Efficient Transciphering with Chacha20

Gabriela M. Jacob[1], **Hilder V. L. Pereira**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Campinas – SP – Brazil

g186087@dac.unicamp.br, hilder@ic.unicamp.br

Abstract. *Transciphering is a powerful technique to reduce communication overhead of fully homomorphic encryption. We propose to use the standardized cipher Chacha20 to construct transciphering. We implement a full bitwise homomorphic version of Chacha20 and analyze different strategies and techniques to implement it with non-binary messages. We also propose a non-black-box way of computing homomorphic XOR gates for free within Chacha20 round evaluation. This yields a significant reduction on the computation cost, with an estimate of 238.08 s to run the cipher in our best serial implementation, with our throughput being up to 2 times higher than the state of the art.*

1. Introduction

Fully homomorphic encryption (FHE) [Gentry 2009] allows a server to compute functions on encrypted data without ever decrypting it, which guarantees security and privacy of the client's data. In recent years, this technology has increasingly gained relevance in practical applications, such as private artificial neural networks, as well as in theoretical works, such as building indistinguishable obfuscation.

However, despite being extremely useful, FHE has practical challenges related to high computational costs and communication overheads, since encrypting a message tends to increase its size in at least 3 orders of magnitude [Chillotti et al. 2020]. Transciphering is a technique that aims to overcome this limitation by combining homomorphic encryption with traditional encryption methods [Thakur et al. 2025]. In summary, transciphering works as follows: a client who wishes to outsource the computation of some function f on their data x, instead of simply encrypting x with FHE and sending $\mathsf{FHE.enc}(x)$ to the server, chooses a symmetric cipher SC with secret key k, then encrypts x with SC, obtaining $\mathsf{SC.enc}(x)$ and uses FHE to encrypt k, getting $\mathsf{FHE.enc}(k)$. Finally, $\mathsf{SC.enc}(x)$ and $\mathsf{FHE.enc}(k)$ are sent to the server. Since symmetric ciphers do not increase the data size when they are encrypted, sending $\mathsf{SC.enc}(x)$ represents essentially the same communication cost as sending x in clear, and only k suffers from ciphertext expansion. However, k is typically very small, say 128 bits, thus, $\mathsf{FHE.enc}(k)$, although having much more than 128 bits, remains small.

The server, on its end, uses FHE capabilities to evaluate the decryption circuit of SC homomorphically, using $\mathsf{FHE.enc}(k)$, i.e., the server sets $F = \mathsf{SC.dec}$ and computes

$$c = \mathsf{Eval}(F, \mathsf{SC.enc}(x), \mathsf{FHE.enc}(k)) = \mathsf{FHE.enc}(\mathsf{SC.dec}_k(\mathsf{SC.enc}(x)) = \mathsf{FHE.enc}(x)$$

After that, the server can continue the homomorphic computation of the desired function f on x, since it has x encrypted under a homomorphic scheme.

There are two ways of constructing transciphering protocols [Thakur et al. 2025]. The first one is by proposing ad hoc symmetric ciphers that are FHE-friendly, i.e., which are designed so that their decryption function can be easily evaluated by FHE, achieving thus more efficient transciphering [Cosseron et al. 2022, Dobraunig et al. 2023]. The drawback of this strategy lies on its security, since ad hoc ciphers lack extensive cryptanalisys. The second approach relies on using standardized and well-established ciphers, trying to obtain the best possible efficiency without giving up security. In this case, AES is the most widely used cipher, but, since it was not designed to be FHE-friendly, it yields transciphering protocols with somewhat high latency and low throughput. This leads to the question of which standardized cipher would be the best one to construct transciphering.

In this paper, we investigate the suitability of using Chacha20 to design transciphering. For this, we propose a basic strategy to homomorphically evaluate Chacha20 and we implement it in Rust using the TFHE-rs library[1]. Our implementation is publicly available[2]. We also analyze other possible homomorphic implementations over various message space sizes and how it affects the efficiency of homomorphic versions of Chacha20.

2. Preliminaries

Hardness assumptions: The homomorphic encryption schemes used in this work are based in the Learning with Errors (LWE) and the Ring Learning with Errors (RLWE) problems [Lyubashevsky et al. 2010]. In a nutshell, those problems define a secret value s and samples of the form (a_i, b_i) with $b_i = a_i \cdot s + e_i$, where e_i is some low-norm term and a_i is uniformly distributed (where all the operations are performed in some ring, like $\mathbb{Z}_q$ for LWE and $\mathbb{Z}_q[X]/\langle X^N + 1\rangle$ for RLWE). Then, an attacker is given access to arbitrarily many LWE/RLWE samples and has to find the secret s.

Fully homomorphic encryption (FHE): FHE is an advanced cryptographic primitive that allows one to evaluate arbitrary functions over encrypted data [Gentry 2009]. In this work, we focus on the so-called 3rd generation FHE, which includes schemes such as FHEW [Ducas and Micciancio 2015], TFHE [Chillotti et al. 2020], FHEZ [Pereira 2021], and FINAL [Bonte et al. 2022]. Those schemes offer the following programming interface: at setup time, when generating the parameters and the cryptographic keys, one can set the precision k, i.e., the number of bits of the message space. Then, one can encrypt messages $m \in \{0, 1\}^t$, generating $c = \mathsf{Enc}(m)$. Given a ciphertext c, it is possible to execute any function $f : \{0, 1\}^t \to \{0, 1\}^t$, yielding $\mathsf{Enc}(f(m))$. This operations is called programmable bootstrapping (PBS) and f is called a t-LUT (look-up-table), or simply LUT. In the particular case where $t = 1$, instead of having LUTs, each bootstrapping can perform one binary gate (e.g., AND, OR, and XOR gates).

Hence, to evaluate an arbitrary function $F(m_1, ..., m_\ell)$, one has to represent F as a sequence of t-LUTs, possibly with some of them simply evaluating binary gates, then perform each t-LUT via PBS. Since PBS is slow, one generally wants to minimize the number of PBS executions in a homomorphic evaluation.

[1]`https://docs.zama.ai/tfhe-rs`
[2]`https://github.com/gabijacob/ETC`

Via a technique called multi-value bootstrapping (MVPBS), it is possible to execute one single PBS on $\mathsf{Enc}(m)$ and extract many LUTs, obtaining $\mathsf{Enc}(f_1(m)), \mathsf{Enc}(f_2(m)), ..., \mathsf{Enc}(f_\ell(m))$, almost for the same cost as obtaining a single LUT [Carpov et al. 2019].

Notice that the cost of homomorphically evaluating a function is roughly the number of bootstrappings multiplied by the running time of each bootstrapping, since all the other operations have negligible running time compared to the bootstrapping. Also, once you fix the precision t of the message space, all your bootstrappings cost the same, regardless of the LUT $f : \{0,1\}^t \to \{0,1\}^t$ that they evaluate.

Chacha20: Chacha20 [Bernstein 2008, Nir and Langley 2018], or simply Chacha, is a stream cipher based on three functions: 32-bit addition, XOR and rotation. Chacha offers 256 bits of security and operates on a state composed of 512 bits into 16 variables of 32 bits each. It performs 20 rounds updating the state with a function called *quarter round*, QR(a, b, c, d), which is composed of the following operations, where addition is modulo 2^{32}, $\oplus$ denotes bitwise XOR, and $\lll$ is a circular shift:

$$\begin{array}{lll} a = (a+b); & d = (d \oplus a); & d = (d \lll 16); \\ c = (c+d); & b = (b \oplus c)\ ; & b = (b \lll 12); \\ a = (a+b); & d = (d \oplus a)\ ; & d = (d \lll 8); \\ c = (c+d); & b = (b \oplus c)\ ; & d = (b \lll 7); \end{array}$$

After the last round, the mixed state is added to the initial state to obtain the keystream block, which is used to encrypt and decrypt messages by XOR'ing with the plaintext or ciphertext. Therefore, evaluating Chacha homomorphically boils down to efficiently evaluating the QR function, which requires a 32-bit full-adder. For this, we considered Ripple Carry Adder (RCA), which, given 32-bit integers $x = (y_0, ..., x_{31})$ and $y = (y_0, ..., y_{31})$, calculates $\mathsf{sum} = x + y \bmod 2^{32}$ as follows: define $\mathsf{carry}_0 = 0$, then, compute for $0 \leq i \leq 31$:

$$\mathsf{sum}_i = x_i \oplus y_i \oplus \mathsf{carry}_i \ \text{ and } \ \mathsf{carry}_{i+1} = (x_i \wedge y_i) \vee (\mathsf{carry}_i \wedge (x_i \oplus y_i)) \qquad (1)$$

3. Our results

We started with the most natural implementation, that is, since Chacha operations are defined in bitwise level (e.g., bitwise XOR), we first used FHE with a binary message space and only evaluating binary gates homomorphically to implement RCA and XOR gates. The rotations are free, as they only require us to rename variables. We performed a theoretical analysis of this version, counting the number of bootstrappings it requires and we implemented in Rust using the TFHE-rs library. Therefore, our first contribution is to provide a baseline for Chacha-based transciphering.

We further proceeded to study other ways of implementing Chacha assuming that we have larger message spaces, say $\{0,1\}^t$ instead of simply $\{0,1\}$, so that we can also execute t-LUTs instead of only binary gates. Thus, our second contribution is proposing and analysing homomorphic versions of Chacha using 2- and 3-LUTs.

3.1. Bitwise Implementation

Our baseline version uses $\{0,1\}$ as the message space and only boolean gates are evaluated homomorphically (requiring one bootstrapping per gate). For the homomorphic rotation, the input vector is rotated by simply rotating its positions. Thus, this function does not need any bootstrapping and its cost is negligible for the total execution time.

For the homomorphic XOR, we iterate through both input vectors, calculating XOR bitwise and pushing the result to an output vector. This results in 32 bootstrappings. Subsequently, the most expensive building-block of Chacha is the 32-bit adder. The full-adder implemented for this purpose was Ripple Carry Adder (RCA), which iterates through pairs of input bits computing the output bit (the actual sum) and the carry bit. As shown in Equation (1), it takes 5 bootstrappings for each pair of bits, hence requiring a total of $5 \cdot 32 = 160$ bootstrappings.

With those "low-level" building blocks implemented, we can implement the Quarter Round (QR), the fundamental function of Chacha, which essentially calls the above-mentioned functions 4 times each, as described in Section 2. Thus, each execution of the QR requires $(32+160) \cdot 4 = 768$ bootstrappings. Then, Chacha performs 20 rounds, each one calling QR 4 times, giving us a total of $768 \cdot 80 = 61440$ bootstrappings.

Finally, as a last step, Chacha adds the secret key to the state, which takes another $512 \cdot 5 = 2560$ bootstrappings. Therefore, in total, Chacha implementation with binary messages requires $61440 + 2560 = 64000$ bootstrappings.

3.2. (Scaled) XOR for free

Our first major optimization consists in using the implementation details of FHE schemes to obtain XOR for free in our homomorphic Chacha when our message space consists in more than one bit (therefore, this optimization does not apply to our baseline implementation). That is, instead of using the FHE scheme as a black box, we use its internal representation of messages to evaluate XOR gates without executing bootstrappings.

Since we know that in LWE-based 3rd-generation FHE schemes a message $m_i \in \{0,1\}^t$ is encrypted under a key $\mathbf{s} \in \mathbb{Z}_q^n$ as $(\mathbf{a}_i, b_i) \in \mathbb{Z}_q^{n+1}$ where $b_i = \mathbf{a}_i \cdot \mathbf{s} + e_i + \Delta \cdot m_i \bmod q$, $\Delta = q/2^{t+1}$, and e_i is some small noise term, we can compute XOR as follows: Given two ciphertexts $(\mathbf{a}_1, b_1)$ and $(\mathbf{a}_2, b_2)$ encrypting bits $m_1, m_2 \in \{0,1\}$, we output $(\mathbf{a}_3, b_3) := (2^t \cdot (\mathbf{a}_1 + \mathbf{a}_2,), \, 2^t \cdot (b_1 + b_2))$.

One can see that $b_3 = \mathbf{a}_3 \cdot \mathbf{s} + e_3 + (q/2^{t+1}) \cdot m_3$, where $e_3 = 2^t \cdot (e_1 + e_2)$ and $m_3 = 2^t \cdot (m_1 + m_2)$. Since t is always a very small constant, typically $2 \leq t \leq 8$, it follows that $(\mathbf{a}_3, b_3)$ is a valid LWE encryption of $m_3 \bmod 2^{t+1}$ with small noise e_3.

Notice that if $m_1 = m_2 = 0$, then clearly $(\mathbf{a}_3, b_3)$ encrypts 0. Also, if $m_1 = m_2 = 1$, then $\Delta \cdot m_3 = (q/2^{t+1}) \cdot 2^{t+1} \equiv 0 \bmod q$, thus, $(\mathbf{a}_3, b_3)$ also encrypts 0. On the other hand, if m_1 and m_2 are different, then $\Delta \cdot m_3 = \Delta \cdot 2^t \cdot 1$, but $1 = \texttt{xor}(m_1, m_2)$. Therefore, in all of the three cases, we conclude that $(\mathbf{a}_3, b_3)$ encrypts $2^t \cdot \texttt{xor}(m_1, m_2)$.

Notice that computing $(\mathbf{a}_3, b_3)$ only requires $n+1$ additions modulo a small q, typically, $q = 2^{32}$ or $q = 2^{64}$, in other words, those are simple integer additions implemented natively by common processors. Therefore, this scaled XOR is essentially for free when compared to the $\Theta(n)$ homomorphic multiplications required by the PBS.

In summary, we have defined a computationally cheap operation mapping two encrypted bits m_1 and m_2 to $\mathsf{Enc}(2^t \cdot \texttt{xor}(m_1, m_2))$. Notice that this operation is not composable, since the input is supposed to be bits while the output can be 2^t.

3.3. Homomorphic Chacha with 2-LUTs

For a message space of 2 bits, we can exploit rotation and scaled XOR for free. It remains to efficiently implement the homomorphic 32-bit adder, which takes integers x and y encrypted bitwise and generates $z = x + y \bmod 2^{32}$, also encrypted bitwise. Thus, consider we have LWE ciphertexts encrypting each bit x_i and y_i of x and y, for $0 \leq i \leq 31$. Define $\mathsf{carry}_0 = 0$ and $\mathbf{c}_0^{\mathsf{carry}} = \mathsf{Enc}(\mathsf{carry}_0)$, then, proceed as follows computing encryptions of each z_i and carry_i, for $0 \leq i \leq 31$.

1. compute $\mathbf{c}^{0||y_i} := 2^t \cdot \mathsf{Enc}(y_i)$
2. compute $\mathbf{c}^{x_i||y_i} := \mathbf{c}^{0||y_i} + \mathsf{Enc}(x_i)$. One can see that $\mathbf{c}^{x_i||y_i} = \mathsf{Enc}(x_i + 2^t y_i)$
3. define functions $f_0(x) := \mathsf{lsb}(x) \oplus \mathsf{msb}(x)$ and $f_1(x) := \mathsf{lsb}(x) \wedge \mathsf{msb}(x)$
4. with one MVPBS, apply f_0 and f_1 to $\mathbf{c}^{x_i||y_i}$, obtaining $\mathbf{c}^{\oplus} = \mathsf{Enc}(f_0(x_i + 2^t y_i)) = \mathsf{Enc}(x_i \oplus y_i)$ and $\mathbf{c}^{\mathsf{and}} = \mathsf{Enc}(f_1(x_i + 2^t y_i)) = \mathsf{Enc}(x_i \wedge y_i)$
5. compute $\mathbf{c}^{\mathsf{carry}||\mathsf{xor}} = 2 \cdot \mathbf{c}^{\oplus} + \mathbf{c}_i^{\mathsf{carry}}$. Notice that $\mathbf{c}^{\mathsf{carry}||\mathsf{xor}} = \mathsf{Enc}(\mathsf{carry}_i + 2(x_i \oplus y_i))$
6. with one single MVPBS, apply f_0 and f_1 to $\mathbf{c}^{\mathsf{carry}||\mathsf{xor}}$, generating $\mathbf{c}_i^{sum} := \mathsf{Enc}((x_i \oplus y_i) \oplus \mathsf{carry}_i)$ and $\mathbf{c}_i^{tmp} := \mathsf{Enc}((x_i \oplus y_i) \wedge \mathsf{carry}_i)$
7. with one PBS compute the OR gate, yielding

$$\mathbf{c}_{i+1}^{\mathsf{carry}} = \mathsf{FHE.OR}(\mathbf{c}^{\mathsf{and}}, \mathbf{c}_i^{tmp}) = \mathsf{Enc}((x_i \wedge y_i) \vee ((x_i \oplus y_i) \wedge \mathsf{carry}_i))$$

Hence, we obtained encryptions of the i-th bit of the sum, $\mathbf{c}_i^{sum} = \mathsf{Enc}(z_i)$, and of the next carry, $\mathbf{c}_{i+1}^{\mathsf{carry}}$, using 3 bootstrappings. The last iteration needs one less bootstrapping, since we do not need to compute the very last carry, thus, the total amount of bootstrappings is $31 \cdot 3 + 2 = 95$.

Finally, notice that when we apply the (scaled) XOR for free, we obtain $\mathsf{Enc}(2^t y_i)$, thus, to perform the RCA as described above, we just skip step 1.

By observing the subsequent lines of QR(a, b, c, d), we can see that it composes bitwise XOR with addition (ignoring the rotation, since it is simply a renaming of variables in our case). For the first and second lines, we evaluate the RCA homomorphically and the XOR for free. This costs $2 \cdot 95 = 190$ bootstrappings. As for the third and fourth lines, we can evaluate RCA the same way, but we cannot use the XOR for free, since it is not composable, as explained in Section 3.2. Thus, the XOR gates have to be performed with bootstrappings. Therefore, these lines cost $2 \cdot 95 + 2 \cdot 32 = 254$ bootstrappings.

As a result, to evaluate one time the QR function, it takes $190 + 254 = 444$ bootstrappings. In the end of the algorithm we have to add the initial state to the final state, calling the homomorphic add function 16 more times, which costs $16 \cdot 95 = 1520$ additional bootstrappings. Therefore, the total number of bootstrappings when evaluating Chacha completely is $444 \cdot 80 + 1520 = 37040$.

3.4. Homomorphic Chacha with 3-LUTs

The actual turning point on the algorithm efficiency is using a message space of 3 bits and MVPBS. In this case, it is possible to calculate the sum using only one bootstrapping per pair of bits instead of 3.

As in Section 3.3, consider we have $\mathbf{c}_0^{\mathsf{carry}} = \mathsf{Enc}(0)$ and encryptions of bits x_i and y_i for $0 \leq i \leq 31$. Then, to evaluate RCA, we proceed as follows:

1. compute $\mathbf{c}^{0||0||y_i} := 2^t \cdot \mathsf{Enc}(y_i)$

2. compute $\mathbf{c}^{x_i||\mathsf{carry}_i||y_i} := \mathbf{c}^{0||0||y_i} + \mathsf{Enc}(x_i) + 2 \cdot \mathbf{c}_i^{\mathsf{carry}}$. Notice that $\mathbf{c}^{x_i||\mathsf{carry}_i||y_i} = \mathsf{Enc}(x_i + 2\mathsf{carry}_i + 4y_i)$
3. define $g_i(u)$ as the i-th bit of u, $f_0(u) = g_0(u) \oplus g_1(u) \oplus g_2(u)$, and $f_1(u) = (g_2(u) \wedge g_0(u)) \vee (g_1(u) \wedge (g_0(u) \oplus g_2(u)))$. Notice that f_0 and f_1 compute the sum and carry bits given by Equation (1)
4. with one single MVPBS, apply f_0 and f_1 to $\mathbf{c}^{x_i||\mathsf{carry}_i||y_i}$ obtaining $\mathbf{c}_i^{sum} := \mathsf{Enc}(x_i \oplus \mathsf{carry}_i \oplus y_i)$ and $\mathbf{c}_{i+1}^{\mathsf{carry}} := \mathsf{Enc}((x_i \wedge y_i) \vee (\mathsf{carry}_i \wedge (x_i \oplus y_i)))$

Thus, by using the message space $\{0, 1\}^3$, RCA can be evaluated homomorphically with one bootstrapping per iteration, meaning 32 bootstrappings in total. To evaluate the QR, we need to run RCA 4 times, one per line, giving us 128 bootstrappings. We can use XOR for free in the first 2 lines, but we need bootstrappingss to evaluate the XOR gates in the 2 last lines, to avoid problems with composability (just as in 2-LUT), which takes 64 bootstrappings. Hence, we have 192 bootstrappings per execution of QR.

Lastly, we need to add the final state to the initial state to finalize the encryption process, which takes 512 bootstrappings, one for each bit. Thus, the total amount of bootstrappings to evaluate Chacha with a message space of 3 bits is $192 \cdot 80 + 512 = 15872$.

3.5. Homomorphic Chacha with 4-LUTs

Analyzing a message space of 4 bits using the same logic leads us to the conclusion that the number of bootstrappings will not decrease. Since RCA computes the output bits in a serial manner, we would still need at least one bootstrappings per bit. Hence, even if the message space is larger than 3 bits, the number of bootstrappings is the same.

4. Comparison

We compare our results to the state-of-the-art transciphering techniques based on standard ciphers, that is, that do not use ad hoc ciphers created on purpose to be FHE-friendly. More specifically, we consider the fastest transciphering using AES [Trama et al. 2023].

Since our work only considered serial implementation, we do not compare with parallel implementations presented in [Trama et al. 2023]. Thus, our main comparison metric is the total number of bootstrappings. It is important to notice that, as one increases the message space to accommodate more bits and to be able to evaluate larger t-LUTs, bootstrapping becomes exponentially more expensive, therefore, it is only possible to handle very small values of t. Given the number of bootstrappings and the value of t, we can estimate the running time of each bootstrapping, then estimate the total execution time, which is the latency. Our second comparison metric is the throughput, which is the number of bits produced per second. To compute this, we divide the latency by the number of bits that are processed at once, which is 128 for AES and 512 for Chacha. Of course, low latency and high throughput are preferable.

We ran our Rust implementation of the bitwise version on an Apple M2 3.49GHz machine with 8 GB of RAM running on a macOS Sequoia 15.3.2 (24D81). Each bootstrapping evaluating one single boolean gate took an average of 0.011 s and the total execution of homomorphic Chacha was 758.29 s, which is close to the expected time, namely, the number of required bootstrappings times the time per bootstrapping, confirming that the bootstrappings dominate the execution time.

To estimate the execution time of the non-binary versions, we used the unitary times for bootstrapping reported in [Trama et al. 2023]: 0.007 s for 2-LUT and 0.015 s for 3-LUT. Then we multiplied these times by the number bootstrappings that each of our versions of homomorphic Chacha uses. In summary, 3-LUT gives us the best latency and throughput while 4-LUT is the best for [Trama et al. 2023]. Considering this, our best latency in worse than theirs by a factor of $238.08/122.5 \approx 1.94$, while our throughput is better than theirs by about $2.15/1.045 \approx 2.06$ times. We present the comparison in more detail in Table 1.

Ultimately, it is important to acknowledge that, even though we made a direct comparison, we studied a cipher that has 256-bit security level, while [Trama et al. 2023] used AES with 128 bits of security. This is a significant difference, since AES-256 has 14 rounds, while AES-128 has only 10. Also, it is important to notice that in a post-quantum scenario, AES-128 is insecure and one has to use AES-256 to achieve 128 bits of security. Considering this, it would be interesting to extend the results of [Trama et al. 2023] to understand the exact overhead of evaluating AES-256 homomorphically, then comparing again with our results. Since evaluating more rounds of AES will obviously be more expensive, our throughput will be more than $2\times$ better than theirs.

Message space	**bootstrappings**	**Latency**	**Throughput**
AES			
8-LUT	1664	2496.00 s	0.051
4-LUT	4224	122.50 s	1.045
3-LUT	17088	256.32 s	0.499
2-LUT	44288	310.02 s	0.412
Chacha			
3-LUT	15872	238.08 s	2.15
2-LUT	37040	259.28 s	1.97
bitwise	64000	704.00 s	0.727

Table 1. Total number of bootstrappings, estimated execution times (latency) and number of bits per second (throughput) for different message spaces

5. Conclusion and Future Work

The implementation already done in this research was focused on a message space of a single bit. The main focus of the future work is towards implementing the cipher using a message space of 3 bits, which is, theoretically, the most optimized version of homomorphic Chacha. Furthermore, the analysis on this research was done only considering a sequential execution. Thus, as future work, we can study strategies to evaluate parts of Chacha in parallel. For example, we can consider other full-adder algorithms that are more parallelizable, such as Carry Lookahead Adder.

Acknowledgments

We thank the anonymous reviewers for the valuable comments and suggestions. This work was supported by the São Paulo Research Foundation (FAPESP), Brasil, process Number 2023/12755-8, and by Teaching, Research and Extension Support Fund (FAEPEX) under the Incentive Program for New Professors (PIND), process number 3389/23.

References

Bernstein, D. J. (2008). Chacha, a variant of salsa20. In *Workshop Record of SASC*, pages 3–5. Citeseer.

Bonte, C., Iliashenko, I., Park, J., Pereira, H. V. L., and Smart, N. P. (2022). FINAL: Faster FHE instantiated with NTRU and LWE. In *Advances in Cryptology – ASIACRYPT 2022*, Cham. Springer International Publishing.

Carpov, S., Izabachène, M., and Mollimard, V. (2019). New techniques for multi-value input homomorphic evaluation and applications. In Matsui, M., editor, *Topics in Cryptology – CT-RSA 2019*, pages 106–126, Cham. Springer International Publishing.

Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (2020). TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology*, 33(1):34–91.

Cosseron, O., Hoffmann, C., Méaux, P., and Standaert, F.-X. (2022). Towards case-optimized hybrid homomorphic encryption. In Agrawal, S. and Lin, D., editors, *Advances in Cryptology – ASIACRYPT 2022*, pages 32–67, Cham. Springer Nature Switzerland.

Dobraunig, C., Grassi, L., Helminger, L., Rechberger, C., Schofnegger, M., and Walch, R. (2023). Pasta: A case for hybrid homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):30–73.

Ducas, L. and Micciancio, D. (2015). Fhew: Bootstrapping homomorphic encryption in less than a second. In Oswald, E. and Fischlin, M., editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gentry, C. (2009). *A fully homomorphic encryption scheme*. PhD thesis, Stanford University. `crypto.stanford.edu/craig`.

Lyubashevsky, V., Peikert, C., and Regev, O. (2010). On ideal lattices and learning with errors over rings. In Gilbert, H., editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg. Springer Berlin Heidelberg.

Nir, Y. and Langley, A. (2018). ChaCha20 and Poly1305 for IETF Protocols. RFC 8439.

Pereira, H. V. L. (2021). Bootstrapping fully homomorphic encryption over the integers in less than one second. In *IACR International Conference on Public-Key Cryptography*, pages 331–359. Springer.

Thakur, I., Karmakar, A., Li, C., and Preneel, B. (2025). A survey on transciphering and symmetric ciphers for homomorphic encryption. Cryptology ePrint Archive, Paper 2025/093.

Trama, D., Clet, P.-E., Boudguiga, A., and Sirdey, R. (2023). A Homomorphic AES Evaluation in Less than 30 Seconds by Means of TFHE. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '23, page 79–90, New York, NY, USA. Association for Computing Machinery.