Post-quantum cryptography proposal:
ThreeBears
(draft)

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

September 30, 2017

Contents

1 Introduction

This is a draft submission of the ThreeBears post-quantum key encapsulation mechanism.

2 Specification

2.1 Notation

Integers Let $\mathbb{Z}$ be the integers and $\mathbb{N}$ the non-negative integers. Let $\mathbb{Z}/N\mathbb{Z}$ denote 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 $[0, n)$.

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.

Sequences Let T^n denote the set of sequences of length n, whose elements have type T. Let T^* denote the set of sequences of any length, whose elements have type T.

If S is a sequence of n elements, let S_i be its ith element (where $0 \leq i < n$). In the reverse direction, we use the notation Let $[\![a, b, \ldots, z]\!]$ for a sequence. We use $[\![S_i]\!]_{i=0}^{n-1}$ for the sequence of n elements whose ith element is S_i.

Let $a \oplus b$ be the bitwise xoring of integers or bit-sequences a and b. When xoring bit-sequences of different lengths, we extend the shorter sequence to the length of the longer one. We use the notation $\bigoplus S$ for the $\oplus$-sum of a sequence S.

2.2 Encoding

Let $\mathcal{B}$ denote the set of bytes, i.e. [0..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 ThreeBears parameter set, so we never strip zeros or use length padding.

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

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

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 capsules contain a sequence of 4-bit *nibbles*, i.e. elements of $[0, 16)$. We encode this sequence by packing two nibbles into a byte in little-endian order. That is, a sequence $[\![s]\!]$ encodes as

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

These sequences always have even length, but if they didn't then the last nibble would be encoded in its own byte. The same rules apply for converting between bit sequences and byte sequences. We will mention explicitly what part of the capsule is encoded as nibbles.

Any other tuple, vector or sequence of items is encoded as the concatenation of the encodings of those items.

2.3 Parameters

An instance of ThreeBears has many parameters. Many of these are lengths of various seeds, which we fix here but might tweak according to future requirements. The list is shown in Table 1. All the parameters are in scope in every function in this specification.

The parameters for the recommended instances are shown in Table 2. Our primary recommendation is MamaBear.

The recommended parameters for ThreeBears are shown in Table 2. Each system has variants for CPA-secure and CCA-secure key exchange. The primary recommendation is MamaBear.

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 15-byte 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, so it is encoded as $128 \cdot \sigma^2$.

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

$$[\![1, 40, 24, 32, 0, 0, 32, 10, 56, 1, 3, 64, 4, 18, 1]\!]$$

Since there are multiple uses of the hash function within ThreeBears, we also separate them with a 1-byte "purpose" p. We define the hash function

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

Here L is the length in bytes of the desired output. The `cSHAKE256` hash function is defined in [6]. We use only one personalization string to avoid

Description	Name	Value
Independent parameters:		
Specification version	`version`	1
Key generation seed bytes	`keygenSeedBytes`	40
Matrix seed bytes	`matrixSeedBytes`	24
Encryption seed bytes	`encSeedBytes`	32
Initialization vector bytes	`ivBytes`	0
Targhi-Unruh tag bytes	`targhiUnruhBytes`	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`	varies: 0 or 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

System	d	σ^2 (`cca`=0)	σ^2 (`cca`=1)	`fecBits`
BabyBear	2	5/8	3/8	0
BabyBear+	2	1	5/8	18
MamaBear	3	1/2	9/32	0
MamaBear+	3	7/8	1/2	18
PapaBear	4	7/16	1/4	0
PapaBear+	4	3/4	7/16	18

Table 2: ThreeBears recommended parameters.

polluting the `cSHAKE` namespace and to enable precomputation of the first hash block.

2.4.2 Sampling

Uniform We will need to expand a short seed into a uniform sample mod N. We will use this to sample a $d \times d$ matrix, so it takes parameters i, j in $[0 \,..\, d-1]$. This is shown in Algorithm 1.

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. Obviously, with only one byte per digit we can only sample distributions with certain variances, as described in that algorithm's requirements. Since $\sigma^2 \leq 1$ for all ThreeBears instances, the inner loop runs at most twice.

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. Unfortunately, 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.

2.4.4 Forward error correction

It is possible to use forward error correction (FEC) with ThreeBears. 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

Function `uniform`(seed, i, j) **is**

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

output : Uniformly pseudorandom number modulo N

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

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

end

Function `noise`(seed, p, 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

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

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

Algorithm 1: Uniform and noise samplers

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

Algorithm 2: Extracting the digit with the ith-least noise

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 2 errors.

Encoding Let $\texttt{seq}_b(n)$ be the b-bit sequence of the bits in 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 follows:

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

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

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

Algorithm 3: Melas FEC encode

This is a field operation with addition being $\oplus$. The multiplicative identity is $\texttt{seq}_9(\texttt{0x100})$. Define $x^{\odot n}$ as the nth power under $\odot$-multiplication.

To decode an n-bit sequence B, carry out the following steps:

Implementation This high-level spec admits many optimizations. See the included `fec.inc.c` for a fast, short, constant-time implementation of the Melas FEC.

2.5 Keypair generation

We explicitly define a deterministic key expander `KeypairDet`. That way, implementations which need to store a private key long-term — i.e. for offline decryption rather than key exchange — can just store the seed. To generate a keypair with a given parameter set, we simply call `KeypairDet` with a uniformly random seed. This procedure is shown in Algorithm 5.

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 quadratic equation from syndrome
// Pad with 6 zero bits because real impls are byte-aligned
```

$s \leftarrow \texttt{syndrome}_{18}(B \mid\mid [\![0,0,0,0,0,0]\!])$;

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

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

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

```
// 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** $n - 1$ **do**

  if $\texttt{step}^i(Q, e_0) = [\![1, 0, \ldots, 0]\!]$ **or** $\texttt{step}^i(Q, e_1) = [\![1, 0, \ldots, 0]\!]$ **then**

    $B_i \leftarrow B_i \oplus 1$;

  end

end

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

end

Algorithm 4: Melas FEC decode

Function `KeypairDet`(seed) **is**

input : Uniformly random seed of length `keypairSeedBytes`

output: Private key sk; public key pk

`// Generate the private key vector`

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

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

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

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**

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

end

`// Output`

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

if `CCA` **then** sk $\leftarrow \left([\![a_j]\!]_{j=0}^{d-1},\ \text{pk}\right)$;

else sk $\leftarrow [\![a_j]\!]_{j=0}^{d-1}$;

return (sk, pk);

end

Function `Keypair`() **is**

return `KeypairDet`(`RandomBytes`(`keypairSeedBytes`));

end

.

Algorithm 5: Keypair generation

Function `EncapsulateDet`(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

`// Extend the seed and generate ephemeral private key`

if `cca` **then** ext_seed $\leftarrow$ pk||seed||iv;

else ext_seed $\leftarrow$ seed||iv;

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

`// Multiply by transpose of random matrix; add noise`

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

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

`// Encrypt seed using approximate shared secret`

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

encoded_seed $\leftarrow$ `FecEncode`(seed as a sequence of bits);

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

  $\text{encr}_i \leftarrow \texttt{extract}_4(C, i) \oplus 8 \cdot \text{encoded\_seed}_i$

end

`// Output`

shared_secret $\leftarrow H_2(\text{ext\_seed}, \texttt{sharedSecretBytes})$;

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

.

Algorithm 6: Deterministic encapsulation subroutine

2.5.1 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 `EncapsulateDet`.

Function `Encapsulate`(pk) **is**
    input : Public key pk
    output: Shared secret; capsule
    seed ←`RandomBytes`(`encSeedBytes`);
    iv ←`RandomBytes`(`ivBytes`);
    return `EncapsulateDet`(pk, iv, seed);
end

.

Algorithm 7: Encapsulation

2.6 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 8.

Function `Decapsulate`(sk, capsule) **is**

input : Private key sk, capsule

output: Shared secret or $\perp$

`// Unpack secret key and capsule`

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

if `cca` **then** $\left([\![a_j]\!]_{j=0}^{d-1},\ \text{pk}\right) \leftarrow$ sk;

else $[\![a_j]\!]_{j=0}^{d-1} \leftarrow$ sk;

`// Decrypt using approximate shared secret`

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

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

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

end

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

if `CCA` **then**

  `// Re-encapsulate to check that capsule was honest`

  (shared_secret, capsule$'$) $\leftarrow$ `EncapsulateDet`(pk, iv, seed);

  if capsule$'$ $\neq$ capsule **then return** $\perp$;

  else return shared_secret;

else

  shared_secret $\leftarrow H_2$(seed||iv, `sharedSecretBytes`);

  return shared_secret

end

end

Algorithm 8: Decapsulation

3 Design Rationale

3.1 Overall design

We borrowed the overall design from KYBER [3]. We liked the way that module-LWE allows parameters to be changed without changing too much code.

We also thought the potential improvements from Saarinen's error correction [8, 7] were too good not to investigate. In the context of THREEBEARS, they give a significant improvement. In earlier versions, the security increase was easily enough to justify the complexity. But after re-tuning to improve the baseline instances' security, we decided that the main recommendation should be the simpler version with no FEC.

3.2 Parameter choices

Seeds The seed sizes in THREEBEARS are designed for an overall 2^{256} 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 to recover one of T keys by brute force. But since protecting key generation is almost free, we set the `keygenSeed` to 320 bits (40 bytes). We could have also added an initialization vector to each ciphertext to prevent multi-target attacks when someone encrypts many ciphertexts to a single key. Since this isn't as cheap, we didn't do it.

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 be safe to use a 128-bit matrix seed, since at worst a k-way collision would enable an attack on k public keys at once, and collisions with adversarially-chosen seeds aren't a problem.

Modulus We chose N to be prime to prevent attacks based on subrings. We would have liked for N to be a Fermat prime, but there are no Fer-

mat primes of the right size. As far as we know, the second-best shape is a "golden-ratio Solinas" prime; when used with a clarifier, this results in less noise amplification than a Mersenne prime, because reduction modulo a Mersenne prime at least doubles the digits being reduced, which multiplies variance by 4. By contrast, a golden-ratio Solinas prime multiplies variance by 3/2 in the center digits. 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 modulus is slightly greater than a power of 2, which causes annoying corner cases.
- The larger modulus better matches the NIST target security levels.
- The larger modulus allows us to use FEC.
- The larger modulus has more efficient limb sizes on 64-bit platforms.

The smaller modulus would have enabled finer granularity in security level, but we thought the other considerations more important.

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 this leads to a much easier implementation.

Variance We chose the noise variance as a simple dyadic fraction. We 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, for BabyBear, we decided that between a 128-bit passive

lattice attack and a 128-bit chosen-ciphertext attack (requiring perhaps 2^{100} messages), the former is both a much greater threat and much more likely to improve over time. Thus we tweaked the parameters to boost lattice security slightly at the expense of CCA security.

No Targhi-Unruh tag We chose not to use a Targhi-Unruh tag, because we believe that it adds no security and is merely an artifact of the proof. In addition, for ThreeBears it should be possible to patch the Targhi-Unruh proof. In the proof, the tag is used by the random oracle to "leak" the seed to the simulator. But because the random oracle affects the low bit of each digit of the ciphertext, this is possible anyway.

Melas code 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. However, decoding these codes in constant time is rather complicated. So we scaled back our ambitions to a 2-error-correcting code, because these are relatively easy to decode with a half-trace computation. The Melas BCH code seemed simplest to us, but we aren't experts in error correcting codes, and there may be a better option.

System	CPA-secure			CCA-secure		
BabyBear+	43k	61k	17k	43k	69k	86k
MamaBear+	87k	103k	22k	83k	109k	131k
PapaBear+	129k	153k	26k	124k	162k	189k

Table 3: Runtime in cycles on an Intel NUC6i5SYH with Core i3-6100U "Skylake" 64-bit processor at 2.3GHz

System	CPA-secure			CCA-secure		
BabyBear+	164k	220k	59k	165k	233k	295k
MamaBear+	325k	400k	80k	320k	414k	499k
PapaBear+	539k	634k	101k	532k	651k	758k

Table 4: Runtime in cycles on a Raspberry Pi 3 with ARM Cortex-A53 64-bit processor at 1.2GHz

4 Performance Analysis

4.1 Time

The time required by ThreeBears is approximately $O(d^2)$, because key generation and encapsulation both require sampling a $d \times d$ random matrix and multiplying it by a vector. Encapsulation and decapsulation require a d-long vector dot product. Additionally, key generation and encapsulation require sampling $2 \cdot d$ and $2 \cdot d + 1$ noise elements.

We benchmarked our code on several different platforms: Intel Skylake in Table 3; for ARM Cortex-A53 in Table 4; and for ARM Cortex-A8 in Table 5. These are intended to represent computers, smartphones, and embedded devices respectively.

For each platform, we compiled the code for all the bears with

```
clang-3.9 -Os -fomit-frame-pointer -DNDEBUG -march=native
```

or `-mcpu=[cpu]` where `clang` doesn't support `-march=native`. In all cases we used 2-level Karatsuba multiplication, and linked the optimized libraries

System	CPA-secure			CCA-secure		
BabyBear+	369k	518k	152k	370k	543k	694k
MamaBear+	770k	974k	211k	770k	1010k	1227k
PapaBear+	1318k	1586k	279k	1310k	1614k	1899k

Table 5: Runtime in cycles on a BeagleBone Black with ARM Cortex-A8 32-bit processor at 1GHz

from the Keccak Code Package [2]. On all platforms except the A53, these libraries are vectorized (the A53 uses `generic64`). The Skylake version uses a small amount of assembly in the inner loop of the multiplication routine; the other platforms do not.

For brevity, we only show the "+" instances. These are slightly slower than the non-"+" instances: the cost of FEC is and (more importantly!) the increased effort to sample with $\sigma^2 > 1/2$.

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 might provide a large improvement.

4.2 Space

Bandwidth and key storage Each field element is serialized into $312 \cdot 10/8 = 390$ bytes. Thus, each instances uses $390 \cdot d + 32$ bytes in its public key; $390 \cdot d$ bytes in its private key (plus the public key for CCA-secure instances); and $390 \cdot d + 32 \cdot 4$ bytes in its capsules (plus 9 bytes of encrypted FEC for the "+" instances). This is shown in Table 6.

Private keys could be compressed, since each digit is required to be small. For example, using 2 bits per digit would shrink a MamaBear private key from 1170 bytes to 234 bytes. We have not done this, because applications looking to save space can simply re-expand their private key from the 40-byte seed.

System	Private (CPA)	Private (CCA)	Public	Capsule
BabyBear	708	1600	804	908
BabyBear+	708	1600	804	917
MamaBear	1170	2372	1194	1298
MamaBear+	1170	2372	1194	1307
PapaBear	1560	3152	1584	1688
PapaBear+	1560	3152	1584	1697

Table 6: ThreeBears object sizes in bytes

Code size We measured the total code size on each platform to implement all proposed instances of ThreeBears, 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. On Skylake, Cortex-A53 and Cortex-A8, the code size was 10898 bytes, 6777 bytes and 7965 bytes[1], respectively.

Memory usage We have not yet measured dynamic stack usage (for some reason this is hard?). The vectorized version of `Encapsulate` materializes the most field elements. It uses $(d+1)^2$ elements of $\mathbb{Z}/N\mathbb{Z}$: d^2 for the matrix and $2 \cdot d + 1$ elements for noise. The non-vectorized version materializes $2 \cdot d + 2$ elements: d noise vectors, one element of the matrix, and one accumulator. This could be reduced to 3 elements total if memory were scarce, by resampling the noise elements each time. If 2-level Karatsuba is used then its intermediates use as much space as another ring element. Each element is represented in reduced-radix form using 416 bytes on a 64-bit machine (radix 2^{60}), or 480 bytes (radix 2^{26}) on a 32-bit machine. On a 32-bit machine, this totals 4800, 8160 and 12480 bytes for vectorized BabyBear, MamaBear and PapaBear, respectively. Typical implementations will use a kilobyte or two of additional stack space for accumulators, temporaries, and cSHAKE contexts. CCA-secure decapsulation must also call the encapsulation routine to recompute the capsule.

[1] It would be 6095 bytes with Thumb2, but the KCP isn't compatible with Thumb2.

5 Analysis of known attacks

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 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 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 Multi-target attacks

Lattice-reduction attacks do not parallelize well, because (with high probability) all honestly-generated keys have a different matrix seed.

5.4 Lattice reduction

The main avenue of cryptanalytic attack against ThreeBears is to recover the private key using lattice reduction. [[**TODO: Expand this sec-**

	CCA, no FEC		Ephem or FEC		Ephem and FEC	
System	Lattice	Hybrid	Lattice	Hybrid	Lattice	Hybrid
BabyBear	130	135	141	181	152	194
MamaBear	202	207	219	233	238	311
PapaBear	276	280	298	321	322	426

Table 7: Log_2 difficulty estimates for hybrid attack. Ephemeral modes with FEC have the same security as CCA-secure modes with no FEC.

tion]]

We analyzed the feasibility of these attacks using the conservative methodology of NewHope [1] and Kyber [3].

5.5 Hybrid attack

Because ThreeBears uses smaller noise than either NewHope or Kyber, we have the additional concern of a hybrid lattice-reduction / meet-in-the-middle attack [4]. We used a script by John Schanck [9] to evaluate the feasibility of this attack. In all cases, we estimate that the hybrid attack will be less effective than a direct lattice reduction attack.

The conservative methodology of NewHope and Kyber makes the hybrid attack look less appealing, since it intentionally underestimates the cost of lattice reduction. A more realistic estimate would put the cost of lattice reduction much higher, so that the saving some work with a hybrid attack is more helpful.

The ephemeral and error-corrected instances have more noise, so they are less vulnerable to a hybrid attack than the CCA-secure ones. The estimates are shown in Table 7. Once the noise passes 1/2, the hybrid attack becomes almost entirely ineffective.

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 [5] 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 $> 2^{133}$ per failure. In CCA-secure versions of ThreeBears, sampling the noise includes the public key, so this effort cannot be re-used across keys.

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 RMS(p_{failure}). This is why the estimated CCA attack cost is less the reciprocal of the failure probability. We consider it unlikely, but not out of the question, that some attack could do this much better than Grover.

5.7 Summary

We show the overall estimated security levels of all recommended instances of ThreeBears in Table 8. Note that quantum brute-force attacks against these systems would take less effort than lattice attacks, but lattice attacks are much more likely to improve.

	CPA-secure version			CCA-secure version			
System	Failure	Lattice	Class	Failure	CCA	Lattice	Class
BabyBear	2^{-54}	141	II	2^{-133}	120	130	II
BabyBear+	2^{-58}	152	II	2^{-148}	122	141	II
MamaBear	2^{-57}	218	IV	2^{-153}	138	201	IV
MamaBear+	2^{-51}	237	V	2^{-147}	137	218	IV
PapaBear	2^{-56}	297	V	2^{-148}	134	275	V
PapaBear+	2^{-52}	320	V	2^{-144}	135	297	V

Table 8: ThreeBears estimated post-quantum security levels against lattice and CCA attacks.

6 Potential future improvements

There are a few areas where ThreeBears has clear potential for future improvements.

Better error correction Using a code which corrects more errors would reduce the failure probability of ThreeBears, leading to higher security for the same size ciphertexts. So would a "soft" error correcting code, i.e. one which can use information from the rounding step.

More precise analysis It may be that a more precise security or failure analysis would lead to a different set of optimal parameters.

Faster sampling We could expand the matrix or private key with something faster than SHAKE, such as ChaCha20 or AES (on platforms that accelerate AES).

Less entropy in noise sampling We could change the noise sampler to use exactly the required number of bits per sample, or in particular to

sample 2-4 coefficients per noise byte when the variance permits it. This would reduce the overhead from cSHAKE.

Private key compression We could compress private keys by taking advantage of their sparse structure. We didn't do this because in most cases where the key size is a problem, it's better to re-expand from the 40-byte seed.

Noise shaping Since reduction mod N slightly distorts the noise, it may be possible to achieve better security or lower failure probability by choosing different noise in each digit, or by choosing non-independent noise for digits which are $D/2$ apart.

References

[1] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - a new hope. In *USENIX Security 2016*, 2016. `http://eprint.iacr.org/2015/1092`.

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

[3] 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`.

[4] Johannes 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 *Africacrypt 2016*, 2016. `http://eprint.iacr.org/2016/089`.

[5] Eiichiro Fujisaki and Tatsuaki Okamoto. *Secure Integration of Asymmetric and Symmetric Encryption Schemes*, pages 537–554. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[6] 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`.

[7] 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`.

[8] Markku-Juhani Olavi Saarinen. Ring-LWE ciphertext compression and error correction: Tools for lightweight post-quantum cryptography. In *Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security*, IoTPTS '17, pages 15–22, New York, NY, USA, 2017. ACM.

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