Supporting Documentation the Mersenne-756839 KEM

Divesh Aggarwal* Antoine Joux† Anupam Prakash‡ Miklos Santha§

November 30, 2017

1 Notations

The Mersenne-756839 key exchange mechanism specified in the present document relies on the choice of the following pameters:

$$\begin{aligned} n &= 756839, \\ h &= 256 \text{ and} \\ \rho &= 2048. \end{aligned}$$

These being fixed, we define $P = 2^n - 1$ which is a Mersenne prime number. Most operations occuring in the cryptosystem are basic arithmetic operations modulo P. Numbers modulo P are represented by their unique representative in $[0, P-1]$.

Moreover, we set $K = 32 \lceil \frac{n}{256} \rceil$ and at the computer level, we represent every number x modulo P by an array of K bytes. We choose K as a multiple of 32 in order to ease respecting memory alignement constraints on modern computers. The array that represents x is denoted by $[x]$ and starts with the low order byte. Conversely, given a byte array B of length L, we let $V(B)$ denote its value computed from the formula:

$$V(B) = \sum_{i=0}^{L-1} 2^{8i} B_i.$$

Thus, we have the relation:

$$x = V([x]).$$

Given a byte b, $HW(b)$ denotes the Hamming weight of b, i.e. the number of bits set to 1 in its binary expansion. Similarly, given an array of bytes B, $HW(B)$ is its Hamming weight, which is equal to the sum of the Hamming weight of the individual bytes of B. The notation $\oplus$ denotes the exclusive-OR of bytes or arrays of bytes.

*School of Computing and CQT, NUS.

†Chaire de Cryptologie de la Fondation de l'UPMC; Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606, Paris, France

‡School of Physical and Mathematical Sciences, Nanyang Technological University and Centre for Quantum Technologies, National University of Singapore, Singapore.

§IRIF, Université Paris Diderot, CNRS, 75205 Paris, France; and Centre for Quantum Technologies, National University of Singapore Singapore 117543

Given an array of bytes B and two integers $i \leq j$, we denote by $B_{[i\ldots j]}$ the sub-array formed of the bytes B_i up to B_j. Given two array B and C, we denote their concatenation by $B\|C$, it consists of the bytes of B followed by those of C.

The cryptosystem also makes use of the XOF (expandable output function) provided by NIST to provide pseudo-randomness from a 256-bit seed. To allow for an easy replacement, we call them from two wrapper functions `InitExpandableState` and `GetExpandableOutput`. Given a seed, `InitExpandableState` produces a state. Given a state and an output length in bytes, `GetExpandableOutput` produces the requested number of bytes and evolves the state.

2 Basic routines

The cryptosystem requires the ability to produce numbers x modulo P whose Hamming weights are precisely equal to h. In order to do that, we write a routine called `GenerateHSparseString` that produces the array $[x]$ representing x. Since this routine needs to generate random numbers modulo values which are close to n, we first provide a subroutine `RandomMod` that constructs such numbers using rejection sampling.

Let $n_0 = 2^{20}$ be the smallest power of 2 greater than n.

Algorithm 1 Pseudo-Random number modulo m

function RandomMod(m, `ExpandableState`)
  repeat
    Get a three bytes array B from `GetExpandableOutput` (on `ExpandableState`)
    Let $v = \left(\sum_{i=0}^{2} 2^{8i} B_i\right) \bmod n_0$
  until $v < m$
  return v
end function

Algorithm 2 Generate sparse byte array in B

procedure GenerateHSparseString(B, h, `ExpandableState`)
  Set the first h bits of the array B to 1
  Set the rest of the bits in the array B to 0
  Let $i = h - 1$
  while $i \geq 0$ **do**
    Let $j =$ `RandomMod`$(n - i,$ `ExpandableState`$)$
    Exchange bits of B in position i and $i + j$
    Let $i = i - 1$
  end while
end procedure

3 Cryptosystem

Our cryptosystem needs to specify three basic routines: key generation, Kem encapsulation and Kem decapsulation. For convenience, we give deterministic versions of key generation and Kem encapsulation and embed them in functions that includes the randomness generation. The deterministic key generation returns an expanded key, containing extra data that is useful for decapsulation but never used outside of these routines.

3.1 Key pair generation

In the system, the private key is a 256-bit value S which is used as a seed to the XOF function in order to generate an expanded private key and a public key.

Using S, we pseudo-randomly generate two numbers f and g modulo P of Hamming weight h and a number R modulo P (without any Hamming weight constraint). Then, we compute $T = (fR + g) \bmod P$.

The private key is the seed S, the public key is the pair (R, T) and the expanded private key is f.

Algorithm 3 Deterministic key generation from seed in array S

procedure DETKEYPAIR(Array `PK`, Array `LongSK`, Seed S)
  Call `InitExpandableState` on S and get `ExpandableState`
  Call `GenerateHSparseString` on `ExpandableState` to generate array A_f of weight h
  Call `GenerateHSparseString` on `ExpandableState` to generate array A_g of weight h
  Call `GetExpandableOutput` to generate array A_R of K bytes.
  Let $f = V(A_f)$, $g = V(A_g)$ and $R = V(A_R) \bmod P$.
  Let $T = (fR + g) \bmod P$.
  Set array `PK` to $[R]\|[T]$
  Set array `LongSK` to $[f]$
end procedure

Algorithm 4 Key generation

procedure KEYPAIR(Array `PK`, Array `SK`)
  Generate 32 random bytes in Array `SK` (256 bits)
  Call `DetKeypair` using `SK` as Seed
  Discard `LongSK`
  Return `PK` and `SK` as public and private key
end procedure

3.2 Kem encapsulation

Given a public key (R, T) and a seed S to the XOF, we create a Kem ciphertext and a shared secret as follows. After initialization of the XOF with the S, we first generate 256-bits of shared secret. Then we generate 3 numbers a, b_1 and b_2 of Hamming weight h and compute $C_1 = (aR + b_1) \bmod P$ and $C_2 = (aT + b_2) \bmod P$. We form a message M by concatening ρ repeated copies of each bit

of S. The ciphertext is the pair $(C_1, \mathrm{Tr}(C_2) \oplus M)$ where $\mathrm{Tr}(C_2)$ is a truncation of $[C_2]$ to the size of M.

Algorithm 5 Deterministic KEM encapsulation from seed in array S

procedure DetKemEnc(Array `CT`, Array `SS`, Array `PK`, Seed S)
  Call `InitExpandableState` on S and get `ExpandableState`
  Call `GetExpandableOutput` to fill 32-byte Array `SS` with shared secret.
  Call `GenerateHSparseString` on `ExpandableState` to generate array A_a of weight h
  Call `GenerateHSparseString` on `ExpandableState` to generate array A_{b1} of weight h
  Call `GenerateHSparseString` on `ExpandableState` to generate array A_{b2} of weight h
  Let $a = V(A_a)$, $b_1 = V(A_{b1})$ and $b_2 = V(A_{b2})$.
  Let $R = V(\mathtt{PK}_{[0\ldots K-1]})$ and $T = V(\mathtt{PK}_{[K\ldots 2K-1]})$
  Let $C_1 = (aR + b_1) \bmod P$.
  Let $C_2 = (aT + b_2) \bmod P$.
  Allocate M an array of 32ρ bytes
  for i from 0 to 255 **do**
    if Bit i of S is 0 **then**
      for j from $i\rho/8$ to $(i+1)\rho/8 - 1$ **do**
        Set byte j of M to 0
      end for
    else
      for j from $i\rho/8$ to $(i+1)\rho/8 - 1$ **do**
        Set byte j of M to 255
      end for
    end if
  end for
  Set Array `CT` to $[C_1] \| \left([C_2]_{[0\ldots 32\rho-1]} \oplus M\right)$
end procedure

Algorithm 6 Kem Encapsulation routine

procedure DetKemEnc(Array `CT`, Array `SS`, Array `PK`)
  Generate 32 random bytes seed S
  Call `DetKemEnc` on `CT`, `SS`, `PK` and S
  Return ciphertext `CT` and shared key `SS`
end procedure

3.3 Kem decapsulation

Given a private key Sk and a ciphertext, we proceed as follows. We first compute the corresponding public key (R, T) and expanded private key f. We also extract C_1 from the ciphertext. Then, we compute $C_2' = fC_1 \bmod P$ and perform an exclusive-or with the rest of the ciphertext. By construction, C_2' is close in Hamming distance to the original C_2 and we thus obtain a noisy copy of the message M that was encrypted. Taking majority in each slice of ρ bits, we recover the seed S used for Kem encapsulation.

Once S is obtained, we re-encapsulate to recover the shared secret and we check that the freshly obtained ciphertext is identical to the one we received. If the check fails, the shared key is erased and an error returned.

Algorithm 7 Kem Decapsulation routine

function KemDec(Array `CT`, Array `SS`, Array `SK`)
  Call `DetKeypair` using `SK` as Seed, producing `PK` and `LongSK`
  Let $f = V(\texttt{LongSK})$, $C_1 = V(\texttt{CT}_{[0\ldots K-1]})$
  Let $C_2' = fC_1 \bmod P$
  Let $M = [C_2']_{[0\ldots 32\rho-1]} \oplus \texttt{CT}_{[K\ldots K+32\rho-1]}$
  Let S' be a 32-byte string (initially set to 0)
  for i from 0 to 255 **do**
    if $HW(M_{[i\rho/8\ldots(i+1)\rho/8-1]}) > \rho/2$ **then**
      Set bit j of S' to 1
    end if
  end for
  Call `DetKemEnc` on `CT2`, `SS`, `PK` and S'
  if `CT` and `CT2` are identical **then**
    Return Array `SS`
  else
    Erase Array `SS`
    Return `Error`
  end if
end function

Note: In order to optimize Kem decapsulation, an implementation may memorize the public key and expanded private key. This avoids having to recompute them for every subsequent decapsulation. In that case, care should be taken to protect the expanded private key, which is as sensitive as the private key itself.

4 Design rationales

The Mersenne cryptosystem can be seen as belonging to a family that started with the Ntru cryptosystem and as been instantiated in many ways [HPS98, Reg09, LPR10, MTSB13]. The common idea behind all these cryptosystems is to work with elements in a ring which are hidden by adding some small noise. This notion of smallness needs to be somewhat preserved under the arithmetic operations. At the same time, it should be somewhat unnatural and not fully compatible with the ring structure in order to lead to hard problems.

Our goal in designing the Mersenne cryptosystem was to find a very simple instantiation of this paradigm based on the least complicated ring we could find, using only an elementary mathematical structure. This led us to consider numbers modulo a prime together with the Hamming weight to measure smallest. In this context, it is natural to restrict ourselves to Mersenne primes, since reduction modulo such a prime cannot increase Hamming weights.

Our first proposal using this structure [AJPS17] only allowed us to encrypt a single bit at a time. This inefficiency forced us to choose parameters that turned out to be vulnerable [BCGN17,

dBDJdW17]. In the present proposal, we describe a variation that permits us to encrypt many bits at a time. This allows in turn to choose much larger parameters which resist the attacks in [BCGN17] and [dBDJdW17], even in their Groverized quantum form. The best quantum attack mentioned in these articles has a complexity of the order 2^h where h is the Hamming weight we allow for low Hamming weight numbers. This explains our choice of h to be equal to the desired quantum security level.

Since it is well-known that cryptosystems of this type can be easily vulnerable to chosen-ciphertext attacks, it is extremely important to bind them together with a CCA-secure wrapper. We chose to present the system as a key encapsulation mechanism because this makes the design of the CCA wrapper very simple.

In addition, like many systems in this family, our system suffers from potential decryption failures. At the present time, we are unable to provide a tight rigorous analysis of the decryption error probability. In order to give a satisfactory bound (say below 2^{-128}), we would need either to enlarge the parameters of the scheme again or to replace the very simple repetition encoding that we are using by a more complex one. One very simple option would be to combine the repetition encoding with a random permutation of the bits of C_2 with are used to mask the encoded value at encryption time. This random permutation could be built from C_1 using the XOF provided by NIST. However, both methods would make the cryptosystem too slow and the second one would add an extra layer of complexity that is really undesirable.

Thus, we propose a heuristic analysis of the decryption error probability. This analysis is based on the distribution of the Hamming weights that are encountered in the decryption blocks corresponding to a single bit. Since, with our choice of parameters, every bit is encoded into $\rho = 2048$ bits, we want to see how often a bit might cross the Hamming weight 1024 boundary. It is easy to equip the code and count the Hamming weights encountered during decryption. We performed experiments involving 10000 of each key generation, encapsulation and decapsulation in order to collect the distribution shown in Figure 1. We see that the distribution looks like a superposition of two Gaussian distributions one corresponding to encryptions of a 0 and one to encryptions of a 1. Our heuristic assumption is that the probability of decryption failure is very close to the one corresponding to these Gaussian distributions. More precisely, we fitted a Gaussian G_0 corresponding to zeroes by searching for best fitting values of p and σ in:

$$G_0(x) = \frac{1}{2\sigma\sqrt{2\pi}} e^{-\frac{(x-p)^2}{2\sigma^2}}.$$

Note the extra 1/2 compared to a usual normal distribution. This is due to the fact that half of the encrypted bits are zeroes and half are ones. By symmetry, the Gaussian distribution corresponding to ones is simply $G_1(x) = G_0(\rho - x)$. We found that taking $p = 499.6$ and $\sigma = 28.64$ yields the very good approximation shown on Figure 2, where the two Gaussian are superposed with the measured data.

As a consequence, the probability that a single bit crosses the 1024 boundary is approximated by:

$$0.5 \operatorname{erfc}\left(\frac{1024 - p}{\sigma\sqrt{2}}\right) < 2^{-247}.$$

Since the encrypted value is formed of 256-bits, the overall probability of decryption failure during a full decapsulation can be heuristically upper bounded by 2^{-239}.

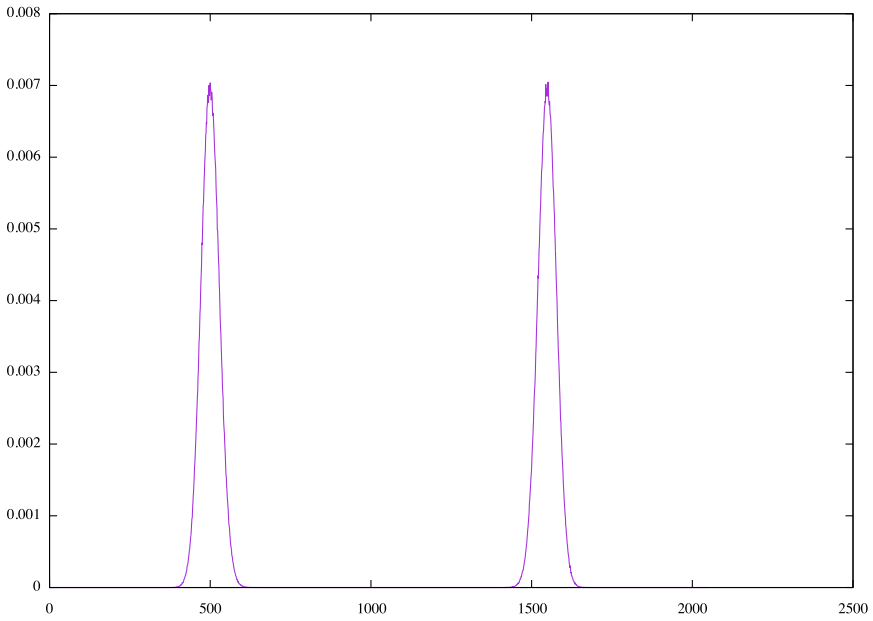

Figure 1: Density distribution of Hamming Weights during decryption

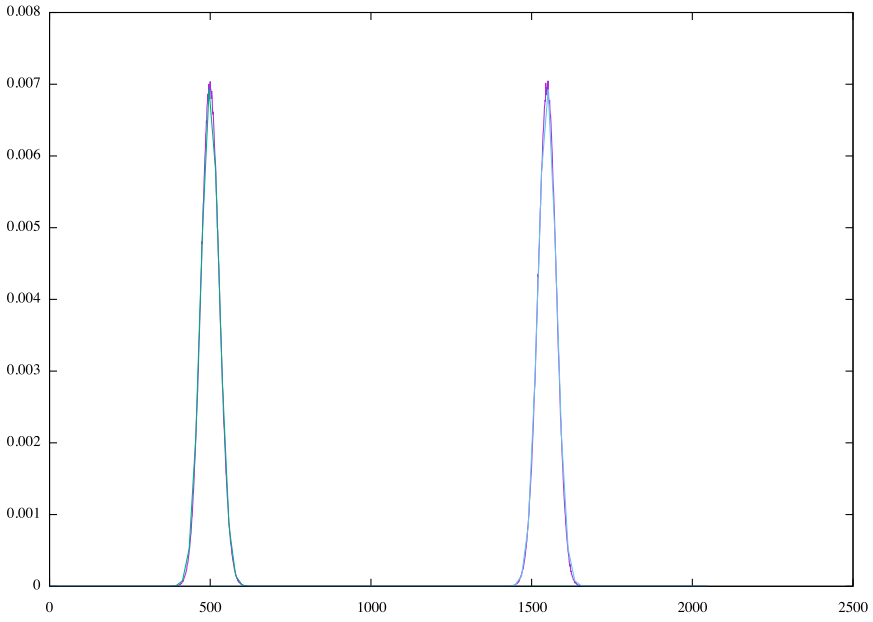

Figure 2: Density distribution with fitted Gaussians

5 Expected Security Strength

The best known classical cryptanalytic attack on our security assumption as discussed in the next section runs in time higher than 2^{2h}. With access to a quantum computer, one could use Grover's algorithm to obtain a quadratic speedup over this attack.

However, it would requite to perform lattice reduction at the botton of Grover's to implement this attack. This would certainly need very sophisticated universal quantum computers and it may well be infeasible for near term quantum devices. Yet, in view of this potential quantum attack and potential cryptananalytic improvements, we found preferable to take this attack into account. As a consequence, our cryptosystem can only be secure if we make sure that h is at least equal to the desired security level. For simplicity, we just set $h = \lambda$.

6 Known Cryptanalytic Attacks

As mentioned in the paper we provide as supporting documentation, the security of the cryptosystem relies on the following hardness assumption.

Definition 1. The *Mersenne Low Hamming Combination Assumption* states that, given an n-bit Mersenne prime $p = 2^n - 1$ and an integer h such that $4\,h^2 < n \leq 16\,h^2$, the advantage of any probabilistic polynomial time adversary running in time poly(n) in attempting to distinguish between

$$\left(\begin{bmatrix} R_1 \\ R_2 \end{bmatrix},\ \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} \cdot A + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}\right) \quad \text{and} \quad \left(\begin{bmatrix} R_1 \\ R_2 \end{bmatrix},\ \begin{bmatrix} R_3 \\ R_4 \end{bmatrix}\right)$$

is at most $O(2^{-h})$, where R_1, R_2, R_3, R_4 are independent and uniformly random n-bit strings, and A, B, are independently chosen n-bit strings each having Hamming weight h.

6.1 Attempts at Cryptanalysis

In this section, we mention the known approaches to break our security assumption and thereby mention the conjectured security guarantee for our scheme. For cryptanalysis, it is often more convenient to talk about search problems. We introduce the following search problem whose solution would imply an attack on our cryptosystem.

Definition 2 (Mersenne Low Hamming Combination Search Problem)**.** *For an n-bit Mersenne number $p = 2^n - 1$ and an integer h, given tuple $(R, FR + G \pmod p)$ where R is a uniformly random n-bit string and F, G have Hamming weight h, find F, G.*

For the remainder of the paper, we call this problem $\mathcal{P}$. It is easy to see that if one can efficiently solve the problem $\mathcal{P}$, then one can break the assumption in Definition 1, and hence the security of our cryptosystem. It is therefore important to study the hardness of this problem.

Hamming Distance Distribution. Let R be a uniformly random n bit string and $Y = FR+G$ where F, G are chosen uniformly at random from n bit strings with Hamming weight h. A basic test for the assumption that Y is pseudorandom given R is to check that the distribution of $\mathsf{Ham}(R, R')$ is close to the distribution of $\mathsf{Ham}(R, T)$ where T is a uniformly random n bit string.

If R is a fixed string and X is a uniformly random n bit string, the random variable $f_R(X) = \frac{\mathsf{Ham}(X,R)-n/2}{\sqrt{n/4}}$ is approximated by the standard normal random variable $N(0, 1)$. We generated R at random and then obtained samples $Y = FR + G$ where F, G are uniformly distributed over strings of Hamming weight $\sqrt{n}$. A quantile-quantile plot of $f_R(Y_i)$ against samples from $N(0, 1)$ is close to a straight line and does not show significant deviations from normality.

One could also perform more advanced statistical tests, such as the NIST suite [RSN+01] to verify the pseudorandomness of Y given R. However, in the context of cryptographic schemes, such tests only serve as sanity checks and it is preferable to focus on dedicated cryptanalysis.

Weak key attack. Following the appearance of a preliminary version of this paper, [BCGN17] found a weak key attack on the Mersenne Low Hamming Ratio search problem where given $H = \text{seq}(\frac{\text{int}(F)}{\text{int}(G)}) \mod P$ with F, G having low Hamming weight, the goal is to find F and G.

The weak key attack of [BCGN17] is based on rational reconstruction. If all the bits of F and G are in the right half of the bits, then both F and G are smaller than $\sqrt{P}$ and they can easily be recovered using a continued fraction expansion of H/P.

Generalization using LLL. The authors of the above weak attack also proposed in [BCGN17] a generalization based on guessing a decomposition of F and G into windows of bits such that in any window all the '1's are on the right. Using such a decompostion and replacing the use of continued fraction by LLL in relatively small dimension they can recover F and G from any compatible window decomposition.

A careful analysis of this method and its cost is presented in [dBDJdW17] and concludes that its running time is $2^{(2+\epsilon)h}$ for some small constant h. For simplicity, we assume that the cost of this attack is 2^{2h} on a classical computer.

Even if this attack was developped for the homogeneous Mersenne Low Hamming Ratio assumption, it is likely that it generalizes to the Mersenne Low Hamming Combination Assumption. We thus assume that it is the case. To the best of our knowledge, this is the most efficient known attack on our security assumption and the security parameters proposed in Section **??** have been revised to withstand it.

Quantum Speedup via Grover's Algorithm. With access to a quantum computer, one could use Grover's algorithm [Gro96] to obtain a quadratic speedup over the above attack.

Note that the attack performs a lattice reduction step for each guess of window decomposition and concludes that they are correct if the lattice reduction step succeeds. As a consequence, the Groverized version would requite to perform lattice reduction at the botton of Grover's to implement it. This would certainly need very sophisticated universal quantum computers and it may well be infeasible for near term quantum devices. However, in view of this potential quantum attack and potential cryptananalytic improvements, we take this attack into account. With this constraint, our cryptosystem can only be secure if we make sure that h is at least equal to the desired security level. For simplicity, we just set $h = \lambda$ and assume that the best possible attack on the Mersenne Low Hamming Combination problem has complexity at least 2^h to derive security estimates.

Meet in the middle attack. In [dBDJdW17], an efficient meet-in-the-middle attack that makes use of locality-sensitive hash functions is also mentioned. Its complexity is $O\left(\binom{n-1}{h-1}^{1/2}\right)$ on classical computers and $O\left(\binom{n-1}{h-1}^{1/3}\right)$ on quantum machines.

For our choice of parameters, this is much bigger than 2^h and thus doesn't affect the security level.

Attacking the system if n is not a prime. We mention here that it seems quite important to choose $2^n - 1$ to be a prime for our cryptosystem. There is at least a partial attack when n is not prime. Indeed if n_0 divides n, then $q = 2^{n_0} - 1$ divides $p = 2^n - 1$, and also F, G have Hamming weight at most h modulo q. Thus, given $Y = FR + G \mod q$, one can try to guess the secret key G modulo q, which can be done in $\sqrt{\binom{n_0}{h}}$ time using a quantum algorithm. This also reveals F modulo q and we can likely use it to guess F, G modulo p much faster than the attacks that work in the prime case.

7 Performance analysis

The performance of the system depends on the speed of two main components. The expandable hash function and the large number arithmetic. The code in our submission uses the XOF provided by NIST, which turns out to be an important limiting factor during keypair generation. This is due to the fact that in order to generate R we need to produce n bits of pseudo-random output. Roughly 3/4 of a Mbit of pseudo-randomness. This could be largely improved by using a different XOF, however, we understood that it was recommended to stay with the provided XOF. We would suggest to replace it by a faster function during the evalation period. The limiting factor for encryption is the speed of large integer multiplication. We simply used the GMP library for that purpose.

In order to help understanding the relative contributions of these two factors, Table 1 provides the amount of randomness needed and the number of multiplications required for every operation. It also gives timing on a MacOs X v10.11.6 laptop equiped with a Intel® Core^{TM} i7-4980HQ CPU at 2.80 GHz. These timings have been obtained by compiling the provided code with `gcc 6.1.0` and option `-Ofast`. Note that for optimized decapsulation, since the cost of the first decapsulation with a given key is the same at for a non-optimized decapsulation, we only provide the cost of subsequent calls.

	Keypair	Encaps	Decaps	Optimized
Pseudo-Random Kbytes	95	3	98	3
Multiplications	1	2	4	3
Time (ms)	5.3	7.2	16.2	10.5

Table 1: Count of critical operations and Timings on reference platform

The cryptosystem can be easily ported to any architecture that supports multiplication of large numbers. It would be interesting to study the extend to which hardware multipliers that were developed for RSA can be re-used with our system.

8 Advantages and Limitations

In this paper, we propose a simple new public-key encryption scheme. The only non trivial arithmetic operations that we require are multiplication and addition for bit strings. As a result, this scheme is easy to implement on platforms supporting a high precision arithmetic library like GMP. The GMP library is available across platforms and supports very efficient arithmetic operations, so it is also easy to implement optimized versions of our scheme.

As with other public-key cryptosystems, the security of our cryptosystem relies on an unproven assumption. The main disadvantage is that our assumption is quite new and requires more cryptanalytic effort before one can be reasonably confident about the security it provides.

References

[AJPS17] Divesh Aggarwal, Antoine Joux, Anupam Prakash, and Miklos Santha. A new public-key cryptosystem via mersenne numbers. *Cryptology ePrint Archive, Report 2017/481, version:20170530.072202*, 2017.

[BCGN17] Marc Beunardeau, Aisling Connolly, Rémi Géraud, and David Naccache. On the hardness of the Mersenne Low Hamming Ratio assumption. Technical report, Cryptology ePrint Archive, 2017/522, 2017.

[dBDJdW17] Koen de Boer, Léo Ducas, Stacey Jeffery, and Ronald de Wolf. Attacks on the ajps cryptosystem. (Personal communication of the preprint), November 2017.

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph Silverman. Ntru: A ring-based public key cryptosystem. *Algorithmic number theory*, pages 267–288, 1998.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo SLM Barreto. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 2069–2073. IEEE, 2013.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):Art. 34, 40, 2009.

[RSN+01] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, DTIC Document, 2001.