McEliece verification

Daniel J. Bernstein

“Who cares? Big keys are unusable!”

“Who cares? Big keys are unusable!”

Let’s look at the facts:

- 1MB is very fast on a modern network.
  Are Netflix and YouTube unusable?

“Who cares? Big keys are unusable!”

Let’s look at the facts:

- 1MB is very fast on a modern network. Are Netflix and YouTube unusable?
- Google’s key can be used to protect any number of ciphertexts to/from Google.

“Who cares? Big keys are unusable!”

Let’s look at the facts:

- 1MB is very fast on a modern network. Are Netflix and YouTube unusable?
- Google’s key can be used to protect any number of ciphertexts to/from Google.
- 1 key + 10^6 ciphertexts for McEliece is several times less network traffic than 1 key + 10^6 ciphertexts for lattices.

“Who cares? Big keys are unusable!”

Let’s look at the facts:

- 1MB is very fast on a modern network.
  Are Netflix and YouTube unusable?
- Google’s key can be used to protect any number of ciphertexts to/from Google.
- 1 key + 10^6 ciphertexts for McEliece is several times less network traffic than 1 key + 10^6 ciphertexts for lattices.
- McEliece deployment is underway: e.g., McEliece is already used in some end-to-end secure-messaging systems and the Mullvad and Rosenpass VPNs.

McEliece security advantages

QROM IND-CCA2 security of Classic McEliece
has tight proof assuming one-wayness
of the original 1978 McEliece system.
Stable attack target for 45 years.

McEliece security advantages

QROM IND-CCA2 security of Classic McEliece has tight proof assuming one-wayness of the original 1978 McEliece system. **Stable attack target for 45 years.**

Nearly 50 papers attacking one-wayness of McEliece have produced only minor attack speedups since 1978: **asymptotically 0% change in pre-quantum security levels.** Post-quantum: like AES. The attack surface is thoroughly explored and well understood.

McEliece security advantages

QROM IND-CCA2 security of Classic McEliece has tight proof assuming one-wayness of the original 1978 McEliece system. **Stable attack target for 45 years.**

Nearly 50 papers attacking one-wayness of McEliece have produced only minor attack speedups since 1978: **asymptotically 0% change in pre-quantum security levels.** Post-quantum: like AES. The attack surface is thoroughly explored and well understood.

New: CryptAttackTester includes full attack circuits + analyses passing systematic tests.

McEliece attack challenges

Classic McEliece parameter selections use "lengths" 3488, 4608, 6688, 6960, 8192.

McEliece attack challenges

Classic McEliece parameter selections use “lengths” 3488, 4608, 6688, 6960, 8192.

Latest records in scaled-down challenges:

- Length-1284 challenge broken as title of a Eurocrypt 2022 paper.

McEliece attack challenges

Classic McEliece parameter selections use “lengths” 3488, 4608, 6688, 6960, 8192.

Latest records in scaled-down challenges:

- Length-1284 challenge broken as title of a Eurocrypt 2022 paper.
- Length-1347 challenge broken: simply ran the faster attack software from PQCrypto 2008 Bernstein–Lange–Peters on a larger computer cluster.

McEliece attack challenges

Classic McEliece parameter selections use "lengths" 3488, 4608, 6688, 6960, 8192.

Latest records in scaled-down challenges:

- Length-1284 challenge broken as title of a Eurocrypt 2022 paper.
- Length-1347 challenge broken: simply ran the faster attack software from PQCrypto 2008 Bernstein–Lange–Peters on a larger computer cluster.

Observed speeds match algorithm analyses. Security levels are remarkably stable.

Classic McEliece implementations

Official software for Classic McEliece is distributed via SUPERCOP benchmarking framework. Four implementations for each parameter set, all passing TIMECOP:

- `ref`: portable, prioritizing simplicity.
- `vec`: portable, 64-bit vectorization.
- `sse`: Intel/AMD, 128-bit vectorization.
- `avx`: Intel/AMD, 256-bit vectorization.

Unofficial implementations: M4, FPGAs, McTiny, McOutsourcing, Bouncy Castle (Java and C#), Rust. Integrations: PQClean, liboqs, Node.js. New: Easy-to-use libmceliece.

Checklist for software verification

Want to verify that each internal operation works correctly for all possible inputs:

- SHAKE256 on constant-length inputs.

Checklist for software verification

Want to verify that each internal operation works correctly for all possible inputs:

- SHAKE256 on constant-length inputs.
- Sorting integer arrays in constant time.

Checklist for software verification

Want to verify that each internal operation works correctly for all possible inputs:

- SHAKE256 on constant-length inputs.
- Sorting integer arrays in constant time.
- “Control bits” for permutations.

Checklist for software verification

Want to verify that each internal operation works correctly for all possible inputs:

- SHAKE256 on constant-length inputs.
- Sorting integer arrays in constant time.
- “Control bits” for permutations.
- Arithmetic in binary fields (e.g., $\mathbb{F}_{8192}$).

Checklist for software verification

Want to verify that each internal operation works correctly for all possible inputs:

- SHAKE256 on constant-length inputs.
- Sorting integer arrays in constant time.
- “Control bits” for permutations.
- Arithmetic in binary fields (e.g., $\mathbb{F}_{8192}$).
- Constant-time row reduction of matrices.

Checklist for software verification

Want to verify that each internal operation works correctly for all possible inputs:

- SHAKE256 on constant-length inputs.
- Sorting integer arrays in constant time.
- “Control bits” for permutations.
- Arithmetic in binary fields (e.g., $\mathbb{F}_{8192}$).
- Constant-time row reduction of matrices.
- Constant-time decoding of Goppa codes.

Checklist for software verification

Want to verify that each internal operation works correctly for all possible inputs:

- SHAKE256 on constant-length inputs.
- Sorting integer arrays in constant time.
- "Control bits" for permutations.
- Arithmetic in binary fields (e.g., $\mathbb{F}_{8192}$).
- Constant-time row reduction of matrices.
- Constant-time decoding of Goppa codes.

Plus: Put everything together into "keygen, enc, dec always work". Automate the entire process to handle many implementations.

Verified constant-time sorting

Want secret permutation of $\{0, 1, \ldots, 8191\}$.
Solution: sort 8192 secret 32-bit integers and their indices; restart if there are collisions.

Verified constant-time sorting

Want secret permutation of $\{0, 1, \ldots, 8191\}$. Solution: sort 8192 secret 32-bit integers and their indices; restart if there are collisions.

`saferewrite` from `pqsrc.cr.yp.to` automatically verifies constant-time min/max code (and more). Relies on `angr`, which uses VEX for code unrolling, Z3 for SMT solving.

Verified constant-time sorting

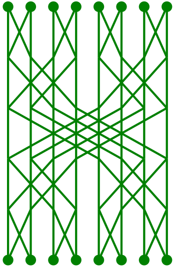
Want secret permutation of $\{0, 1, \ldots, 8191\}$.
Solution: sort 8192 secret 32-bit integers and their indices; restart if there are collisions.

`saferewrite` from `pqsrc.cr.yp.to` automatically verifies constant-time min/max code (and more). Relies on `angr`, which uses VEX for code unrolling, Z3 for SMT solving.

`sorting.cr.yp.to` includes fast constant-time *N*-input sorting built from min/max ("sorting networks") for `int32`; automated verif with `angr` + DAG analysis.
Classic McEliece also uses `int16`, `int64`.

Verified formulas for control bits

Can permute 8192 items in constant time via sorting. Simpler, faster: “Control bits” specify

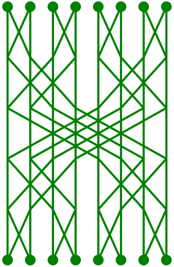


- swap 0 with 1? swap 2 with 3? etc.;
- swap 0 with 2? swap 1 with 3? etc.;
- swap 0 with 4? swap 1 with 5? etc.;
- and so on: $1, 2, 4, 8, \ldots, 8, 4, 2, 1$.

This pattern is a “Beneš network”.

Verified formulas for control bits

Can permute 8192 items in constant time via sorting. Simpler, faster: "Control bits" specify



- swap 0 with 1? swap 2 with 3? etc.;
- swap 0 with 2? swap 1 with 3? etc.;
- swap 0 with 4? swap 1 with 5? etc.;
- and so on: $1, 2, 4, 8, \ldots, 8, 4, 2, 1$.

This pattern is a "Beneš network".

cr.yp.to/papers.html#controlbits
presents a proof of fast formulas mapping any given permutation *to* control bits.
Proof is computer-verified using HOL Light.

Verified formulas for decoding

`mceliece8192128` secrets: deg-128 irred poly $g \in \mathbb{F}_{8192}[x]$; distinct $s_0, \ldots, s_{8191} \in \mathbb{F}_{8192}$.

Verified formulas for decoding

`mceliece8192128` secrets: deg-128 irred poly $g \in \mathbb{F}_{8192}[x]$; distinct $s_0, \ldots, s_{8191} \in \mathbb{F}_{8192}$.

"Goppa codeword": bits $c_0, \ldots, c_{8191}$ with $\sum_i c_i s_i^d / g(s_i) = 0$ for each $d \in \{0, 1, \ldots, 127\}$.

Verified formulas for decoding

`mceliece8192128` secrets: deg-128 irred poly $g \in \mathbb{F}_{8192}[x]$; distinct $s_0, \ldots, s_{8191} \in \mathbb{F}_{8192}$.

"Goppa codeword": bits $c_0, \ldots, c_{8191}$ with $\sum_i c_i s_i^d / g(s_i) = 0$ for each $d \in \{0, 1, \ldots, 127\}$.

"Goppa decoding": recover a codeword, given the codeword with $\leq$128 bits flipped. (The most complicated step in McEliece dec.)

Verified formulas for decoding

`mceliece8192128` secrets: deg-128 irred poly $g \in \mathbb{F}_{8192}[x]$; distinct $s_0, \ldots, s_{8191} \in \mathbb{F}_{8192}$.

"Goppa codeword": bits $c_0, \ldots, c_{8191}$ with $\sum_i c_i s_i^d / g(s_i) = 0$ for each $d \in \{0, 1, \ldots, 127\}$.

"Goppa decoding": recover a codeword, given the codeword with $\leq$128 bits flipped. (The most complicated step in McEliece dec.)

`cr.yp.to/papers.html#goppadecoding`: minicourse on decoding formulas used in the Classic McEliece software. New: Proofs are computer-verified in HOL Light and Lean.

The end is in sight

What I'm working on: More code-analysis tools, automatically matching up stages in the Classic McEliece keygen/enc/dec specification to segments of machine code.

HOL Light already includes a model of basic machine instructions; `angr` already includes a model of instructions through AVX2.

Binary-field mult is challenging to optimize, but the optimized code is easy to verify: simply trace bilinear operations on bits.