The Picnic Signature Scheme

Specification

Contact: Greg Zaverucha (`gregz@microsoft.com`)

September 29, 2017
DRAFT Version 0.8

Contents

1 Introduction

This document specifies the Picnic public-key digital signature scheme. 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 Scheme

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+17], and the additional documentation submitted to the NIST Post-Quantum Standardization process.

The public key in Picnic is the pair (C, p) where $C = E_{sk}(p)$, and E is a block cipher. This document specifies E as the block cipher 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 a specialized version of ZKBoo [GMO16], called ZKB++. Informally, the prover simulates a multiparty computation protocol (MPC) that allows players to jointly compute $E_{sk}(p)$, when each player has a share of sk. For Picnic, the number of players is always three. The idea is to have the prover commit to the simulated state and transcripts of all players, then have the verifier "corrupt" a random subset of the simulated players by seeing their complete state. The verifier then checks that the computation was done correctly from the perspective of the corrupted players, and if so, he has some assurance that the output is correct. The MPC protocol ensures that corrupting any two of the three players 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.

1.2 Contributors

The Picnic signature algorithm was designed by the following team.
Melissa Chase, Microsoft
David Derler, Graz University of Technology
Steven Goldfeder, Princeton
Claudio Orlandi, Aarhus University
Sebastian Ramacher, Graz University of Technology
Christian Rechberger, Graz University of Technology & DTU
Daniel Slamanig, AIT Austrian Institute of Technology
Greg Zaverucha, Microsoft

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 blocksize, in bits.
k	The LowMC key size, in bits. This is also the signing key.
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.
ℓ_H	The output length of H, in bytes.
$\oplus$	the binary exclusive or (XOR) of equal-length bitstrings.

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 a non-interactive MPC protocol. 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 $n \times k$ 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 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 outputs in different uses. When computing commitments we use H_0 defined as $H_0(x) = H(\texttt{0x00}||x)$ and when computing the challenge we use H_1 defined as $H_1(x) = H(\texttt{0x01}||x)$. The UR parameter sets also use H when computing the function G (defined in Section 4.5.6), and here we use H_2 defined as $H_2(x) = H(\texttt{0x02}||x)$. Before each use of a seed value, used in multiple places, we hash it before use with H_2 in one instance and H_4, H_5 in the others, which prepend the bytes `0x02`, `0x04` and `0x05` respectively.

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 a 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.

3.4 Views

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

`view.iShare` The input key share of this player, k 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 player, k 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.

4 The Picnic Signature Algorithm

This section describes the parameter sets for Picnic, and the three main operations key generation, signing and verifying.

4.1 Parameters

Table 2 gives parameters for three security levels L1, L3 and L5, as described in [oST16], corresponding to the security of AES-128, AES-192 and AES-256 (respectively). For each of the three security levels there are two possible signature algorithms, one based on the Fiat-Shamir transform (FS), and one based on the Unruh

(UR) transform. For discussion of the differences between these two variants, see [CDG+17].

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.

Parameter Set	S	n	k	s	r	Hash/KDF	Digest length	T
`picnic-L1-FS` `picnic-L1-UR`	128	128	128	10	20	SHAKE128	256	219
`picnic-L3-FS` `picnic-L3-UR`	192	192	192	10	30	SHAKE256	384	329
`picnic-L5-FS` `picnic-L5-UR`	256	256	256	10	38	SHAKE256	512	438

Table 2: Parameters by security level.

4.2 Key Generation

This section describes how to generate a signing key pair. The public key is denoted $pk = (C, p)$ and the secret key is denoted sk. The input to key generation is a security level (one of $S = 128$, 192 or 256). Note that for a key pair of security level S it is technically possible to use it with both signature algorithms defined at this level, e.g., a key pair created with the 128-bit parameter set may be used with both `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 k-bit string sk.
2. Using LowMC with the parameters given in Table 2, compute the encryption of p with sk, $C = E(sk, p)$.
3. Output: The pair (sk, pk). The secret key is sk, and the public key pk is (C, p).

4.3 Signing Operation

The functions `matrix_mul`, `mpc_sbox`, `mpc_xor`, `mpc_and` and `H3` used to specify sign are specified in later sections (Sections 4.5.4, 4.5.1, 4.5.3, 4.5.2 and 4.5.5 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), and a list of seeds `seeds[0..T-1][0..2]`. 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. 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 $3T(S/8)$ (three seeds for each of T iterations, each of size $S/8$ bytes).

   The test vectors associated with this document will use this method to simplify testing. However, the specific method of generating `seed` does not affect interoperability, and implementations may differ (e.g., by choosing the seeds uniformly at random, using an alternative derivation method, or including alternative inputs to derivation).

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. The seed is hashed with H_2 then the digest and the output length are concatenated and input to the KDF. The output length is encoded as a 16-bit little-endian integer. Tape `rand[0]` and `rand[1]` have length $k + 3rs$ bits, and tape `rand[2]` has length $3rs$ bits. 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 k bits:

      i. `x[0]` = first k bits of tape `rand[1]`

      ii. `x[1]` = first k bits of tape `rand[2]`

      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). For `i` from 0 to 2:
      `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 4.5.4.
      B. Apply substitution layer (s-boxes) to `state`:
         `state = mpc_sbox(state, rand, views[t])`
         The function `mpc_sbox` is defined in Section 4.5.1.
      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 4.5.3.
      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],]
     pk, M)
```

   The function H3 is defined in Section 4.5.5, 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 $(e_0, \ldots, e_{t-1})$ 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 $(e, b_0, \ldots, b_{t-1}, z_0, \ldots, z_{t-1})$ as described in Section 5.1 and output it as the signature.

4.4 Verification Operation

This section describes the Verify operation, to verify a signature created by the Sign operation in Section 4.3. The functions `matrix_mul`, `mpc_sbox_verify`, `mpc_xor`, `mpc_and` and `H3` used to specify verify are specified in later sections (Sections 4.5.4, 4.5.1, 4.5.3, 4.5.2 and 4.5.5 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 $(e, b_0, \ldots, b_{t-1}, z_0, \ldots, z_{t-1})$ as described in Section 5.2. If deserialization fails, reject the signature and output `invalid`. Write e as $(e_0, \ldots, e_{t-1})$ 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][2]` to recompute the random tape `rand[1]`.
      iii. Set `view[0].iShare` and `x[0]` to the first k bytes of `rand[0]`.
      iv. Set `view[1].iShare` and `x[1]` to the first k bytes 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 k bytes 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 k bytes 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 4.5.4.

  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],]
        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`.

4.5 Supporting Functions

The Sign (§4.3) and Verify (§4.4) operations use similar functions to simulate the MPC protocol used in the proof of knowledge. This section describes these functions.

4.5.1 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 r, 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 4.3. 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*r - 1)
    for j from 0 to 2
        a[j] = state[j][n - 1 - i - 2]
        b[j] = state[j][n - 1 - i - 1]
        c[j] = state[j][n - 1 - 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][n - 1 - i - 2] = a[j] XOR bc[j]
        state[j][n - 1 - i - 1] = a[j] XOR b[j] XOR ca[j]
        state[j][n - 1 - i] = a[j] XOR b[j] XOR c[j] XOR ab[j]
```

4.5.2 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
```

4.5.3 MPC XOR operation: `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..2][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.)

4.5.4 Binary Vector-Matrix Multiplication: `matrix_mul`

This function computes a vector-matrix product, with elements in GF(2). For signing, three vectors x, y, and z in $\mathrm{GF}(2)^k$ are input along with a single matrix $M \in \mathrm{GF}(2)^{k\times k}$, and three vectors xM, yM and zM in $\mathrm{GF}(2)^k$ are output. For signature verification, only x and y are input, and xM and yM are output. The pseudocode below is modified for verification by omitting lines depending on z.

The function parity(v) is the usual parity function: on input a vector v, of length k, it returns 1 if the number of 1 bits in v is odd, and zero otherwise. It can be implemented as $v_0 \oplus v_1 \oplus ... \oplus v_{k-1}$.

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: three k-bit vectors x, y, z, a k-bit by k-bit matrix M
Output: three k-bit vectors $a = xM$, $b = yM$ and $c = zM$
Pseudocode:

```
tempA, tempB, tempC are k-bit vectors
for i = 0 to k - 1
    for j = 0 to k - 1
```

```
        tempA[j] = x[j] AND M[i][j]
        tempB[j] = y[j] AND M[i][j]
        tempC[j] = z[j] AND M[i][j]
   a[k - 1 - i] = parity(tempA)
   b[k - 1 - i] = parity(tempB)
   c[k - 1 - i] = parity(tempC)
Output (a,b,c)
```

Notes

1. If inputs and outputs may overlap (e.g., when computing $x = xM$) a temporary variable is required for the output.
2. There are many ways to compute this function, implementations may use an alternative algorithm for better efficiency. For example, see [Alb17].

4.5.5 Computing the challenge: `H3`

The function `H3` hashes an arbitrary length bitstring to a length T output in $\{0, 1, 2\}$ (i.e., `H3` : $\{0,1\}^* \rightarrow \{0,1,2\}^t$). 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 $H_1(x) = H(0x01||x)$.

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 $h = H(h)$ and return to Step 2.

4.5.6 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:

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

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

5 Serialization

In this section we specify how to serialize and deserialize Picnic keys and signatures.

5.1 Serialization of Signatures

This section specifies how to serialize signatures created in Section 4.3.

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 $(e, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$, as computed in Section 4.3, 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. 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, 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.

3. Output B.

5.2 Deserialization of Signatures

This section describes how to deserialize a byte array created by Section 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 $(e, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$, as computed in Section 4.3, 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 §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. 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 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. Recall that the length of the transcript is $3rs$ bits (where r and s are specified in Table 2).
      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.

3. Output $(e, b_0, \ldots, b_{T-1}, z_0, \ldots, z_{T-1})$.

5.3 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, are both S bits long, and S is guaranteed to be a multiple of eight. 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 S bits of sk, as a byte array. 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.

6 Additional Considerations

6.1 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 must 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.

6.2 Test Vectors

The reference implementation 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 (§4.3, 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).

References

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

[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+17] 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.

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

[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.

[oST16] 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.

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