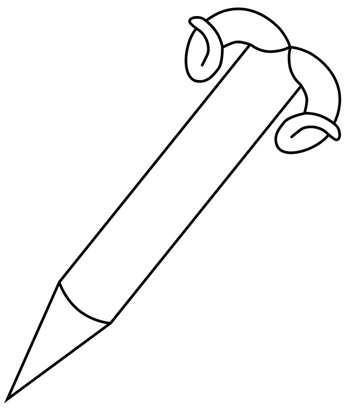
Ramstake

KEM Proposal for NIST PQC Project

November 30, 2017



cryptosystem name	Ramstake
principal submitter	Alan Szepieniec
	imec-COSIC KU Leuven
	`alan.szepieniec@esat.kuleuven.be`
	tel. +3216321953
	Kasteelpark Arenberg 10 bus 2452
	3001 Heverlee
	Belgium
auxiliary submitters	-
inventors / developers	same as principal submitter; relevant prior work is credited as appropriate
owner	same as principal submitter
backup contact info	`alan.szepieniec@gmail.com`
signature	

Contents

1. Introduction

The long-term security of confidential communication channels relies on their capacity to resist attacks by quantum computers. To this end, NIST envisions a transition away from public key cryptosystems that are known to fail in this scenario, and towards the so-called *post-quantum* cryptosystems. One of the functionalities in need of a post-quantum solution that is essential for securing online communication is *ephemeral key exchange*. This protocol enables two parties to agree on a shared secret key at a cost so insignificant as to allow immediate erasure of all secret key material after execution, as an additional security measure. In the case where the order of the messages need not be interchangeable, this functionality is beautifully captured by the *key encapsulation mechanism* (KEM) formalism of Cramer and Shoup [6]. The same formalism has the added benefit of capturing the syntax and security of the first part of IND-CCA-secure arbitrary-length hybrid encryption schemes, enabling a separation of the public key layer from the symmetric key layer.

The desirable properties of a post-quantum KEM are obvious upon consideration. It should be fast and it should generate short messages, not require too much memory and be implementable on a small area or in a few lines of code. It should inspire confidence by relying on long-standing hard problems or possibly even advertising a proof of security. However, this design document is predicated on the greater importance of a property not included in the previous enumeration: *simplicity.* The requirement for advanced degrees in mathematics on the part of the implementers presents a giant obstacle to mass adoption, whereas no such obstacle exists for mathematically straightforward schemes. More importantly, complexity has the potential to hide flaws and insecurities as they can only be exposed by experts in the field. In contrast, a public key scheme that is accessible to a larger audience is open to scrutiny from that same larger audience, and should therefore engender a greater confidence than a scheme that only a few experts were not able to break.

This document presents Ramstake, a post-quantum key encapsulation mechanism that excels in this category of simplicity. Aside from the well-established tools of hash functions, pseudorandom number generators, and error-correcting codes, Ramstake requires only high school mathematics. Though not optimized for message size and speed, Ramstake is still competitive in these categories with messages of less than one hundred kilobytes generated in a handful of milliseconds on a regular desktop computer at the highest security level. For security, Ramstake relies on a relatively new and under-studied hard problem, which requires several years of attention attention from the larger cryptographic community before it inspires confidence. The flipside of this drawback is the advantage associated with problem diversity: Ramstake is likely to remain immune to attacks that affect other branches of post-quantum cryptography.

Innovation. In a nutshell, this hard problem requires finding *sparse* solutions to linear equations modulo a large Mersenne prime, *i.e.* a prime of the form $p = 2^\pi - 1$. The binary expansions of the solution (x_1, x_2) consist overwhelmingly of zeros. Specifically, these integers can be described as

$$x_i = \sum_{j=1}^{w} 2^{e_j} \quad . \tag{1}$$

We refer to the integer's *Hamming weight* w as the number of ones; their positions e_j are generally chosen uniformly at random from $\{0, \ldots \pi - 1\}$. Ramstake's analogue of the discrete logarithm problem requires finding x_1 and x_2 of this form from G and $H = x_1 G + x_2 \bmod p$. This is an affine variant of the Low Hamming Weight Ratio problem of the Aggarwal *et al.* Mersenne prime cryptosystem [1], whose task is to obtain f and g of this form (1) from $H = fg^{-1} \bmod p$.

Where the Aggarwal *et al* cryptosystem builds on the indistinguishability of low Hamming weight ratios, Ramstake builds on a noisy Diffie-Hellman protocol [2, 3] instead. Alice and Bob agree on a random integer G between 0 and p. Alice chooses sparse integers x_1 and x_2 and sends $H = x_1G + x_2 \bmod p$ to Bob. Bob chooses sparse integers y_1 and y_2 and sends $F = y_1G + y_2 \bmod p$ to Alice. Alice computes $S_a = x_1F \bmod p$ and Bob computes $S_b = y_1G \bmod p$ and both integers approximate $S = x_1y_1G \bmod p$ in the following sense: since p is a Mersenne prime, reduction modulo p does not increase the integer's Hamming weight and as a result the differences $S_a - S = x_1y_2 \bmod p$ and $S_b - S = y_1x_2 \bmod p$ have a sparse binary expansion. Therefore, if x_1, x_2, y_1, y_2 have a sufficiently low Hamming weight, the binary expansions of S_a and S_b agree in most places. Alice and Bob have thus established a shared noisy secret stream of data, or since it will be used as a one-time pad, a *shared noisy one-time pad* (SNOTP, "snow-tipi").

From SNOTP to KEM. There are various constructions in the literature for obtaining KEMs from SNOTPs, each different in its own subtle way. The next couple of paragraphs give a high-level description of a generic transformation targeting IND-CCA security, which is inspired by the "encryption-based approach" of NewHope-Simple [4]. This construction makes abstraction of the underlying sparse integer mathematics.

The encapsulation algorithm is a deterministic algorithm taking a fixed-length random seed s as an explicit argument. If more randomness is needed than is contained in this seed, it is generated from a cryptographically secure pseudorandom number generator (CSPRNG). The algorithm outputs a ciphertext c and a symmetric key k.

The encapsulation algorithm uses an error-correcting code such as Reed-Solomon or BCH to encode the seed s into a larger bitstring. Then the ciphertext c consists of three parts: 1) a contribution to the noisy Diffie-Hellman protocol; 2) the encoding of the seed but one-time-padded with the encapsulator's view of the SNOTP; and 3) the hash of the seed. The decapsulation algorithm computes its own view of the SNOTP using the Diffie-Hellman contribution and undoes the one-time pad to obtain the encoding up to some errors. Under certain conditions, the error-correcting code is capable of retrieving the original seed s from this noisy codeword. At this point, the decapsulation algorithm runs the encapsulation algorithm with the exact same arguments, thus guaranteeing that the produced symmetric key k is the same for both parties. Robust IND-CCA security comes from the fact that the decapsulator can compare bit by bit the received ciphertext against the one that was recreated from the transmitted seed, in addition to verifying the seed's hash against the one that was part of the ciphertext.

2. Specification

2.1. Parameters

The generic description of the scheme refers the following parameters without reference to their value. Concrete values are given in Section 2.4.

- p — the Mersenne prime modulus, satisfies $p = 2^\pi - 1$;
- π — the number of bits in the binary expansion of p;
- w — the Hamming weight, which determine the number of ones in the binary expansion of secret sparse integers;
- ν — the number of codewords to encode the transmitted seed into;
- n — the length of a single codeword (in number of bytes);
- κ — the targeted security level (in log_2 of classical operations);
- λ — the length of seed values (in number of bits);
- χ — the length of the symmetric key (in number of bits).

2.2. Tools

2.2.1. Error-Correcting Codes

Ramstake relies on Reed-Solomon codes over $\mathrm{GF}(2^8)$ with designed distance $\delta = 224$ and dimension $k = 32$. Codewords are $n = 255$ field elements long and if there are 111 or fewer errors they can be corrected. With this choice of finite field, one field element coincides with one byte. The following subroutines are used abstractly:

- encode takes a string of $8k = 256$ bits and outputs a sequence of $8n$ bits that represents the Reed-Solomon encoding of the input.

- decode takes a string of $8n$ bits representing a noisy codeword and tries to decode it. If the codeword is decodable, this routine returns the error symbol $\perp$.

This abstract interface suffices for the description of the KEM. Moreover, any concrete instantiation can be exchanged for any other instantiation that adheres to the same interface, or that modifies the interface slightly to retain compatibility.

2.2.2. CSPRNG

Both key generation and encapsulation require a seed expander. All randomness can be generated up front; there is no need to record state and update it as pseudorandomness is generated. We use $\mathsf{xof}(s, \ell)$ to denote the invocation of the CSPRNG to generate a string of ℓ pseudorandom bytes from the seed s.

This abstract interface suffices for the description of the KEM. In the implementations, xof is instantiated with `SHAKE256`. Like in the case of the Reed-Solomon codec, any concrete instantiation can be exchanged for any other instantiation that adheres to the same interface.

2.3. Description

2.3.1. Serialization of Integers

All big integers represent elements in $\{0, \ldots, p-1\}$ and are therefore fully defined by π bits. Denote by $\mathsf{serialize}(a)$ the array if $\lceil \frac{\pi}{8} \rceil$ bytes satisfying

$$a = \sum_{i=0}^{\lceil \frac{\pi}{8} \rceil - 1} \mathsf{serialize}(a)[i] \times 256^i \ . \tag{2}$$

This serialization puts the least significant byte first and pads the array with zeros to meet the given length if the integer is not large enough. It is essentially Little-Endian padded to length $\lceil \frac{\pi}{8} \rceil$, and corresponds with the GMP function `mpz_export`($\cdot$, NULL, -1, 1, 1, 0, a) regardless of whether the integer a is large enough.

2.3.2. Data Structures

Ramstake uses five data structures: a random seed, a secret key, a public key, a ciphertext, and a symmetric key. Random seeds are bitstrings of length λ, whereas symmetric keys are bitstrings of length χ. The other three data structures are more involved.

Secret key. A secret key consists of the following items:

- `seed` — a random seed which fully determines the rest of the secret key in addition to the public key;
- a, b — sparse integers, represented by π bits each.

Public key. A public key consists of the following items:

- `g_seed` – a random seed which is used to generate the random integer G;
- C — integer between 0 and p which represents a noisy Diffie-Hellman contribution. This value satisfies $C = aG + b \bmod p$.

Ciphertext. A ciphertext consists of the following items:

- D — integer between 0 and p which represents a noisy Diffie-Hellman contribution; this value satisfies $D = a'G + b' \bmod p$ where a', b' are secret sparse integers sampled by the encapsulator;
- `seedenc` — string of $8n\nu$ bits; this value is the bitwise xor of the binary expansion of the first $n\nu$ bytes of $\mathsf{serialize}(S)$ and the sequence of ν times $\mathsf{encode}(s)$, where s is the random seed that is the argument to the encapsulation algorithm, and where S is the encapsulator's view of the SNOTP: $S = a'(aG + b) \bmod p$.
- h — hash of the seed s; the purpose of this value is twofold: 1) to speed up decapsulation by enabling the decoder to recognize correct decodings, and 2) to anticipate a proof technique in which the simulator answers decapsulation queries by finding this value's inverse.

These objects are serialized by appending the serializations of their member items in the order presented above. No length information is necessary as the size of each object is a function of the parameters. We overload *serialize* to denote that operation.

In this notation, the symmetric key $k \in \{0,1\}^\chi$ satisfies $k = \mathsf{H}(\mathsf{serialize}(pk) \| coins)$, where pk is the public key and where $coins$ is the byte string of random coins used by the encapsulator. Ramstake instantiates H with SHA3-256 with output truncated to χ bits, but any other secure hash function suffices.

2.3.3. Algorithms

A KEM consists of three algorithms, KeyGen, Encaps, and Decaps. Pseudocode for Ramstake's three algorithms is presented in Algorithms 3, 4, and 5. All three functionalities obtain a pseudorandom integer G from a short seed; this subprocedure is called generate_g and is shown in Algorithm 1. Algorithms KeyGen and Encaps rely on a common subroutine called sample_sparse_integer which deterministically samples a sparse integer given enough random bytes and a target Hamming weight, and which is described in Algorithm 2.

```
algorithm generate_g
input: seed $\in \{0,1\}^\lambda$ — random seed
output: $g \in \{0, \ldots, p-1\}$ — pseudorandom integer

1: r $\leftarrow$ xof(seed, $\lfloor \frac{\pi}{8} \rfloor + 2$)
2: $g \leftarrow 0$
3: for $i$ from $0$ to $\lfloor \frac{\pi}{8} \rfloor + 1\}$ do:
4:     $g \leftarrow 256 \times g +$ r[$i$]
5: end
6: return $g \bmod p$
```

Algorithm 1: Procedure to sample a random integer from $\{0, \ldots, p-1\}$.

```
algorithm sample_sparse_integer
input: r ∈ $\{0,\ldots,255\}^{4\times\texttt{weight}}$ — enough random bytes
       weight ∈ $\{0,\ldots,\pi\}$ — number of one bits
output: $a \in \{0,\ldots,p-1\}$ — a sparse integer

1: $a \leftarrow 0$
2: for $i$ from 0 to $\texttt{weight}-1$ do:
3:     $u \leftarrow (\texttt{r}[4i] \times 256^3 + \texttt{r}[4i+1] \times 256^2 + \texttt{r}[4i+2] \times 256 + \texttt{r}[4i+1]) \bmod \pi$
4:     $a \leftarrow a + 2^u$
5: end
6: return $a$
```

Algorithm 2: Procedure to sample a sparse integer from a CSPRNG.

```
algorithm KeyGen
input: seed ∈ $\{0,1\}^\lambda$ — random seed
output: $sk$ — secret key
        $pk$ – public key

    ▷ expand randomness
1: $\texttt{r} \leftarrow \mathsf{xof}(\texttt{seed}, 4 \times w + 4 \times w + \lambda/8)$
    ▷ grab seed for $G$ and generate $G$
2: $\texttt{seed\_g} \leftarrow \texttt{r}[0 : (\lambda/8)]$
3: $G \leftarrow \mathsf{generate\_g}(\texttt{seed\_g})$
    ▷ get sparse integers $a$ and $b$
4: $a \leftarrow \mathsf{sample\_sparse\_integer}(\texttt{r}[(\lambda/8) : (\lambda/8 + 4 \times w)], w)$
5: $b \leftarrow \mathsf{sample\_sparse\_integer}(\texttt{r}[(\lambda/8 + 4 \times w) : (\lambda/8 + 4 \times w + 4 \times w)], w)$
    ▷ compute Diffie-Hellman contribution
6: $C \leftarrow aG + b \bmod p$
7: return $sk = (s, a, b)$, $pk = (\texttt{g\_seed}, C)$
```

Algorithm 3: Generate a secret and public key pair.

```
algorithm Encaps
input: seed $\in \{0,1\}^\lambda$ — random seed
       $pk$ — public key
output: $ctxt$ — ciphertext
        $k \in \{0,1\}^\chi$ – symmetric key

    ▷ extract randomness and generate $G$ from seed
 1: r $\leftarrow$ xof(seed, $4 \times w + 4 \times w$)
 2: $G \leftarrow$ generate_g($pk$.seed_g)
    ▷ sample sparse integers
 3: $a' \leftarrow$ sample_sparse_integer(r[$0 : (4 \times w)$], $w$)
 4: $b' \leftarrow$ sample_sparse_integer(r[$(4 \times w) : (4 \times w + 4 \times w)$], $w$)
    ▷ compute Diffie-Hellman contribution and SNOTP
 5: $D \leftarrow a'G + b' \bmod p$
 6: $S \leftarrow a'\, pk.C \bmod p$
    ▷ encode random seed and apply SNOTP
 7: seedenc $\leftarrow$ serialize($S$)[$0 : (n\nu)$]
 8: for $i$ from $0$ to $\nu - 1$ do:
 9:     seedenc[$(in) : ((i+1)n)$] $\leftarrow$ seedenc[$(in) : ((i+1)n)$] $\oplus$ encode(seed)
10: end
    ▷ compute symmetric key
11: $k \leftarrow$ H(serialize($pk$)$\|$r)
    ▷ complete ciphertext; and return ciphertext and symmetric key
12: $h \leftarrow$ H(seed)
13: return $ctxt = (D, \text{seedenc}, h), k$
```

Algorithm 4: Encapsulate: generate a ciphertext and a symmetric key.

```
algorithm Decaps
input: $ctxt = (D, \texttt{seedenc}, h)$ — ciphertext
       $sk = (\texttt{seed}, a, b)$ — secret key
output: $k$ — symmetric key on success; otherwise $\perp$

    ▷ recreate public key from secret key seed
 1: $\texttt{seed\_g} \leftarrow \mathsf{xof}(sk.\texttt{seed}, \lambda/8)$
 2: $G \leftarrow \mathsf{generate\_g}(\texttt{seed\_g})$
 3: $C \leftarrow \texttt{sk}.a\, G + \texttt{sk}.b \bmod p$
    ▷ obtain SNOTP and decode seedenc
 4: $S' \leftarrow \texttt{sk}.a\, \texttt{ctxt}.D \bmod p$
 5: $\texttt{str} \leftarrow \mathsf{serialize}(S')[0 : (n\nu)] \oplus ctxt.\texttt{seedenc}$
 6: for $i$ from $0$ to $\nu - 1$ do:
 7:     $s \leftarrow \mathsf{decode}(\texttt{str}[(in) : ((i+1)n)])$
 8:     if $s \neq \perp$ and $\mathsf{H}(s) = ctxt.h$ then:
 9:         break
10:     end
11: end
12: if $s = \perp$ then:
13:     return $\perp$
14: end
    ▷ recreate and test ciphertext
    $ctxt', k \leftarrow \mathsf{Enc}(s, \texttt{pk} = (\texttt{g\_seed}, C))$
15: if $ctxt \neq ctxt'$ do:
16:     return $\perp$
17: end
18: return $k$
```

Algorithm 5: Decapsulate: generate symmetric key and test validity of the given ciphertext.

2.4. Parameter Sets

This document proposes two sets of parameters, called "Ramstake RS 216091", "Ramstake RS 756839". These parameter sets target security levels 128 and 256 in terms of $\log_2$ of required number of operations to mount a successful attack on a classical computer. Both attacks considered in Section 4.3 are fully Groverizable, thus enabling the quantum adversary to divide these target security levels by two. All parameter sets use SHA3-256, SHAKE256, and Reed-Solomon error correction over $\mathbb{F}_{2^8}$ with code length $n = 255$ and design distance $\delta = 224$.

Table 1: Ramstake parameter sets, resulting public key and ciphertext size in kilobytes, and targeted security notion and NIST security level.

π	216091	756839
w	64	128
ν	4	6
λ	256	256
χ	256	256
$\lvert pk \rvert$	26.41 kB	92.42 kB
$\lvert ctxt \rvert$	27.41 kB	93.91 kB
security	IND-CCA	IND-CCA
NIST level	1	5

3. Performance

3.1. Failure Probability

There is a nonzero probability of decapsulation failure even without malicious activity. This event occurs when the two views of the SNOTP are too different, requiring the correction of too many errors. It is possible to find an exact expression for this probability. However, the following argument opts for a more pragmatic approach.

The Reed-Solomon code used has design distance $\delta = 224$, meaning that it can correct up to $t = \lfloor \frac{\delta-1}{2} \rfloor = 111$ byte errors. Decapsulation fails when all ν codewords contain more than 111 errors. By treating the number of errors e in each codeword as independent normally distributed variables, one can obtain a reasonable estimate of the failure probability.

The Sage script `Scripts/parameters.sage`, which is included in the submission package, computes the mean (μ) and standard deviation (σ) of these distributions empirically. For many different random G and appropriately sparse a, b, a', b', the number of different bytes between $\mathsf{serialize}(aa'G + ba' \bmod p)[0 : 255]$ and $\mathsf{serialize}(aa'G + b'a \bmod p)[0 : 255]$ is computed. From many such trials it computes μ and σ and a recommended number of codewords ν such that the failure probability drops below 2^{-64}. (Indeed, this script is where the values for ν in the parameter sets of Table 2.4 come from.) The statistics are shown in Table 2.

It is possible to push the failure probability even lower by increasing ν. However, this increase results in a larger ciphertext.

Table 2: Mean μ and standard deviation σ of number of errors in a codeword, along with recommended number of codewords ν for a failure probability less than 2^{-64}.

	216091	756839
μ	72.56	81.38
σ	7.89	7.93
ν	4	6
$\left(1 - \Phi(\frac{e-\mu}{\sigma})\right)^{\nu}$	$\leq 2^{-64}$	$\leq 2^{-64}$

3.2. Complexity

3.2.1. Asymptotic

The loops in the pseudocode of Algorithms 1—5 run through a number of iterations determined by the parameters ν, w, π. Of these parameters, ν is independent of the security parameter κ. The relations between w, π and the security parameter κ are more complex. First π must be large enough to spread out roughly $2w^2$ burst-errors so as to guarantee a low enough byte-error-rate and hence non-failure. Second, the slice-and-dice attack of Section 4.3 must be taken into account as well. These parameters are constrained for non-failure by

$$\frac{2w^2}{\pi} \leq c \ , \tag{3}$$

for some constant c roughly around 0.04. For security, the constraint is

$$2w \geq \kappa \ . \tag{4}$$

These equations thus require $\pi \sim \kappa^2$. The size of the public key and ciphertext grows linearly with this number.

While KeyGen, Encaps and Decaps contain only a small fixed number of big field operations, the modulus of this field is p and the field elements involved therefore have an expansion of up to π bits. Nevertheless, there are two available optimizations to ameliorate this cost. (However, none of the provided implementations employ them.)

- Mersenne form. Reduction modulo p does not require costly division as it does for generic moduli. Instead, shifting and adding does the trick. Let $a = a_o \times p + a_r$ with $a_r < p$. Then $a_r + a_o = a \bmod p$.

- Sparsity. In every big field operation, at least one term or factor is sparse. As a result, the sums can be computed through w localized bitflips with carry. The products can be computed through w shifts and as many full additions.

Consequently, the cost of integer arithmetic is linear π and in w. Therefore, the complexity of all three algorithms is $O(\kappa^3)$.

3.2.2. Pratice

The file `perform.c`, which is included in the submission package, runs a number of trials and collects timing and cycle count information. Table 3 presents the information collected from the optimized implementations during 10 000 trials on a Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz machine with 6144 kB of cache on each of its four cores, with 7741 MB of RAM, and running CentOS linux.

Table 3: Implementation statistics — time and cycle count.

	time (ms)	cycles
Ramstake RS 216091		
KeyGen	2.8	9445009
Encaps	5.4	17700978
Decaps	11.1	36706919
Total	19.3	63852906
Ramstake RS 756839		
KeyGen	13.0	43148424
Encaps	24.1	79342014
Decaps	46.9	154721609
Total	84.1	277212047

It is not surprising that Decaps takes the longest, because it runs Encaps as a subprocedure. The striking difference between Encaps and KeyGen is due to the encoding procedure of the error correcting code. Dealing with this error-correcting code is even more costly in Decaps where the errors are corrected.

3.2.3. Memory and Pseudorandomness

It is difficult to estimate the memory requirements of the error-correcting code algebra as well of the big integer arithmetic for two reasons. 1) The current implementation outsources this operation to another library. 2) because this content is highly dynamic: how much memory is needed depends on the value of the mathematical object being represented. By contrast, the memory requirements of the three main functionalities' outputs is easily determined.

The secret key consists of one $\lambda/8$ byte seed and two integers of (after serialization) $\lceil \pi/8 \rceil$ bytes each, although the integers can be generated anew from the seed. The public key contains one seed of $\lambda/8$ bytes and one integer of $\lceil \pi/8 \rceil$ bytes. The ciphertext consists of one integer of $\lceil \pi/8 \rceil$ bytes, a stream of $n\nu$ bytes representing the one-time-padded repetition code, and a hash of $\chi/8$ bytes. Table 4 summarizes these sizes and presents concrete values for the given parameter sets.

All pseudorandomness is generated (*i.e.* extracted from a short seed) in the first line of those functions that need it. So this is $8w + \lambda/8$ for KeyGen, and $8w$ for Encaps. The Decaps function does not require pseudorandomness but it must get the $\lambda/8$-byte seed for G from the secret key seed via the same CSPRNG. Since Decaps invokes Encaps as a subprocedure, it inherits those requirements for extracting and storing pseudorandomness also.

Table 4: Size (in bytes) of output objects.

	secret key	public key	ciphertext
formula	$\lambda/8 + 2\lceil\pi/8\rceil$	$\lambda/8 + \lceil\pi/8\rceil$	$\lceil\pi/8\rceil + n\nu + \chi/8$
Ramstake 216019	54056	27044	28064
Ramstake 756839	189242	94637	96111

4. Security

4.1. Hard Problems

Ramstake relies on the hardness of at least two problems related to finding sparse solutions to affine equations modulo a pseudo-Mersenne prime p. The formal problem statement of the first is as follows.

Low Hamming Combination (LHC) Problem.
Given: Two coefficients $A, B \in \mathbb{F}_p$ in a large Mersenne prime field $\mathbb{F}_p$.
Task: Find two elements $x_1, x_2 \in \mathbb{F}_p$ with binary expansions of Hamming weight at most w_1 and w_2 respectively, such that $B = Ax_1 + x_2 \bmod p$.

The problem was implicitly introduced by Aggarwal *et al.* [1] in the form of an assumption, which states that the distribution $(A, Ax_1 + x_2)$ is indistinguishable from (A, C) when C is drawn uniformly at random and x_1, x_2 uniformly at random subject to having the required Hamming weight. The same paper also introduces the Low Hamming Ratio Search (LHRS) Problem, which asks to find a pair of low Hamming weight integers x_1, x_2 satisfying $x_2/x_1 = H$. The LHRS Problem is equivalent to the subset of the LHC Problem where $B = 0$. (To see this, set $H = -A$. $\square$)

The LHC problem is only the analogue of the discrete logarithm problem in Diffie-Hellman key agreement. The adversary does not need to compute discrete logarithms; he merely needs to break the Diffie-Hellman problem, which comes in search and decisional variants. The analogues of these problems for sparse integers is formally stated below.

Low Hamming Diffie-Hellman Search (LHDHS) Problem.
Given: Three integers (G, H, F) where $H = x_1G + x_2 \bmod p$ and $F = y_1G + y_2 \bmod p$ for some integers x_1, y_1 of Hamming weight w_1 and x_2, y_2 of Hamming weight w_2.
Task: Find an integer S whose Hamming distance with $x_1F \bmod p$ is at most t, and whose Hamming distance with $y_1H \bmod p$ is also at most t.

Low Hamming Diffie-Hellman Decision (LHDHD) Problem.
Given: Four integers (G, H, F, S) where $H = x_1G + x_2 \bmod p$ and $F = y_1G + y_2 \bmod p$ for some integers x_1, y_1 of Hamming weight w_1 and x_2, y_2 of Hamming weight w_2.
Task: Decide whether or not the Hamming distances between S and $x_1F \bmod p$, and between S and $y_1H \bmod p$, are at most t.

Security requires these problems to be hard, meaning that all polynomial-time quantum adversaries decide the LHDHD Problem with a success probability negligibly far from that of a random guess. The assumed hardness of LHDHD implies that LHDHS is hard as well, which in turn implies that LHC is hard also. It is unclear how to solve LHDHD in a way that avoids implicitly solving LHC.

It is clear that breaking LHDHS is enough to break the scheme, as that allows the attacker to unpad the seed encoding and recover the seed from there. It is not clear whether security also relies on the LHDHD problem but we include that problem for the sake of completeness, because many Diffie-Hellman type cryptosystems rely on the proper analogue of the Decisional Diffie-Hellman problem.

4.2. SNOTP-to-KEM Construction

There is a gap between the Low Hamming Diffie-Hellman Decision Problem and the IND-CCA (or even IND-CPA) security of Ramstake, originating from the SNOTP-to-KEM construction. I am working on a proof of security but it is unavailable at this point. The following obstacles make such a proof highly non-trivial.

- Failure events in the noisy Diffie-Hellman protocol affect security, especially in the chosen ciphertext model.
- The search problems may be solved in more than one way.
- Circular encryption: the one-time pad is not independent of the message it hides.
- The hash functions should be modeled as quantum-accessible random oracles. However, many classical proof techniques fail in the quantum random oracle model.

It is conceivable that a security proof can only be made to work conditioned on some changes being made to the construction, for instance by changing the inputs to the hash functions. Nevertheless, I do not expect the proof to recommend big changes, thus leaving the construction's big picture intact:

- generate noisy Diffie-Hellman protocol contributions from a short random seed;
- use the noisy Diffie-Hellman key to one-time-pad the error-correcting encoding of the seed;
- undo the noisy one-time pad and decode the codeword;
- invoke the encapsulation algorithm with identical arguments and test if the generated ciphertext matches the received one exactly.

4.3. Attacks

4.3.1. Slice and Dice

Beunardeau *et al.* present an attack exploiting the sparsity of the solutions to the LHRS Problem [5], but it applies equally to the LHC Problem. The attack proceeds as follows.

For each trial, partition the range $R = \{0, \ldots, \pi - 1\}$ into a number of subranges. This number should not be too large, at most a couple hundred. Do this once for x_1 and once for x_2. This yields

$$R_1^{(0)} \sqcup \cdots \sqcup R_1^{(k-1)} = R_2^{(0)} \sqcup \cdots \sqcup R_2^{(\ell-1)} = R \ . \tag{5}$$

Set each such subrange to active or inactive at random. Ensure that the total cardinality of all inactive subranges is at least π.

Each subrange corresponds to a variable $r_i^{(j)}$ whose binary expansion matches that of x_i but restricted to that subrange. Formulaically, this means

$$x_i = \sum_{j=0}^{k-1} 2^{\mathsf{min}(R_i^{(j)})} r_i^{(j)} \quad \text{and} \quad 0 \leq r_i^{(j)} < 2^{\# R_i^{(j)}} \ . \tag{6}$$

At this point, trim the sums in the left side of Eqn. 6 by dropping the terms that correspond to inactive subranges and replace x_1 and x_2 by their corresponding trimmed sums in the equation $B = Ax_1 + x_2 \bmod p$. Use LLL to find a short solution vector.

A single trial is successful if LLL succeeds in finding the solution that corresponds to the sparse solution. This happens if the guess at inactive subranges is correct, namely if their respective variables are indeed zero (because then their omission does not change the value of the sum).

For the sake of generality, assume x_1 has Hamming weight w_1 and x_2 has Hamming weight w_2. The optimal attacker activates a proportion $\frac{w_1}{w_1+w_2}$ of the range associated to x_1, and a proportion $\frac{w_2}{w_1+w_2}$ of the range associated to x_2. Then the probability of all 1-bits being located inside the active subranges is given by

$$P = \left(\frac{w_1}{w_1 + w_2}\right)^{w_2} \times \left(\frac{w_2}{w_1 + w_2}\right)^{w_1} \ . \tag{7}$$

The formula is a lot simpler when $w_1 = w_2 = w$, and in this case security mandates that

$$2w \geq \kappa \ . \tag{8}$$

This algorithm is fully Groverizable. Therefore, the security level halves when considering quantum adversaries with unlimited circuit depth.

4.3.2. Spray and Pray

Spray and pray is essentially a smart brute force search. Choose a random assignment for x_1 with Hamming weight w_1, compute x_2 from the given information and test if its Hamming weight is at most w_2. Assuming the solution is unique, the success probability of a single trial is one over the size of the search space, or $1/\binom{\pi}{w}$. So κ bits of security requires

$$\mathsf{log}_2 \binom{\pi}{w} \geq \kappa \ . \tag{9}$$

For the parameter sets 216091 and 756839, the left-hand-side of Eqn. 9 is over 838 and 1783, respectively. While the algorithm is fully Groverizable, dividing these numbers by two in order to account for quantum adversaries still results in wildly infeasible complexity.

4.3.3. Stupid Brute Force

Instead of guessing one variable and computing the other from that guess, stupid brute force guesses both at once. A single such guess succeeds with probability $1/\binom{\pi}{w}^2$, *i.e.*, much less likely than the intelligent brute force of the spray-and-pray strategy described above.

Another stupid brute force attack attempts to guess the input of the CSPRNG. By design, these seeds are all 256 bits in length, making for a classical complexity of 2^{256} and 2^{128} quantumly (again assuming unlimited depth).

4.3.4. Lattice Reduction

Aggarwal *et al.* already consider lattice attacks on their cryptosystem and in particular on the LHRS Problem. They observe that it is possible to generate basis vectors for a lattice in which the sought after solution is a short vector. However, that same lattice will contain even shorter vectors that do not correspond to a sparse solution to the original problem. It might be possible to eliminate these parasitical solutions by running lattice reduction with respect to the infinity norm instead of the Euclidean norm, but it is not clear how to do this.

4.3.5. Algebraic System Solving

It is possible in theory to formulate the sparsity constraint algebraically, by constructing polynomials over $\mathbb{F}_p$ that evaluate to zero in all points that satisfy the constraint. At this point a Gröbner basis algorithm can be used to compute a solution. However, the degree of this constraint polynomial is infeasibly large, roughly $\binom{\pi}{w}$. Constructing it requires more work than exhaustively enumerating all potential solutions and testing to see if the linear equations are satisfied.

Another option is to treat the coefficients of the binary expansion of the solutions, as variables in and of themselves. This strategy requires adding polynomials to require that each coefficient lie in $\{0, 1\}$, and that at most w of them are different from zero. The result is a nonlinear system of roughly $4\pi + 2\binom{\pi}{w+1}$ equations in 2π variables with some polynomials having degree $\binom{\pi}{w+1}$. For any practical parameter set, it is infeasible to fully represent this system of equations, let alone to solve it.

4.3.6. Error Triggering

An attacker who can query the decapsulation oracle can obtain feedback on whether the decapsulator was able to decode the transmitted codeword. With enough failures, the attacker can infer the decapsulator's view of the SNOTP. Once the attacker is in possession of this value, he can proceed to decapsulate any ciphertext.

However, in order to exploit this channel of information, the attacker must generate ciphertexts that fail during decapsulation. If his query ciphertext is not the exact output of the encapsulation algorithm upon invocation with the transmitted seed, then the manipulation will trigger a decapsulation failure regardless of whether decoding was successful. In other words, in order to obtain meaningful information about failure events, the attacker must restrict himself to querying only legitimate outputs of Encaps. Worse still, he has no way of knowing beforehand whether or not a ciphertext is more or less likely to cause failure before the first failure response. Since the failure probability is less than 2^{-64}, the attacker has to make on the order 2^{64} honest queries to get this first failure response.

5. Advantages and Limitations

Advantage: Simplicity. Simplicity is the key selling point of Ramstake. Simple schemes are easier to implement, easier to debug, and easier to analyze. While simpler schemes are sometimes also easier to break, a scheme's resilience to attacks should not rely on its complexity.

Advantage: Problem Diversity. Ramstake relies on different hard problems compared other branches of post-quantum cryptography. Consequently, breakthroughs in cryptanalysis or hard problem solving that break or severely harm other schemes may leave Ramstake intact.

Limitation: New Hard Problem. The hard problem on which Ramstake relies is new and understudied. As a result, it does not offer much assurance of security compared to schemes that have existed (and remained unbroken) for much longer.

Limitation: No Proof. Ramstake claims to offer IND-CCA security even though there is no security reduction to the underlying hard problem. It is therefore conceivable that an attack might break the scheme even without solving the hard problem. Nevertheless, simply because something has not been proven secure yet does not mean it is insecure.

Limitation: Bandwidth and Speed. Lattice-based KEMs are likely to be faster and to require less bandwidth. Nevertheless, Ramstake is competitive in comparison to the very first lattice-based and code-based cryptosystems, and it is conceivable that sparse integer cryptosystems will undergo a similar evolution. However, potential future improvements should not be considered for standardization at this point.

Acknowledgments

The author is thankful to Fré Vercauteren, Reza Reyhanitabar and Ward Beullens for useful discussions and insights. Also, the feedback from NIST after the September deadline was highly useful and highly appreciated. The author is being supported by a Ph.D. Fellowship from the Institute for the Promotion of Innovation through Science and Technology in Flanders (VLAIO, formerly IWT).

References

[1] Aggarwal, D., Joux, A., Prakash, A., Santha, M.: A new public-key cryptosystem via mersenne numbers. IACR Cryptology ePrint Archive 2017, 481 (2017), http://eprint.iacr.org/2017/481

[2] Aguilar, C., Gaborit, P., Lacharme, P., Schrek, J., Zémor, G.: Noisy diffie-hellman protocols (2010), https://pqc2010.cased.de/rr/03.pdf, PQCrypto 2010 The Third International Workshop on Post-Quantum Cryptography (recent results session)

[3] Aguilar, C., Gaborit, P., Lacharme, P., Schrek, J., Zémor, G.: Noisy diffie-hellman protocols or code-based key exchanged and encryption without masking (2010), https://rump2010.cr.yp.to/fae8cd826597867589335232978 6cea2.pdf, CRYPTO 2010 (rump session)

[4] Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Newhope without reconciliation. IACR Cryptology ePrint Archive 2016, 1157 (2016), http://eprint.iacr.org/2016/1157

[5] Beunardeau, M., Connolly, A., Géraud, R., Naccache, D.: On the hardness of the mersenne low hamming ratio assumption. IACR Cryptology ePrint Archive 2017, 522 (2017), http://eprint.iacr.org/2017/522

[6] Cramer, R., Shoup, V.: Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. IACR Cryptology ePrint Archive 2001, 108 (2001), http://eprint.iacr.org/2001/108

A. IP Statement

A.1. Statement by Submitter

I, Alan Szepieniec, of Kasteelpark Arenberg 10 / 3001 Heverlee / Belgium , do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as Ramstake, is my own original work, ~~or if submitted jointly with others, is the original work of the joint submitters~~.

I further declare that (check one):

✓ 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 Ramstake; OR (check one or both of the following):

☐ to the best of my knowledge, the practice of the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as Ramstake, may be covered by the following U.S. and/or foreign patents: "none";

☐ I do hereby declare that, to the best of my knowledge, the following pending U.S. and/or foreign patent applications may cover the practice of my submitted cryptosystem, reference implementation or optimized implementations: "none".

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:	Alan Szepieniec
Title:	ir.
Date:	
Place:	

A.2. Statement By Implementation Owner

I, Alan Szepieniec, Kasteelpark Arenberg 10 / 3001 Heverlee / Belgium, am the owner ~~or authorized representative of the owner (print full name, if different than the signer)~~ 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:	Alan Szepieniec
Title:	ir.
Date:	
Place:	