FIBS : Fast Isogeny Based Digital Signature*

Suhri Kim[1], Youngdo Lee[2], and Kisoon Yoon[2]

[1] Sungshin Women University
suhrikim@sungshin.ac.kr
[2] NSHC Inc.,
ydlee, ksyoon@nshc.net

Abstract. In this paper, we present FIBS : Fast isogeny-based digital signature based on isogeny-based hash function. We combine the CGL hash function and LMS/SPHINCS hash-based digital signature algorithm. For a 128-bit quantum security level, our implementation in C takes 121.66s for key generation, 2837.04s for signing, and 172.37s for verification.

Keywords: CGL hash· isogeny-based cryptography· hash-based digital signature· PQC

1 Introduction

Due to the increasing interest in post-quantum cryptography, isogeny-based cryptography regained its attention after the work of Couveignes [9]. Especially, due to the introduction of CGL hash function [7] and SIDH (Supersingular Isogeny Diffie-Hellman) key exchange [12], many algorithms have been proposed based on the difficulty of finding the isogeny between two supersingular elliptic curves defined over a finite field. The main advantage of isogeny-based cryptography is that it offers the smallest key sizes among post-quantum cryptography (PQC) primitives. However, as the implementation of isogeny-based cryptography involves isogeny operations in addition to the standard elliptic curve arithmetic over a finite field, its performance is slower than most of the PQC algorithms. On the other hand, for researchers studying isogeny, the lack of diversity in cryptographic primitive compared to other PQC algorithms is also a problem for isogeny-based cryptography.

Constructing a digital signature scheme is much harder to achieve than key exchange for isogeny-based cryptography. The first isogeny-based digital signature algorithm was proposed by Yoo et al. [19] through the application of Unruh's construction of non-interactive zero-knowledge proofs to an interactive zero-knowledge proof. However, not only was their scheme inefficient, but the size of the signature is larger than other post-quantum signature schemes. Moreover, as SIDH was completely broken by Castryck and Decru in [6], this signature is

* This work is submitted to 'Korean Post-Quantum Cryptography Competition' (www.kpqc.or.kr).

no longer secure. Meanwhile, the signature scheme proposed by Galbraith, Petit, and Silva was the first signature scheme based on KLPT algorithm [14]. However, although the algorithm is mathematically complete, its performance was inefficient for practical use. In 2019, SeaSign, a CSIDH-based digital signature scheme proposed by De Feo and Galbraith, alleviated the problem of revealing the secret key through rejection sampling in [9], although several minutes are still required to sign a message [11]. Later, by computing the class group having a 154-digit discriminant, CSI-FiSh [4] offers a practical digital signature scheme that requires 390ms to sign a message. For isogeny-based cryptography, this is a remarkable result, which shed light that various cryptographic primitives can be constructed through elliptic curve isogenies. However, the work in [8] showed that the size of the prime field used in CSI-FiSh must be about 4096-bit to achieve a 128-bit quantum security level. This means that a computation of a class group excessively larger than the 154-digit discriminant is required to reach a practical performance level, which is out of the current computing level.

This paper proposes a moderately fast isogeny-based digital signature based on isogeny-based hash functions. Currently, SQISign proposed by De Feo et al. is considered the most practical digital signature algorithm in isogeny-based cryptography [13]. The performance of the proposed algorithm is not faster than SQISign. However, the aim of this work is to provide diversity in isogeny-based cryptographic primitive. The following list details the main contributions of this work.

1.1 Design rationale

Design rationale and advantages We present FIBS – Fast isogeny-based digital signature based on isogeny-based-hash functions. Isogeniests have long devoured to develop practical isogeny-based digital signature algorithms. However, out of all the hard efforts, the main bottleneck is that its performance is slower than any other digital signature algorithm. The proposal of FIBS simply came from the idea that if an isogeny-based hash function is used for a hash-based digital signature algorithm, the performance could be moderate in an isogeny world. FIBS is 'fast' not 'fastest' as SQISign is the fastest algorithm in isogeny-based digital signatures. However, after the introduction of [8] and [6] it is faster than CSI-FiSh, and safer than [19].

Limitations The major drawback of FIBS is its performance. As an isogeny-based hash function is much slower than cryptographic hash functions such as SHA, FIBS is a magnitude slower than the original hash-based digital signature. However, there is room for improvement. First, finite field arithmetic can be further optimized. Unlike SIDH or CSIDH-based algorithms, the CGL hash function uses Montgomery-friendly primes, so that reduction is much more efficient than other primes of the same bit size. Secondly, isogeny computation can be further optimized by using Richelot (2,2)-isogeny. Lastly, the CGL hash algorithm itself can be modified. The major difference between the CGL hash function and the

general cryptographic hash functions is the size of the input. Unlike the general cryptographic hash functions, the CGL hash function has limited the input size due to the collision attack. There is a lot of research regarding this point and we plan to investigate it as well. The listed above will be later reflected in the optimized implementation.

2 Preliminaries

In this section, we briefly introduce the isogeny-based hash function and hash-based digital signature, which are the main ingredients to the proposed algorithm of the paper.

2.1 Isogeny graph and CGL hash function

In [7], a provable collision-resistant hash function based on the pseudo-random behavior of expander graphs is introduced. As the isogeny graph of supersingular elliptic curves is a Ramanujan graph which is an expander graph with extra properties, this can be exploited for the hash function proposed in [7]. In this subsection, we briefly describe the CGL hash function and its improved version proposed in [17].

The CGL hash function It is well known to complexity theorists that expander graphs produce pseudo-random behavior. In [7], they exploited this characteristic to produce a collision-resistant hash function. In a high-level view, the input message to the hash function is used as directions for walking around a graph, and the output of the hash function is the final vertex of the walk.

Let p and ℓ be two distinct prime numbers. Consider the graph $G = (V, E)$. Let V be the vertex set of the graph G, the set of supersingular elliptic curves defined over the finite field $\mathbb{F}_{p^2}$. The vertices are labeled with their j-invariants. The number of vertices in G is $\lfloor \frac{p}{12} \rfloor + \epsilon$, where $\epsilon \in \{0, 1, 2\}$, depending on the congruence class of p modulo 12. Let E be the set of edges of the graph G. We defined edge $e \in E$ between vertex $v_1, v_2 \in V$ where there exist ℓ-isogeny between elliptic curves E_1 and E_2 having j-invariant corresponding to v_1 and v_2, respectively. For a separable isogeny, there is a one-to-one correspondence with a finite subgroup of elliptic curves and its isogeny. Let ϕ be a separable ℓ-isogeny. Then $\ker \phi$ has ℓ-elements. If $gcd(p, \ell) = 1$ then ℓ-torsion subgroup is isomorphic to the product of two cyclic groups, and as there are $\ell + 1$ different subgroups of order ℓ, it follows that there are $\ell + 1$ isogenies of degree ℓ. Hence the graph G is $(\ell + 1)$-regular graph. Let A be the adjacency matrix of G. Then the eigenvalues λ of A satisfies the Ramanujan bound $\mid \lambda \mid \leq 2\sqrt{\ell}$, so that the isogeny graph of G is the Ramanujan graph.

As an isogeny graph is a special form of expander graph, it can be exploited to instantiate the hash function proposed in [7]. For the rest of the paragraph, we shall describe the CGL hash function using 2-isogenies. Let p be a large prime of the form $p = 2^n \cdot f \pm 1$. Then since we use supersingular elliptic curve

E, $E(\mathbb{F}_{p^2}) = (p \mp 1)^2$. For a 2-isogeny on E, there are 3 possible choices, and since no backtracking is allowed in a walk, there are two choices that can be determined by a single bit of input.

Improved version of CGL hash function Later, it was investigated that the CGL hash function on isogeny graphs of supersingular elliptic curves is insecure under collision attack when the endomorphism ring of the starting curve is known [18].

Let E_0 be the starting curve of the hash function and $O = End(E_0)$ be the endomorphism of E_0. Since E_0 is a supersingular elliptic curve, O is an order in a quaternion algebra. The hash collision in the CGL hash function corresponds to the cycles in the ℓ-isogeny graph containing E_0 which is equivalent to the ℓ-power norm endomorphism of E_0. By using the Deuring correspondence, isogeny from E_0 to some supersingular elliptic curve E corresponds to the left ideals of O. Hence, when an endomorphism $\alpha in O$ of norm ℓ^n for some n is found, an attacker computes the ideal $O\alpha + O\ell^k, (1 \leq k \leq n-1)$. Then using the KLPT algorithm, each of these ideals is transformed into an equivalent ideal of the power-smooth norm to compute the corresponding codomain curve. Then an attacker obtains a sequence of elliptic curves $(E_0, E_1, \ldots, E_{n-1}, E_n = E_0)$, which corresponds to the collision of the CGL hash function.

One might think that this attack can be eliminated if an elliptic curve of an unknown endomorphism ring is used as the starting curve. However, since there is no known way to generate a supersingular elliptic curve with an unknown endomorphism ring, the CGL hash function can only be used after a trusted setup. In [17], an efficient method is proposed which prevents collision attacks and permits the use of arbitrary starting curves.

The main idea in [17] is to use only a fraction of the available edges at each step. Note that the attack in [18] assumes that all of the ℓ-isogeny cycles in the graph yield a collision. Suppose we use ℓ-isogeny graph. The algorithm selects one of r outgoing edges at each step. Then the probability that a cycle of length C yields a collision is roughly upper-bounded by $(r/\ell)^C$. If ℓ is increased, r/ℓ gets small, which decreases the chance that a given cycle yields a collision, but decreases the expected cycle length of the graph as well which increases the chance of the attack. For implementation, the r, ℓ is adjusted to suit the security level and the performance.

2.2 Primitives for hash-based digital signatures

In this subsection, we mainly explain the primitives used in FIBS. FIBS is a stateless hash-based signature framework. It can be composed of one-time signature scheme (OTS), WOTS+ (Winternitz one-time signature scheme plus), Few-time signature scheme (FTS), Forest of Random Subsets (FORS), and binary hash trees.

Winternitz one-time signature scheme plus (WOTS+) We now describe the WOTS+ [15]. WOTS+ is an OTS scheme. A description of the algorithms for key generation and signing is as follows.

Parameters
WOTS+ is parameterized by security parameter $n \in \mathbb{N}$, message length $m \in \mathbb{N}$ and Winternitz parameter w. And define

$$l_1 = \lceil \frac{m}{\log(w)} \rceil, l_2 = \lfloor \frac{\log(l_1(w-1))}{\log(w)} \rfloor + 1, \quad l = l_1 + l_2.$$

WOTS+ uses a cryptographic hash function

$$\mathcal{F}_n : \{f_k : \{0,1\}^* \to \{0,1\}^n | k \in \mathcal{K}_n\}$$

with key space $\mathcal{K}_n$.

We use the notation $c_k^i(x, \mathbf{r})$ for input of value $x \in \{0,1\}^n$, iteration counter $i \in \mathbb{N}$ and random bitmask $\mathbf{r} = (r_1, ..., r_j) \in \{0,1\}^{n \times j}$, Let $\mathbf{r}_{a,b}$ be the set $\{r_a, ..., r_b\}$. We define c recursively as

$$c_k{}^i(x, \mathbf{r}) = f_k(c_k{}^{i-1}(x, \mathbf{r}) \oplus r_i), \quad c^0(x) = x.$$

Key Generation Algorithm $(sk, pk \leftarrow WOTSKeyGen(S, \mathbf{r}))$
On input of seed $S \in \{0,1\}^n$ and random bitmask $\mathbf{r}$, the key generation algorithm computes secret key $sk = (sk_1, ..., sk_l)$, $sk_i \in \{0,1\}^n$. The public key pk is computed as

$$pk = (pk_0, ..., pk_l) = ((\mathbf{r}, k), c_k^{w-1}(sk_1, \mathbf{r}), ..., c_k^{w-1}(sk_l, \ \mathbf{r})), \text{ for } pk_i \in \{0,1\}^n$$

Signature Algorithm $(\Sigma \leftarrow WOTSSign(M, sk, \mathbf{r}))$
On input of m-bit message M, the secret key $\{sk_i\}$ and the random bitmask $\mathbf{r}$. WOTS+ signature algorithm first computes a base-w representation of $M := (M_1, ...M_{l_1}), M_i \in \{0, ..., w-1\}$. Also, it computes the checksum $C = \sum_{i=1}^{l_1}(w - 1 - M_i)$ and its base w representation $C = (C_1, ..., C_{l_2})$.

Define $B = (b_1, ..., b_l) = M||C$. The signature is computed as

$$\Sigma = (\Sigma_1, ..., \Sigma_l) = (c_k^{b_1}(sk_1, \mathbf{r}), ..., c_n^{b_l}(sk_l, \mathbf{r})).$$

Verification Algorithm $(0 \text{ or } 1 \leftarrow WOTSVrfy(M, \Sigma, pk))$
On input of m-bit message M, a signature Σ and a public key pk, the verification algorithm first computes $b_i, 1 \leq i \leq l$ as described above. Then it returns:

$$\begin{aligned} pk &= (pk_0, pk_1, ...pk_l) \\ &\stackrel{?}{=} (c_k^{w-1-b_1}(\Sigma_1, \mathbf{r}_{b_1+1,w-1}), ..., c_k^{w-1-b_l}(\Sigma_l, \mathbf{r}_{b_l+1,w+1})) \end{aligned}$$

If the comparison holds, it returns true and false otherwise.

Hyper tree Hash-based digital signature schemes often use the Merkle tree technique to merge the public keys contained in leaf nodes in the bottom layer into smaller public keys. We use the construction proposed in [10]. The Merkle tree method used in FIBS is essentially the same as the XMSS [5] structure. A Merkle tree of height h always has 2^h leaves which are n bit string $L_i, i \in [2^h - 1]$.Each node $N_{i,j}$ for $0 < j \leq h, 0 \leq i < 2^{h-j}$ of the tree stores an n-bit string. To construct the tree, h bitmasks $\mathbf{Q_j} \in \{0,1\}^{2n}$, are used. The value of the nodes $N_{i,j}$ are computed as

$$N_{i,j} = H((N_{2i,j-1}||N_{2i+1,j-1}) \oplus \mathbf{Q_j})$$

for $H : \{0,1\}^{2n} \to \{0,1\}^n$.
The notion of authentication path $AUTH_i = (A_0, ..., A_{h-1})$ of a leaf node $N_{i,0} = L_i$. $AUTH_i$ consists of all the sibling nodes of the nodes contained in the path L_i to th root. We then extend this to a multi-tree setting, in the same style as XMSS^{MT}. A hyper-tree constructs the multi-level XMSS. The bottom layer tree is used to sign the message, and the rest of the tree is used to sign the root node of the XMSS tree in each layer.

FORS: Forest of Random Subsets SPHINCS+ defines and uses FORS, a few-time signature improved from HORST [2]. FORS is defined in terms of integers k and $t = 2^a$, and can be used to sign strings of ka bit string. The FORS private key consists of kt random n-bit values, grouped into k sets of t values each. In the context of SPHINCS+, these values are deterministically derived from seed using PRF. To obtain the FORS public key, k hash trees are on top of the sets of private key elements. Each t value is used as a leaf node and k binary hash trees with height a are created. We compress the root nodes using a call to the hash function. The resulting n-bit value is the FORS public key.
Given a message of ka bits, we divide it into k strings of a bit. Each of these bit strings corresponds to an index of a single leaf node in each FORS tree. Then the signature consists of authentication paths of k binary tree.

2.3 SPHINCS+

SPHINCS+ [3] is a quantum-resistant hash-based digital signature scheme. It is a stateless hash-based signature using WOTS+, the Merkle tree, and the FTS algorithm FORS. Use a hyper-tree structure, a Merkle tree where each tree has a public key of WOTS+ as a leaf node. The WOTS+ in the leaf node of the lowest layer of the hyper tree is used to sign the public key of FORS. When signing a message, select the leaf node by determining the index of the hyper-tree that one part of the message digest will use. The FORS key pair corresponding to the leaf node is used to sign the digest, and the FORS public key is signed to the corresponding WOTS+ for the leaf node. Each Merkle root in the path from the leaf to the root of the hyper-tree is signed with a WOTS+ key pair in the tree leaf of the layer above it. This continues until the root of the public key, the highest level tree, is reached. The signature contains all FORS and WOTS+

signatures used and the authentication path of the Merkle tree required by the validator to calculate the root of the hypertree.

3 Specification

In this section, the structure of the FIBS proposal in this paper is explained. Based on the security of the CGL hash function mentioned in the section, we apply CGL hash to the part of the chain of hash functions for the private key sk_i in the WOTS+ algorithm.

3.1 Notation

For the rest of the paper, we will use a common notation for various parameters for describing FIBS algorithms.

Table 1: Meanings for symbols used in this paper

Symbols	Meaning
M	The message space which is a subset of $\{0,1\}^*$
w	The Winternitz parameter
n	The security parameter
h	The height of the hypertree
d	The number of layers in the hypertree
k	The number of trees in FORS
t	The number of leaves of a FORS

Table 2: Meanings for variables used in this paper

Variables	Meaning
`Sk.seed`	Used to generate the WOTS+ and FORS private key elements
`Sk.prf`	Used to deterministically generate randomized values for randomized message hashes
`Pk.seed`	Used to generate random values used in hash functions of WOTS+ and Merkle tree, etc
`Pk.root`	Denote the top root node in the hypertree of FIBS
`WOTS.Sk, WOTS.Pk`	Denote the private key and public key of WOTS+ algorithm
`FORS.Pk, FORS.Sig`	Denote the public key and signature of FORS algorithm
`AUTH`	Denote the authentication path of Merkle tree
`XMSS.Pk, XMSS.Sig`	Denote the public key and signature of XMSS algorithm

3.2 Specification of FIBS

Tweakable Hash Function The SPHINCS+ used tweakable hash function [3]. A tweakable hash function takes public seed PK.seed and context information in form of an address `ADRS` in addition to the message input. The address `ADRS` are five types of addresses for the different use cases. They are the WOTS+ hash chain, compression of the WOTS+ public key, hashes in the main Merkle tree, hashes in the FORS Merkle tree, and compression of the tree roots of FORS. These types largely share a common format. The address consists of the following: It always starts with a layer address of one word in the most significant bit, followed by a tree address of three words. The address for each use case consists of the following:

1. WOTS+ hash address
   - Layer address: type = 0
   - Tree address: key pair address ‖ chain address ‖ hash address

2. WOTS+ public key compression address
   - Layer address: type = 1
   - Tree address: key pair address ‖ zero padding ‖ zero padding

3. Hash tree address
   - Layer address: type = 2
   - Tree address: zero padding ‖ tree height ‖ tree index

4. FORS tree address
   - Layer address: type = 3
   - Tree address: key pair address ‖ tree height ‖ tree index

5. FORS tree roots compression address
   - Layer address: type = 4
   - Tree address: key pair address ‖ zero padding ‖ zero padding

Therefore, if the given address is used for functions with different use cases within the function, it should be used after initialization. These processes are omitted in this section.

The SPHINCS+ defined the function family using the tweakable hash function. FIBS does not define all hash functions differently because it uses the CGL hash function exclusively. All compression hash functions and keying hash functions are used the CGL hash function. The CGL hash function can process arbitrary length input:

$$\text{CGL-HASH } \psi : \{0,1\}^* \rightarrow \{0,1\}^n$$

The hash functions used in FIBS are organized as follows:

- $\mathbf{H_{msg}}$: Additional key hash function that can handle messages of arbitrary length.
- **PRF**: Pseudo-random function for generating pseudo-random keys.
- $\mathbf{PRF_{msg}}$: Using PRF to generate randomness for message compression.
- **F**: Second-preimage resistant, one-way function
- **H**: Second-preimage resistant hash function
- $\mathbf{T_l}$:Tweakable hash functions of the form mapping an ln-byte message M to an n-byte hash value

The compression function and hash function that we actually use in FIBS are as follows:

- $\mathbf{H_{msg}}$(R, `PK.seed`, `PK.root`, M) = **CGL-HASH**(R$\|$`PK.seed`$\|$`PK.root`$\|$M),
- **PRF**(SEED, `ADRS`) = **CGL-HASH**(SEED$\|$`ADRS`)
- $\mathbf{PRF_{msg}}$(`SK.prf`, Optrand, M) = **CGL-HASH**(`SK.prf`$\|$Optrand$\|$M),
- **F**(`PK.seed`, `ADRS`, M1) = **CGL-HASH**(`PK.seed`$\|$`ADRS`$\|$M1),
- **H**(`PK.seed`, `ADRS`, M1$\|$M2) = **CGL-HASH**(`PK.seed`$\|$`ADRS`$\|$M1$\|$M2)
- $\mathbf{T_l}$(`PK.seed`, `ADRS`, M) = **CGL-HASH**(`PK.seed`$\|$`ADRS`$\|$M).

WOTS+ WOTS+ uses the security parameter n, Winternitz parameter w, the number of n-byte string elements in a WOTS+ private key, public key, and signature. The l is defined as $l = l_1 + l_2$, where l_1 and l_2 are as follows:

$$l_1 = \lceil \frac{n}{\log(w)} \rceil, l_2 = \lfloor \frac{\log(l_1(w-1))}{\log(w)} \rfloor + 1.$$

The functions used in WOTS+ are `base_`$w(X,$ $w,$ $len)$, `chain(`$X, i, s,$ `PK.seed, ADRS )`, `WOTS_KeyGen(Sk.seed, Pk.seed, ADRS)`, `WOTS_Sign(`M`, Sk.seed, Pk.seed, ADRS)`, `WOTS_PkFromSig(sig,` M`, Pk.seed, ADRS)`.

`base_w(X, w, len)` is a function that for a given X of a byte length len it returns its base-w value. The details of the function is shown in Algorithm 1.

Algorithm 1 base_w

Require: X, w, len
Ensure: base_w
in, out, total, bits, consumed $\leftarrow 0$
for consumed $= 0, ..., len - 1$ **do**
  if bits==0 **then**
    total $= X$[in]
    in++
    bits+=8
  end if
  bits $-=$ $log(w)$
  `base_w` [out] = (total >> bits) && (w - 1)
  out++
end for
return base_w

`chain(`X, i, s`, PK.seed, ADRS)` function computes an iteration of hash function F on an n-byte input using a WOTS+ hash address `ADRS` and a public seed `PK.seed`. The chain function takes as input an n-byte string X, a starting index i, a number of steps s, and `ADRS` and `PK.seed`. The details of the function is shown in Algorithm 2.

Algorithm 2 chain

Require: X, i, s, `PK.seed`, `ADRS`
Ensure: $Hval$
if $s == 0$ **then**
  return X
end if
if $(i + s) > (w - 1)$ **then**
  return $NULL$
end if
byte[n] $Hval \leftarrow$ `chain`$(X, i, s - 1,$ `Pk.seed`, `ADRS`$)$
$Hval \leftarrow$ **F**(`Pk.seed`, `ADRS`, $Hval$)
return $Hval$

`WOTS_KeyGen(Sk.seed, Pk.seed, ADRS)` function computes a key pair in WOTS+ algorithm. The WOTS+ private key `WOTS.Sk` must not be used to sign more than one message. The public key is the end nodes of the tweakable hash chains. The details of the function is shown in Algorithm 3.

Algorithm 3 WOTS_KeyGen

Require: `Sk.seed`, `Pk.seed`, `ADRS`
Ensure: WOTS.Sk, WOTS.Pk
1: **for** $i = 0, ..., l-1$ **do**
2: `WOTS.Sk`[i] $\leftarrow$ PRF(`Sk.seed`, `ADRS`)
3: tmp[i] $\leftarrow$ `chain`(`WOTS.Sk`[i], 0, $w-1$, `Pk.seed`, `ADRS`)
4: **end for**
5: `WOTS.Pk` $\leftarrow$ $\mathbf{T_l}$(`Pk.seed`, `ADRS`, tmp)
6: **return** `WOTS.Sk`, `WOTS.Pk`

`WOTS_Sign(`M`, SK.seed, PK.seed, ADRS)` function computes the signature in WOTS+. First M is converted to `base-`w form, then the function calculates the checksum for the message. By concatenating checksum to the message, the chain function is repeated as many times as the message and checksum value to call the hash function. The iteration of the hash function uses the chain function. The details of the function is shown in Algorithm 4.

Algorithm 4 WOTS_SigGen

Require: M, Sk.seed, Pk.seed, `ADRS`
Ensure: WOTS.sig
1: msg = `base_` $w(M, w, l_1)$
2: **for** $i = 0, ..., l_1 - 1$ **do**
3: csum = csum+w $-1-$msg[i]
4: **end for**
5: msg $\leftarrow$ msg $\|$ `base_` w(csum,w,l_2)
6: **for** $i = 0, ..., l-1$ **do**
7: sk $\leftarrow$ **PRF**(`Sk.seed`, `ADRS`)
8: WOTS.Sig[i] = `chain`(sk, 0, msg[i], `PK.seed`, `ADRS`)
9: **end for**
10: **return** `WOTS.sig`

`WOTS_PkFromSig(sig, M, Pk.seed, ADRS)` function computes the public key from sign. FIBS uses implicit signature verification for WOTS+. To verify a WOTS+ signature on a message M, the verifier computes a WOTS+ public key value from the signature. The details of the function is shown in Algorithm 5.

Algorithm 5 WOTS_PkFromSig

Require: `WOTS.Sig`, M, `Pk.seed`, `ADRS`
Ensure: `WOTS.Pk`
msg = `base_` $w(M, w, l_1)$
for $i = 0, ..., l_1 - 1$ **do**
  csum = csum+$w - 1-$msg[i]
end for
for $i = 0, ..., l - 1$ **do**
  sk $\leftarrow$ **PRF**(`Sk.seed`)
  tmp[i] = `chain`(sig[i], msg[i],$w - 1-$msg[i], `PK.seed`, `ADRS`)
end for
`WOTS.sig` $\leftarrow$ $\mathbf{T_l}$(`PK.seed`, `ADRS`, tmp)
return `WOTS.Sig`

XMSS The parameter h in XMSS denotes the height of the tree. There are 2^h leaves in the tree. The functions used in XMSS are `treehash(Sk.seed,` s, z, `Pk.seed)`, `XMSS_Sign(`M `, SK.seed,` ind`, PK.seed, ADRS)`, `XMSS_PkFromSig(`ind, `XMSS.Sig`, M , `PK.seed`, `ADRS`).

`treehash(Sk.seed,` s, z, `Pk.seed)` function computes the root node of a Merkle tree of height z(the height) at index s(starting index). In this function use stack function `push()` and `pop()`. The details of the function is shown in Algorithm 6.

Algorithm 6 treehash

Require: `Sk.seed`, s, z,`Pk.seed`
Ensure: Stack.pop()
for $i = 0, ..., 2^z$ **do**
  node = WOTS_KeyGen(`Sk.seed, Pk.seed`, `ADRS`)
  while Top node on Stack has same heigt as node **do**
    node $\leftarrow$ **H**(`PK.seed`, `ADRS`, (Stack.pop()||node))
  end while
  Stack.push(node)
end for
return Stack.pop()

`XMSS_Sign(`M `, SK.seed,` ind`, PK.seed, ADRS)` function compute XMSS signature of the message M according to the WOTS+ index of a key pair ind. This function computes the WOTS+ signature `WOTS.Sig` and the XMSS signature `XMSS.Sig`. The details of the function is shown in Algorithm 7.

Algorithm 7 XMSS_Sign

Require: M, `SK.seed,` ind`, PK.seed, ADRS`
Ensure: XMSS.Sig
1: **for** $j = 0, ..., h-1$ **do**
2: $k = \lfloor \frac{ind}{2^j} \rfloor$
3: AUTH[j]=`treehash(SK.seed,` $k \cdot 2^j$, j, `Pk.seed)`
4: **end for**
5: `WOTS.Sig` ← `WOTS_Sign(`M`, Sk.seed, Pk.seed)`
6: `XMSS.Sig` = `WOTS.Sig` ∥ AUTH
7: **return** `XMSS.Sig`

`XMSS_PkFromSig(`ind`, XMSS.Sig,` M `, PK.seed, ADRS)` function computes the public key of XMSS from `XMSS.Sig`. The public key of XMSS, such as WOTS+, is not explicitly used. In this function, WOTS+ public key is calculated using the `WOTS_PkFromSig` function, and the root node is computed using WOTS+ public key and authentication path in `XMSS.Sig`. The details of the function is shown in Algorithm 8.

Algorithm 8 XMSS_PkFromSig

Require: ind, `XMSS.Sig`, M, `PK.seed`, `ADRS`
Ensure: node[0]
1: `WOTS.Sig` ← Get `WOTS.Sig` from `XMSS.Sig`
2: `AUTH` ← Get `AUTH` from `XMSS.Sig`
3: node[0] ← `WOTS_PkFromSig`(`WOTS.Sig`, M, `Pk.seed`, `ADRS`)
4: **for** $k = 0, ..., h-1$ **do**
5: **if** ($\lfloor \frac{ind}{2^k} \rfloor$ mod 2) == 0 **then**
6: node[1] = **H**(`PK.seed`, `ADRS`, (node[0]∥`AUTH`[k]))
7: **else**
8: node[1] = **H**(`PK.seed`, `ADRS`, (`AUTH`[k]∥node[0]))
9: **end if**
10: node[0] = node[1]
11: **end for**
12: **return** node[0]

FORS In FORS, the parameter k denotes the number of private key sets and t denotes the number of elements per private key set, and t must be a power of 2. The functions used in FORS are `FORS_treehash(SK.seed,` $s, z,$ `PK.seed, ADRS)`, `FORS_Sign(`M`, SK.seed, Pk.seed, ADRS)`, `FORS_PkFromSig(FORS.Sig,` M`, PK.seed, ADRS)`.

`FORS_treehash(SK.seed,` $s, z,$ `PK.seed, ADRS)` computes the root node of a FORS trees of height z at index s. This function is essentially the same as the treehash function (Algorithm 6), although the method of calculating the address value `ADRS`, and secret key is different from that of treehash function. The details of the function is shown in Algorithm9.

Algorithm 9 FORS_treehash

Require: `SK.seed`, s, z, `PK.seed`, `ADRS`
Ensure: Stack.push(node)
1: **for** $i = 0, ..., 2^z$ **do**
2: sk $\leftarrow$ **PRF**(`Sk.seed`, `ADRS`)
3: node $\leftarrow$ **F**(`Pk.seed`, `ADRS`, sk)
4: **while** Top node on Stack has same height as node **do**
5: node $\leftarrow$ **H**(`Pk.seed`, `ADRS`, (Stack.pop() $\|$ node)
6: **end while**
7: Stack.push(node)
8: **end for**
9: **return** Stack.pop()

`FORS_Sign(`M`, SK.seed, Pk.seed, ADRS)` computes the `FORS.Sig`. This function uses the authentication path `AUTH` generated by `fors_treehash` as the part of the signature. The details of the function is shown in Algorithm10.

Algorithm 10 FORS_Sign

Require: M, `SK.seed, Pk.seed`, `ADRS`
Ensure: `FORS.Sig`
1: **for** $i = 1, ..., k$ **do**
2: `FORS.Sig = FORS.Sig` $\|$ **PRF**(`Sk.seed`, `ADRS`)
3: **for** $j = 0, .., a$ **do**
4: s=$\lfloor ind/2^j \rfloor$
5: AUTH[j] = `FORS_treehash(Sk.seed,` $i \cdot k + s \cdot 2^j, j,$ `Pk.seed, ADRS)`
6: **end for**
7: `FORS.Sig` = `FORS.Sig` $\|$ `AUTH`
8: **end for**
9: **return** `FORS.Sig`

`FORS_PkFromSig(FORS.Sig,` M`, PK.seed, ADRS)` function computes the public key from FORS signature `FORS.Sig`. It first computes the root nodes of k

hash trees using `FORS_treehash`, then the roots are hashed. The details of the function is shown in Algorithm 11.

Algorithm 11 FORS_PkFromSig

Require: `FORS.Sig`, M, `PK.seed`, `ADRS`
Ensure: pk
1: sk ← Get sk from `FORS.Sig`
2: AUTH ← Get `AUTH` from `FORS.Sig`
3: **for** $i = 1, ..., k$ **do**
4: node[0] ← **F**(`Sk.seed`, `ADRS`, sk)
5: **for** $j = 0, .., a$ **do**
6: **if** $(\lfloor \frac{ind}{2^j} \rfloor \mod 2) == 0$ **then**
7: node[1] = **H**(`Pk.seed`, `ADRS`, (node[0]‖ `AUTH`[j]))
8: **else**
9: node[1] = **H**(`PK.seed`, `ADRS`,(AUTH[j] ‖ node[0]))
10: **end if**
11: node[0] = node[1]
12: **end for**
13: root[i] = node[0]
14: **end for**
15: pk = $\mathbf{T_k}$(`PK.seed`, `ADRS`, root)
16: **return** pk

FIBS Key generation The FIBS private key consists of two elements. The n-byte secret seed `Sk.seed`, which is used to generate all WOTS+ and FORS private key elements, and the n-byte pseudorandom function key `Sk.prf`, which is used to generate random values for random message hashes. The FIBS public key contains two elements. The XMSS public key, i.e. the root of the tree at the top level, `Pk.root`, and a randomly sampled n-byte public seed value `Pk.seed`, which generate random value used WOTS+ and hyper-tree. The details of the function is shown in Algorithm 12.

Algorithm 12 FIBS_KeyGen

Require:
Ensure: `Pk.root, Pk.seed, Sk.seed,Sk.prf`
1: Sk.seed $\xleftarrow{\$}$ $\{0,1\}^n$
2: Sk.prf $\xleftarrow{\$}$ $\{0,1\}^n$
3: Pk.seed $\xleftarrow{\$}$ $\{0,1\}^n$
4: **for** $i = 1, ..., 2^{h/d}$ **do**
5: `WOTS.Sk`[i], `WOTS.Pk`[i] ← `WOTS_KeyGen`(`Sk.seed, Pk.seed`, `ADRS`)
6: **end for**
7: `Pk.root` ← `XMSS_PkGen`(`SK.seed, PK.seed`, `ADRS`)
8: **return** `Pk.root, Pk.seed, Sk.seed, Sk.prf`

FIBS Signature Generation algorithm The process of the FIBS signature algorithm is as follows: First, it generates a random value R using the message and `Sk.prf` as the private key. These values are included in the signature. Then, it creates a message digest of R, `Pk.seed`, and `Pk.root`, using the $\mathbf{H_{msg}}$ function. The message digest is truncated to a certain length by using the function `Split_md`. The outputs to the function `Split_md` are md, ind.tree, and ind.leaf. md is later used as a input to the FORS signature. ind.tree indicates the number of trees to be signed, and ind.leaf indicates the number of leaves to be signed. Through this step, the algorithm generates a random index for the message. After a signature FORS is generated using md, `Sk.seed`, and `Pk.seed` values, the algorithm repeats the WOTS+ signature and hypertree operations for the signature value of FORS. The details of the function is shown in Algorithm13.

Algorithm 13 FIBS Signature Generation

Require: M, `Sk.seed, Sk.prf, Pk.seed, Pk.root`
Ensure: FIBS.Sig

OptRand $\xleftarrow{\$} \{0,1\}^n$
$R \leftarrow \mathbf{PRF_{msg}}$(`Sk.prf`, OptRand, M)
FIBS.Sig = FIBS.Sig $\|$ R
Digest $\leftarrow \mathbf{H_{msg}}(R$, `Pk.seed, Pk.root`, M)
md, ind.tree, ind.leaf $\leftarrow$ `Split_md`(Digest)
`FORS.Sign` $\leftarrow$ `FORS_Sign`(md, `Sk.seed, Pk.seed`, `ADRS`)
FIBS.Sig = FIBS.Sig $\|$ FORS.Sign
`FORS.Sign` $\leftarrow$ `FORS_PkFromSig`(`FORS.Sign`, M, `Pk.seed`, `ADRS`)
Sign.tmp $\leftarrow$ `XMSS_Sign`(`FORS.Sign, SK.seed`, ind.leaf, `PK.seed`, `ADRS`)
HT.Sig $\leftarrow$ HT.Sig $\|$ Sign.tmp
root $\leftarrow$ `XMSS_PkFromSig`(ind.leaf, Sig.tmp, M, `Pk.seed`, `ADRS`)
for $j = 1, ..., d-1$ **do**
  Sig.tmp = `XMSS_Sign`(root, `Sk.seed`, ind.leaf, `Pk.seed`, `ADRS`)
  HT.Sig = HT.Sig $\|$ Sig.tmp
  if $j < d-1$ **then**
    root = `XMSS_PkFromSig`(ind.leaf, Sig.tmp, root, `PK.seed`, `ADRS`)
  end if
end for
FIBS.Sig = FIBS.Sig $\|$ HT.sig
return FIBS.Sig

FIBS Verification algorithm FIBS signature verification algorithm proceeds as follows: First, it recomputes the message digest and index, and computes a FORS public key. Then it verifies the XMSS tree that public key. The details of the function is shown in Algorithm14.

Algorithm 14 FIBS Signature Verification

Require: M, FIBS.Sig, pk
Ensure: 0 or 1
1: $R \leftarrow$ Get R from FIBS.Sig
2: `FORS.Sig` $\leftarrow$ Get `FORS.Sig` from FIBS.Sig
3: HT.Sig $\leftarrow$ Get HT.Sig from FIBS.Sig
4: Sig.tmp $\leftarrow$ Get Sig.tmp from HT.sig
5: Digest $\leftarrow \mathbf{H_{msg}}(R$, `Pk.seed, Pk.root`, $M)$
6: md, ind.tree, ind.leaf $\leftarrow$ Split_md(Digest)
7: `FORS.Pk` $\leftarrow$ `FORS_PkFromSig`(`FORS.Sig`, md, `Pk.seed`, `ADRS`)
8: node $\leftarrow$ `XMSS_PkFromSig`(ind.leaf, Sig.tmp, M, `Pk.seed`, `ADRS`)
9: **for** $j = 1, ..., d-1$ **do**
10: node $\leftarrow$ `XMSS_PkFromSig`(ind.leaf, Sig.tmp, node, `Pk.seed`, `ADRS`)
11: **end for**
12: **if** node == `PK.root` **then**
13: **return** true
14: **else**
15: **return** false
16: **end if**

4 Performance analysis

4.1 Parameter sets

Parameter for the CGL hash function For the CGL hash function, we use the 607-bit prime $p = 2^{607} - 1$. Over finite field $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ for $i^2 = -1$, we used the supersingular Montgomery curve of the form as the base curve:

$$M : y^2 = x^3 + x.$$

Let P, Q be the generator of $E[2^{607}]$. Then the x-coordinate of P, Q is denoted as Px, Qx which is as follows:

$$Px = 5 + i$$
$$Qx = 15 + i$$

Parameter for FIBS Using the parameters for the CGL hash function as defined above, FIBS uses the following parameters for NIST security level 1.

Table 3: FIBS parameter for NIST security level 1

	n	h	d	$\log t$	k	w
FIBS_128	16	66	22	6	33	16

4.2 Performance

We evaluate our reference implementation on the Intel Core i9-10980XE running Ubuntu 20.04.5 LTS. For our parameter, the resulting cycle counts are listed below.

Table 4: Key and signature size in bytes

	Public Key	Private Key	Signatrue
FIBS_128	32	64	17,088

Table 5: Performance results of FIBS

	Keygen	Sig. Gen.	Sig. Ver.
FIBS_128	121.66 s	2837.04 s	172.37 s

5 Security Evaluation

The SPHINCS+ security is based on the assumption that the PRF used within the instantiation of the tweakable hash function to generate the bitmask and the standard properties of the function family used can be modeled as a random oracle.

In this section, we give a security analysis for FIBS based on the above claim. The security reduction of SPHINCS+ is described [1,3]. We explain the security reduction for FIBS based on SPHINCS+. In reduced security, we assume that each call for a hash function used to instantiate an adjustable hash is given a different value and is XORed with a bitmask before the input is processed and assume that a bitmask is created using PRF called PRF_{BM}. We assume the following statistical properties of the hash function F. The CGL Hash function used in FIBS is F, and all elements of the image must have at least two preimages, i.e,

$$(\forall k \in \{0,1\}^n)(\forall y \in \mathrm{IMG}(\mathrm{F}_k))(\exists x, x' \in \{0,1\}^n) : x \neq x' \wedge \mathrm{F}_k(x) = f_k(x'). \quad (1)$$

We will prove the following theorem, where F,H, and T.

Theorem 1. *For security parameter $n \in \mathbb{N}$, parameters w, h, d, m, t, k as described above, FIBS is existentially unforgeable under post-quantum adaptive chosen message attacks if*

- *F, H, and T are post-quantum distinct-function multi-target second-preimage resistant function families,*
- *F fulfills the requirement of equation 1,*
- *PRF, PRF_{msg} are post-quantum pseudorandom function families,*
- *PRF_{BM} is modeled as a quantum-accessible random oracle,*
- *H_{msg} is a post-quantum interleaved target subset resilient hash function family.*

The security function $InSec^{PQ-EU-CMA}(FIBS;\xi, 2^h)$ describing the maximum success probability over all adversaries running in time $\leq \xi$ against the PQ-EU-CMA security of FIBS is bounded by

$$\begin{aligned} &InSec^{PQ-EU-CMA}(FIBS;\xi) \leq \\ &2(InSec^{PQ-PRF}(PRF;\xi) + InSec^{PQ-PRF}(PRFmsg;\xi) \\ &+ InSec^{pq-itsr}(H_{msg};\xi) + InSec^{PQ-DM-SPR}(F;\xi) + \\ &InSec^{PQ-DM-SPR}(H;\xi) + InSec^{PQ-DM-SPR}(T;\xi)) \end{aligned} \tag{2}$$

5.1 Considering Security by PQ-DM-SPR, PQ-itsr

SPHINCS+ defines two properties to bounded the success probability of adversary $\mathcal{A}$ for PQ-EU-CMA security in SPHINCS+ [1]. The first is a variant of post-quantum multi-function multi-target second-preimage resistance called post-quantum distinct-function multi-target second-preimage resistance, and the second is a variant of subset-resilience which captures the use of FORS in SPHINCS+ called (post-quantum) interleaved target subset resilience. FIBS also proves this way as we bounded the probability of success for PQ-EU-CMA security in SPHINCS+.

5.2 Security Against Generic Attacks

The security of FIBS depends on the properties of the functions used to hash functions. In this section, we consider only generic attacks on the assumption that the CGL hash function used in FIBS is structurally secure.

Distinct-Function Multi-Target Second Preimage Resistance When evaluating the complexity of a generic attack on a hash function, the hash function is usually modeled as a random function family. As shown in [16] it was shown

that the success probability of any classical q_{hash}-query adversary against multi-function multi-target second-preimage resistance of a random function of range $\{0,1\}^{8n}$ is exactly $q_{hash}+1/2^{8n}$.For q_{hash}-query quantum adversaries the success probability is $\Theta((q_{hash}+1)^2/2^{8n})$.

Pseudorandomness of Function Families An exhaustive search is generally considered as the most common attack on pseudorandomness of the function family. According to [16], the probability of success of classical adversaries evaluating function family for function with key space $\{0,1\}^{8n}$ in q_{key} is bounded by $q_{key}+1/2^{8n}$. For q_{key}-query quantum adversaries, the probability of success of the exhaustive search is $\Theta((q_{key}+1)^2/2^{8n})$.

Interleaved Target Subset Resilience The interleaved target subset resilience for FORS is described in [1]. Assume that the used hash function is a random function. For any classical adversary which makes q_{hash} queries to function family $\mathcal{H}_n$ the success probability is

$$(q_{hash}+1)\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}$$

For q_{hash}-query quantum adversaries the success probability is

$$\mathcal{O}\left((q_{hash}+1)^2\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right)$$

5.3 Security Level

Let q denote the number of adversarial signature queries. For classical adversaries that make no more than q_{hash} queries to the cryptographic hash function used, this leads to

$$\begin{aligned}
&InSec^{PQ-EU-CMA}(FIBS;q_{hash}) \leq \\
&2(\frac{q_{hash}+1}{2^{8n}}+\frac{q_{hash}+1}{2^{8n}} \\
&+InSec^{pq-itsr}(H_{msg};q_{hash})+\frac{q_{hash}+1}{2^{8n}}+\frac{q_{hash}+1}{2^{8n}}+\frac{q_{hash}+1}{2^{8n}}) \\
&=10\frac{q_{hash}+1}{2^{8n}}+2\,(q_{hash}+1)\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}} \\
&=\mathcal{O}\left(\frac{q_{hash}}{2^{8n}}+(q_{hash})\sum_{\gamma}\left(1-\left(1-\frac{1}{t}\right)^{\gamma}\right)^{k}\binom{q}{\gamma}\left(1-\frac{1}{2^h}\right)^{q-\gamma}\frac{1}{2^{h\gamma}}\right)
\end{aligned}$$

For quantum adversaries that make no more than q_{hash} queries to the cryptographic hash function used, this leads to

$$
\begin{aligned}
&InSec^{PQ-EU-CMA}(FIBS; q_{hash}) \leq \\
&2(\frac{(q_{hash}+1)^2}{2^{8n}} + \frac{(q_{hash}+1)^2}{2^{8n}} \\
&+ InSec^{pq-itsr}(H_{msg}; q_{hash}) + \frac{(q_{hash}+1)^2}{2^{8n}} + \frac{(q_{hash}+1)^2}{2^{8n}} + \frac{(q_{hash}+1)^2}{2^{8n}}) \\
&= 10\frac{(q_{hash}+1)^2}{2^{8n}} + 2(q_{hash}+1)^2 \sum_{\gamma} \left(1 - \left(1 - \frac{1}{t}\right)^{\gamma}\right)^{k} \binom{q}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q-\gamma} \frac{1}{2^{h\gamma}} \\
&= \mathcal{O}\left(\frac{(q_{hash})^2}{2^{8n}} + (q_{hash})^2 \sum_{\gamma} \left(1 - \left(1 - \frac{1}{t}\right)^{\gamma}\right)^{k} \binom{q}{\gamma} \left(1 - \frac{1}{2^h}\right)^{q-\gamma} \frac{1}{2^{h\gamma}}\right)
\end{aligned}
$$

6 Security of CGL hash

In the previous sections, we assume that the used CGL hash function has preimage-resistant and collision-resistant properties. Recently, it turned out that there exists a efficient attack to construct a collision pair of CGL hash using the KLTP algorithm if the endomorphism ring of a starting elliptic curve is known [18]. In such cases our proposed scheme is insecure, therefore we require that either the endomorphism ring of the starting elliptic curve is unknown or there exists a secure way of computing the isogeny chain defending against such attacks. Currently, there is no known algorithm to generate a random supersingular elliptic curve with an unknown endomorphism ring from a random seed. So if we choose this setting we need to assume a trusted third party that provides such elliptic curves securely. However, it is clearly undesirable to use a domain parameter with such a trapdoor in it. So we do not assume unawareness of the endomorphism ring. L. Panny proposed a secure way of computing CGL hash despite known endomorphisms [17]. We rewrite the proposals of [17] here. Panny's method proposes to choose only $r(< \ell)$ isogenies from ℓ possibilities at each step of computing an isogeny chain. Then if the length of the chain is n, the probability that an attacker can transform a given random isogeny cycle into two colliding CGL hash function inputs is approximately bound by

$$n \cdot \left(\frac{r}{\ell}\right)^n$$

If we assume $n \approx \log_\ell p$ then taking the cost of finding cycle by KLTP algorithm as $\Omega(\log p) \geq n$, the estimation of total attacking cost is

$$\left(\frac{r}{\ell}\right)^n \approx \left(\frac{r}{\ell}\right)^{\log_\ell p} = \left(\ell^{1-\log_\ell r}\right)^{\log_\ell p} = p^{1-\log_\ell r}$$

For a target security level λ, there is a trade-off curve between the size of p and the relative size of ℓ and r. If we take $r^2 < \ell$, the cycle finding attack is

infeasible. Therefore, to protect the CGL hash function against both the Petit-Lauter attack and generic collision finding attack, the choice of p and $\log_\ell p$ should satisfy

$$\log_2 p \geq \max\left\{2\lambda, \frac{\lambda}{1-\log_\ell r}\right\} \tag{3}$$

7 Optimizing parameter choices

We rewrite the discussion about optimizing parameters of CGL hash of [17]. For the performance of the CGL hash function, we need to choose smaller p and larger $\log_\ell r$ as possible. For the efficient computation of isogeny, $p = \ell^n \cdot f - 1$ with $\ell = 2$ where f is a small cofactor is a popular choice. With the starting supersingular curve $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + x$, the ℓ^n-isogeny can be evaluated efficiently using tree-based strategy using $O(\log\log n)$ operations. For a fixed choice of $\log p$, ℓ, and r, the cost per bit of evaluating CGL hash function is scaled as

$$(\log p)^2(n\log n)/\log r \approx (\log p)^2(\log\log p)/\log_\ell r$$

The optimization is to minimize this function under the condition (3).

λ	p	ℓ	r
128	$2^{256}\cdot 45 - 1$	2^{256}	2^{128}
192	$2^{291}\cdot 3 - 1$	2^{390}	2^{195}
256	$2^{512}\cdot 243 - 1$	2^{512}	2^{256}

Table 6: Optimal choices of p, ℓ, and r for security level λ

References

1. Aumasson, J.P., Bernstein, D.J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Hülsing, A., Kampanakis, P., Kölbl, S., Lange, T., et al.: Sphincs+– submission to the 2nd round of the nist post-quantum project. specification document (part of the submission package) (2019)
2. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: Sphincs: practical stateless hash-based signatures. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 368–397. Springer (2015)
3. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs+ signature framework. Cryptology ePrint Archive, Paper 2019/1086 (2019). https://doi.org/10.1145/3319535.3363229, https://eprint.iacr.org/2019/1086, https://eprint.iacr.org/2019/1086
4. Beullens, W., Kleinjung, T., Vercauteren, F.: CSI-FiSh: efficient isogeny based signatures through class group computations. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 227–247. Springer (2019)
5. Buchmann, J., Dahmen, E., Hülsing, A.: Xmss - a practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.Y. (ed.) Post-Quantum Cryptography. pp. 117–129. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
6. Castryck, W., Decru, T.: An efficient key recovery attack on sidh (preliminary version). Cryptology ePrint Archive (2022)
7. Charles, D.X., Lauter, K.E., Goren, E.Z.: Cryptographic hash functions from expander graphs. Journal of CRYPTOLOGY **22**(1), 93–113 (2009)
8. Chávez-Saab, J., Chi-Domínguez, J.J., Jaques, S., Rodríguez-Henríquez, F.: The sqale of csidh: sublinear vélu quantum-resistant isogeny action with low exponents. Journal of Cryptographic Engineering **12**(3), 349–368 (2022)
9. Couveignes, J.M.: Hard homogeneous spaces. IACR Cryptology ePrint Archive **2006**, 291 (2006)
10. Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital signatures out of second-preimage resistant hash functions. In: Buchmann, J., Ding, J. (eds.) Post-Quantum Cryptography. pp. 109–123. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
11. De Feo, L., Galbraith, S.D.: Seasign: compact isogeny signatures from class group actions. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 759–789. Springer (2019)
12. De Feo, L., Kieffer, J., Smith, B.: Towards practical key exchange from ordinary isogeny graphs. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 365–394. Springer (2018)
13. De Feo, L., Kohel, D., Leroux, A., Petit, C., Wesolowski, B.: Sqisign: compact post-quantum signatures from quaternions and isogenies. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 64–93. Springer (2020)
14. Galbraith, S.D., Petit, C., Silva, J.: Identification protocols and signature schemes based on supersingular isogeny problems. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 3–33. Springer (2017)

15. Hülsing, A.: W-ots+ – shorter signatures for hash-based signature schemes. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) Progress in Cryptology – AFRICACRYPT 2013. pp. 173–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
16. Hülsing, A., Rijneveld, J., Song, F.: Mitigating multi-target attacks in hash-based signatures. In: Cheng, C.M., Chung, K.M., Persiano, G., Yang, B.Y. (eds.) Public-Key Cryptography – PKC 2016. pp. 387–416. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
17. Panny, L.: Isogeny-based hashing despite known endomorphisms. Cryptology ePrint Archive (2019)
18. Petit, C., Lauter, K.: Hard and easy problems for supersingular isogeny graphs. Cryptology ePrint Archive (2017)
19. Yoo, Y., Azarderakhsh, R., Jalali, A., Jao, D., Soukharev, V.: A post-quantum digital signature scheme based on supersingular isogenies. In: International Conference on Financial Cryptography and Data Security. pp. 163–181. Springer (2017)