Post-quantum cryptography proposal: ThreeBears

Inventor, developer and submitter
Mike Hamburg
Rambus Security Division
E-mail: mhamburg@rambus.com
Telephone: +1-415-390-4344
425 Market St, 11th floor
San Francisco, California 94105
United States

Owner
Rambus, Inc.
Telephone: +1-408-462-8000
1050 Enterprise Way, Suite 700
Sunnyvale, California 94089
United States

December 2, 2017

Contents

1 Introduction

This is the specification of the ThreeBears post-quantum key encapsulation mechanism.

ThreeBears is based on the Lyubashevsky-Peikert-Regev [LPR10] and Ding [DXL12] ring learning with errors (RLWE) cryptosystems. More directly, it is based on Alkim et. al's NewHope [ADPS15] and Bos et. al's Kyber [BDK+17], the latter of which uses module learning with errors (MLWE). We replaced the polynomial ring underlying this module with the integers modulo a generalized Mersenne number, thereby making it integer module learning with errors (I-MLWE), as in Gu's work [Chu17]. We also use forward error correction, like Saarinen's trunc8 and Hila5 [Saa16, Saa17].

ThreeBears's name comes from the fact that its modulus has the same "golden-ratio Solinas" shape as Ed448-Goldilocks [Ham15], and indeed some of the arithmetic code in its implementation is derived from Goldilocks' arithmetic code.

One of our goals with ThreeBears is to encourage exploration of potentially desirable but less conventional designs. This is why ThreeBears uses I-MLWE instead of MLWE; why the private key as only a seed; why we use explicit rejection; and why we omit the Targhi-Unruh hash.

1.1 System overview

At a high level, ThreeBears works like [LPR10]. All computations take place modulo N, which is a large generalized Mersenne prime. A first party, Alice, chooses a uniformly random matrix M and random vectors a and ϵ_a from a "small" distribution χ. She sends M and $Ma + \epsilon_a$ to a second party, Bob. Bob likewise chooses vectors b and ϵ_b from the same distribution χ, and sends back $b^\top M + \epsilon_b^\top$. Then Alice can compute $C_a := b^\top Ma + \epsilon_b^\top a$, and Bob can compute $C_b := b^\top Ma + b^\top \epsilon_a$. C_a and C_b are roughly equal if

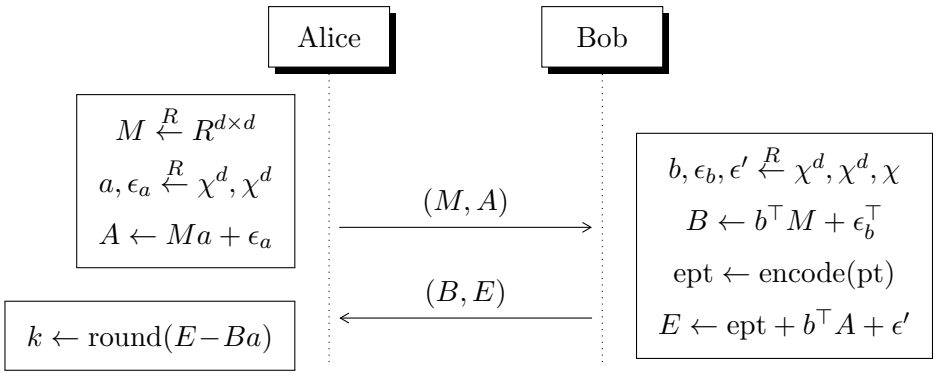

Figure 1: Simplified I-MLWE key exchange from [LPR10]

a, b, ϵ_a and ϵ_b are small enough. Bob uses this approximately-shared value to encrypt a plaintext: he encodes it so that small differences will not change its value, and adds his estimate of C_b to it. Alice subtracts her estimate C_a, and rounds off the hopefully-small error to recover the plaintext. This process is shown in Figure 1.

A few simple changes from [DXL12, ADPS16, BDK+17] make the system more practical. Alice need not send the large matrix M itself, since it's uniformly random: she can just send a seed which is hashed to make M. Also, Bob need not send E: he can send just the coefficients where he has actually encrypted plaintext bits, and he can round those coefficients to only a few bits each. Since the system can only send a small number of plaintext bits, it is best used as a key encapsulation mechanism (KEM). It sends a key, and then the key is used to encrypt an arbitrary message.

Finally, while this system is secure against eavesdropping, it needs modification to prevent chosen-ciphertext attacks [HGNP+03], namely the Fujisaki-Okamoto transform [FO99]. To make sure that Bob acted honestly, instead of sampling χ at random, he samples it pseudorandomly using a seed. Then instead of encrypting a message directly, he encrypts the seed. Alice decrypts the seed, checks that Bob's ciphertext was formed honestly, and then

uses the seed to generate a key to decrypt Bob's message. We use a variant of this transform, influenced by KYBER's interpretation of the Targhi-Unruh transform [BDK+17, TU16].

We specify versions of THREEBEARS with and without the Fujisaki-Okamoto transform. The variants without it are faster and slightly more secure against passive attacks, but they're insecure against chosen-ciphertext attacks. So they are suitable for ephemeral key exchange, but not for encryption with long-term keys.

2 Specification

Here is the detailed specification of THREEBEARS.

2.1 Notation

Integers Let $\mathbb{Z}$ denote the integers, and $\mathbb{Z}/N\mathbb{Z}$ the ring of integers modulo some integer N. For an element $x \in \mathbb{Z}/N\mathbb{Z}$, let $\mathrm{res}(x)$ be its value as an integer in $\{0, \ldots, N-1\}$.

For a real number r, $\lfloor r \rfloor$ ("floor of r") is the greatest integer $\leq r$; $\lceil r \rceil$ ("ceiling of r") is the least integer $\geq r$; and $\lfloor r \rceil := \lfloor r + 1/2 \rfloor$ is the rounding of r to the nearest integer, with half rounding up.

Sequences Let T^n denote the set of sequences of n elements each of type T. We use the notation $[\![a, b, \ldots, z]\!]$ or $[\![S_i]\!]_{i=0}^{n-1}$ for such a sequence. If S is a sequence of n elements and $0 \leq i < n$, then S_i is its ith element.

We describe our error-correcting code in terms of bit-sequences, i.e. elements of $\{0,1\}^n$. Let $a \oplus b$ be the bitwise exclusive-or of two bit-sequences. If a and b aren't the same length, we zero-pad the shorter sequence to the length of the longer one. We use the notation $\bigoplus S$ for the $\oplus$-sum of many sequences.

2.2 Encoding

Let $\mathcal{B}$ denote the set of bytes, i.e. $\{0, \ldots, 255\}$.

Public keys, private keys and capsules are stored and transmitted as fixed-length sequences of bytes, that is, as elements of $\mathcal{B}^n$ for some n which depends on system parameters. To avoid filling the specification with concatenation and indexing, we will define common encodings here.

The encodings used in ThreeBears are pervasively *little-endian* and *fixed-length*. That is, when converting between a sequence of smaller numbers (bits, bytes, nibbles...) and a larger number, the first (or rather, 0th) element is always the least significant. Also, the number of elements in a sequence is always fixed by its type and the parameter set, so we never strip zeros or use length padding.

An element z of $\mathbb{Z}/N\mathbb{Z}$ is encoded as a little-endian byte sequence B of length $\texttt{bytelength}(N) := \lceil \log_{256} N \rceil$, such that

$$\sum_{i=0}^{\texttt{bytelength}(N)-1} B_i \cdot 256^i = \mathrm{res}(z)$$

To decode, we simply compute $B_i \cdot 256^i \bmod N$ without checking that the encoded residue is less than N. This encoding is malleable, but capsules in our CCA-secure scheme are not malleable.

ThreeBears's encapsulated keys contain a sequence of 4-bit *nibbles*, i.e. elements of $\{0, \ldots, 15\}$. We encode this sequence by packing two nibbles into a byte[1] in little-endian order. So a nibble sequence $[\![s]\!]$ encodes as

$$[\![s_{2\cdot i} + 16 \cdot s_{2\cdot i+1}]\!]_{i=0}^{\lceil \mathrm{length}(s)/2 \rceil}$$

We will mention explicitly what part of the capsule is encoded as nibbles.

[1] These sequences always have even length, but if they didn't then the last nibble would be encoded in its own byte.

Description	Name	Value
Independent parameters:		
Specification version	`version`	1
Private key bytes	`privateKeyBytes`	40
Matrix seed bytes	`matrixSeedBytes`	24
Encryption seed bytes	`encSeedBytes`	32
Initialization vector bytes	`ivBytes`	0
Shared secret bytes	`sharedSecretBytes`	32
Bits per digit	`lgx`	10
Ring dimension	D	312
Module dimension	d	varies: 2 to 4
Noise variance	σ^2	varies: $\frac{1}{4}$ to 1
Encryption rounding precision	ℓ	4
Forward error correction bits	`fecBits`	18
CCA security	`cca`	varies: 0 or 1
Derived parameters:		
Radix	x	$2^{\texttt{lgx}}$
Modulus	N	$x^D - x^{D/2} - 1$
Clarifier	`clar`	$x^{D/2} - 1$

Table 1: ThreeBears global parameters

The same little-endian rules apply for converting between bit sequences and byte sequences. Any other tuple, vector or sequence of items is encoded as the concatenation of the byte encodings of those items.

2.3 Parameters

An instance of ThreeBears has many parameters. About half of these are lengths of various seeds, which are fixed according to security requirements. The list is shown in Table 1. These parameters are in scope in every function in this specification. For example, when we refer to d, we mean the d-parameter in the current parameter set.

System	d	cca	σ^2	Failure	Quantum security CCA	Lattice	Class
BabyBear	2	0	1	2^{-58}		152	II
		1	5/8	2^{-148}	122	141	II
MamaBear	3	0	7/8	2^{-51}		237	V
		1	1/2	2^{-147}	137	219	IV
PapaBear	3	0	3/4	2^{-52}		320	V
		1	3/8	2^{-188}	173	292	V

Table 2: ThreeBears recommended parameters. Security levels are given as the $\log_2$ of the estimated work to break the system using a lattice or chosen-ciphertext attack on a quantum computer.

System	`lgx`	D	d	`cca`	σ^2	Failure	Classical security CCA	Lattice
TeddyBear	9	240	1	1	3/4	2^{-58}		56
DropBear	10	312	2	1	2	2^{-6}	6	187

Table 3: ThreeBears toy parameters.

The parameters for the recommended instances are shown in Table 2. Each system has variants for CPA-secure and CCA-secure key exchange. Our primary recommendation is MamaBear. For each system, we estimated the failure probability, the difficulty of attacking the mode with lattice attacks, and (for CCA-secure variants) the difficulty of a chosen-ciphertext attack with a quantum computer. See Section 5 for a detailed analysis.

We also define two sets of toy parameters, shown in Table 3. TeddyBear is simply too small: it has dimension $1 \cdot 240$ compared to BabyBear's $2 \cdot 312$. We estimate that it will still take about 2^{56} effort to break this system, which is why we reduced the dimension from 312 to 240. On the other hand, DropBear should be secure against CPA attacks, but its failure rate of around 1.1% should make it vulnerable to CCA attacks.

2.4 Common subroutines

2.4.1 Hash functions

In order to make sure that the hash functions called by instances of ThreeBears are all distinct, they are prefixed with a parameter block `pblock`. This is formed by concatenating the independent parameters listed in Table 1, using one byte per parameter with the following exceptions: D is greater than 256, so it is encoded as two bytes (little-endian); and σ^2 is a real number where $0 < \sigma^2 \leq 2$, so it is encoded as $128 \cdot \sigma^2 - 1$. The total size of the parameter block is 14 bytes.

As an example, the parameter block for MamaBear in CCA-secure mode is

$$[\![1, 40, 24, 32, 0, 32, 10, 56, 1, 3, 63, 4, 18, 1]\!]$$

Since there are multiple uses of the hash function within ThreeBears, we also separate them with a 1-byte "purpose" p. For word-alignment purposes, we add a zero byte between the parameter block and the purpose. The hash function is therefore

$H_p(\text{data}, L) :=$ cSHAKE256(`pblock` $||$ $[\![0, p]\!]$ $||$ data, $8{\cdot}L$, "", "ThreeBears")

Here L is the length in bytes of the desired output. The `cSHAKE256` hash function is defined in [KjCP16]. We use only one personalization string to avoid polluting the `cSHAKE` namespace, and to enable precomputation of the first hash block.

2.4.2 Sampling

Uniform We construct the $d \times d$ matrix M by sampling each element separately. We do this with a function that expand a short seed, and coordinates $0 \leq i, j < d$, into a uniform sample mod N. This is shown in Algorithm 1.

Algorithm 1: Uniform and noise samplers

Function $\texttt{uniform}(\text{seed}, i, j)$ **is**

input : Seed of length `matrixSeedBytes` bytes; i and j in $[0 \,..\, d-1]$

output : Uniformly pseudorandom number modulo N

$B \leftarrow H_0(\text{seed} \,||\, [\![d \cdot j + i]\!],\ \texttt{bytelength}(N))$;

return B decoded as an element of $\mathbb{Z}/N\mathbb{Z}$;

end

Function $\texttt{noise}_p(\text{seed}, i)$ **is**

input : Purpose p; seed whose length depends on purpose; index i

require: σ^2 must be either $\begin{cases} \text{in } [0..\frac{1}{2}] \text{ and divisible by } \frac{1}{128} \\ \text{in } [\frac{1}{2}..1] \text{ and divisible by } \frac{1}{32} \\ \text{in } [1..\frac{3}{2}] \text{ and divisible by } \frac{1}{8} \\ \text{exactly } 2 \end{cases}$

output : Noise sample modulo N

$B \leftarrow H_p(\text{seed} \,||\, [\![i]\!],\ D)$;

for $j = 0$ **to** d **do**

// Convert each byte to a digit with var σ^2

sample $\leftarrow B_j$;

$\text{digit}_j \leftarrow 0$;

for $k = 0$ **to** $\lceil 2 \cdot \sigma^2 \rceil - 1$ **do**

$v \leftarrow 64 \cdot \min(1,\ 2\sigma^2 - k)$;

$\text{digit}_j \leftarrow \text{digit}_j + \left\lfloor \frac{\text{sample}+v}{256} \right\rfloor + \left\lfloor \frac{\text{sample}-v}{256} \right\rfloor$;

sample $\leftarrow$ sample $\cdot\ 4 \bmod 256$;

end

end

return $\sum_{j=0}^{D-1} \text{digit}_j \cdot x^j \bmod N$

end

Noise We will also need to sample noise modulo N from a distribution whose "digits" are small, of variance σ^2. The noise sampler is shown in Algorithm 1. It works by expanding a seed to one byte per digit, and then converting the digit to an integer with the right variance. With only one byte per digit we can only sample distributions with certain variances, as described in that algorithm's requirements.

2.4.3 Extracting bits from a number

In order to encrypt using ThreeBears, we need to extract bits from an approximate shared secret S mod N. Because our ring isn't cyclotomic, the digits of S don't all have the same noise: the lowest and highest bits have the least noise, and the middle ones have the most. We define a function $\texttt{extract}_b(S, i)$ which returns the top b bits from the coefficient with the ith-least noise, as shown in Algorithm 2.

Algorithm 2: Extracting the top b bits of the digit with the ith-least noise

Function $\texttt{extract}_b(S, i)$ **is**
    if i **is even then** $j \leftarrow i/2$;
    else $j \leftarrow D - (i+1)/2$;
    return $\lfloor \mathrm{res}(S) \cdot 2^b / x^{j+1} \rfloor$;
end

2.4.4 Forward error correction

ThreeBears uses forward error correction (FEC). Let $\texttt{FecEncode}_b$ and $\texttt{FecDecode}_b$ implement an error-correcting encoder and decoder, respectively, where the decoder appends $b = \texttt{fecBits}$ bits of error correction informatioon. Because b might not be a multiple of 8, and because the output of the FEC is encrypted on a bit-by-bit basis, we specify that the encoder and decoder operate on bit sequences. If $\texttt{fecBits} = 0$, then no

error correction is used:

$$\texttt{FecEncode}_0(s) = \texttt{FecDecode}_0(s) = s$$

The rest of this section describes a Melas FEC encoder and decoder which add 18 bits and correct up to 2 errors, roughly as in [LW87]. This FEC is used by all our recommended parameters.

Encoding Let $\texttt{seq}_b(n)$ be the b-bit sequence of the bits of an integer n, in little-endian order. For a bit a and sequence B, let

$$a \cdot B := [\![a \cdot B_i]\!]_{i=0}^{\texttt{length}(B)-1}$$

For bit-sequences R and s of length $b+1$ and b respectively, let

$$\texttt{step}(R, s) := [\![(s \oplus (s_0 \cdot R))_i]\!]_{i=1}^{b}$$

Let $\texttt{step}^i(R, s)$ denote the ith iterate of $\texttt{step}(R, \cdot)$ applied to s. Then $\texttt{FecEncode}_{18}$ appends an 18-bit syndrome as shown in Algorithm 3.

Algorithm 3: Melas FEC encode

Function $\texttt{syndrome}_{18}(B)$ **is**
  input : Bit sequence of length n
  output: Syndrome, a bit sequence of length 18.
  $P \leftarrow \texttt{seq}_{18+1}(\texttt{0x46231})$;
  $s \leftarrow 0$;
  for $i = 0$ **to** $n - 1$ **do** $s \leftarrow \texttt{step}(P, s \oplus [\![B_i]\!])$;
  return s;
end

Function $\texttt{FecEncode}_{18}(B)$ **is**
  return $B \mid\mid \texttt{syndrome}_{18}(B)$
end

Decoding Decoding is more complicated, because to locate two errors it must solve a quadratic equation. Let $Q := \texttt{seq}_{9+1}(\texttt{0x211})$. For 9-bit sequences a and b, define the 9-bit sequence

$$a \odot b := \bigoplus_{i=0}^{8} \left(b_{8-i} \cdot \texttt{step}^i(Q, a)\right)$$

The operations $\oplus$ and $\odot$ define a field with 2^9 elements, with additive identity 0 and multiplicative identity $\texttt{seq}_9(\texttt{0x100})$. That is, $\odot$ is Montgomery multiplication. Define $a^{\odot n}$ as the nth power of a under $\odot$-multiplication.

The rest of the decoding algorithm is shown in Algorithm 4.

Implementation This specification admits many optimizations. See the `melas_fec.c` from the Optimized_Implementation for a fast, short, constant-time implementation of the Melas FEC.

2.5 Keypair generation

We define key generation so that the private key is a uniformly random byte string. Key online exchange implementations might cache intermediate values, such as the private vector or matrix, but ThreeBears is fast enough that this isn't necessary.

2.6 Encapsulation

The encapsulation function is shown in Algorithm 6. It includes a deterministic version which is used for CCA-secure decapsulation. As with `Keypair`, `Encapsulate` simply passes a random seed and iv to `EncapsDet`.

In the CCA-secure implementation of encapsulation, the noise is derived from a seed, and the seed is used as plaintext, as required by the Fujisaki-Okamoto transform. But in the ephemeral implementation, the noise and plaintext are both derived from the seed using the hash function H_2. The

Algorithm 4: Melas FEC decode

Function $\texttt{FecDecode}_{18}(B)$ **is**

input : Encoded bit sequence B of length n, where $18 \leq n \leq 511$

output: Decoded bit sequence of length $n - 18$

// Form a quadratic equation from syndrome.

$s \leftarrow \texttt{syndrome}_{18}(B)$;

$Q \leftarrow \texttt{seq}_{9+1}(\texttt{0x211})$;

$c \leftarrow \texttt{step}^9(Q, s) \odot \texttt{step}^9(Q, \texttt{reverse}(s))$;

$r \leftarrow \texttt{step}^{17}\left(Q,\ c^{\odot 510}\right)$;

$s_0 \leftarrow \texttt{step}^{511-n}(Q, s)$;

// Solve quadratic for error locators using half-trace

$\texttt{halfTraceTable} \leftarrow [\![36, 10, 43, 215, 52, 11, 116, 244, 0]\!]$;

$\texttt{halfTrace} \leftarrow \bigoplus_{i=0}^{8} (r_i \cdot \texttt{seq}_9(\texttt{halfTraceTable}_i))$;

$(e_0, e_1) \leftarrow (s_0 \odot \texttt{halfTrace},\ (s_0 \odot \texttt{halfTrace}) \oplus s_0))$;

// Correct the errors using the locators

for $i = 0$ **to** $n - 18 - 1$ **do**

  if $\texttt{step}^i(Q, e_0) = \texttt{seq}_9(1)$ **or** $\texttt{step}^i(Q, e_1) = \texttt{seq}_9(1)$ **then**

    $B_i \leftarrow B_i \oplus 1$;

  end

end

return $[\![B_i]\!]_{i=0}^{n-18-1}$;

end

Algorithm 5: Keypair generation

Function `GetPubKey`(sk) **is**

input : Uniformly random private key sk of length `privateKeyBytes`

output: Public key pk

`// Generate the private vector`

for $i = 0$ **to** $d - 1$ **do** $a_i \leftarrow \texttt{noise}_1(\text{sk}, i)$;

`// Generate a random matrix, multiply and add noise`

matrixSeed $\leftarrow H_1(\text{sk}, \texttt{matrixSeedLen})$;

for $i, j = 0$ **to** $d - 1$ **do** $M_{i,j} \leftarrow \texttt{uniform}(\text{matrixSeed}, i, j)$;

for $i = 0$ **to** $d - 1$ **do**

  $A_i \leftarrow \texttt{noise}_1(\text{sk}, d + i) + \sum_{j=0}^{d-1} M_{i,j} \cdot a_j \cdot \texttt{clar}$

end

`// Output`

pk $\leftarrow$ (matrixSeed, $[\![A_i]\!]_{i=0}^{d-1}$);

return pk;

end

Function `Keypair`() **is**

sk $\leftarrow$ `RandomBytes`(`privateKeyBytes`);

return (sk, `GetPubKey`(sk));

end

reason is to avoid depending on circular security: in a quantum context it is difficult to prove that deriving the noise from the plaintext is secure, even in the random oracle model.

2.7 Decapsulation

The decapsulation algorithm, `Decapsulate`, takes as input a private key sk and a capsule. It returns either a shared secret or the failure symbol $\perp$, as shown in Algorithm 7.

Algorithm 6: Encapsulation

Function `EncapsDet`(pk, seed, iv) **is**

input : Public key pk

input : Uniformly random seed of length `encSeedBytes`

input : Uniformly random iv of length `ivBytes`

output: Shared secret; capsule

`// Parse the public key`

(matrixSeed, $[\![A_i]\!]_{i=0}^{d-1}$) $\leftarrow$ pk;

`// Generate ephemeral private key and make I-MLWE instance`

for $i = 0$ **to** $d-1$ **do** $b_i \leftarrow \texttt{noise}_2(\text{matrixSeed}||\text{seed}||\text{iv}, i)$;

for $i, j = 0$ **to** $d-1$ **do** $M_{i,j} \leftarrow \texttt{uniform}(\text{matrixSeed}, i, j)$;

for $i = 0$ **to** $d-1$ **do**

  $B_i \leftarrow \texttt{noise}_2(\text{seed}, d+i) + \sum_{j=0}^{d-1} M_{j,i} \cdot b_j \cdot \texttt{clar}$;

end

`// Form plaintext; encrypt using approximate shared secret`

$C \leftarrow \texttt{noise}_2(\text{seed}, 2 \cdot d) + \sum_{j=0}^{d-1} A_j \cdot b_j \cdot \texttt{clar}$;

if `CCA` **then** pt $\leftarrow$ seed;

else pt $\leftarrow H_2(\text{matrixSeed}||\text{seed}||\text{iv}, \texttt{encSeedBytes})$;

encpt $\leftarrow$ `FecEncode`(pt as a sequence of bits);

for $i = 0$ **to** length(encpt) $- 1$ **do**

  encr$_i \leftarrow \texttt{extract}_4(C, i) + 8 \cdot \text{encoded_seed}_i \bmod 16$;

end

`// Output`

shared_secret $\leftarrow H_2(\text{matrixSeed}||\text{pt}||\text{iv}, \texttt{sharedSecretBytes})$;

capsule $\leftarrow \left([\![B_j]\!]_{j=0}^{d-1},\ \text{nibbles}\ [\![\text{encr}_i]\!]_{i=0}^{\text{length(pt)}-1}, \text{iv}\right)$;

return (shared_secret, capsule);

end

Function `Encapsulate`(pk) **is**

(seed, iv) $\leftarrow$(`RandomBytes`(`encSeedBytes`), `RandomBytes`(`ivBytes`));

return `EncapsDet`(pk, seed, iv);

end

Algorithm 7: Decapsulation

Function `Decapsulate`$(\mathrm{sk}, \mathrm{capsule})$ **is**

input : Private key sk, capsule

output: Shared secret or $\perp$

// Unpack private key and capsule

for $i = 0$ **to** $d - 1$ **do** $a_i \leftarrow \mathtt{noise}_1(\mathrm{sk}, i)$;

$\left(\llbracket B_j \rrbracket_{j=0}^{d-1},\ \text{nibbles } \llbracket \mathrm{encr}_i \rrbracket,\ \mathrm{iv}\right) \leftarrow \text{capsule}$;

// Calculate approximate shared secret and decrypt seed

$C \leftarrow \mathtt{noise}_2(\text{seed}, 2 \cdot d) + \sum_{j=0}^{d-1} B_j \cdot a_j \cdot \mathtt{clar}$;

for $i = 0$ **to** length(encr) **do**

  $\text{encoded\_seed}_i \leftarrow \left\lfloor \frac{2 \cdot \mathrm{encr}_i - \mathtt{extract}_5(C, i)}{2^\ell} \right\rceil$

end

seed $\leftarrow$ `FecDecode`(encoded_seed);

if `CCA` **then**

  // Re-encapsulate to check that capsule was honest

  $(\text{shared\_secret}, \text{capsule}') \leftarrow \mathtt{EncapsDet}(\mathtt{GetPubKey}(\mathrm{sk}), \text{seed}, \mathrm{iv})$;

  if $\text{capsule}' \neq \text{capsule}$ **then return** $\perp$;

  else return shared_secret;

else

  // Don't check: just calculate the shared secret

  matrixSeed $\leftarrow H_1(\mathrm{sk}, \mathtt{matrixSeedLen})$;

  shared_secret $\leftarrow H_2(\text{matrixSeed}||\text{seed}||\mathrm{iv}, \mathtt{sharedSecretBytes})$;

  return shared_secret

end

end

3 Design Rationale

We based our overall design on KYBER [BDK+17]. We liked that module-LWE allows parameters to be changed without changing too much code. From that starting point we made many changes, as described in this section.

3.1 Integer MLWE problem

We originally studied the integer version of the MLWE problem simply because it hadn't received much attention before. We expected it to be strictly worse than polynomial MLWE, and thus not worthy of a NIST submission. But in fact, I-MLWE gives a range of desirable parameter sets which are comparable to polynomial MLWE in efficiency, ease of implementation, and (hopefully) security.

Private key as seed We chose to make the private key merely a seed, because the key generation process is so fast that we might as well save on storage. Applications which have lots of memory and only need keys ephemerally can cache the intermediates a_i, but that's an implementation decision. Furthermore, public-key regeneration can be efficiently fused with re-encryption. This is because to re-encrypt, the recipient needs to compute

$$B = b^\top M + \epsilon_b^\top,\ C = b^\top(Ma + \epsilon_a) + \epsilon_b'$$

The latter term can be rewritten as $(b^\top M)a + b^\top\epsilon_a + \epsilon_b'$, which costs a total of $d\cdot(d+2)$ multiplications. Caching the public key component $A := Ma+\epsilon_a$ would only reduce this to $d\cdot(d+1)$ multiplications, because $b^\top M$ has to be computed anyway.[2]

[2]The recipient could alternatively try to verify B, C by checking that $C \stackrel{?}{=} b^\top A + \epsilon_b'$ and $(B - \epsilon_b^\top)\cdot a \stackrel{?}{=} b^\top A$, which would cost only $2\cdot d$ multiplications. However, this would also verify with any value B' where $B' - B$ is orthogonal to the secret key a, and giving the adversary this information would be a huge boost to cryptanalysis. While our ring's

Explicit rejection We had two options on how to deal with decapsulation failures. We could reject explicitly by returning a special failure symbol "$\perp$". Or we could reject implicitly by returning a pseudorandom key, which would cause later protocol steps to fail.

Theoretical work such as [SXY17] and [JZC+17] suggest that implicit rejection is easier to analyze, and it provides a slightly simpler API. However, explicit rejection is simpler and faster, and has one fewer target for side-channel analysis.

On balance, we have decided to go with explicit rejection.

No Targhi-Unruh hash We chose not to use an extra hash (as was done in [TU16]), because we believe that it adds no security and is merely an artifact of the proof. It is not required by [SXY17] or [JZC+17]. Our work in progress shows that it doesn't significantly impact security for ThreeBears even though we use explicit rejection.

No hashing of public keys and ciphertexts We chose not to hash the entire public key or entire ciphertext. Doing so would complicate and slow down the implementation; would require more memory; and would prevent efficient fusing of key generation and decryption. While hashing the public key and ciphertext would be more conservative, our proof shows that it is not required for CCA2 security.

However, we must hash some part of the public key into the encryption seed to prevent batch failure attacks. Since the purpose of the matrix seed is to prevent batch attacks in general, we chose to hash the matrix seed into the encryption seed.

Of course, in a key exchange protocol the messages must be authenticated somehow, and hashing them is one way to do that. But it composes better to do this at the protocol layer, instead of having the KEM do it.

lack of zero divisors probably makes it hard to find such a B', we cannot prove that it is as hard as a full lattice attack, so we don't allow this verification algorithm.

	CPA-secure			CCA-secure			
d	σ^2	Failure	Lattice	σ^2	Failure	CCA	Lattice
2	5/8	2^{-54}	141	3/8	2^{-133}	120	130
3	1/2	2^{-57}	219	9/32	2^{-153}	138	202
4	7/16	2^{-56}	297	7/32	2^{-181}	162	269

Table 4: Alternative parameters without error correction. $D = 312, x = 2^{10}$.

Melas code We thought the potential improvements from Saarinen's error correction [Saa16, Saa17] were too good not to investigate. In the context of ThreeBears, they give a significant improvement, which must be traded off against the increase in complexity.

We wanted to design the strongest possible error-correcting code in the least amount of space and code complexity. The obvious choice was a BCH code, which would add $9n$ bits to correct n errors. This would enable us to correct up to 6 errors, since we have $312 - 256 = 56$ bits of space, but decoding many errors in constant time is rather complex. One error needs only LFSR evaluation; two errors require solving a quadratic; three require Gaussian elimination; and four or more require more advanced root-finding or brute-force evaluation. So a two-error-correcting code seemed like a good tradeoff between complexity and correction ability, and the Melas BCH code seemed like the simplest variant.

Our Melas implementation has small code and memory requirements, runs in constant time, and is so fast that its runtime is almost negligible. Its downsides are increased complexity, and a correspondingly wider attack surface for side-channel and fault attacks.

Table 4 shows alternative parameters with no forward error correction. Table 5 compares the effectiveness of BCH error-correcting codes which would correct n errors using $9n$ bits. This allows more noise, and therefore increases security at the cost of complexity. Preliminary work suggests that a soft parity code should be almost as effective as a 1-error-correcting code, which may be an interesting avenue for future work.

Errors corrected	0	soft	1	2	3	4	5	6
Variance in 32nds	9	12	13	16	18	20	22	24
Lattice security	202	210	213	219	223	227	230	233

Table 5: Effectiveness of error correction to increase security. Alternative parameters with more or less error correction, with noise variance and lattice security vs. a quantum computer. $D = 312, d = 3, x = 2^{10}$, CCA_2-secure.

3.2 Parameter choices

Seeds The seed sizes in ThreeBears are designed for an overall 2^{256} or larger search space. Thus the encryption seeds and transported keys are 256 bits. We don't believe that multi-target key recovery attacks are a problem, since they would take $2^{256}/T$ time on a classical machine to recover one of T keys by brute force, and do not admit a significant quantum speedup. But protecting key generation is almost free, just by setting the private to 320 bits (40 bytes). This means a classical multi-target key-recovery attack on 2^{64} keys would take 2^{256} effort.

Since encryption seeds are 256 bits, there is a multi-target attack when someone encrypts many ciphertexts under a single key. This can be mitigated by attaching an initialization vector (IV) to each ciphertext. We specify how to do this, but in our recommended parameters the IV is zero bytes long (i.e. unused). This is because we don't think that that the multi-target brute force attack is a meaningful risk.

We chose a 192-bit matrix seed so that matrix seeds will almost certainly never collide even with 2^{64} honestly-generated keys. It would even be safe to use a 128-bit matrix seed, since at worst a k-way collision would enable a multi-target attack on k public keys at once.

Modulus We chose N to be prime to rule out attacks based on subrings. We would have liked for N to be a Fermat prime, but there are no Fermat primes of the right size. The next obvious choice would be a Mersenne prime

	CPA-secure			CCA-secure			
d	σ^2	Failure	Lattice	σ^2	Failure	CCA	Lattice
2	3/4	2^{-64}	117	9/16	2^{-108}	98	112
3	5/8	2^{-63}	184	3/8	2^{-156}	141	170
4	9/16	2^{-59}	251	5/16	2^{-166}	151	231

Table 6: Alternative parameters with $D = 260, x = 2^{10}$, soft parity FEC

$2^p - 1 = 2^k \cdot x^D - 1$, where k at best k can be ± 1: k cannot be 0 because p is prime. Therefore reduction modulo this prime would at least double the noise amplitude and quadruple its variance.

So as far as we know, the best prime shape is a "golden-ratio Solinas" prime $x^D - x^{D/2} - 1$. Multiplying by clar $:= x^{D/2} - 1$ and reducing modulo this prime will amplify variance by 3/2 in the center digits. With this amplification we needed $x \geq 2^{10}$ for an acceptable failure probability, and $D \geq 256$ to transport a 256-bit key. This left the primes

$$2^{2600} + 2^{1300} - 1 \text{ and } 2^{3120} - 2^{1560} - 1$$

We chose the latter for several reasons:

- The smaller prime is slightly greater than a power of 2. This would require implementations to handle low-probability corner cases which cannot happen with the larger prime.
- The larger prime better matches the NIST target security levels.
- The larger prime allows us to use FEC. The smaller prime would accommodate a soft parity code, but no more.
- The larger prime is slightly simpler to implement efficiently.

The smaller prime would have enabled finer granularity in security level, but we thought that the other considerations were more important. Potential parameterizations with the smaller prime are shown in Table 6.

If ThreeBears's small noise variance becomes a concern, then we can use the same large modulus with $D = 260$ and $x = 2^{12}$ and much larger

noise. This would be useful if the hybrid attack [BGPW16] becomes a major threat. But according to current estimates, it is much more difficult to attack $D = 312$ and $x = 2^{10}$.

Rounding precision The encryption rounding precision ℓ is a tradeoff. Larger ℓ adds to ciphertext size, but it decreases the failure probability. This in turn allows more noise to be added, which increases security. According to our security estimates, the greatest ratio of security strength to ciphertext size is achieved with $\ell = 3$, but with $\ell = 4$ only slightly less. We chose $\ell = 4$ because it's simpler to implement.

Variance We chose the noise variance as a simple dyadic fraction. We initially aimed to set the failure probability for CPA-secure instances below 2^{-50}, and the CCA-security of CCA-secure instances to at least 2^{128}. This exceeds the CCA-security of most constructions using a 128-bit block cipher or MAC tag

However, two considerations caused us to adjust the variance. For BabyBear, we decided that between a 128-bit passive lattice attack and a 128-bit chosen-ciphertext attack (requiring perhaps 2^{100} chosen messages), the former is both a much greater threat and much more likely to improve over time. So we tweaked the parameters to boost lattice security slightly, dropping CCA security to around 2^{120}.

We also were concerned about an adversary using Grover's algorithm to speed up chosen-ciphertext attacks. We know that Grover can provide a small speedup by finding ciphertexts with larger than expected noise. We have accounted for this in our CCA security estimates, and we expect it to be the best attack along these lines. But our QROM security analysis doesn't rule out a full square-root speedup. In order to hedge our bets, we slightly reduced the variance of the CCA-secure `PapaBear` instance, so that even if there is a square-root speedup it will require 2^{94} effort per key to break.

There are some disadvantages to using so small a variance, such as hybrid attacks [BGPW16]. But Micciancio-Peikert [MP13] suggests that even binary noise should be safe so long as the number of LWE samples available to the adversary is small. In our case the adversary sees only $d+1$ ring samples of dimension D, which is at least small enough that no known attacks apply.

3.3 Primary recommendation

With increasing focus on post-quantum cryptography, we expect lattice and MLWE cryptanalysis to attract more attention than they did before. The art of breaking these systems may improve considerably. In addition, integer MLWE is an entirely new variant of the problem, and might be significantly easier or harder than polynomial MLWE. So we wanted to be conservative in our recommendations.

We have estimated the effort to break BabyBear at around 2^{156} for a classical computer and 2^{141} effort for a quantum computer, which makes it a Class II cryptosystem in NIST's terminology. But post-quantum cryptography is currently a field for very conservative users. Since I-MLWE has seen little analysis, we are not confident enough in BabyBear's security to make it our primary recommendation, but it could still be used for lightweight devices. After a few years of cryptanalysis it might be mature enough.

The stronger MamaBear seems comfortably out of reach of known attacks, requiring 2^{242} effort with a classical computer or 2^{219} effort with a quantum computer, which would put it in Class IV. This seems sufficiently conservative, and is our primary recommendation.

PapaBear demonstrates that ThreeBears can reach Class V, and can be used for applications that require a very high security level.

4 Security analysis

4.1 The I-MLWE problem

The I-MLWE problem [Chu17] is to distinguish between two distributions, which depend on a ring R, dimensions d and e, and a noise distribution χ. We choose $R := \mathbb{Z}/N\mathbb{Z}$ and χ_{σ^2} as the distribution

$$\sum_{i=0}^{d-1} c_i x^i \quad \text{where} \quad x^i \leftarrow \begin{cases} -1 & \text{with probability } \sigma^2/2 \\ 0 & \text{with probability } 1-\sigma^2 \\ 1 & \text{with probability } \sigma^2/2 \end{cases}$$

when $\sigma^2 \leq 1/2$, and $\chi_{\sigma^2-1/2} + \chi_{1/2}$ otherwise. This is the distribution sampled by `noise`. Let $\chi_{\sigma^2}^d$ be the distribution of vectors formed by sampling χ_{σ^2} d times i.i.d. The $d \times e$ integer module learning with errors (I-MLWE$_{d\times e}$) problem is to distinguish the distributions:

$$\begin{aligned} (M, Ma + \epsilon_a) &: M \xleftarrow{R} (\mathbb{Z}/N\mathbb{Z})^{e\times d};\ a \leftarrow \chi_{\sigma^2}^d;\ \epsilon_a \leftarrow \chi_{\sigma^2}^e \\ (M, b) &: M \xleftarrow{R} (\mathbb{Z}/N\mathbb{Z})^{e\times d};\ b \xleftarrow{R} (\mathbb{Z}/N\mathbb{Z})^{e} \end{aligned}$$

We expect the difficulty of this problem to be similar to the traditional problem over cyclotomic rings.

4.2 The CCA$_2$ transform

Included with this submission is a proof sketch which analyzes ThreeBears' CCA$_2$ transform in the quantum random oracle model. It tentatively shows that for a quantum CCA$_2$ adversary $\mathcal{A}$ which uses cSHAKE as a quantum-accessible random oracle, there is a quantum algorithm $\mathcal{B}$ using approximately the same resources as $\mathcal{A}$, such that

$$\begin{aligned} \mathrm{Adv}_{\text{IND-CCA}_2}(\mathcal{A}) &\leq \sqrt{2(q+1)\cdot(\mathrm{Adv}_{\text{I-MLWE}}(\mathcal{B})q^2/2^{8\cdot\texttt{encryptionSeedBytes}})} \\ &+ 2q/\sqrt{2^{8\cdot\texttt{privateKeyBytes}}} + 4q\sqrt{\delta} + \text{negl.} \end{aligned}$$

where

- $\mathrm{Adv}_{\text{IND-CCA}_2}(\mathcal{A})$ is the KEM distinguishing advantage for $\mathcal{A}$.
- $\mathrm{Adv}_{\text{I-MLWE}}(\mathcal{B})$ is $\mathcal{B}$'s distinguishing advantage for $\text{I-MLWE}_{(d+1)\times d}$.
- q is the number of times the adversary calls cSHAKE.
- δ is the failure probability.

Our proof technique doesn't assume that the decryption queries are classical, and it doesn't place any limits other than feasibility on the number of decryption queries the adversary can make. Thus it uses a relatively strong attack model, though we haven't yet attempted to bound batch attacks.

Parsing the bound, the possible attacks on the system appear to be:

- Breaking I-MLWE. Our analysis is loose by a factor of about q for this.
- Grover's algorithm to discover the private key.
- Grover's algorithm on the encryption seed; our analysis is loose by a factor of about q for this.
- Grover's algorithm to find ciphertexts that are likely to cause failures. We believe that this attack is much weaker than $4q\sqrt{\delta}$ if the decryption queries are classical, because the attacker cannot recognize such ciphertexts (except insofar as more noise results in higher probability of failure).

5 Analysis of known attacks

Here is a more precise analysis of the best known attack strategies.

5.1 Brute force

An attacker could attempt to guess the seeds used in ThreeBears by brute force. This is infeasible because they are all at least 256 bits long, so

a classical attack would take 2^{256} effort, and a quantum attack would take $2^{256}/\text{maxdepth} > 2^{128}$ effort. He could mount a classical multi-target key-recovery attack, but this would take $2^{320}/n$ time, where the number of target keys n is likely much less than 2^{64}. He could also mount a classical multi-target attack in $2^{256}/n$ time on n ciphertexts encrypted with the same public key. We could prevent this last attack by setting `ivBytes` to 8 instead of 0, but we don't consider this attack a serious threat because it isn't remotely feasible, probably can't be improved, and probably won't run faster on a quantum computer.

5.2 Inverting the hash

If the adversary can find preimages for cSHAKE, then he could recover information about the private key from the matrix seed. However, this wouldn't actually yield the whole private key because the matrix seed is 24 bytes and the secret key is 40 bytes, so the adversary would need to find 2^{128} cSHAKE primages.

5.3 Lattice reduction

The main avenue of cryptanalytic attack against ThreeBears is to recover the private key using lattice reduction. We analyzed the feasibility of these attacks using the conservative methodology of NewHope [ADPS15] and Kyber [BDK+17]. The results for primal attacks are shown in Table 7.

Some instantiations of Ring-LWE over non-cyclotomic rings are much more vulnerable to dual attacks, because noise which is roughly spherical in the primal form ends up badly skewed in the dual form [Pei16]. Initial calculations by Arnab Roy and Hart Montgomery show that for golden Solinas rings, the map between the primal and dual lattices has singular values in the range $[0.513, 2.176]\sqrt{D}$. That is, it roughly halves the noise in some coefficients and doubles it in others. We didn't finish evaluating the security against dual attacks in time for the submission deadline. However, it will

	Classical		Quantum		
System	Lattice	Hybrid	Lattice	Hybrid	Class
BabyBear	156	192	141	181	II
BabyBear ephem	168	207	152	194	II
MamaBear	242	247	219	233	IV
MamaBear ephem	263	330	238	311	V
PapaBear	322	329	292	311	V
PapaBear ephem	355	450	322	426	V

Table 7: Log_2 difficulty estimates for primal hybrid attack.

be much less problematic than halving the noise in every coefficient, which would knock 10-15% off our security estimates.

5.4 Multi-target lattice attacks

We expect that lattice reduction attacks don't batch well, because with high probability all honestly-generated keys have different matrix seeds.

5.5 Hybrid attack

Because ThreeBears uses less noise than either NewHope or Kyber, we had the additional concern of a hybrid lattice-reduction / meet-in-the-middle attack [BGPW16]. We used a script by John Schanck [Sch] to evaluate the feasibility of this attack using the same difficulty metrics as for the direct attack. We see that this attack is ineffective, especially when $\sigma^2 > 1/2$.

5.6 Chosen ciphertext

If an adversary can cause a decryption failure, he may be able to learn something about the private key. In the CCA-secure version of the system, the Fujisaki-Okamoto transform [FO99] prevents the adversary from modifying

ciphertexts. Instead, he must choose a random seed, and hope that the ciphertext causes a failure. This happens with probability less than 2^{133} for all recommended instances of ThreeBears.

Not all ciphertexts have the same probability of causing a failure. Rather, the failure probability p_{failure} depends on the amount of noise in the ciphertext. Since that noise is random, some ciphertexts will have higher p_{failure} and some lower. Classically, an adversary can use this property to decrease the number of queries required, but not the work of formulating those queries, which is still more than 2^{133} per failure. In CCA-secure versions of ThreeBears, sampling the noise includes the public key, so this effort cannot be re-used across keys.

For the same reason, not all private keys have the same probability of causing a failure. Distinguishing failure-prone public keys should be as hard as breaking them, so the adversary can't use this to his advantage.

A quantum attacker could try to use Grover's algorithm to find ciphertexts with higher p_{failure}. One may show that this raises the expected failure probability per random-oracle query from mean(p_{failure}) to at most root-mean-square(p_{failure}). We took this into account in Table 2, which is why the estimated post-quantum CCA attack cost is less the reciprocal of the failure probability. We consider it unlikely that some quantum attack could generically cause failures much faster than root-mean-square(p_{failure}).

CPU	Arch flags	Keccak	asm
Intel Skylake	`-march=native -mno-adx`	`Haswell`	Yes
ARM Cortex-A8	`-march=native -mthumb`	`ARMv7A`	No
ARM Cortex-A53	`-mcpu=cortex-a53 -DVECLEN=1`	`generic64`	No

Table 8: Compilation settings. We added `-mno-adx` on Skylake because ADX breaks `valgrind`'s memory profiler; `-mthumb` on Cortex-A8 for space savings; and `-DVECLEN=1` on Cortex-A53 because its NEON unit is slow.

6 Performance Analysis

6.1 Time

ThreeBears key generation and encapsulation both require sampling a $d \times d$ random matrix and multiplying it by a vector. For our N, Karatsuba multiplication is appropriate [KO62], so these operations take approximately $O(d^2 \cdot (\log N)^{\log_2 3}/b^2)$ time on a CPU with a b-bit multiplier. This is comparable to an RSA encryption with small encryption exponent. Encapsulation and decapsulation require a d-long vector dot product, which is d times faster. Additionally, key generation and encapsulation require sampling $2 \cdot d$ and $2 \cdot d + 1$ noise elements, respectively.

We benchmarked our code on several different platforms: Intel Skylake in Table 9; for ARM Cortex-A53 in Table 10; and for ARM Cortex-A8 in Table 11. These are intended to represent computers, smartphones, and embedded devices respectively.

For each platform, we compiled each instance with

```
clang-5.0 -Os -fomit-frame-pointer -DNDEBUG
```

and the additional flags shown in Table 8. In all cases we used 2-level Karatsuba multiplication, and linked the optimized libraries from the Keccak Code Package [BDP+17]. The Skylake implementation uses a small amount of assembly in the multiplication routine; the other platforms use no assembly.

System	CPA-secure			CCA-secure		
BabyBear	41k	62k	28k	41k	60k	101k
MamaBear	84k	103k	34k	79k	97k	152k
PapaBear	125k	154k	40k	119k	145k	213k

Table 9: Runtime in cycles on an Intel NUC6i5SYH with Core i3-6100U "Skylake" 64-bit processor at 2.3GHz

System	CPA-secure			CCA-secure		
BabyBear	159k	221k	83k	159k	218k	364k
MamaBear	316k	397k	114k	313k	391k	597k
PapaBear	525k	627k	146k	522k	622k	889k

Table 10: Runtime in cycles on a Raspberry Pi 3 with ARM Cortex-A53 64-bit processor at 1.2GHz

System	CPA-secure			CCA-secure		
BabyBear	342k	495k	175k	341k	491k	802k
MamaBear	722k	933k	254k	716k	921k	1375k
PapaBear	1228k	1497k	323k	1217k	1482k	2075k

Table 11: Runtime in cycles on a BeagleBone Black with ARM Cortex-A8 32-bit processor at 1GHz

We believe that the Skylake and Cortex-A53 code is reasonably close to optimal, but maybe careful tuning of the multiplication algorithm could knock 25% off. For Cortex-A8 and other ARMv7 platforms, optimizing the multiplication algorithm with NEON should provide a large improvement.

In profiling runs, we found that the FEC added between 0.1% and 2% overhead. In fact, the more significant overhead from adding FEC is that enables larger noise, which can result in more iterations in the `noise` function.

System	Private key	Public key	Capsule
BabyBear	40	804	917
MamaBear	40	1194	1307
PapaBear	40	1584	1697

Table 12: ThreeBears object sizes in bytes

Component	Skylake	Cortex-A53	Cortex-A8
Arithmetic	2212	1920	1444
Melas FEC	645	573	449
cSHAKE	1426	832	856
Main system	3130	2795	2105
Total	7413	6120	4854

Table 13: Code size for MamaBear in bytes.

6.2 Space

Bandwidth and key storage Each field element is serialized into $312 \cdot 10/8 = 390$ bytes. Each instance uses $390 \cdot d + 24$ bytes in its public key; 40 bytes in its private key; and $390 \cdot d + (256 + 18)/2$ bytes in its capsules. This is shown in Table 12.

Code size We measured the total code size on each platform to implement MamaBear, using the same compilation flags that we used to measure performance. The code size does not include the Keccak Code Package, which is dynamically linked. The sizes are shown in Table 13.

Memory usage We measured the stack memory usage of each top-level function on Skylake using Valgrind's `lackey` tool. The memory usage includes everything in the function's stack frame, but not input or output. The measured usage probably depends on shared libraries and stack alignment. We measured three implementations:

1. The optimized implementation described above.

	CPA-secure			CCA-secure		
System	Keygen	Enc	Dec	Keygen	Enc	Dec
Speed-optimized						
BabyBear	6200	6616	4216	6200	6648	8184
MamaBear	9112	9528	4632	9112	9560	11512
PapaBear	12856	13272	5048	12856	13304	15672
No vectorization; 1-level Karatsuba						
BabyBear	2792	2840	2168	2792	2856	4744
MamaBear	3208	3256	2168	3208	3272	5576
PapaBear	3624	3672	2168	3624	3688	6408
No vectorization; 1-level Karatsuba; re-derive						
All instances	2392	2424	2168	2392	2424	3064

Table 14: ThreeBears memory usage bytes on Skylake, excluding input and output.

2. An implementation without vectorization and with only one level of Karatsuba. This takes about 1.5× as long per operation.

3. An implementation which re-derives the private key elements a_i, b_i instead of remembering them across loop iterations during the matrix multiply. This takes constant memory, but up to twice the time.

The lower-memory implementations also have slightly less code size. The results are shown in Table 14. We expect memory usage on a 32-bit platform to be slightly higher, because our implementation of field elements uses only 26/32 bits instead of 60/64 for easier carry propagation.

7 Advantages and limitations

We originally designed ThreeBears because we thought that variants of RLWE (in this case, I-MLWE) should be studied more before the community chooses a standard. Our analysis shows that it is quite competitive with its predecessors Kyber [BDK+17] and Hila5 [Saa17].

7.1 Advantages

Speed ThreeBears shares the advantages of most RLWE and MLWE systems. Key generation is very fast, so fast that it is typically better to store the seed instead of the expanded private key. Encryption and decryption are also very fast, typically significantly faster than elliptic curves.

Size Public keys and ciphertexts are reasonable sizes, about 1.2kB and 1.3kB respectively for MamaBear. This is small enough to be practical for most Internet-connected systems, but not as small as an isogeny-based system. Private keys are only 40 bytes. Code sizes are under 10kB plus Keccak, and stack requirements can be pushed near 3kB plus input and output.

Hardware support ThreeBears can be used with existing big-integer software and hardware, which is useful for smart cards and hardware security modules. This reduces hardware area in systems that must support both pre-quantum and post-quantum algorithms.

Simplicity ThreeBears has a relatively simple specification, especially since it doesn't use the number-theoretic transform. On most platforms, ThreeBears doesn't need vectorization to achieve respectable speed. These advantages mean that its code is small, simple and easy to audit. Forward error correction adds complexity, but it's only some 75 lines of C code and it's easy to test separately.

Efficiency To hedge our new security assumption, we have chosen larger instances than other RLWE systems. These instances are nonetheless efficient. MamaBear is conjecturally about as strong as Kyber's "paranoid" parameter set, while having roughly 15% smaller public keys and ciphertexts. PapaBear is conjecturally stronger than NewHope and Hila5, while again having about 15% smaller public keys and ciphertexts.

7.2 Limitations

Novelty ThreeBears doubles down on RLWE's main disadvantage: the integer MLWE problem has not been studied as extensively as either plain LWE or polynomial RLWE. Gu showed that integer and polynomial RLWE are asymptotically comparable [Chu17], but we aren't aware of either positive or negative results for practical parameter sizes.

Noise ThreeBears' efficiency comes in part from a large dimension and low noise. This might put it at risk from new hybrid attacks, even though existing ones are not a threat.

Speed None of the NIST candidates have final code yet, so we cannot test performance conclusively. ThreeBears' big-integer arithmetic ought to make it slower than an NTT-based system, especially on tiny machines and on machines with vector units. Big-integer arithmetic may also complicate hardware implementations, at least in hardware that doesn't support pre-quantum algorithms. Also, some cryptographers were probably hoping never to see a carry chain again.

Flexibility ThreeBears can only be used for key encapsulation and encryption. So far there is no I-MLWE signature scheme.

7.3 Suitability for constrained environments

ThreeBears is suitable for smart card implementation, and implementors can reuse their RSA big-number engines, but it may be inefficient on 8-bit processors without multiprecision arithmetic support. It requires more RAM and bandwidth than most classical algorithms, but we expect it to be competitive with other post-quantum algorithms. Its high speed and small private keys should also help in constrained environments.

8 Absence of backdoors

I, the designer of ThreeBears, faithfully declare that I have not deliberately inserted any hidden weaknesses in the algorithms.

9 Acknowledgments

Thanks to Arnab Roy and Hart Montgomery for their analysis of the ring shape.

Thanks to Dominique Unruh, Eike Kiltz, Fernando Virdia and Amit Deo for discussions of the quantum analysis of the CCA transform.

Thanks to Mark Marson for many helpful discussions.

Thanks to Rambus for supporting this work.

References

[ADPS15] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - a new hope. Cryptology ePrint Archive, Report 2015/1092, 2015. http://eprint.iacr.org/2015/1092.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. NewHope without reconciliation. Cryptology ePrint Archive, Report 2016/1157, 2016. http://eprint.iacr.org/2016/1157.

[BDK+17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. CRYSTALS – kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. http://eprint.iacr.org/2017/634.

[BDP+17] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Vladimir Sedach. Keccak code package, 2017. https://github.com/gvanas/KeccakCodePackage.

[BGPW16] Johannes A. Buchmann, Florian Göpfert, Rachel Player, and Thomas Wunderer. On the hardness of LWE with binary error: Revisiting the hybrid lattice-reduction and meet-in-the-middle attack. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16*, volume 9646 of *LNCS*, pages 24–43. Springer, Heidelberg, April 2016. doi:10.1007/978-3-319-31517-1_2.

[Chu17] Gu Chunsheng. Integer version of ring-LWE and its applications. Cryptology ePrint Archive, Report 2017/641, 2017. http://eprint.iacr.org/2017/641.

[DXL12] Jintai Ding, Xiang Xie, and Xiaodong Lin. A simple provably secure key exchange scheme based on the learning with errors problem. Cryptology ePrint Archive, Report 2012/688, 2012. http://eprint.iacr.org/2012/688.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, August 1999.

[Ham15] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. http://eprint.iacr.org/2015/625.

[HGNP+03] Nick Howgrave-Graham, Phong Q. Nguyen, David Pointcheval, John Proos, Joseph H. Silverman, Ari Singer, and William Whyte. The impact of decryption failures on the security of NTRU encryption. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 226–246. Springer, Heidelberg, August 2003.

[JZC+17] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. Post-quantum ind-cca-secure kem without additional hash. Cryptology ePrint Archive, Report 2017/1096, 2017. https://eprint.iacr.org/2017/1096.

[KjCP16] John Kelsey, Shu jen Chang, and Ray Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash, 2016. https://doi.org/10.6028/NIST.SP.800-185.

[KO62] A Karabutsa and Yu Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akademii Nauk SSSR*, 145(2):293, 1962.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert,

editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May 2010.

[LW87] Gilles Lachaud and Jacques Wolfmann. Sommes de kloosterman, courbes elliptiques et codes cycliques en caractéristique 2. *CR Acad. Sci. Paris Sér. I Math*, 305(20):881–883, 1987.

[MP13] Daniele Micciancio and Chris Peikert. Hardness of SIS and LWE with small parameters. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 21–39. Springer, Heidelberg, August 2013. doi:10.1007/978-3-642-40041-4_2.

[Pei16] Chris Peikert. How (not) to instantiate ring-LWE. Cryptology ePrint Archive, Report 2016/351, 2016. http://eprint.iacr.org/2016/351.

[Saa16] Markku-Juhani O. Saarinen. Ring-LWE ciphertext compression and error correction: Tools for lightweight post-quantum cryptography. Cryptology ePrint Archive, Report 2016/1058, 2016. http://eprint.iacr.org/2016/1058.

[Saa17] Markku-Juhani O. Saarinen. On reliability, reconciliation, and error correction in ring-LWE encryption. Cryptology ePrint Archive, Report 2017/424, 2017. http://eprint.iacr.org/2017/424.

[Sch] John Schanck. LWE hybrid attack scripts. Personal communication.

[SXY17] Tsunekazu Saito, Keita Xagawa, and Takashi Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. Cryptology ePrint Archive, Report 2017/1005, 2017. http://eprint.iacr.org/2017/1005.

[TU16] Ehsan Ebrahimi Targhi and Dominique Unruh. Post-quantum security of the fujisaki-okamoto and OAEP transforms. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, vol-

ume 9986 of *LNCS*, pages 192–216. Springer, Heidelberg, October / November 2016. doi:10.1007/978-3-662-53644-5_8.

A Intellectual property statements

This appendix is a LaTeX rendering of the intellectual property statements we mailed to NIST.

A.1 Statement by Each Submitter

I, Michael Hamburg, of 425 Market St 11th Floor, San Francisco CA 94105, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as ThreeBears, is my own original work, or if submitted jointly with others, is the original work of the joint submitters.

I further declare that:

- *I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as ThreeBears).*

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive financial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem?s specifications (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3 in the Call For Proposals for any patent or patent application identified to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3 of the Call For Proposals, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed: [in the mailed version]
Title: Senior Principal Engineer
Date: Sept 22, 2017
Place: San Francisco, CA

A.2 Statement by Reference/Optimized Implementations? Owner

I, Martin Scott, 425 Market St 11th Floor, San Francisco CA 94105, am the owner or authorized representative of the owner (Rambus Inc.) of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed: [in the mailed version]
Title: Senior Vice President / General Manager
Date: Sept 22, 2017
Place: San Francisco, CA