International Journal of Scientific Research in Science, Engineering and Technology

Print ISSN - 2395-1990
Online ISSN : 2394-4099

OPEN ACCESS

Available Online at : www.ijsrset.com
doi : https://doi.org/10.32628/IJSRSET2310556

Computational Group Theory and Quantum-Era Cryptography

Michael N. John[1], Udoaka O. G[2]

[1, 2]Department of Mathematics, Akwa Ibom State University, Nigeria

ARTICLEINFO

Article History :

Accepted: 05 Oct 2023
Published: 03 Nov 2023

Publication Issue :

Volume 10, Issue 6
November-December-2023

Page Number :

01-10

ABSTRACT

This paper provides an overview of the significant role of computational group theory in cryptography. Group theory plays a crucial role in various cryptographic applications, such as key exchange, encryption, and digital signatures. This paper examines the fundamental concepts, algorithms, and applications of computational group theory in cryptography, using polycyclic groups with a focus on key findings and recent developments in quantum-era cryptography.

Keywords : Computational Group theory, Lattice, Cryptography, Quantum computers, Encryption, Decryption, Polycyclic group

I. INTRODUCTION

With the advent of quantum computers, traditional cryptographic algorithms face a significant threat as they can potentially break them through their immense computational power. As quantum computers continue to advance, there is a growing need to develop new cryptographic algorithms that can withstand the quantum computing power.

Cryptography is essential for securing digital communication and data. Over the years, computational group theory has become a key tool in the development and analysis of cryptographic protocols. [13] presented a foundational paper, presenting a group-theoretic framework for constructing cryptographic primitives, leading to the development of zero-knowledge proofs and other advanced cryptographic techniques. [12] introduced the idea of constructing cryptographic systems based on the hardness of certain group problems and lattice reduction, demonstrating the connection between computational group theory and lattice-based cryptography.

Group theory provides a framework for understanding the mathematical structures that underlie cryptography. On this paper we explore polycyclic groups to see its efficacy in quantum-era cryptography. [11] provided a survey of pairing-based cryptography, which heavily relies on group theory, and explores its applications in identity-based cryptography and other areas.

Polycyclic groups offer a promising solution for quantum-era cryptography as they possess certain properties that make them resistant to attacks by quantum computers. These properties include the ability to provide post-quantum security, resistance against Shor's algorithm, and the potential for efficient implementation. By utilizing polycyclic groups in quantum-era cryptography, we can enhance the security of our cryptographic systems and protect sensitive information from being compromised by quantum computers.

As quantum computing threatens the security of classical cryptographic schemes, [14] has studied cryptography within the realm of post-quantum cryptography. Computational group theory on the other hand has played a role in the development of post-quantum cryptographic algorithms that are resistant to quantum attacks [8]

II. BASICS OF COMPUTATIONAL GROUP THEORY

2.1. Group Theory Fundamentals

Group theory provides a framework for understanding the mathematical structures that underlie cryptography. It involves concepts such as groups, subgroups, generators, and group isomorphisms. see [17]. A thorough understanding of these basics is crucial for applying group theory in cryptography can be seen in [1].

2.2. Computational Aspects

Group theory computations, such as group membership tests, subgroup enumeration, and discrete logarithm problem solutions, are at the heart of many cryptographic protocols. These computations are vital in ensuring the security of cryptographic schemes, and [2] has worked extensively on it.

III. APPLICATIONS OF COMPUTATIONAL GROUP THEORY IN CRYPTOGRAPHY

3.1. Public Key Cryptography

Public key cryptography schemes like Diffie-Hellman and RSA rely on group structures for key exchange and encryption. The use of computational group theory in analyzing and designing secure public key systems is well-documented in [3]. Pollard's Rho algorithm is a widely-used method for solving the discrete logarithm problem in various group structures. Its impact on the security analysis of cryptographic systems is noteworthy in [6]. Homomorphic encryption schemes often rely on mathematical structures related to group theory to perform computations on encrypted data. Research in this area continues to expand the practicality of privacy-preserving computations [9].

3.2. Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) utilizes the group structure of elliptic curves. Computational group theory is integral in ECC for discrete logarithm problems on elliptic curves, making ECC a popular choice for secure cryptography [4]. Schoof's algorithm is essential for counting points on elliptic curves over finite fields, which is crucial for determining the security of elliptic curve cryptography. Its application has made ECC more practical and secure [7].

3.3. Lattice-based Cryptography

Lattice-based cryptography involves the use of algebraic structures related to group theory for the construction of secure encryption and digital signature schemes. [10] has worked on cube-lattice-based cryptography. Lattice problems are notoriously hard, and computational group theory plays a role in their analysis [5].

IV. FUNDAMENTAL DEFINATIONS AND CONCEPT

In mathematics, a polycyclic group is a solvable group that satisfies the maximal condition on subgroups (that is, every subgroup is finitely generated). Polycyclic groups are finitely presented, which makes them interesting from a computational point of view.

Equivalently, a group G is polycyclic if and only if it admits a subnormal series with cyclic factors, that is a finite set of subgroups, let's say G_0, ..., G_n such that

- G_n coincides with G
- G_0 is the trivial subgroup
- G_i is a normal subgroup of G_{i+1} (for every i between 0 and n - 1)
- and the quotient group G_{i+1} / G_i is a cyclic group (for every i between 0 and n - 1)

That is;

A series of a group G is a chain of subgroups $\{1\} = G_0 \leq G_1 \leq \cdots \leq G_{n-1} = G$ such that G_i is normal in G_{n+1}

A group G is said to be polycyclic if it has a normal subnormal series $\{1\} = G_0 \leq G_1 \leq \cdots \leq G_{n-1} \leq G_n = G$ such that the quotient group G_{i+1}/G_i are cycle.

Every polycyclic group can be described by a polycyclic presentation of the following form;

$$\ll g_i, \ldots, g_n | g_i^{-1} g_j g_i = u_{ij} \text{ for } 1 \leq i < j \leq n$$

$$g_i g_j g_i^{-1} = v_{ij} \text{ for } 1 \leq i < j \leq n$$

$$g_i^{r_i} = w_{ii} \text{ for } i \in 1 \gg$$

Where u_{ij}, v_{ij}, w_{ii} are words in the generators $g_{i+1}, \ldots, g_n$ and I is a set of indices $i \in \{1, \ldots, n\}$ such that$[G_{i+1} : G_i]$

4.1 PYTHON COMPUTATIONAL ATTRIBUTES OF POLYCYCLIC GROUP

- pc_sequence : Polycyclic sequence is formed by collecting all the missing generators between the adjacent groups in the derived series of given permutation group.
- pc_series : Polycyclic series is formed by adding all the missing generators of der[i+1] in der[i], where der represents derived series.
- relative_order : A list, computed by the ratio of adjacent groups in pc_series.
- collector : By default, it is None. Collector class provides the polycyclic presentation.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> len(PcGroup.pcgs)
4
>>> pc_series = PcGroup.pc_series
>>> pc_series[0].equals(G)  # use equals, not literal `==`
True
>>> gen = pc_series[len(pc_series) - 1].generators[0]
>>> gen.is_identity
True
>>> PcGroup.relative_order
[2, 3, 2, 2]
```

See [15] for a handbook on computational group theory

4.2 PYTHON COMPUTATION OF POLYCYCLIC PRESENTATION

The computation normally starts from the bottom of the pcgs and polycyclic series. Storing all the previous generators from pcgs and then taking the last generator as the generator which acts as a conjugator and conjugates all the previous generators in the list.

To get a clear picture, start with an example of SymmetricGroup(4). For S(4) there are 4 generators in pcgs say $[x_0, x_1, x_2, x_3]$ and the relative_order vector is [2, 3, 2, 2]. Starting from bottom of this sequence the presentation is computed in order as below.

using only $[x_3]$ from pcgs and pc_series(4) compute:

- x_3^2

using only $[x_3]$ from pcgs and pc_series(3) compute:

- x_2^2
- $x_2^{-1}x_3x_2$

using $[x_3, x_2]$ from pcgs and pc_series(2) compute:

- x_1^3
- $x_1^{-1}x_3x_1$
- $x_1^{-1}x_2x_1$

using $[x_3, x_2, x_1]$ from pcgs and pc_series(1) compute:

- x_1^2
- $x_0^{-1}x_3x_0$
- $x_1^{-1}x_2x_0$
- $x_0^{-1}x_1x_0$

Note that same group can have different pcgs due to variying derived_series which will results in different polycyclic presentations.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup

>>> from sympy.combinatorics.permutations import Permutation

>>> G = SymmetricGroup(4)

>>> PcGroup = G.polycyclic_group()

>>> collector = PcGroup.collector

>>> pcgs = PcGroup.pcgs

>>> len(pcgs)

4
```

```
>>> free_group = collector.free_group
>>> pc_resentation = collector.pc_presentation
>>> free_to_perm = {}
>>> for s, g in zip(free_group.symbols, pcgs):
...     free_to_perm[s] = g
>>> for k, v in pc_resentation.items():
...     k_array = k.array_form
...     if v != ():
...         v_array = v.array_form
...     lhs = Permutation()
...     for gen in k_array:
...         s = gen[0]
...         e = gen[1]
...         lhs = lhs*free_to_perm[s]**e
...     if v == ():
...         assert lhs.is_identity
...         continue
...     rhs = Permutation()
...     for gen in v_array:
...         s = gen[0]
...         e = gen[1]
...         rhs = rhs*free_to_perm[s]**e
...     assert lhs == rhs
```

4.3 COMPUTATION OF EXPONENT VECTOR

Any generator of the polycyclic group can be represented with the help of its polycyclic generating sequence. Hence, the length of exponent vector is equal to the length of the pcgs.

A given generator g of the polycyclic group, can be represented as $g = x_i^{e_i} \dots x_n^{e_n}$ *where* x_i represents polycyclic generators and n is the number of generators in the free_group equal to the length of pcgs.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> G = SymmetricGroup(4)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> pcgs = PcGroup.pcgs
>>> collector.exponent_vector(G[0])
[1, 0, 0, 0]
>>> exp = collector.exponent_vector(G[1])
>>> g = Permutation()
>>> for i in range(len(exp)):
...     g = g*pcgs[i]**exp[i] if exp[i] else g
>>> assert g == G[1]
```

4.4 COMPUTATIONAL DEPTH OF POLYCYCLIC GENERATOR

Depth of a given polycyclic generator is defined as the index of the first non-zero entry in the exponent vector.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(3)
>>> PcGroup = G.polycyclic_group()
>>> collector = PcGroup.collector
>>> collector.depth(G[0])
```

```
2

>>> collector.depth(G[1])

1
```

4.5 COMPUTATION OF LEADING EXPONENT

Leading exponent represents the exponent of polycyclic generator at the above depth.

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup

>>> G = SymmetricGroup(3)

>>> PcGroup = G.polycyclic_group()

>>> collector = PcGroup.collector

>>> collector.leading_exponent(G[1])

1
```

V. MAIN RESULT

Let's explore a simplified example of how you can use Python to implement encryption and decryption using a hypothetical cryptographic scheme based on algebraic structures inspired by polycyclic groups.
Please note that this is a highly simplified and abstracted example to illustrate the concept. In practice, encryption algorithms are far more complex and involve many additional considerations for security.

5.1 Python Codes

```
import random
# Simulated polycyclic group operations (hypothetical)
def polycyclic_multiply(a, b):
    return a * b
def polycyclic_inverse(a):
    return 1 / a
# Encryption function
def encrypt(message, public_key):
    ciphertext = []
    for character in message:
        # Convert the character to a numerical representation (e.g., ASCII)
```

```
        numeric_value = ord(character)
        # Apply a simulated polycyclic group operation on the numeric value
        encrypted_value = polycyclic_multiply(numeric_value, public_key)
        ciphertext.append(encrypted_value)
    return ciphertext
# Decryption function
def decrypt(ciphertext, private_key):
    decrypted_message = ''
    for encrypted_value in ciphertext:
        # Apply the inverse operation to decrypt
        decrypted_value = polycyclic_inverse(encrypted_value)
        # Convert the numerical value back to a character
        decrypted_character = chr(int(decrypted_value))
        decrypted_message += decrypted_character
    return decrypted_message
# Generate public and private keys (random values for demonstration)
public_key = random.randint(1, 100)
private_key = polycyclic_inverse(public_key)
# Message to be encrypted
message = "HELLO"
# Encrypt the message
encrypted_message = encrypt(message, public_key)
print("Encrypted Message:", encrypted_message)
# Decrypt the message
decrypted_message = decrypt(encrypted_message, private_key)
print("Decrypted Message:", decrypted_message)
```

5.2 DETAILED EXPLANATION OF THE CODE

- polycyclic_multiply and polycyclic_inverse are hypothetical functions representing operations within a polycyclic-like group. In a real-world scenario, these operations would be much more complex.
- The encrypt function takes a message, converts each character to a numerical value, applies a simulated polycyclic group operation (multiplication), and stores the encrypted values in a list (ciphertext).

- The decrypt function reverses the encryption process by applying the inverse operation (division) to each value in the ciphertext, converting the numerical values back to characters, and reconstructing the decrypted message.
- Random public and private keys are generated for demonstration purposes. In a real cryptographic system, these keys would be generated using secure processes
- The message "HELLO" is encrypted and then decrypted to verify the correctness of the implementation.
- Please note that this is a highly simplified example for educational purposes and doesn't represent a secure or practical encryption scheme. In real-world scenarios, cryptographic algorithms are much more complex and undergo rigorous analysis and testing for security.

VI.CONCLUSION

Computational group theory is a foundational tool in modern cryptography. Its role in developing, analyzing, and securing cryptographic schemes cannot be understated. This paper has provided an overview of its fundamental concepts, applications, and recent developments, highlighting the critical role it plays in ensuring the confidentiality, integrity, and authenticity of digital communication.

VII. REFERENCES

[1]. Fraleigh, J. B. (2003). A First Course in Abstract Algebra. Pearson.

[2]. Shoup, V. (1996). Lower bounds for discrete logarithms and related problems. Advances in Cryptology - CRYPTO '96.

[3]. Diffie, W., & Hellman, M. E. (1976). New Directions in Cryptography. IEEE Transactions on Information Theory.

[4]. Koblitz, N. (1987). Elliptic Curve Cryptosystems. Mathematics of Computation.

[5]. Peikert, C. (2016). Lattice Cryptography for the Internet. arXiv:1612.00988.

[6]. Pollard, J. M. (1978). Monte Carlo Methods for Index Computation (Mod p). Mathematics of Computation.

[7]. Schoof, R. (1995). Counting Points on Elliptic Curves over Finite Fields. Journal of the London Mathematical Society.

[8]. Bernstein, D. J., Lange, T., & Schwabe, P. (2016). Post-Quantum Cryptography. Springer.

[9]. Brakerski, Z., & Vaikuntanathan, V. (2014). Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. Advances in Cryptology - CRYPTO '11.

[10]. Michael N. John & Udoaka O. G (2023). Algorithm and Cube-Lattice-Based Cryptography. International journal of Research Publication and reviews, Vol 4, no 10, pp 3312-3315 October 2023.

[11]. Groth, J., Sahai, A., & Waters, B. (2008). Pairing-Based Cryptography: A Survey. Advances in Cryptology - EUROCRYPT '08, 2-16.

[12]. Ajtai, M. (1996). Public-Key Cryptosystems from Lattice Reduction Problems. STOC '96, 99-108.

[13]. Goldwasser, S., & Rackoff, C. (1989). A Group Theoretic Framework for Cryptographic Applications. Advances in Cryptology - CRYPTO '85, 368-383.

[14]. Alagiannis, I., et al. "Lattice-based cryptography as a Post-Quantum candidate." Journal of Cryptographic Engineering (2014)

[15]. Derek F. Holt, Handbook of Computational Group theory. In the series 'Discrete Mathematics and its Application',Chapman & Hall/CRC 2005, xvi + 514 p.

[16]. Udoaka O. G. & Frank E. A. (2022). Finite Semi-group Modulo and Its Application to Symmetric Cryptography, International Journal of Pure Mathematics DOI: 10.46300/91019.2022.9.13.

[17]. Udoaka, O. G. (2022). Generators and inner automorphism. THE COLLOQUIUM -A Multi-disciplinary Thematc Policy Journal www.ccsonlinejournals.com. Volume 10, Number 1 , Pages 102 -111 CC-BY-NC-SA 4.0 International Print ISSN : 2971-6624 eISSN: 2971-6632

Cite this article as :

Michael N. John, Udoaka O. G., "Computational Group Theory and Quantum-Era Cryptography", International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Online ISSN : 2394-4099, Print ISSN : 2395-1990, Volume 10 Issue 6, pp. 01-10, November-December 2023. Available at doi : https://doi.org/10.32628/IJSRSET2310556
Journal URL : https://ijsrset.com/IJSRSET2310556