Class-Group Additively Homomorphic Encryption (CG-AHE) Rust Implementation Audit

DWLX-02

Prepared for dWallet Labs

Dr. Nadim Kobeissi
Symbolic Software

May 14, 2025

Abstract

This report presents the findings of a comprehensive cryptographic audit of dWallet Labs' Class-Group based Additive Homomorphic Encryption (CG-AHE) library implemented in Rust. The audit, conducted over a two-week period, evaluated the library's security and correctness across four critical domains: class-group arithmetic, cryptographic protocol correctness, implementation security, and parameter generation. Our analysis identified four security issues of varying severity, with parameter validation vulnerabilities being the most critical concern. Despite these issues, the implementation demonstrates robust memory safety, strong constant-time discipline, and well-structured architecture. The CG-AHE library represents a significant advancement in homomorphic encryption by eliminating the trusted setup requirement that has limited Paillier encryption in decentralized contexts. With the implementation of our recommended fixes, particularly addressing parameter validation, primitivity enforcement, stronger primality testing, and memory zeroization, this library will offer substantially improved security guarantees for threshold cryptography in decentralized applications.

Contents

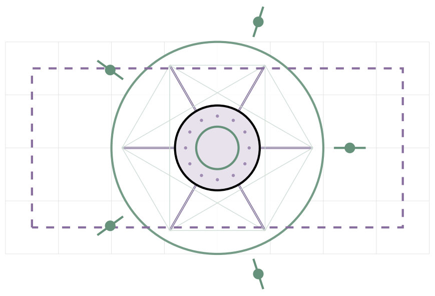
List of Acronyms

API	Application Programming Interface	7
CG-AHE	Class-Group Additively Homomorphic Encryption	i
DoS	Denial of Service	3
FFI	Foreign Function Interface	2
GCD	Greatest Common Divisor	21
IND-CPA	Indistinguishability under Chosen Plaintext Attack	22
LoC	Lines of Code	
NIST	National Institute of Standards and Technology	3
XGCD	Extended Greatest Common Divisor	2

1

Executive Summary

For the busy executive who just doesn't have the time!



At the request of **dWallet Labs**, Symbolic Software conducted a comprehensive cryptographic audit of the organization's innovative CG-AHE library implemented in Rust. This initiative addresses critical limitations in traditional Paillier encryption, specifically its trusted-setup requirement that creates friction with modern decentralized and multi-party architectures. The successful implementation of CG-AHE represents a significant advancement, enabling threshold decryption, secure ledger aggregation, and confidential state updates without dependencies on external primitives or ceremony transcripts—enhancing both security and operational efficiency.

Audit in a Nutshell

Audit window	14 April 2025 – 25 April 2025 (2 weeks)
LoC	8 700 LoC (in-scope `src/` subtree + tests)
Auditors	1 applied cryptographer/Rust engineer
Artifacts	1 Rust crate, rough design paper, test vectors
Communication	Shared Slack channel, daily sync, GitHub issues
Threat model	Malicious ciphertext authors and malicious parameter generators; honest-but-curious infrastructure; local side-channel adversaries

1.1 | Technical Scope

Our audit encompassed four critical domains of analysis:

T1 Class-Group Arithmetic. Thorough verification of binary quadratic-form composition (`nucomp`) and duplication (`nudupl`) operations, reduction algorithms, partial-Euclidean shortcuts, and constant-time Extended Greatest Common Divisor (XGCD) sub-routines—essential components for secure cryptographic operations.

T2 Cryptographic Protocol Correctness. Comprehensive inspection of key-pair generation mechanisms, message encoding/decoding algorithms, encryption/decryption processes, and the additive homomorphism interface that underpins downstream threshold protocols.

T3 Implementation-Level Security. Rigorous assessment of constant-time discipline to prevent timing attacks, memory-safety invariants, error-propagation hygiene, potential panic surfaces, and Foreign Function Interface (FFI) exposures that could introduce vulnerabilities.

T4 Parameter Generation & Hardness Assumptions. In-depth review of discriminant construction methodologies, primality testing robustness, randomness generation requirements, and adherence to state-of-the-art security margins against both current and emergent threats.

1.2 | Methodology

Our audit employed a multi-faceted approach to ensure comprehensive security analysis:

M1 **White-box source inspection**: Meticulous line-by-line code review with particular focus on algebraic invariants and potential subtle panics that could compromise security guarantees.

M2 **Adversarial parameter injection**: Systematic testing with malicious q, p, k values crafted from Carmichael and Lehmer pseudoprimes to probe resistance against sophisticated cryptographic attacks.

M3 **Cross-validation**: Verification of implementation against mathematical specifications and established cryptographic standards.

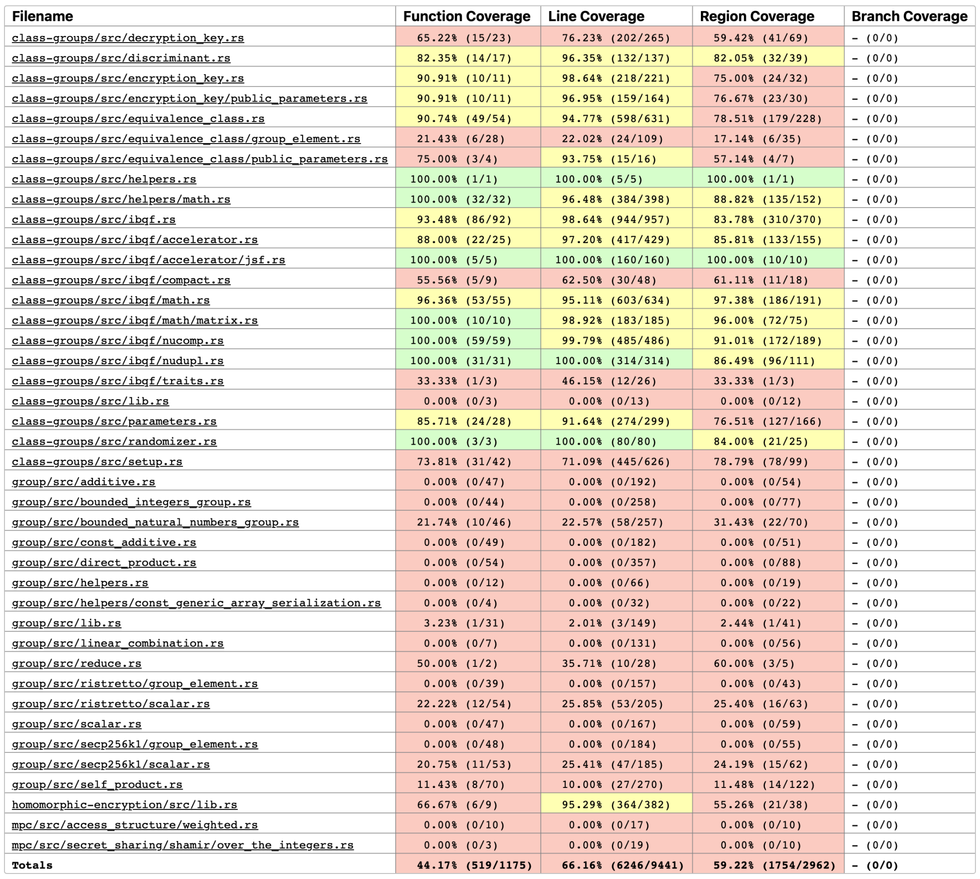
M4 **Edge-case analysis**: Exploration of boundary conditions and rare execution paths that might reveal vulnerabilities.

1.3 | Key Findings

Identifier	Severity	Synopsis
DWLX-02-001 (§4)	High	*Missing primitivity enforcement in `Ibqf::new`.* Admits non-primitive forms, breaking group-law assumptions and opening malleability + Denial of Service (DoS) vectors.
DWLX-02-002 (§4)	Medium	*Inadequate primality testing.* Only three SPRP checks (one deterministic, two probabilistic) on p and q; far below 40-round National Institute of Standards and Technology (NIST) recommendation—composites accepted with $P \geq 12.5\%$.
DWLX-02-003 (§4)	Low	*Single `Discriminant` type masks semantic hazards.* No type-level separation between fundamental and non-fundamental Δ, encouraging misuse and latent security downgrades.
DWLX-02-004 (§4)	Low	*No zeroization of secrets in memory.* Sensitive data (secret keys, nonces, intermediate values) not erased after use, creating vulnerability to memory-disclosure attacks and compromising protocol security.

Positive Observations.

- **Robust memory safety**: All unsafe blocks are effectively encapsulated in a well-reviewed big-integer library; no out-of-bounds access or undefined behavior detected throughout the codebase.
- **Strong constant-time discipline**: Functions not explicitly designated as variable-time consistently demonstrated constant-time execution, minimizing timing side-channel risks.
- **Adequate test coverage**: The test suite is adequate, but leaves room for improvement, with several cases and error paths remaining untested, potentially masking latent bugs (see Figure 1.1 for detailed coverage metrics).
- **Clean architecture**: Well-structured code organization with clear separation of concerns and appropriate abstraction layers.



Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
class-groups/src/decryption_key.rs	65.22% (15/23)	76.23% (202/265)	59.42% (41/69)	- (0/0)
class-groups/src/discriminant.rs	82.35% (14/17)	96.35% (132/137)	82.05% (32/39)	- (0/0)
class-groups/src/encryption_key.rs	90.91% (10/11)	98.64% (218/221)	75.00% (24/32)	- (0/0)
class-groups/src/encryption_key/public_parameters.rs	90.91% (10/11)	96.95% (159/164)	76.67% (23/30)	- (0/0)
class-groups/src/equivalence_class.rs	90.74% (49/54)	94.77% (598/631)	78.51% (179/228)	- (0/0)
class-groups/src/equivalence_class/group_element.rs	21.43% (6/28)	22.02% (24/109)	17.14% (6/35)	- (0/0)
class-groups/src/equivalence_class/public_parameters.rs	75.00% (3/4)	93.75% (15/16)	57.14% (4/7)	- (0/0)
class-groups/src/helpers.rs	100.00% (1/1)	100.00% (5/5)	100.00% (1/1)	- (0/0)
class-groups/src/helpers/math.rs	100.00% (32/32)	96.48% (384/398)	88.82% (135/152)	- (0/0)
class-groups/src/ibqf.rs	93.48% (86/92)	98.64% (944/957)	83.78% (310/370)	- (0/0)
class-groups/src/ibqf/accelerator.rs	88.00% (22/25)	97.20% (417/429)	85.81% (133/155)	- (0/0)
class-groups/src/ibqf/accelerator/jsf.rs	100.00% (5/5)	100.00% (160/160)	100.00% (10/10)	- (0/0)
class-groups/src/ibqf/compact.rs	55.56% (5/9)	62.50% (30/48)	61.11% (11/18)	- (0/0)
class-groups/src/ibqf/math.rs	96.36% (53/55)	95.11% (603/634)	97.38% (186/191)	- (0/0)
class-groups/src/ibqf/math/matrix.rs	100.00% (10/10)	98.92% (183/185)	96.00% (72/75)	- (0/0)
class-groups/src/ibqf/nucomp.rs	100.00% (59/59)	99.79% (485/486)	91.01% (172/189)	- (0/0)
class-groups/src/ibqf/nudupl.rs	100.00% (31/31)	100.00% (314/314)	86.49% (96/111)	- (0/0)
class-groups/src/ibqf/traits.rs	33.33% (1/3)	46.15% (12/26)	33.33% (1/3)	- (0/0)
class-groups/src/lib.rs	0.00% (0/3)	0.00% (0/13)	0.00% (0/12)	- (0/0)
class-groups/src/parameters.rs	85.71% (24/28)	91.64% (274/299)	76.51% (127/166)	- (0/0)
class-groups/src/randomizer.rs	100.00% (3/3)	100.00% (80/80)	84.00% (21/25)	- (0/0)
class-groups/src/setup.rs	73.81% (31/42)	71.09% (445/626)	78.79% (78/99)	- (0/0)
group/src/additive.rs	0.00% (0/47)	0.00% (0/192)	0.00% (0/54)	- (0/0)
group/src/bounded_integers_group.rs	0.00% (0/44)	0.00% (0/258)	0.00% (0/77)	- (0/0)
group/src/bounded_natural_numbers_group.rs	21.74% (10/46)	22.57% (58/257)	31.43% (22/70)	- (0/0)
group/src/const_additive.rs	0.00% (0/49)	0.00% (0/182)	0.00% (0/51)	- (0/0)
group/src/direct_product.rs	0.00% (0/54)	0.00% (0/357)	0.00% (0/88)	- (0/0)
group/src/helpers.rs	0.00% (0/12)	0.00% (0/66)	0.00% (0/19)	- (0/0)
group/src/helpers/const_generic_array_serialization.rs	0.00% (0/4)	0.00% (0/32)	0.00% (0/22)	- (0/0)
group/src/lib.rs	3.23% (1/31)	2.01% (3/149)	2.44% (1/41)	- (0/0)
group/src/linear_combination.rs	0.00% (0/7)	0.00% (0/131)	0.00% (0/56)	- (0/0)
group/src/reduce.rs	50.00% (1/2)	35.71% (10/28)	60.00% (3/5)	- (0/0)
group/src/ristretto/group_element.rs	0.00% (0/39)	0.00% (0/157)	0.00% (0/43)	- (0/0)
group/src/ristretto/scalar.rs	22.22% (12/54)	25.85% (53/205)	25.40% (16/63)	- (0/0)
group/src/scalar.rs	0.00% (0/47)	0.00% (0/167)	0.00% (0/59)	- (0/0)
group/src/secp256k1/group_element.rs	0.00% (0/48)	0.00% (0/184)	0.00% (0/55)	- (0/0)
group/src/secp256k1/scalar.rs	20.75% (11/53)	25.41% (47/185)	24.19% (15/62)	- (0/0)
group/src/self_product.rs	11.43% (8/70)	10.00% (27/270)	11.48% (14/122)	- (0/0)
homomorphic-encryption/src/lib.rs	66.67% (6/9)	95.29% (364/382)	55.26% (21/38)	- (0/0)
mpc/src/access_structure/weighted.rs	0.00% (0/10)	0.00% (0/17)	0.00% (0/10)	- (0/0)
mpc/src/secret_sharing/shamir/over_the_integers.rs	0.00% (0/3)	0.00% (0/19)	0.00% (0/10)	- (0/0)
Totals	**44.17% (519/1175)**	**66.16% (6246/9441)**	**59.22% (1754/2962)**	**- (0/0)**

Figure 1.1: Test coverage metrics across the target, showing function and branch coverage percentages by module.

1.4 | Risk Posture

When operating under honest parameter generation conditions (specifically utilizing genuine primes p, q and a fundamental discriminant), our assessment revealed no practical attack vectors against the system's indistinguishability or message confidentiality properties.

However, we identified critical parameter-validation vulnerabilities that could permit an adversary to introduce weak moduli, resulting in silent security degradation without detection. Given that such compromised parameters would propagate throughout the public network state, we consider this a systemic risk requiring immediate remediation. The provided recommendations outline a clear path to address these concerns.

1.5 | Conclusion

The dWallet Labs CG-AHE implementation represents a significant advancement in homomorphic encryption by eliminating the trusted setup requirement that has limited Paillier encryption in decentralized contexts. Our audit identified four security issues of varying severity that require remediation before production deployment, with parameter validation vulnerabilities being the most critical concern. Despite these issues, the codebase demonstrates strong memory safety, consistent constant-time discipline, and well-structured organization.

With the implementation of our recommended fixes—particularly addressing parameter validation, primitivity enforcement, stronger primality testing, and memory zeroization—this library will offer substantially improved security guarantees. The fundamental advantage of this system is significant: no amount of collusion can reveal the class group order, unlike in Paillier where parties must trust the setup phase. The commitment to security best practices evident throughout the implementation gives us confidence that the dWallet Labs team can successfully address the identified issues, creating a solid foundation for threshold cryptography in decentralized applications.

2

Target Overview

This chapter describes the three principal targets that fell within the scope of the CG-AHE audit:

1. **CG-AHE Core Rust Crate.** The primary artefact under review implements all algebraic operations on binary quadratic forms together with the encryption, decryption, and homomorphic-addition interfaces described in the CG-AHE white-paper. Our objective was to verify the functional correctness of those low-level algorithms, their adherence to constant-time discipline, and their resistance to malleability and subgroup attacks.
2. **CG-AHE Design Notes.** Informal notes and third-party references [1–3] specify parameter generation, security rationale, and performance benchmarks. We checked that the Rust implementation matches the specification and that every invariant assumed in the paper is enforced in code.

Consistent with dWallet Labs' Statement of Work, the audit comprised two orthogonal assessments:

1. **Functional Correctness Assessment** — ensuring that each algorithm behaves as prescribed by the informal specification that encoders are injective, that class-group composition is associative, and that decryption is the inverse of encryption.
2. **Security Assessment** — evaluating the robustness of the code against timing, fault, and algebraic attacks; confirming the soundness of parameter validation; and checking that no unsafe block or third-party call invalidates the assumed threat model.

As always, the size of the code-base and the two-week engagement window limited the depth attainable for every component. The "Coverage Level" icons below are therefore meant purely as a *qualitative* snapshot of how much engineering effort could be invested in each target — not as a formal guarantee of completeness.

2.1 | CG-AHE Core Crate

Coverage Level: ●●●◐○

- **Repository:** https://github.com/dwallet-labs/cryptography-private/tree/main/class-groups
- **Branch:** main
- **Commit:** e105b4dad26a2350051788bd527e7a844aeaba54

2.1.1 | Major Code Domains

Class-group arithmetic & form handling DIR ibqf/

- FILE nucomp.rs, FILE nudupl.rs provide algorithms for composing and duplicating forms within the class group.
- FILE math.rs, FILE math/matrix.rs contain auxiliary number-theoretic functions and matrix operations used in form arithmetic.
- FILE compact.rs implements compressed encoding schemes for efficient storage and transmission of quadratic forms.
- FILE traits.rs defines the public Application Programming Interface (API)s that expose the functionality of the core library to external users.
- FILE accelerator.rs offers generic helpers to accelerate exponentiation operations within the class group.

Focus: correctness of group law, constant-time reduction loops, and matrix-GCD helpers.

Accelerated exponentiation DIR ibqf/accelerator/

- FILE jsf.rs implements the joint-sparse-form ladder, which is used by the add-on ”accelerator” module to perform fixed-base exponentiation efficiently; this module is integrated with the FILE accelerator.rs file described above.

Discriminant & system parameters DIR .

- FILE discriminant.rs is responsible for generating both fundamental and non-fundamental discriminants used throughout the system.
- FILE parameters.rs performs validation checks such as verifying Legendre symbol conditions and enforcing size bounds on parameters.
- FILE setup.rs handles the initialization of all global CG-AHE parameters during the system setup phase.

Equivalence-class abstraction DIR equivalence_class/

- FILE `group_element.rs` provides a wrapper around reduced forms to encapsulate the equivalence classes.
- FILE `public_parameters.rs` manages metadata related to the group structure and parameters.

This module exposes a clean, high-level interface designed for use by client crates.

Key material & cryptographic operations DIR `encryption_key/`

- FILE `encryption_key.rs` responsible for sampling secret keys securely.
- FILE `decryption_key.rs` manages ciphertext construction procedures.

2.2 | Auxiliary Components (Limited to No Review)

- DIR `dkg/`, DIR `decryption_key_share.rs`, DIR `publicly_verifiable_secret_sharing/`: examined only when directly exercising CG-AHE ciphertexts or parameters.
- DIR `benches/`, DIR `examples/`: spot-checked for unsafe `unwrap()`s and insecure timing harnesses but not exhaustively audited.

2.3 | Security Boundary Analysis

Independent of file layout we traced four critical boundaries:

1. **Ciphertext interface** — ensures that adversarial (c_1, c_2) cannot trigger panic paths or variable-time exponent ladders.
2. **Key isolation** — verifies that secret exponents remain in constant-time memory and are wiped on drop.
3. **Parameter validation** — rejects composite p, q, non-fundamental Δ, and degenerate randomness spaces.
4. **Homomorphic addition** — checks that fresh randomness is mandatory and that no low-order subgroup confinement is possible.

This structured scope mirrors the Statement of Work and guided the effort distribution reported in the Coverage Level indicators above.

3

Functional Correctness Assessment

This chapter details the mathematical foundations, algorithms, and sketches of correctness proofs for the CG-AHE scheme. The system aims to provide strong security guarantees while supporting efficient homomorphic operations over encrypted data without requiring trusted setup procedures.

3.1 | Mathematical Foundations

3.1.1 | Binary Quadratic Forms and Class Groups

The cryptosystem operates within the class group of binary quadratic forms, an algebraic structure with properties ideal for homomorphic encryption.

Definition 3.1 (Binary Quadratic Form)**.** A binary quadratic form is a polynomial $f(x, y) = ax^2 + bxy + cy^2$ with integer coefficients, often represented as the triplet (a, b, c).

Definition 3.2 (Discriminant)**.** The discriminant of a binary quadratic form (a, b, c) is defined as $\Delta = b^2 - 4ac$.

Let $\Delta < 0$ be a *fundamental discriminant*[1], and denote by:

$$\mathrm{Cl}(\Delta) = \{(a, b, c) \in \mathbb{Z}^3 : b^2 - 4ac = \Delta,\ \gcd(a, b, c) = 1\}/\sim \qquad (3.1)$$

the class group of binary quadratic forms of discriminant Δ, where $\sim$ denotes proper equivalence of forms.

The class group $\mathrm{Cl}(\Delta)$ forms a finite abelian group under the *composition* operation, which generalizes multiplication to quadratic forms. Two important properties make class groups suitable for cryptography:

1. The group operation is efficiently computable
2. Computing the group order is computationally hard for large discriminants

[1]A fundamental discriminant satisfies $\Delta \equiv 0, 1 \pmod 4$ and is square-free except for a possible factor of 4 when $\Delta \equiv 0 \pmod 4$.

3.1.2 | Reduced Forms and Efficient Composition

For computational efficiency, forms are kept in *reduced* representation, characterized by:

$$a > 0, \quad |b| < a < c \tag{3.2}$$

Each equivalence class contains exactly one reduced form, providing a unique canonical representation. For efficient implementation, the scheme employs:

- `nucomp` and `nudupl` algorithms for composing forms without computing intermediate forms of large coefficient size [1, 2]
- *Partial* Euclidean reduction to minimize reduction steps

Algorithm 1: Construction of a Binary Quadratic Form

```
Function IBQF::new($a$ : int, $b$ : int, $\Delta$ : Discriminant) $\to$ IBQF:
    Input: Coefficients $a, b$ and discriminant $\Delta$
    Output: A reduced binary quadratic form
    $x \leftarrow b^2 - \Delta$;
    assert $4a$ divides $x$;
    $c \leftarrow x/(4a)$;
    form $\leftarrow (a, b, c)$;
    reduced_form $\leftarrow$ reduce(form);
    return reduced_form;
```

Algorithm 2: Construction of the Identity Element

```
Function IBQF::unit_for($\Delta$ : Discriminant) $\to$ IBQF:
    Input: A fundamental discriminant $\Delta$
    Output: The identity element of Cl($\Delta$)
    $a \leftarrow 1$;
    $b \leftarrow |\Delta| \bmod 2$;
    return IBQF::new($a, b, \Delta$);
```

Algorithm 3: Construction of an Equivalence Class

```
Function EquivalenceClass::try_from(form : IBQF) → EquivalenceClass:
    Input: A binary quadratic form
    Output: An equivalence class representation
    $\Delta \leftarrow$ form.discriminant();
    assert $\Delta$ is a valid discriminant;
    return EquivalenceClass(form);
```

3.1.3 | Exponentiation in Class Groups

Exponentiation (repeated composition) of a form f by an exponent $e \in \mathbb{N}$ is fundamental to the cryptographic operations. The scheme implements this using the square-and-multiply algorithm:

Algorithm 4: Exponentiation in Class Groups

```
Input: A form $f \in \mathrm{Cl}(\Delta)$ and exponent $e \in \mathbb{N}$
Output: $f^e \in \mathrm{Cl}(\Delta)$
result $\leftarrow$ identity element;
temp $\leftarrow f$;
while $e > 0$ do
    if $e \bmod 2 = 1$ then
        result $\leftarrow$ nucomp(result, temp);
    end
    temp $\leftarrow$ nudupl(temp);
    $e \leftarrow \lfloor e/2 \rfloor$;
end
return result;
```

For bases that are fixed and reused across multiple operations (like the generator h), the scheme employs optimized methods such as windowed non-adjacent form (wNAF) representation to reduce the number of group operations.

3.1.4 | Plaintext Space and Discriminant Construction

The CG-AHE cryptosystem operates on plaintexts in $\mathbb{Z}_q$ for a prime q. The working discriminant is constructed as follows:

Definition 3.3 (Working Discriminant). Given a prime plaintext modulus q and a secondary prime p, the working discriminant is defined as:

$$\Delta = -p \cdot q^{2k+1} \tag{3.3}$$

where k is a small positive integer (typically $k = 1$, giving $\Delta = -p \cdot q^3$).

Given appropriately chosen p and q, this construction ensures:

- The discriminant is fundamental
- The class group has desirable cryptographic properties
- The size of $|\Delta|$ meets the required security level

Algorithm 5: Construction of System Parameters

```
Function Parameters::new($q$ : int, $k$ : int, $p$ : int, security_parameter :
  int) $\rightarrow$ SetupParameters:
    Input: Plaintext modulus $q$, parameter $k$, prime $p$, and security
      parameter
    Output: System parameters for the cryptosystem
    assert $q > 0$ and $q$ is prime;
    assert $p > 0$ and ($p$ is prime or $p = 1$);
    $\Delta_k \leftarrow -p \cdot q$;
    bits $\leftarrow$ minimal_discriminant_bit_size(security_parameter);
    assert bitsize($\Delta_k$) > bits;
    assert $\Delta_k \equiv 0 \pmod 4$ or $\Delta_k \equiv 1 \pmod 4$;
    assert $p = 1$ or legendre_symbol($q, p$) $= -1$;
    $\Delta_{Qk} \leftarrow -p \cdot q^{2k+1}$;
    return Parameters($q, k, p, \Delta_k, \Delta_{Qk}$);
```

3.1.5 | Generator Construction

The cryptosystem requires a generator element $h \in \mathrm{Cl}(\Delta)$ with specific properties:

Algorithm 6: Generator Construction

```
Function Parameters::h() → EquivalenceClass:
    Output: A generator element $h \in \mathrm{Cl}(\Delta_{Qk})$
    $kp \leftarrow$ smallest prime $p$ such that the Kronecker symbol $\left(\frac{\Delta_{Qk}}{p}\right) = 1$;
    // Find a form with first coefficient $kp$ $b \leftarrow$ solve for $b$ such that
     $b^2 \equiv \Delta_{Qk} \pmod{4kp}$;
    $t \leftarrow$ IBQF::new($kp, b, \Delta_{Qk}$);
    // Exponentiate to ensure desired order properties $h \leftarrow t^{2q^k}$;
    return $h$;
```

3.1.6 | Message Encoding Mechanism

For encryption to work properly, an efficient, injective mapping from plaintexts to class group elements is needed:

Definition 3.4 (Message Encoding)**.** For any message $m \in \mathbb{Z}_q$ where $m \neq 0$, the encoding defines:

$$L_m \equiv m^{-1} \pmod{q}, \quad L_m \in (-q, q) \text{ and } L_m \text{ is odd} \tag{3.4}$$

This L_m is used to construct a form:

$$f^m = \left(a = L_m^2,\ b = q \cdot L_m,\ c = \frac{b^2 - \Delta}{4a}\right) \in \mathrm{Cl}(\Delta) \tag{3.5}$$

For $m = 0$, it is encoded as the identity element of the class group.

Algorithm 7: Message Encoding

```
Function encode($m$ : Plaintext) $\rightarrow$ IBQF:
    Input: A plaintext message $m \in \mathbb{Z}_q$
    Output: A binary quadratic form encoding $m$
    if $m = 0$ then
        return IBQF::unit_for($\Delta_{Qk}$);
    end
    $q \leftarrow$ plaintext space order;
    // Compute inverse modulo q, ensuring it's odd $L_m \leftarrow m^{-1}$ mod $q$ in
      range $(-q, q)$ such that $L_m$ is odd;
    $a \leftarrow L_m^2$;
    $b \leftarrow q \cdot L_m$;
    return IBQF::new($a, b, \Delta_{Qk}$);
```

3.2 | Cryptosystem Construction

3.2.1 | System Parameters and Setup

The CG-AHE cryptosystem requires the following public parameters:

- **Plaintext modulus:** A prime q defining the message space $\mathbb{Z}_q$
- **Discriminant:** $\Delta = -p \cdot q^{2k+1}$ (typically with $k = 1$)
- **Generator:** A group element $h \in \mathrm{Cl}(\Delta)$ constructed as described in Algorithm 6
- **Randomness space:** A modulus R defining the range from which random values are drawn (typically $R \approx 2^{2\lambda}$ to $R \approx 2^{3\lambda}$ for λ-bit security)

All parameters are public and verifiable, eliminating the need for trusted setup procedures.

3.2.2 | Key Generation

The cryptosystem utilizes an asymmetric key pair:

Algorithm 8: Secret Key Generation

```
Function sample_secret_key($randomness\_params$ : $RandomnessParameters$) $\rightarrow$ $SecretKey$:
    Input: Parameters defining the randomness space
    Output: A secret key
    $R \leftarrow$ order of the randomness space;
    $sk \xleftarrow{\$} \mathbb{Z}_R$ ;                    // Uniform random sampling
    return $sk$;
```

Algorithm 9: Public Key Construction

```
Function construct_public_key($h$ : $IBQF$, $sk$ : $SecretKey$) $\rightarrow$ $IBQF$:
    Input: Generator $h$ and secret key $sk$
    Output: Public key as a form in $\mathrm{Cl}(\Delta)$
    return $h^{sk}$ ;             // Exponentiation in the class group
```

3.2.3 | Encryption Process

The encryption scheme follows the familiar ElGamal pattern but operates in the class group:

Algorithm 10: Encryption

```
Function encrypt($m$ : $Plaintext$, $pk$ : $PublicKey$) $\rightarrow$ $Ciphertext$:
    Input: Plaintext $m$ and recipient's public key $pk$
    Output: A ciphertext pair $(c_1, c_2)$
    $f^m \leftarrow$ encode($m$) ;      // Encode message to class group element
    $r \xleftarrow{\$} \mathbb{Z}_R$ ;                  // Sample fresh randomness
    $c_1 \leftarrow h^r$ ;                  // First ciphertext component
    $c_2 \leftarrow f^m \cdot pk^r$ ;            // Second ciphertext component
    return $(c_1, c_2)$;
```

3.2.4 | Homomorphic Addition

The cryptosystem supports homomorphic addition of ciphertexts, enabling computation on encrypted data:

Algorithm 11: Homomorphic Addition

```
Input: Ciphertexts $\mathbf{c} = (c_1, c_2)$ encrypting $m$ and $\mathbf{c}' = (c'_1, c'_2)$ encrypting $m'$
Output: A ciphertext encrypting $m + m' \bmod q$
$r \xleftarrow{\$} \mathbb{Z}_R$ ;                        // Fresh randomness for re-randomization
$c_{add,1} \leftarrow c_1 \cdot c'_1 \cdot h^r$;
$c_{add,2} \leftarrow c_2 \cdot c'_2 \cdot pk^r$;
return $(c_{add,1}, c_{add,2})$;
```

The re-randomization step is crucial for security, preventing leakage that might occur from simply multiplying the ciphertext components.

3.2.5 | Decryption Process

Decryption consists of two phases: recovering the encoded form and then extracting the original message:

Algorithm 12: Ciphertext Decoding

```
Function discrete_log_in_F($enc : IBQF$) $\rightarrow$ Plaintext:
    Input: An encoded form $f^m$
    Output: The original plaintext message $m$
    $q \leftarrow$ plaintext space order;
    $u \leftarrow enc.b$ ;                        // Extract middle coefficient
    while $u \equiv 0 \pmod{q}$ do
        $u \leftarrow u/q$;
    end
    $m \leftarrow u^{-1} \bmod q$ ;                  // Compute modular inverse
    return $m$;
```

Algorithm 13: Decryption

```
Function decrypt($ciphertext$ : Ciphertext, $sk$ : SecretKey) → Plaintext:
    Input: A ciphertext pair $(c_1, c_2)$ and secret key $sk$
    Output: The decrypted plaintext message
    $(c_1, c_2) \leftarrow ciphertext$;
    // Use secret key to recover encoded message $g \leftarrow c_2 \cdot (c_1^{sk})^{-1}$ ;
      // Division in class group
    // Extract message from encoded form $m \leftarrow$ discrete_log_in_F$(g)$;
    return $m$;
```

3.2.6 | Correctness Proof

The decryption algorithm correctly recovers the original message because:

$$
\begin{aligned}
g &= c_2 \cdot (c_1^{sk})^{-1} && (3.6)\\
&= f^m \cdot pk^r \cdot (h^{r \cdot sk})^{-1} && (3.7)\\
&= f^m \cdot (h^{sk})^r \cdot (h^{r \cdot sk})^{-1} && (3.8)\\
&= f^m \cdot h^{sk \cdot r} \cdot h^{-sk \cdot r} && (3.9)\\
&= f^m && (3.10)
\end{aligned}
$$

Therefore, the scheme successfully recovers the encoded form f^m, from which it can extract the original message m using the discrete logarithm function in the small group defined by the plaintext space.

3.3 | Implementation Considerations

3.3.1 | Constant-Time Implementation

To prevent timing side-channel attacks, all cryptographic operations must be implemented in constant time:

- **XGCD**: Both `nucomp` and `nudupl` algorithms rely on XGCD operations. The CG-AHE implementation uses a constant-time implementation that ensures execution time does not leak information about operands.
- **Modular Exponentiation**: The square-and-multiply algorithm must process all bits of the exponent, regardless of their value, to prevent timing attacks.

3.3.2 | Performance Optimization

Several optimizations are crucial for practical performance:

- **Partial Euclidean Reduction**: Cohen's `partial_euclidean` reduction algorithm significantly reduces computational burden:
  - Without partial reduction: Approximately 200 reduction passes on average
  - With partial reduction: Typically ≤ 4 reduction passes
  - Performance impact: Approximately 40× speedup in practice
- **Optimized Exponentiation**: For fixed-base operations (like computing with the generator h), precomputation techniques can significantly accelerate exponentiation.

3.4 | Security Analysis

The security of the cryptosystem relies on two core computational hardness assumptions:

1. **Class Group Order Problem**: Computing the exact order of $\mathrm{Cl}(\Delta)$ is computationally difficult for large discriminants.
2. **Class Group Discrete Logarithm Problem**: Given h and h^x in $\mathrm{Cl}(\Delta)$, finding x is computationally difficult.

3.4.1 | Security Parameter Selection

The best-known attacks against these problems have sub-exponential time complexity of $L_{|\Delta|}(1/2)$. To achieve λ-bit security, the parameters must ensure:

$$|\Delta| \gtrsim 2^{\lambda^2} \tag{3.11}$$

For example:

- For 128-bit security: $|\Delta| \approx 2^{16384}$ (theoretical requirement)
- In practice: Optimizations and more precise security estimates allow for somewhat smaller parameters

3.4.2 | Advantages Over Other Systems

Unlike other homomorphic cryptosystems such as Paillier:

- No trusted setup with secretly factored modulus
- All parameters are public and independently verifiable
- Security based on different hardness assumptions, providing diversification

3.5 | Performance Characteristics

All key operations in the cryptosystem scale quasi-linearly with respect to $\log|\Delta|$, ensuring reasonable performance even with large security parameters:

Operation	Group Exp.	Reductions	Notes
Key Generation	1	≈ 3	One fixed-base exponentiation for computing h^s.
Encryption	2	≈ 5	Two exponentiations plus encoding overhead.
Homomorphic addition	1	≈ 2	Requires fresh randomness for re-randomization.
Decryption	1	≈ 2	Additional small discrete logarithm computation in $\mathbb{Z}_q$.

Table 3.1: Computational complexity of cryptographic operations

When implemented with partial reduction techniques and optimized exponentiation algorithms, the CG-AHE encryption scheme demonstrates performance competitive with established systems like Paillier (using $\Delta \approx 2^{4096}$). Crucially, it achieves this efficiency while eliminating the need for trusted setup procedures, providing a significant advantage for applications requiring transparent security assumptions.

4

Security Assessment

4.1 | **DWLX-02-001** Missing Primitivity Check in `Ibqf::new`

Finding Overview **High**

The constructor for integral binary quadratic forms does not verify that the newly-created triple (a, b, c) is *primitive* (i.e. $\gcd(a, b, c) = 1$). Accepting non-primitive forms silently expands the state space with elements that are *not* members of the intended class group, breaking algebraic invariants relied upon by key generation, encryption, and correctness proofs. An attacker can supply or manipulate ciphertext components so that internal reductions act on an invalid subgroup, causing decryption failures, malleability, or, under some scenarios, facilitating subgroup-order leakage.

The check is implemented in the helper `is_primitive()`, yet the constructor never invokes it:

FILE

ibqf.rs

```
let (c, remainder) = b_sqr_min_d.checked_div_rem(&four_a);
...
let c = c.unwrap().resize::<LIMBS>().to_nz().unwrap();
Ok(Self {              // <-- returned without validating gcd(a,b,c)
^^I^^I^^I^^Ia, b, c,
^^I^^I^^I^^Idiscriminant_bits: discriminant.bits_vartime(),
})
```

Impact. 1. **Group-law violations.** Composition assumes primitivity; non-primitive inputs can leave the reduced-form domain, leading to undefined behaviour or panics. 2. **Ciphertext malleability.** A malicious sender can craft (c_1, c_2) whose components collapse under repeated reductions, producing predictable low-order elements that partially linearise discrete logs. 3. **Fault induction.** Failing reductions increase the iteration count drastically (hundreds of passes), opening timing and DoS vectors.

Recommendation: Enforce Primitivity at Construction

- Call `self.is_primitive()` (or an equivalent constant-time Greatest Common Divisor (GCD) routine) inside `Ibqf::new` and return `Err(Error::InvalidFormParameters)` on failure.
- Unit-test with corner-case triples such as (qa, qb, qc) where q is small, ensuring rejection.
- Where performance is critical, embed a short-circuit check exploiting the known co-primality of (a, b) for typical inputs, but always fall back to a full GCD in constant time to avoid side-channels.
- Document the invariant so that downstream constructors and FFI bindings cannot bypass it.

4.2 | **DWLX-02-002** Insufficient Primality Testing for Public Parameters p and q

Finding Overview **Medium**

The scheme's security relies on both the plaintext-space modulus q and the auxiliary prime p being *provably prime*. The current validation routine, however, accepts a candidate after just (i) one Miller–Rabin iteration with base 2, (ii) a Lucas strong probable-prime (SPRP) test, and (iii) *one* additional Miller–Rabin iteration with a single random base. For 256-bit and larger integers this is far below the 40 independent randomized Miller-Rabin rounds recommended for cryptographic applications. Consequently, a composite passing these lax checks with probability $\approx 2^{-3}$ could be accepted as "prime," invalidating the class-group order assumptions, breaking semantic security.

Prime validation flow:

FILE parameters.rs

```
fn validate_parameters(...) -> Result<(), Error> {
    ...
    // Verify q is valid
    Self::validate_q(q, rng)?;

    // Verify that p is prime, or one.
    if !(p.get() == Uint::ONE || is_prime_with_rng(rng, p.deref())) {
        return Err(Error::InvalidPublicParameters);
    }
}

fn validate_q(q: &Uint<...>, rng: &mut impl CryptoRngCore) -> Result<(),
    ↪ Error> {
    if !is_prime_with_rng(rng, q) {
        return Err(Error::InvalidPublicParameters);
    }
}
```

Underlying test:

FILE primality.rs

```
fn _is_prime_with_rng<T: Integer + RandomMod>(rng: &mut impl CryptoRngCore
    ↪ ,
                                              candidate: Odd<T>) -> bool {
    let mr = MillerRabin::new(candidate.clone());

    if !mr.test_base_two().is_probably_prime() { return false; }
    if lucas_test(candidate, AStarBase, LucasCheck::Strong).is_composite()
    ↪  { return false; }

    // only ONE random -MillerRabin base
    if mr.bits() > 2 && !mr.test_random_base(rng).is_probably_prime() {
    ↪ return false; }
    true
}
```

Impact.

- A composite q invalidates the injective message encoder (§3.1.6), allowing distinct messages to share the same encoding and breaking Indistinguishability under Chosen Plaintext Attack (IND-CPA).
- A composite p (or $p = 1$ with composite q) can yield a non-fundamental discriminant whose class group has small subgroups, enabling subgroup confinement or faster discrete-log attacks.

- Attackers can craft malicious parameter sets that pass validation with probability 2^{-3} and later cause decryption failures or timing distinguishers, leading to denial-of-service.

Recommendation: Harden Primality Testing

1. Replace the current routine with at least **40** independent Miller–Rabin rounds with cryptographically secure random bases.
2. Unit-test the validator with Carmichael numbers (or similar) is not necessary, due to this already being covered by the `crypto-primes` crate.
3. Document strict primality requirements so downstream integrations do not replicate this vulnerability.

4.3 | **DWLX-02-003** Lack of Type-Level Distinction Between Fundamental and Non-Fundamental Discriminants

Finding Overview Low

The code base represents every quadratic discriminant with the same `Discriminant` wrapper, even though *fundamental* ($\Delta = -pq$) and *non-fundamental* ($\Delta = -pq^{2k+1}$, $k \geq 1$) variants have different algebraic — and therefore security — properties. Because the two flavours share one constructor and a common API, call-sites cannot express or enforce which flavour they require. A single misplaced parameter therefore compiles and only manifests later as a logic or subtle correctness error (e.g. reduced class-group order, failure of `partial_euclidean` heuristics).

Impact.

- **Security downgrade.** Accidentally feeding a non-fundamental discriminant into routines that assume fundamentalness may create extra automorphisms, shrinking the effective group size and weakening the discrete-log assumption.
- **Maintenance risk.** Future contributors cannot see from the type signature alone whether a function expects Δ_k or Δ_{q^k}, leading to fragile / duplicated validation code and higher defect density.

- **Bug-hiding.** Tests that mix discriminant kinds silently pass compilation, so unit coverage does not guarantee semantic correctness.

Recommendation: Introduce Strong Discriminant Types

Adopt a zero-cost, new-type discipline:

```
/// Security-critical: class-group order is unknown and maximal.
struct FundamentalDiscriminant(Discriminant);
impl FundamentalDiscriminant {
    fn new(p: Prime, q: Prime) -> Result<Self, Error> {
        // validate p·q ≡ 3 (mod 4), size ≥ λ bits, etc.
        Ok(Self(Discriminant::from(-p*q)))
    }
}

/// Acceptable only when the protocol explicitly needs Δ = -p·q^{2k+1}.
struct NonFundamentalDiscriminant(Discriminant);
impl NonFundamentalDiscriminant {
    fn new(q: Prime, k: u8, p: Prime) -> Result<Self, Error> {
        Ok(Self(Discriminant::from(-p*q.pow(2*k+1))))
    }
}
```

1. Tag every API that requires a specific discriminant kind with the corresponding wrapper type.
2. Move validity checks into the `new` constructors so they cannot be bypassed.
3. Use the compiler as proof-obligation: a function that demands `FundamentalDiscriminant` cannot accidentally be called with `NonFundamentalDiscriminant`.
4. Document the semantic difference in the public API to guide downstream users.

This change incurs zero runtime overhead while eliminating an entire class of parameter-mixing bugs and clarifying security guarantees at the type level.

4.4 | **DWLX-02-004** No Zeroization of Secrets in Memory

Finding Overview Low

The CG-AHE crate does not zeroize secrets in memory after they are no longer needed, potentially leaving sensitive data vulnerable to unauthorized access.

The CG-AHE crate performs various cryptographic operations that involve sensitive data, such as secret keys, nonces, and other private values. However, the crate does not take any measures to securely erase these secrets from memory once they are no longer required. This lack of zeroization leaves the sensitive data vulnerable to potential unauthorized access or exploitation.

In the event of a memory disclosure vulnerability or a memory dump, an attacker could potentially access the unzeroized secrets, compromising the security of the cryptographic operations and the overall protocol. This is particularly concerning given the distributed nature of the CG-AHE protocol, where multiple parties are involved, and the security of the entire system depends on the confidentiality of the shared secrets.

The Rust implementation of CG-AHE involves numerous complex cryptographic operations and protocol steps, resulting in the generation and handling of a significant number of intermediate values. These intermediate values often contain sensitive information derived from the secret keys, nonces, and other private data used in the protocol.

Due to the high complexity of the codebase and the extensive use of generic types and traits, identifying and tracking all the intermediate values that require zeroization can be a challenging task. The intermediate values may be stored in various data structures, passed as function arguments, or returned as results, making it difficult to ensure comprehensive zeroization without a thorough analysis of the entire codebase.

To effectively implement zeroization in the CG-AHE crate, a comprehensive deep-dive into the codebase is necessary. This involves carefully examining each cryptographic operation and protocol step to identify all the intermediate values that contain sensitive information. Special attention should be given to the following aspects:

1. **Identifying Sensitive Data**: Analyze the codebase to identify all the data structures, variables, and function parameters that store or handle sensitive information, such as secret keys, nonces, and intermediate values derived from them.

2. **Tracing Data Flow**: Follow the flow of sensitive data throughout the codebase, including function calls, assignments, and data transformations, to ensure that all intermediate values are properly identified and tracked.

3. **Determining Intermediate Values Lifecycle**: Assess the lifecycle of each intermediate value to determine the appropriate point at which it should be zeroized. This includes identifying when the value is no longer needed and ensuring that zeroization occurs before the memory is deallocated or reused.

4. **Implementing Zeroization**: Modify the codebase to implement zeroization for all identified intermediate values. This may involve adding calls to the 'zeroize()' method at the appropriate points, ensuring that the memory is securely erased before it is released or overwritten.

Conducting a comprehensive deep-dive into the CG-AHE codebase to identify and zeroize all intermediate values is a substantial undertaking. It requires a thorough understanding of the cryptographic operations, protocol steps, and the overall structure of the codebase.

Recommendation: Use the Rust `zeroize` crate

Utilize the Rust `zeroize` crate [**zeroize**] to securely erase sensitive data from memory. The `zeroize` crate provides a simple and efficient way to zeroize memory locations that contain secrets. By implementing the `Zeroize` trait for sensitive data structures and calling the `zeroize()` method when the data is no longer needed, the crate ensures that the secrets are overwritten with zeroes, effectively preventing their recovery.

5 Conclusions

This chapter presents the final assessment and recommendations resulting from our audit of dWallet Labs' CG-AHE library. Our two-week engagement comprised a detailed review of the codebase, cryptographic protocols, and mathematical foundations that underpin this innovative implementation. The findings below consolidate our security assessment and functional correctness evaluation.

5.1 | Summary of Findings

Our audit identified four security issues of varying severity, alongside several positive aspects of the implementation. This mixed profile indicates a codebase that is fundamentally sound in its mathematical approach but requires specific security enhancements before production deployment.

High-Severity Security Issues. The most severe finding (DWLX-02-001) involved a missing primitivity check in the binary quadratic form constructor. This omission could allow malicious actors to craft inputs that break group-law assumptions, potentially enabling ciphertext malleability or denial-of-service vectors. The medium-severity issue (DWLX-02-002) concerned insufficient primality testing for public parameters, which could undermine the cryptographic guarantees of the entire system.

Design and Implementation Concerns. Two low-severity issues were identified: a lack of type-level distinction between fundamental and non-fundamental discriminants (DWLX-02-003), which introduces maintenance risks and potential security downgrades; and the absence of proper memory zeroization for secret data (DWLX-02-004), exposing sensitive information to potential memory disclosure attacks.

Positive Observations. Despite these concerns, the codebase demonstrated several commendable qualities:

- Strong memory safety with effective encapsulation of unsafe blocks
- Consistent constant-time discipline in cryptographic operations

- Adequate test coverage according to industry standards
- Well-structured code organization with appropriate abstraction layers

5.2 | Risk Assessment

Our risk assessment indicates that the CG-AHE implementation can achieve its security objectives under honest parameter generation conditions. However, the identified parameter validation vulnerabilities represent a systemic risk that must be addressed. If exploited, these vulnerabilities could lead to silent security degradation across the network, potentially compromising message confidentiality.

Trust Model Considerations. The CG-AHE design eliminates the trusted setup requirement present in traditional Paillier encryption—a significant advancement for decentralized applications. This is particularly valuable because the scheme has no trapdoor, meaning that even if all parties were to collude, they could not extract the order of the class group. This represents a fundamental security improvement over Paillier, where parties must trust that a threshold of participants during setup were honest. However, this benefit is partially undermined by weak parameter validation, which reintroduces trust assumptions that the system was specifically designed to avoid. Proper implementation of our recommendations will restore the intended trust-minimization properties.

Mathematical Foundations. Class groups have been extensively studied in mathematics dating back to Gauss, with significant interest from both cryptographers and number theorists. It's worth noting that computing class group orders appears to be computationally harder than integer factorization—only an $L(1/2)$ algorithm is known for class groups compared to $L(1/3)$ for factoring. While there are no direct reductions from class group problems to factoring, determining whether a number is square-free (which can be accomplished with class group order computation) is believed by many to be equivalent to factoring. The CG-AHE scheme builds upon this rich mathematical foundation, using a classic class-group encryption approach that has been well-defined in peer-reviewed literature.

5.3 | Recommendations

Based on our findings, we recommend the following actions to enhance the security and reliability of the CG-AHE library:

1. **Enforce primitivity at construction** by implementing GCD validation in the `IBQF::new` constructor, preventing non-primitive forms that violate group-law assumptions.
2. **Strengthen primality testing** with at least 40 independent Miller-Rabin rounds for cryptographically secure validation of p and q parameters.
3. **Introduce strong discriminant types** to provide clear type-level distinctions between fundamental and non-fundamental discriminants, leveraging the type system to prevent parameter-mixing errors.
4. **Implement memory zeroization** using the Rust `zeroize` crate to securely erase sensitive data, protecting against memory disclosure vulnerabilities.
5. **Enhance documentation** to explicitly state security assumptions, required parameter properties, and potential risks associated with implementation-specific aspects of class group cryptography.

5.4 | Remediation Prioritization Framework

To assist the dWallet Labs team in efficiently addressing the identified security issues, we provide the following remediation prioritization framework based on severity, implementation complexity, and potential security impact.

Finding	Severity	Estimated Effort	Recommended Timeline
Missing Primitivity Check (DWLX-02-001)	High	Low	Immediate (1-2 days)
Insufficient Primality Testing (DWLX-02-002)	Medium	Low	Short-term (1 week)
Lack of Type Distinction (DWLX-02-003)	Low	Medium	Medium-term (2-4 weeks)
No Memory Zeroization (DWLX-02-004)	Low	High	Long-term (4-6 weeks)

5.4.1 | Implementation Considerations

DWLX-02-001: Missing Primitivity Check.

- **Estimated Effort:** Low (< 1 developer day)
- **Implementation:** Add a single method call to `is_primitive()` within the constructor with appropriate error handling.
- **Testing:** Create unit tests with non-primitive inputs to verify rejection.
- **Interim Mitigation:** If immediate implementation is not possible, add input validation at all public API boundaries to reject potentially non-primitive forms.

DWLX-02-002: Insufficient Primality Testing.

- **Estimated Effort:** Low (1-2 developer days)
- **Implementation:** Modify the primality testing routine to increase Miller-Rabin rounds from 3 to at least 40.
- **Tradeoffs:** This will increase parameter generation and validation time, but the security benefit far outweighs the performance cost.
- **Interim Mitigation:** Add a prominent warning in documentation and use trusted, pre-generated parameters until fixed.

DWLX-02-003: Lack of Type Distinction.

- **Estimated Effort:** Medium (3-8 developer days)
- **Implementation:** Create new type wrappers with proper constructors and modify existing API to use these types.
- **Scope:** This requires refactoring several components and updating all code that interacts with discriminants.
- **Interim Mitigation:** Add runtime assertions and validation checks at critical boundaries while refactoring is in progress.

DWLX-02-004: No Memory Zeroization.

- **Estimated Effort:** High (5-10 developer days)
- **Implementation:** Requires comprehensive analysis of all cryptographic operations to identify sensitive intermediate values, then implementing the `Zeroize` trait for all relevant data structures.

- **Complexity:** The extensive use of generic types and complex data flow makes this a more involved process.
- **Interim Mitigation:** Focus on zeroizing the most critical secrets first (private keys, nonces) while the full solution is developed.

5.4.2 | Phased Remediation Plan

We recommend the following phased approach:

1. **Phase 1 (Week 1):** Address high-severity finding (DWLX-02-001) and implement interim mitigations for all other issues.
2. **Phase 2 (Weeks 1-2):** Address medium-severity finding (DWLX-02-002) and begin design work for type distinction (DWLX-02-003).
3. **Phase 3 (Weeks 3-6):** Implement type distinction system (DWLX-02-003) and begin incremental implementation of memory zeroization (DWLX-02-004).
4. **Phase 4 (Weeks 7-8):** Complete comprehensive memory zeroization and perform regression testing on all changes.

This phased approach ensures that the most critical security issues are addressed immediately while allowing appropriate time for the more complex architectural changes. We recommend a follow-up security review after the completion of Phase 2 to verify the effectiveness of the implemented fixes.

5.5 | Future Directions

While addressing the immediate security concerns will significantly improve the system's resilience, we recommend several additional measures to enhance long-term security:

- **Formal implementation specification:** Develop formal specifications for the CG-AHE implementation, particularly focusing on side-channel resistance and leakage analysis which are less studied for class groups than for more widely deployed schemes.
- **Advanced testing:** Expand the test suite with adversarial examples specifically targeting edge cases in parameter validation and group operations.
- **Academic engagement:** Continue collaboration with the cryptographic research community to further strengthen resistance against side-channel attacks in the specific context of class group operations.

- **Threat modeling:** Conduct comprehensive threat modeling exercises for the specific deployment contexts where CG-AHE will be used.

5.6 | Final Assessment

The dWallet Labs CG-AHE implementation represents a significant advancement in homomorphic encryption, successfully eliminating the trusted setup requirement that has limited the applicability of Paillier encryption in decentralized contexts. With the implementation of our recommended fixes, particularly addressing parameter validation and primitivity enforcement, this library will offer substantially improved security guarantees.

The implementation builds upon well-established cryptographic schemes with rigorously proven security properties based on concrete and widely acknowledged assumptions for class groups. While class group cryptography has extensive mathematical foundations dating back centuries, its practical implementations for modern cryptographic applications face novel challenges, especially regarding side-channel resistance and implementation-specific vulnerabilities.

The security benefits of using a trapdoor-free system are substantial—no amount of collusion can reveal the class group order, unlike in Paillier where parties must trust the setup phase. This fundamental security advantage, combined with the optimizations and implementation work by dWallet Labs, provides a strong foundation for threshold cryptography in decentralized applications.

The commitment to security best practices evident throughout the implementation gives us confidence that the dWallet Labs team can successfully address the identified issues. Once remediated, the CG-AHE library will provide a solid foundation for threshold cryptography in decentralized applications, though continued cryptographic review remains advisable as the system matures and deployment contexts evolve.

6

About Symbolic Software

Symbolic Software[1], established in Paris, France in 2017, is a software consultancy specializing in applied cryptography and software security. The firm has executed over 300 cryptographic software audits within the European information security sector and has made significant contributions to the field by publishing peer-reviewed cryptographic research software.

Offering wide-ranging expertise in cryptographic software audits, Symbolic Software has audited critical cryptographic components of global platforms, ranging from password managers to cryptocurrencies. The company has developed Verifpal® and Noise Explorer, innovative research software for cryptographic engineering, which have contributed to peer-reviewed scientific publications. Symbolic Software's portfolio is marked by collaboration with leading entities such as Cure53 and the Linux Foundation, and they have successfully audited critical technologies like MetaMask and key COVID-19 contact tracing applications in Europe.

[1] Stay updated on Symbolic Software's latest work by visiting `https://symbolic.sofware`.

Bibliography

[1] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Vol. 138. Graduate Texts in Mathematics. Berlin: Springer-Verlag, 1993.

[2] Cyril Bouvier et al. *I want to ride my BICYCL: BICYCL Implements CryptographY in CLass groups*. Cryptology ePrint Archive, Paper 2022/1466. 2022. DOI: 10.1007/s00145-023-09459-1. URL: https://eprint.iacr.org/2022/1466.

[3] Lipa Long. *Binary Quadratic Forms*. White Paper. Accessed 25 Apr 2025. Chia Network, 2019. URL: https://github.com/Chia-Network/vdf-competition/blob/main/classgroups.pdf.

A

Acknowledgements

The author would like to express sincere gratitude to Erik Takke from 3MI Labs for his invaluable contributions to this audit, particularly his comprehensive introductory notes on class-group cryptography and the high-level pseudocode he provided for the fundamental algorithms involved. His expertise significantly enhanced understanding of the complex cryptographic principles underpinning this project.

The author extends deep thanks to Tomer Ashur from 3MI Labs, as well as Yehonathan Cohen Scaly, David Lachmish, and Omer Sadika from dWallet Labs for their exceptional project coordination and facilitation. Their organizational skills and dedication ensured the smooth progression of this audit. Additionally, special thanks are extended to Dolev Mutzari and Offrir Friedman from dWallet Labs for their insightful feedback on class-group cryptography, which greatly contributed to the depth and accuracy of the analysis.