Code-based crypto for small servers

Tanja Lange

Technische Universiteit Eindhoven

SIAM Conference on Applied Algebraic Geometry
13 July 2019
Code-based encryption

- **1971 Goppa**: Fast decoders for many matrices $H$.
- **1978 McEliece**: Use Goppa codes for public-key crypto.
  - Original parameters designed for $2^{64}$ security.
  - 2008 Bernstein–Lange–Peters: broken in $\approx 2^{60}$ cycles.
  - Easily scale up for higher security.
- **1986 Niederreiter**: Simplified and smaller version of McEliece.
- **1962 Prange**: Simple attack idea guiding sizes in 1978 McEliece.

The McEliece system (with later key-size optimizations)
uses $(c_0 + o(1))\lambda^2(\lg \lambda)^2$-bit keys as $\lambda \to \infty$
to achieve $2^\lambda$ security against Prange’s attack.
Here $c_0 \approx 0.7418860694$. 
Some papers studying algorithms for attackers:
Consequence of security analysis

- The McEliece system (with later key-size optimizations) uses $(c_0 + o(1))\lambda^2(\lg \lambda)^2$-bit keys as $\lambda \to \infty$ to achieve $2^\lambda$ security against all these attacks.
Consequence of security analysis

- The McEliece system (with later key-size optimizations) uses \((c_0 + o(1))\lambda^2(\lg \lambda)^2\)-bit keys as \(\lambda \to \infty\) to achieve \(2^\lambda\) security against all these attacks. Here \(c_0 \approx 0.7418860694\).
- 256 KB public key for \(2^{146}\) pre-quantum security.
- 512 KB public key for \(2^{187}\) pre-quantum security.
- 1024 KB public key for \(2^{263}\) pre-quantum security.
Consequence of security analysis

- The McEliece system (with later key-size optimizations) uses \((c_0 + o(1))\lambda^2(\lg \lambda)^2\)-bit keys as \(\lambda \to \infty\) to achieve \(2^\lambda\) security against all these attacks. Here \(c_0 \approx 0.7418860694\).
- 256 KB public key for \(2^{146}\) pre-quantum security.
- 512 KB public key for \(2^{187}\) pre-quantum security.
- 1024 KB public key for \(2^{263}\) pre-quantum security.
- Post-quantum (Grover): below \(2^{263}\), above \(2^{131}\).
The Niederreiter cryptosystem

Developed in 1986 by Harald Niederreiter as a variant of the McEliece cryptosystem. This is the schoolbook version.

- Use $n \times n$ permutation matrix $P$ and $n - k \times n - k$ invertible matrix $S$.
- Public Key: a scrambled parity-check matrix $K = SHP \in \mathbb{F}_2^{(n-k)\times n}$.
- Encryption: The plaintext $e$ is an $n$-bit vector of weight $t$. The ciphertext $s$ is the $(n - k)$-bit vector $s = Ke$.
- Decryption: Find a $n$-bit vector $e$ with $wt(e) = t$ such that $s = Ke$.
- The passive attacker is facing a $t$-error correcting problem for the public key, which seems to be random.
The Niederreiter cryptosystem II

- Public Key: a scrambled parity-check matrix $K = SHP$.
- Encryption: The plaintext $e$ is an $n$-bit vector of weight $t$. The ciphertext $s$ is the $(n - k)$-bit vector

$$s = Ke.$$

- Decryption using secret key: Compute

$$S^{-1}s = S^{-1}Ke = S^{-1}(SHP)e = H(Pe)$$

and observe that $\text{wt}(Pe) = t$, because $P$ permutes. Use efficient syndrome decoder for $H$ to find $e' = Pe$ and thus $e = P^{-1}e'$. 

Note on codes

- McEliece proposed to use binary Goppa codes. These are still used today.
- Niederreiter described his scheme using Reed-Solomon codes. These were broken in 1992 by Sidelnikov and Chestakov.
- More corpses on the way: concatenated codes, Reed-Muller codes, several Algebraic Geometry (AG) codes, Gabidulin codes, several LDPC codes, cyclic codes.
- Some other constructions look OK (for now). NIST competition has several entries on QCMDPC codes.
**Binary Goppa code**

Let $q = 2^m$. A binary Goppa code is often defined by

- a list $L = (a_1, \ldots, a_n)$ of $n$ distinct elements in $\mathbb{F}_q$, called the **support**.

- a square-free polynomial $g(x) \in \mathbb{F}_q[x]$ of degree $t$ such that $g(a) \neq 0$ for all $a \in L$. $g(x)$ is called the **Goppa polynomial**.

- E.g. choose $g(x)$ irreducible over $\mathbb{F}_q$.

The corresponding binary Goppa code $\Gamma(L, g)$ is

$$\left\{ c \in \mathbb{F}_2^n \left| S(c) = \frac{c_1}{x - a_1} + \frac{c_2}{x - a_2} + \cdots + \frac{c_n}{x - a_n} \equiv 0 \mod g(x) \right. \right\}$$

- This code is linear $S(b + c) = S(b) + S(c)$ and has length $n$.

- Bounds on dimension $k \geq n - mt$ and minimum distance $t \geq 2t + 1$. 

Reminder: How to hide nice code?

- Do not reveal matrix $H$ related to nice-to-decode code.
- Pick a random invertible $(n - k) \times (n - k)$ matrix $S$ and random $n \times n$ permutation matrix $P$. Put
  \[ K = SHP. \]
- $K$ is the public key and $S$ and $P$ together with a decoding algorithm for $H$ form the private key.
- For suitable codes $K$ looks like random matrix.
Reminder: How to hide nice code?

- Do not reveal matrix $H$ related to nice-to-decode code.
- Pick a random invertible $(n - k) \times (n - k)$ matrix $S$ and random $n \times n$ permutation matrix $P$. Put

  $$K = SHP.$$ 

- $K$ is the public key and $S$ and $P$ together with a decoding algorithm for $H$ form the private key.
- For suitable codes $K$ looks like random matrix.
- For Goppa code use secret polynomial $g(x)$.
- Use secret permutation of the $a_i$; this corresponds to secret permutation of the $n$ positions; this replaces $P$.
- Use systematic form $K = (K'|I)$ for key;
  - This implicitly applies $S$.
  - No need to remember $S$ because decoding does not use $H$.
  - Public key size decreased to $(n - k) \times k$.
- Secret key is polynomial $g$ and support $L = (a_1, \ldots, a_n)$. 
NIST submission Classic McEliece

- Security asymptotics unchanged by 40 years of cryptanalysis.
- Efficient and straightforward conversion 
  OW-CPA PKE $\rightarrow$ IND-CCA2 KEM.
- Open-source (public domain) implementations.
  - Constant-time software implementations.
  - FPGA implementation of full cryptosystem.
- No patents.

<table>
<thead>
<tr>
<th>Metric</th>
<th>mceliece6960119</th>
<th>mceliece8192128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public-key size</td>
<td>1047319 bytes</td>
<td>1357824 bytes</td>
</tr>
<tr>
<td>Secret-key size</td>
<td>13908 bytes</td>
<td>14080 bytes</td>
</tr>
<tr>
<td>Ciphertext size</td>
<td>226 bytes</td>
<td>240 bytes</td>
</tr>
<tr>
<td>Key-generation time</td>
<td>1108833108 cycles</td>
<td>1173074192 cycles</td>
</tr>
<tr>
<td>Encapsulation time</td>
<td>153940 cycles</td>
<td>188520 cycles</td>
</tr>
<tr>
<td>Decapsulation time</td>
<td>318088 cycles</td>
<td>343756 cycles</td>
</tr>
</tbody>
</table>

See [https://classic.mceliece.org](https://classic.mceliece.org) for more details. More parameters in round 2.
Key issues for McEliece

- Very conservative system, expected to last; has strongest security track record.
- Ciphertexts are among the shortest.
- Secret keys can be compressed.
- But public keys are really, really big!
- Sending 1MB takes time and bandwidth.

Google–Cloudflare experiment: in some cases the public-key + ciphertext size was too large to be viable in the context of TLS and even 10KB messages dropped. If server accepts 1MB of public key from any client, an attacker can easily flood memory. This invites DoS attacks.
Key issues for McEliece

- Very conservative system, expected to last; has strongest security track record.
- Ciphertexts are among the shortest.
- Secret keys can be compressed.
- But public keys are really, really big!
- Sending 1MB takes time and bandwidth.
- Google–Cloudflare experiment:

  *in some cases the public-key + ciphertext size was too large to be viable in the context of TLS*

  and even 10KB messages dropped.
Key issues for McEliece

▶ Very conservative system, expected to last; has strongest security track record.
▶ Ciphertexts are among the shortest.
▶ Secret keys can be compressed.
▶ But public keys are really, really big!
▶ Sending 1MB takes time and bandwidth.
▶ Google–Cloudlare experiment:

\[
\text{in some cases the public-key + ciphertext size was too large to be viable in the context of TLS}
\]

and even 10KB messages dropped.
▶ If server accepts 1MB of public key from any client, an attacker can easily flood memory. This invites DoS attacks.
Goodness, what big keys you have!

- Public keys look like this:

$$K = \begin{pmatrix}
1 & 0 & \ldots & 0 & 1 & \ldots & 1 & 0 & 1 \\
0 & 1 & \ldots & 0 & 0 & \ldots & 0 & 1 & 1 \\
\vdots & \vdots & \ddots & \vdots & 1 & \ldots & 1 & 1 & 0 \\
0 & 0 & \ldots & 1 & 0 & \ldots & 1 & 1 & 1
\end{pmatrix}$$

Left part is \((n - k) \times (n - k)\) identity matrix (no need to send) right part is random-looking \((n - k) \times k\) matrix.

E.g. \(n = 6960\), \(k = 5413\), so \(n - k = 1547\).
Goodness, what big keys you have!

- Public keys look like this:

\[ K = \begin{pmatrix}
1 & 0 & \ldots & 0 & 1 & \ldots & 1 & 0 & 1 \\
0 & 1 & \ldots & 0 & 0 & \ldots & 0 & 1 & 1 \\
\vdots & \vdots & \ddots & \vdots & 1 & \ldots & 1 & 1 & 0 \\
0 & 0 & \ldots & 1 & 0 & \ldots & 1 & 1 & 1
\end{pmatrix} \]

Left part is \((n - k) \times (n - k)\) identity matrix (no need to send)
right part is random-looking \((n - k) \times k\) matrix.
E.g. \(n = 6960, k = 5413\), so \(n - k = 1547\).

- Encryption xors secretly selected columns, e.g.

\[
\begin{pmatrix}
0 \\
1 \\
0 \\
0
\end{pmatrix} + \begin{pmatrix}
0 \\
1 \\
0 \\
0
\end{pmatrix} + \begin{pmatrix}
1 \\
0 \\
1 \\
1
\end{pmatrix} + \begin{pmatrix}
1 \\
1 \\
0 \\
0
\end{pmatrix} = \begin{pmatrix}
0 \\
1 \\
0 \\
0
\end{pmatrix}
\]
Can servers avoid storing big keys?

\[
K = \begin{pmatrix}
1 & 0 & \ldots & 0 & 1 & \ldots & 1 & 0 & 1 \\
0 & 1 & \ldots & 0 & 0 & \ldots & 0 & 1 & 1 \\
\vdots & \vdots & \ddots & \vdots & 1 & \ldots & 1 & 1 & 0 \\
0 & 0 & \ldots & 1 & 0 & \ldots & 1 & 1 & 1
\end{pmatrix} = (I_{n-k}|K')
\]

- Encryption xors secretly selected columns.
- With some storage and trusted environment:
  Receive columns of \( K' \) one at a time, store and update partial sum.
Can servers avoid storing big keys?

\[
K = \begin{pmatrix}
1 & 0 & \ldots & 0 & 1 & \ldots & 1 & 0 & 1 \\
0 & 1 & \ldots & 0 & 0 & \ldots & 0 & 1 & 1 \\
\vdots & \vdots & \ddots & \vdots & 1 & \ldots & 1 & 1 & 0 \\
0 & 0 & \ldots & 1 & 0 & \ldots & 1 & 1 & 1
\end{pmatrix} = (I_{n-k} | K')
\]

- Encryption xors secretly selected columns.
- With some storage and trusted environment:
  Receive columns of \( K' \) one at a time, store and update partial sum.
- On the real Internet, without per-client state:
Can servers avoid storing big keys?

Encryption xors secretly selected columns.

With some storage and trusted environment:
Receive columns of $K'$ one at a time, store and update partial sum.

On the real Internet, without per-client state:
Don’t reveal intermediate results!
Which columns are picked is the secret message!
Intermediate results show whether a column was used or not.
McTiny (Bernstein/Lange)

Partition key

\[ K' = \begin{pmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \ldots & K_{1,\ell} \\ K_{2,1} & K_{2,2} & K_{2,3} & \ldots & K_{2,\ell} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ K_{r,1} & K_{r,2} & K_{r,3} & \ldots & K_{r,\ell} \end{pmatrix} \]

- Each submatrix \( K_{i,j} \) small enough to fit into network packet (plus some extra).
- Client feeds the \( K_{i,j} \) to server & handles storage for the server.
- Server computes \( K_{i,j}e_j \), puts result into cookie.
- Cookies are encrypted by server to itself using some temporary symmetric key (same key for all server connections).
  No per-client memory allocation.
- Cookies also encrypted & authenticated to client.
- Client sends several \( K_{i,j}e_j \) cookies, receives their combination.
- More stuff to avoid replay & similar attacks.
McTiny (Bernstein/Lange)

Partition key

\[
K' = \begin{pmatrix}
K_{1,1} & K_{1,2} & K_{1,3} & \ldots & K_{1,\ell} \\
K_{2,1} & K_{2,2} & K_{2,3} & \ldots & K_{2,\ell} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
K_{r,1} & K_{r,2} & K_{r,3} & \ldots & K_{r,\ell}
\end{pmatrix}
\]

- Each submatrix \( K_{i,j} \) small enough to fit into network packet (plus some extra).
- Client feeds the \( K_{i,j} \) to server & handles storage for the server.
- Server computes \( K_{i,j}e_j \), puts result into cookie.
- Cookies are encrypted by server to itself using some temporary symmetric key (same key for all server connections).
  No per-client memory allocation.
- Cookies also encrypted & authenticated to client.
- Client sends several \( K_{i,j}e_j \) cookies, receives their combination.
- More stuff to avoid replay & similar attacks.
- Several round trips, but no per-client state on the server.