The impact of post-quantum bugs

Daniel J. Bernstein

Crypto software has many exploitable bugs

2021 Blessing–Specter–Weitzner studied hundreds of vulnerability announcements (CVEs) issued 2010–2020 for 8 well-known crypto libraries.

Crypto software has many exploitable bugs

2021 Blessing–Specter–Weitzner studied hundreds of vulnerability announcements (CVEs) issued 2010–2020 for 8 well-known crypto libraries.

Some important lessons:

- About 1 CVE per 1000 lines of code.

Crypto software has many exploitable bugs

2021 Blessing–Specter–Weitzner studied hundreds of vulnerability announcements (CVEs) issued 2010–2020 for 8 well-known crypto libraries.

Some important lessons:

- About 1 CVE per 1000 lines of code.
- Each bug stays in code for 5 years *on average*.

Crypto software has many exploitable bugs

2021 Blessing–Specter–Weitzner studied hundreds of vulnerability announcements (CVEs) issued 2010–2020 for 8 well-known crypto libraries.

Some important lessons:

- About 1 CVE per 1000 lines of code.
- Each bug stays in code for 5 years *on average*.
- About 1/8 of "cryptographic" CVEs are severe.

Post-quantum software isn't an exception

My evaluation, October 2018: "NISTPQC ... has produced the largest regression *ever* in the quality of cryptographic software. This will not be easy to fix."

Post-quantum software isn't an exception

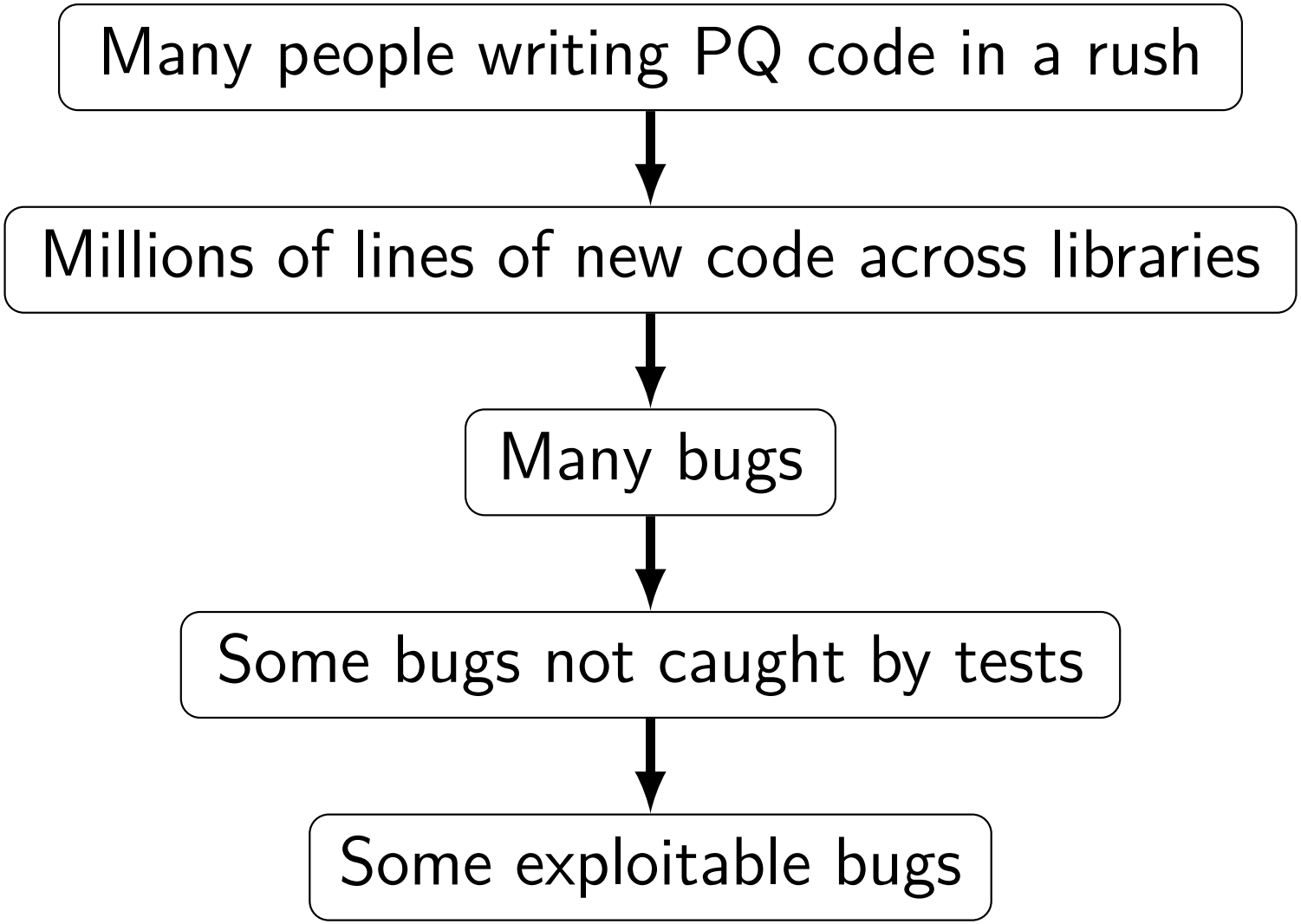
My evaluation, October 2018: "NISTPQC ... has produced the largest regression *ever* in the quality of cryptographic software. This will not be easy to fix."

Examples of bug announcements in PQ software:

2018 Dilithium	2019 Picnic	2024 SMAUG-T
2018 SPHINCS+	2019 qTESLA	2024 ML-DSA
2018 NTRU	2023 HAETAE	2026 ML-KEM
2019 Falcon	2023 PERK	2026 ML-DSA
2019 Picnic	2024 PALOMA	2026 ML-KEM
2019 NTRU	2024 NCC-Sign	2026 ML-DSA

Also many announcements of timing variations.

The PQ code ecosystem is getting worse



Case study: Dilithium (ML-DSA)

The first announced Dilithium bug

"The CRYSTALS Team" (Dilithium+Kyber), 2018:

"We are very grateful to Peter Pessl for notifying us of an implementation error in a randomness generator of our NIST submission. ... the result of the bug was that the same randomness ended up being used for pairs of consecutive coefficients ... This reuse of randomness can easily be exploited to recover the secret key and we thus emphasize that the software, in the state submitted to NIST, should not be used in any real application."

One way to make this mistake yourself

Fixed `poly.c` obtains random `t0`, `t1`. Then:

```
if(t0 <= 2*GAMMA1 - 2)
  a[ctr++] = Q + GAMMA1 - 1 - t0;
if(t1 <= 2*GAMMA1 - 2 && ctr < len)
  a[ctr++] = Q + GAMMA1 - 1 - t1;
```

Imagine typing `t1` instead of `t0`. Less random than it should be, but passes interoperability tests!

One way to make this mistake yourself

Fixed `poly.c` obtains random `t0`, `t1`. Then:

```
if(t0 <= 2*GAMMA1 - 2)
  a[ctr++] = Q + GAMMA1 - 1 - t0;
if(t1 <= 2*GAMMA1 - 2 && ctr < len)
  a[ctr++] = Q + GAMMA1 - 1 - t1;
```

Imagine typing `t1` instead of `t0`. Less random than it should be, but passes interoperability tests!

(Original code actually used just one variable `t`.
Code intent: random `t`, use `t`, random `t`, use `t`.
Actual code: random `t`, random `t`, use `t`, use `t`.)

ML-DSA: new name, same fragility

I wrote a casual implementation of ML-DSA in Sage and added the same bug.

ML-DSA: new name, same fragility

I wrote a casual implementation of ML-DSA
in Sage and added the same bug.

I also wrote an exploit script.
Given a public key and two signatures,
the exploit computes an equivalent secret key
in 1 second on 1 core of my laptop.
(I think C code would be <1 ms on 1 core.)

ML-DSA: new name, same fragility

I wrote a casual implementation of ML-DSA
in Sage and added the same bug.

I also wrote an exploit script.
Given a public key and two signatures,
the exploit computes an equivalent secret key
in 1 second on 1 core of my laptop.
(I think C code would be <1 ms on 1 core.)

Also checks that it can use the key it computed
to forge signatures on attacker-chosen messages.

Details: https://cr.yp.to/papers.html#mldsa

Estimating the damage done by bugs

Typical ML-DSA code: 2000–4000 lines per library.

Estimating the damage done by bugs

Typical ML-DSA code: 2000–4000 lines per library.

Estimate 2 ML-DSA vulnerabilities per library;
1 *severe* ML-DSA vulnerability per 4 libraries.
Many ML-DSA libraries so treat this statistically.

Estimating the damage done by bugs

Typical ML-DSA code: 2000–4000 lines per library.

Estimate 2 ML-DSA vulnerabilities per library;
1 *severe* ML-DSA vulnerability per 4 libraries.
Many ML-DSA libraries so treat this statistically.

Model libraries as 50% new in 2025; 50% new in 2026; bug rate then multiplied by 0.8 each year.

Estimating the damage done by bugs

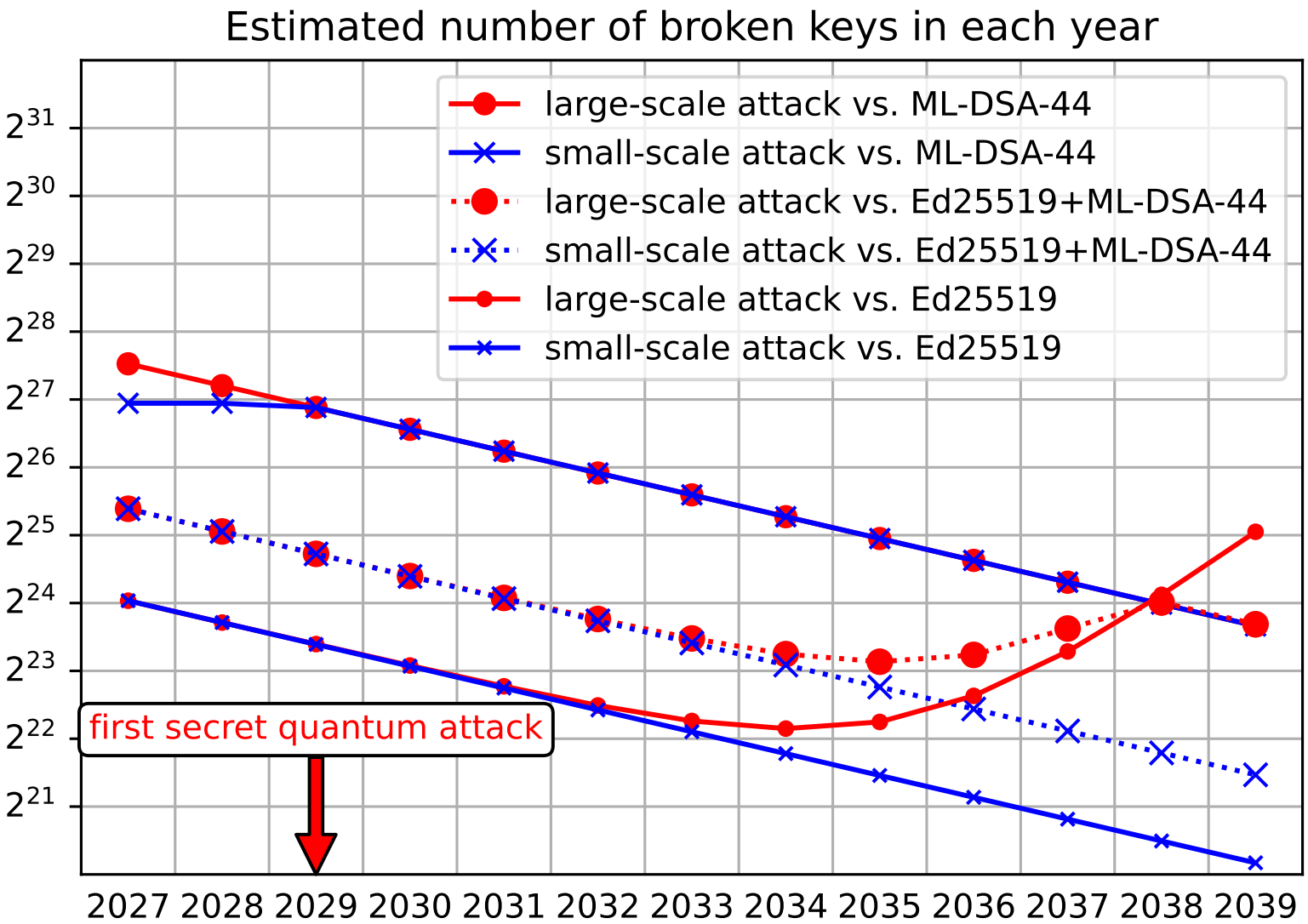
Typical ML-DSA code: 2000–4000 lines per library.

Estimate 2 ML-DSA vulnerabilities per library;
1 *severe* ML-DSA vulnerability per 4 libraries.
Many ML-DSA libraries so treat this statistically.

Model libraries as 50% new in 2025; 50% new in 2026; bug rate then multiplied by 0.8 each year.

Estimate 2^{30} target ML-DSA keys.
Account for limits on attacker CPU time.

See paper for more



The future

Some useful steps to take

1. Admit that we have a problem.

Some useful steps to take

1. Admit that we have a problem.
2. Make sure to deploy ECC+PQ, not solo PQ.

Some useful steps to take

1. Admit that we have a problem.
2. Make sure to deploy ECC+PQ, not solo PQ.
3. Write specs in Python or Sage. Test that vs. C.

Some useful steps to take

1. Admit that we have a problem.
2. Make sure to deploy ECC+PQ, not solo PQ.
3. Write specs in Python or Sage. Test that vs. C.
4. Use memory-safety tools: at least valgrind+asan.

Some useful steps to take

1. Admit that we have a problem.
2. Make sure to deploy ECC+PQ, not solo PQ.
3. Write specs in Python or Sage. Test that vs. C.
4. Use memory-safety tools: at least valgrind+asan.
5. Use SUPERCOP's automatic test mechanisms.
   (These also catch many timing leaks.)

Some useful steps to take

1. Admit that we have a problem.
2. Make sure to deploy ECC+PQ, not solo PQ.
3. Write specs in Python or Sage. Test that vs. C.
4. Use memory-safety tools: at least valgrind+asan.
5. Use SUPERCOP's automatic test mechanisms. (These also catch many timing leaks.)
6. Try using formal-verification tools. Examples: CryptoLine, s2n-bignum, saferewrite, sortverif. But beware overconfident claims of verification.