications (2017). https://doi.org/10.1016/j.jnca.2017.03.005
21. Sakurai, K., Nishide, T., Syalim, A.: Improved proxy re-encryption scheme for symmetric key cryptography. In: 2017 International Workshop on Big Data and Information Security (IWBIS). IEEE (2017). https://doi.org/10.1109/IWBIS.2017.8275110
22. Satriawan, A., Mareta, R., Lee, H.: A complete beginner guide to the number theoretic transform (NTT). Cryptology ePrint Archive (2024). https://doi.org/10.1109/ACCESS.2023.3294446
23. Satriawan, A., Syafalni, I., Mareta, R., Anshori, I., Shalannanda, W., Barra, A.: Conceptual review on number theoretic transform and comprehensive review on its implementations. IEEE Access (2023). https://doi.org/10.1109/ACCESS.2023.3294446
24. Vikström, V.: Developing PCI DSS compliant configuration standards (2025)

Appendix A High multiplicative order IV finding algorithm

Here we present an algorithm for generating an IV a whose multiplicative order on $\mathcal{R}_q^{(\gamma)}$ is greater than the limit $l \in \mathbb{N}$, for all $\gamma \in \Lambda$. A proof of concept is also given at https://github.com/thomasarmel/block_cipher_updatable_security/blob/master/iv_generator_good_mult_order.sage.

In practice, we find empirically in our tests that the probability that $\text{ord}_{\mathcal{R}_q^{(\gamma)}}(a) > l$ for all $\gamma \in \Lambda$, for $a \in \mathcal{R}_q^{(\lambda)}$ random, and $l = 10^8$ arbitrarily chosen, is close to 1. This means that an IV a chosen randomly in $\mathcal{R}_q^{(\lambda)}$ will have a high chance of validating the multiplicative order condition.

Algorithm 1 divisors(n, l): Compute all divisors of n that are below a limit l

Require: $n \in \mathbb{N}$
Require: $l \in \mathbb{N}$
  primes_dict $\leftarrow \{\}$ ▷ Empty dict
  for p in prime_factors(n) **do**
    if $p > l$ **then**
      break loop
    end if
    if p in primes_dict **then**
      primes_dict$[p] \leftarrow$ primes_dict$[p] + 1$
    else
      primes_dict$[p] \leftarrow 1$
    end if
  end for
  factors_list $\leftarrow []$ ▷ Empty array
  upcoming $\leftarrow [(1, 0)]$ ▷ Queue of factor and index
  while length(upcoming) > 0 **do** ▷ The queue is not empty
    (factor, next_i) $\leftarrow$ upcoming.pop()
    upcoming.append(factor)
    for i in $[\![$next_i, length(primes_dict) $- 1]\!]$ **do**
      (p, p_count) $\leftarrow i^{th}$ (key, value) tuple of primes_dict
      p_pow $\leftarrow p$
      for _ in $[\![0,$ p_count $- 1]\!]$ **do**
        next $\leftarrow$ factor $\cdot$ p_pow
        if next $\leq l$ **then**
          upcoming.push(next, $i + 1$)
          p_pow $\leftarrow$ p_pow $\cdot\ p$
        end if
      end for
    end for
  end while
  return factors_list

Algorithm 2 mult_order_enough(a, l): Check that $\text{ord}_{\mathcal{R}_q^{(\gamma)}}(l) > l$, for all $\gamma \in \Lambda$

Require: $a \in \mathcal{R}_q^{(\lambda)} \backslash \{0_{\mathcal{R}_q^{(\lambda)}}\}$
Require: $l \in \mathbb{N}$
  for γ in Λ **do**
    for d in divisors(γ, l) **do**
      if $a^d = 1$ **then**
        return False
      end if
    end for
  end for
  return True