Classic McEliece on the ARM Cortex-M4
(iacr/2021/492)

Ming-Shing Chen, Tung Chou

Ruhr University Bochum, Germany
Academia Sinica, Taiwan

9 June, 2021
We put the public keys in flash, the cycle counts include time to read/write pk from/to flash.

All optimizations work when streaming is used.

With a bit more effort, should be able to do key generation for mceliece460896*.

Should be able to run all operations of all parameter sets on larger M4 boards (e.g., Giant Gecko).

Encapsulation time is close to that of lattice-based finalists.

Decapsulation time is 4–7 times as slow but still reasonably efficient.

Can trade decapsulation speed for key generation speed by omitting control-bit generation.

Our implementation is constant-time.

## Cycle counts on stm32f4-discovery (at 168 MHz)

<table>
<thead>
<tr>
<th>parameter set</th>
<th>level</th>
<th>decap.</th>
<th>encap.</th>
<th>key generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mceliece348864f</td>
<td>1</td>
<td>2 706 681</td>
<td>582 199</td>
<td>1 430 811 294</td>
</tr>
<tr>
<td>mceliece348864</td>
<td>1</td>
<td>6 535 186</td>
<td>1 081 335</td>
<td>2 146 932 033</td>
</tr>
<tr>
<td>mceliece460896*</td>
<td>3</td>
<td>6 535 186</td>
<td>1 081 335</td>
<td></td>
</tr>
<tr>
<td>mceliece6688128*</td>
<td>5</td>
<td>7 412 111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mceliece8192128*</td>
<td>5</td>
<td>7 481 747</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• With a bit more effort, should be able to do key generation for mceliece460896*.
• Should be able to run all operations of all parameter sets on larger M4 boards (e.g., Giant Gecko).
• Encapsulation time is close to that of lattice-based finalists.
• Decapsulation time is 4–7 times as slow but still reasonably efficient.
• Can trade decapsulation speed for key generation speed by omitting control-bit generation.

<table>
<thead>
<tr>
<th>parameter set</th>
<th>level</th>
<th>decap.</th>
<th>encap.</th>
<th>key generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mceliece348864f</td>
<td>1</td>
<td>2 706 681</td>
<td>582 199</td>
<td>1 430 811 294</td>
</tr>
<tr>
<td>mceliece348864</td>
<td>1</td>
<td>2 706 681</td>
<td>582 199</td>
<td>2 146 932 033</td>
</tr>
<tr>
<td>mceliece460896*</td>
<td>3</td>
<td>6 535 186</td>
<td>1 081 335</td>
<td></td>
</tr>
<tr>
<td>mceliece6688128*</td>
<td>5</td>
<td>7 412 111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mceliece8192128*</td>
<td>5</td>
<td>7 481 747</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Our implementation is constant-time.
• We put the public keys in flash, the cycle counts include time to read/write pk from/to flash.
• All optimizations work when streaming is used.
## Cycle counts on stm32f4-discovery (at 168 MHz)

<table>
<thead>
<tr>
<th>parameter set</th>
<th>level</th>
<th>decap.</th>
<th>encap.</th>
<th>key generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mceliece348864f</td>
<td>1</td>
<td>2706681</td>
<td>582199</td>
<td>1430811294</td>
</tr>
<tr>
<td>mceliece348864</td>
<td>1</td>
<td>2706681</td>
<td>582199</td>
<td>2146932033</td>
</tr>
<tr>
<td>mceliece460896*</td>
<td>3</td>
<td>6535186</td>
<td>1081335</td>
<td></td>
</tr>
<tr>
<td>mceliece6688128*</td>
<td>5</td>
<td>7412111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mceliece8192128*</td>
<td>5</td>
<td>7481747</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Our implementation is constant-time.
- We put the public keys in flash, the cycle counts include time to read/write pk from/to flash.
- All optimizations work when streaming is used.
- With a bit more effort, should be able to do key generation for mceliece460896*.
- Should be able to run all operations of all parameter sets on larger M4 boards (e.g., Giant Gecko).
Cycle counts on stm32f4-discovery (at 168 MHz)

<table>
<thead>
<tr>
<th>parameter set</th>
<th>level</th>
<th>decap.</th>
<th>encap.</th>
<th>key generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>mceliece348864f</td>
<td>1</td>
<td>2706681</td>
<td>582199</td>
<td>1430811294</td>
</tr>
<tr>
<td>mceliece348864</td>
<td>1</td>
<td>2706681</td>
<td>582199</td>
<td>2146932033</td>
</tr>
<tr>
<td>mceliece460896*</td>
<td>3</td>
<td>6535186</td>
<td>1081335</td>
<td></td>
</tr>
<tr>
<td>mceliece6688128*</td>
<td>5</td>
<td>7412111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mceliece8192128*</td>
<td>5</td>
<td>7481747</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Our implementation is constant-time.
- We put the public keys in flash, the cycle counts include time to read/write pk from/to flash.
- All optimizations work when streaming is used.
- With a bit more effort, should be able to do key generation for mceliece460896*.
- Should be able to run all operations of all parameter sets on larger M4 boards (e.g., Giant Gecko).
- Encapsulation time is close to that of lattice-based finalists.
- Decapsulation time is 4–7 times as slow but still reasonably efficient.
- Can trade decapsulation speed for key generation speed by omitting control-bit generation.
Public key generation: previous implementations

• For non-f parameter sets, the task is to convert $H = [M| T]$ into $[I|M^{-1}T]$.

1. Previous AVX/SSE implementations mostly by Chou
   • supercop-20200531 and later versions.
   • 3rd-round submission package of Classic McEliece.

2. “Classic McEliece implementation with low memory footprint” by Roth, Karatsiolis and Krämer
Public key generation: previous implementations

- For non-f parameter sets, the task is to convert \( H = [M| T] \) into \([I|M^{-1}T]\).
- The implementations below
  - use almost-inplace LUP decompositions (with \( PM = LU \)) and
  - generate column blocks \( T_i \)'s on demand
to save time and space.

1. Previous AVX/SSE implementations mostly by Chou
   - supercop-20200531 and later versions.
   - 3rd-round submission package of Classic McEliece.

2. “Classic McEliece implementation with low memory footprint” by Roth, Karatsiolis and Krämer
Public key generation: previous implementations

• For non-f parameter sets, the task is to convert \( \begin{bmatrix} H \end{bmatrix} = [M | T ] \) into \([I | M^{-1}T]\).

• The implementations below
  • use almost-inplace LUP decompositions (with \( PM = LU \)) and
  • generate column blocks \( T_i \)'s on demand

1. Previous AVX/SSE implementations mostly by Chou
  • supercop-20200531 and later versions.
  • 3rd-round submission package of Classic McEliece.

\[
\begin{bmatrix} M \end{bmatrix} \rightarrow \begin{bmatrix} U_{L^{-1}} \end{bmatrix} \begin{bmatrix} P \end{bmatrix} \quad pk_i \leftarrow (U^{-1}(L^{-1}(PT_i)))
\]

2. “Classic McElierce implementation with low memory footprint” by Roth, Karatsiolis and Krämer

\[
\begin{bmatrix} M \end{bmatrix} \rightarrow \begin{bmatrix} U \end{bmatrix} \begin{bmatrix} L \end{bmatrix} \begin{bmatrix} P \end{bmatrix} \quad \text{Compute } U^{-1} \text{ and } L^{-1}, \quad M^{-1} \leftarrow U^{-1}L^{-1}P, \quad pk_i \leftarrow M^{-1}T_i
\]
Public key generation: our implementation

- (RKK) \( M \rightarrow L, U, P \)
Public key generation: our implementation

- (RKK) $M \rightarrow L, U, P$
- (C) Apply $P$ to $T_i$ using a sorting network.
  - Represent $P^{-1}$ as an array of indices $p_1, \ldots, p_{n-k}$.
  - Sort $(p_1, \text{row}_1), \ldots, (p_{n-k}, \text{row}_{n-k})$ based on $p_i$. 
Public key generation: our implementation

- (RKK) \( M \rightarrow L, U, P \)
- (C) Apply \( P \) to \( T_i \) using a sorting network.
  - Represent \( P^{-1} \) as an array of indices \( p_1, \ldots, p_{n-k} \).
  - Sort \((p_1, \text{row}_1), \ldots, (p_{n-k}, \text{row}_{n-k})\) based on \( p_i \).
- (C) Multiply by \( L^{-1} \) or \( U^{-1} \) without computing the inverse matrices.

\[
L = \begin{pmatrix}
1 & 0 & 0 \\
\ell_0 & 1 & 0 \\
\ell_1 & \ell_2 & 1
\end{pmatrix}, \quad L^{-1} = \begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & \ell_2 & 1
\end{pmatrix} \begin{pmatrix}
1 & 0 & 0 \\
1 & 0 & 0 \\
\ell_0 & 1 & 0
\end{pmatrix} \begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}.
\]
Public key generation: our implementation

- (RKK) $M \rightarrow L, U, P$
- (C) Apply $P$ to $T_i$ using a sorting network.
  - Represent $P^{-1}$ as an array of indices $p_1, \ldots, p_{n-k}$.
  - Sort $(p_1, \text{row}_1), \ldots, (p_{n-k}, \text{row}_{n-k})$ based on $p_i$.
- (C) Multiply by $L^{-1}$ or $U^{-1}$ without computing the inverse matrices.

$$L = \begin{pmatrix} 1 & 0 & 0 \\ \ell_0 & 1 & 0 \\ \ell_1 & \ell_2 & 1 \end{pmatrix}, \quad L^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \ell_2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \ell_1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ \ell_0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$
Public key generation: our implementation

- (RKK) $M \to L, U, P$
- (C) Apply $P$ to $T_i$ using a sorting network.
  - Represent $P^{-1}$ as an array of indices $p_1, \ldots, p_{n-k}$.
  - Sort $(p_1, \text{row}_1), \ldots, (p_{n-k}, \text{row}_{n-k})$ based on $p_i$.
- (C) Multiply by $L^{-1}$ or $U^{-1}$ without computing the inverse matrices.

\[
L = \begin{pmatrix} 1 & 0 & 0 \\ \ell_0 & 1 & 0 \\ \ell_1 & \ell_2 & 1 \end{pmatrix}, \quad L^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \ell_2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \ell_1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.
\]

- (new) Makes use of blocking to optimize multiplications by $L^{-1}$ and $U^{-1}$.
- We use $T_i$'s with 32/640 columns.
- Our implementation and (C) both support f parameter sets and decapsulation, while (RKK) does not.
Encapsulation

• Generation of the weight-$t$ error vector $e$

• Matrix vector product $[I\ |\ pk] \cdot e^T$
Encapsulation

- Generation of the weight-$t$ error vector $e$
  - Specification (roughly): generate an array of $t$ indices of 1's in $e$.
  - We sort the indices to check for repetition. Sorting networks are safe.
  - Observation: information of $e$ only lies in the set of indices.
  - Actually any comparison-based sorting algorithm can be used: we use quicksort.
  - Might be useful for other code-based cryptosystems (e.g., BIKE and HQC).

- Matrix vector product $\begin{bmatrix} l & pk \end{bmatrix} \cdot e^T$
Encapsulation

• Generation of the weight-$t$ error vector $e$
  • Specification (roughly): generate an array of $t$ indices of 1's in $e$.
  • We sort the indices to check for repetition. Sorting networks are safe.
  • Observation: information of $e$ only lies in the set of indices.
  • Actually any comparison-based sorting algorithm can be used: we use quicksort.
  • Might be useful for other code-based cryptosystems (e.g., BIKE and HQC).

• Matrix vector product $[l \  pk] \cdot e^T$
  • Want to reduce the number of memory accesses.
  • Divide $pk$ into $4 \times 96$ blocks so that each piece of $e$ can be reused.

$$
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
\text{Blue} \\
\text{Red} \\
\text{Green} \\
\text{Purple}
\end{bmatrix}
$$
https://github.com/pqcryptotw/mceliece-arm-m4