Informal Specification for Class-Group Additively Homomorphic Encryption Scheme

Erik Takke[1,2]

[1] dWallet Labs, Tel Aviv, Israël
[2] 3MI Labs, Leuven, Belgium

Abstract. This is an informal specification of a novel class-group additively homomorphic encryption scheme. It's better than Paillier in that it requires no trusted setup, but it's slower.

1 Main Algorithms

Function *Discriminant::new(d: int) → Discriminant* **is**
  assert($d < 0$);
  assert($d \equiv 0 \bmod 4$ or $d \equiv 1 \bmod 4$);
  return d;
end

Algorithm 1: Constructing a Discriminant

Function *Ibqf::new(a: int, b: int, Δ: Discriminant) → Ibqf* **is**
  $x \leftarrow b^2 - \Delta$;
  assert($4a$ divides x);
  $c \leftarrow x/4a$;
  form $\leftarrow (a, b, c)$;
  reduced $\leftarrow$ reduce(form);
  return reduced;
end
Function *Ibqf::unit_for(Δ: Discriminant) → Ibqf* **is**
  $a \leftarrow 1$;
  $b \leftarrow |\Delta| \bmod 2$;
  return Ibqf::new(a, b, Δ);
end

Algorithm 2: Constructing an Ibqf

References

Function *EquivalenceClass::try_from(form: Ibqf)* $\rightarrow$ *EquivalenceClass* **is**
  $\Delta \leftarrow$ form.discriminant();
  assert(Δ is a valid discriminant);
  return EquivalenceClass(form);
end

Algorithm 3: Constructing an `EquivalenceClass`

Function *Parameters::new(q: int, k: int, p: int, security_parameter: int)* $\rightarrow$ *SetupParameters* **is**
  assert($q > 0$ and q prime);
  assert($p > 0$ and (p prime || $p = 1$));
  $\Delta_k \leftarrow -pq$;
  bits $\leftarrow$ minimal_discriminant_bit_size(security_parameter);
  assert(bitsize(Δ_K) > bits);
  assert($\Delta_K \equiv 0 \bmod 4$ || $\Delta_k \equiv 1 \bmod 4$);
  assert($p = 1$ or legendre_symbol$(q, p) = -1$);
  $\Delta_{QK} \leftarrow -pq^{2k+1}$ // note: we're currently hardcoding $k = 1$;
  return Parameters(q, k, p, Δ_K, Δ_{QK});
end
Function *Parameters::h()* $\rightarrow$ *`EquivalenceClass`* **is**
  $kp \leftarrow$ smallest prime p such that the kronecker extension of the legendre symbol of p and Δ_{QK} equals 1;
  $t \leftarrow$ Ibqf::new$(kp, ?, \Delta_{QK})$ where ? is computed from context;
  $h \leftarrow t^{2q^k}$;
  return h;
end

Algorithm 4: Construction class-group `Parameters`

Function *sample_secret_key(randomness_pp: `RandomnessParameters`)* $\rightarrow$ *SecretKey* **is**
  $R \leftarrow$ order of the randomness space;
  $sk \leftarrow_{\$} \mathbb{Z}/R\mathbb{Z}$;
  return sk;
end

Algorithm 5: Secret Key

Function *construct_public_key(h: Ibqf, sk: SecretKey)* $\rightarrow$ *Ibqf* **is**
  return h^{sk};
end

Algorithm 6: Public Key

Function *encoding(m: Plaintext)* $\rightarrow$ *Ibqf* **is**
// This function is a shortcut for the computation of f^m with f a specifically chosen form with Δ_{QK} as its discriminant;
If $(m = 0)$ return Ibqf::unit_for(Δ_{QK});
$q \leftarrow$ plaintext space order;
$Lm \leftarrow m^{-1}$ mod $q \in \{-q, q\}$ such that it is odd;
$a \leftarrow Lm^2$;
$b \leftarrow q \cdot Lm$;
return Ibqf::new(a, b, Δ_{QK});
end

Algorithm 7: Encoding

Function *encrypt(m: Plaintext, pk: PublicKey)* $\rightarrow$ *Ciphertext* **is**
$f^m \leftarrow$ encode(m);
$r \leftarrow_{\$} \mathbb{Z}/R\mathbb{Z}$ with R the randomness order;
ciphertext $\leftarrow (h^r, f^m \cdot pk^r)$;
return ciphertext;
end

Algorithm 8: Encryption

Function *discrete_log_in_F(enc: Ibqf)* $\rightarrow$ *Plaintext* **is**
$q \leftarrow$ plaintext space order;
$u \leftarrow enc.b$;
while $u \equiv 0$ mod q do $u \leftarrow u/q$;
$m \leftarrow u^{-1}$ mod q;
return m;
end

Algorithm 9: Decoding

Function *decrypt(ciphertext: Ciphertext, sk: SecretKey)* $\rightarrow$ *Plaintext* **is**
$(c_1, c_2) \leftarrow$ ciphertext
decryption $\leftarrow c_2/c_1^{\mathrm{sk}}$
decoding $\leftarrow$ discrete_log_in_F(decryption)
return decoding
end

Algorithm 10: Decryption