The Picnic Signature Algorithm

Specification

Contact: Greg Zaverucha (gregz@microsoft.com)

April 15, 2020
Version 3.0

Contents

1 Introduction

This document specifies the Picnic public-key digital signature algorithm. It also describes cryptographic primitives used to construct Picnic, and methods for serializing signatures and public keys.

Picnic is designed to provide security against attacks by quantum computers, in addition to attacks by classical computers. The building blocks are a zero-knowledge proof system (with post-quantum security), and symmetric key primitives like hash functions and block ciphers, with well-understood post-quantum security. In particular, Picnic *does not* require number-theoretic, or structured hardness assumptions.

1.1 Overview of the Picnic Signature Algorithm

This section gives a very brief overview of the Picnic design. For a detailed description and a complete list of references to related work see [CDG+17a, KKW18], and the additional documentation submitted to the NIST Post-Quantum Standardization process. A reference implementation is available at `https://github.com/Microsoft/Picnic`.

The public key in Picnic is the pair (C, p) where $C = E(sk, p)$, and where E is a block cipher, sk a secret key and p is a plaintext block. The block cipher E is LowMC [ARS+16, ARS+15]. To create a signature, the signer creates a non-interactive proof of knowledge of sk, and binds the proof with the message to be signed. LowMC was chosen because the resulting signature size is smaller than alternative choices.

The proof of knowledge is either a specialized version of ZKBoo [GMO16], called ZKB++ [CDG+17b], or the proof from [KKW18]. Informally, in the interactive version of either proof system, the prover simulates a multiparty computation protocol (MPC protocol) that allows parties to jointly verify that $E(sk, p) = C$, when each party has a share of sk. For Picnic, the number of parties is a parameter. The idea is to have the prover commit to the simulated state and transcripts of all parties, then have the verifier open a random subset of the simulated parties by seeing their complete state. The verifier then checks that the computation was done correctly from the perspective of the opened parties, and if so, he has some assurance that the output is correct. The MPC protocol ensures that opening a subset of the parties does not reveal information about the secret. Iterating this process multiple times in parallel gives the verifier high assurance that the prover knows the secret.

To make the proof non-interactive there are two options. The Fiat-Shamir transform (FS) yields a signature scheme that is secure in the random oracle model (ROM), whereas the Unruh transform (UR), yields a signature scheme that is secure in the quantum ROM (QROM). The UR signatures are larger, however. We note that while the QROM security analysis of UR signatures is in a stronger formal model with respect to quantum attacks, there are no known quantum attacks on the FS transform. We specify the ZKB++ proofs with the FS and UR transforms, and the [KKW18] proofs with the FS transform.

1.2 Contributors

The Picnic signature algorithm was designed by the following team.
Melissa Chase, Microsoft
David Derler, DFINITY
Steven Goldfeder, Princeton
Daniel Kales, Graz University of Technology
Jonathan Katz, University of Maryland
Vladimir Kolesnikov, Georgia Tech
Claudio Orlandi, Aarhus University
Sebastian Ramacher, Graz University of Technology
Christian Rechberger, Graz University of Technology & DTU
Daniel Slamanig, AIT Austrian Institute of Technology
Xiao Wang, Northwestern University
Greg Zaverucha, Microsoft

Acknowledgments We are grateful for help from Itai Dinur and Niv Nadler, who identified an attack on an earlier version of this spec, when input used to derive random tapes did not use a salt value. We also acknowledge Alexander Grass for pointing out inconsistencies between the spec and implementation.

2 Notation

This section describes the notation used in this document. In addition to the notation in Table 1, the notation `vec[0..2]` denotes a vector of three elements: `vec[0]`, `vec[1]`, `vec[2]`. When `vec` is used without an index it refers to the entire vector. All indexing is zero-based.

S	The expected security strength in bits (against classical attacks).
n	The LowMC key and blocksize, in bits.
s	The LowMC number of s-boxes.
r	The LowMC number of rounds.
KDF	A key derivation function (defined in 3.3).
H	A hash function.
T	Number of parallel repetitions required for soundness of the proof of knowledge.
u	Number of challenged repetitions.
ℓ_H	The output length of H, in bytes.
$\oplus$	the binary exclusive or (XOR) of equal-length bitstrings.
N	The number of parties in the simulated MPC protocol.

Table 1: Notation used in this document.

3 Cryptographic Components

This section describes the cryptographic components that are used in the Picnic algorithm.

3.1 LowMC

Signing and verification compute the LowMC circuit, as part of the non-interactive MPC protocol simulation. The signing and verification algorithms specified here include sufficient detail to implement LowMC. However, implementations need some constants that are part of the LowMC definition. These parameters are different for each of the three LowMC parameter sets in Table 2.

`Kmatrix` an array of $n \times n$ binary matrices, one for initial whitening, and one for each LowMC round ($r + 1$ in total)

`Lmatrix` an array of $n \times n$ binary matrices, one for each LowMC round (r in total)

`roundconstant` an array of n-bit vectors, one for each LowMC round

We use the LowMC constants from the LowMC reference implementation [Tie17], without modification. These are included in the Picnic reference implementation, in the header file `lowmc_constants.c`.

3.2 Hash functions

The hash functions in this specification are all based on the SHAKE128 or SHAKE256 SHA-3 functions [NIS15] that have variable output length. In this document when we write H, this is the SHAKE function given in Table 2 with the fixed output length also specified in Table 2.

There are multiple hashing operations when computing signatures, once to compute commitments, once to compute the challenge, (optionally) when computing a second type of commitment, and when using a seed value in multiple places. We prepend a fixed byte to the input of H in order to differentiate hash inputs in different uses. Define $H_i(x) = H(\texttt{0x0}i \| x)$, for $i = 0, \ldots, 5$. When computing commitments we use H_0 and when computing the challenge we use H_1. The UR parameter sets also use H when computing the function G (defined in Section 6.4.6), and here we use H_2. Before each use of a seed value, used in multiple places, we hash it before use with H_2, H_4, or H_5.

3.3 Key Derivation Functions

When creating and verifying signatures we must expand a short random value (128 to 512 bits) called the seed, into a longer one (about 1KB). This is done with an extendable-output function (XOF), based on SHA3, called SHAKE [NIS15]. This choice allows a single function family (SHA3) for both hashing and key derivation, as SHAKE with a fixed output length is also a secure hash function. At security level 1 we use SHAKE128 and security levels 3 and 5 we use SHAKE256. In this specification all calls to the KDF specify the complete input as a bitstring, i.e., additional values such as the context, label and output length, must be encoded as described here, and passed to the XOF as a single input.

4 Picnic Parameters

This section describes the parameter sets for Picnic.

Table 2 gives parameters for three security levels L1, L3 and L5, as described in [Nat16], corresponding to the security of AES-128, AES-192 and AES-256 (respectively). For each of the three security levels there are two signature algorithms that use the ZKB++ proof system, one based on the Fiat-Shamir transform (FS): `picnic-L1-FS`, `picnic-L3-FS` and `picnic-L5-FS`, and one based on the Unruh (UR) transform: `picnic-L1-UR`, `picnic-L3-UR`, and `picnic-L5-UR`. For discussion of the differences between the FS and UR variants, see [CDG+17a]. There are also three parameter sets that use the FS transform and the proof system from [KKW18]: `picnic3-L1`, `picnic3-L3` and `picnic3-L5`. Version 2 of this spec included the `picnic2` parameter sets, which were replaced by `picnic3` in Version 3 of the spec, see Appendix A. Version 3 also adds the parameter sets `picnic-L1-full`, `picnic-L3-full` and `picnic-L5-full`, which are nearly the same as the `-FS` variants, but use the same LowMC instances as the `picnic3` parameters.

All parameters are chosen such that they are expected to provide S bits of security against classical attacks, and at least $S/2$ bits of security against quantum attacks.

The parameter u, the number of challenged repetitions, is only applicable to the `picnic3` parameter sets. The parameter N, the number of parties in the MPC simulation is always 3 for `picnic` parameter sets, and always 16 for `picnic3` parameter sets.

Parameter Set	S	n	s	r	Hash/KDF	ℓ_H	T	u
`picnic-L1-FS`	128	128	10	20	SHAKE128	256	219	
`picnic-L1-UR`					SHAKE128	256	219	
`picnic-L3-FS`	192	192	10	30	SHAKE256	384	329	
`picnic-L3-UR`					SHAKE256	384	329	
`picnic-L5-FS`	256	256	10	38	SHAKE256	512	438	
`picnic-L5-UR`					SHAKE256	512	438	
`picnic3-L1`	128	129	43	4	SHAKE128	256	250	36
`picnic-L1-full`					SHAKE128	256	219	
`picnic3-L3`	192	192	64	4	SHAKE256	384	419	52
`picnic-L3-full`					SHAKE256	384	329	
`picnic3-L5`	255	255	85	4	SHAKE256	512	601	68
`picnic-L5-full`					SHAKE256	512	438	

Table 2: Parameters sets. The first half of the table use LowMC parameters with a partial S-box layer, and the second half use LowMC parameters with a full S-box layer.

5 Key Generation

This section describes how to generate a signing key pair. The process is the same for `picnic` and `picnic3` parameter sets, and differs only based on the LowMC parameters. The public key is denoted $pk = (C, p)$ and the secret key is denoted sk. The input to key generation is one of the parameter sets (e.g, `picnic-L1-FS`). Note

Parameter Set	Public key	Private key	Sig (max)	Sig (avg., std. dev.)
`picnic-L1-FS`	32	16	34032	32838, 107
`picnic-L1-UR`	32	16	53961	
`picnic-L1-full`	34	17	320161	30809, 119
`picnic3-L1`	34	17	13802	12359, 213
`picnic-L3-FS`	48	24	76772	74134, 198
`picnic-L3-UR`	48	24	121845	
`picnic-L3-full`	48	24	71179	68493, 215
`picnic3-L3`	48	24	29750	27173, 443
`picnic-L5-FS`	64	32	132856	128176, 315
`picnic-L5-UR`	64	32	209506	
`picnic-L5-full`	64	32	126286	121616, 269
`picnic3-L5`	64	32	54732	46282, 613

Table 3: Key and signature sizes (in bytes) by security level. For all but the UR variants, the signature length varies based on the challenge, therefore we give the maximum possible size, along with the average size and standard deviation computed over 100 signatures.

that for a key pair of security level S it is technically possible to use it with multiple signature algorithms defined at level S (e.g., a key pair created with $S = 128$ will work with `picnic-L1-FS` and `picnic-L1-UR`). It is not recommended to use a key pair with multiple signature algorithms.

1. Choose a random n-bit string p, and a random n-bit string sk.
2. Using LowMC with the parameters given in Table 2, compute the encryption of p with sk, denoted $C = E(sk, p)$.
3. Output: The pair (sk, pk). The secret key is sk, and the public key pk is (C, p).

5.1 Serialization of Picnic Keys

A Picnic public key (C, p) should be serialized as the bits of C, followed by the bits of p. Both are first converted to byte arrays. When n (the length of the bitstrings C and p) is not a multiple of 8, the unused bits in the last byte must be padded with zero bits. For a given parameter set, public keys can therefore be unambiguously parsed. Note that the length of a serialized public key uniquely identifies the security level, but not the exact parameter set, e.g., public keys for both `Picnic-L1-FS` and

`Picnic-L1-UR` have the same length. Applications that handle multiple parameter sets are responsible for encoding the parameter set along with the public key.

Serializing the private key is done by serializing the n bits of sk, as a byte array (also zero-padded to the nearest byte, if required). As with public keys, the length of the private key identifies the security level, but not the parameter set. Applications working with private keys for multiple parameter sets must also serialize the parameter set.

Deserialization of public keys When deserializing a public key when $n = 255$ or $n = 129$, the public key values (C, p) are padded with 1 or 7 padding bits (respectively). Implementations must ensure that the padding bits are 0, and must reject keys with any nonzero padding bits.

6 Signing and Verification for `picnic` Parameter Sets

This section specifies the signing and verification operations for the six parameter sets: `picnic-L1-FS`, `picnic-L1-UR`, `picnic-L3-FS`, `picnic-L3-UR`, `picnic-L5-FS`, `picnic-L5-UR`, `picnic-L1-full`, `picnic-L3-full`, and `pincic-L5-full`.

6.1 Views

Signing and verification must compute the views of the three parties in the MPC protocol simulation. An individual view object has three components

`view.iShare` The input key share of this party, n bits long.

`view.transcript` The transcript of all communication during the protocol. The length of this depends on the number of AND gates in the LowMC instance being used. In particular, the number of AND gates is $3rs$, so the length of the transcript is the number of bytes required to store $3rs$ bits.

`view.oShare` The output share of this party, n bits long.

Views must be serialized as the simple concatenation of the above three values when serialized to compute commitments. In the `UR` variants we also compute additional commitments with the function G. The input to G includes the input share only if the view is index 2 (corresponding to the third party) followed by the transcript, and not the output share. When n is not a whole number of bytes, values are zero-padded to the next byte.

6.2 Signing Operation

The functions `matrix_mul`, `mpc_sbox`, `mpc_xor`, `mpc_and` and `H3` used to specify sign are specified in later sections (Sections 6.4.4, 6.4.1, 6.4.3, 6.4.2 and 6.4.5 resp.). The description of signature generation is independent of the security level, but changes for the signature algorithms using the Unruh transform: `picnic-L1-UR`, `picnic-L3-UR` and `picnic-L5-UR`. The description below is with respect to a fixed security parameter, and the flag *UR* indicates whether the Unruh transform is used.

Input: Signer's key pair (sk, pk), a message to be signed the byte array M, such that $1 \leq |M| \leq 2^{55}$.

Output: Signature on M as a byte array.

1. Initialize a list of triples of views `views[0..T-1][0..2]`, a list of commitments `C[0..T-1][0..2]` (byte arrays, each of length ℓ_H), a list of seeds `seeds[0..T-1][0..2]`, and a 256-bit salt value *salt*. If *UR* is set, initialize a list of commitments `G[0..T-1][0..2]` (byte arrays of variable length, not exceeding the length of a view, including the input share. See Step 3d below.).

2. Populate `seeds` with $3T$ random seeds, each of length S bits, and set *salt* to a random 256-bit value. It is recommended that these be derived deterministically, by calling the KDF in Table 2, with input

   $$sk \| M \| pk \| S$$

   where S is encoded as a 16-bit little endian integer. The number of bytes requested is $(3T)(S/8) + 32$ (three seeds for each of T iterations, each of size $S/8$ bytes, and one salt value of size 32 bytes).

   The test vectors associated with this document will use this method to simplify testing. However, the specific method of generating `seed` and *salt* does not affect interoperability, and implementations may differ (e.g., by choosing the values uniformly at random, using an alternative derivation method, or including alternative inputs to derivation). For implementations seeking to randomize the signature function, it is recommended to use the derivation described here, but to append a $2S$-bit random value to the KDF input.

3. For each parallel iteration t from 0 to $T-1$:

   (a) Create three random tapes, denoted `rand[0..2]`, using the KDF specified in Table 2, and the input seeds from Step 2, as follows

   $$\texttt{rand[}j\texttt{]} = \text{KDF}(H_2(\texttt{seed[}t\texttt{][}j\texttt{]})\|salt\|t\|j\|\text{output\_length})$$

   The integers output_length, t and j are encoded as 16-bit little-endian integers. Tape `rand[0]` and `rand[1]` have output_length $n + 3rs$ bits, and tape `rand[2]` has length $3rs$ bits. When output_length is not a whole number of bytes, it is rounded up to the next byte. We use the notation `rand[i].nextBit()` to read the next bit of the tape.

   (b) Compute three shares of sk, denoted `x[0..2]`, each of length n bits:

      i. `x[0]` = first n bits of tape `rand[0]`

      ii. `x[1]` = first n bits of tape `rand[1]`

      iii. `x[2]` = $sk \oplus$ `x[0]` $\oplus$ `x[1]`

   (c) Simulate the MPC protocol to compute the LowMC encrypt circuit, recording the views of the three players. Let `state[0..2]`, be a triple of n-bit vectors.

      i. Compute the initial key shares, and whitening:
      `key = matrix_mul(x, Kmatrix[0])`

      ii. XOR the round key with p, the plaintext portion of the public key (C, p).
      `state = mpc_xor_constant(key, p)`

      iii. For each LowMC round `i` from 1 to r

         A. Compute the round `i` key shares:
         `key = matrix_mul(x, Kmatrix[i])`
         The function `matrix_mul` is defined in Section 6.4.4.

         B. Apply substitution layer (s-boxes) to `state`:
         `state = mpc_sbox(state, rand, views[t])`
         The function `mpc_sbox` is defined in Section 6.4.1.

         C. Apply affine layer to `state`:
         `state = matrix_mul(state, Lmatrix[i-1])`

D. Update the state with the XOR of the round constant and the state:
`state = mpc_xor_constant(state, roundconstant[i-1])`
The function `mpc_xor_constant` is defined in Section 6.4.3.

E. Update the state with the XOR of the round key and the state:
`state = mpc_xor(state, key)`

iv. Store the output shares in the views, for `i` from 0 to 2:
`views[t][i].oShare = state[i]`

(d) Form commitments `C[t][0..2]`. For `i` from 0 to 2:
`C[t][i] =` H_0`(`H_4`(seed[i]), view[i])`
If the flag UR is set, for `i` from 0 to 2, compute:
`G[t][i]`$=G($$H_4$`(seed[i]), view[i]`$)$
Note that G is length-preserving, and when $e_t = 0$, the length of `G[t][i]` is longer by n bits, since the view includes the input share in addition to the transcript.

4. Compute the challenge e, by hashing the output shares, commitments, the signer's public key pk and the message M.

```
e =  H3(
     view[0][0].oShare, view[0][1].oShare, view[0][2].oShare,
     ...
     view[t-1][0].oShare, view[t-1][1].oShare, view[t-1][2].oShare,
     C[0][0], C[0][1], C[0][2],
     ...
     C[t-1][0], C[t-1][1], C[t-1][2],
     [G[0][0], G[0][1], G[0][2],
     ...
     G[t-1][0], G[t-1][1], G[t-1][2],]
     salt, pk, M)
```

The function `H3` is defined in Section 6.4.5, it is a hash function with output in $\{0,1,2\}^t$. The commitments `G[i][j]` must be included when the flag UR is set, and omitted otherwise. We write e as $(e_0, \dots, e_{t-1})$ where $e_i \in \{0,1,2\}$.

5. For each round t from 0 to $T-1$, assemble the proof. For the challenge $e_t \in \{0,1,2\}$, compute $i = e_t + 2 \pmod 3$ and set
b_t `= C[t]`$[i]$, `[G[t][i]]`
Note that `G[t][i]` is only present if UR is set. Then,

if $e_t = 0$, set z_t to
 `view[t][1].transcript, seed[t][0], seed[t][1]`
else if $e_t = 1$, set z_t to
 `view[t][2].transcript, seed[t][1], seed[t][2], view[t][2].iShare`
else if $e_t = 2$, set z_t to
 `view[t][0].transcript, seed[t][2], seed[t][0], view[t][2].iShare`

6. Serialize $(e, salt, b_0, \dots, b_{t-1}, z_0, \dots, z_{t-1})$ as described in Section 6.5.1 and output it as the signature.

6.3 Verification Operation

This section describes the Verify operation, to verify a signature created by the Sign operation in Section 6.2. The functions `matrix_mul`, `mpc_sbox_verify`, `mpc_xor`, `mpc_and` and `H3` used to specify verify are specified in later sections (Sections 6.4.4, 6.4.1, 6.4.3, 6.4.2 and 6.4.5 resp.). As with signing, the steps below work for all security levels, and the flag *UR* is set for parameter sets using the Unruh transform.

Input: Signer's public key *pk*, a message as a byte array M, such that $1 \leq |M| \leq 2^{55}$, a signature σ (also a byte array).

Output: `valid` if σ is a signature of M with respect to *pk* or `invalid` if not.

1. Deserialize the signature σ to $(e, salt, b_0, \dots, b_{t-1}, z_0, \dots, z_{t-1})$ as described in Section 6.5.2. If deserialization fails, reject the signature and output `invalid`. Write e as $(e_0, \dots, e_{t-1})$ where $e_i \in \{0, 1, 2\}$.

2. Initialize lists to contain the three commitments `C[0..t-1][0..2]`, output shares `outputs[0..t-1][0..2]`, and extra commitments `G[0..t-1][0..2]` (if *UR* is set only), for each parallel iteration. These will be inputs to `H3`, verification will re-compute some of these values, and use some provided as part of the signature.

3. For each parallel iteration t from 0 to $T - 1$:

   (a) Initialize two views `view[0]` and `view[1]`, random tapes `rand[0]` and `rand[1]`, and key shares `x[0]` and `x[1]`.

(b) For this step there are three cases, one for each challenge value, as in Step 5 of the Sign operation.

If $e_t = 0$:

i. Use the provided `seed[t][0]` to recompute the random tape `rand[0]`.
ii. Use the provided `seed[t][1]` to recompute the random tape `rand[1]`.
iii. Set `view[0].iShare` and `x[0]` to the first n bits of `rand[0]`.
iv. Set `view[1].iShare` and `x[1]` to the first n bits of `rand[1]`.

If $e_t = 1$:

i. Use the provided `seed[t][1]` to recompute the random tape `rand[0]`.
ii. Use the provided `seed[t][2]` to recompute the random tape `rand[1]`.
iii. Set `view[0].iShare` and `x[0]` to the first n bits of `rand[0]`.
iv. Set `view[1].iShare` and `x[1]` to the input share in z_t.

If $e_t = 2$:

i. Use the provided `seed[t][2]` to recompute the random tape `rand[0]`.
ii. Use the provided `seed[t][0]` to recompute the random tape `rand[1]`.
iii. Set `view[0].iShare` and `x[0]` to the input share in z_t.
iv. Set `view[1].iShare` and `x[1]` to the first n bits of `rand[1]`.

(c) Simulate the MPC protocol to compute the LowMC encrypt circuit. This is similar to signing since the circuit is the same, but because we are only simulating two of the parties instead of all three, the MPC subroutines are slightly different.

i. Compute initial round keys `key[0]` and `key[1]`:
   `key = matrix_mul(x, Kmatrix[0])`
   The function `matrix_mul` is defined in Section 6.4.4.
ii. Initialize shares of the state `state[0]` and `state[1]` with p, the plaintext portion of the public key (C, p), and the key.
   `state = mpc_xor_constant_verify(key,` p, e_t `)`
iii. For each LowMC round `i` from 1 to r
   A. Compute the round i key shares
      `key = matrix_mul(x, Kmatrix[i])`
   B. Apply substitution layer (s-boxes) to `state`:
      `state = mpc_sbox_verify(state, rand, views[t])`

C. Apply affine layer to `state`:
`state = matrix_mul(state, Lmatrix[i-1])`

D. Update the state with the XOR of the round constant and the state:
`state = mpc_xor_constant_verify(state, roundconstant[i-1],` e_t`)`

E. Update the state with the XOR of the round key and the state:
`state = mpc_xor(state, key)`

iv. Store the output shares in the views:
`view[0].oShare = state[0]`
`view[1].oShare = state[1]`

v. Update the list of commitments. Two commitments are recomputed based on the recomputed views, and the third is provided in the proof.
`C[t][`e_t`] = `H_0`(`H_4`(seed[0]), view[0])`
`C[t][`$e_t + 1 \bmod 3$`] = `H_0`(`H_4`(seed[1]), view[1])`
`C[t][`$e_t + 2 \bmod 3$`] = `c
where c is the commitment provided as part of the proof, the first element in b_t. If UR is set, additionally update `G` as follows:
`G[t][`e_t`] = `G`(`H_4`(seed[0]), view[0])`
`G[t][`$e_t + 1 \bmod 3$`] = `G`(`H_4`(seed[1]), view[1])`
`G[t][`$e_t + 2 \bmod 3$`] = `c'
where c' is the commitment provided as part of the proof, the second element in b_t.

vi. Update the list of output shares
`outputs[t][`e_t`] = view[0].oShare`
`outputs[t][`$e_t + 1$`] = view[1].oShare`
`outputs[t][`$e_t + 2$`] = view[0].oShare` $\oplus$ `view[1].oShare` $\oplus$ C
where C is the ciphertext component of the public key (C, p), and the addition is done modulo 3 (as above).

(d) Recompute the challenge

```
e′ =  H3(
      outputs[0][0], outputs[0][1], outputs[0][2],
      ...
      outputs[T-1][0], outputs[T-1][1], outputs[T-1][2],
      C[0][0], C[0][1], C[0][2],
      ...
      C[T-1][0], C[T-1][1], C[T-1][2],
      [G[0][0], G[0][1], G[0][2],
      ...
      G[T-1][0], G[T-1][1], G[T-1][2],]
      salt, pk, M)
```

The commitments `G[i][j]` must be included when the flag *UR* is set, and omitted otherwise.

(e) If e and e' are equal, output `valid` and otherwise output `invalid`.

6.4 Supporting Functions

The Sign (§6.2) and Verify (§6.3) operations use similar functions to simulate the MPC protocol used in the proof of knowledge. This section describes these functions.

6.4.1 LowMC S-Box Layer: `mpc_sbox`, `mpc_sbox_verify`

This section describes how the internal LowMC state is updated in the s-box layer. The number of s-boxes is fixed per parameter set, see Table 2. The input is the three shares of the state, random tapes and views. The tapes and the views are input because the operations in the s-box layer use ANDs and so this function must update the transcript of the MPC protocol. This function also depends on the parameter s, the number of s-boxes, defined in Table 2. The function `mpc_sbox` is used when signing, and verification uses `mpc_sbox_verify`, which has the same definition, but calls to `mpc_and` are replaced with calls to `mpc_and_verify`.

In the following pseudocode, indexing is *bitwise* and zero-based. The temporary variables are triples of bits `a[0..2]`, `b[0..2]` and `c[0..2]`, of each of the three input shares (`ab`, `bc` and `ca` have the same type).

Input: Shares of LowMC state `state`, random tapes `rand`, and `views` as defined in Section 6.2. The input `views` a triple of views, corresponding to one parallel round.
Output: The input variable `state` is modified in place
Pseudocode:

```
for i from 0 to (3*s - 1), in steps of 3
    for j from 0 to 2
        a[j] = state[j][i + 2]
        b[j] = state[j][i + 1]
        c[j] = state[j][i]

    ab = mpc_AND(a, b, rand, views)
    bc = mpc_AND(b, c, rand, views)
    ca = mpc_AND(c, a, rand, views)

    for j from 0 to 2
        state[j][i + 2] = a[j] XOR bc[j]
        state[j][i + 1] = a[j] XOR b[j] XOR ca[j]
        state[j][i] = a[j] XOR b[j] XOR c[j] XOR ab[j]
```

6.4.2 MPC AND Operations: `mpc_and`, `mpc_and_verify`

These functions take secret shares of bits `a`, `b` and compute the binary AND `c = a AND b`, updating the transcript of the MPC protocol. The randomness is read from the pre-computed random tapes, also provided as input. For signing, `mpc_and` takes three inputs, and for verification, a simpler two-input version, `mpc_and_verify` is used. Note that in verification, one of the players' output shares is provided as input.

`mpc_and`
Input: random tapes `rand`, the triple of views for this parallel round `views`, and secret-shared inputs `a[0..2]`, `b[0..2]`
Output: secret shares `c[0..2] = a` AND `b`, updates to the transcripts in `views`
Pseudocode:

```
r[0] = rand[0].nextBit()
r[1] = rand[1].nextBit()
r[2] = rand[2].nextBit()

for i from 0 to 2
    c[i] = (a[i] AND b[(i + 1) % 3]) XOR
           (a[(i + 1) % 3] AND b[i]) XOR
           (a[i] AND b[i]) XOR
           r[i] XOR r[(i + 1) % 3]
```

```
    views[i].transcript.append(c[i])
return c
```

`mpc_and_verify`
Input: random tapes `rand`, the pair of views for this parallel round `views`, and secret-shared inputs `a[0..1]`, `b[0..1]`
Output: secret shares `c[0..1] = a` AND `b`, updates to the transcripts in `views`
Pseudocode:

```
r[0] = rand[0].nextBit()
r[1] = rand[1].nextBit()

c[0] = (a[0] AND b[1]) XOR (a[1] AND b[0]) XOR
       (a[0] AND b[0]) XOR r[0] XOR r[1]
views[0].transcript.append(c[0])

c[1] = views[1].transcript.nextBit()

return c
```

6.4.3 MPC XOR Operations: `mpc_xor`, `mpc_xor_constant`

This function takes secret-shared input bits a, b and computes the secret shares of $c = a \oplus b$. Unlike the AND operation, which requires communication between players, the XOR operation is done locally in the MPC protocol, and does not need to update the views.

Input: m bit vectors of length L: `a[0..m - 1][0..L - 1]` and `b[0..m - 1][0..L - 1]`
Output: XOR of the two inputs `c[0..m][0..L - 1]`
Pseudocode:

```
for i = 0 to m - 1
  c[i] = a[i] XOR b[i] // XOR of L-bit strings
```

Note that (i) m is always 3 during the Sign operation, and 2 during verify, and (ii) implementations may work on multiple bits simultaneously using the processor's XOR instruction on word size operands.

XOR with a constant When one of the operands is a public constant instead of a secret share vector, the constant is XORed with only one of the secret shares. When signing, in `mpc_xor_constant`, the first share is always XORed with the constant. When verifying, in `mpc_xor_constant_verify`, if the challenge $e_t = 0$ then we XOR the first secret share with the constant, and when $e_t = 2$ we XOR the second secret share with the constant. (This is because the state corresponding to the first player is in a different position depending on the challenge.)

6.4.4 Binary Vector-Matrix Multiplication: `matrix_mul`

This function computes a vector-matrix product, with elements in GF(2). Let `x[i]` denote the i-th bit of x, and `M[i][j]` denote the bit in the i-th row and j-th column of M.

Input: an n-bit vector x, an n-bit by n-bit matrix M
Output: an n-bit vector $a = xM$
Pseudocode:

```
temp is a bit
for i = 0 to n - 1
    temp = 0
    for j = 0 to n - 1
        temp = temp XOR (x[j] AND M[i][j])
   a[i] = temp
Output a
```

Notes

1. In signature generation for the `picnic` parameters, three vectors x, y, and z in $\mathrm{GF}(2)^n$ are input along with a single matrix $M \in \mathrm{GF}(2)^{n \times n}$, and three vectors xM, yM and zM in $\mathrm{GF}(2)^n$ are output. Simiarly for verification, two vectors are input.

2. If inputs and outputs may overlap (e.g., when computing $x = xM$) a temporary variable is required for the output.

3. There are many ways to compute this function, implementations may use an alternative algorithm for better efficiency. For example, see [Alb17].

6.4.5 Computing the Challenge: `H3`

The function `H3` hashes an arbitrary length bitstring to a length T output in $\{0,1,2\}$ (i.e., `H3` $: \{0,1\}^* \to \{0,1,2\}^t$). The hash function H is called on the input, then iterated as required, to compute an output of length T.

In the pseudocode below, the hash function H is given in Table 2, along with the value for the parameter T. Recall that H_1 is defined as $H_1(x) = H(\texttt{0x01}||x)$.

Input: bitstring b
Output: vector e, of integers in $\{0,1,2\}$
Pseudocode:

1. Compute $h = H_1(b)$, write h in binary as $(h_0, h_1, ..., h_S)$.

2. Iterate over pairs of bits $(h_0, h_1), (h_2, h_3), \ldots$. If the pair is

   $(0,0)$, append 0 to e,

   $(0,1)$, append 1 to e,

   $(1,0)$, append 2 to e,

   $(1,1)$, do nothing.

   If e has length T, return.

3. If all pairs are consumed and e still has fewer than T elements, set $h = H_1(h)$ and return to Step 2.

6.4.6 Function G

The function G has two inputs: a *seed* of length S bits, and a view, v, of varying length. The output has length ℓ_G, computed as the sum of the length of the seed and the length of the view. Recall that not all views are equal length, ℓ_G differs depending on which player computed the view. G is implemented with the KDF from Table 2, namely, with SHAKE and the following input:

$$H_5(\mathit{seed})\|v\|\ell_G$$

The integer ℓ_G is encoded as a 16-bit little endian integer.

6.5 Serialization

In this section we specify how to serialize and deserialize Picnic signatures with the nine `picnic` parameter sets.

6.5.1 Serialization of Signatures

This section specifies how to serialize signatures created in Section 6.2.

This is a binary, fixed-length encoding, designed to minimize the space required by the signature. The components of the signature (views, seeds, commitments, etc.) are all of fixed length for a given parameter set. The Fiat-Shamir parameter sets have signatures that vary in size, depending on the challenge; note that in Step 5, an additional input share is output when the challenge is 1 or 2. The serialization does not include an identifier indicating the parameter set, as not all applications require it.

Input: The signature $(e, \mathit{salt}, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$, as computed in Section 6.2, Step 5.

Output: A byte array B, encoding the signature.

1. Write the challenge to B, using $2T$ bits, padding with zero bits on the right to the nearest byte.
2. Write *salt* to B, using 32 bytes.
3. For each t from 0 to $T-1$, append (b_t, z_t) as follows. For values that do not use an even number of bytes, pad with zero bits to the next byte.
   (a) Append b_t, a commitment of length ℓ_H bytes, and if the *UR* flag is set, also append the second commitment (denoted `G[t][i]` in Step 3d of signing).
   (b) Append z_i (in the order presented in Step 5 of Sign)
      i. Append the transcript.
      ii. Append the two seed values in z_t,
      iii. If e_t is 1 or 2, append the input share.
4. Output B.

6.5.2 Deserialization of Signatures

This section describes how to deserialize a byte array created by Section 6.5.1 to a signature for use in verification. The deserialization process reads the input bytes linearly. Since the signature length can vary depending on the challenge (encoded first in the byte array), it is recommended that implementations first compute the expected length from e, and reject the signature before parsing further, if B does not have the expected number of remaining bytes.

Input: A byte array B, encoding the signature.

Output: The signature $(e, salt, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$, as computed in Section 6.2, Step 5, or `null` if deserialization fails.

1. Read the first $(2T + 7)/8$ bytes from B. If the read fails, return `null`. Ensure that each pair of bits in the first $2T$ bits are in $\{0, 1, 2\}$ and return `null` if not. If padding bits are required for this value of T (see §6.5.1), ensure that all padding bits are zero, and return `null` if not. Assign these bytes to e. We use the notation $e = (e_0, \ldots, e_{T-1})$ to denote the individual pairs of bits.
2. Read the next 32 bytes from B, and assign them to *salt*. If the read fails, return `null`.
3. For each t from 0 to $T - 1$, read (b_t, z_t) from B as follows. If any of the reads are not possible because B is too short, abort and return `null`.
   (a) Create b_t by reading a commitment of length ℓ_H bytes from B. If *UR* is set, also read a second commitment from B, of length $3rs + n$ bits when $e_t == 0$ and $3rs$ bits otherwise.
   (b) Read z_t, as follows:
      i. Read the transcript from B, assign it to the first component of z_t. The length of the transcript is $3rs$ bits.
      ii. Read the first seed value of length S bits from B, append it to z_t.
      iii. Read the second seed value of length S bits from B, append it to z_t.
      iv. If e_t is 1 or 2, read an input share of length S bits from B and append it to z_t.
4. Output $(e, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$.

7 Signing and Verification for `picnic3` Parameter Sets

This section specifies the signing operation for the three parameter sets `picnic3-L1`, `picnic3-L3` and `picnic3-L5`.

7.1 Sign Operation

The functions `mpc_simulate`, `HCP` and `compute_aux` used to specify signing are specified in Sections 7.5, 7.6 and 7.4 respectively. Section 7.3 defines the algorithms for working with binary tree data structures; signing and verification use trees to derive pseudorandom seeds, and to commit to values using Merkle's construction. The description of signature generation is independent of the security level.

Input: Signer's key pair (sk, pk), a message to be signed, the byte array M, such that $1 \leq |M| \leq 2^{55}$.

Output: Signature on M as a byte array.

1. Initialize the following lists of values. Lists `C[0..T-1][0..N-1]`, `Ch[0..T-1]` and `Cv[0..T-1]` of commitments, where each value has length ℓ_H bytes. A list of T masked inputs to the MPC simulation `masked_key[0..T-1]`, each of length n bits. A list of random tapes `tapes[0..T-1][0..N-1]`, one per party, per parallel iteration, each of length $6rs+n$ bits. A list of auxiliary information bitstrings `aux[0..T-1]`, each of length $3rs$ bits. A list of broadcast messages `msgs[0..T-1][0..N-1]`, bitstrings of length $n + 3rs$ bits.

2. Generate a root seed of length S bits and a salt of length 256 bits. It is recommended that these be derived deterministically, by calling the KDF in Table 2, with input

   $$sk \| M \| pk \| S$$

   where S is encoded as a 16-bit little endian integer. The number of bytes requested is $2(S/8)$ (one salt, and one seed, each of size S bits). The salt value is denoted *salt*. How the root seed and salt are chosen does not affect interoperability, and some implementations may prefer a randomized signing function. For implementations seeking to randomize the signature function, it is recommended to use the derivation described here, but to append a $2S$-bit random value to the KDF input. See the discussion in Section 8.3.

3. Expand the root seed and salt into T initial seeds, denoted `iSeed[0..T-1]`, using the tree method described in Section 7.3.1. An integer parameter is also required for this method (denoted t in §7.3.1); this must be zero.

4. For each parallel repetition t from 0 to $T - 1$:

(a) Using `iSeeds[t]`, *salt* and the integer t, derive N seeds, using the tree method described in Section 7.3.1. Denote these seeds `seeds[0..N-1]`.

(b) Derive N random tapes `tapes[t][0..N-1]`, using the KDF from Table 2:

$$\texttt{tapes}[\texttt{t}][\texttt{i}] = \text{KDF}(\texttt{seeds}[i]\|\mathit{salt}\|t\|i)$$

The number of bits per tape must be at least $6rs$. The inputs t and i are encoded as 16-bit little-endian integers.

(c) Compute the auxiliary tape bits, with the algorithm `compute_aux`, using `tapes` as input, as described in Section 7.4. Denote these bits `aux[t]`. Note that `compute_aux` also updates the N-th party's tape with the auxiliary bits, and returns the string `mpcInputs`, that will be used below.

(d) Compute the N commitments `C[t][0..N-1]` as follows:

$$\texttt{C}[\texttt{t}][\texttt{i}] = H(\texttt{seeds}[\texttt{i}]\|\mathit{salt}\|t\|i)$$

for i from 0 to $N-2$, and

$$\texttt{C}[\texttt{t}][\texttt{N}-\texttt{1}] = H(\texttt{seeds}[\texttt{N}-\texttt{1}]\|\texttt{aux}[\texttt{t}]\|\mathit{salt}\|t\|i) \ .$$

where the bit string `aux[t]` is padded with zeroes to the next byte boundary if necessary.

(e) Create and store a masked version of the private key, used to simulate the online phase of the MPC protocol: `maskedKey[t]` = `mpcInputs` $\oplus$ *sk*.

(f) Run the `mpc_simulate` algorithm from Section 7.5, with inputs `maskedKey[t]`, `tapes[t]` and *pk*. The output is the list of broadcast messages for each of the N parties `msgs[t][0..N-1]`. This step may fail in exceptional circumstances, depending on the implementation, see Section 8.4.

(g) Compute the commitment `Ch[t]` as follows;

$$\texttt{Ch}[\texttt{t}] = H(\texttt{C}[\texttt{t}][\texttt{0}]||\ldots||\texttt{C}[\texttt{t}][\texttt{N}-\texttt{1}]) \ .$$

(h) Compute the commitment `Cv[t]` as follows:

$$\texttt{Cv}[\texttt{t}] = H(\texttt{maskedKey}[t]\|\texttt{msgs}[t][0]\|\ldots\|\texttt{msgs}[t][N-1])$$

where the bit strings in `msgs` are padded to the nearest byte with zeros.

5. Create a Merkle tree with the T commitments `Cv[0..T-1]` as the leaves, as described in Section 7.3.2. Let `Cv_root` be the root node.

6. Compute the challenge using the function `HCP` defined in Section 7.6. The output is a digest `h` and two lists of length u, of 16-bit integers, `LC` and `LP`.

$$(\mathtt{h}, \mathtt{LC}, \mathtt{LP}) = \mathtt{HCP}(\mathtt{Ch}[0],\ \ldots,\ \mathtt{Ch}[\mathtt{T}-1],\ \mathtt{Cv\_root},\ \mathtt{salt},\ pk,\ M)$$

   The integers in `LC` are unique and in the range $[0, T-1]$ and those in `LP` are in the range $[0, N-1]$.

7. Compute the opening information for the Merkle tree of commitments `Cv`, as described in Section 7.3.2. The set of missing leaves is $\{0, \ldots, T-1\} \setminus \mathtt{LC}$. Denote the opening information `cvInfo`.

8. Compute the information required to recompute the initial seeds for parallel repetitions $t \notin \mathtt{LC}$, as described in Section 7.3.1. The set of leaf indices that remain hidden is `LC`. Denote this information `iSeedInfo`.

9. Assemble the signature. The signature is $(\mathtt{h}, salt, \mathtt{iSeedInfo}, \mathtt{cvInfo}, Z)$ where Z is a list of u 5-tuples. For each i from 0 to $u-1$, define $(t_i, P_i) = (\mathtt{LC}[i], \mathtt{LP}[i])$ and append the following 5-tuple to Z:

   - `seedInfo`[t_i]: computed to reveal all seeds in round t_i except seed P_i,
   - `aux`[t_i] if $P_i \neq N-1$, and `null` otherwise
   - `maskedKey`[t_i],
   - `msgs`[t_i],
   - `C`[t_i][P_i]

10. Serialize $(\mathtt{h}, salt, \mathtt{iSeedInfo}, \mathtt{cvInfo}, Z)$ as described in Section 7.7.1 and output it as the signature.

7.2 Verification Operation

This section describes the Verify operation, to verify a signature created by the Sign operation in Section 7.1. The functions `mpc_simulate` and `HCP` used to specify verification are specified in Sections 7.5 and 7.6 respectively. Section 7.3 defines the algorithms for working with binary trees for deriving seeds and creating commitments. As with signing, the steps below work for all security levels.

Input: Signer's public key pk, a message as a byte array M, such that $1 \leq |M| \leq 2^{55}$, a signature σ (also a byte array).

Output: `valid` if σ is a signature of M with respect to pk or `invalid` if not.

1. Deserialize the signature σ to $(\mathtt{h}, salt, \mathtt{iSeedInfo}, \mathtt{cvInfo}, Z)$ as described in Section 7.7.2. If deserialization fails, reject the signature and output `invalid`. Expand `h` to the two lists `LC` and `LP` as described in Section 7.6.

2. Create a seed tree with T leaves, and reconstruct the seeds using `iSeedInfo`, *salt* and $t = 0$ as described in Section 7.3.1. The indices of the missing leaves is the list `LC`. If reconstruction fails, return `invalid`. The leaves of the tree are denoted `iSeed[0..T-1]`, though leaves with indices in `LC` are `null` (and will not be referenced below).

3. For each parallel repetition t from 0 to $T - 1$:

   (a) If $t \in \mathtt{LC}$, let z_t be the index of t in `LC`, and define $P_t = \mathtt{LP}[z_t]$. Read a 5-tuple from Z with the values

      - `seedInfo[t]`
      - `aux`, defined if $P_t \neq N - 1$, or `null` otherwise,
      - `maskedKey[t]`,
      - `msgs[t]`,
      - `C[t]`[P_t]

      Note that checks during deserialization will ensure this succeeds.

   (b) Populate `seeds[0..N-1]`, the seeds for each party used in repetition t.

      i. If $t \notin \mathtt{LC}$, use `iSeed[t]`, *salt* and t to generate the N seeds as the signer did in Step 4a.

      ii. If $t \in \mathtt{LC}$, use `seedInfo[t]`, *salt* and t to reconstruct the seeds for all N parties except P_t (as described in Section 7.3.1). The reconstructed tree has N leaves, these are the seeds `seeds[0..N-1]`, with `seeds`[P_t] being `null`. If reconstruction fails, abort and return `invalid`.

   (c) Expand the seeds to compute random tapes `tapes[0..N-1]` for all parties (except possibly P_t), using `seeds`, *salt* and t, as the signer did in Step 4b.

   (d) Compute the commitments `C`[t]`[0..N-1]`.

      i. If $t \notin \mathtt{LC}$, we have the seeds and tapes for all parties, compute the auxiliary information `aux` as the signer did in Step 4c, and form commitments `C`[t]`[0..N-1]` as the signer did in Step 4d.

ii. If $t \in$ `LC`, we have all seeds but `seeds[`P_t`]`. For all but the last party and P_t we can recompute `C[`t`][`i`]` from `seeds[`i`]`. If we need to recompute the last party's commitment (when $P_t \neq N-1$), we have `seeds[`P_t`]` and we are given `aux`. The commitment `C[`t`][`P_t`]` is provided as part of the signature, so all N commitments are present.

(e) Compute the commitment `Ch[t]`, using `C[`t`][0..N-1]`, as the signer did in Step 4g.

(f) If $t \in$ `LC`, simulate the MPC protocol, with $N-1$ parties.

i. If $P_t \neq N-1$, use `set_aux_bits(tapes[N-1], aux)` to write the `aux` bits in the correct positions of the last party's tape. The function `set_aux_bits` is defined in Section 7.5.3.

ii. Set party P_t's tape to zeroes.

iii. Set party P_t's broadcast messages to `msgs[`t`]`, provided by the signer. These will be read by other parties during the simulation.

iv. Run the `mpc_simulate` algorithm with inputs `maskedKey`, `tapes`, `msgs` and pk.

If simulation fails, fail and return `invalid`. Upon success, the output of `mpc_simulate` is the broadcast messages of the $N-1$ opened parties.

(g) If $t \in$ `LC`, recompute the commitment `Cv[`t`]`, as the signer did in Step 4h.

4. Create a Merkle tree with T leaves, initialized to `Cv[`t`]`, for $t \in$ `LC`. The remaining leaves are left `null`. Using `cvInfo`, verify the Merkle tree as described in Section 7.3.2. Upon success, this will rebuild the tree up to the root (denoted `Cv_root`. If verifying the tree fails, fail and return `invalid`.

5. Recompute the challenge hash,

$$(\mathtt{h}') = \mathtt{HCP}(\mathtt{Ch}[0], \ldots, \mathtt{Ch}[\mathtt{T}-1], \mathtt{Cv\_root}, \mathtt{salt}, pk, M)$$

and compare `h'` to `h` from the signature. If they match, σ is a valid signature on M with respect to pk, return `valid`, otherwise return `invalid`.

7.3 Tree Data Structures

Two types of tree data structures are used when creating and verifying Picnic signatures (for the `picnic3` parameter sets). In both cases we use binary trees, that are almost complete: we use the smallest tree that has the required number of leaves, use only the leftmost leaves, and are only concerned with intermediate nodes that are on a path from a leaf back to the root.

The basic tree data structure is common to both types of trees, so implementations may re-use code. In both cases, each node in the tree contains a bit string as data, of the same length for all nodes.

Each node in the tree has an integer index associated to it, assigned in breadth-first order (i.e., the root has index 0, the root's left child has index 1, the right child has index 2, and in general node i's left child has index $2i + 1$, and right child has index $2i + 2$).

The signer must reveal information as part of the signature, and the size of this information is variable, depending on the challenge value computed as part of the signature. During verification, the expected size must be recomputed once the challenge is known (and validated as well-formed). The algorithms given here to output the data can be easily modified to compute the amount of data that should be output for a given challenge.

7.3.1 Seed Trees

When signing, the signer must generate a set of seeds, then reveal a subset of these based on the challenge. The seeds are then used by the verifier to check that the MPC protocol was setup or simulated correctly. By deriving seeds deterministically in a binary tree, then using the leaf seeds in the protocol, the signer can reveal large subsets of the seeds efficiently by revealing intermediate nodes in the tree. In `picnic3-L1`, `picnic3-L3` and `picnic3-L5`, the signer must reveal $T - u$ of the initial seeds and one of the N seeds in each of the u parallel repetitions that are checked by the verifier.

Generate Seeds

Input: A tree with a root seed, a salt value *salt* and an integer t. The size of the root seed and derived seeds are all S bits.

Output: The seeds are computed at each non-root node.

For each non-leaf node in the tree, having seed `parent_seed`, compute the $2S$-bit digest

$$H_1(\texttt{parent\_seed} \| salt \| t \| j)$$

where j is the index of the parent node, both t and j are encoded as little-endian integers, and the function H_1 is defined in Section 3.2. Then set the left child of the node to the leftmost S bits, and the right child to the rightmost S bits.

Note that only a given number of leaf nodes are required by the protocol, and only intermediate nodes on the path back to the root from these leaves are required.

Reveal Seeds
When revealing seeds, note that the order of the seeds must match the order given here, otherwise implementations will not interoperate. For the algorithm below, the order of the input is important. The function sibling(n) returns the sibling of node n, and the function numChildren(n) returns the number of children node n has. It may be zero if n is a leaf, two for most intermdeiate nodes, except those on the right side of the tree (which is incomplete) where it may be one. Because the tree is not complete, on the right hand side some nodes may be at the end of a chain. In these cases the algorithm below will output the node directly, rather than a parent along the chain.
Input: A tree created with the process above, and a set of leaf nodes L that must remain hidden. All leaves not in L will be revealed.
Output: An ordered list of seeds Z.

1. For each leaf node $\ell \in L$, compute the path from ℓ to the root. A path is an array, where index 0 is the leaf and the highest index, denoted R, is the root. Let *Paths* be the set of all paths.

2. For each path position i from $R - 1$ down to 0, consider the set N_i of nodes in all paths in *Paths* at position i. N_i is ordered according to L.

   (a) For each node $n \in N_i$, set $sib = \text{sibling}(n)$. If $sib \notin N_i$ reveal a seed:
      - if $\text{numChildren}(sib) = 1$, set $sib = \text{child}(sib)$ until sib is a node with two children, or a leaf.
      - Append sib to Z if it has not already been added to Z. Otherwise do nothing.

3. Return Z.

Reconstruct Seeds
Input: An empty tree, and a set of leaf nodes L that will not be reconstructed. A list of seeds Z, output by the reveal seeds function.
Output: On success, the tree is updated to have seeds for all leaves, except those in L. In case of failure, `invalid` is returned.

1. Repeat the algorithm for revealing seeds; however, instead of appending a seed to Z in Step 2a, read from Z and assign it to the node *sib*. If Z contains too few or too many seeds, fail and return `invalid`.

2. Starting at the root, proceed down the tree in breadth-first order, until all non-leaf nodes are visited. For nodes where the seed is present, derive children seeds that are not present. Child seeds are derived from the parent seed by hashing, as described above.

7.3.2 Merkle Trees

When signing, the signer commits to a set of values, then opens a subset of these commitments. More precisely, the verifier is given information required to recompute the subset of opened commitments. Since all commitments are required by the verifier to recompute the challenge, the prover must send all unopened commitments as part of the signature. By using a Merkle tree to commit to all values at once, the prover can reduce the data sent to the verifier. The verifier only requires enough information to check that the recomputed values were committed to by the Merkle tree.

Build Merkle Tree
Input: A binary tree with T leaves, a salt value *salt*, and a list of T commitments.
Output: The Merkle tree is computed, with the root being a commitment to the T input commitments.

1. Assign the T commitments to the T leaves, in order given, i.e., commitment zero should be the leftmost leaf, commitment one should be its sibling, and so on.

2. Proceed bottom-up to compute the intermediate nodes. For a node a with children hashes c_ℓ and c_r, compute the hash for a as

$$H_3(c_\ell \| c_r \| salt \| j)$$

where j is the index of node a (as defined at the beginning of Section 7.3), encoded as a little-endian integer. The function H_3 is defined in Section 3.2. Note that the right child may not exist in the tree, depending on the number of leaves, and in this case c_r may be `null` and omitted from the digest.

Open Merkle Tree
Input: A tree created with the process above, and a set of leaf nodes L that will be missing by the verifier. The leaves not in L will be recomputed by the verifier.
Output: An ordered list of hash values Z.

1. Initialize a set of missing nodes $N_M = \emptyset$.
2. Add the missing leaf nodes to N_M.
3. Proceed up the tree, adding node n to N_M if both of n's children are in N_M. (For n with only one child, add n to N_M if the child is missing.)
4. For each missing leaf node in L (processed in the order given), walk the path back to the root, and find the node n_h nearest the root that is in N_M (but not the root itself). If n_h was not already appended to Z, append n_h to Z.
5. Output Z.

Verify Merkle Opening
The verifier uses this function to attempt to recompute the Merkle tree using the recomputed commitments and the intermediate nodes provided by the prover. If the root can be recomputed, the root hash is then used when recomputing the challenge, confirming that the parts of the tree the verifier recomputed matched the prover's tree.
Input: An empty Merkle tree with T leaves, a set of recomputed leaf nodes, and an ordered list of missing leaf nodes L. A list of hashes Z, output by the open algorithm.
Output: The tree is updated with all intermediate nodes, up to the root. If the root could not be computed `invalid` is output.

1. Assign the recomputed leaf nodes to the tree.
2. Using the tree and L, repeat the algorithm used to compute Z given above, but read hashes from Z instead of writing, and assign them to the corresponding node in the tree. If Z is too short or too long, fail and return `invalid`.
3. The tree now has some leaves and intermediate nodes. Proceed bottom-up recomputing parent nodes using the process the signer uses when building the tree (computing nodes only where possible, i.e., hashes are present for all children of a node).
4. If the root node was not computed, return `invalid`.

7.4 MPC Preprocessing Step: `compute_aux`

This is a preprocessing step for the MPC algorithm, where the auxiliary information is computed based on the random tapes of the parties. The auxiliary information is given to the N-th party (as an update to thier random tape), and used when computing AND gates. We compute a simplified version of LowMC, that operates on shares of the mask values that will be used during the MPC simulation. The preprocessing step is simpler because XOR by a constant is a no-op.

In addition to binary vectors, we also work with vectors of length n (the LowMC key and block size), where the entries are N-bit words. Each word is a packed representation of the secret share held by the N parties. (i.e., bit 0 is the secret share of party 0, bit 1 the share of party 1, and so on.) This particular representation is used here to make presentation concrete, but it is not required for interoperability.

Input: One random tape per party, N in total.
Output: The N-th party's random tape is updated with the auxiliary information. The bits in the tape are set so that they are read in the correct order during the MPC simulation, as described above. During signature generation an n-bit string `mpc_inputs` is returned.

The functions `matrix_mul`, `aux_sbox` are defined in Sections 6.4.4 and 7.4.1 respectively.

1. Allocate bitstrings `key0`, `key`, `round_key`, `x`, and `y` each of length n. Set `x` $= 0$.

2. Read an n-bit string from each of the N tapes, then compute the bitwise XOR of all N strings to get a single bitstring of length n, assign it to `key0`.

3. Compute
   `key = matrix_mul(key0, KMatrixInv[0])`

4. If this is signature generation, assign `key` to `mpc_inputs`, to be returned to the caller at the end of this function. During signature verification no values are returned.

5. For each LowMC round `i` from r down to 1

   (a) Compute the `i`-th round key:
       `round_key = matrix_mul(key, Kmatrix[i])`

(b) Compute
    `x = x ⊕ round_key`

(c) Compute
    `y = matrix_mul(x, LMatrixInv[i-1]`

(d) If `i` == 1, set `x` = `key0`, otherwise read n bits from each of the N tapes, at position $2n(i-1)$, and compute their XOR to get a single bitstring, and assign it to `x`. Set the position of the random tapes to $2n(i-1)+n$.

(e) Compute
    `aux_sbox(x, y, tapes)`

6. Reset the position of the random tapes to 0, so that the MPC simulation begins reading at the start of each tape.

7.4.1 Preprocessing S-Boxes: `aux_sbox`

Input: Two bitstrings `x` and `y` of length n, indexed bitwise. Random tapes for N parties, `tapes`.
Output: Random bits are read from the tapes, and the last party's tape is updated with the auxiliary bits.

Pseudocode:

```
a, b, c, d, e, f are bits
for i = 0 to 3*s, in steps of 3 (i += 3)
    a = x[i + 2]
    b = x[i + 1]
    c = x[i]

    d = y[i + 2]
    e = y[i + 1]
    f = y[i]

    fresh_mask_ab = f XOR a XOR b XOR c
    fresh_mask_bc = d XOR a
    fresh_mask_ca = e XOR a XOR b
```

```
aux_AND(a, b, fresh_mask_ab, tapes)
aux_AND(b, c, fresh_mask_bc, tapes)
aux_AND(c, a, fresh_mask_ca, tapes)
```

7.4.2 Preprocessing AND gates: `aux_AND`

The helper function `tapes_to_word` reads the next bit from each of the N tapes, and packs them into an N-bit word. The function `parity()` computes the parity of a word (equivalent to reconstructing a secret shared bit, where shares are packed into a word). In the last step, the N-th party's tape is updated, note that we're updating the last bit that was read (one bit before the bit position returned by the next call to `nextBit`), since we read random values for all other $N - 1$ parties.

Input: bits `a`, `b`, and `fresh_mask_ab`, Random tapes for all N parties `tapes`.
Output: Random bits are read from the tapes, and the last party's tape is updated with the aux bit.

Pseudocode:

```
mask_a = parity(a)
mask_b = parity(b)

and_helper = tapes_to_word(tapes)
bit c : the N-th bit of and_helper
and_helper = parity(and_helper) ^ c

aux_bit = (a AND b) XOR and_helper XOR fresh_mask_ab
set the previous bit of the N-th tape to aux_bit
```

7.5 MPC Simulation: `mpc_simulate`

This function simulates the MPC protocol, used during signing and verification. The functions `matrix_mul`, and `mpc_sbox3` are defined in Sections 6.4.4, and 7.5.1 respectively. Note that `mpc_simulate` is nearly the same as a LowMC evaluation, with the exception of the S-box layer.

Input: masked_key a binary vector of length n. The signer's public key $pk = (C, p)$. tapes[0..N-1] random tapes for the N parties. During signature verification, the input also contains the index of the unopened party P_t, and broadcast messages in msgs[P_t].
Output: The set of broadcast messages from each party msgs[0..N-1].

1. Compute round_key = matrix_mul(masked_key, KMatrix[0].
2. Compute state = round_key $\oplus$ p.
3. Initialize a vector of secret shares, tmp_shares, a length n vector of N-bit words.
4. For each LowMC round i from 1 to r
   (a) Read n bits from each of the N tapes, packing the shares into tmp_shares.
   (b) Call mpc_sbox3(state, tmp_shares, tapes, msgs).
   (c) Compute state = matrix_mul(state, LMatrix[i - 1]).
   (d) Compute state = state $\oplus$ roundConstant[i-1].
   (e) Compute round_key = matrix_mul(maskedKey, KMatrix[i]).
   (f) Compute state = state $\oplus$ round_key.
5. Compare state and the C component of pk. If they differ, fail and return invalid.

Note that the output, the broadcast messages msgs are updated in mpc_sbox3.

7.5.1 LowMC S-Box Layer: mpc_sbox3

Input: state, an n-bit binary vector, and corresponding mask shares state_masks, a vector of length n, of N-bit words. Random tapes of the N parties, tapes[0..N-1] and the broadcast messages of the N parties msgs[0..N-1].
Output: state, state_masks and msgs are updated, and $3s$ random bits are consumed from tapes.
The function mpc_and is defined in Section 7.5.2.
Pseudocode:

```
a, b, c, ab, bc, ca are bits
mask_* are N-bit words
for i = 0 to 3*s, in steps of 3 (i += 3)
    a = state[i + 2]
    mask_a = state_masks[i + 2]
    b = state[i + 1]
    mask_b = state_masks[i + 1]
    c = state[i]
    mask_c = state_masks[i]

    ab = mpc_AND3(a, b, mask_a, mask_b, tapes, msgs)
    bc = mpc_AND3(b, c, mask_b, mask_c, tapes, msgs)
    ca = mpc_AND3(c, a, mask_c, mask_a, tapes, msgs)

    state[i + 2] = a XOR bc
    state[i + 1] = a XOR b XOR ca
    state[i] = a XOR b XOR c XOR ab
```

7.5.2 MPC AND operation: `mpc_AND3`

The helper functions `tapes_to_word`, `parity` and `extend` are defined in Section 7.4.2.

Input: Two masked bits a and b, along with packed shares of their masks `mask_a` and `mask_b`. Random tapes for all N parties, `tapes`. Broadcast message lists for all N parties, `msgs`. If this is signature verification, the input must include the index of the unopened party P_t, and their broadcast bits `msgs[`P_t`]`.
Output: The masked bit $c = ab$. One random bit is read from each tape, and the broadcast messages are updated.

Pseudocode:

```
and_helper = tapes_to_word(tapes)
s_shares = (extend(a) AND mask_b) XOR
           (extend(b) AND mask_a) XOR
           and_helper

if verification
    read the broadcast message bit d of the unopened party from msgs
    set the share of the unopened party to d in s_shares
```

```
/* Broadcast shares of s */
append bit i of s_shares to msgs[i]

c = parity(s_shares) XOR (a AND b)

return c
```

7.5.3 Tape Update: `set_aux_bits`

The function `set_aux_bits` is used to verify a `picnic3` signature. It takes a bitstring `aux` provided by the signer, and writes it to the the last party's random tape (provided as input), in the same order used during signature generation (in `mpc_simulate`.

Input: a tape to be updated, and a vector of aux bits
Output: the tape is updated

```
pos = 0
for j from 0 to r
    for i from 0 to n
        tape[n + 2*n*j + i] = aux[pos]
        pos++
```

7.6 Computing the Challenge: `HCP`

The function `HCP` hashes an arbitrary length bitstring and outputs a hash digest and (optionally) expands this digest to two lists of length u. The hash function H is called on the input to create the digest `h`, then iterated as required, to expand `h` to the output lists ($\texttt{LC}, \texttt{LP}$) of the required length.

In the pseudocode below, the hash function H is given in Table 2, along with the value for the parameters T, u and N.

Input: A bitstring b.
Output: The digest `h`, and two lists, `LC` and `LP`. `LC` is a list of distinct integers in the range $[0, T-1]$ and `LP` is a list of integers in the range $[0, N-1]$.

1. Compute $\texttt{h} = H(b)$.
2. Initialize the list `LC`.

(a) Iterate over chunks of `h`, each of size $\lceil \log_2(T)) \rceil$ bits. If there are trailing bits shorter than one chunk, ignore them.

(b) For each chunk c, interpret c as an unsigned integer in little endian representation. If $c < T$ and $c \notin \texttt{LC}$, append c to `LC`. If `LC` has length u, end Step 2.

(c) If all chunks are processed, set $\texttt{h} = H(\texttt{h})$ and continue at Step 2a.

3. Set $\texttt{h} = H(\texttt{h})$.

4. Initialize the list `LP`.

   (a) Iterate over the bits of `h` in $\lceil \log_2(N) \rceil$-bit chunks. If there are trailing bits shorter than one chunk, ignore them.

   (b) For each chunk p, interpret p as an unsigned integer in little endian representation. If $p < N$ append p to `LP`. If P has length u, end Step 4.

   (c) If all chunks are processed, set $\texttt{h} = H(\texttt{h})$ and continue at Step 4a.

5. Return (`h`, `LC`, `LP`).

7.7 Serialization

In this section we specify how to serialize and deserialize Picnic signatures with the `picnic3-L1`, `picnic3-L3`, and `picnic3-L5` parameter sets.

7.7.1 Serialization of Signatures

This section specifies how to serialize signatures created in Section 7.1.

This is a binary, fixed-length encoding, designed to minimize the space required by the signature. The components of the signature are all of fixed length for a given parameter set and challenge $(\texttt{LC}, \texttt{LP})$; note that the amount of information required to check commitments with the Merkle tree, or to recompute (initial) seeds can be recomputed from the challenge. Also the opening information for the last party is larger than the other parties, and which party is opened depends on the challenge. In all cases, given the parameter set and the challenge, the verifier can compute the exact size required for all signature components. The serialization does not include an identifier indicating the parameter set, as not all applications require it.

Input: The signature $(\texttt{LC}, \texttt{LP}, \mathit{salt}, \texttt{iSeedInfo}, \texttt{cvInfo}, Z)$, as computed in Section 7.1, Step 9.

Output: A byte array B, encoding the signature.

1. Write the challenge hash `h` using ℓ_H bytes. These integers should be encoded in little-endian byte order.
2. Write *salt* to B, using 32 bytes.
3. Write `iSeedInfo` and `cvInfo` to B. The number of bytes required by each of these will vary per signature.
4. Append each of the u 5-tuples in Z, (in the order presented in Step 9 of signing).
   (a) Append `seedInfo` to B, the length varies per signature.
   (b) Append `aux`, which may be `null`.
   (c) Append `masked key`, which is $S/8$ bytes.
   (d) Append `msgs`, which is $\lceil (S + 3rs)/8 \rceil$ bytes.
   (e) Append `C` which is ℓ_H bytes.
5. Output B.

7.7.2 Deserialization of Signatures

This section describes how to deserialize a byte array created by Section 7.7.1 to a signature for use in verification. The deserialization process reads the input bytes linearly. Since the signature length can vary depending on the challenge (encoded first in the byte array), it is recommended that implementations first compute the expected length from $(\mathtt{LC}, \mathtt{LP})$, and reject the signature before parsing further if B does not have the expected number of remaining bytes.

When deserializing the values `iSeedInfo`, `cvInfo` and `seedInfo`, the number of bytes for each must be recomputed using the challenge, using the same algorithm used by the signer to compute them. One way to do this is to create a tree of the correct size, without any seed or hash data, and compute which nodes must be revealed for a given challenge.

Input: A byte array B, encoding the signature.

Output: The signature ($\texttt{h}, \texttt{LC}, \texttt{LP}, salt, \texttt{iSeedInfo}, \texttt{cvInfo}, Z$), as computed in Section 7.1, Step 9, or `null` if deserialization fails.

1. Read the first ℓ_H bytes from B and assign them to `h`. If the read fails, return `null`. Expand `h` to two lists `LC` and `LP` using the method described in Section 7.6.

2. Read the next 32 bytes from B, and assign them to *salt*. If the read fails, return `null`.

3. Using `LC`, compute the size required for `iSeedInfo`, and read this number of bytes from B. If the read fails, and return `null`.

4. Using `LC`, compute the size required for `cvInfo`, and read this number of bytes from B. If the read fails, and return `null`.

5. Read u 5-tuples from B, and append them to Z. Recompute the length of `seedInfo`, note that this will be same for all 5-tuples. When reading the i-th value, let $(t_i, P_i) = (\texttt{LC}[i], \texttt{LP}[i])$. If any of the reads below fail, return `null`.

   (a) Read `seedInfo` from B,

   (b) If $P_i \neq N - 1$ read `aux`, which is $\lceil 3rs/8 \rceil$ bytes from B. If $3rs$ is not an integer number of bytes, and the padding bits to the next byte boundary are non-zero, return `null`.

   (c) Read `masked_key` from B, which is $S/8$ bytes

   (d) Read `msgs`, which is $\lceil 3rs/8 \rceil$ bytes from B. If $3rs$ is not an integer number of bytes, and the padding bits to the next byte boundary are non-zero return `null`.

   (e) Read `C` from B, which is ℓ_H bytes.

6. Output ($\texttt{LC}, \texttt{LP}, salt, \texttt{iSeedInfo}, \texttt{cvInfo}, Z$).

8 Additional Considerations

8.1 Signing Large Messages

Note that the sign operation makes two passes over M, once to generate the per-signature randomness, and once when computing the challenge. In applications where this cost is prohibitive, it is recommended to first hash M, and pass $H(M)$

to the signature algorithm specified here. The function H must be collision resistant, and the performance of Picnic signatures is only weakly affected by the output length. Implementations that pre-hash M should use SHAKE-256 with 512-bit digests, SHA3-512, or SHA-512.

A signing key used with pre-hashing must not be used without it, and vice-versa.

8.2 Test Vectors

The reference implementation[1] and the submission package for the NIST Post-Quantum Standardization process contain test vectors that implementations may use to verify conformance with this specification. The test vectors contain serialized versions of Picnic key pairs, messages and the corresponding Picnic signature. The intermediate values list the individual components of the signature, that should be produced after deserialization.

Note that key generation tests the correctness of an implementation's LowMC implementation, and in particular, that all of the constants required by LowMC are correct. In order to test the output of signing against a known value, implementations must use the de-randomized implementation specified here (§6.2, Step 2 and 7.1, Step 2), where the per-signature ephemeral random values are derived from the signer's secret key and the message to be signed (as opposed to being randomly generated).

8.3 Randomized Signatures

Signatures are specified in this document with a deterministic implementation, where the per-signature ephemeral random values are derived from the signer's secret key and the message to be signed (as opposed to being randomly generated). See §6.2, Step 2 and 7.1, Step 2). This helps testing against known values, can simplify debugging, and mitigates the risk of creating signatures with a poor random number generator (RNG), provided the signer's private key pair was generated with a strong RNG.

However, a deterministic implementation may be more vulnerable to certain kinds of side-channel and fault attacks. In some of these attacks the attacker collects multiple noisy observations of the implementation, then combines this information to recover information about the secret key. Some attacks in this class can be mitigated by randomizing each run. Similarly, a fault that causes a message bit to be flipped after the per-signature randomness is derived, and before the challenge is computed

[1] Available online at https://github.com/Microsoft/Picnic.

will cause different messages to be signed with the same randomness (i.e., signatures for the original and faulted messages will use the same randomness).

For implementations seeking to randomize the signature function, it is recommended to use the deterministic derivation method described here, but to append a $2S$-bit random value to the KDF input. This randomizes the signature, but provides a hedge against a poor RNG. In the extreme, when the poor RNG outputs 0, security degrades to the deterministic case, instead of collapsing entirely as when the RNG outputs are used directly. See [AOTZ20] for a security analysis of this construction against fault attacks.

8.4 MPC Simulation Errors During Signing

When creating a signature with the `picnic3` parameter sets, the MPC simulation step should always succeed (Step 4f). However, in exceptional cases like a fault attack or a corrupted private key, it may produce incorrect results. Since the signer can efficiently detect this, by comparing the MPC output to their public key, this check is recommended. If a simulation error is detected, the signer should abort.

A similar check can be added when signing with the `picnic` parameter sets, at Step 3(c)iv.

References

[Alb17] Martin Albrecht. m4ri: Further reading. m4ri Wiki, 2017. Accessed June 2017.

[AOTZ20] Diego F. Aranha, Claudio Orlandi, Akira Takahashi, and Greg Zaverucha. Security of hedged Fiat-Shamir signatures under fault attacks. To appear at EUCROCRYPT, 2020. https://eprint.iacr.org/2019/956.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, 2015.

[ARS+16] Martin Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. Cryptology ePrint Archive, Report 2016/687, 2016.

[CDG+17a] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. Cryptology ePrint Archive, Report 2017/279 and ACM CCS 2017, 2017.

[CDG+17b] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1825–1842. ACM Press, October / November 2017.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. In *USENIX Security*, 2016.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[Nat16] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, December 2016. https://beta.csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[NIS15] NIST. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. National Institute of Standards and Technology (NIST), FIPS PUB 202, U.S. Department of Commerce, 2015.

[Tie17] Tyge Tiessen. LowMC reference implementation, September 2017. Available at https://github.com/LowMC/lowmc, HEAD was 3994bc857661ac33134b36163b131a215f0fe9c3 when constants were generated.

A Change History

Version 1.0 – Version 1.1 Version 1.1 updates generation of random tapes used in signing and verification to prevent a birthday attack on the seed. Given D signatures from one or more signers, an attacker can guess a seed, re-derive the tape, and compare it to about $2^7 D$ tapes to find a match and recover the seed value. Given the seed value, the attacker can solve for the signer's secret key. The changes add a random salt and additional inputs to derivation, so that no two tapes are derived with identical additional inputs (with overwhelming probability). The attack was reported by Itai Dinur and Niv Nadler. This change breaks interoperability with Version 1.0.

Version 1.1 – Version 2.0 Version 2.0 adds the new parameter sets `picnic2-L1-FS`, `picnic2-L3-FS`, and `picnic2-L5-FS`, that replace ZKB++ with an alternative zero-knowledge proof, from [KKW18]. The spec was re-organized to group the description of algorithms required to implement the existing and new parameter sets. The existing parameter sets remain interoperable with Version 1.1.

Version 2.0 – Version 2.1 Version 2.1 changes the salt length from S bits to 256 bits for all parameter sets. Minor changes were made to the seed tree construction in the Picnic2 parameter sets. All parameter sets no longer interoperate with previous versions of this specification.

Version 2.1 – Version 3.0 In Version 3.0 parameter sets `picnic-X-FS` and `picnic-X-UR` (where `X` is `L1`, `L3` and `L5`) are unchanged and interoperate with Version 2.1. Version 3.0 adds six new parameter sets. The first three, `picnic3-X`, are replacements for `picnic2-X-FS`. Picnic3 is similar to Picnic2, as both use the KKW proof protocol, but Picnic3 makes changes to the LowMC parameters and the MPC protocol, in order to reduce the CPU cost of signing and verification. The new LowMC parameters use a full S-box layer, which means they require fewer rounds rounds for comparable security. The other three new parameter sets, `picnic-X-full`, use the ZKB++ proof system, and the LowMC parameters with a full S-box layer, as in Picnic3.