So I’ve been doing some basic testing via adding \texttt{cpucycles} calls to the \texttt{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 \texttt{pqsigRM} (specifically \texttt{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 \texttt{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 \texttt{cpucycles})

\begin{verbatim}
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;
}
\end{verbatim}

—Jacob Alperin-Sheriff
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.

<table>
<thead>
<tr>
<th></th>
<th>security</th>
<th>key generation</th>
<th>singing</th>
<th>verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>pqsigRM-4-12</td>
<td>128</td>
<td>14639777783</td>
<td>3971208456</td>
<td>139814898</td>
</tr>
<tr>
<td>pqsigRM-6-12</td>
<td>196</td>
<td>6395769782</td>
<td>3275234719</td>
<td>198607502</td>
</tr>
<tr>
<td>pqsigRM-6-13</td>
<td>256</td>
<td>72162115384</td>
<td>1087667252</td>
<td>956410761</td>
</tr>
</tbody>
</table>

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
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
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

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>original p</th>
<th>modified p</th>
<th>w_p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>pqsigRM-4-12</td>
<td>20</td>
<td>16</td>
<td>7</td>
<td>59/256</td>
</tr>
<tr>
<td>pqsigRM-5-11</td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>220/256</td>
</tr>
<tr>
<td>pqsigRM-6-12</td>
<td>20</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>pqsigRM-6-13</td>
<td>30</td>
<td>16</td>
<td>4</td>
<td>23/256</td>
</tr>
</tbody>
</table>

Table 2. Average number of iterations for signing

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Avg. number of iter.</th>
<th>Avg. number of iter.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(submitted)</td>
<td>(modified)</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>pqsigRM-4-12</td>
<td>58.165</td>
<td>269.886 (4.64 times)</td>
</tr>
<tr>
<td>pqsigRM-5-11</td>
<td>6090.298</td>
<td>77590.397 (12.74 times)</td>
</tr>
<tr>
<td>pqsigRM-6-12</td>
<td>1774.464</td>
<td>277153.557 (156.19 times)</td>
</tr>
<tr>
<td>pqsigRM-6-13</td>
<td>7.4</td>
<td>637.880 (84.2 times)</td>
</tr>
</tbody>
</table>

From: Perlner, Ray (Fed) [mailto:ray.perlner@nist.gov]
Sent: Wednesday, January 3, 2018 7:05 AM
To: 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
--

You received this message because you are subscribed to the Google Groups "pqc-forum" group.
To unsubscribe from this group and stop receiving emails from it, send an email to pqc-forum+unsubscribe@list.nist.gov.
Visit this group at https://groups.google.com/a/list.nist.gov/group/pqc-forum/.
--

You received this message because you are subscribed to the Google Groups "pqc-forum" group.
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^\ast(Q')^\ast(-1) is a block diagonal matrix, i.e.

\[
Q^\ast(Q')^\ast(-1) = \begin{bmatrix}
U_1 & 0 \\
0 & U_2
\end{bmatrix}
\]

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^\ast(Q')^\ast(-1), where H' is the public key.
Now, note that

\[
\begin{bmatrix}
P'^{T} \mid I_{n-k-p} \end{bmatrix} U_1 \mid 0
\]

\[
S^{-1} X = H_m * Q * (Q')^{-1} = \begin{bmatrix}
RU_1 \\
U_2
\end{bmatrix}
\]

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:

\[e'_p^{T} = s'_p + R_{(n-p)}^{T}\]

where s'_p is generated from the output of SHA512.
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 $2^r(rm - r(r-1))$ 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 $n-k-p \times n-p$ 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 $k-p+1 \times n-p+1$ 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.