# McTiny: Fast High-Confidence Post-Quantum Key Erasure for Tiny Network Servers

Daniel J. Bernstein[1,2] and Tanja Lange[3]

[1]University of Illinois at Chicago

[2]Ruhr University Bochum

[3]Eindhoven University of Technology

USENIX Security 2020

# Post-quantum cryptography

Cryptography designed under the assumption that the **attacker** (not the user!) has a large quantum computer.

Options: code-based, hash-based, isogeny-based, lattice-based, multivariates.

1978 McEliece: Public-key encryption using error-correcting codes.

- Original parameters designed for $2^{64}$ security.
- 2008 Bernstein–Lange–Peters: broken in $\approx 2^{60}$ cycles.
- Easily scale up for higher security.
- 1962 Prange: simple attack idea guiding sizes in 1978 McEliece. The McEliece system (with later key-size optimizations) achieves $2^\lambda$ security against Prange's attack using $(0.741186\ldots + o(1))\lambda^2(\log_2 \lambda)^2$-bit keys as $\lambda \to \infty$.

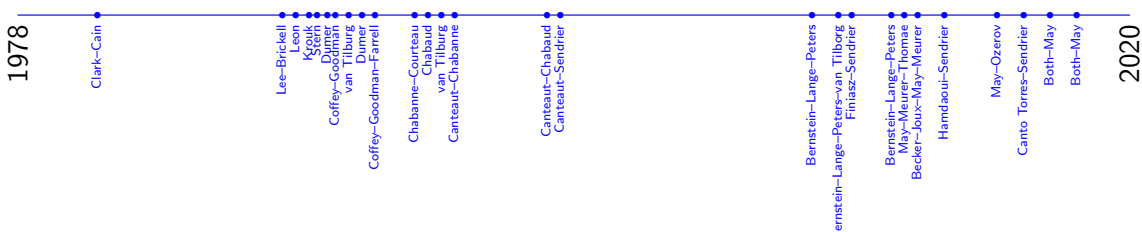
# Security analysis of McEliece encryption

Some papers studying algorithms for attackers:
1962 Prange; 1981 Clark–Cain, crediting Omura; 1988 Lee–Brickell; 1988 Leon; 1989 Krouk;
1989 Stern; 1989 Dumer; 1990 Coffey–Goodman; 1990 van Tilburg; 1991 Dumer;
1991 Coffey–Goodman–Farrell; 1993 Chabanne–Courteau; 1993 Chabaud; 1994 van Tilburg;
1994 Canteaut–Chabanne; 1998 Canteaut–Chabaud; 1998 Canteaut–Sendrier;
2008 Bernstein–Lange–Peters; 2009 Bernstein–Lange–Peters–van Tilborg;
2009 Bernstein (**post-quantum**); 2009 Finiasz–Sendrier; 2010 Bernstein–Lange–Peters;
2011 May–Meurer–Thomae; 2012 Becker–Joux–May–Meurer; 2013 Hamdaoui–Sendrier;
2015 May–Ozerov; 2016 Canto Torres–Sendrier; 2017 Kachigar–Tillich (**post-quantum**);
2017 Both–May; 2018 Both–May; 2018 Kirshanova (**post-quantum**).

All of these attacks involve huge searches, like attacking AES.
The quantum attacks (Grover etc.) leave at least half of the bits of security.

# Attack progress over time

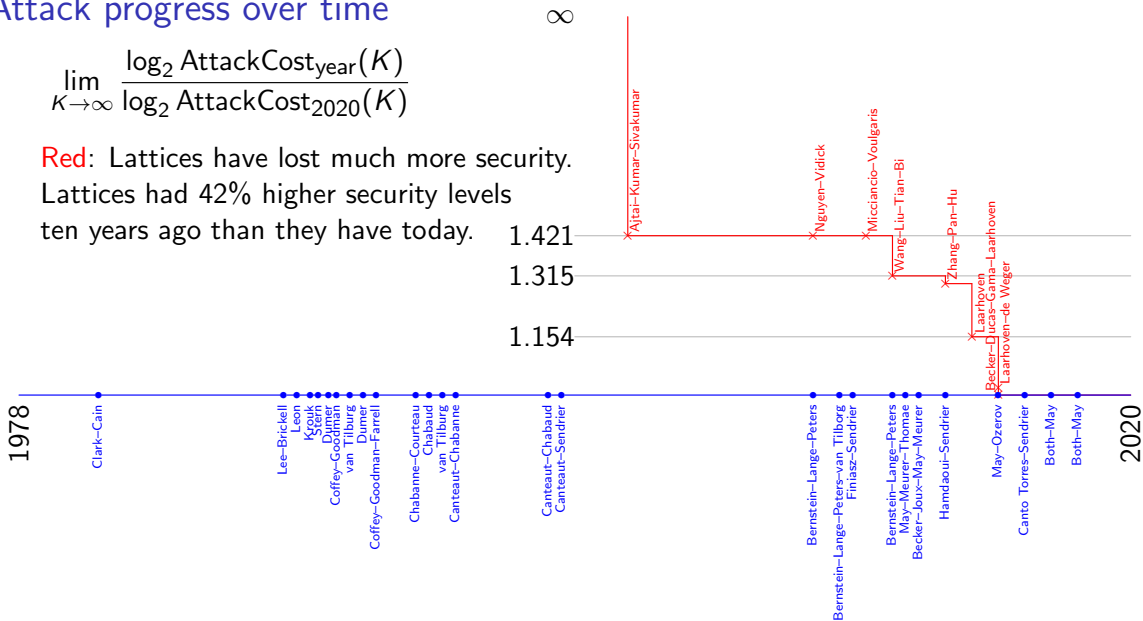
$$\lim_{K\to\infty} \frac{\log_2 \mathsf{AttackCost}_{\mathsf{year}}(K)}{\log_2 \mathsf{AttackCost}_{2020}(K)}$$



# Attack progress over time

$$\lim_{K\to\infty} \frac{\log_2 \mathrm{AttackCost}_{\mathrm{year}}(K)}{\log_2 \mathrm{AttackCost}_{2020}(K)}$$

Red: Lattices have lost much more security. Lattices had 42% higher security levels ten years ago than they have today.



# NIST PQC submission Classic McEliece

No patents.
Shortest ciphertexts.
Fast open-source constant-time software implementations.
Very conservative system, expected to last; has strongest security track record.

Sizes with similar post-quantum security to AES-128, AES-192, AES-256:

| **Metric** | **mceliece348864** | **mceliece460896** | **mceliece6960119** |
|---|---|---|---|
| Public-key size | 261120 bytes | 524160 bytes | 1047319 bytes |
| Secret-key size | 6452 bytes | 13568 bytes | 13908 bytes |
| Ciphertext size | 128 bytes | 188 bytes | 226 bytes |
| Key-generation time | 52415436 cycles | 181063400 cycles | 417271280 cycles |
| Encapsulation time | 43648 cycles | 77380 cycles | 143908 cycles |
| Decapsulation time | 130944 cycles | 267828 cycles | 295628 cycles |

See https://classic.mceliece.org for authors, details & parameters.

# Key issues for McEliece

## Key issues for McEliece

# BIG PUBLIC KEYS.

# Key issues for McEliece

Users send big data anyway. We have lots of bandwidth. Maybe 1MB keys are okay.

Each client spends a small fraction of a second generating new ephemeral 1MB key.

# Key issues for McEliece

Users send big data anyway. We have lots of bandwidth. Maybe 1MB keys are okay.

Each client spends a small fraction of a second generating new ephemeral 1MB key.

But: If any client is allowed to send a new ephemeral 1MB McEliece key to server, an attacker can easily flood server's memory. **This invites DoS attacks.**

# Key issues for McEliece

Users send big data anyway. We have lots of bandwidth. Maybe 1MB keys are okay.

Each client spends a small fraction of a second generating new ephemeral 1MB key.

But: If any client is allowed to send a new ephemeral 1MB McEliece key to server, an attacker can easily flood server's memory. **This invites DoS attacks.**

Our goal: Eliminate these attacks by eliminating all per-client storage on server.

# Goodness, what big keys you have!

Public keys look like this:

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

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:

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

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.

$$\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

# Can servers avoid storing big keys?

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

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?

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

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?

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

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:
Don't reveal intermediate results!
Which columns are picked is the secret message!
Intermediate results show whether a column was used or not.

# McTiny

Partition key

$$K' = \begin{pmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \dots & K_{1,\ell} \\ K_{2,1} & K_{2,2} & K_{2,3} & \dots & K_{2,\ell} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ K_{r,1} & K_{r,2} & K_{r,3} & \dots & K_{r,\ell} \end{pmatrix}$$

- Each submatrix $K_{i,j}$ small enough to fit (including header) into network packet.
- 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.

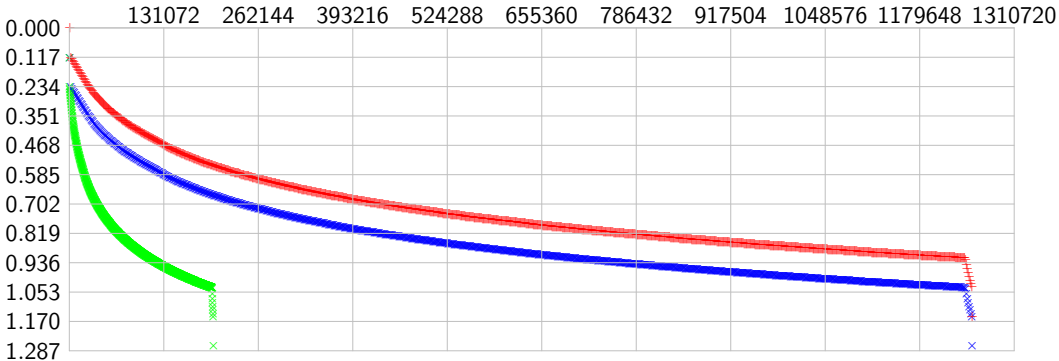
# McTiny

Partition key

$$K' = \begin{pmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \dots & K_{1,\ell} \\ K_{2,1} & K_{2,2} & K_{2,3} & \dots & K_{2,\ell} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ K_{r,1} & K_{r,2} & K_{r,3} & \dots & K_{r,\ell} \end{pmatrix}$$

- Each submatrix $K_{i,j}$ small enough to fit (including header) into network packet.
- 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.

# Measurements of our software (https://mctiny.org)




Client time vs. bytes sent, bytes acknowledged, bytes in acknowledgments.
Curve shows packet pacing from our new user-level congestion-control library.