Post-quantum cryptography proposal: ThreeBears

Inventor, developer and submitter
Mike Hamburg
Rambus Cryptography Products Group
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

July 25, 2019

Contents

0 Changelog

This second-round submission differs from the first-round submission in several ways:

- We have added second-round tweaks.
- We corrected typos, and improved the exposition of the scheme.
- We added benchmarks on Cortex-M4. We updated the rest of the benchmarks as well, with negligible changes.
- We updated the proof sketch to a formal proof of security, to be found in the file `proof.pdf`.
- We added a study of decryption failure rates for systems with error-correcting codes, to be found in the file `failure.pdf`.

After the second round, we have made additional changes:

- We have made implicit rejection mandatory.
- We have increased the PRF key for implicit rejection to 40 bytes, the same size as the private seed. This gives the same multi-target security for the PRF as for the main system. Since this only affects rejected ciphertexts, it doesn't meaningfully affect compatibility. So we chose not to increase the version number this change.
- We have updated the benchmarks, which are roughly 10% slower for CCA decapsulation due to implicit rejection (except on Cortex-M4, where they are offset by improvements in the PQM4 project).
- Due to implicit rejection, the proof of security is mostly obsoleted by [BHHP19], except for the multi-key cases.
- We have added a toy system, GummyBear, which is extremely weak and intended to be broken in practice. After initially posting this with $D = 80$, which is too weak to be interesting, we changed it to $D = 120$.

- We have added two toy systems, KOALA and KOALAEPHEM, which approximate what THREEBEARS might look like in a lightweight context. We also fixed a typo: these systems have 645-byte capsules, not 657 bytes.

0.1 Second-round tweaks

Lower failure probability We slightly reduced the noise (and lattice security) in CCA instances, to greatly reduce the probability of decryption failure. We are unaware of any attacks on the 2^{-140}-ish failure probability in the original submission, but we would like to rule out attacks more conclusively. This changed the security estimates. We also changed these estimates due to a revised version of Schanck's security estimator [Sch]. For the CCA-secure schemes, we also changed the method used to estimate failure probability, so that it is more conservative. This reduces the risk of a design mistake. We detailed the failure analysis in the `failure.pdf` document in this submission package.

We have also reduced the noise and failure probability for the alternative parameter sets, but we didn't estimate their security using the newer failure estimator. This is because the alternative parameter sets use different error-correction schemes (generally either no error correction or parity), so the modifications made in the estimator don't apply.

Implicit rejection The second-round system used an optional implicit rejection mode. Revision 2 changes to full implicit rejection.

0.2 Typo corrections

We corrected several typos for the second-round version.

- In the first-round submission, Table 2 specified PAPABEAR with $d = 3$. It should use $d = 4$. Thanks to Fernando Virdia for pointing this out.

- In Algorithm 1 (`noise`), "for $j = 0$ to d" has been corrected to "for $j = 0$ to $D - 1$".
- In Algorithm 6 (`Encapsulate`), the first-round spec had some `noise` calls operating on seed, and some on matrixSeed||seed||iv. This has been corrected to always use the latter.
- In Algorithm 7 (`Decapsulate`), the first-round spec had an extra term $\texttt{noise}_2$(seed) copy-pasted from `Encapsulate`. This has been removed.
- In the first-round version, we used $\texttt{extract}_4$ resp. $\texttt{extract}_5$ instead of the correct $\texttt{extract}_\ell$ resp. $\texttt{extract}_{\ell+1}$. This was still correct for the recommended parameters, which all have $\ell = 4$.

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 being based on the 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' 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 originally used explicit rejection; and why we don't use a plaintext-confirmation hash.

1.1 Module Learning With Errors

At a high level, ThreeBears is based on [LPR10], and more directly on Kyber [BDK^{+}17], with other improvements found in [DXL12, ADPS16]. The message flow and overall design my be regarded as a variant of ElGamal encryption [ElG84], shown in Figure 1, which computes a shared secret abG, where a is Alice's secret, b is Bob's secret, and G is a generator of some finite cyclic group. Unfortunately, ElGamal encryption using scalar multiplication on a cyclic group will not resist quantum attack, because it can be broken by Shor's algorithm.

A tempting alternative is to use some other commutative or associative operation, such as "ElGamal with matrices", in which the shared secret is $b^\top \cdot M \cdot a$. This is shown in Figure 2, and can be instantiated with any ring R and dimension d. Matrix ElGamal is insecure even against classical attack, because it is easy to compute a (up to coset) from $M \cdot a$. However, if a small amount of noise e_a is added, then depending on R and d, it may be much more difficult to recover a, or even distinguish $M \cdot a + e$ from a random vector. This is called the "Module Learning with Errors" (MLWE) problem, which is defined as follows:

Definition 1 (MLWE)**.** *Let R be a finite ring. Let χ be a probability distribution over R. Let d_1 and d_2 be positive integers. The (R, χ, d_1, d_2)-MLWE problem is to distinguish the MLWE distribution*

$$\mathcal{D}_1 := \{(M, Ma + e) : M \stackrel{R}{\leftarrow} R^{d_1 \times d_2}, a \leftarrow \chi^{d_1}, e \leftarrow \chi^{d_2}\}$$

from the uniform distribution

$$\mathcal{D}_0 := \{(M, r) : M \stackrel{R}{\leftarrow} R^{d_1 \times d_2}, r \stackrel{R}{\leftarrow} R^{d_2}\}$$

Adding noise turns the insecure Matrix ElGamal scheme into the encryption scheme shown in Figure 3; this is roughly [LPR10] plus the generalization from rings ($d = 1$) to modules ($d \geq 1$) from [ADPS16]. Because of the noise, Alice and Bob don't quite agree on the shared secret $b^\top Ma$; rather Alice computes $b^\top Ma + e_b^\top a$ whereas Bob computes $b^\top Ma + b^\top e_a + e'$. If (a, b, e_a, e_b, e') are small enough, this difference can be reconciled: Bob encodes his message so that different messages encode to values that are far apart, and Alice decodes by rounding, in some manner that encode$(m) + b^\top e_a + e' - e_b^\top a$ probably rounds to m.

The noisy matrix ElGamal scheme is easily seen to be secure against passive attack if the $(R, \chi, d, d + 1)$-MLWE problem is difficult. The public key is simply a (R, χ, d, d)-MLWE sample, so the adversary will not notice if it is replaced by a uniformly random (M, A). Once this has been done, $(b^\top M + e_b, b^\top A + e')$ is the transpose of a $(R, \chi, d, d+1)$-MLWE sample, so again the adversary will not notice if it is replaced by a uniformly random value. At that point, the message is blinded by a uniformly random ring element, so it gives no information about the message. See `proof.pdf` for a more formal analysis.

1.2 Polynomial and Integer MLWE

It remains to choose R, χ, d and the encoding and rounding algorithms. Most systems use Polynomial Ring or Module LWE, meaning that R is chosen as the polynomial ring $(\mathbb{Z}/q)[x]/\phi(x)$, where q is an integer on the order of 2^8 up to 2^{16}, and $\phi(x)$ is a sparse polynomial — typically either a prime cyclotomic or a power-of-2 cyclotomic. Let D be the degree of ϕ. The distribution χ chooses each of the D coefficients of its output independently from some distribution χ_1 over $\mathbb{Z}/q$. That is:

$$\chi := \left\{ \sum_{i=0}^{D-1} c_i \cdot x^i : c_i \overset{\text{iid}}{\leftarrow} \chi_1 \right\}$$

The distribution χ_1 is usually either a discretized Gaussian distribution, a binomial distribution or a fixed distribution on $\{-1, 0, 1\}$. A message $m = [\![m_i]\!]_{i=0}^{k-1}$ is encoded as

$$\text{encode}(m) := \lfloor q/2 \rfloor \cdot \sum_{i=0}^{k-1} m_i \cdot x^i$$

To decode an encoded message Z, write $Z := \sum_{i=0}^{D-1} z_i \cdot c^i$, and set $m_i = 1$ if $z_i \in [q/4, 3q/4]$ and $m_i = 0$ otherwise.

Instead of Polynomial MLWE, ThreeBears uses Integer MLWE (I-MLWE), as defined by Gu [Chu17], who proved that I-RLWE and P-RLWE have asymptotically similar security. I-MLWE is the case of MLWE, where instead of reducing each coefficient of the polynomial mod q, instead we reduce it by setting $x = q$. This makes the ring isomorphic to $\mathbb{Z}/N$, where $N = \phi(q)$

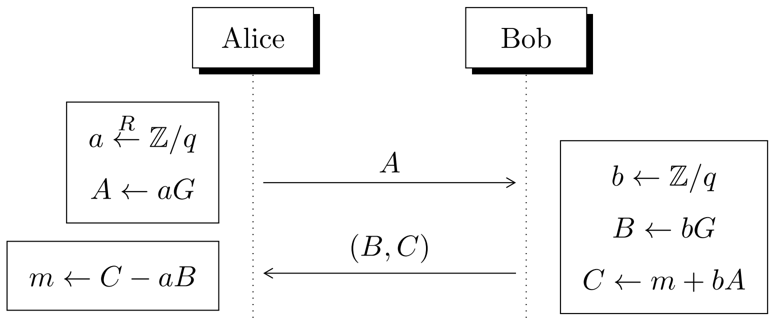

Figure 1: Pre-quantum ElGamal encryption over a group generated by G

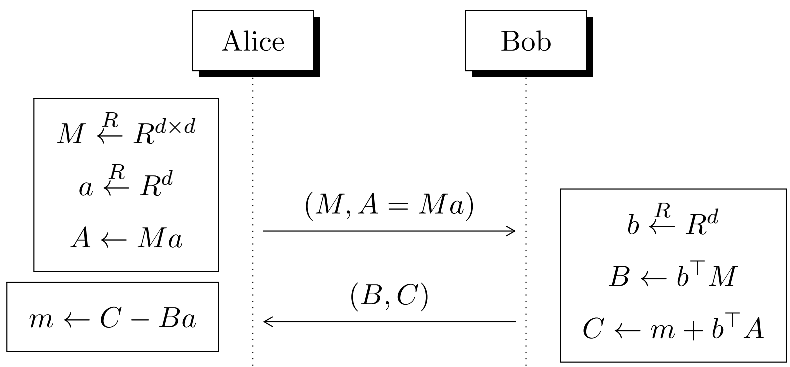

Figure 2: Insecure ElGamal encryption with matrices over a ring R.

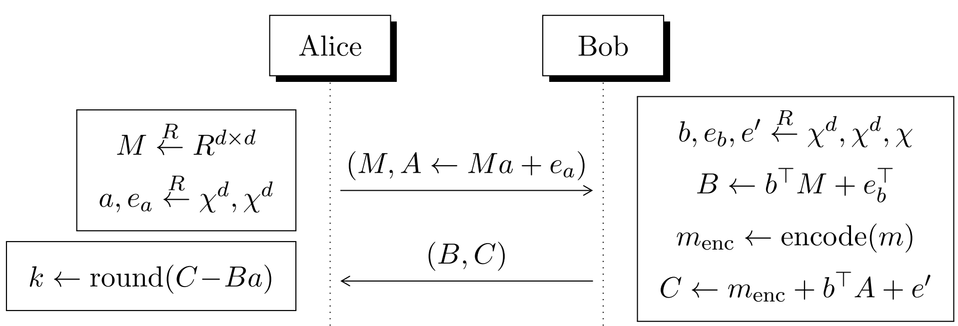

Figure 3: Simplified Module Learning With Errors encryption over a ring R and noise distribution χ. This follows [BDK$^+$17], which in turn is based on [LPR10] (which uses $d = 1$, i.e. Ring-LWE).

is a generalized Mersenne number. The noise, encoding and decoding functions for polynomial LWE still work, with the substitution $x = q$. ThreeBears follows this pattern with some small variations, taking $x = q = 2^{10}$ and $\phi(x) = x^{312} - x^{156} - 1$. See Section 3.2 for why we didn't use a cyclotomic polynomial.

1.3 Practical details

The simplified MLWE encryption system shown in Figure 3 works fine in theory, but it is much more secure and efficient with some practical improvements.

Key encapsulation Of course, we do not actually encrypt a long message with this scheme. Instead, a short (256-bit) message m is chosen at random, and is used to derive a session key k for some symmetric encryption algorithm. So we are building a Key Encapsulation Mechanism (KEM) rather than a true encryption scheme.

Round C It is inefficient to send the entire ciphertext C: it will be rounded when Alice uses it, and she only needs as many coefficients as there are ciphertext digits. Therefore it is better to send only the digits of C that Alice needs, and only a few bits of each digit.

Clarifier When multiplying two numbers mod N, the product contains terms with significance greater than $x^{3D/2}$. Reducing this value mod $\phi(x) = x^D - x^{D/2} - 1$ requires two reduction steps: first to $x^D + x^{D/2}$ and then to $2 \cdot x^{D/2} + 1$. This double-reduction distorts and amplifies the noise which is added to the message, which increases the probability of decryption failure. We could instead use Montgomery multiplication $\mathrm{montmul}(a, b) := a \cdot b / x^D$, but this would have the same effect on the least-significant digits. It is better to use an operation halfway between normal and Montgomery multiplication, namely:

$$\begin{aligned} a * b &:= a \cdot b / x^{D/2} \bmod N \\ &= a \cdot b \cdot (x^{D/2} - 1) \bmod N \end{aligned}$$

The ring $\mathbb{Z}/N$ is still a ring (and if N is prime, a field) with the operations $(+, *)$ instead of $(+, \cdot)$. We call the value $x^{D/2} - 1$ a *clarifier* because it reduces distortion of the noise. Because the clarifier has a special form, it is efficient to compute $a * b$ directly, rather than using two multiplications. Even with the clarifier, there is still more noise in the digits near $x^{D/2}$, so the digits used for encryption are the ones farthest from $x^{D/2}$.

Private key as seed Alice's keys can easily be created pseudorandomly from a small seed. We define explicitly how to do this, so that the seed functions as a private key. This makes private keys easier to store. If ThreeBears becomes a standard, it may be preferable to allow other methods of key expansion that produce the same distribution, in case they should be superior for some property like side-channel resistance.

Matrix as seed Alice's public key includes a large matrix M, which is expanded pseudorandomly from a seed. There's no need to actually send M. Instead, ThreeBears generates M from a small (24-byte) derived seed, and sends that seed instead of M.

Fujisaki-Okamoto transform As described above, our MLWE encryption system would be secure against passive attack but not against a chosen-ciphertext attack [HNP+03]. The standard defense is to use a KEM variant of the Fujisaki-Okamoto transform [FO99]. In this transform, instead of choosing his randomness (b, e_b, e') at random, Bob generates them deterministically by hashing the public key and m; so the entire ciphertext is a deterministic function of the public key and m. After Alice recovers m

she can check that encryption was performed properly. If not, then she rejects the ciphertext as invalid. We specify variants of ThreeBears with Fujisaki-Okamoto (for public-key encryption) and without it (for ephemeral key exchange).

Error-correcting code Like most LPR10-based encryption algorithms, ThreeBears is not perfectly correct: there is a small chance that the noise may exceed the rounding threshold and decryption may fail. For Fujisaki-Okamoto to work, this failure probability must be cryptographically negligible. To reduce the failure probability for a given noise level, we apply a forward error-correcting code to m. There are many options for such a code. We chose a Melas-type BCH code that corrects 2 errors, since this affords a decent security increase and is easy to perform in constant time.

2 Specification

Here is the detailed specification of ThreeBears.

2.1 Notation

Integers Let $\mathbb{Z}$ denote the integers, and $\mathbb{Z}/N$ the ring of integers modulo some integer N. For an element $x \in \mathbb{Z}/N$, 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 $\mathbb{Z}/N$ 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' 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.

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.

[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 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 five sets of toy parameters, shown in Table 3. TeddyBear is simply too small: it has dimension $1 \cdot 240$ compared to BabyBear's $2 \cdot 312$. This system has a core-sieve difficulty of 2^{60}, but core-sieve is an underestimate and TeddyBear exceeds the LWE challenges that have been solved so far. In order to make a challenge that might actually be broken, we propose the even smaller GummyBear, which also features a non-prime N. In the opposite direction, DropBear should be secure against CPA attacks, but its failure rate of around 1.1% makes it vulnerable to CCA attacks. This should make it much easier to break than TeddyBear in practice.

Finally, Koala is a candidate for lightweight use. Since it might eventually be implemented, we specify that it has

$$\texttt{privateKeyBytes} = \texttt{encryptionSeedBytes} = 24$$

and `matrixSeedBytes=16`, which gives it a public key of 556 bytes and a capsule of 645 bytes.

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

System	d	cca	σ^2	Failure	Lattice security Classical	Quantum	Class
BabyBear	2	0	1	$\approx 2^{-58}$	168	153	II
		1	9/16	$< 2^{-156}$	154	140	II
MamaBear	3	0	7/8	$\approx 2^{-51}$	262	238	V
		1	13/32	$< 2^{-206}$	235	213	IV
PapaBear	4	0	3/4	$\approx 2^{-52}$	351	318	V
		1	5/16	$< 2^{-256}$	314	280	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	Security
GummyBear	9	120	1	0	1	$\approx 2^{-62}$	21
TeddyBear	9	240	1	1	3/4	$\approx 2^{-58}$	60
DropBear	10	312	2	1	2	$\approx 2^{-6}$	184
Koala	9	240	2	1	11/32	$\approx 2^{-130}$	115
KoalaEphem	9	240	2	0	21/32	$\approx 2^{-42}$	128

Table 3: ThreeBears toy parameters. “Security” is classical core-sieve.

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

$$[\![1, 40, 24, 32, 0, 32, 10, 56, 1, 3, 51, 4, 18, 1]\!]$$

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

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

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.

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.

Algorithm 1: Uniform and noise samplers

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

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

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

end

Function $\texttt{noise}_p$(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** $D - 1$ **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

2.4.3 Extracting bits from a number

In order to encrypt using ThreeBears, we need to extract bits from an approximate shared secret $S \bmod N$. 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 $\left\lfloor \text{res}(S) \cdot 2^b / x^{j+1} \right\rfloor$;
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 we 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.

Algorithm 4: Melas FEC decode

```
Function FecDecode_18(B) is
    input  : Encoded bit sequence B of length n, where 18 ≤ n ≤ 511
    output: Decoded bit sequence of length n − 18

    // Form a quadratic equation from syndrome.
    s ← syndrome_18(B);
    Q ← seq_{9+1}(0x211);
    c ← step^9(Q, s) ⊙ step^9(Q, reverse(s));
    r ← step^17 (Q, c^{⊙510});
    s_0 ← step^{511−n}(Q, s);

    // Solve quadratic for error locators using half-trace
    halfTraceTable ← ⟦36, 10, 43, 215, 52, 11, 116, 244, 0⟧;
    halfTrace ← ⊕_{i=0}^{8} (r_i · seq_9(halfTraceTable_i));
    (e_0, e_1) ← (s_0 ⊙ halfTrace, (s_0 ⊙ halfTrace) ⊕ s_0);

    // Correct the errors using the locators
    for i = 0 to n − 18 − 1 do
        if step^i(Q, e_0) = seq_9(1) or step^i(Q, e_1) = seq_9(1) then
            B_i ← B_i ⊕ 1;
        end
    end
    return ⟦B_i⟧_{i=0}^{n−18−1};
end
```

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 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, and returns a shared secret as shown in Algorithm 7.

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 ← noise_1(sk, i);

    // Generate a random matrix, multiply and add noise
    matrixSeed ← H_1(sk, matrixSeedLen);
    for i, j = 0 to d − 1 do M_{i,j} ← uniform(matrixSeed, i, j);
    for i = 0 to d − 1 do
        A_i ← noise_1(sk, d + i) + Σ_{j=0}^{d−1} M_{i,j} · a_j · clar
    end

    // Output
    pk ← (matrixSeed, ⟦A_i⟧_{i=0}^{d−1});
    return pk;
end

Function Keypair() is
    sk ← RandomBytes(privateKeyBytes);
    return (sk, GetPubKey(sk));
end
```

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}) ← pk;

    // Generate ephemeral private key and make I-MLWE instance
    for i = 0 to d − 1 do b_i ← noise_2(matrixSeed||seed||iv, i);
    for i, j = 0 to d − 1 do M_{i,j} ← uniform(matrixSeed, i, j);
    for i = 0 to d − 1 do
        B_i ← noise_2(matrixSeed||seed||iv, d + i) + Σ_{j=0}^{d-1} M_{j,i} · b_j · clar;
    end

    // Form plaintext; encrypt using approximate shared secret
    C ← noise_2(matrixSeed||seed||iv, 2 · d) + Σ_{j=0}^{d-1} A_j · b_j · clar;
    if CCA then pt ← seed;
    else pt ← H_2(matrixSeed||seed||iv, encSeedBytes);
    encpt ← FecEncode(pt as a sequence of bits);
    for i = 0 to length(encpt) − 1 do
        encr_i ← extract_ℓ(C, i) + 8 · encoded_seed_i mod 16;
    end

    // Output
    shared_secret ← H_2(matrixSeed||pt||iv, sharedSecretBytes);
    capsule ← (⟦B_j⟧_{j=0}^{d-1}, nibbles ⟦encr_i⟧_{i=0}^{length(pt)-1}, iv);
    return (shared_secret, capsule);
end

Function Encapsulate(pk) is
    (seed, iv) ←(RandomBytes(encSeedBytes), RandomBytes(ivBytes));
    return EncapsDet(pk, seed, iv);
end
```

2.8 Implicit rejection

As of the July 2019 update, the CCA-secure parameters of ThreeBears uses implicit rejection of invalid ciphertexts. This means that if decapsulation fails, instead of returning an error code, we return a pseudorandom function of the ciphertext. The algorithm used is $U_m^{\not\perp}$, as described in [HHK17].

Algorithm 7: Decapsulation

```
Function Decapsulate(sk, capsule) is
    input  : Private key sk, capsule
    output: Shared secret

    // Unpack private key and capsule
    for i = 0 to d − 1 do a_i ← noise_1(sk, i);
    ( [[B_j]]_{j=0}^{d−1}, nibbles [[encr_i]], iv ) ← capsule;

    // Calculate approximate shared secret and decrypt seed
    C ← Σ_{j=0}^{d−1} B_j · a_j · clar;
    for i = 0 to length(encr) do
        encoded_seed_i ← ⌊ (2·encr_i − extract_{ℓ+1}(C, i)) / 2^ℓ ⌉
    end
    seed ← FecDecode(encoded_seed);

    if CCA then
        // Re-encapsulate to check that capsule was honest
        (shared_secret, capsule′) ← EncapsDet(GetPubKey(sk), seed, iv);
        if capsule′ = capsule then return shared_secret;
        else
            prfk ← H_1(sk||[[0xFF]], privateKeyBytes);
            shared_secret ←
                H_3(prfk||capsule, sharedSecretBytes);
            return shared_secret
        end
    else
        // Don't check:  just calculate the shared secret
        matrixSeed ← H_1(sk, matrixSeedLen);
        shared_secret ← H_2(matrixSeed||seed||iv, sharedSecretBytes);
        return shared_secret
    end
end
```

3 Design Rationale

We based our overall design on the KYBER MLWE system [BDK+17]. We liked that MLWE allows systems of different security levels to use the same ring 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 estimated 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 plenty of memory and only need keys ephemerally can cache the intermediates a_i and M_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

$$B = b^\top M + e_b^\top,\ C = b^\top (Ma + e_a) + e'_b$$

The latter term can be rewritten as $(b^\top M)a + b^\top e_a + e'_b$, which costs a total of $d\cdot(d+2)$ multiplications. Caching the public key component $A := Ma+e_a$ would only reduce this to $d \cdot (d + 1)$ multiplications, because $b^\top M$ and $b^\top A$ have to be computed anyway, but it would save on hashing.[2]

[2]It is safe to compute a uniformly random projection $b^\top (M \cdot r) \stackrel{?}{=} (B - e_b) \cdot r$ instead, which costs $3d$ multiplications instead of $d(d + 1)$ if Mr is cached. It probably isn't safe to replace r with the private key s here, because s doesn't have full entropy.

No plaintext confirmation hash We chose not to use an extra hash (as was used in [TU16]). Our security proof shows why it would be reduntant for ThreeBears. Roughly, the plaintext is already hashed to produce noise for encryption, and the e_b component propagates almost directly to the ciphertext. This suffices in the proof, in place of an extra hash.

Implicit rejection We had two options on how to deal with decapsulation failures. We could reject explicitly by returning a special failure symbol "⊥", or in C, an error code. Or we could reject implicitly by returning a pseudorandom key and no error, which would cause later protocol steps to fail.

Previous versions of ThreeBears used explicit rejection, which is provably secure for ThreeBears. This is because explicit rejection is simpler and about 10% faster. However, implicit rejection is easier to analyze in general [SXY17, JZC+17, BHHP19, BHHP19]. It also provides a simpler API and is easier to audit for constant-time behavior. So in v2, we converted to implicit rejection.

When implementing implicit rejection, we had a choice between variants of the $U^{\not\perp}$ transform, which hashes the message and ciphertext, and the $U^{\not\perp}_m$ transform, which hashes only the message [HHK17]. The two options have equal security, at least in the quantum random oracle model [BHHP19]. $U^{\not\perp}_m$ is slightly simpler and has faster encapsulation, so we chose it.

Most other proposals use $U^{\not\perp}$ instead of $U^{\not\perp}_m$. The main reason is that $U^{\not\perp}$ removes incentives for implementers to skip the (otherwise slower) PRF calculation when the message is accepted. In that case, a side-channel attack could reveal whether the message was accepted. In itself, this wouldn't be a serious problem for ThreeBears, since it is provably secure with explicit rejection. We strongly recommend that implementors make the decapsulation routines constant-time anyway, since this is important for auditability.

	CPA-secure			CCA-secure		
d	σ^2	Failure	Q-core-sieve	σ^2	Failure	Q-core-sieve
2	5/8	2^{-54}	142	3/8	2^{-133}	132
3	1/2	2^{-57}	219	7/32	2^{-218}	194
4	7/16	2^{-56}	298	3/16	2^{-224}	257

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

Hash matrix seed but not ciphertext We chose not to hash the entire public key or entire ciphertext. Doing so would complicate and slow down the implementation, require more memory, and prevent efficient fusing of key generation and decryption. Furthermore, our proof indicates that it would not affect CCA 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.

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. Decoding only two errors avoids a tricky constant-time Berlekamp-Massey step, and 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 down-

Errors corrected	0	parity	1	2	3	4	5	6
Variance in 32nds	7	9	10	13	15	17	18	20
Q-core-sieve security	194	202	205	213	217	221	223	227

Table 5: Effectiveness of error correction to increase security. Alternative parameters with more or less error correction. $D = 312, d = 3, x = 2^{10}$, CCA_2-secure, failure $< 2^{-192}$.

sides 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. While the system is still viable in this case, it is not as easy to convincingly reach Class V IND-CCA security. It is probably better to use a larger digit x in that case, which would reduce efficiency. Table 5 compares the effectiveness of BCH error-correcting codes which would correct n errors using $9n$ bits [3]. This allows more noise, and therefore increases security at the cost of complexity. The table also includes the option of using a single-bit parity code on each 64-bit section of the public key, with maximum-likelihood decoding if the parity check fails. Overall, most of the security improvement is seen when moving from correcting no errors to correcting 1 or 2 errors.

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

[3] The failure estimates in this table were made using our original estimation technique, which extrapolates the probability of n-bit failures from that of 1- and 2-bit failures with a given amount of ciphertext noise.

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. We show how to mitigate this attack by attaching an initialization vector (IV) to each ciphertext. But our recommended parameters set the IV length to 0 bytes (unused), because we don't think that the multi-target brute force attack is a real risk.

We chose a 192-bit matrix seed so that matrix seeds will almost certainly never collide even with 2^{64} honestly-generated keys. If they do collide, it only gives the adversary a tiny advantage anyway. See `proof.pdf` for more details.

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 $2^p - 1 = 2^k \cdot x^D - 1$, where at best k can be ± 1: it can't be 0 because p is prime but $D \cdot \lg x$ is composite. Therefore reduction modulo a Mersenne 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 core-sieve and q-core-sieve security estimates for the larger prime better match the NIST target security levels.
- The larger prime allows us to use FEC. The smaller prime would accommodate a parity code, but no more.
- The larger prime is simpler to implement efficiently. In particular we don't have to worry about overflow beyond 2^{2600}.

	CPA-secure			CCA-secure		
d	σ^2	Failure	Q-core-sieve	σ^2	Failure	Q-core-sieve
2	3/4	2^{-64}	118	9/16	2^{-108}	112
3	5/8	2^{-63}	184	3/8	2^{-156}	171
4	9/16	2^{-59}	251	9/32	2^{-196}	228

Table 6: Alternative parameters with $D = 260, x = 2^{10}$, parity check with maximum-likelihood decoding.

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' 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 combinatorial attacks become 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 best tradeoff of security strength against ciphertext size is achieved with $\ell = 3$, but with $\ell = 4$ only slightly worse. We chose $\ell = 4$ because it's simpler to implement.

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}. For the CCA-secure instances, we set the noise so that the failure rate is around $2^{-\lambda}$, where λ is the core-sieve estimated bit security level. For a classical attacker, no single-key attack can cause a failure in expected time less than $1/\delta > 2^{\lambda}$. Known attacks with bounded queries are much weaker than this, even if a quantum computer is available [DVV18b, DVV18a].

For BabyBear, we set δ closer to λ in the hope that, since both the lattice security and CCA-security estimates are underestimates, BabyBear might actually reach Class III security. We also wanted to keep it well below 2^{-128} so that a failure attack is clearly entirely infeasible, even with a quantum computer and significant improvement in attack strategy.

For the other systems, we set δ just past the target security level. This is due to a philosophy of risk mitigation. Nobody is actually worried about an attacker performing 2^{192} operations, but about a breakthrough that reduces the work to a feasible level. CCA attacks have less room for improvement than lattice attacks, and so are less likely to impact the practical security of ThreeBears. We just wanted to make sure that the failure rate isn't even a certificational weakness.

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, and in fact the latest attack family doesn't yet have an efficient performance model [ADH+19]. 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^{154} for a classical computer and 2^{140} 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 is still suitable for lightweight devices. It might become the primary recommendation in the future.

The stronger MamaBear seems comfortably out of reach of known attacks, requiring 2^{236} effort with a classical computer or 2^{214} 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, even under its core-sieve security (under)estimate. But this is probably overkill for most users. MamaBearEphem reaches Class V anyway, and it's possible that MamaBear does too once all costs are accounted for.

We recommend the CCA instances in general, and we remind designers that within the CCA security model, the public key must be authenticated. The CPA instances are designed primarily to be used within an authenticated key exchange protocol, which will have to provide CCA security at the AKE level. For the CPA instances, each keypair must be used only once.

4 Security analysis

4.1 The I-MLWE problem

ThreeBears' security is based on the difficulty of the Integer Module Learning with Errors (I-MLWE) problem, as defined in Section 1.1. Gu proved that asymptotically, Integer RLWE and Polynomial RLWE have similar security [Chu17], and this proof should carry over directly to I-MLWE. As is often the case with lattice security reductions, this proof is asymptotic and does not apply to practical parameters. But we also see no reason for I-MLWE to be easier than P-MLWE, so we expect the two problems to be similar in practical complexity.

4.2 The CCA transform

Included with this submission in `proof.pdf` is a proof which analyzes ThreeBears' IND-CCA security in the quantum random oracle model. It shows that for an IND-CCA adversary $\mathcal{A}$ which makes q quantum queries at depth d to cSHAKE as a quantum-accessible random oracle, there is a quantum algorithm $\mathcal{B}$ using only slightly more resources than $\mathcal{A}$, such that

$$\begin{aligned}\mathrm{Adv}_{\text{IND-CCA}}(\mathcal{A}) \quad &\lessapprox \quad 4\sqrt{2(d+1)\cdot(\mathrm{Adv}_{\text{I-MLWE}}(\mathcal{B}) + q/2^{8\cdot\texttt{encryptionSeedBytes}-3})} \\ &+ \quad 4\sqrt{qd/2^{8\cdot\texttt{privateKeyBytes}}} + 16qd\delta + \text{negl.}\end{aligned}$$

where

- $\mathrm{Adv}_{\text{IND-CCA}}(\mathcal{A})$ 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.
- negl. is much less than the other terms, at least for the recommended parameters, and is made more precise in `proof.pdf`.

Parsing the bound, the attacks on ThreeBears are limited to roughly:

- Breaking I-MLWE. Our analysis is loose by a factor of about d.
- Grover's algorithm to discover the private key.
- Grover's algorithm to discover the encryption seed.
- Grover's algorithm to find ciphertexts that are likely to cause failures. This is also loose: known attacks to find decryption failures are much less effective.

The proof also shows security bounds for multiple victim keys and multiple challenge ciphertexts per key. The CCA transform appears to have nearly optimal security for attacks which use multiple targets to lower the adversary's effort (e.g. by breaking only one of many keys). We didn't model attacks which increase the adversary's reward (e.g. by breaking many keys in a batch). We also did not analyze multiple-target I-MLWE attacks in either of these models. There is some work on batch attacks to find short lattice vectors in a ring [PMHS19], but nothing yet which would imply a batch attack on ThreeBears.

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 really run faster on a quantum computer.

5.2 Inverting the hash

If the adversary could 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 "core-sieve" technique of NewHope [ADPS15] and Kyber [BDK+17], specifically using Schanck's estimator [Sch]. The results for primal attacks are shown in Table 7.

The "core-sieve" estimator lags behind the state of the art, which currently appears to be G6K [ADH+19]. Unfortunately, G6K is very complicated, and we are not aware of any scripts that efficiently model it.

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. Overall, we expect the dual attack to be more difficult than the primal attack.

5.4 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 Schanck's security estimator [Sch] to evaluate the feasibility of this attack using the same "core-sieve" estimate as for the direct attack. We see that this attack does not appear to reduce the security of any of the recommended parameters.

Since the hybrid attack trades off combinatorial work for lattice-reduction work, and our lattice reduction estimates are very optimistic from the attacker's point of view, the best attack is probably a hybrid attack. But we only expect this if the lattice part of the attack is harder than the estimate, and therefore infeasible.

System	Classical Lattice	Classical Hybrid	Quantum Lattice	Quantum Hybrid	Class
BabyBear	154	190	140	180	II
BabyBearEphem	168	210	153	197	II
MamaBear	235	241	213	228	IV
MamaBearEphem	262	333	238	314	V
PapaBear	314	317	285	300	V
PapaBearEphem	354	452	321	428	V

Table 7: Log_2 difficulty estimates for primal hybrid attack.

5.5 Quantum Ideal-SVP or DCP algorithm

Combining Regev's reduction from the shortest vector problem to the Dihedral Coset Problem (DCP) [Reg02] with Kuperberg's subexponential-time algorithm for the DCP [Kup05] could lead to a quantum algorithm for SVP. However, it is unlikely to perform better than classical sieves [Epe14].

In a similar vein, there is a polynomial-time algorithm which solves the approximate shortest vector problem in an ideal [CDW17]. But since the approximation is subexponentially bad, this appears not to improve attacks on practical parameters [DPW19].

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^{-156} for all recommended CCA-secure 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^{156} per failure. This issue is studied in [DVV18b, DVV18a]. In CCA-secure versions of ThreeBears, sampling the noise includes the public key, so this effort cannot be re-used across keys, which prevents attacks on earlier versions of LAC [AS18, Ham18a] and Round5 [Ham18b].

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

A quantum attacker could try to use Grover's algorithm to find ciphertexts with higher p_{failure}. We believe that this cannot be more than marginally effective. One may show that Grover's algorithm raises the expected failure probability per random-oracle query from mean(p_{failure}) to at most root-mean-square(p_{failure}), and that no quantum random-oracle algorithm can reduce the work by more than MAXDEPTH. So even if an adversary could semi-accurately evaluate whether a given ciphertext would cause a failure, a single-key Grover attack would only reduce the security class from IV to III, or from II to I.

We are confident that failure attacks would have been infeasible for our first round parameters. But given the problems they caused for LAC and Round5, we reduced the failure probability in the second round so that they will be entirely infeasible, even with significant improvements. Perhaps with more study the noise level can be raised again.

5.7 Malleability and kleptography

The CPA-secure variants of ThreeBears have malleable ciphertexts. If a few low-significance bits in the capsule are changed, the resulting shared secret probably remains the same. Both the CPA- and CCA-secure variants have this property for the public key as well. We could have reduced malleability at a cost in performance and complexity, by hashing the entire public key and ciphertext into the final output. We decided against this because malleability doesn't usually matter, and when it does matter, the proper defense is at the protocol level and not the KEM level.

Malleability is not a serious problem for IND-CCA-secure encryption, because the public key must be authenticated anyway to prevent man-in-the-middle attacks. One can invent a threat model where malleability costs a few bits of security, such as an adversary who can modify public keys but not ciphertexts, but such a threat model probably isn't realistic.

Malleability is a greater threat for authenticated key exchange (AKE) protocols. But ThreeBears is not an AKE, and AKEs require their own design and analysis with protocol-level countermeasures against modification of packets. We expect that authors using ThreeBears in an AKE would use a dedicated FO mode, as in [HKSU18]. An AKE might be built on a IND-CPA-secure KEM with negligible failure probability. This can be obtained by using our IND-CCA parameter sets, but setting the `cca` flag to zero.

Malleability can also be used for kleptography, in which the adversary subtly modifies the public key or ciphertext to leak secret information [KLT17]. Our IND-CCA mode mitigates kleptographic attacks on encapsulation, but not key generation. Additional hashing wouldn't significantly reduce the attack surface for kleptograhpy.

CPU	Arch flags	Keccak	asm
Intel Skylake	`-march=native`	`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 `-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.

To measure concrete performance, we benchmarked our code on several different platforms, as shown in Table 9.

For each platform except for Cortex-M4, we compiled each instance with

```
clang-8 -Os -fomit-frame-pointer -DNDEBUG
```

and the additional flags shown in Table 8.[4] 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. For the other platforms, we used C code only.

[4] `-Os` means to optimize for size. But whereas `gcc -Os` actually optimizes for size, `clang -Os` effectively optimizes for speed while keeping the size reasonable.

For Cortex-M4, we integrated ThreeBears into the PQM4 project [KRSS]. Its benchmarking scripts compiled each instance with

```
gcc-8 -O2 -fomit-frame-pointer -DNDEBUG
```

We used 1-level Karatsuba multiplication, since this was faster and used less memory than 2-level, and we linked the optimized Keccak libraries from PQM4. Our Cortex-M4 code contains no assembly.

Speed-optimized implementations will probably trade memory for decapsulation time, by caching some components of the public and private key. The amount to be cached depends on the constraints of the application. Our implementations follow this specification's API, so they don't cache anything.

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, optimizing the multiplication algorithm with NEON should provide a large improvement. For Cortex-M4, we didn't closely investigate how to optimize.

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 it enables larger noise, which can result in more iterations in the `noise` function.

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. The concrete measurements are shown in Table 10.

Code size We measured the total code size on each platform to implement MamaBear, using the same compilation flags that we used to measure

System	CPA-secure KeyGen	CPA-secure Enc	CPA-secure Dec	CCA-secure KeyGen	CCA-secure Enc	CCA-secure Dec
Skylake (high speed)						
BabyBear	41k	62k	28k	41k	60k	113k
MamaBear	84k	103k	34k	79k	97k	169k
PapaBear	124k	153k	40k	117k	144k	230k
Cortex-A53						
BabyBear	155k	214k	82k	156k	211k	373k
MamaBear	307k	383k	112k	301k	373k	597k
PapaBear	508k	602k	143k	500k	590k	879k
Cortex-A8						
BabyBear	345k	503k	177k	351k	498k	846k
MamaBear	733k	946k	257k	724k	938k	1436k
PapaBear	1244k	1516k	326k	1236k	1501k	2159k
Cortex-M4 (high speed)						
BabyBear	590k	771k	251k	590k	756k	1308k
MamaBear	1160k	1394k	350k	1150k	1394k	2146k
PapaBear	1919k	2208k	449k	1906k	2178k	3177k
Cortex-M4 (low memory)						
BabyBear	681k	952k	251k	681k	937k	1487k
MamaBear	1433k	1803k	350k	1416k	1766k	2544k
PapaBear	2464k	2933k	449k	2436k	2884k	3884k

Table 9: Runtime of (KeyGen, Encaps, Decaps) in cycles.

The platforms tested were:

- Intel NUC6i5SYH with Core i3-6100U “Skylake” 64-bit CPU (2.3GHz)
- Raspberry Pi 3 with ARM Cortex-A53 64-bit CPU (1.2GHz)
- BeagleBone Black with ARM Cortex-A8 32-bit CPU (1.0GHz)
- STM32F407G-DISC1 with ARM Cortex-M4 32-bit CPU (168 MHz)

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

Table 10: ThreeBears object sizes in bytes

				Cortex-M4	
Component	Skylake	Cortex-A53	Cortex-A8	Speed	Mem
Arithmetic	2194	1892	1424	852	852
Melas FEC	659	541	431	417	417
cSHAKE	1490	900	866	660	660
Main system	3397	2995	2173	2303	2074
Total	7740	6328	4894	4232	4087

Table 11: Code size for MamaBear in bytes.

performance. The code size does not include Keccak, since we linked an external Keccak library[5], nor does it include system libraries like `libc`. It includes an API wrapper for `pqm4`, but not on other platforms. The sizes are shown in Table 11. Ephemeral instances are slightly smaller.

Memory usage We measured the stack memory usage of each top-level function on Skylake using Valgrind's `lackey` tool. We also measured the Cortex-M4 implementations using `pqm4`'s benchmarking tool. These measurements included the memory used by ThreeBears internally, including hash contexts, function calls and wrappers, but not the input or output. The results are shown in Table 12, and should be regarded as approximate[6].

[5]This is actually a little silly, because a fully-unrolled Keccak implementation produces much more object code than ThreeBears.

[6] Here are some examples of things that were not accounted for: variation in the stack's alignment, variation in shared libraries, and memory used by Keccak Code Package's initialization routines. If these use extra memory, it is probably negligible in practice.

System	CPA-secure Keygen	CPA-secure Enc	CPA-secure Dec	CCA-secure Keygen	CCA-secure Enc	CCA-secure Dec
Skylake (high speed)						
BabyBear	6216	6648	4232	6216	6648	8200
MamaBear	9128	9544	4648	9128	9560	11528
PapaBear	12872	13288	5064	12872	13304	15688
Skylake (low memory)						
All instances	2392	2424	2168	2392	2424	3080
Cortex-M4 (high speed)						
BabyBear	2752	2824	2080	2752	2824	4944
MamaBear	3248	3304	2080	3280	3312	5904
PapaBear	3760	3792	2080	3760	3824	6896
Cortex-M4 (low memory)						
All instances	2280	2344	2112	2312	2344	3024

Table 12: ThreeBears memory usage bytes, excluding input and output. In some cases, "all instances" actually vary by a few bytes, in which case the maximum is reported.

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

Simplicity ThreeBears has a relatively simple specification and admits a relatively simple implementation. On most platforms, ThreeBears doesn't need vectorization to achieve respectable speed, except perhaps in a separate Keccak library. These advantages mean that its code is small, simple and easy to audit. Forward error correction adds some complexity, but it's only some 75 lines of C code and it's easy to test separately.

Size To hedge our new security assumption, we have chosen larger instances than other RLWE systems. Despite this, 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. Private keys are just random seeds, and are only 40 bytes. Code sizes are under 10kB plus Keccak, and stack requirements can be pushed near 3kB plus the input and output.

Speed As in most RLWE and MLWE systems, key generation, encapsulation and decapsulation are also very fast. They are typically significantly faster even than elliptic curve Diffie-Hellman.

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.

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 any results for practical parameter sizes.

Design auditing The analysis of failure probabilities for ThreeBears is very complex. Furthermore, its proof of security uses properties specific to LWE, rather than following generic, modular proofs of security. This raises the risk of an auditing mistake.

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.

Flexibility ThreeBears can only be used for key encapsulation and encryption. So far there is no I-MLWE signature scheme. Furthermore, ThreeBears' parameters are less tunable than a cyclotomic RLWE scheme.

Side channels Because ThreeBears has long carry chains, it may be more complex to protect the system against side channels than a polynomial LWE scheme.

7.3 Suitability for constrained environments

ThreeBears is suitable for smart card implementation, and implementors can reuse their RSA big-number engines. We don't expect ThreeBears to be as competitive on 8-bit microcontrollers.

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, Amit Deo and Andris Ambainis for discussions of the quantum analysis of the CCA transform.

Thanks to Mark Marson for many helpful discussions, including feedback on drafts of this work.

Thanks to Rambus for supporting this work.

References

[ADH+19] Martin R. Albrecht, Lo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. Cryptology ePrint Archive, Report 2019/089, 2019. https://eprint.iacr.org/2019/089.

[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.

[AS18] Jacob Alperin-Sheriff. Official comment: LAC. NIST PQC forum email list, 2018. https://csrc.nist.gov/CSRC/.../round-1/official-comments/LAC-official-comment.pdf.

[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.

[BHHP19] Nina Bindel, Mike Hamburg, Andreas Hlsing, and Edoardo Persichetti. Tighter proofs of cca security in the quantum random oracle model. Cryptology ePrint Archive, Report 2019/590, 2019. https://eprint.iacr.org/2019/590.

[CDW17] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-SVP. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 324–348. Springer, Heidelberg, April / May 2017. doi:10.1007/978-3-319-56620-7_12.

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

[DPW19] Lo Ducas, Maxime Planon, and Benjamin Wesolowski. On the shortness of vectors to be found by the ideal-svp quantum algorithm. Cryptology ePrint Archive, Report 2019/234, 2019. https://eprint.iacr.org/2019/234.

[DVV18a] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. The impact of error dependencies on Ring/Mod-LWE/LWR based schemes. Cryptology ePrint Archive, Report 2018/1172, 2018. https://eprint.iacr.org/2018/1172.

[DVV18b] Jan-Pieter D'Anvers, Frederik Vercauteren, and Ingrid Verbauwhede. On the impact of decryption failures on the security of LWE/LWR based schemes. Cryptology ePrint Archive, Report 2018/1089, 2018. https://eprint.iacr.org/2018/1089.

[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.

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 10–18. Springer, Heidelberg, August 1984.

[Epe14] Daniel Epelbaum. On quantum sieve approaches to the lattice shortest vector problem, 2014. https://www.scottaaronson.com/showcase3/epelbaum-daniel.pdf.

[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. doi:10.1007/3-540-48405-1_34.

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

[Ham18a] Mike Hamburg. Official comment: LAC. NIST PQC forum email list, 2018. https://csrc.nist.gov/CSRC/.../round-1/official-comments/LAC-official-comment.pdf.

[Ham18b] Mike Hamburg. Official comment: Round5 = round2 + hila5. NIST PQC forum email list, 2018. https://csrc.nist.gov/CSRC/...1/official-comments/Round5-official-comment.pdf.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, November 2017. doi:10.1007/978-3-319-70500-2_12.

[HKSU18] Kathrin Hvelmanns, Eike Kiltz, Sven Schge, and Dominique Unruh. Generic authenticated key exchange in the quantum random oracle model. Cryptology ePrint Archive, Report 2018/928, 2018. `https://eprint.iacr.org/2018/928`.

[HNP+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. `doi:10.1007/978-3-540-45146-4_14`.

[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`.

[KLT17] Robin Kwant, Tanja Lange, and Kimberley Thissen. Lattice klepto: Turning post-quantum crypto against itself. Cryptology ePrint Archive, Report 2017/1140, 2017. `https://eprint.iacr.org/2017/1140`.

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

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`.

[Kup05] Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM Journal on Computing*, 35(1):170–188, 2005.

[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 / June 2010. doi:10.1007/978-3-642-13190-5_1.

[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.

[PMHS19] Alice Pellet-Mary, Guillaume Hanrot, and Damien Stehl. Approx-svp in ideal lattices with pre-processing. Cryptology ePrint Archive, Report 2019/215, 2019. https://eprint.iacr.org/2019/215.

[Reg02] Oded Regev. Quantum computation and lattice problems. In *43rd FOCS*, pages 520–529. IEEE Computer Society Press, November 2002. doi:10.1109/SFCS.2002.1181976.

[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. Scripts for estimating the security of lattice based cryptosystems. https://github.com/jschanck/estimator, retrieved Feb 26, 2019.

[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*, volume 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 11^{th} 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