Nibbling MAYO

Ward Beullens,
Fabio Campos,
Sofía Celi,
Basil Hess, and
Matthias J. Kannwischer.
Summary of results

We propose new high-speed implementations of MAYO for x86 and Cortex-M4 platforms.

• Our implementations use a new representation of public key. Nibble-sliced instead of bit-sliced.

• Our implementations are based on the “Method of the four Russians” which we found to be more efficient than bitsliced arithmetic.
5-part API

Compact KeyGen

SK Expansion

PK Expansion

Sign

Verify
Ice Lake performance MAYO 1
AVX2 + AESNI Bit-sliced | Nibble-sliced implementation
Ice Lake performance MAYO 1
AVX2 + AESNI Bit-sliced | Nibble-sliced implementation

KeyGen

110 k | 44 k

Sign

459 k | 218 k

Verify

148 k | 53 k

Dilithium2: KeyGen 81 k, Sign 219 k, Verify 79 k
Cortex-M4 performance MAYO 1

ST NUCLEO-L4R5ZI @ 20 MHz  Bit-sliced | Nibble-sliced

KeyGen: 5.2 M | 4.4 M

Sign: 9.1 M | 8.2 M

Verify: 4.8 M | 4.8 M

Dilithium2: KeyGen 1.6 M, Sign 4.0 M, Verify 1.6 M
Overly simplified description of MAYO1

Key Gen:
• Multiply 64 matrices of size 58-by-58 by a 58-by-8 matrix

Sign:
• Multiply 64 matrices of size 58-by-58 by a 58-by-8 matrix
• Multiply 64 matrices of size 58-by-58 by a 58-by-9 matrix
• Solve a system of 64 linear equations in 72 variables

Verify:
• Multiply 64 matrices of size 66-by-66 by a 66-by-9 matrix
Bit-sliced v.s. Nibble-sliced representations

How to represent matrices over $GF(16)$? Representation is irrelevant for security, but important for interoperability and efficient implementation.

<table>
<thead>
<tr>
<th>Bit-sliced representation</th>
<th>Nibble-sliced representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0 + a_1 x + a_2 x^2 + a_3 x^3$</td>
<td>$d_0 + d_1 x + d_2 x^2 + d_3 x^3$</td>
</tr>
<tr>
<td>$b_0 + b_1 x + b_2 x^2 + b_3 x^3$</td>
<td>$e_0 + e_1 x + e_2 x^2 + e_3 x^3$</td>
</tr>
<tr>
<td>$c_0 + c_1 x + c_2 x^2 + c_3 x^3$</td>
<td>$f_0 + f_1 x + f_2 x^2 + f_3 x^3$</td>
</tr>
</tbody>
</table>

(Column major) bit-sliced representation: $a_0 b_0 c_0 \ a_1 b_1 c_1 \ a_2 b_2 c_2 \ a_3 b_3 c_3 \ ...$  
(Column major) nibble-sliced representation: $a_0 a_1 a_2 a_3 \ b_0 b_1 b_2 b_3 \ c_0 c_1 c_2 c_3 \ ...$

Good for bit-sliced arithmetic on embedded platforms.  
Good for AVX2 shuffle-based arithmetic on “big” CPUs

Initially, we chose the bit-sliced representation, because on “big” CPUs MAYO is fast enough anyway.
Bit-sliced v.s. Nibble-sliced representations

How to represent matrices over $GF(16)$? Representation is irrelevant for security, but important for interoperability and efficient implementation.

<table>
<thead>
<tr>
<th>$a_0 + a_1 x + a_2 x^2 + a_3 x^3$</th>
<th>$d_0 + d_1 x + d_2 x^2 + d_3 x^3$</th>
<th>$g_0 + g_1 x + g_2 x^2 + g_3 x^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_0 + b_1 x + b_2 x^2 + b_3 x^3$</td>
<td>$e_0 + e_1 x + e_2 x^2 + e_3 x^3$</td>
<td>$h_0 + h_1 x + h_2 x^2 + h_3 x^3$</td>
</tr>
<tr>
<td>$c_0 + c_1 x + c_2 x^2 + c_3 x^3$</td>
<td>$f_0 + f_1 x + f_2 x^2 + f_3 x^3$</td>
<td>$i_0 + i_1 x + i_2 x^2 + i_3 x^3$</td>
</tr>
</tbody>
</table>

(Column major) bit-sliced representation: $a_0, b_0, c_0, a_1, b_1, c_1, a_2, b_2, c_2, a_3, b_3, c_3, \ldots$

Good for bit-sliced arithmetic on embedded platforms.

Initially, we chose the bit-sliced representation, because on “big” CPUs MAYO is fast enough anyway.

Contribution of this paper: Nibble-sliced representation is also good for Cortex M4, so we should switch.
Method of the 4 Russians
Arlazarov, Dinic, Kronrod, Faradzev (1974)

Method for computing matrix multiplication $A \times B$, where $A$ has dimensions $l$ by $m$, and $B$ has dimensions $m$ by $n$.
Using only $O\left(\frac{lmn}{\log q} \right)$ additions and table lookups.

Step 1: Reduce to to case where $A$ is very tall and narrow

Step 2: Do multiplication by $A$ using table lookups
Step 1: split $A$ and $B$ in narrow strips

$A_i$ have width $t \approx \log_q l$, where $l$ is the height of $A$, and $q$ is the size of the finite field.
Step 2: Multiplication by table lookup

• Make a table that contains all the linear combinations of rows of $B_i$. (Table has size $q^t n = ln$, requires $ln$ additions to construct)

• Compute $A_i * B_i$ by looking up each row in the table. ($l$ lookups of rows of $n$ elements)

• Cost is $O(ln)$ and needs to be repeated $\frac{m}{\log_q l}$ times, so total cost is $O\left(\frac{lmn}{\log_q l}\right)$
## Results on Cortex M4:

<table>
<thead>
<tr>
<th></th>
<th>((P_i^{(1)} + P_i^{(1)^T})O)</th>
<th>(P_i^{(1)}V^T)</th>
<th>(P_i^{(1)}O)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sign</td>
<td>Sign</td>
<td>KeyGen</td>
</tr>
<tr>
<td>MAYO(_1)</td>
<td>bitsliced 2 165 337 (1.74 ×)</td>
<td>1 323 797 (1.18 ×)</td>
<td>1 177 752 (1.65 ×)</td>
</tr>
<tr>
<td>M4R</td>
<td>1 244 009</td>
<td>1 119 136 (1.18 ×)</td>
<td>714 332 (1.65 ×)</td>
</tr>
<tr>
<td>MAYO(_2)</td>
<td>bitsliced 5 199 607 (1.79 ×)</td>
<td>629 400</td>
<td>2 830 681</td>
</tr>
<tr>
<td>M4R</td>
<td>2 906 460</td>
<td>681 081 (0.92 ×)</td>
<td>1 683 616 (1.68 ×)</td>
</tr>
<tr>
<td>MAYO(_3)</td>
<td>bitsliced 9 535 835 (1.45 ×)</td>
<td>5 635 495</td>
<td>5 126 000</td>
</tr>
<tr>
<td>M4R</td>
<td>6 576 258</td>
<td>3 452 417 (1.63 ×)</td>
<td>3 525 668 (1.45 ×)</td>
</tr>
</tbody>
</table>

Conclusion: We can switch to Nibble-based representation and get a nice speedup on embedded platforms, as well as a huge speedup on AVX2 platforms.
Other contributions

• Improved AVX2 shuffle-based matrix multiplication
  New records: 56.5 multiply-and-accumulates / cycle (Skylake)
  78.8 multiply-and-accumulates / cycle (Ice Lake)

• Constant time Gaussian elimination for rectangular matrices.

• Read paper for more ...