From: Alperin-Sheriff, Jacob (Fed)
Sent: Thursday, December 28, 2017 5:17 PM
To: pqc-comments
Cc: pqc-forum@list.nist.gov; jsno@snu.ac.kr; leewj422@ccl.snu.ac.kr; mypurist@gmail.com; Yongwoo Lee
Subject: OFFICIAL COMMENT: pqsigRM

So I've been doing some basic testing via adding cpucycles calls to the PQCgenKAT_*.c files, just to see if the numbers I get are somewhere in the ballpark of what was in specifications (note that I've indeed gotten numbers somewhere in the ballpark for several so it doesn't seem to be an issue with my code).

For pqsigrm (specifically pqsigRM-4-12, I haven't checked the others), I am getting on the order of thousands of times as many cycles for each of key generation, signing and verification as what the pqsigRM team gave in their supporting documentation.

It's quite possible that the submitters meant thousands of cycles instead of total cycles, but I don't see that anywhere. If not, I'd like the discrepancy to be explained. Thanks.

(For reference, the definition of cpucycles)

```
long long cpucycles(void) {
  unsigned long long result;
  __asm__ volatile(".byte 15;.byte 49;shlq $32,%%rdx;orq %%rdx,%%rax" : "=a" (result) ::  "%rdx");
  return result;
}
```

—Jacob Alperin-Sheriff

From: Yongwoo Lee <yongwool@ccl.snu.ac.kr>
Sent: Monday, January 01, 2018 10:31 AM
To: Alperin-Sheriff, Jacob (Fed); pqc-comments
Cc: pqc-forum@list.nist.gov; jsno@snu.ac.kr; leewj422@ccl.snu.ac.kr; mypurist@gmail.com
Subject: RE: OFFICIAL COMMENT: pqsigRM

Dear Dr. Alperin-Sheriff

We appreciate for your comments.

We measured the CPU clocks but mistakenly wrote 'cycles' instead of 'clocks'.
We measured the cpu cycles again using the function you sent.
The new measurement is reflected in the table below.

	security	key generation	singing	verification
pqsigRM-4-12	128	14639777783	3971208456	139814898
pqsigRM-6-12	196	6395769782	3275234719	198607502
pqsigRM-6-13	256	72162115384	1087667252	956410761

In addition, We are constantly updating the program, you can always check the latest version of our submission on the website below:
: https://sites.google.com/view/pqsigrm

We will reflect your comments in our documentation and update it soon.

Happy new year!
Jong-Seon No,
Wijik Lee
Young-Sik Kim
Yong-Woo Lee

From: Alperin-Sheriff, Jacob (Fed) [mailto:jacob.alperin-sheriff@nist.gov]
Sent: Friday, December 29, 2017 7:17 AM
To: pqc-comments <pqc-comments@nist.gov>
Cc: pqc-forum@list.nist.gov; jsno@snu.ac.kr; leewj422@ccl.snu.ac.kr; mypurist@gmail.com; Yongwoo Lee <yongwool@ccl.snu.ac.kr>
Subject: OFFICIAL COMMENT: pqsigRM

From: Perlner, Ray (Fed)
Sent: Tuesday, January 02, 2018 5:05 PM
To: pqc-comments
Cc: pqc-forum@list.nist.gov; Alperin-Sheriff, Jacob (Fed)
Subject: OFFICIAL COMMENT: pqsigRM

Dear pqsigRM submitters,

Jacob and I believe we have found an attack on pqsigRM. We believe the punctured columns of the public parity check matrix can be identified statistically from a few hundred signatures. E.g. When we ran the submitted code to produce signatures for parameter set 4-12, we found that the bits of the signature corresponding to punctures were set to 1 about 45% of the time, while the other bits were only set to 1 about 31% of the time.

Best regards,
Ray Perlner

From: Yongwoo Lee <yongwool@ccl.snu.ac.kr>
Sent: Monday, January 08, 2018 7:08 PM
To: Perlner, Ray (Fed); pqc-comments
Cc: pqc-forum@list.nist.gov; Alperin-Sheriff, Jacob (Fed); 노종선 교수님; 이위직; 김영식교수님
Subject: RE: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Dear Perlner, Dear All;

Thank for your valuable comments.

As you mentioned, we have checked that the probability of 1's among the punctured/inserted elements is higher than that of the unpunctured elements in our proposed pqsigRM algorithms.

As you can see Algorithm 3 in the supporting documentation, the punctured/inserted part of the signature is generated in the following way;

e'_p^T = s'_p + Re_(n-p)^T

where s'_p is generated from the output of SHA512.

Hence the probability of occurrence of ones in the punctured/inserted part of the signatures is about to 1/2 and the probability of occurrence of ones in the unpunctured part is about w/n.
(Precisely, since we choose e's having Hamming weight smaller than or equal to w as signature, e_p having larger Hamming weight is likely to be discarded. Hence, the probability of the occurrence of ones in the punctured/inserted part is slightly lower than 1/2. As you mentioned, it is about 45% in pqsigRM-4-12.)

As you mentioned, using this difference of the probabilities, an attacker can figure out the punctured/inserted elements. However, we think that this is not a major threat to the security of our proposed algorithm. Even though the attacker knows which bits are punctured/inserted in the signature, he cannot figure out the exact locations of the punctured/inserted bits before permutation.
The number of possible permutation matrices Q's becomes p!(n-p)! (= 2^(43071)in pqsigRM-4-12, 128-bit security) from n!(= 2^43250 in pqsigRM-4-12) if the locations of the punctured/inserted elements are known and it is still very large number and secure.
Moreover, to our knowledge, it does not reduce the complexity of any known attacks on RM code-based digital signature schemes such as Minder-Shokrollahi's attack, Chizhov-Borodin's attack, Square code attack, or information set decoding.

However, in order to avoid the possible threats, we'd like to slightly modify the algorithm and the parameters such that the probabilities of ones in the unpunctured and punctured/inserted parts are the same. The colored lines are slightly modified in Algorithm 3. (https://sites.google.com/view/pqsigrm/home/documentation, page 9)

Further, some parameters are also modified as in Table 1.
Table 2 shows the average numbers of iterations for signing the submitted algorithm and the modified algorithm, where the probabilities of ones in the unpunctured and punctured/inserted parts are the same.

Table 1. Parameters of the modified algorithms

Algorithms	original p	modified p	w_p	q
pqsigRM-4-12	20	16	7	59/256
pqsigRM-5-11	10	8	2	220/256
pqsigRM-6-12	20	8	2	1
pqsigRM-6-13	30	16	4	23/256

Table 2. Average number of iterations for signing

Algorithms	Avg. number of iter. (submitted)	Avg. number of iter. (modified)
pqsigRM-4-12	58.165	269.886 (4.64 times)
pqsigRM-5-11	6090.298	77590.397 (12.74 times)
pqsigRM-6-12	1774.464	277153.557 (156.19 times)
pqsigRM-6-13	7.4	637.880 (84.2 times)

From: Perlner, Ray (Fed) [mailto:ray.perlner@nist.gov]
Sent: Wednesday, January 3, 2018 7:05 AM
To: pqc-comments <pqc-comments@nist.gov>
Cc: pqc-forum@list.nist.gov; Alperin-Sheriff, Jacob (Fed) <jacob.alperin-sheriff@nist.gov>
Subject: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Dear pqsigRM submitters,

Jacob and I believe we have found an attack on pqsigRM. We believe the punctured columns of the public parity check matrix can be identified statistically from a few hundred signatures. E.g. When we ran the submitted code to produce signatures for parameter set 4-12, we found that the bits of the signature corresponding to punctures were set to 1 about 45% of the time, while the other bits were only set to 1 about 31% of the time.

Best regards,
Ray Perlner

From: Jacob Alperin-Sheriff <jacobmas@gmail.com>
Sent: Monday, January 08, 2018 10:04 PM
To: Yongwoo Lee
Cc: Perlner, Ray (Fed); pqc-comments; pqc-forum@list.nist.gov; Alperin-Sheriff, Jacob (Fed); 노종선 교수님; 이위직; 김영식교수님
Subject: Re: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Quick response tonight at home, Ray may add something tomorrow if he wants.

1. "As you mentioned, using this difference of the probabilities, an attacker can figure out the punctured/inserted elements. However, we think that this is not a major threat to the security of our proposed algorithm. Even though the attacker knows which bits are punctured/inserted in the signature, he cannot figure out the exact locations of the punctured/inserted bits before permutation.

The number of possible permutation matrices Q's becomes p!(n-p)! (= 2^(43071)in pqsigRM-4-12, 128-bit security) from n!(= 2^43250 in pqsigRM-4-12) if the locations of the punctured/inserted elements are known and it is still very large number and secure."

The total number of permutation matrices is irrelevant. The key point is that, by using the difference of probabilities, an attacker can find some permutation matrix Q' that moves each of the punctured bits to one of the final p positions of the vector (and ensures the non-punctured bits are all in the first n-p positions of the vector). Concretely, we may choose Q' to move the leftmost punctured bit to the leftmost of the final p positions, the next-leftmost punctured bit to the next leftmost of the final p positions, and so on. Obviously, we will (except with very very very very small probability) have that Q != Q'.

However, we WILL have that Q*(Q')^{-1} is a block diagonal matrix, i.e.

$$Q*(Q')^{-1} = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix}$$

where U_1 is an (n-p) x (n-p) permutation matrix and U_2 is a p x p permutation matrix.

Let X=H'*(Q')^{-1}, where H' is the public key.

Now, note that

$$\left[\begin{array}{c|c|c} [P'^{T} \mid I_{n-k-p}]U_1 & 0 \\ \hline RU_1 & U_2 \end{array}\right]$$

$$S^{-1}X = H_m*Q*(Q')^{-1}$$

If I'm not mistaken (I will check with Ray and sources of previous attacks tomorrow morning), I believe this means we can fully break the scheme.

2. As we said in the call for proposals and have reiterated on this forum, "because of limited resources, and also to avoid moving evaluation targets (i.e., modifying the submitted algorithms undergoing public review), NIST will NOT accept modifications to the submitted algorithms during this initial phase of evaluation."

The change you have proposed here is clearly a modification to the submitted algorithms, so we will not be accepting it and we will judge the algorithm as submitted (the same goes for all submissions).

On Mon, Jan 8, 2018 at 7:08 PM, Yongwoo Lee <yongwool@ccl.snu.ac.kr> wrote:

> Dear Perlner, Dear All;
>
> Thank for your valuable comments.
>
> As you mentioned, we have checked that the probability of 1's among the punctured/inserted elements is higher than that of the unpunctured elements in our proposed pqsigRM algorithms.
>
> As you can see Algorithm 3 in the supporting documentation, the punctured/inserted part of the signature is generated in the following way;
>
> $e'_p^T = s'_p + Re_{(n-p)}^T$
>
> where s'_p is generated from the output of SHA512.

From: Perlner, Ray (Fed)
Sent: Wednesday, January 10, 2018 5:58 PM
To: Jacob Alperin-Sheriff; Yongwoo Lee
Cc: pqc-comments; pqc-forum@list.nist.gov; Alperin-Sheriff, Jacob (Fed); 노종선 교수님; 이위직; 김영식 교수님
Subject: RE: [pqc-forum] OFFICIAL COMMENT: pqsigRM

We believe that once the punctured columns are identified, we can reconstruct the entire RM code (with permuted columns), at which point standard RM attacks like Minder-Shokrollahi and Chizhov-Borodin can be applied.

We also believe that, even if rejection sampling is applied, preventing signatures from giving away information about the punctured columns, the same information can be recovered relatively inexpensively from the public key alone. According to the estimate of Minder and Shokrollahi, the original RM generator matrix has at least 2^(rm – r(r-1)) minimum weight codewords (weight 2^(m-r)). These will all be orthogonal to the n-k-p dimensional subcode of the parity check matrix, which lacks support on the punctured columns. They can all be modified to codewords of the public code (with modestly increased weight) by substituting the appropriate bits in the punctured columns. (This is analogous to the signature procedure of the original scheme.) Such near-minimum-weight codewords can be found by standard information set decoding techniques, at a cost which we estimate to be significantly less than the claimed security level of any of the submitted parameter sets. Moreover, the punctured columns will be overrepresented in near minimum weight code words found by this technique.

Here's the procedure for reconstructing the code once you have the punctured columns:

First take the subcode of the public parity check matrix that lacks support on the punctured columns. (Remove the all-zero punctured columns from this subcode.) Now, you have a n - k-p x n-p submatrix of a parity check matrix for the original permuted r, m reed muller code. (Note it’s also a submatrix of the permuted parity check matrix if up to p columns of zeroes are appended.) The dual code of this matrix contains in its rowspace the truncation of all the reed muller code words from the original code.

Recall that the minimum weight codewords of the original code all have hamming 2^r. Find k-p linearly independent codewords from the truncated code that have weight 2^r. This can be done by information set decoding.

Now find a word in the truncated code with weight 2^r-1. (This can also be done by information set decoding.) Append a 1 to this code word, and append a zero to each of the k-p codewords from the previous step. These generate a k-p+1 x n-p+1 submatrix of a generator matrix of the permuted r,m reed muller code. Likewise, p-1 columns of zeroes could be appended, and it would still be a submatrix. Repeat this process, switching generator and parity check matrices each time to fill in all the missing columns of the generator and parity matrices of the punctured RM code.

We haven’t done a full complexity analysis of the above, but crude heuristic estimates suggest the cost to be somewhere around 2^70 for the originally submitted 128 and 192 bit parameters, and 2^100 for the 256 bit parameters.

From: Jacob Alperin-Sheriff [mailto:jacobmas@gmail.com]
Sent: Monday, January 08, 2018 10:04 PM

From: Wijik Lee <leewj422@gmail.com>
Sent: Thursday, January 18, 2018 11:23 AM
To: pqc-forum
Cc: jacobmas@gmail.com; yongwool@ccl.snu.ac.kr; pqc-comments; Alperin-Sheriff, Jacob (Fed); jsno@snu.ac.kr; leewj422@ccl.snu.ac.kr; iamyskim@chosun.ac.kr; Perlner, Ray (Fed)
Subject: Re: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Dear Perlner, Dear All;

Thank for your valuable comments.

1. By slightly modifying our proposed algorithm, we can make the probabilities of the punctured/inserted and the unpunctured bits equal.
In this case, we believe that it is hard to find the exact locations of the punctured columns from the public key H'.
In your comments, the near-minimum-weight codewords can be found by standard information set decoding techniques.
In fact, the codewords generated from H' are not true codewords of RM code but the vectors replaced by the random bits in the unknown punctured locations.
Further, the Hamming weight of those vectors is larger than or equal to d_min – p and we dont know their weight distribution.

2. If we do not modify the our proposed algorithm, we need to increase the parameters of RM(r,m) to increase the security level.
In case of RM(6, 13), the security level will be close to 128 bits.
We didn't calculate exact security level yet, however:
-The number of codewords with Hamming weight d_min in the punctured RM codes is reduced.
-The complexiy of finding n-p "independent" codewords with Hamming weight d_min needs more than that of finding n-p codewords with d_min.

2018년 1월 11일 목요일 오전 7시 58분 7초 UTC+9, Perlner, Ray (Fed) 님의 말:

> We believe that once the punctured columns are identified, we can reconstruct the entire RM code (with permuted columns), at which point standard RM attacks like Minder-Shokrollahi and Chizhov-Borodin can be applied.
>
> We also believe that, even if rejection sampling is applied, preventing signatures from giving away information about the punctured columns, the same information can be recovered relatively inexpensively from the public key alone. According to the estimate of Minder and Shokrollahi, the original RM generator matrix has at least 2^(rm – r(r-1)) minimum weight codewords (weight 2^(m-r)). These will all be orthogonal to the n-k-p dimensional subcode of the parity check matrix, which lacks support on the punctured columns. They can all be modified to codewords of the public code (with modestly increased weight) by substituting the appropriate bits in the punctured columns. (This is analogous to the signature procedure of the original scheme.) Such near-minimum-weight codewords can be found by standard information set decoding techniques, at a cost which we estimate to be significantly less than the claimed

From: Perlner, Ray (Fed)
Sent: Friday, January 19, 2018 10:16 AM
To: Wijik Lee; pqc-forum
Cc: jacobmas@gmail.com; yongwool@ccl.snu.ac.kr; pqc-comments; Alperin-Sheriff, Jacob (Fed); jsno@snu.ac.kr; leewj422@ccl.snu.ac.kr; iamyskim@chosun.ac.kr
Subject: Re: [pqc-forum] OFFICIAL COMMENT: pqsigRM

I'm confused why you think your point 1 contradicts our claim that we can recover the locations of the punctured columns from the private key alone.

"In fact, the codewords generated from H' are not true codewords of RM code but the vectors replaced by the random bits in the unknown punctured locations."

Indeed. If the modified RM codeword in question is a minimum weight codeword, the punctured bits will have probability 1/2 to be set to 1, while the non-punctured bits will only be 1 with probability dmin/n. Since there is such a modified codeword for every minimum weight codeword in the original RM code, and their weight is only expected to be larger than dmin by a little less than p/2, I don't believe it would be difficult to recover enough such modified codewords to detect the puncturing locations.

While I reiterate, we are not accepting modifications to submitted parameters at this time, I don't think making the weight distribution of signatures more uniform is sufficient to hide the locations of the punctured columns.

From: Wijik Lee <leewj422@gmail.com>
Sent: Thursday, January 18, 2018 11:22:57 AM
To: pqc-forum
Cc: jacobmas@gmail.com; yongwool@ccl.snu.ac.kr; pqc-comments; Alperin-Sheriff, Jacob (Fed); jsno@snu.ac.kr; leewj422@ccl.snu.ac.kr; iamyskim@chosun.ac.kr; Perlner, Ray (Fed)
Subject: Re: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Dear Perlner, Dear All;
Thank for your valuable comments.

1. By slightly modifying our proposed algorithm, we can make the probabilities of the punctured/inserted and the unpunctured bits equal.
In this case, we believe that it is hard to find the exact locations of the punctured columns from the public key H'.
In your comments, the near-minimum-weight codewords can be found by standard information set decoding techniques.
In fact, the codewords generated from H' are not true codewords of RM code but the vectors replaced by the random bits in the unknown punctured locations.
Further, the Hamming weight of those vectors is larger than or equal to d_min – p and we dont know their weight distribution.
2. If we do not modify the our proposed algorithm, we need to increase the parameters of RM(r,m) to increase the security level.
In case of RM(6, 13), the security level will be close to 128 bits.
We didn't calculate exact security level yet, however:
-The number of codewords with Hamming weight d_min in the punctured RM codes is reduced.
-The complexiy of finding n-p "independent" codewords with Hamming weight d_min needs more than that of finding n-p codewords with d_min.

From: Moody, Dustin (Fed)
Sent: Monday, February 05, 2018 2:12 PM
To: Wijik Lee; pqc-forum
Cc: jacobmas@gmail.com; yongwool@ccl.snu.ac.kr; pqc-comments; Alperin-Sheriff, Jacob (Fed); jsno@snu.ac.kr; leewj422@ccl.snu.ac.kr; iamyskim@chosun.ac.kr; Perlner, Ray (Fed)
Subject: RE: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Pqsigrm team,

We are working on the program for the 1st NIST PQC workshop, which is quite challenging. We received a large number of submissions, and only have 2 days.

There are several submissions which have been attacked, with the submitter(s) acknowledging (to some degree) that the attack(s) are successful. NIST will be evaluating the security of all submissions with respect to the original algorithm and parameters contained in the submission; for the 1st round, as per the call for proposals: we are not allowing changes to submitted algorithm or parameters to avoid moving targets. Any team whose submission has successfully been broken should probably consider withdrawing their submission.

pqsigrm is one of these submissions with an attack. We ask you to consider sending us slides or a video in place or presenting, as it would help us in creating the workshop program. Right now, the time constraints we have, combined with the number of submissions is making the schedule difficult. We are still encouraging you to attend, but feel the time for presentations might be best served by being used for submissions which have not yet been successfully attacked. Please let us know if you would still like to present, or will instead send us slides or a video.

Please let us know as soon as you can. Thanks,

Dustin

From: Wijik Lee [mailto:leewj422@gmail.com]
Sent: Thursday, January 18, 2018 11:23 AM
To: pqc-forum <pqc-forum@list.nist.gov>
Cc: jacobmas@gmail.com; yongwool@ccl.snu.ac.kr; pqc-comments <pqc-comments@nist.gov>; Alperin-Sheriff, Jacob (Fed) <jacob.alperin-sheriff@nist.gov>; jsno@snu.ac.kr; leewj422@ccl.snu.ac.kr; iamyskim@chosun.ac.kr; Perlner, Ray (Fed) <ray.perlner@nist.gov>
Subject: Re: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Dear Perlner, Dear All;

Thank for your valuable comments.

1. By slightly modifying our proposed algorithm, we can make the probabilities of the punctured/inserted and the unpunctured bits equal.
In this case, we believe that it is hard to find the exact locations of the punctured columns from the public key H'.
In your comments, the near-minimum-weight codewords can be found by standard information set decoding techniques.
In fact, the codewords generated from H' are not true codewords of RM code but the vectors replaced by the random bits in the unknown punctured locations.

From: Perlner, Ray (Fed)
Sent: Friday, March 09, 2018 12:30 PM
To: pqc-comments
Cc: pqc-forum@list.nist.gov
Subject: OFFICIAL COMMENT: pqsigRM
Attachments: Equivalent Key for pqsigRM -- Sage.pdf; Equivalent Key for pqsigRM.sws

Dear pqrmsig submitters,

Jacob, Dustin and I have dramatically improved our attack on your proposed 128 and 192 bit parameters. Our implementation of the attack on the 192 bit parameters can recover an equivalent private key in a matter of seconds and we expect similar performance for the 128 bit parameters.

1) We can trivially locate the punctured columns by taking the support of the intersection of the public code and its dual code. The code will have support everywhere except on the punctured columns. (This applies to all three parameter sets.)
2) For the 192 and 128 bit parameters (rm4,12 and rm 6,11) we can apply the Chizov-Borodin attack starting from the intersection of the code and its dual code. This will be a subcode of rm4,12 for the 128 bit parameters and rm5, 12 for the 192 bit parameters. The attack performs best when, instead of simply computing a product code when the Chizov-Borodin attack calls for it, we start with the union of the two codes being multiplied and add codewords from the product code until we reach the desired rank (e.g. when squaring the subcode of rm5,12 with 30 punctures, we stop when the rank is 30 less than the expected rank of an rm10,12 code.) The attack yields a permutation that takes the columns of the public parity check matrix to the columns of a punctured reed muller code of the appropriate size.

In summary, combining these results with our previous observations, it seems that all the known attacks on the Sidelnikov cryptosystem carry over with minimal overhead to the punctured case. It is possible that a self-dual instance of the Sidelnikov cryptosystem might be secure, but it likely requires a code larger than even your largest parameter set (which is itself based on a self-dual code.) The next large Reed Muller code would be a rm(7,15) code, which would yield a key size of 32 megabytes. It should also be noted that all disguised Reed Muller codes, including their punctured variants, are detectably non-random, since, unlike random codes, they have a large intersection with their dual codes.

I have attached the sage file implementing our attack on the 192 bit parameters and a pdf record of the output. Dustin wishes to apologize for the amateurish coding.

Ray Perlner

Equivalent Key for pqsigRM

```
W1='/Users/dmoody/Desktop/pqsigrm3.txt'
f=open(W1)
L=f.readlines()
```

```
pk=L[6].rstrip()[5:]
```

```
def hextobin(hx):
    """
    return hex string to binary string
    """
    b=bin(ZZ(hx,base=16))
    return b[2:]

def hexbin(st):
    W=''
    for ct in range(0,len(st)):
        ts=hextobin(st[ct])
        while len(ts)<4:
            ts='0'+ts
        W=W+ts
    return W
```

```
PK=hexbin(pk)
```

```
len(PK)/4096
```

```
1586
```

```
def dual(mat):
    mat1=mat.rref()
    T1=[]
    for j in range(0,mat1.nrows()):
        if mat1.row(j)==0:
            T1.append(j)
    mat2=mat1.delete_rows(T1)
    T3=[]
    T2=[]
    mr=mat2.rank()
    for j in range(0,mat2.ncols()):
        if j<mr and j not in mat2.pivots():
            T3.append(j)
        if j>mr-1 and j in mat2.pivots():
            T2.append(j)
    for j in range(0,len(T2)):
        mat2.swap_columns(T3[j],T2[j])
    mat2=mat2.rref()
    mat3=mat2.submatrix(0,mr,mr,mat2.ncols()-mr)
    mat4=mat3.transpose()
    i5=matrix.identity(GF(2),mat4.nrows())
    mat5=mat4.augment(i5)
    for j in range(0,len(T2)):
        mat5.swap_columns(T3[j],T2[j])
    return mat5
```

```
R=GF(2)
```

```
M=matrix(GF(2), 1586, 4096, lambda i, j: R(PK[j+4096*i]));
M
```

```
1586 x 4096 dense matrix over Finite Field of size 2 (use the
'.str()' method to see the entries)
```

```
M.rank()
```

```
1586
```

```
M2=dual(M)
M2
```

```
2510 x 4096 dense matrix over Finite Field of size 2 (use the
'.str()' method to see the entries)
```

```
M2.rank()
```

```
2510
```

```
M3=M.stack(M2)
M3
```

```
4096 x 4096 dense matrix over Finite Field of size 2 (use the
'.str()' method to see the entries)
```

```
M3.rank()
```

```
2570
```

```
M4=dual(M3)
M4
```

```
1526 x 4096 dense matrix over Finite Field of size 2 (use the
'.str()' method to see the entries)
```

```
Z3=[]
for j in range(0,M4.ncols()):
    if M4.column(j)==0:
        Z3.append(j)
len(Z3)
```

```
30
```

```
print Z3
```

```
[154, 345, 571, 601, 958, 1123, 1430, 1471, 1739, 2021, 2186, 2195,
2240, 2441, 2468, 2620, 2770, 2840, 2888, 2932, 2940, 3232, 3490,
3535, 3591, 3875, 3936, 4004, 4067, 4084]
```

```
mr4=M4.rank()
mr4
```

```
1526
```

```
flg=0
flg1=0
for c1 in range(0,mr4):
    for c2 in range(c1,mr4):
      r1=M4.row(c1)
      r2=M4.row(c2)
      if flg==0:
          M5=copy(M4)
          flg=1
      mm=matrix(GF(2),1,M4.ncols(),lambda i,j: r1[j]*r2[j])
      M5=M5.stack(mm)
      if c2==mr4-1:
          M5=M5.rref()
          W=[]
          for rw in range(0,M5.nrows()):
              if M5.row(rw)==0:
                  W.append(rw)
          M5=M5.delete_rows(W)
          mr=M5.rank()
          print c1,mr
          if mr==M5.ncols()-13-len(Z3):
              print 'done'
              flg1=2
              break
          if flg1==2:
              break
      if flg1==2:
          break
```

```
0 2768
1 3399
2 3726
3 3888
4 3973
5 4018
6 4036
7 4044
8 4047
9 4049
10 4051
11 4052
12 4053
done
1525 4053
done
```

```
M5
```

```
4053 x 4096 dense matrix over Finite Field of size 2 (use the
'.str()' method to see the entries)
```

```
M6=dual(M5)
M6
```

```
43 x 4096 dense matrix over Finite Field of size 2 (use the '.str()'
```

```
    method to see the entries)
```

```
M7=M6.rref()
M7
```

```
    43 x 4096 dense matrix over Finite Field of size 2 (use the '.str()'
    method to see the entries)
```

```
Msub=M7.submatrix(0,0,13,4096)
Msub
```

```
    13 x 4096 dense matrix over Finite Field of size 2 (use the '.str()'
    method to see the entries)
```

```
M8=matrix(Msub.row(0)+Msub.row(1))
for j in range(1,12):
    M8=M8.stack(matrix(Msub.row(j)+Msub.row(j+1)))
M8
```

```
    12 x 4096 dense matrix over Finite Field of size 2 (use the '.str()'
    method to see the entries)
```

```
ct=0
CSet=M8.columns()
CSet.sort()
for j in range(0,len(CSet)-1):
    if CSet[j]==CSet[j+1]:
        ct=ct+1
print ct
```

```
    30
```

```
ZM=matrix.zero(GF(2),1586,4096)
ZM
```

```
    1586 x 4096 dense matrix over Finite Field of size 2 (use the
    '.str()' method to see the entries)
```

```
def coltobin(col):
    sm=0
    for j in range(0,12):
        sm=sm+2^j*ZZ(col[11-j])
    return sm
```

```
for j in range(0,4096):
    col=M8.column(j)
    tn=coltobin(col)
    ZM.set_column(tn,M.column(j))
```

```
ZM
```

```
    1586 x 4096 dense matrix over Finite Field of size 2 (use the
    '.str()' method to see the entries)
```

```
S.<x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12>=GF(2)[]
S
```

```
    Multivariate Polynomial Ring in x1, x2, x3, x4, x5, x6, x7, x8, x9,
    x10, x11, x12 over Finite Field of size 2
```

```
P=[]
for j in range(0,32):
    P.append(S.random_element(5))
```

```
def func(f1,j):
    st=bin(j)[2:].zfill(12)
    return
f1(ZZ(st[0]),ZZ(st[1]),ZZ(st[2]),ZZ(st[3]),ZZ(st[4]),ZZ(st[5]),ZZ(st[6]),ZZ(st[7]),ZZ(st[8]),ZZ(st[9]),ZZ(st[10]),
```

```
NM=matrix(GF(2), 1, 4096, lambda i, j: func(P[0],j));
```

```
for jj in range(1,32):
    NM1=matrix(GF(2), 1, 4096, lambda i, j: func(P[jj],j));
    NM=NM.stack(NM1)
NM
```

```
    32 x 4096 dense matrix over Finite Field of size 2 (use the '.str()'
    method to see the entries)
```

```
Z5=[]
for j in range(0,4096):
    if ZM.column(j)==0:
        Z5.append(j)
len(Z5)
```

```
    30
```

```
NM2=matrix(GF(2), 32, 30, lambda i, j: NM[i,Z5[j]]);
NM2
```

```
32 x 30 dense matrix over Finite Field of size 2 (use the '.str()'
method to see the entries)
```

```
B1=NM2.left_kernel().basis()
```

```
NM3=B1[0]*NM
```

```
NM3 in ZM.row_space()
```

```
True
```

```

```

```

```

From: Yongwoo Lee <yongwool@ccl.snu.ac.kr>
Sent: Monday, April 02, 2018 3:44 AM
To: pqc-comments
Cc: pqc-forum@list.nist.gov
Subject: RE: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Dear Dr. Perlner, Dr. Moody and Dr. Alperin-Sheriff.

Thank you for your comments.
We found that the proposed attacks can be prevented by simply changing the random matrix part of the generator matrix to another position of the generator matrix.

The public key of pqsigRM is a permuted parity check matrix corresponding to the generator matrix of the RM code, in which p columns are replaced by random vectors.
Here, we will simply replace another position of the generator matrix with random matrix, instead of "p col umns".

For example, in pqsigRM-6-13, G represents the generator matrix of RM (6,13).
We replace the partial matrices, G[3534:3790, 6144: 6656], G[3534:3790, 6656:7168], G[3534:3790, 7168: 76 80], and G[3534:3790, 7680: 8192] with [R|R], where R is a 256 * 256 binary random non-singular matrix an d this modified generator matrix is referred to as G_m.
(G[3534:3790, 6144: 6656], G[3534:3790, 6656:7168], G[3534:3790, 7168: 7680], and G[3534:3790, 7680: 819 2] originally corresponds to the generator matrix of RM(4,9).)
G_m is described as in Fig.1.

Fig. 1. Generator matrix of modified RM code

Then we can build the parity check matrix H_m corresponding G_m, and generate an (n-k)*(n-k) scrambler matrix S and n*n permutation matrix Q.
The public key is given as H′ = S*H_m*Q.
The private keys are given as S, Q.

In this case, we do not need to obtain e_p separately when signing.
Instead, we can include this process in syndrome decoding.
You can decode this code by simply adding two lines to the original recursive decoding of RM code. Modi fications are shown in Algorithm 1.(the first two lines in red).

With this modification, there are no all-zero position on the hull of public key.
The probability of 1's in the elements of the signature is not different.
Near-minimum weight codewords are no longer useful for locating the punctured/inserted positions.
Because 1/4 elements of each codeword are replaced by random elements and the minimum weight of th e code is much less than n/4.
Modifying the generator matrix in this way also prevents square code attack, Chizhov-Borodin attack, and Minder-Shokrollahi attack.

For 196-bit security,
we replace G[1868: 2124, 2560: 3072] and G[1868: 2124, 3584: 4096] with [R|R], where G is a generator ma trix of RM(6,12).

For 128-bit security,
we replace G[894: 958, 1536: 1664], G[894: 958, 1664: 1792], G[894: 958, 1792: 1920] and G[894: 958, 1920: 2048] with [R|R], where G is a generator matrix of RM(5,11).

Algorithm 1. Modified decoding of pqsigRM

```
rec_dec(y, r, m, rear, front):
  if front == 3534 and rear == 3790:
    y[front : rear] <- the nearest 2-repetition codeword from y[front : rear]
  else if r == 0:
      d1 <- distance(y[front:rear],  [-1 -1 -1 ... -1]);
      d2 <- distance(y[front:rear],  [ 1 1 1 ... 1]);
      if d1 < d2:
              y[front:rear] <- [-1 -1 -1 ... -1]
      else:
              y[front:rear] <- [ 1  1 1 ... 1]
   elif r == m:
```

```
    for i from front to rear:
            y[i] <- (y[i] >= 0)? 1: -1
  else:
    mid = (front + rear)/2
    y_v <- copy( y[mid : rear] )
    y [ mid : rear ] <- y[mid : rear] * y[ front : mid ]
    rec_dec( y, r-1, m-1, mid, rear)
    y [ front : mid ] <- (y [ front : mid ] + y [ mid : rear ] * y_v)/2
    rec_dec( y, r, m-1, rear, mid)
    y [ mid : rear ] <- y[mid : rear] * y[ front : mid ]
```

Yongwoo Lee.

From: Perlner, Ray (Fed) <ray.perlner@nist.gov>
Sent: Saturday, March 10, 2018 2:30 AM
To: pqc-comments <pqc-comments@nist.gov>
Cc: pqc-forum@list.nist.gov
Subject: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Dear pqrmsig submitters,

Jacob, Dustin and I have dramatically improved our attack on your proposed 128 and 192 bit parameters. Our implementation of the attack on the 192 bit parameters can recover an equivalent private key in a matter of seconds and we expect similar performance for the 128 bit parameters.

1) We can trivially locate the punctured columns by taking the support of the intersection of the public code and its dual code. The code will have support everywhere except on the punctured columns. (This applies to all three parameter sets.)
2) For the 192 and 128 bit parameters (rm4,12 and rm 6,11) we can apply the Chizov-Borodin attack starting from the intersection of the code and its dual code. This will be a subcode of rm4,12 for the 128 bit parameters and rm5, 12 for the 192 bit parameters. The attack performs best when, instead of simply computing a product code when the Chizov-Borodin attack calls for it, we start with the union of the two codes being multiplied and add codewords from the product code until we reach the desired rank (e.g. when squaring the subcode of rm5,12 with 30 punctures, we stop when the rank is 30 less than the expected rank of an rm10,12 code.) The attack yields a permutation that takes the columns of the public parity check matrix to the columns of a punctured reed muller code of the appropriate size.

In summary, combining these results with our previous observations, it seems that all the known attacks on the Sidelnikov cryptosystem carry over with minimal overhead to the punctured case. It is possible that a self-dual instance of the Sidelnikov cryptosystem might be secure, but it likely requires a code larger than even your largest parameter set (which is itself based on a self-dual code.) The next large Reed Muller code would be a rm(7,15) code, which would yield a key size of 32 megabytes. It should also be noted that all disguised Reed Muller codes, including their punctured variants, are detectably non-random, since, unlike random codes, they have a large intersection with their dual codes.

I have attached the sage file implementing our attack on the 192 bit parameters and a pdf record of the output. Dustin wishes to apologize for the amateurish coding.

Ray Perlner

From: Perlner, Ray (Fed)
Sent: Wednesday, April 04, 2018 12:25 PM
To: Yongwoo Lee; pqc-comments
Cc: pqc-forum@list.nist.gov
Subject: RE: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Dear submitters,

We believe that the modification suggested in your previous email makes the RM(6,13) code significantly weaker than the unmodified code and we believe all of the modified codes in the previous email can be practically broken.

A couple of points to make the notation easier. First, note that WLOG, we may assume that the 256x256 matrix R is an identity matrix, since (0….0|R|R|R|R|R|R|R) and R^-1(0….0|R|R|R|R|R|R|R) = (0….0|I||I||I||I||I||I||I) generate the same code. Second, we will think of codewords as being polynomials over the variables (x0, x1, x2 x3 … x12), where x0 represents the high order bit of the column index and x12 represents the low order bit of the column index. So, for example, the support of x0x1 is identical to the locations of the modified columns, and the codewords generated by (0….0|I||I||I||I||I||I||I) are of the form x0*x1*p(x5, x6, x7, … x12), for some polynomial p. Note also that the symmetries of the modified code are such that it will have the same form if we substitute any degree 1 polynomials for (x0, x1, x2 x3 … x12) and reorder the columns accordingly, as long as the above two properties hold. We now describe the procedure for key recovery.

1) Use information set decoding to find codewords in the modified code that have hamming weight 8. There will be 256 such codewords and their combined support will be the support of x0*x1.
2) We can also easily recover the code generated by x0 and x1. If we take the subcode of the public code that lacks support on x0*x1 (i.e. the dual code of (1+ x0x1) times the public code) and square it, we get a code whose dual code (restricted to the columns where x0*x1 lacks support) is generated by 1, x0, and x1.
3) WLOG we may pick two weight 4096 codewords from this dual code, each containing the support of x0x1, and call them x0 and x1.
4) We may now apply Chizov Borodin to the submatrices of the public code consisting of the support columns of 1+ x0 and 1+x1. Each submatrix has a rowspace equal to an RM (6,12) code and can be attacked cheaply since GCD(6,12-1) = 1. We need to make sure that the two column orderings agree, but this just amounts to a linear constraint that the same degree 1 codeword is assigned to be x2, x3 … x12 in both cases.
5) The Chizov Borodin attack may also be applied to x1 times the public code, but in this case there is a slight complication. The rowspace of that code contains an RM(6,12) code, but it also contains the codewords generated by (0….0|I||I||I||I||I||I||I). To get rid of these, we may take the intersection of the code with its dual code, resulting in a subcode of RM (5,12). This will not contain all the codewords from RM(5,12), but it will contain all degree 4 monomials that do not include a factor of x2*x3*x4. As such, the square of this code will be identical to RM(8,12), since we can get degree 8 monomials containing x2*x3*x4 e.g. by multiplying x2*x5*x6*x7*x8 by x3*x4*x9*x10*x11 to get x2*x3*x4*x5*x6*x7*x8*x9*x10*x11. Again we have some linear constraints on the choice of degree 1 codewords. In particular, all three assignments of degree 1 codewords to variables (x0, x1, … x12) must agree, x0 and x1 must agree with our original assignment of variables, and x5, … x12 must either be identically 1 or identically 0 on the support of each of the weight 8 codewords extracted in step 1.

We've verified some of this experimentally, but haven't yet implemented the whole attack.

From: Yongwoo Lee [mailto:yongwool@ccl.snu.ac.kr]
Sent: Monday, April 02, 2018 3:44 AM
To: pqc-comments <pqc-comments@nist.gov>

From: Yongwoo Lee <yongwool@ccl.snu.ac.kr>
Sent: Wednesday, April 11, 2018 10:31 AM
To: Perlner, Ray (Fed)
Cc: pqc-forum@list.nist.gov; pqc-comments; Jong-Seon No; 김영식; ccl 이위직형
Subject: Re: [pqc-forum] OFFICIAL COMMENT: pqsigRM

Follow Up Flag: Follow up
Flag Status: Flagged

Dear Dr. Perlner.

Thank you for your comment.

After reviewing the attacking algorithm you proposed, we found that we could modify our proposed algorithm to defend against that attack.

To be more specific, the inserted matrix does not have to be a generator matrix of 2-repitition codes.

This can be replaced by a generator matrix of any code that has a decoding algorithm which returns a codeword even in the presence of a large error.

For example, we can partially modify the generator matrix of RM(5,11) with the permuted generator matrix of RM(4, 9), instead of 2-repitition codeas below.

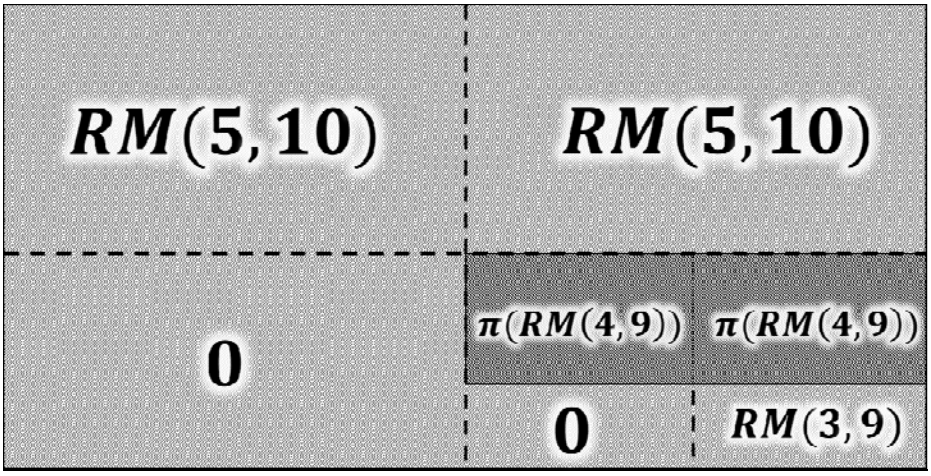
Experiments have shown that decoding performance is good in this case.(Of course, the decoding requires additional depermutation.)

Applying this idea, we have devised a way to replace the larger part of the generator matrix.

For example, for 128-bit security, the public code is, H' = S*H_m*Q, where H_m is the parity check matrix of the modified RM code, generated by the modified generator matrix of RM(5,11) as below:

.

And then, the decoding algorithm becomes:



Algorithm 1. Modified decoding of pqsigRM

```
rec_dec(y, r, m, rear, front):
 if r == 0:
    perform MD decoding on RM(0,m)
 elif r == m:
     perform MD decoding on RM(r,r)
  else:
    if front == 1024 and rear == 1536:
       depermutation on y[front : rear]
     mid = (front + rear)/2
     y_uv <- copy( y[mid : rear] )
     y [ mid : rear ] <- y[mid : rear] * y[ front : mid ]
     rec_dec( y, r-1, m-1, mid, rear)
     y [ front : mid ] <- (y [ front : mid ] + y [ mid : rear ] * y_uv )/2
     rec_dec( y, r, m-1, rear, mid)
     y [ mid : rear ] <- y[mid : rear] * y[ front : mid ]
   if front == 1024 and rear == 1536:
     permutation on y[front : rear]
```

This modification allows the huge part of generator matrix replaced while achieving good decoding performance.

We also will upload our modified document.

Best regards.

Yongwoo Lee