Exploiting ML-DSA bugs

Daniel J. Bernstein[1,2]

[1] Department of Computer Science, University of Illinois at Chicago, USA
[2] Institute of Information Science, Academia Sinica, Taiwan
`djb@cr.yp.to`

Abstract. At least four Dilithium software vulnerabilities have been announced so far, including an identical vulnerability in each of the two official Dilithium 1.0 implementations and two different vulnerabilities in a "verified" implementation of Dilithium 3.4, also known as ML-DSA. However, there do not appear to have been any demos showing exploitability of any of these vulnerabilities.

This paper shows that a small change in ML-DSA software creates an ML-DSA version of the Dilithium 1.0 software vulnerability, can occur by accident as in the original vulnerability, interoperates with authentic ML-DSA, passes typical tests, and is exploitable in 1 second on 1 laptop core. This paper provides an open-source attack demo that inspects a public key and two signatures, obtains an equivalent secret key, and uses this key to rapidly forge signatures on attacker-chosen messages.

This paper also shows that another small change in ML-DSA software creates a different software vulnerability, can occur by accident as in the Sony PlayStation 3 ECDSA vulnerability, interoperates with authentic ML-DSA, passes typical tests, and is exploitable in 1 second on 1 laptop core. This paper again provides an open-source attack demo that rapidly forges signatures on attacker-chosen messages, after inspecting a public key and a few signatures.

This paper then uses standard techniques to estimate exploitability rates for ML-DSA software, and to estimate the number of ML-DSA keys that the attacker will be able to break in year Y, as a function of Y.

This paper also reviews evidence in the literature regarding quantum timelines, costs of quantum attacks, and non-quantum security failures in ECC, so as to estimate the number of Ed25519+ML-DSA double-signing keys that the attacker will be able to break in year Y. The main conclusion is that, even years after the first quantum attack, this number will still be much smaller than the number of breakable ML-DSA keys.

Qualitative security benefits of ECC+PQ compared to solo PQ have been pointed out before, but not with quantified estimates of the number of breakable keys. Some recent postings gave arguments disputing these benefits; this paper closes by pointing out flaws in those arguments.

Keywords: software vulnerabilities, post-quantum cryptography

---

Permanent ID of this document: `ddd73b60e0c76be3f8236a304feba7d5805f9f60`. Date: 2026-06-01. This work was funded by the Taiwan's Executive Yuan Data Safety

1 Introduction

There is a current panic to upgrade cryptographic libraries and applications to use post-quantum signatures. How many PQ signature keys will be breakable because of exploitable bugs in the new PQ signature software?

1.1 Failures of cryptographic software

Let's start with general evidence regarding how often cryptographic software fails. In 2021, Blessing, Specter, and Weitzner [49] analyzed 312 vulnerability announcements, specifically "CVEs" during 2010–2020 from "eight widely used cryptographic libraries". [49, Table 7] reports that there were 0.450 CVEs per 1000 lines of code added to NSS and 1.187 CVEs per 1000 lines of code added to OpenSSL. [49, Table 2] reports that 27.2% of CVEs were "cryptographic". [49, Table 3] reports an "exploitable lifetime" of 5.13 years on average, standard deviation 4.29, for 201 CVEs across OpenSSL, GnuTLS, and NSS.

Presumably adding thousands of lines of code per library for a PQ signature system will add many new vulnerabilities to the ecosystem, often exploitable for 5 or more years. But optimists might hope that the situation isn't so bad:

- Maybe exploiting a vulnerability is expensive in computer time or in the amount of data collected, limiting the number of signatures that an attacker can forge or at least limiting the number of keys that an attacker can target. Some cryptographic attacks are very expensive. [102] claims that the costs of exploiting a particular OpenSSL DH bug would be "very significant and likely only accessible to a limited number of attackers". Expensive breaks of signature systems are still a security disaster for the broken keys and for everything relying on those keys (see, e.g., the Flame malware, which according to [118] used a difficult attack to break a software-update key from Microsoft), but not as bad as low-cost breaks. Only about 1/3 of the CVEs in [49] were rated severe, and only 11 of those were characterized by [49] as "cryptographic" rather than "memory buffer issues" et al.
- Maybe the PQ signature systems being deployed are designed to use very few lines of code, reducing the vulnerability rates correspondingly. Cryptographic systems vary in the number of lines of code they end up using; see, e.g., [27] analyzing the complexity of reference libraries for lattice KEMs, measured by lines of code and various other metrics.
- Maybe these signature systems are designed to reduce vulnerability rates per line of code. As a similar example in the context of pre-quantum ECDH, consider the way that X25519 protects against programmers who skip point-validity checks (see generally [42, Section 9]) or who skip exceptional cases in elliptic-curve formulas (see generally [42, Section 10]).

and Talent Cultivation Project (AS-KPQ-109-DSTCP) and by an Amazon Research Award. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s)."

- Maybe programming practices have improved enough since 2021 to noticeably reduce vulnerability rates. Well, hmmm, new code is appearing more rapidly than ever and is often buggy (see, e.g., [71])—but maybe this is outpaced by improved code review, code testing, and code verification.
- Maybe messages are being double-signed with ECC *and* the new PQ signature system, with verifiers checking *both* signatures, to reduce the impact of PQ vulnerabilities. See Section 1.4.

This paper collects evidence regarding each of these points. Let me be clear from the outset about my motivation: I would like risk assessment and risk management to be founded upon facts rather than upon wishful thinking.

1.2 A case study: Failures of Dilithium software

I'll focus on Dilithium in this paper. Dilithium is a PQ signature system that was submitted to NIST in 2017, selected by NIST for standardization in 2022, and standardized by NIST as ML-DSA in 2024.[3] The current upgrade panic includes deployment of Dilithium, and in particular ML-DSA-44; see, e.g., [125].[4] I'll use ML-DSA-44 as a specific example throughout this paper.

At least four Dilithium software vulnerabilities have been announced so far. The first two vulnerabilities—an identical vulnerability shared by both of the official Dilithium implementations submitted to NIST in 2017—were announced in [87] in 2018, which said the vulnerabilities "can easily be exploited to recover the secret key". The hope that these were merely teething troubles, the only bugs that would ever be found in Dilithium software, collapsed in 2026 when Kobeissi [81] announced two further vulnerabilities, both of which were bugs in Cryspen's libcrux, a library that claimed to have been "formally verified" to be "functionally correct".[5] Firefox and OpenSSH use libcrux; they had not rolled

[3] Actually, there have been six incompatible versions of Dilithium so far: Dilithium 1.0 in 2017 [65], Dilithium 2.0 in 2019 [66], Dilithium 3.0 in 2020 [7] (in particular, moving to larger cryptosystem parameters), Dilithium 3.1 in 2021 [8] (changing randomness-expansion details without listing the changes in its changelog), Dilithium 3.3 as a draft of ML-DSA in 2023 [96], and Dilithium 3.4 as the ML-DSA standard in 2024 [97]. (The 3.3 and 3.4 numbers aren't part of common terminology; I've chosen them as reminders of the 2023 and 2024 dates.) Furthermore, each version provides incompatible parameter sets. All versions are treated together in, e.g., Lyubashevsky's 2024 review [88] of "Dilithium (ML-DSA)"; Lyubashevsky was listed in [65] as the "Principal Submitter" of Dilithium to NIST. See also [89].

[4] "My position on the urgency of rolling out quantum-resistant cryptography has changed compared to just a few months ago. ... Hybrid classic + post-quantum authentication makes no sense to me anymore and will only slow us down; we should go straight to pure ML-DSA-44." See Section 10 for analysis of this recommendation.

[5] See [80]: "Formally Verified Post-Quantum Cryptography ... Our implementations are verified to be panic free, functionally correct, and secret independent. ... We have been working with Mozilla to upstream our verified implementations into the NSS cryptographic library. ... Cryspen provides open source high-performance, high-assurance implementations of ML-KEM (FIPS 203) and ML-DSA (FIPS 204)."

out ML-DSA when I last checked, but this paper assumes that the current panic will rapidly change that.[6]

After I started writing this paper, [85] reported a bug in another ML-DSA library, a bug that occasionally produces "non-conformant signatures". Perhaps those signatures leak information about the secret key. Tests, including known-answer tests (KATs), had not been applied to enough inputs to catch the bug before the code was released.

These examples show that the Dilithium software ecosystem is not magically immune to bugs—and not magically immune to vulnerabilities. But optimists might still see reasons for hope:

- The two vulnerabilities in [81] were reported to allow merely "signature malleability" (i.e., replacing one signature with another signature *on the same message*; also called "violating strong unforgeability") and "consensus violation". Such issues matter for *some* applications, but not for applications that simply want to ensure that all messages passing verification were signed by the legitimate signer—the primary goal of signatures. The mere fact that ECDSA allows easy signature malleability is not commonly accepted as an argument that every usage of ECDSA in libraries and applications should result in a CVE. For this paper, I'm counting vulnerabilities as severe only if the attacker can quickly forge signatures on new attacker-chosen messages.
- [85] does not claim a vulnerability, and in particular does not analyze the possibility of secret-key leakage, so I'm counting this only as a bug, not as a vulnerability. See also Appendix B for a list of eight CVEs that mention Dilithium or ML-DSA, along with explanations of why I'm not counting those as severe vulnerabilities.
- The wording "can easily be exploited to recover the secret key" in [87] sounds much more devastating, but wasn't quantified and wasn't backed by a demo. Maybe it's referring to something that would be within reach for a large-scale attacker for breaking a few keys but still too expensive to apply to most keys. Also, there were various changes across versions of Dilithium; even if the Dilithium 1.0 attack in [87] actually exists, maybe the changes made the attack inapplicable to Dilithium 3.4 (ML-DSA).

1.3 Contributions of this paper, part 1: quantifying the impact of ML-DSA bugs

Section 3 shows that a small change in ML-DSA software creates an ML-DSA version of the Dilithium 1.0 software vulnerability that was announced in [87]. This change interoperates with authentic ML-DSA, passes typical tests, and can easily occur by accident as in the original vulnerability. Section 4 presents an open-source demo (available in a supplement [31] to this paper) that inspects a public key and two signatures from the modified software, obtains an equivalent

[6] The Collins dictionary says that "panic" means "a very strong feeling of anxiety or fear, which makes you act without thinking carefully".

secret key in 1 second of computation on 1 laptop core, and uses this key to rapidly forge signatures.

Section 6 shows that another small change in ML-DSA software creates an ML-DSA version of the Sony PlayStation 3 ECDSA vulnerability that was announced and exploited in [53]. This change interoperates with authentic ML-DSA, passes typical tests, and can easily occur by accident as in the Sony vulnerability. The same section presents an open-source demo that inspects a public key and a few signatures from the modified software, obtains an equivalent secret key in 1 second of computation on 1 laptop core, and uses this key to rapidly forge signatures.

In both of these attacks, attackers are free to choose their own messages to sign, destroying the security that these signature keys were supposed to provide to the user's application. The results are as devastating as an application-layer failure to check signatures—a well-known type of failure that one normally tries to eliminate through reviewing application usage of signatures and through negative tests of how applications handle modified messages under the same signature. These reviews and tests do nothing to stop this paper's attacks.

Section 7 uses standard techniques to estimate the overall exploitability rates of ML-DSA software, and to estimate the number of ML-DSA keys that an attacker will be able to break in year Y, for each $Y \in \{2027, 2028, \ldots, 2039\}$. Here "break" refers to quickly forging signatures on attacker-chosen messages ("universal forgeries"; see Appendix A), as in Sections 4 and 6. Note that this does *not* count the vulnerability announcements from [81] as breaks.

1.4 Signing twice

The literature describes a straightforward mechanism to reduce the damage of PQ security failures: deploy PQ systems as an *extra* layer of security beyond ECC, rather than as a *replacement* for ECC. (See, e.g., [17] from 2016.)

This means, in particular, double-encrypting data with ECC+PQ rather than using solely PQ encryption, and double-signing data with ECC+PQ rather than using solely PQ signatures. The verifier for an ECC+PQ signature checks the ECC signature *and* the PQ signature, so that many potential types of PQ security failures also face the attacker with the problem of breaking the ECC signature. (Often ECC+PQ is called a "hybrid" of ECC and PQ, although this terminology can be confusing since the word "hybrid" is also used in many other ways; saying "double encryption" and "double signatures" is more informative.)

ECC+PQ does not *eliminate* the damage from PQ security failures. There is ample literature describing ways the ECC part can fail: for example, the primary motivation for PQ deployment is that quantum computers are expected to start breaking some ECC keys. But I have not found literature *quantitatively* estimating how many ECC+PQ keys will be breakable, so as to support quantified cost-benefit comparisons between ECC+PQ and solo PQ.

Readers who look at costs and observe ECC+PQ having negligible cost beyond solo PQ might think that quantifying the benefit of ECC+PQ is unnecessary: obviously one should simply replace every PQ deployment with an

ECC+PQ deployment as an affordable way to sometimes save the day, without worrying how often "sometimes" is. However, I have seen arguments that solo PQ is better than ECC+PQ; see, e.g., my 2024 posting [**25**] for an analysis of such arguments from NSA and GCHQ. Typically these arguments portray ECC+PQ as having non-negligible cost[7] beyond solo PQ. Even if none of those cost claims hold up to scrutiny, some decision-makers will be left with an impression of non-negligible cost; it then becomes important to evaluate the benefits.

As an analogy: Last century, data regarding car crashes and the impact of seatbelts helped spur the introduction of seatbelt requirements, despite claims from the car industry that seatbelts were a cost problem. See, e.g., [**122**].

1.5 Contributions of this paper, part 2: quantifying the impact of quantum computers and ECC bugs

Section 8 estimates the number of Ed25519+ML-DSA double-signing keys that the attacker will be able to break in year Y, again for each $Y \in \{2027, 2028, \ldots, 2039\}$.

This includes analyzing the number of Ed25519+ML-DSA keys that an attacker can break by (1) breaking the ML-DSA part as in Section 7 and (2) using a quantum computer to break the Ed25519 part. But this also includes various ways that the Ed25519 part might be broken *without* a quantum computer.

1.6 Contributions of this paper, part 3: comparisons

Section 9 combines this paper's estimates into a graph, Figure 9.1.1, comparing the number of Ed25519+ML-DSA keys breakable by attackers in year Y to the number of ML-DSA keys breakable by attackers in year Y and the number of Ed25519 keys breakable by attackers in year Y, for all Y between 2027 and 2039.

The main conclusion is that far fewer Ed25519+ML-DSA keys will be breakable than ML-DSA keys, not just before the first quantum attack but also continuing for years after the first quantum attack. This conclusion is robust against uncertainties in the underlying numbers, such as bug rates.

Finally, Section 10 surveys recent arguments that Ed25519+ML-DSA does *not* have a security benefit compared to solo ML-DSA. Section 10 points out various flaws in those arguments. Beware that Section 10 does not survey flaws in claims that Ed25519+ML-DSA adds non-negligible cost beyond ML-DSA.

1.7 Acknowledgments

Thanks to Tanja Lange for her comments, including suggestions of (1) improvements in this paper's first demo, (2) the target of this paper's second demo, and (3) also comparing Ed25519 to Ed25519+ML-DSA.

[7] See, e.g., the claim in [**131**] that ECC+PQ would slow down high-frequency trading. This claim is implausible (presumably the use of public-key cryptography to set up a secure session occurs before the trading day begins, so high-frequency trading cares only about the performance of symmetric cryptography), was challenged, and was never substantiated, but was also never withdrawn.

2 Review of selected aspects of ML-DSA

For conciseness, the following description focuses on aspects of ML-DSA relevant to the attack descriptions in Sections 4, 5, and 6. Also, for concreteness, the description focuses specifically on ML-DSA-44; this is the first ML-DSA parameter set, and the one recommended in [**125**].

Define q as the prime number $2^{23} - 2^{13} + 1$, and define R as the ring $(\mathbb{Z}/q)[x]/(x^{256} + 1)$. Each element of R can be written as a polynomial of the form $p_0 + p_1 x + \cdots + p_{255} x^{255}$, where each p_j is an integer between 0 and $q - 1$, and thus viewed as a length-256 vector $(p_0, p_1, \ldots, p_{255})$ of integers modulo q. Addition in R is addition of vectors modulo q, and multiplication in R is negacyclic convolution of vectors modulo q.

ML-DSA-44 key generation calls a separately provided RNG (an "approved RBG") to generate some secret bytes, and transforms those in a particular way into a secret key that includes (among other things) secret polynomials $s_0, s_1, s_2, s_3 \in R$ and a 32-byte secret K. For each message, the signer calls the RNG again to generate 32 more bytes (this is the "default" form of ML-DSA; there is also an "optional deterministic variant" of ML-DSA in which these bytes are all 0), hashes those in a particular way together with K and the message, expands the hash output into a much longer stream of secret bytes, extracts a stream of secret 18-bit integers, and subtracts each integer from 2^{17} modulo q, obtaining 1024 coefficients of 4 secret polynomials $y_0, y_1, y_2, y_3 \in R$. The signer also computes a public "challenge" $c \in R$, computes $z_i = cs_i + y_i$ using multiplication and addition in R, and releases z_0, z_1, z_2, z_3 as part of a signature. Exception: the signer sometimes rejects a signature and retries with a new hash.

Each signature also includes some "hints" used to handle some issues that appear in verification. For further details of ML-DSA key generation, signing, and verification, see the demos provided in the supplement [**31**] to this paper.

3 AABBCC coefficients in ML-DSA

Section 1's review of known Dilithium bugs began with a 2018 vulnerability announcement [**87**] for the official Dilithium software that had been submitted to NIST in 2017. Here is part of the text of the announcement:

> 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.

Optimists might interpret a bug in "a randomness generator of our NIST submission" to mean that the Dilithium submission to NIST included (1) correct software for the Dilithium signature system along with (2) a separate new buggy

RNG that was duplicating outputs; presumably any deployment would have replaced that RNG with a safe RNG from a cryptographic library, making the deployment safe. But this interpretation is incorrect. The bug was in the software for Dilithium per se, specifically in the code for generating signatures, not in software for a separate RNG.

As noted in Section 1, [**87**] doesn't quantify or justify the "can easily be exploited" statement. Section 4 below reports a very fast exploit. This attack does not recover "the secret key" (and does not justify [**87**]'s claim regarding that), but it recovers an equivalent key—part of the secret key—and uses the equivalent key to rapidly forge signatures on arbitrary attacker-chosen messages, signatures that verify under the original ML-DSA-44 public key.

This section looks at the common software-engineering processes that allow this type of bug to happen. This bug is not just a mistake that *was made*; it is an *easy* mistake to make. The changes from Dilithium 1.0 to Dilithium 3.4 (ML-DSA) do not prevent programmers from making the mistake.[8] Software with the bug is fully interoperable with software without the bug, so the bug cannot be caught by interoperability tests. The bug also passed the known-answer tests that were included in the Dilithium submission in 2017. This section explains a more stringent testing approach that would have been likely to catch the bug, but also explains why cryptographic libraries normally have much weaker tests than this.

3.1 How easily ML-DSA software can produce AABBCC

Recall from Section 2 that, for each message, an ML-DSA-44 signer generates coefficients of secret polynomials y_i by converting secret bytes into secret 18-bit integers and subtracting each integer from 2^{17} modulo q.

FIPS 204 describes the conversion here as first expanding a list of $32 \cdot 18$ bytes into a list of $256 \cdot 18$ bits ("BytesToBits") and then converting each successive 18-bit sublist into an 18-bit integer ("BitsToInteger"); see the calls in [**97**, "BitUnpack"], which then subtracts each of the 256 resulting integers from 2^{17}. For efficiency, programmers will often use a faster conversion subroutine along the lines of `convert18` in Figure 3.1.1, unrolling each bit-by-bit loop and merging bit operations into larger operations. Often programmers will also unroll the subtract-from-2^{17} loop, along the lines of `ygen` in Figure 3.1.1.

These code segments are full of opportunities for typos. Consider, for example, a programmer copying and pasting the `y[2*i]` line in Figure 3.1.1 to obtain the next line, and correctly changing `y[2*i]` to `y[2*i+1]`, but neglecting to change `c[2*i]` to `c[2*i+1]`. Or consider a programmer accidentally inserting an extra `+1`, changing the first `c[2*i]` to `c[2*i+1]`. Either way produces an AABBCC pattern of coefficients in the `y` array; Section 4 will show how to exploit this.

[8] I also have not found any literature indicating that the changes were *meant* to address this mistake, nor have I found literature analyzing potential ways to modify Dilithium so that this mistake would not occur.

```
void convert18(int *c,const char *b)
{
  for (int i = 0;i < 64;i += 1) {
    *c++ = 0x3ffff&(b[0]|((int)b[1]<<8)|((int)b[2]<<16));
    *c++ = 0x3ffff&((b[2]>>2)|((int)b[3]<<6)|((int)b[4]<<14));
    *c++ = 0x3ffff&((b[4]>>4)|((int)b[5]<<4)|((int)b[6]<<12));
    *c++ = 0x3ffff&((b[6]>>6)|((int)b[7]<<2)|((int)b[8]<<10));
    b += 9;
  }
}

void ygen(int *y,const int *c)
{
  for (int i = 0;i < 128;i += 1) {
    y[2*i] = Q+GAMMA1-c[2*i];
    y[2*i+1] = Q+GAMMA1-c[2*i+1];
  }
}
```

Fig. 3.1.1. Example of how a programmer might write two ML-DSA-44 subroutines.

I wrote (and did not bother to audit or test) the `convert18` and `ygen` examples in Figure 3.1.1, but similar complications are pervasive in current ML-DSA libraries. For example, Figure 3.1.2 is the corresponding subroutine in the current version of the "mldsa-native" library at the time of this writing. Most programmers writing such code would write it with considerable copying and pasting. A moment of forgetfulness in updating lines near the end after a paste changes `1` to `0` and `3` to `2` on the right side of those lines, producing the AABBCC pattern, or changes `2` to `0` and `3` to `1`, producing an ABABCDCD pattern, which I assert is exploitable as easily as AABBCC. (But Section 4 and the accompanying demo focus on the AABBCC case, not ABABCDCD.)

As another example, Figure 3.1.3 is the corresponding subroutine in the current version of OpenSSL at the time of this writing. Inside that code, accidentally changing an early `-` to `&` would clear `mask_18_bits` and clear every second polynomial coefficient, producing an A0B0C0 pattern, again exploitable as easily as AABBCC.

The official Dilithium 1.0 code illustrates yet another route to AABBCC coefficients. The official code had lines

```
t  = buf[pos];
t |= (uint32_t)buf[pos + 1] << 8;
t |= (uint32_t)buf[pos + 2] << 16;
t &= 0xFFFFF;
```

to extract a 20-bit integer from three bytes (Dilithium 1.0 specified 20-bit coefficients and rejection sampling rather than 18-bit coefficients), and lines

```
MLD_STATIC_TESTABLE void mld_polyz_unpack_c(
    mld_poly *r, const uint8_t a[MLDSA_POLYZ_PACKEDBYTES])
{
  unsigned int i;
  for (i = 0; i < MLDSA_N / 4; ++i)
  {
    r->coeffs[4 * i + 0] = a[9 * i + 0];
    r->coeffs[4 * i + 0] |= (int32_t)a[9 * i + 1] << 8;
    r->coeffs[4 * i + 0] |= (int32_t)a[9 * i + 2] << 16;
    r->coeffs[4 * i + 0] &= 0x3FFFF;

    r->coeffs[4 * i + 1] = a[9 * i + 2] >> 2;
    r->coeffs[4 * i + 1] |= (int32_t)a[9 * i + 3] << 6;
    r->coeffs[4 * i + 1] |= (int32_t)a[9 * i + 4] << 14;
    r->coeffs[4 * i + 1] &= 0x3FFFF;

    r->coeffs[4 * i + 2] = a[9 * i + 4] >> 4;
    r->coeffs[4 * i + 2] |= (int32_t)a[9 * i + 5] << 4;
    r->coeffs[4 * i + 2] |= (int32_t)a[9 * i + 6] << 12;
    r->coeffs[4 * i + 2] &= 0x3FFFF;

    r->coeffs[4 * i + 3] = a[9 * i + 6] >> 6;
    r->coeffs[4 * i + 3] |= (int32_t)a[9 * i + 7] << 2;
    r->coeffs[4 * i + 3] |= (int32_t)a[9 * i + 8] << 10;
    r->coeffs[4 * i + 3] &= 0x3FFFF;

    r->coeffs[4 * i + 0] = MLDSA_GAMMA1 - r->coeffs[4 * i + 0];
    r->coeffs[4 * i + 1] = MLDSA_GAMMA1 - r->coeffs[4 * i + 1];
    r->coeffs[4 * i + 2] = MLDSA_GAMMA1 - r->coeffs[4 * i + 2];
    r->coeffs[4 * i + 3] = MLDSA_GAMMA1 - r->coeffs[4 * i + 3];
  }
}
```

Fig. 3.1.2. One of the "mldsa-native" subroutines inside ML-DSA-44 for which errors will not be caught by ML-DSA-44 functionality tests. This figure removes conditionally compiled non-ML-DSA-44 code and comments specifying invariants.

```
if(t <= 2*GAMMA1 - 2)
  a[ctr++] = Q + GAMMA1 - 1 - t;
```

to use that integer if the integer was small enough. The code also had lines

```
t  = buf[pos + 2] >> 4;
t |= (uint32_t)buf[pos + 3] << 4;
t |= (uint32_t)buf[pos + 4] << 12;
```

to extract the next 20-bit integer, and lines

```
if(t <= 2*GAMMA1 - 2 && ctr < len)
```

```
static int poly_decode_signed_two_to_power_17(POLY *p, PACKET *pkt)
{
    uint32_t *out = p->coeff;
    const uint32_t *end = out + ML_DSA_NUM_POLY_COEFFICIENTS;
    const uint8_t *in;
    static const uint32_t range = 1u << 17;
    static const uint32_t mask_18_bits = (1u << 18) - 1;

    do {
        uint32_t a1, a2, a3;

        if (!PACKET_get_bytes(pkt, &in, 9))
            return 0;
        in = OPENSSL_load_u32_le(&a1, in);
        in = OPENSSL_load_u32_le(&a2, in);
        a3 = (uint32_t)*in;

        *out++ = mod_sub(range, a1 & mask_18_bits);
        *out++ = mod_sub(range, (a1 >> 18) | ((a2 & 0xF) << 14));
        *out++ = mod_sub(range, (a2 >> 4) & mask_18_bits);
        *out++ = mod_sub(range, (a2 >> 22) | (a3 << 10));
    } while (out < end);
    return 1;
}
```

Fig. 3.1.3. One of the OpenSSL subroutines inside ML-DSA-44 for which errors will not be caught by ML-DSA-44 functionality tests.

```
a[ctr++] = Q + GAMMA1 - 1 - t;
```

to use that integer if the integer was small enough (and if further integers were still needed). The bug was that the second and third code segments were swapped, with `t` being generated and generated and used and used, so the first `t` was never used while the second `t` was used twice.

A code reviewer carefully studying these particular code segments will easily spot the bug, but this begs the question of how much Dilithium code is carefully reviewed. See Section 7.2 for analysis of overall Dilithium code size. One could imagine a compiler flagging the fact that the first `t` was computed and then overwritten without being saved, but `gcc-12 -Wall -Wextra -Werror` does not flag this, never mind the extra challenges for a compiler to carry out such data-flow analysis via arrays in, e.g., Figure 3.1.2. Formal verification might *sound* like comprehensive code review (see, e.g., [**4**]), but recall that [**81**] reported vulnerabilities in an ML-DSA library that claimed formal verification; more importantly, almost all ML-DSA library advertisements that I've looked at make no mention of formal verification. Having almost all ML-DSA libraries passing careful formal verification would make a big difference in estimates of vulnerability rates, but the current situation is very far from that.

3.2 How easily these bugs can pass common types of tests

Test suites in cryptographic libraries consistently include keygen-sign-verify tests to see whether signatures pass verification. There are frequent efforts to carry out cross-library interoperability tests, checking whether keys and signatures from one library can be verified by another library. Protocols often have multiple implementations based on multiple libraries, and then testing that the protocols work correctly naturally ends up checking interoperability across those libraries.

However, none of these functionality tests are capable of catching bugs that produce AABBCC coefficients in y in ML-DSA. The issue here isn't the conventional complaint about tests, namely that tests try only a limited number of inputs;[9] coefficient-repetition bugs corrupt every signature. The issue is instead that these bugs lead to valid signatures that are accepted by correct ML-DSA verifiers.

3.2.1 The volume of untested code in the trusted computing base. There are some non-ML-DSA-specific types of bugs that also pass functionality tests and have a similarly devastating impact: consider an extremely weak library RNG, for example, or a programmer accidentally using an RNG to generate 3 bytes instead of 32 bytes. But the RNG in a cryptographic library is typically stable code *reused* by all cryptosystems in the library, and each use of the RNG to generate 32 bytes is just *one more line of code* for auditors to check, whereas ML-DSA has *many more lines of ML-DSA-specific code* converting 32 bytes into pseudorandom small polynomials. Each new ML-DSA library thus adds many lines of code for which bugs are not caught by functionality tests. See Section 5 for further analysis of the consequences of this observation.

3.2.2 Checksums of outputs for known derandomized inputs. For many years, Lange and I have been running a performance-measurement project, eBACS [43]. The eBACS benchmarking tool, called SUPERCOP, also includes a variety of correctness tests. In particular, SUPERCOP automatically computes "checksums" as hashes of software outputs, for comparison to checksums recorded as hashes of outputs of other implementations of the same cryptographic primitive.

SUPERCOP has been comparing checksums for signature systems since 2011 (see [14]). These known-answer tests are more powerful than functionality tests such as verifying signatures: a code change that modifies a signature will produce a checksum mismatch even if the modified signature passes verification.

Wait a minute: keys and signatures are generated randomly, so how can these comparisons ever pass? Answer: SUPERCOP's checksum mechanism replaces SUPERCOP's centralized RNG, `randombytes`, with a deterministic RNG. See [18] and [19] for more information on SUPERCOP's RNGs.

[9] For comparison, this conventional complaint *is* the issue for the ML-DSA overflow bugs identified in [85].

At the time of this writing, SUPERCOP contains 4913 implementations (from hundreds of contributors) of 1470 primitives, so on average there are more than 3 implementations per primitive. But only 1308 of the primitives are marked as having checksums. For other primitives, the implementations obtain randomness from sources other than `randombytes`—e.g., from OpenSSL's RNG, which I have not yet taught SUPERCOP to replace. Furthermore, there is still much more cryptographic software *not* included in SUPERCOP. Libraries sometimes include checksums in their own test suites, but this begs the question of how comprehensively those checksums are tested against other libraries.

3.2.3 An example of how known-answer tests fail. When NIST called for post-quantum signature submissions in 2017, it required submissions to use SUPERCOP's signature API, to use `randombytes` for randomness generation, and to provide known-answer tests using KAT software that NIST had supplied. NIST's KATs are much more verbose than SUPERCOP's checksums since they show all outputs rather than just a hash,[10] but KATs and checksums both provide the same basic feature of seeing examples of how cryptosystem software is expanding RNG results, so a bug that frequently enough changes the expansion will also change the known-answer tests.

The official Dilithium 1.0 software submitted to NIST in 2017 included a reference implementation *and* an AVX2-optimized implementation. Why, then, did the KATs not catch the AABBCC bug? Answer: The same bug was in *both* implementations. The AVX2-optimized implementation started as a copy of the reference implementation and then replaced code for some subroutines with vectorized code; the changes had no effect on the bug.

3.2.4 One way to make known-answer tests more effective. Writing an *independent* implementation of the Dilithium specification in Python, and using the Python script to generate checksums to compare against checksums (or KATs) from C code, would have had a much better chance of catching this bug. I haven't found literature quantifying this, but my experience is that mistakes in C code for cryptosystems are very often not shared by independent Python scripts, in part because different programmers have different habits and in part because so many mistakes in C code come from the pursuit of speed.

It is important to redo the Python validation whenever cryptosystem outputs are changed by a cryptosystem "tweak". Checksums are disconnected not just across different parameter sets but across different cryptosystem versions, such as Dilithium 1.0, 2.0, 3.0, 3.1, 3.3, and 3.4 (ML-DSA). Cryptosystem instability makes it more difficult to obtain a critical mass of implementations being checked against each other. Old checksums won't catch bugs introduced in tweaked randomness-expansion code. New checksums have to be validated.

What I'm describing here is affordable. For example, the Classic McEliece team systematically uses independent Sage scripts to validate checksums for the

[10] Years later, in 2024, Valsorda [**124**] advertised the conciseness of "accumulated test vectors", which he claimed not to have "seen documented before".

official Classic McEliece software; see my presentation [28] for further discussion of the assurance mechanisms. The production libmceliece software library [36] is directly tested against the same checksums.

3.2.5 More examples of inadequate usage of known-answer tests. I have looked at test suites in many more libraries, typically finding tests that are far less stringent. Many cryptographic implementors seem unaware that randomized functions are testable.

Consider, e.g., Valsorda [123] recommending in 2023 that every cryptographic primitive using randomness be defined as a "deterministic function that takes a fixed-size string of random bytes" since "specifying randomness as just another fixed-size input of a deterministic function makes it possible to provide 'known answer' test cases". This misses the better option of replacing a central RNG with a deterministic RNG, as SUPERCOP had already done for many years, providing checksums *without* the API constraint of requiring cryptosystems to be deterministic functions taking fixed-size seeds.

It is easy to see how allowing cryptosystems to internally call `randombytes`, rather than imposing this API constraint, simplifies cryptographic engineering and reduces security risks. Think about, e.g., a simple RSA-2048 key-generation function that calls a prime-generation function twice; a simple prime-generation function that repeatedly calls an integer-generation function until the result is prime; and a simple integer-generation function that produces an odd integer between 2^{1023} and 2^{1024} by asking `randombytes` for 128 bytes of output and then setting the top and bottom bits. Simply plugging this code into SUPERCOP generates checksums with no extra work.

Insisting on rewriting this RSA-2048 key-generation function as something deterministic would also allow checksums but would mean

- complicating the key-generation interface to accept a seed,
- adding code to the key-generation function to initialize an RNG state,
- complicating the prime-generation interface to receive an RNG state,
- complicating the integer-generation interface to receive an RNG state, and
- adding code inside the integer-generation function to update the RNG state while generating 128 bytes from that state.

The added RSA code to suddenly handle RNG initialization and update would take the RSA implementation farther away from how RSA is normally described in the literature, and would add new opportunities for security failures. The RSA specification would need to be extended to describe these RNG details so that cryptanalysts can look for attacks. Code reviewers and people developing tests would need to think about whether the added code matches the extended specification.

Scaling this type of determinism to every implementation of public-key cryptosystems in a library, often implementations with many more layers of internal functions than RSA key generation, would involve far more code than leaving the cryptosystem code untouched and having the necessary RNG state

managed by a central cryptosystem-independent coroutine in the same library. Generating known-answer tests by derandomizing the RNG is much simpler than generating known-answer tests by derandomizing every cryptographic primitive and every cryptographic implementation.

Even when a primitive is defined to first generate fixed-size randomness and then call a deterministic function, programmers often won't bother providing the deterministic function. It's an extra function; it has a more complicated API; it doesn't provide any obvious benefits for applications. Even when programmers are told that tests are important and that some libraries can test only deterministic functions, programmers are often overconfident and won't think that *they* need to do extra work for tests.

Part of the supplement [**31**] to this paper is a Sage implementation of ML-DSA. Except for adding Sage adaptations of the bugs exploited in this paper's demos, I tried to do what I would expect typical ML-DSA programmers to do if they were asked to use Python or Sage. I didn't worry about speed. I didn't worry about security against timing attacks.[11] I closely followed FIPS 204. I included various assertions for properties highlighted in the FIPS 204 algorithm statements. FIPS 204 said that the "default" signing mechanism is randomized, so I implemented the randomization mechanism described there, skipping the "optional deterministic variant". FIPS 204 specified moments to compute a particular hash called μ inside the signing and verification algorithms, so I did that, ignoring the brief comments saying that μ "may optionally be computed in a different cryptographic module". Presumably typical programmers will follow the defaults.

I then looked at the ML-DSA known-answer tests in Wycheproof [**58**]. Each signing test provides a key pair, a message, this hash μ, and a signature. It seems that the intention is to have a signature function taking the key pair, a message, and μ as input, but this doesn't obviously match any of the interfaces that I noticed in FIPS 204 or that I included in my Sage script. Maybe there's a way to test the script's signing routine with [**58**], but insisting on determinism makes this unnecessarily complicated: instead of the literature's canonical description of signing as taking a message and private key as input, there seem to be multiple ad-hoc ML-DSA-specific interfaces that take more inputs. I don't think I was being obtuse here; I think many ML-DSA programmers will end up skipping the signing tests from [**58**] even if they've heard about those tests.

Meanwhile I didn't find *any* ML-DSA key-generation tests in [**58**]. I did run a few verification tests from [**58**] without incident, but that's testing only part of the ML-DSA software.

To summarize, even though the technology was already readily available before 2017 to test randomness expansion inside cryptographic software, that technology is still far from universally used. Even when it *is* used, it is often not used in a way that provides solid links between implementations and specifications. The bottom line is that AABBCC coefficients can easily appear in ML-DSA software and can easily pass tests, as they did for Dilithium 1.0.

---

[11] Timing attacks can be devastating too, but this paper focuses primarily on bugs.

4 Exploiting AABBCC coefficients in ML-DSA

Let's now look at how efficiently the bug from Section 3 can be exploited. Recall that an ML-DSA-44 signature reveals z_0, z_1, z_2, z_3 where $z_i = cs_i + y_i$; c is also public, y_i is secret, and s_i is a long-term secret reused by each signature. The effect of the bug is that the coefficients of each y_i follow an AABBCC pattern: the x^0 and x^1 coefficients are equal, the x^2 and x^3 coefficients are equal, etc.

Write the polynomial c as $C + xC'$ where $C, C' \in (\mathbb{Z}/q)[x^2]/(x^{256} + 1)$; e.g., if $c = 31 + 41x + 59x^2 + 26x^3$ then $C = 31 + 59x^2$ and $C' = 41 + 26x^2$. Similarly write s_i, y_i, and z_i as $S_i + xS'_i$, $Y_i + xY'_i$, and $Z_i + xZ'_i$ respectively. Then $Z_i + xZ'_i = (C + xC')(S_i + xS'_i) + (Y_i + xY'_i)$. Inspect even-indexed coefficients and odd-indexed coefficients to see that $Z_i = CS_i + x^2C'S'_i + Y_i$ and $Z'_i = CS'_i + C'S_i + Y'_i$.

The bug forces $Y_i = Y'_i$. Subtract to obtain $Z_i - Z'_i = (C - C')S_i + (x^2C' - C)S'_i$. The values $Z_i - Z'_i$, $C - C'$, and $x^2C' - C$ are known; i.e., we now have 128 known linear equations in the 256 coefficients of S_i, S'_i. Two signatures produce 256 known linear equations in the same 256 coefficients, and then the attacker can reasonably hope for the equations to be linearly independent, revealing S_i and S'_i by linear algebra.

A side effect of this paper's demo is experimental demonstration that the equations are independent with high probability; see Section 4.2. In non-independent cases, one can simply try another signature.[12]

4.1 Exploiting Dilithium's polynomial structure

Using generic linear algebra on a 256×256 matrix with entries in $\mathbb{Z}/q$ for each i would be treating Dilithium as an unstructured-lattice system. But Dilithium is a structured-lattice system. Let's take advantage of that for a faster attack.

One signature gives, as explained above, one equation $H = FS_i + GS'_i$ where F, G, H are known. Another signature gives another equation $H' = F'S_i + G'S'_i$ where F', G', H' are known. We now have $S_i = (HG' - H'G)/(FG' - F'G)$ and $S'_i = (H'F - HF')/(FG' - F'G)$, as long as the determinant $FG' - F'G$ is invertible in $(\mathbb{Z}/q)[x^2]/(x^{256}+1)$. So we simply have to compute an inverse in that polynomial ring, alnog with some polynomial multiplications and subtractions. Note also that F, G, F', G' are independent of i, so the work of computing $1/(FG' - F'G)$ is shared across values of i.

4.2 Speed of the demo

The above computation, either via generic linear algebra or the faster polynomial approach, produces (in independent/invertible cases) all four secrets s_0, s_1, s_2, s_3 needed for signature forgery. This is, as noted above, not the entire Dilithium

[12] Alternatively, one could enumerate small kernels or carry out more general low-dimension lattice attacks, using a fact that I didn't mention in Section 2, namely that S_i, S'_i have small coefficients.

secret key—the key has further components used to generate "hints" in signatures—but some extra work in signing lets the attacker compute "hints" that pass verification even without knowing the rest of the secret key. The extra work is essentially one matrix-vector multiplication, on top of the matrix-vector multiplication (and other computations) normally used in signing.

The attack demo included in the supplement [31] to this paper carries out a complete attack all the way through forging signatures on attacker-selected messages and checking that the forged signatures are accepted. The demo uses Sage for polynomial arithmetic and all other ML-DSA computations. The demo takes under 1 second on 1 laptop core to recover s_0, s_1, s_2, s_3, plus under 1 second to generate each forged signature. Exact timings don't matter for this paper.

I tried 50 runs of the demo, each of which successfully broke another randomly generated key, in each case using 2 signatures. The demo is also intended to catch the occasional non-independent (i.e., non-invertible) cases and to try further signatures in those cases, but I didn't test this.

4.3 Anticipated further speedups

I predict that a C demo will take less than 1 ms to recover s_0, s_1, s_2, s_3, and less than 1 ms for each forged signature. However, this paper's estimates of the number of breakable ML-DSA keys use the speed of the existing Sage demo.

The speedup that I expect comes not just from removing Sage overhead, but also from exploiting NTTs for polynomial arithmetic: compute six forward size-128 NTTs for F, G, H, F', G', H'; carry out pointwise multiplications, subtractions, and divisions in $\mathbb{Z}/q$; compute two inverse size-128 NTTs for S_i and S'_i. About half of this work will be shared across values of i.

5 The bigger picture: many ways that ML-DSA software can fail

An ad-hoc black-box test for the AABBCC bug highlighted in Section 3 is not expensive—for example, simply try running the fast attack from Section 4. But, even if all libraries could magically be convinced to run this test, the test wouldn't catch the A0B0C0 bug. It wouldn't catch the ABABCDCD bug. It wouldn't address the larger problem that each new ML-DSA library adds many lines of code for which bugs are not caught by typical tests.

The AABBCC, A0B0C0, and ABABCDCD bugs are just a few examples of how these lines can create an exploitable vulnerability. The rest of this section gives what might be the easiest example. Section 6 gives yet another example. The example below and the example in Section 6 reflect two types of vulnerabilities that have been seen in other areas of cryptography, while Section 3 involved more specific ML-DSA details, although in all cases the exploits involve some ML-DSA details.

Here is this section's example. Recall from Section 2 that the secret polynomials y_0, y_1, y_2, y_3 are obtained from a 32-byte secret K included in the

secret key, another 32 bytes of RNG output produced for this signature, and public data. FIPS 204 includes an "optional deterministic variant" of ML-DSA in which the per-signature 32 bytes of RNG output are cleared. What happens if a programmer of this variant accidentally also clears K?

For simplicity, I'll assume common programming practice of 0-initializing a secret-key buffer and then copying data into that buffer. The K-generation code is not as simple as `randombytes(K,32)`: there are multiple layers of hashing, and a portion of hash output is supposed to be copied into K. If this copying is skipped then K ends up cleared.

This bug has no effect on functionality. It can't be caught by interoperability tests and other conventional tests. It can't be caught by the nonexistent ML-DSA keygen tests in [**58**]. It *will* be caught by SUPERCOP's checksums and by similar tests in *some* libraries, but many libraries do not include such tests; see Section 3.2.5. So there's no reason to think that the bug won't end up in production.

Whenever this bug happens, it allows an even faster attack than in Section 4, an attack using just one signature instead of two. The attacker simply computes y_0, y_1, y_2, y_3 by following through the signature-generation procedure; all of the randomness has been lost. The attacker then computes each s_i as $(z_i - y_i)/c$ (presumably c will be invertible), and uses s_0, s_1, s_2, s_3 to forge signatures on attacker-chosen messages.

There are many other cryptosystems, including cryptosystems I've designed, that have internal secret keys that are similar to Dilithium's K in that changing the keys won't affect interoperability: for example, the nonce-generation key in [**39**] or implicit-rejection keys inside typical KEMs. I'm not saying that Dilithium is uniquely vulnerable to bugs that accidentally clear K: it isn't. Bugs can appear in the analogous computations in any of these cryptosystems, and can slip past typical tests. What makes Dilithium particularly worrisome is that it has many more lines of new code being suddenly rolled out in a rush, with many more opportunities for bugs.

6 Repeated nonces in ML-DSA

The security system of the Sony PlayStation 3 had many flaws; see [**53**]. The flaws weakened security in many ways, but one particular flaw *obliterated* security: Sony was reusing the nonce inside ECDSA signatures. ECDSA signing includes a nonce that's supposed to be generated as a fresh random secret for each signature; Sony instead generated the nonce as a random secret that was reused for all signatures. This nonce reuse allowed attackers to quickly find the secret signature key and forge attacker-chosen messages.

Back in 1985, ElGamal [**69**, Note 2] had already explained the importance of generating a fresh random nonce for each signature. However, as the Sony incident illustrates, the fact that something has been *explained* doesn't mean that the software ecosystem is comprehensively *testing* for it.

Sections 6.1 and 6.2 look at how easily an analogous bug can appear in ML-DSA software and pass tests. Section 6.3 reports a very fast exploit for this bug.

6.1 How easily ML-DSA software can repeat nonces

Here's one easy way to accidentally revive the Sony PlayStation 3 vulnerability in the ML-DSA context. Recall again from Section 2 that a secret-key component called K is hashed together with the message and 32 bytes of RNG output, and that the result is transformed somehow into the secret polynomials y_0, y_1, y_2, y_3. What happens if the programmer accidentally truncates the hash input to just K, for example by simply forgetting to append the other inputs, or by miscalculating the input length? The polynomials y_0, y_1, y_2, y_3 will end up being repeated in signatures for different messages.

The hash input doesn't have to be truncated all the way down to K for the ML-DSA nonce to repeat. For example, with the "optional deterministic variant" of ML-DSA, the next 32 bytes are predictable; the programmer might accidentally truncate after those, or might accidentally compute y starting from 64 bytes of the hash *input* instead of the 64-byte hash *output*.

Someone looking at C code with a miscalculated input length—copy `K` into the beginning of a temporary buffer, copy `rnd` after that, copy `mu` after that, then call `H(rho,buf,32)` instead of `H(rho,buf,96)`—oops, wait, does this fix get the 96 wrong? can someone remind me how long `mu` is?—might reasonably recommend a language with first-class vectors and automatic vector-length calculation, allowing much more concise code such as `rho = H(K+rnd+mu)`, maybe even with enough data-flow tracking to complain that a buggy `rho = H(K)` isn't using `rnd`, maybe even in a CI environment that rejects code triggering this complaint. But trying to arrange for all cryptographic software to be written in better languages is an ongoing project, not the status quo.

6.2 How easily repeated ML-DSA nonces will pass common types of tests

Any of these bugs will pass typical tests, even though there *exist* tests that will catch these bugs. The analysis is the same as in Sections 3 and 5.

If ML-DSA were designed for signature generation to *always* be deterministic as in [**39**], then one could reasonably hope for libraries to consistently apply known-answer tests to signature generation, even without adding support for the central RNG derandomization described in Section 3.2.2. This would catch not just nonce repetition but also the AABBCC bug.

However, for ML-DSA the default is that signature generation is randomized. The basic goal of the extra randomization is to protect against side channels, but this has a tradeoff against real-world test quality: randomizing signature generation reduces the chance that libraries will apply known-answer tests. Qualitatively, this is the same as the tradeoff for ECC signatures explained in [**41**, Section 2.4], but the risks of inadequate testing are higher for ML-DSA

since ML-DSA software is newer and more complicated than software for ECC signatures. (Presumably the side-channel risks are also higher, but evaluating this is outside the scope of this paper.)

The lib25519 ECC library uses derandomization to test checksums as in SUPERCOP, uses randomized signature generation, and explains in [**45**] the importance of tying these features together: "Including new randomness also has the disadvantage of interfering with the use of test vectors. This disadvantage does not apply to lib25519: lib25519's test vectors already handle randomness." The situation is different in the larger ecosystem, with many libraries that *don't* have known-answer tests for randomized functions: allowing randomized signature generation for such libraries means that bugs are less likely to be caught.

6.3 Exploiting repeated nonces in ML-DSA

Describing an exploit of ML-DSA nonce repetition involves less notation than in Section 4, since entire polynomials are being repeated rather than individual coefficients. (For the same reason, someone writing a fast exploit can conveniently reuse the size-256 NTTs in existing ML-DSA software. The size-128 NTTs mentioned in Section 4 are more efficient but involve more code modifications.)

As in Section 4, an ML-DSA-44 signature reveals z_0, z_1, z_2, z_3 where $z_i = cs_i + y_i$, along with c being public. Also as in Section 4, the attack uses a second signature. Let's use the following notation: the second signature reveals z'_0, z'_1, z'_2, z'_3 where $z'_i = c's_i + y'_i$, along with c' being public. The effect of this section's bug is that $y'_i = y_i$. Subtracting gives $z_i - z'_i = (c - c')s_i$. Dividing the known $z_i - z'_i$ by the known $c - c'$ thus gives s_i, as long as $c - c'$ is invertible. Repeating for each i gives s_0, s_1, s_2, s_3, which are usable for forgeries as in Section 4.

There's one complication here. Recall from Section 2 that the signature generation sometimes rejects a signature and restarts with a new hash. The attack will have $y'_i = y_i$ only when there are the same number of restarts in the two final signatures. Experimentally, this happens roughly 80% of the time.[13] To check for success, recompute y_i from s_i and see whether y_i has coefficients below 2^{17} in absolute value modulo q. In failure cases, try another signature.

An attack demo in the supplement [**31**] to this paper exploits nonce repetition. This demo, like the demo in Section 4, includes forging signatures on attacker-selected messages and checking that the forged signatures are accepted. The demo has similar performance features to Section 4. I again predict that a C demo will take less than 1 ms to recover s_0, s_1, s_2, s_3, and less than 1 ms for each forged signature, but again I'm not relying on this speedup for this paper's estimates of the number of breakable keys.

[13] An attacker who can observe approximate timings of signature generation can see the number of restarts and pick signatures with the same number of restarts.

7 Estimating the number of breakable ML-DSA keys

This section returns to the starting question from Section 1. How many ML-DSA signature keys will be breakable because of exploitable bugs in ML-DSA signature software?

Qualitatively, there's a well-known process by which software bugs are created, enter production, and exit production:

- People write many initial lines of code (including, these days, many lines of AI-generated code).
- People (and AI) make mistakes—sometimes exploitable mistakes. The number of bugs is normally modeled as being linear in the volume of code, as is the number of vulnerabilities, although the literature also describes some superlinear effects.
- Some of the bugs are caught by tests or by review (including, these days, AI review) and fixed before deployment.
- Further bugs are caught and patched at variable amounts of time after deployment.

Note that this paper is focusing on *accidental* bugs, where each stage here can be reasonably modeled as a random process (with the stages not necessarily independent—e.g., a library written with above-average care might have fewer mistakes than average *and* better tests than average). Maliciously introduced bugs would instead be non-randomly designed to be exploitable and escape tests; the usual defense against those is to try to keep attackers out of the software supply chain.[14]

The rest of this section provides quantitative estimates for how often ML-DSA will go wrong. Section 7.2 estimates the number of lines of code that will be spent on ML-DSA per library supporting ML-DSA. Section 7.3 estimates the number of libraries that will provide ML-DSA, and draws conclusions about the overall number of lines of code for ML-DSA. Section 7.4 estimates the fraction of lines containing vulnerabilities that escape tests and reach deployment, and draws conclusions about the number of deployed ML-DSA vulnerabilities. Section 7.5 estimates the percentage of vulnerabilities that will be severe—allowing attackers to quickly forge signatures on attacker-chosen messages—and draws conclusions about the number of deployed ML-DSA libraries that will allow such breaks. Section 7.6 estimates the number of years where each vulnerability will be exploitable. Section 7.7 estimates the overall number of keys that an attacker will be able to break each year because of these vulnerabilities.

It would be interesting to develop a model of the damage done by breaking each key, to go beyond merely estimating the number of breakable keys. For

[14] Perhaps this implies keeping AI out of the supply chain, given the ability of attackers to manipulate AI training data. On the other hand, attackers can also manipulate humans writing software. It is most comforting to set up software structures that minimize the trusted computing base, so that security problems cannot be caused by other software, whatever the source of the other software might be.

example, consider again the problem of protecting software supply chains. Today software is getting so large that an attacker able to compromise these chains will have endless places to hide, even if some of the compromises are detected. Security-aware projects typically limit their component imports and limit the people they trust, but all of these limits rely on authenticating the supply-chain links. Signature forgeries would directly compromise many of the existing links—in essence, a free pass for attackers to suddenly take over many of the world's most important software projects, using those as a springboard for further attacks. Recovery from forgeries can be very difficult. A model of forgery damage that quantifies such effects could help justify extra investment in ensuring signature-system security.

7.1 Simplification: ignoring the risk of breaks of the ML-DSA specification

About half of the submissions to the NIST PQ competition were broken within five years; see [**24**]. These are breaks of the *specified cryptosystems*, not breaks exploiting software bugs. Sometimes these breaks were very fast, like the attacks from Sections 4 and 6.3. The original breaks of SIKE weren't so fast, but the improved SIKE break from [**63**] took a matter of seconds.

I am concerned about the risk of such attacks against ML-DSA (and I'll return to this briefly in Section 10.2), but I am also concerned that trying to include that risk in this paper's estimates would distract from assessment of a definite problem, namely that many ML-DSA keys will be breakable because of the predictable influx of ML-DSA software vulnerabilities.

This paper's estimates of the number of breakable keys assume that there will *not* be a severe break of the ML-DSA standard per se. Vastly more keys could end up breakable if there *is* a severe break of the ML-DSA standard.

7.2 Estimating the number of lines of ML-DSA code per ML-DSA library

I'm measuring non-blank non-comment lines of code (e.g., using `cloc` for C code) for comparability to [**49**], without repeatedly saying "non-blank non-comment". Investigating other metrics as in [**27**] would also be of interest.

The reference implementation of each parameter set in the Dilithium 1.0 submission [**65**] has 1333 lines of code. Here I'm not counting `*KAT*` and `*test*` and `*randombytes*` and `Makefile`; also, for this paper I'm making the questionable assumption that Keccak code is shared across all ML-DSA libraries and is bug-free, so I'm not counting `*keccak*` and `*fips202*`.

The Dilithium 1.0 code is essentially the same across all parameter sets. A few code segments are conditionally compiled for specific parameter sets, so code focusing on a particular parameter set would be slightly smaller.

The AVX2-optimized implementation has 2448 lines of code. Merging the reference and AVX2-optimized implementations produces 2580 lines of code.

This increased to 3235 in Dilithium 2.0 [66] and (with `*aes256*` also removed) 4215 in Dilithium 3.0 [7]. I haven't studied most of the code, but I noticed more assembly files being added across software versions. The reference code by itself was up to 1735 lines for Dilithium 3.0.

I next looked at the current ML-DSA code in OpenSSL (specifically in the `crypto/ml_dsa` directory), finding 2331 lines for portable code, plus 1360 lines for a Perl script generating AVX2-optimized NTT code. Glancing at the code gives me the impression that this is newly written code for OpenSSL, not derived from the official submission.

These examples show considerable variation in code sizes. In particular, more and more optimization can cause code sizes to explode. It won't be surprising to see a library optimizing multiple subroutines for multiple platforms and ending up above 10000 lines of code, but it also won't be surprising to see a library that stops with 2000 lines for portable software if that software is fast enough for applications of that library. For purposes of this paper, I'll estimate that the average number of lines of ML-DSA code per ML-DSA library will end up between 2000 and 4000.

My Sage code (without bugs added) is shorter, only 699 lines to cover all three ML-DSA parameter sets,[15] but is also not suitable for deployment, even for applications that don't need more speed: the code certainly leaks secret information through timing.

7.3 Estimating the number of ML-DSA libraries

It is more difficult to estimate how many different ML-DSA libraries will end up being deployed. Looking merely at "eight widely used cryptographic libraries" as in [49] misses a long tail of libraries that are less well known but still used in applications.

There are 18 TLS libraries currently listed in Wikipedia (Botan, BoringSSL, Bouncy Castle, BSAFE, cryptlib, GnuTLS, JSSE, LibreSSL, MatrixSSL, Mbed TLS, NSS, OpenSSL, Rustls, s2n, Schannel, Secure Transport, wolfSSL, and Erlang's SSL library), and there are many cryptographic applications beyond TLS. Further libraries mentioned in [5] that seem to cover ML-DSA include CIRCL (Go), Google Tink (C++, Go, Java, Obj-C, Python), libcrux (Rust), liboqs, noble-post-quantum (JavaScript), PQ Code Package, pqm4 (for ARM Cortex-M4 microcontrollers), RustCrypto/signatures (obviously again Rust), and std.crypto (Zig). Dozens of further software and hardware libraries supporting ML-DSA are listed in [59].

Sometimes libraries simply import ML-DSA software from upstream libraries, for example for the reasons explained in [30]. These libraries are then copying any existing upstream vulnerabilities rather than introducing potential new

[15] I recently wrote Sage scripts in a similar style for four KEM families for a separate testing project. Those scripts are not online yet, but I will mention the line counts here in case they are of interest: 309 lines for `sntrup`, 330 lines for `mlkem`, 355 lines for `frodokem`, and 665 lines for `mceliece`.

vulnerabilities. But programmers very often write new code, for example arguing that it's important to have new code to fit another language or another platform or another software desideratum.

Under the assumption from Section 7.1 that the ML-DSA standard isn't broken (and isn't quickly superseded by newer, smaller alternatives such as HAETAE from [55]), I think it's reasonable to estimate that there will be 50 ML-DSA libraries that acquire noticeable (although certainly unequal) levels of deployment over the next 5 years, overall between 100000 and 200000 lines of new code. Variations in the number of libraries would have little effect on the conclusions below; what matters is simply that there are enough new ML-DSA libraries to provide overwhelming statistical confidence that there are frequent ML-DSA software vulnerabilities.

7.4 Estimating the vulnerability rate per ML-DSA line of code

[49] focuses on cryptographic libraries that seem to be in the habit of publicly reporting vulnerabilities, as discussed in [49, Section 3.5]. Recall from Section 1.1 that the results include 0.450 CVEs per 1000 lines of code added to NSS and 1.187 CVEs per 1000 lines of code added to OpenSSL.

This paper takes these as estimates for the vulnerability rate per line of ML-DSA code when the code is initially released. (See Section 7.5 below regarding *severe* vulnerabilities, and Section 7.6 below regarding the subsequent patch rate.) Multiplying by the volume of new ML-DSA code estimated in Section 7.2 produces estimates for the number of vulnerabilities in ML-DSA code, again at the time of initial release. On the low end, multiplying 0.450 vulnerabilities per 1000 lines of code by 100000 lines predicts 45 vulnerabilities in the new ML-DSA code. On the high end, multiplying 1.187 vulnerabilities per 1000 lines of code by 200000 lines predicts 237 vulnerabilities in the new ML-DSA code.

A reasonable intermediate estimate is 100: i.e., 2 vulnerabilities on average per ML-DSA library. Higher or lower rates of vulnerabilities would quantitatively change Figure 9.1.1; Sections 7.4.1 and 7.4.2 hypothesize some reasons that ML-DSA's vulnerability rate could differ from the rates observed in [49].

7.4.1 Some reasons that the ML-DSA vulnerability rates might be higher than average. One expects public bug discovery to take more time when software is a moving target or is in a library that is perceived as having less deployment than the libraries considered in [49]. For example, there were no immediate threats to deploy the official Dilithium 1.0 reference software in 2017 (as far as I know), and the real-world relevance of that software disappeared in favor of the official Dilithium 2.0 reference software (not compatible with Dilithium 1.0) in 2019. For public researchers looking for software vulnerabilities, software with more users and more stability is generally more attractive to study.

Certainly some code is shared across Dilithium 1.0, 2.0, 3.0, 3.1, 3.3, and 3.4 (ML-DSA), so new studies of ML-DSA software might incidentally say something

about the original Dilithium code, but it's not as if the Dilithium 1.0 reference software has been a major topic of study. There's no reason to believe that the one *known* vulnerability in those 1333 lines of code is the only vulnerability, i.e., that this code had a vulnerability rate of only 0.750 per 1000 lines of code. Similarly, pointing to a new ML-DSA library with no known vulnerabilities is uninformative. As noted in, e.g., [**49**], bug rates for new software will be higher than what one sees by looking merely at *known* vulnerabilities.

I also found it worrisome to see how many lines in my ML-DSA Sage script are devoted to randomness-expansion mechanisms that automatically pass typical functionality tests no matter how buggy they are (not just the lines featured in Sections 3, 5, and 6), not to mention the diversity of arithmetic operations for which optimization will encourage subtle bugs (see [**85**]). These issues aren't qualitatively unique to ML-DSA, but quantitatively I would guess that ML-DSA software is a more fertile breeding ground for bugs than average cryptographic software.

7.4.2 Some reasons that the ML-DSA vulnerability rates might be lower than average.

I see some improvements in the cryptographic-software ecosystem. For example, there are continual increases in the number of cryptographic implementations tested by the checksums described in Section 3.2.2. As another example explained in [**30**], I am impressed by the level of verification in s2n-bignum. These improvements are, however, still far from the norm in cryptographic libraries.

I *hope* that AI-assisted code review is eliminating bugs more quickly than AI-assisted code generation is generating bugs. I won't try to quantify the AI wildcard in this paper.

7.5 Estimating the rate of severe ML-DSA vulnerabilities

In [**49**], about 1/3 of the CVEs were classified as severe ("high" or "critical" severity), but only about 1/8 of the CVEs classified as "cryptographic" were also classified as severe. I'll take the lower number, 1/8, as a model of the chance that an ML-DSA vulnerability is severe.

One can again hypothesize reasons for higher or lower rates. My impression is that vulnerabilities tend to be more severe for implementations of new cryptosystems. On the other hand, this paper's focus on vulnerabilities allowing fast forgeries of signatures on attacker-chosen messages, such as the vulnerabilities with demos in Sections 4 and 6.3, can be a more stringent notion of severity than in [**49**].

The fact that Dilithium software is generally so new means that there's very little vulnerability data at this point. (The two original Dilithium 1.0 implementations had severe vulnerabilities, the AABBCC bugs. Newer Dilithium libraries have produced eight CVEs so far, none severe; see Appendix B. The ML-DSA vulnerabilities reported in [**81**] aren't severe.) It will be interesting in 5 years to compare the 1/8 model to the data accumulated by then.

Recall from Sections 7.2, 7.3, and 7.4 that I'm estimating, based on ML-DSA line counts and general vulnerability rates, that there will be about 2 vulnerabilities in each new ML-DSA library. If only 1/8 of those vulnerabilities are severe then each new ML-DSA library will have about a 1/4 chance of a severe vulnerability. Figure 9.1.1 assumes 1/4 chance of an ML-DSA key being generated by a library that upon initial release had a severe vulnerability. See Section 7.6 regarding subsequent patches.

One of the ML-DSA vulnerabilities from [**81**] was a buffer overread. Severe vulnerabilities in ML-DSA software might include leaks of secret data from RAM, or buffer overflows that give the attacker complete control over the program. I'll model severe ML-DSA vulnerabilities as allowing just forgery for 4/5 of these vulnerabilities, while allowing complete program control for 1/5. I'll assume the same cost for either type of exploit, so this distinction does not affect this paper's estimates of the number of breakable ML-DSA keys. The distinction does matter in Section 8, which considers double-signing with ECC+ML-DSA: program control via an ML-DSA bug would allow the attacker to skip ECC signature verification.[16]

7.6 Estimating the lifetime of each ML-DSA vulnerability

Recall that [**49**, Table 3] found an "exploitable lifetime" of 5.13 years on average, standard deviation 4.29, for 201 CVEs across OpenSSL, GnuTLS, and NSS.

For estimating lifetimes of severe vulnerabilities, Figure 9.1.1 models each vulnerability as having a 20% chance of being patched each year after release, so the exploitable lifetime of the vulnerability is exactly 1 year with probability 0.2, exactly 2 years with probability 0.16, exactly 3 years with probability 0.128, etc. This geometric distribution has average 5 years, close to the 5.13 observed in [**49**], and standard deviation $\sqrt{20} \approx 4.47$ years, close to the 4.29 observed in [**49**]. See also [**2**] analyzing 11959 CVEs for 11 open-source projects, finding an average lifetime slightly over 5 years and a good fit of the lifetime distribution to an exponential distribution (although there are noticeable deviations starting around 12 years).

Recall from Section 7.5 that severe vulnerabilities are assumed to be in 25% of the ML-DSA libraries written in 2026. The drop by a factor 0.8 each year means that by 2027 only 20% of those ML-DSA libraries still have unpatched severe vulnerabilities, by 2028 only 16% have unpatched severe vulnerabilities, etc. Figure 9.1.1 models ML-DSA libraries as being evenly split between releases at the beginning of 2025 and the beginning of 2026, so by 2027 only (1/2)20% + (1/2)16% = 18% still have unpatched severe vulnerabilities. This continues to drop by a factor 0.8 each year, for example dropping to 9.2% by 2030.

[16] I'm assuming here that the application calls an integrated ECC+ML-DSA library. There *could* be program-partitioning mechanisms keeping the ECC part safe from the ML-DSA part, or preventing buffer overflows from happening in the first place—some languages enforce memory safety and other data-flow requirements—but, as noted in Section 6.1, having the whole ecosystem move to better languages is an ongoing project, not the status quo.

I see multiple reasons that this model could be too optimistic (even ignoring the AI wildcard, which could make it too optimistic or too pessimistic). One issue is that the papers [49] and [2] study well-known open-source projects; I would expect discovery delays for more obscure libraries.

Another issue is that the "lifetime" of a bug is defined in [49] by easily measured software-release dates, so the real-world lifetime can be much longer. Not all deployments instantly apply patches, even in cases where a lack of patching is publicly visible and can cause complaints and embarrassment: see, e.g., [67] measuring patch rates as a function of time for Heartbleed.

Further delays are incurred by one of the core ways to use signature systems. As background, the cryptographic software ecosystem would be simpler if signatures were eliminated in favor of public-key authenticated encryption (by the mechanism explained in, e.g., [10], [13], [17], [20, Section 8], and [114]), but this elimination assumes that all parties are online. Applications where the vouching party is *not* online are a clear motivation to support signatures—but, in the same situation, one cannot expect parties to instantly provide new signatures and public keys to address a security problem that has compromised the old signatures and public keys. Some applications require periodic rotation of signature keys, but even in those applications it is common for signatures to remain valid for a year or more. See, e.g., the payment card industry's PCI-DSS standard [60, Section 3.7.4] requiring each key to be replaced after a defined "cryptoperiod"; for quantitative guidance, [60] cites [95, Section 5.3, Table 1], which recommends using private signature keys for at most "1 to 3 years" and using public signature keys for at most "several years".

Note that these effects also increase the time needed to deploy secure software for a post-quantum signature system, even under the assumption from Section 7.1 that one has found a secure signature system in the first place. There is not just the initial software-development time and the initial rollout time, but also the delay in finding software vulnerabilities and then the time to roll out patches, along with any replacement signatures and keys.

Back in 2017, after commenting on the role of signatures in protecting software upgrades, Lange [84] wrote "Protect upgrades *now* with post-quantum signatures". Sufficient community effort to develop libraries and application integrations for conservative signature systems, followed by several years of careful auditing and then deployment, would have avoided the 2026 panic for any application that could afford those systems—e.g., for signing hashes of software upgrades. Instead the community was sidetracked into a speed competition, favoring unstable, bleeding-edge systems and deterring most of the necessary software work. There were official implementations of the early versions of Dilithium in 2017, 2019, 2020, etc., but that's only a small corner of all Dilithium software. There is endless new code for Dilithium 3.4, ML-DSA, with many opportunities for bugs that have not been discovered yet.

7.7 Estimating the number of breakable ML-DSA keys

This paper focuses on attackers that (1) have enough network access to carry out attacks—as in NSA's QUANTUMINSERT attacks, which were "highly successful" starting in 2005 (according to [98]) and were not publicly detected until the Snowden documents revealed them in 2013—and (2) are good enough at their jobs to promptly find vulnerabilities in ML-DSA software.

The number of keys broken by an attacker is still limited by the number of keys involving vulnerable software and by the attacker's computational resources for attacking those keys. Section 7.7.1 estimates the second limit; Section 7.7.2 estimates the first limit. Figure 9.1.1, which graphs this paper's estimates of the number of breakable ML-DSA keys, accounts for these limits.

7.7.1 Limits on the attacker's computational resources. Evaluating the second limit mentioned above means evaluating per-key attack costs, as in Sections 4 and 6.3, and comparing those costs to the resources available to the attacker.

For example, if 1 laptop core takes 1 second to break a key and 1 second to forge an attacker-chosen signature under that key, then an attacker armed with a current $700 laptop with 4 such cores can afford to break 2^{26} keys per year. Electricity increases the cost to only about 2^{10} dollars; see, e.g., [26, Section 4.2]. The dollar costs of communicating a key and some signatures are comparable to the dollar costs of millions of cycles of computation (see generally [26]), but this is a small fraction of a second per key.

Attackers will typically be able to buy or steal more resources. For example, 2^{20} dollars of computer equipment will break 2^{36} keys per year, and 2^{30} dollars of computer equipment will break 2^{46} keys per year. Figure 9.1.1 describes these two resource levels as "small-scale" and "large-scale" attacks respectively.

Recovering an equivalent key in Sections 4 and 6.3 is about as fast as forging one signature. Figure 9.1.1 models attacks as forging 2^{10} signatures per broken key, so the computation cost increases from about 2 seconds to about 2^{10} seconds. The limit on the number of breakable keys then drops to 2^{27} per year for 2^{20} dollars of computer equipment, or 2^{37} per year for 2^{30} dollars of computer equipment. An attack that uses more signatures would move this limit down correspondingly.

As noted earlier, I expect C code to carry out this paper's attacks much more quickly than this paper's Sage demos (but then networking costs would no longer be negligible in comparison). Also, presumably technological improvements will continue to gradually improve the price-performance ratio of computers. (Prices have recently increased in response to a surge of demand from AI companies, but presumably will drop as supply increases.) Large-scale attackers might also build attack chips more efficient than mass-market computers. Figure 9.1.1 does not account for the improvements in this paragraph.

7.7.2 Limits on the pool of vulnerable keys. The first limit mentioned above, the number of keys involving vulnerable software, can be more

constraining—not just because of some software not being vulnerable but because of limits on the total number of keys in use. Consider, e.g., [111] studying "5,499,675 keys" downloaded from PGP keyservers in 2019—including some old keys that, presumably, have been abandoned and will not be replaced with post-quantum keys.

On the other hand, PGP is just one of many applications of public-key signatures. Consider, e.g., [75] scanning 5.8 million TLS certificates and 6.2 million SSH host keys (and factoring more than 100000 of them); [35] scanning more than 2 million RSA keys from Taiwan's Citizen Digital Certificates (and factoring 184 of them); [100] estimating that "at least tens of millions" of devices used Infineon's RSA library (and factoring many RSA keys from that library); and, for an idea of the breadth of signature applications, [52] for a list of 700 "things that use Ed25519".

Figure 9.1.1 models ML-DSA applications together as having 2^{30} signature keys active at each moment. One might, for example, imagine 2^{10} different applications having their own collections of signature keys, with on average 2^{20} keys per application—obviously some applications having far fewer keys, but some applications having more.

Recall from Section 7.6 the estimate of 18% of ML-DSA keys in 2027 coming from libraries with severe software vulnerabilities. With 2^{30} total keys, this means about $2^{27.5}$ vulnerable keys. The estimated number of vulnerable keys drops by a factor 0.8 each year after that.

Having more active signature keys would move the ML-DSA curves in Figure 9.1.1 up, except to the extent limited by attacker computations. Having fewer active signature keys (for example, because of limited rollout in 2027) would move the curves down. It would be interesting to collect data regarding the actual distribution of the number of keys in many different applications.

8 Estimating the number of breakable Ed25519 keys and the number of breakable Ed25519+ML-DSA keys

Under the assumption of a sufficiently large, sufficiently reliable quantum computer, Shor's algorithm [115] breaks many popular cryptosystems, including essentially all discrete-logarithm cryptosystems, including essentially all of ECC, including Ed25519. On the other hand:

- Publicly reported quantum computers are not that large yet. Section 8.1 reviews an estimate of the timeline for public demos of quantum attacks and, more importantly, an estimate of the timeline for secret quantum attacks.
- Public cost estimates indicate that the first quantum attacks will be expensive enough to limit the number of keys that will be breakable even by large-scale attackers. See Section 8.2.

There is a separate threat from ECC software vulnerabilities. See Section 8.3. Such vulnerabilities are obviously important before the first quantum attacks, but will remain important for years after that because of the expense of quantum attacks.

8.1 Estimating the timeline for quantum computers

In 2023, I gave a talk titled "Post-quantum cryptography: risk assessment" [**23**] for the Federal Reserve TechLab. I started with the basic risk that "attackers in year Y have a large enough quantum computer to break RSA-2048 with Shor's algorithm".

I noted that the 2022 Global Risk Institute survey[17] included, at one extreme, someone predicting 50% risk as soon as $Y = 2027$, and, at the other extreme, someone predicting an under-30% risk by 2052, while the median prediction was a 50% risk by 2037. I also reported my own assessment, namely that the risk "reaches 50% in $Y = 2029$; 50% for public demonstration in $Y = 2032$".[18]

Why were my predictions earlier than the predictions of most (not all) experts in that survey? I pointed out "two common mistakes analyzing this risk". The first mistake was "assuming attackers aren't ahead of us". The second was "watching advances in #qubits and in qubit error rates but not in algorithms". As background, quantum attacks are normally structured in three layers:

- At the top layer, there are quantum algorithms such as Shor's algorithm. Each quantum algorithm is built from some number of quantum bits, "qubits", carrying out some number of qubit operations. (As an analogy, conventional computations are built from bit operations.) The literature presents many improvements beyond Shor's original algorithm; the improvements reduce the required number of qubits and/or qubit operations.
- At the next layer—also algorithmic—there is quantum error correction. Quantum error correction is a way to combine "physical qubits" with only about 99% reliability into "logical qubits" that are reliable enough to plug into quantum algorithms at the top layer. The literature presents many improvements in quantum error correction, such as reducing the number of physical qubits per logical qubit and reducing the number of physical qubit operations involved in a logical qubit operation. The reason there's a cutoff around 99%, depending on error-correction details, is that otherwise the error-correction process itself creates more errors than it corrects.
- At the bottom layer there are quantum-computer engineers aiming to build physical qubits that meet the requirements of the higher layers. The physical qubits have to meet the 99% reliability level (or preferably even better to allow more efficient error correction), and there have to be enough physical qubits hooked together to create enough logical qubits for the

[17] See [**93**] for the 2024 edition of the survey. These surveys include a list of names of the experts surveyed, but individual survey responses were anonymized by default. There does not appear to be a public record of who was predicting 2027, or of who was predicting beyond 2052.

[18] These quotes disprove the following "no one" claim from Valsorda [**125**] in 2026: "Heather Adkins and Sophie Schmieg are telling us that 'quantum frontiers may be closer than they appear' and that **2029** is their deadline. That's in 33 months, and no one had set such an aggressive timeline until this month." (Boldface in original.) It's less clear which expectations Westerbaan [**127**] is referring to in saying "much sooner than expected".

top-layer quantum algorithm. The literature presents many improvements in engineering physical qubits.

See, e.g., [56], [72], and [50].

Starting in 2021, Jaques has issued useful yearly "Landscape of quantum computing" graphs [77] surveying what quantum-computer engineers report for the achieved number of qubits and reliability rates, compared to the qubits and reliability rates that would enable a recent error-corrected quantum attack against RSA-2048 (and against other targets). Each graph shows (via shades of gray) how the engineering is improving over time—but shows only one snapshot of the algorithmic layers of the attack. I objected in January 2023 that this was misleading since in fact the algorithmic layers are a moving target. Jaques responded "I don't expect much movement in those lines so I'm glad the diagram conveys that". See [22].

I was tracking improvements in all three layers. All three layers were unstable; recent improvements were not just in quantum-computer engineering but also in quantum error correction and in quantum attacks against, e.g., RSA-2048. Further improvements after 2023 prompted visually large changes in subsequent graphs in [77] and an erratum from Jaques—but each graph continues to show just one algorithmic snapshot; readers have to flip between graphs to see the issue. Overall the improvements are staying in line with what I've been predicting for years; my median estimates remain 2029 for secret attacks and 2032 for public attacks, as in [23].

There's much more that can be said about the evidence in the literature, about how to quantitatively extrapolate timelines from that, and about the remaining uncertainties. For example, I still see some room for improvements in quantum attacks against 256-bit ECC, and more room for improvements in (currently slower) quantum attacks against RSA-2048. I've been monitoring public information for many years regarding investments by attackers (see, e.g., my presentation [15] from 2012) and building models to predict how far ahead the attackers will be as a result, but these predictions involve large error bars; I wouldn't be surprised by the attacker advantage being 1 year or 5 years instead of my median estimate of 3 years.

This paper models a large-scale attacker as having a quantum computer in 2029 big enough and reliable enough to start breaking Ed25519 keys. Changing by a year or two wouldn't make dramatic changes in Figure 9.1.1 below.

8.2 Estimating the cost per Ed25519 key of a quantum attack

A Reuters report in May 2026 [106] stated that IBM "plans to invest more than $10 billion in quantum computing over five years as it aims to build by 2029 the first large-scale quantum computer capable of running complex calculations reliably and without errors". This is consistent with earlier reports regarding trends in public quantum investments by various companies.

Let's estimate that IBM's 2029 computer—after $6 billion are spent—will be capable of running a long series of computations on a few logical qubits, and that

by around 2032—after \$10 billion are spent—IBM will have scaled the number of qubits up enough to run cryptographic attacks. This is consistent with the timeline estimates from Section 8.1, but now let's focus on costs.

IBM's first quantum computer demonstrating a real cryptographic break won't cost the full \$10 billion. Some of the costs are non-recurring engineering costs: research, building prototypes, etc. Figure 9.1.1 assumes that the attacker's first quantum computer—already in 2029—costs only \$1 billion to build and run.

How many keys will this quantum computer break? Starting with a plausible model of such a computer, a March 2026 paper [6] concludes that each secp256k1 key can be broken in 18 minutes, i.e., $2^{-14.8}$ years. Figure 9.1.1 assumes that the attacker's secret quantum computer in 2029 will have essentially this size and speed, breaking 2^{15} Ed25519 keys that year.

Figure 9.1.1 also models costs as dropping by a factor 2 each year, through a combination of algorithmic improvements and engineering improvements (as in Section 8.1), so 2^{30} dollars spent on quantum attacks in 2030 will break 2^{16} keys, 2^{30} dollars spent on quantum attacks in 2031 will break 2^{17} keys, etc. Note that an attacker carrying out each of these attacks over a stretch of 10 years will spend $10 \cdot 2^{30}$ dollars, whereas the non-quantum computers in Figure 9.1.1 will probably not need to be replaced every year; recall that Section 7 does not assume any cost improvements for non-quantum computers.

8.3 Estimating vulnerability rates for Ed25519 software

The security of ECC software sometimes fails even without quantum computers, although the chance of this happening depends on the ECC details; see [42] for a survey. The rest of this section reviews CVEs specifically for Ed25519 software, and estimates the frequency of an Ed25519 key being breakable via software vulnerabilities in the future.

8.3.1 Ed25519 CVE review. Ed25519 is a well-known signature system with many years of deployment in a wide range of applications—see again [52] listing 700 "things that use Ed25519". Ed25519 libraries listed in [52] include libsodium since 2013, libgcrypt/GnuPG since 2014, wolfSSL since 2015, Nettle since 2015, BoringSSL since 2015, etc.

Given the general lifetime data from [49] (see also [2] and Section 7.6), an *initial* rate of, say, 1/4 of Ed25519 libraries having severe vulnerabilities would be visible as many severe vulnerabilities that are *known* by now. (For comparison, the much lower age of most Dilithium libraries limits the data available so far regarding Dilithium vulnerabilities, making Dilithium vulnerability rates much more difficult to directly test; see Section 7.4.)

I searched for CVEs mentioning Ed25519 or EdDSA, finding 25 for Ed25519 and 18 for EdDSA, with a few overlaps. Only three of these CVEs report potentially severe vulnerabilities in Ed25519 libraries. (I've listed and classified the other CVEs in Appendix C for double-checking.) Evidently the initial rate of severe vulnerabilities per Ed25519 library was well below 1/4, and the *current* rate of severe vulnerabilities is likely even lower.

The following paragraphs look more closely at these three announcements of potentially severe vulnerabilities.

CVE-2017-9526. This CVE, not backed by an exploit demo, claims the following timing-attack vulnerability: "In Libgcrypt before 1.7.7, an attacker who learns the EdDSA session key (from side-channel observation during the signing process) can easily recover the long-term secret key".

The literature explains how even a minor timing leak of the length of DSA or ECDSA nonces is a security disaster; see, e.g., [**109**]. One might leap to the conclusion that this is also true for EdDSA. However, I had already explained in 2014 [**16**] that the double-size nonces in Ed25519 would make a length leak unexploitable in simple Ed25519 libraries, libraries that don't go to extra effort to reduce nonces. It turned out that libgcrypt was in that situation; see [**21**] for details. CVE-2017-9526 wasn't justifying its claim that a libgcrypt side-channel leak would reveal the nonce ("session key"), the long-term secret key, or anything else allowing forgeries.

I don't mean to suggest that the variable-time computations in libgcrypt were acceptable. It's certainly possible that these were exploitable, even though the argument for this stated in the CVE was incorrect. Timing leaks should be fixed even without an analysis of exploitability.

CVE-2021-20305. This CVE, also not backed by an exploit demo, claims a potential vulnerability from a bug: "several Nettle signature verification functions (GOST DSA, EDDSA & ECDSA) result in the Elliptic Curve Cryptography point (ECC) multiply function being called with out-of-range scalers, possibly resulting in incorrect results".

The usual issue with out-of-range scalars in ECC is that scalar multiplication can then pass through exceptional cases in elliptic-curve addition formulas. This issue doesn't apply to the *complete* Edwards-curve addition formulas from [**40**] typically used in Ed25519 libraries; see [**42**, Section 10]. Checking the Nettle code, I see that the addition formula cited in `ecc-add-ehh.c` is complete. Presumably CVE-2021-20305 wasn't exploitable.

CVE-2026-3562. This CVE was reportedly exploited at Pwn2Own Ireland 2025: see [**57**], which says that the "Philips Hue Bridge" was hacked via "a crypto bypass and a heap overflow". The CVE says that the "Philips Hue Bridge" allowed "network-adjacent attackers to execute arbitrary code" because of "improper verification" of Ed25519 signatures.

This sounds like a disastrous case of a programmer forgetting to return verification errors, so the attacker could (1) trivially forge signatures and then (2) attack vulnerable application code that would otherwise have been shielded by the signatures. However, one can also consider the possibility of a program takeover via a heap overflow *in the verification software*. For this paper, I'll assume that each of these disaster cases occurs on occasion. This assumption is quantified in Section 8.3.3 below.

Summary. Out of the CVEs mentioning Ed25519 or EdDSA, 1 or possibly 2 are severe vulnerabilities.

8.3.2 Interlude: A quadratic effect of software stability. Consider two collections of software, collection E and collection D, that have had the same number of vulnerabilities discovered by now (e.g., 90 vs. 90).

Assume one major difference between E and D: collection D is newly written code with only 10% of its vulnerabilities discovered by now, while collection E has been around for much longer and has had 90% of its vulnerabilities discovered by now. Evidently the initial release of E had 9× fewer vulnerabilities than the initial release of D (e.g., 100 vulnerabilities in E vs. 900 in D).

Also assume that patches are issued promptly for known vulnerabilities. Then 90% of the E vulnerabilities have patches by now while only 10% of the D vulnerabilities have patches by now (e.g., 90 patches out of 100 vulnerabilities vs. 90 patches out of 900 vulnerabilities), so overall the vulnerability rate in the current patched E software is 81× lower than the vulnerability rate in the current patched D software (e.g., $100 - 90 = 10$ remaining vulnerabilities vs. $900 - 90 = 810$ remaining vulnerabilities).

If the objective is to determine the vulnerability rate per library, then one has to scale by the number of libraries in E and D respectively. If the objective is to determine the vulnerability rate per key, then all of the computations have to be weighted by the keys per library. These are only linear effects.

8.3.3 Ed25519 vulnerability estimates. I'll estimate that Ed25519 libraries are old enough on average (weighted by keys) that vulnerabilities have about a 3/4 chance of being found and announced by now. (I'm not assuming the 90% chance from Section 8.3.2.) Recall the model from Section 7.6 of each vulnerability having a 20% chance of being discovered each year; 3/4 chance means about 6 years at that discovery rate.

If the number of severe vulnerabilities in the initial software releases was 5 or more, each discovered by now with 3/4 chance, then there would be more than a 98% chance of 3 or more severe vulnerabilities being discovered by now. But the CVE review from Section 8.3.1 found 1 or possibly 2 severe vulnerabilities. So I'll estimate that there were originally 4 severe vulnerabilities, i.e., 2 severe vulnerabilities not yet discovered.

That's across all of the Ed25519 libraries, including obscure implementations such as the Philips Hue Bridge. I'll estimate 100 Ed25519 libraries overall, with a 2% rate of remaining severe vulnerabilities. Accordingly, Figure 9.1.1 estimates that an average Ed25519 key has a 2% chance of coming from an Ed25519 library that still has a severe vulnerability in 2026. With such a small number of severe vulnerabilities, it would be interesting to consider a discrete model that includes the possibility of eliminating all of the remaining severe vulnerabilities; Figure 9.1.1 instead averages over possible futures, treating the decay of vulnerabilities as multiplying the number of vulnerable keys by 0.8 every year.

I'll model severe Ed25519 vulnerabilities as having 1/5 chance per vulnerability of being buffer overflows or otherwise making Ed25519+ML-DSA breakable even when ML-DSA is not, the same way that Section 7.5 modeled severe ML-DSA vulnerabilities as each having 1/5 chance of making Ed25519+ML-DSA breakable even when Ed25519 is not. There are also a few lines of combiner code in which bugs could skip checking parts of an Ed25519+ML-DSA signature; this doesn't have a qualitatively different effect from bugs considered above in larger volumes of code, so I won't quantify this separately.

9 Comparisons

Section 9.1 presents a graph, Figure 9.1.1, summarizing this paper's estimates of the number of keys breakable in each year. Section 9.2 compares safety of Ed25519 and ML-DSA. Section 9.3 compares safety of Ed25519+ML-DSA and ML-DSA. Section 9.4 compares safety of Ed25519+ML-DSA and Ed25519. Each of these comparisons includes a robustness analysis. Section 9.5 makes recommendations.

9.1 Graphing the estimates

There are six curves in Figure 9.1.1. There are three attack scenarios: attacking a pool of 2^{30} Ed25519 signature keys, attacking a pool of 2^{30} ML-DSA signature keys, and attacking a pool of 2^{30} Ed25519+ML-DSA signature keys. Each scenario is split across a red curve for large-scale attackers and a blue curve for small-scale attackers.

Recall that Sections 8.1 and 8.2 model a large-scale attacker as already being able to build a secret billion-dollar quantum computer by 2029, with the computer fast enough to break 2^{15} ECC keys in a year. Each subsequent year is modeled as doubling the capacity of a billion-dollar quantum attack, so a billion dollars break 2^{25} ECC keys in 2039. This doubling is what produces the upwards slope for the red curve in the number of breakable Ed25519 keys.

The small-scale attacker is limited to 2^{20} dollars of computer equipment. If a quantum computer becomes $2\times$ cheaper each year while still breaking 2^{15} ECC keys per quantum computer per year (rather than the $2\times$ cost improvement coming partially from an increase in the number of ECC keys broken per quantum computer per year), then the small-scale attacker would just barely be able to afford a quantum computer in 2039, breaking just 2^{15} ECC keys with it. Figure 9.1.1 ignores this possibility and instead treats the small-scale attacker as not having a quantum computer.

The reason the small-scale attacker can break many keys even without a quantum computer is the main story of this paper: software bugs. This is also why the large-scale attacker can break many keys before the 2029 advent of a secret quantum computer. This is also why, for the first 5 or 6 years of quantum

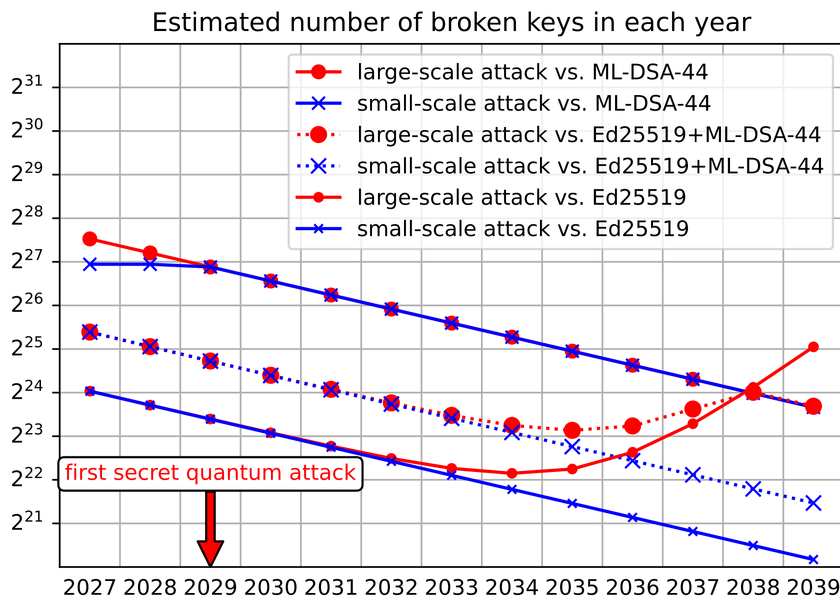

Fig. 9.1.1. Estimate of the number of keys breakable in year Y, as a function of Y.

attacks after that, more breakable keys are coming from bugs than from quantum attacks.

Quantitatively, recall from Section 7.7.2 the estimate of $0.18 \cdot 2^{30} \approx 2^{27.5}$ ML-DSA keys that will be breakable in 2027 because of severe software vulnerabilities. The small-scale attacker's limited computer power, in the model from Section 7.7.1, allows attacks against only 2^{27} of those keys; this is what produces the horizontal segment in one curve in Figure 9.1.1 (actually slightly below 2^{27} because there are slightly fewer than 2^{25} seconds in a year). The downward slopes elsewhere come from the model of 20% of vulnerabilities being fixed each year.

Also recall from Section 8.3.3 the estimate of a 2% breakability rate for Ed25519 keys in 2026, and thus 1.6% in 2027, i.e., $0.016 \cdot 2^{30} \approx 2^{24}$ Ed25519 keys breakable in 2027. For Ed25519+ML-DSA, the calculation is more complicated:

- Recall that 1/5 of severe software vulnerabilities are modeled as neutralizing the other signature system too (for example, via buffer overflows). For example, in total 18% of ML-DSA keys are breakable, so (1/5)18% = 3.6% of ML-DSA keys have Ed25519+ML-DSA broken no matter how secure Ed25519 would have been by itself.
- The other 14.4% of ML-DSA keys are breakable in a way that Ed25519+ML-DSA can rescue via Ed25519 staying secure. This still fails for 1.6% of the Ed25519 keys, the ones with their own vulnerabilities:

Ed25519+ML-DSA is broken by a joint vulnerability with probability $0.016 \cdot 14.4\% = 0.2304\%$.

- The 82% of ML-DSA keys that aren't breakable can still have Ed25519+ML-DSA compromised by an Ed25519 software bug that also neutralizes ML-DSA. This happens for 1/5 of severe Ed25519 vulnerabilities, i.e., $(1/5)1.6\% = 0.32\%$ of Ed25519 keys, overall producing breaks in a further $0.0032 \cdot 82\% = 0.2624\%$ of Ed25519+ML-DSA keys.

Overall $3.6\% + 0.2304\% + 0.2624\% = 4.0928\%$ of Ed25519+ML-DSA keys are breakable in 2027, about $2^{25.4}$ Ed25519+ML-DSA keys overall.

For spot-checks, the graphing script included in the supplement [31] to this paper prints out the numbers that it's graphing. For Ed25519+ML-DSA (`pq True ecc True`), for 2027, it prints out 43946105.37267201. This matches 4.0928% of 2^{30}. (I'm not truncating these numbers to one or two digits—that would make spot-checks less effective.)

The middle red curve, large-scale attacks against Ed25519+ML-DSA, has a down-up-down pattern as the years progress:

- The initial down is because the percentage of keys breakable by software vulnerabilities drops as vulnerabilities are found and fixed.
- The up is because quantum attacks are gradually breaking more Ed25519 keys.
- There is eventually another down because the number of vulnerable ML-DSA keys drops below the number of Ed25519 keys that the attacker can afford to break with a quantum attack.

For example, in 2035 (8 years after the 4.0928% example above), the large-scale attacker breaks $0.8^8 \cdot 3.6\% \approx 0.6040\%$ of Ed25519+ML-DSA keys via ML-DSA buffer overflows etc., breaks $0.8^{16} \cdot 0.2304\% \approx 0.0065\%$ of keys via other ML-DSA vulnerabilities combined with Ed25519 vulnerabilities, breaks $(1 - 0.8^8 \cdot 18\%) \cdot 0.8^8 \cdot (1/5)1.6\% \approx 0.0521\%$ of keys where ML-DSA is fine but there are Ed25519 buffer overflows etc., and breaks another 2^{21} keys via other ML-DSA vulnerabilities combined with targeting the corresponding (otherwise unbroken) Ed25519 keys with quantum attacks.

As a spot-check, $0.8^8 \cdot 3.6\% + 0.8^{16} \cdot 0.2304\% + (1 - 0.8^8 \cdot 18\%) \cdot 0.8^8 \cdot (1/5)1.6\%$ is exactly 0.66253075479756013568%, and what the graphing script prints for `pq True ecc True large True` for 2035 is 9211021.811124295, which matches $2^{21} + 0.0066253075479756013568 \cdot 2^{30}$.

9.2 Comparing safety of Ed25519 and ML-DSA

In Figure 9.1.1, "upgrading" the pool of signature keys from Ed25519 to ML-DSA within the next 5 years would damage security. This "upgrade" means an order of magnitude more keys broken in the short term by small-scale attackers and by large-scale attackers. Quantum attacks will eventually be fast enough for large-scale attackers to break more Ed25519 keys than ML-DSA keys, but only a decade *after* the first quantum computer.

To evaluate how robust the conclusion of security damage is against changes in the underlying numbers, let's look at how implausible those changes would have to be to flip the conclusion.

The basic reason for the large increase in the number of keys broken is that the frequency of severe software vulnerabilities during 2027, 2028, etc. is modeled as much higher for ML-DSA than for Ed25519. Can one justify a claim that, no, the frequency is the same (or even lower)?

The starting obstacle here is the vulnerability-lifetime data from [**49**] and [**2**], where software vulnerabilities often remain undetected for 5 or more years. ML-DSA is suddenly adding many new lines of software. Ed25519 software has typically been around for much longer, giving many more opportunities for vulnerabilities to be discovered.

One could try pointing to the 5077 lines in `crypto/ec/curve25519.c` in OpenSSL, compared to the 2331 lines mentioned in Section 7.2 for OpenSSL's portable ML-DSA code (in both cases non-blank non-comment lines counted by `cloc`). However, even if it's possible to argue that being half the size is enough to compensate for how new typical ML-DSA software is, the size difference doesn't stand up to scrutiny. In `crypto/ec/curve25519.c`, 2114 lines for a table `k25519Precomp` aren't actually source code—they were auto-generated by a 65-line Python script—so there are only 3028 source lines. Furthermore, `curve25519.c` implements X25519 key exchange *and* Ed25519 signatures, so those 3028 source lines should be compared to 2331 + 1305 = 3636 lines for ML-DSA plus ML-KEM, minus a few discounts for auto-generated tables such as the 34-line `zetas_montgomery` table in `crypto/ml_dsa/ml_dsa_ntt.c`.

Here's another attempt: argue that, despite the similar line counts, Ed25519 software is more prone to severe vulnerabilities than average code, while ML-DSA software isn't, again compensating for how new typical ML-DSA software is. But there's already enough Ed25519 data to contradict the first part of this (see Section 8.3.1). Also, where's the mechanism that would make Ed25519 more bug-prone than ML-DSA? Ed25519 software might have overflow bugs; same for ML-DSA (see [**85**]). Ed25519 software might have bugs repeating nonces; ML-DSA can too, and discourages the relevant tests (see Section 6). Ed25519 software might have bugs producing predictable nonces; same for ML-DSA (see Section 5). Meanwhile Section 3 shows how easily bugs can appear in various lines of ML-DSA code that don't seem to correspond to any lines in Ed25519.

The reason that the conclusion flips in the late 2030s in Figure 9.1.1 is that a 2^{30}-dollar quantum attack in 2039 is modeled as breaking 2^{25} ECC keys per year, i.e., being 2^{10} times faster than the currently hypothetical quantum computer described in [**6**]. Assuming that this speed is available in 2029 rather than 2039 would be vastly more extreme than merely assuming that the attacker can start carrying out the first quantum attacks in 2029—and this *still* wouldn't flip the comparison until 2031.

9.3 Comparing safety of Ed25519+ML-DSA and ML-DSA

In Figure 9.1.1, using ML-DSA rather than Ed25519+ML-DSA within the next 5 years would damage security. It would mean several times more keys broken in the short term by small-scale attackers and by large-scale attackers.

The conclusion that ML-DSA damages security for the next 5 years compared to Ed25519+ML-DSA is even more robust than Section 9.2's conclusion regarding ML-DSA vs. Ed25519. Here's the basic point: even if one imagines the rate of ML-DSA vulnerabilities somehow being as low as the rate of Ed25519 vulnerabilities, Ed25519+ML-DSA usually forces the attacker to break ML-DSA *and* Ed25519, which is less likely than being able to break just ML-DSA.

There's an exception: recall that this paper models a severe vulnerability in ML-DSA or in Ed25519 as having 1/5 chance of allowing the attacker to skip breaking the other algorithm, for example via a buffer overflow. But having this exception increase the Ed25519+ML-DSA breakability rate up to the ML-DSA breakability rate would require Ed25519 to have vulnerabilities *more* often than ML-DSA.

Figure 9.1.1 shows the comparison flipping a decade after the first quantum computer—allowing about 2% more keys to be broken at that point with Ed25519+ML-DSA than with ML-DSA. The attacker isn't quite limited to the cases of ML-DSA vulnerabilities: again, an Ed25519 vulnerability is modeled as sometimes letting the attacker skip breaking ML-DSA. The 2% is 1/5 of the ratio between the Ed25519 vulnerability rate and the ML-DSA vulnerability rate.

9.4 Comparing safety of Ed25519+ML-DSA and Ed25519

In Figure 9.1.1, using Ed25519+ML-DSA rather than Ed25519 within the next 5 years would damage security. However, this conclusion is *not* robust against changes in the underlying numbers.

This comparison is mainly between two competing factors. In one direction, a severe ML-DSA vulnerability, starting probability 18% in 2027, could be in the $(1/5)18\% = 3.6\%$ case of allowing an Ed25519+ML-DSA key to be broken without the Ed25519 key being broken. In the opposite direction, a severe Ed25519 vulnerability, starting probability 1.6% in 2027, could be in the $(1 - 1/5)1.6\% = 1.28\%$ case of still requiring the attacker to break ML-DSA.

Moderate adjustments in these numbers don't matter for the comparisons in Sections 9.2 and 9.3, but they do matter for this comparison. For example, if 1/5 is replaced by 1/20 then the second effect is more important than the first.

One can object that more than 1/20 of severe vulnerabilities are buffer overflows in general software and even in cryptographic libraries; see [**49**]. However, inside low-level implementations of cryptographic primitives, eliminating data flow from attacker-controlled data to array indices is often feasible (see, e.g., [**37**]), generally desirable (see, e.g., [**12**]), and sometimes convincingly tested (see, e.g., [**29**, Section 5]). This has the side effect of eliminating exploitable buffer overflows for that code. These protections are far from complete—consider again Kobeissi [**81**] in 2026 finding a buffer overread in

"verified" ML-DSA software—but it still seems possible that buffer overflows are gradually becoming less prevalent in this type of software, while various other types of severe vulnerabilities in cryptographic software don't seem to be going away.

9.5 Recommendations

Seeing manifold mechanisms by which cryptography fails to do its job suggests a corresponding variety of ways to reduce risks. For example, cryptosystem designers should modify cryptosystems to be more robust against programmer mistakes; see, e.g., [**42**]. Test developers should look for systematic ways to catch more bugs, such as RNG derandomization for known-answer tests as in Section 3.2.2. Applications choosing cryptosystems should account not just for the safety of the specified cryptosystems but also for how frequently bugs will occur in software for those cryptosystems; see, e.g., Section 7.

The rest of this section looks more closely at signature-system selection among Ed25519, ML-DSA, and Ed25519+ML-DSA.

One part of this is easy. Using solo ML-DSA rather than Ed25519+ML-DSA would damage security, so don't do that.

The more difficult question is whether Ed25519+ML-DSA is a safer option than sticking to solo Ed25519. There are clear risks of quantum attacks, but there are also clear risks of attacks exploiting the vulnerabilities that will arise from deploying large volumes of new ML-DSA code in a panic, and if those vulnerabilities include buffer overflows then more keys could end up broken. Figure 9.1.1 estimates a larger impact from the second risk; but, as explained in Section 9.4, this conclusion can be reversed by moderate changes in the model. So I don't have a recommendation at this point between Ed25519 and Ed25519+ML-DSA.

The way I try to settle thorny risk-management questions is to find or build technological workarounds that resolve the underlying tension. For example, consider ML-DSA software in a language that guarantees proper isolation for the software, including protection against buffer overflows. I would expect using Ed25519+ML-DSA *with such software* to be less risky than sticking to Ed25519. Exploiting other vulnerabilities in the ML-DSA software (or in the ML-DSA standard) will still face the attacker with the problem of breaking Ed25519, except in the unusual case of an exploitable bug in a few lines of combiner code. This assessment isn't changed by the risk of languages failing to meet their guarantees: that risk has to be compared to the risk of attackers already having quantum computers.

Considering further cryptographic deployment options, including further cryptosystems, can also reduce risks. I'll close this section with one example, namely PQConnect [**44**], which has the deployment advantage of not requiring changes to applications; it's adding an external cryptographic layer that's designed to protect confidentiality and integrity even when TLS and other protocols used by the applications fail to do so. The software supports various isolation mechanisms to prevent bugs in the software from damaging security;

I'm comfortably running the software on my primary servers. The PQ choices and protocol choices in PQConnect are different from current usage of TLS—for example, PQConnect's long-term server keys are X25519+McEliece keys used for authenticated encryption rather than signature keys—which should reduce the risk of an attacker using a correlated software bug to break both.

10 A survey of flawed security comparisons between PQ and ECC+PQ

This section quotes a representative sample of recent arguments that ECC+PQ does not reduce security risks compared to PQ. Each argument was given in the context of either ML-KEM or ML-DSA or both as a choice of PQ system. This section identifies a fatal flaw in each argument.

10.1 Denying the risks of software bugs for ML-KEM and ML-DSA

Valsorda [**125**] wrote the following: "Why trust the new stuff? ... On the implementation side, I am actually very qualified to have an opinion, having made cryptography implementation and testing my niche. ML-KEM and ML-DSA are a lot easier to implement securely than their classical alternatives, and with the better testing infrastructure we have now I expect to see exceedingly few bugs in their implementations." This argument is flawed on multiple levels.

The claim about being "a lot easier to implement securely" is unsubstantiated, ignores the history of security failures in software for these supposedly safe algorithms (see, e.g., [**87**] and [**33**]), is difficult to reconcile with how complicated these algorithms are (see [**27**], Section 7.2, and Section 9.2), and ignores the amply documented importance of software maturity (see, e.g., [**49**]). The claim is also focusing on comparing PQ to ECC, which is not the same as comparing PQ to ECC+PQ (see Section 9.3) and not the same as saying PQ is safe.

The "better testing infrastructure" cited in [**125**] is [**58**]. As noted in Section 3.2.5, [**58**] seems to require a nonstandard interface to test ML-DSA signature generation, and it doesn't test ML-DSA key generation at all. Better tests have already been available for many years—and yet we've seen recent bugs in ML-DSA software. For example, the "verified" ML-DSA code in libcrux at the beginning of 2026 used an incorrect `2<<EXPONENT` instead of `1<<EXPONENT` and an incorrect `current` instead of `previous`, as pointed out in [**81**]. Reactively patching [**58**] to address one bug at a time doesn't address the systemic problem: the ecosystem is adding more and more lines of ML-DSA code where bugs won't be caught by the tests actually being applied to that code.

A single software bug can be devastating. For claiming that ML-DSA should be trusted and is as safe as ECC+ML-DSA, claiming merely that there will be "exceedingly few bugs" isn't good enough. The same posting [**125**] said "a single broken key per month can be catastrophic" and said that a disaster chance above 1% is unacceptable since "you are betting with your users' lives"; it's incoherent for the posting to set a target of "exceedingly few bugs" rather than zero bugs.

Meanwhile the lack of quantification means that the "exceedingly few bugs" claim isn't falsifiable. It isn't even obvious that announcements of 10 severe vulnerabilities in ML-DSA software would cause a retraction of [**125**]'s conclusion: one could say that this is "exceedingly few" compared to, say, the number of bugs ever discovered in cryptographic software.

10.2 Denying the risks of mathematical breaks of ML-KEM and ML-DSA

There were claims that lattice PQ systems, or specifically Kyber (ML-KEM) and Dilithium (ML-DSA), had been thoroughly studied, eliminating the risk of breaks and thus eliminating the value of ECC+PQ.

For example, Kobeissi [**82**] described the NIST post-quantum competition as follows: "hundreds of researchers from dozens of countries actively tried to break every candidate ... The lattice candidates survived." Apon [**3**] wrote the following: "There has been a full decade of entirely open, international analysis and debate over the security of quantum-resistant cryptography through the NIST PQC process. ML-KEM was fully vetted through this process ... No new cryptanalyses have been offered."

In fact, some of the lattice candidates were broken by very fast attacks. See [**34**] breaking the IND-CCA2 claim for HILA5, for example, or [**11**] breaking Round2. So it is not true that "the lattice candidates survived".

Furthermore, the statement that "hundreds of researchers from dozens of countries actively tried to break every candidate" wildly exaggerates the level of scrutiny applied specifically to, e.g., Dilithium. Certainly hundreds of researchers are involved in public cryptanalysis of post-quantum systems, but this work is spread across several different types of post-quantum systems, and across many proposed cryptosystems of each type. Even after NIST narrowed its lattice candidates to Dilithium, Falcon, Kyber, NTRU, and SABER, lattice papers typically looked at general lattice problems, considering specific candidates only to make tables of numerical results for those sizes of lattice problems.

Dilithium was also a moving target during the NIST competition, modifying various details to try to increase security levels after improvements in lattice attacks reduced the security of the previous versions. See in particular [**7**] discarding Dilithium's "level 1" parameter set and adding a "level 5" parameter set. The earlier versions of Dilithium didn't survive; "tried to break" understates the damage done.

The reader understands "ML-KEM was fully vetted through this process" to mean that ML-KEM was proposed from the outset and then studied for "a full decade". But Kyber, like Dilithium, was a moving target. The final ML-KEM, like the final ML-DSA, appeared much later.

"No new cryptanalyses have been offered" is incorrect. There is a continued drumbeat of advances in lattice attacks. See, e.g., [**79**] from October 2025 and [**101**] from February 2026, these papers speed up attacks against short-secret lattice problems and against structured lattice problems, reducing the security levels of a wide range of short-secret lattice cryptosystems, including ML-KEM

and ML-DSA. It is unfortunate that NIST does not provide an official tracker of the security losses in its standards.

More broadly, these claims from [**82**] and [**3**] are trying to argue that *the cryptosystem specifications* are safe,[19] but what deployments need is for *the deployed software* to be safe. Bugs usually change the software into software for another cryptosystem, often a new cryptosystem allowing fast attacks that do not apply to the specified cryptosystem, as illustrated by this paper's demos.[20]

10.3 Denying the plans to use PQ rather than ECC+PQ

In IETF, there are documents specifying solo ML-KEM for TLS key exchange, documents specifying ECC+ML-KEM for TLS key exchange, documents specifying solo ML-DSA for TLS signatures, and documents specifying ECC+ML-DSA for TLS signatures.

In late 2025, regarding key exchange, Schmieg [**112**] claimed that the planned users of ML-KEM, rather than ECC+ML-KEM, were NSA and nobody but NSA: "In the end, yes, hybrids are the best way to go, and indeed, this is what the IETF enabled people to do. . . . Well it turns out there is one customer who really really hates hybrids . . . And that customer happens to be the NSA . . . if you really think ML-KEM is broken, then yes, the NSA has successfully undermined the IETF in order to make their own systems less secure, while not impacting anyone else."

The claim that NSA "hates hybrids" seems impossible to reconcile with [**99**], an official NSA document describing an NSA program to use two independent encryption layers "to mitigate the ability of an adversary to exploit a single cryptographic implementation". The claim about what others would use was contradicted by, e.g., Apon [**3**]: "I represent a major vendor in the space of secure communications. For our primary communications platform, we intend to offer ML-KEM-only as the straightforward implementation of TLS for our systems, at $100B+ scale." But the original statement hasn't been withdrawn.

The 2026 panic produced a completely different story regarding ML-DSA: there was suddenly a "last call" for solo ML-DSA for TLS signatures, there were claims that everyone should immediately use solo ML-DSA, and there were objections to ECC+ML-DSA as supposedly slowing down urgent deployment of ML-DSA. See, e.g., [**125**].

In fact, a detailed specification of ECC+ML-DSA for TLS is already available (see [**105**]). The new code it uses is almost entirely the new code used for solo ML-DSA. Furthermore, even if there's so much delay that ECC signature keys remain non-upgraded by the time an attacker has a quantum computer, the damage will still be limited because of the limited number of ECC keys that the quantum computer will break; see Section 8. Meanwhile using solo ML-DSA will expose far more keys; see Figure 9.1.1.

[19] One wonders whether this safety claim includes the earlier—discarded—versions of Dilithium and Kyber.

[20] Software can also be broken by timing attacks, but, as noted earlier, this paper focuses primarily on bugs.

10.4 Shifting comparison baselines

Kobeissi [82] gave the following three-step argument that ECC+PQ is better than PQ for KEMs but not for signatures. The fatal flaw is in the third step.

First, [82] said "When an adversary captures ciphertext today, they can store it indefinitely and decrypt it once a cryptographically relevant quantum computer (CRQC) arrives"; [82] emphasized the urgency of protecting against this "store now, exploit later" attack. Indeed, users sending confidential data in 2026 will often be very unhappy about an attacker seeing the data in 2036.

Second, [82] said that "A quantum adversary who arrives in 2036 cannot retroactively forge a signature that was verified and acted upon in 2026"; [82] concluded that "There is no stockpile of signatures waiting to be broken".

This is overstated. At each moment the attacker has a stockpile of attack targets, including still-valid public keys for signature systems. Furthermore, the validity time varies across applications; some applications want signatures to stay valid for decades. As an extreme example, the signer of a last will often isn't available to generate a new signature. Last wills are a classic target of forgeries and should be cryptographically protected.

However, it's true that many applications rotate keys on a shorter schedule. It's also true that a key expiration *narrows* the stockpile: a key that expires a year from now can't be attacked by a quantum computer two years from now (as long as clocks aren't severely desynchronized). This is quantitatively different from the situation of users asking for 10 or more years of confidentiality. I've seen many more people talking about the "store now, exploit later" attack than about protecting long-lived signatures.

Third, [82] described "store now, exploit later" as "the threat model that justifies hybrid KEMs", and on this basis concluded that "the harvest-now-decrypt-later (HNDL) threat that motivates hybrid KEMs has no analogue for signatures", so "hybrid signatures are far less necessary" than "hybrid KEMs" for "post-quantum native design".

What are ECC+PQ hybrid KEMs being *compared to* in this third step? Consider the following two comparisons:

- Solo PQ and ECC+PQ have an advantage *compared to solo ECC*, namely mitigating the risk of ECC being broken by quantum computers.
- ECC+PQ has an advantage *compared to solo PQ*, namely mitigating the risk of PQ breaks.

Even if one narrows the first comparison to ECC+PQ vs. ECC (skipping solo PQ), the first comparison is not the same as the second comparison. The baselines that ECC+PQ is being compared to are different: solo ECC in the first comparison, solo PQ in the second comparison. The attack resources are different: quantum computers in the first comparison, usually non-quantum computers in the second comparison (see, e.g., [24], [33], and Section 4). Quantum attacks against ECC still seem years away, so the first comparison is regularly backed by arguments for the urgency of handling a future threat, whereas PQ attacks are frequently demonstrated on current computers.

Saying "When an adversary captures ciphertext today, they can store it indefinitely and decrypt it once a cryptographically relevant quantum computer (CRQC) arrives" and saying that this is "the threat model that justifies hybrid KEMs" was presenting a KEM-specific rationale for ECC+PQ rather than *solo ECC*. [**82**] omitted the comparison baseline and slid into incorrectly portraying this as a KEM-specific rationale for ECC+PQ *instead of solo PQ*. The conclusion from [**82**] that "hybrid signatures are far less necessary" than "hybrid KEMs" for "post-quantum native design" was talking about comparisons of ECC+PQ to *solo PQ*.

The actual motivation for ECC+PQ over solo PQ is the risk of PQ breaks from problems in PQ specifications or in PQ software. Many of the PQ breaks so far (in, e.g., [**24**]) are for signatures; this risk is not KEM-specific. The shift of baseline in [**82**] avoided addressing this risk.

10.5 Denying the continued value of ECC

Valsorda [**125**] wrote that the "only benefit" of ECC+ML-DSA is "protection if ML-DSA is classically broken *before the CRQCs come*" (emphasis in original). Here "CRQCs" means "cryptographically relevant quantum computers"—a confusing phrase that can be interpreted in many different ways, but in context it seems to be referring to the first quantum computers powerful enough to break an ECC key.

In fact, there are three basic ways that ECC+ML-DSA can rescue broken ML-DSA keys:

- Delaying quantum attacks until the attacker has a quantum computer.
- Limiting those attacks to attackers who can afford a quantum computer.
- Even after attackers have quantum computers, limiting the number of keys per year that those computers can break, simply because of the expense of breaking each key.

See Section 8 for quantification. The "only benefit" statement in [**125**] denies the second and third mechanisms—not by giving an argument that the first quantum computers will have low cost, but by simply ignoring the quantitative cost question.

Furthermore, keeping ECC would be amply justified even if the only benefit of ECC were delaying quantum attacks, rescuing security for the next year or two or three in many cases of breakable ML-DSA keys.

The posting [**125**] said that the quantum threat demands immediate action ("Concretely, what does this mean? It means we need to ship ... we've got to roll out what we have"). The immediate action that it advocated was *turning off ECC* in favor of solo ML-DSA, rather than signing with ML-DSA *and* continuing to sign with ECC to mitigate the risk of ML-DSA keys being broken. The posting admitted that this will add "protection if ML-DSA is classically broken" soon, so the posting was betting that a break won't happen. But this is the same posting that said that "a single broken key per month can be catastrophic" and that "you are betting with your users' lives". This is again incoherent.

10.6 Denying that forgeries matter

Finally, specifically in the context of signatures, there were arguments that (1) seemed to be admitting that solo ML-DSA will allow forgeries because of ML-DSA bugs, and (2) seemed to be admitting that ECC+ML-DSA will sometimes stop those forgeries, but (3) denied the importance of forgeries. For example, Schmieg [**113**] wrote that “the blast radius for signatures has a strict end with revocation of the key” and that “this is good enough in most threat models” so “I don’t consider hybrid signatures essential in those use cases”.

Revocation of a key means that all computers that would otherwise trust a key stop trusting it. Attackers can no longer forge signatures under that key *after* the revocation. But forgeries can do tremendous damage *before* the revocation.

Consider Iran’s DigiNotar attack, which according to a post-mortem [**70**] allowed “successful man-in-the-middle attacks against hundreds of thousands of Internet users inside and outside of Iran”. The fact that this particular attack was eventually detected—and that the relevant DigiNotar keys were eventually revoked—doesn’t change the fact that the attack was tremendously damaging *before* it was detected. As another example (also mentioned in Section 7.7), NSA said in [**98**] that its QUANTUMINSERT attacks were “highly successful” starting in 2005; those attacks were not publicly detected until the Snowden documents revealed them in 2013.

The primary defenses later added against these two types of attacks are better control over TLS certificate authorities and more use of TLS. Signature forgeries would compromise both of these defenses. When signature software is completely broken, as in Sections 4 and 6.3, TLS using that software provides no protection for the integrity of data and no protection for the confidentiality of data.

Some applications, such as the aforementioned [**60**], periodically revoke signature keys. Normally the keys are replaced with new keys for the same signature system. This rotation of keys accomplishes nothing if the signature software is completely broken.

Furthermore, forging a signature often allows such severe system compromise that there is no longer any convincing way to carry out revocation. For example, ssh accounts are typically used for remote system administration; if an attacker uses signature forgeries to break ssh authentication then I would expect the attacker to have persistent hard-to-detect access installed (as in, e.g., [**119**]) within a matter of seconds after logging in. As another example, see Section 7 regarding software supply chains.

I’m not saying that revocation is useless. Revocation sometimes reduces the damage caused by forgeries. But forgeries remain tremendously damaging, and ECC+PQ reduces this damage compared to solo PQ.

References

[1] — (no editor), *32nd annual network and distributed system security symposium, NDSS 2025, San Diego, California, USA, February 24–28, 2025*, The Internet Society, 2025. URL: https://www.ndss-symposium.org/ndss2025/. See [44].

[2] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser, *How long do vulnerabilities live in the code? A large-scale empirical measurement study on FOSS vulnerability lifetimes*, in Proceedings of the 31st USENIX Security Symposium (2022), 359–376. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos. Cited in §7.6, §7.6, §8.3.1, §9.2.

[3] Daniel Apon, *Re: Status of ML-KEM WGLC* (2026). URL: https://archive.cr.yp.to/2026-03-24/02:49:57/lf-LpxkTZyi0VoeuZS0icL2JESxqwRgSnqUelbtl4Uw/https/mailarchive.ietf.org/arch/msg/tls/nVeE4qhVAnLCOfGZcNloENNF34Q/. Cited in §10.2, §10.2, §10.3.

[4] Apple Security Engineering and Architecture (SEAR) and Hardware Technologies Formal Verification, *A blueprint for formal verification of Apple corecrypto* (2026). URL: https://security.apple.com/blog/formal-verification-corecrypto/. Cited in §3.1.

[5] Jean-Philippe Aumasson (editor), *Awesome post-quantum* (2026). URL: https://web.archive.org/web/20260504062413/https://github.com/veorq/awesome-post-quantum. Cited in §7.3.

[6] Ryan Babbush, Adam Zalcman, Craig Gidney, Michael Broughton, Tanuj Khattar, Hartmut Neven, Thiago Bergamaschi, Justin Drake, and Dan Boneh, *Securing elliptic curve cryptocurrencies against quantum vulnerabilities: resource estimates and mitigations* (2026). URL: https://web.archive.org/web/20260331050640/https://quantumai.google/static/site-assets/downloads/cryptocurrency-whitepaper.pdf. Cited in §8.2, §9.2.

[7] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé, *CRYSTALS-DILITHIUM* (2020). URL: https://web.archive.org/web/20220705164000/https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Dilithium-Round3.zip. Cited in §1.2, §7.2, §10.2.

[8] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé, *CRYSTALS-DILITHIUM* (2021). URL: https://web.archive.org/web/20211110120400/https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf. Cited in §1.2.

[9] Magali Bardet and Ruben Niederhagen (editors), *Post-Quantum cryptography—17th international workshop, PQCrypto 2026, Saint-Malo, France, April 14–16, 2026, proceedings, part II*, Lecture Notes in Computer Science, 16492, Springer, 2026. ISBN 978-3-032-22697-6. DOI: 10.1007/978-3-032-22698-3. See [64].

[10] Mihir Bellare, Ran Canetti, and Hugo Krawczyk, *A modular approach to the design and analysis of authentication and key exchange protocols (extended abstract)*, in STOC 1998 [**126**] (1998), 419–428. URL: https://cseweb.ucsd.edu/~mihir/papers/modular.pdf. DOI: 10.1145/276698.276854. Cited in §7.6.

[11] Mihir Bellare, Hannah Davis, and Felix Günther, *Separate your domains: NIST PQC KEMs, oracle cloning and read-only indifferentiability*, in Eurocrypt 2020.2 [**54**] (2020), 3–32. URL: https://eprint.iacr.org/2020/241. Cited in §10.2.

[12] Daniel J. Bernstein, *Cache-timing attacks on AES* (2005). URL: https://cr.yp.to/papers.html#cachetiming. Cited in §9.4.

[13] Daniel J. Bernstein, *Elliptic vs. hyperelliptic, part 1* (2006). URL: https://cr.yp.to/talks.html#2006.09.20. Cited in §7.6.

[14] Daniel J. Bernstein, *supercop-20111120 diffs* (2011). URL: https://web.archive.org/web/20260525173247/https://github.com/floodyberry/supercop/commit/5c3843095364c13c5222513fb06c712f45ef1ba7. Cited in §3.2.2.

[15] Daniel J. Bernstein, *Cryptography for the paranoid* (2012). URL: https://cr.yp.to/talks.html#2012.09.24. Cited in §8.1.

[16] Daniel J. Bernstein, *Re: Mishandling twist attacks* (2014). URL: https://web.archive.org/web/20210615201318/https://mailarchive.ietf.org/arch/msg/cfrg/8z3ZcujGRxFSGEBI-uE7C1tjw4c/. Cited in §8.3.1.

[17] Daniel J. Bernstein, *The post-quantum Internet* (2016). URL: https://cr.yp.to/talks.html#2016.02.24. Cited in §1.4, §7.6.

[18] Daniel J. Bernstein, *Benchmarking post-quantum cryptography: News regarding the SUPERCOP benchmarking system, and more recommendations to NIST* (2017). URL: https://blog.cr.yp.to/20170719-pqbench.html. Cited in §3.2.2.

[19] Daniel J. Bernstein, *Fast-key-erasure random-number generators* (2017). URL: https://blog.cr.yp.to/20170723-random.html. Cited in §3.2.2.

[20] Daniel J. Bernstein, *Internet: Integration* (2018). URL: https://pqcrypto.eu.org/deliverables/d2.5.pdf. Cited in §7.6, §A.4.

[21] Daniel J. Bernstein, *Why EdDSA held up better than ECDSA against Minerva: Cryptosystem designers successfully predicting, and protecting against, implementation failures* (2019). URL: https://blog.cr.yp.to/20191024-eddsa.html. Cited in §8.3.1.

[22] Daniel J. Bernstein, *It's fascinating to see how the historical data in the bottom-left corner of the graph in https://sam-jaques.appspot.com/quantum_landscape_2022 (from @sejaques, aka @sejaques@ioc.exchange) leads readers to guess the number of years to the top right without realizing that the top right is a moving target.* (2023), screenshot including response from Jaques. URL: https://web.archive.org/web/20240428222307/https://archive.ph/d8I3D. Cited in §8.1.

[23] Daniel J. Bernstein, *Post-quantum cryptography: risk assessment* (2023). URL: https://cr.yp.to/talks.html#2023.06.15. Cited in §8.1, §8.1.

[24] Daniel J. Bernstein, *Quantifying risks in cryptographic selection processes* (2023). URL: https://cr.yp.to/papers.html#qrcsp. Cited in §7.1, §10.4, §10.4, §A.2.

[25] Daniel J. Bernstein, *Double encryption: Analyzing the NSA/GCHQ arguments against hybrids* (2024). URL: https://blog.cr.yp.to/20240102-hybrid.html. Cited in §1.4.

[26] Daniel J. Bernstein, *Predicting performance for post-quantum encrypted-file systems* (2024). URL: https://cr.yp.to/papers.html#pppqefs. Cited in §7.7.1, §7.7.1.

[27] Daniel J. Bernstein, *Analyzing the complexity of reference post-quantum software: the case of lattice-based KEMs* (2024). URL: https://cr.yp.to/papers.html#pqcomplexity. Cited in §1.1, §7.2, §10.1.

[28] Daniel J. Bernstein, *Classic McEliece: conservative code-based cryptography* (2024). URL: https://cr.yp.to/talks.html#2024.09.17. Cited in §3.2.4.

[29] Daniel J. Bernstein, *The* `cryptoint` *library* (2025). URL: https://cr.yp.to/papers.html#cryptoint. Cited in §9.4.

[30] Daniel J. Bernstein, *Fast, constant-time, correct: pick three* (2025). URL: https://cr.yp.to/talks.html#2025.10.07. Cited in §7.3, §7.4.2.

[31] Daniel J. Bernstein, *mldsabugs-20260601* (2026), supplement to this paper. URL: https://cr.yp.to/2026/mldsabugs-20260601.tar.gz. Cited in §1.3, §2, §3.2.5, §4.2, §6.3, §9.1.

[32] Daniel J. Bernstein, *One of the OpenSSL disasters announced last week (CVE-2025-15469) is really the fault of OpenSSL's detached-signature interface. With a signed-message/message-recovery interface, the bug would have had no effect on security, and would have been easier to catch. Interfaces matter.* (2026). URL: https://microblog.cr.yp.to/1770277954/. Cited in §B, §C.

[33] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada, *KyberSlash: exploiting secret-dependent division timings in Kyber implementations*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2025.2** (2025), 209–234. URL: https://cr.yp.to/papers.html#kyberslash. DOI: 10.46586/tches.v2025.i2.209-234. Cited in §10.1, §10.4.

[34] Daniel J. Bernstein, Leon Groot Bruinderink, Tanja Lange, and Lorenz Panny, *HILA5 Pindakaas: On the CCA security of lattice-based encryption with error correction*, in Africacrypt 2018 [**78**] (2018), 203–216. URL: https://cr.yp.to/papers.html#hila5. Cited in §10.2.

[35] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren, *Factoring RSA keys from certified smart cards: Coppersmith in the wild*, in Asiacrypt 2013 [**110**] (2013), 341–360. URL: https://cr.yp.to/papers.html#smartfacts. DOI: 10.1007/978-3-642-42045-0_18. Cited in §7.7.2.

[36] Daniel J. Bernstein and Tung Chou, *libmceliece* (2025), accessed 2026-05-31. URL: https://lib.mceliece.org. Cited in §3.2.4.

[37] Daniel J. Bernstein, Tung Chou, and Peter Schwabe, *McBits: fast constant-time code-based cryptography*, in CHES 2013 [**46**] (2013), 250–272. URL: https://cr.yp.to/papers.html#mcbits. Cited in §9.4.

[38] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang, *High-speed high-security signatures*, in CHES 2011 [**104**] (2011), 124–142; see also newer version [**39**].

[39] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering **2** (2012), 77–89; see also older version [**38**]. URL: https://eprint.iacr.org/2011/368. Cited in §5, §6.2.

[40] Daniel J. Bernstein and Tanja Lange, *Faster addition and doubling on elliptic curves*, in Asiacrypt 2007 [**83**] (2007), 29–50. URL: https://eprint.iacr.org/2007/286. Cited in §8.3.1.

[41] Daniel J. Bernstein and Tanja Lange, *Failures in NIST's ECC standards, part 2* (2020). URL: https://web.archive.org/web/20230806153700/https://csrc.nist.gov/files/pubs/sp/800/186/final/docs/sp800-186-draft-comments-received.pdf. Cited in §6.2.

[42] Daniel J. Bernstein and Tanja Lange, *Safe curves for elliptic-curve cryptography*, in Dawson proceedings [**51**] (2025), 124–191. URL: https://eprint.iacr.org/2024/1265. DOI: 10.1007/978-3-031-83490-5_7. Cited in §1.1, §1.1, §8.3, §8.3.1, §9.5.

[43] Daniel J. Bernstein and Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems* (2026), accessed 2026-05-31. URL: https://bench.cr.yp.to. Cited in §3.2.2.

[44] Daniel J. Bernstein, Tanja Lange, Jonathan Levin, and Bo-Yin Yang, *PQConnect: automated post-quantum end-to-end tunnels*, in NDSS 2025 [1] (2025). URL: https://www.ndss-symposium.org/ndss-paper/pqconnect-automated-post-quantum-end-to-end-tunnels/. Cited in §9.5.

[45] Daniel J. Bernstein and Kaushik Nath, *lib25519: Security* (2024), accessed 2026-05-31. URL: https://lib25519.cr.yp.to/security.html. Cited in §6.2.

[46] Guido Bertoni and Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, Lecture Notes in Computer Science, 8086, Springer, 2013. ISBN 978-3-642-40348-4. See [37].

[47] Alexandre Berzati, Andersson Calle Viera, Maya Chartouny, Steven Madec, Damien Vergnaud, and David Vigilant, *Exploiting intermediate value leakage in Dilithium: A template-based approach*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2023.4** (2023), 188–210. DOI: 10.46586/tches.v2023.i4.188-210. Cited in §B.

[48] G. R. Blakley and David Chaum (editors), *Advances in cryptology, proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19–22, 1984, proceedings*, Lecture Notes in Computer Science, 196, Springer, Berlin, 1985. ISBN 3-540-15658-5. MR 86j:94003. See [68].

[49] Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner, *You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries* (2021). URL: https://arxiv.org/abs/2107.04940. Cited in §1.1, §1.1, §1.1, §1.1, §1.1, §1.1, §7.2, §7.3, §7.4, §7.4, §7.4, §7.4.1, §7.4.1, §7.5, §7.5, §7.6, §7.6, §7.6, §7.6, §7.6, §8.3.1, §9.2, §9.4, §10.1.

[50] Dolev Bluvstein, Alexandra A. Geim, Sophie H. Li, Simon J. Evered, J. Pablo Bonilla Ataides, Gefen Baranes, Andi Gu, Tom Manovitz, Muqing Xu, Marcin Kalinowski, Shayan Majidy, Christian Kokail, Nishad Maskara, Elias C. Trapp, Luke M. Stewart, Simon Hollerith, Hengyun Zhou, Michael J. Gullans, Susanne F. Yelin, Markus Greiner, Vladan Vuletić, Madelyn Cain, and Mikhail D. Lukin, *A fault-tolerant neutral-atom architecture for universal quantum computation*, Nature **649** (2026), 39–46. URL: https://www.nature.com/articles/s41586-025-09848-5. Cited in §8.1.

[51] Colin Boyd, Reihaneh Safavi-Naini, and Leonie Simpson (editors), *Information security in a connected world: celebrating the life and work of Ed Dawson*, Lecture Notes in Computer Science, 15600, Springer, 2025. ISBN 978-3-031-83489-9. DOI: 10.1007/978-3-031-83490-5. See [42].

[52] Nicolai Brown, *Things that use Ed25519* (2026), accessed 2026-05-31. URL: https://ianix.com/pub/ed25519-deployment.html. Cited in §7.7.2, §8.3.1, §8.3.1.

[53] "Bushing", Hector Martin "marcan" Cantero, Segher Boessenkool, and Sven Peter, *PS3 epic fail* (2010). URL: https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf. Cited in §1.3, §6.

[54] Anne Canteaut and Yuval Ishai (editors), *Advances in cryptology—EUROCRYPT 2020—39th annual international conference on the theory and applications of cryptographic techniques, Zagreb, Croatia, May 10–14, 2020, proceedings, part II*, Lecture Notes in Computer Science, 12106, Springer, 2020. ISBN 978-3-030-45723-5. See [11].

[55] Jung Hee Cheon, Hyeongmin Choe, Julien Devevey, Tim Güneysu, Dongyeon Hong, Markus Krausz, Georg Land, Marc Möller, Damien Stehlé, and MinJune

Yi, *HAETAE: shorter lattice-based Fiat-Shamir signatures*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2024.3** (2024), 25–75. DOI: 10.46586/tches.v2024.i3.25-75. Cited in §7.3.

[56] Clémence Chevignard, Pierre-Alain Fouque, and André Schrottenloher, *Reducing the number of qubits in quantum discrete logarithms on elliptic curves*, in Eurocrypt 2026.1 [**61**] (2026), 371–401. URL: https://eprint.iacr.org/2026/280. DOI: 10.1007/978-3-032-25291-3_13. Cited in §8.1.

[57] Dustin Childs, *Pwn2Own Ireland 2025: day three and master of pwn* (2025). URL: https://web.archive.org/web/20251025013815/https://www.zerodayinitiative.com/blog/2025/10/23/pwn2own-ireland-2025-day-three-and-master-of-pwn. Cited in §8.3.1.

[58] Community Cryptography Specification Project, *Project Wycheproof tests crypto libraries against known attacks* (2026), accessed 2026-05-31. URL: https://github.com/C2SP/wycheproof. Cited in §3.2.5, §3.2.5, §3.2.5, §3.2.5, §3.2.5, §5, §10.1, §10.1, §10.1.

[59] PKI Consortium, *PQC Capabilities Matrix (PQCCM)* (2026). URL: https://web.archive.org/web/20260421013046/https://pkic.org/wg/pqc/pqccm/. Cited in §7.3.

[60] PCI Security Standards Council, *Payment Card Industry Data Security Standard: Requirements and Testing Procedures* (2024), version 4.0.1. URL: https://web.archive.org/web/20250526070822/https://www.middlebury.edu/sites/default/files/2025-01/PCI-DSS-v4_0_1.pdf?fv=AKHVQBp6. Cited in §7.6, §7.6, §10.6.

[61] Joan Daemen and Emmanuel Thomé (editors), *Advances in cryptology—EUROCRYPT 2026—45th annual international conference on the theory and applications of cryptographic techniques, Rome, Italy, May 10–14, 2026, proceedings, part I*, Lecture Notes in Computer Science, 16541, Springer, 2026. ISBN 978-3-032-25290-6. DOI: 10.1007/978-3-032-25291-3. See [56].

[62] Joan Daemen and Emmanuel Thomé (editors), *Advances in cryptology—EUROCRYPT 2026—45th annual international conference on the theory and applications of cryptographic techniques, Rome, Italy, May 10–14, 2026, proceedings, part II*, Lecture Notes in Computer Science, 16542, Springer, 2026. ISBN 978-3-032-25316-3. DOI: 10.1007/978-3-032-25317-0. See [76].

[63] Thomas Decru and Sabrina Kunzweiler, *Efficient computation of $(3^n, 3^n)$-isogenies*, in Africacrypt 2023 [**94**] (2023), 53–78. URL: https://eprint.iacr.org/2023/376. Cited in §7.1.

[64] Julien Devevey, Morgane Guerreau, and Maxime Roméas, *Compact, efficient and non-separable hybrid signatures*, in PQCrypto 2026 [**9**] (2026), 143–177. URL: https://eprint.iacr.org/2025/2059. DOI: 10.1007/978-3-032-22698-3_5. Cited in §A.4.

[65] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé, *CRYSTALS-DILITHIUM* (2017). URL: https://web.archive.org/web/20220705163949/https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/CRYSTALS_Dilithium.zip. Cited in §1.2, §1.2, §7.2.

[66] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé, *CRYSTALS-DILITHIUM* (2019). URL: https://web.archive.org/web/20220705163951/https://csrc.nist.gov/

CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/CRYSTALS-Dilithium-Round2.zip. Cited in §1.2, §7.2.

[67] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael D. Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson, *The matter of Heartbleed*, in IMC 2014 [**128**] (2014), 475–488. DOI: 10.1145/2663716.2663755. Cited in §7.6.

[68] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, in Crypto '84 [**48**] (1985), 10–18; see also newer version [**69**]. MR 87b:94037.

[69] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory **31** (1985), 469–472; see also older version [**68**]. ISSN 0018-9448. MR 86j:94045. Cited in §6.

[70] Eva Galperin, Seth Schoen, and Peter Eckersley, *A post mortem on the Iranian DigiNotar attack* (2011). URL: https://www.eff.org/deeplinks/2011/09/post-mortem-iranian-diginotar-attack. Cited in §10.6.

[71] Ruofan Gao, Amjed Tahir, Peng Liang, Teo Susnjak, and Foutse Khomh, *A survey of bugs in AI-generated code* (2025). URL: https://arxiv.org/abs/2512.05239. Cited in §1.1.

[72] Craig Gidney, Noah Shutty, and Cody Jones, *Magic state cultivation: growing T states as cheap as CNOT gates* (2024). URL: https://arxiv.org/abs/2409.17595. Cited in §8.1.

[73] Shafi Goldwasser (editor), *35th annual IEEE symposium on the foundations of computer science. Proceedings of the IEEE symposium held in Santa Fe, NM, November 20–22, 1994*, IEEE, 1994. ISBN 0-8186-6580-7. MR 98h:68008. See [115].

[74] Goichiro Hanaoka and Bo-Yin Yang (editors), *Advances in cryptology—ASIACRYPT 2025—31st international conference on the theory and application of cryptology and information security, Melbourne, VIC, Australia, December 8–12, 2025, proceedings, part III*, Lecture Notes in Computer Science, 16247, Springer, 2026. ISBN 978-981-95-5098-2. DOI: 10.1007/978-981-95-5099-9. See [79].

[75] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman, *Mining your Ps and Qs: detection of widespread weak keys in network devices*, in Proceedings of the 21st USENIX Security Symposium (2012). URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger. Cited in §7.7.2.

[76] Jonas Janneck, *Bird of prey: practical signature combiners preserving strong unforgeability*, in Eurocrypt 2026.2 [**62**] (2026), 222–250. URL: https://eprint.iacr.org/2025/1844. DOI: 10.1007/978-3-032-25317-0_8. Cited in §A.4.

[77] Sam Jaques, *Landscape of quantum computing* (2026), accessed 2026-05-31. URL: https://sam-jaques.appspot.com/quantum_landscape. Cited in §8.1, §8.1.

[78] Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi (editors), *Progress in cryptology—AFRICACRYPT 2018—10th international conference on cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, proceedings*, Lecture Notes in Computer Science, 10831, Springer, 2018. ISBN 978-3-319-89338-9. See [34].

[79] Alexander Karenin, Elena Kirshanova, Julian Nowakowski, and Alexander May, *Fast slicer for Batch-CVP: making lattice hybrid attacks practical*, in Asiacrypt 2025 [**74**] (2025), 100–132. URL: https://eprint.iacr.org/2025/1910. DOI: 10.1007/978-981-95-5099-9_4. Cited in §10.2.

[80] Franziskus Kiefer and Karthikeyan Bhargavan, *Formally verified post-quantum cryptography* (2024). URL: https://web.archive.org/web/20240907101344/https://cryspen.com/post/fospqc/. Cited in §1.2.

[81] Nadim Kobeissi, *Verification theatre: false assurance in formally verified cryptographic libraries* (2026). URL: https://eprint.iacr.org/2026/192. Cited in §1.2, §1.2, §1.3, §3.1, §7.5, §7.5, §9.4, §10.1, §A.4.

[82] Nadim Kobeissi, *Hybrid constructions are a safety blanket, and that's fine* (2026). URL: https://web.archive.org/web/20260418021002/https://symbolic.software/blog/2026-04-13-hybrid-constructions/. Cited in §10.2, §10.2, §10.4, §10.4, §10.4, §10.4, §10.4, §10.4, §10.4, §10.4, §10.4.

[83] Kaoru Kurosawa (editor), *Advances in cryptology—ASIACRYPT 2007, 13th international conference on the theory and application of cryptology and information security, Kuching, Malaysia, December 2–6, 2007, proceedings*, Lecture Notes in Computer Science, 4833, Springer, 2007. ISBN 978-3-540-76899-9. See [40].

[84] Tanja Lange, *Practical post-quantum cryptography* (2017). URL: https://hyperelliptic.org/tanja/vortraege/SIAM-AG-17.pdf. Cited in §7.6.

[85] Sunwoo Lee, Hyuk Lim, and Seunghyun Yoon, *When removing reductions goes wrong: auditing reduction placement in production ML-DSA implementations* (2026). URL: https://eprint.iacr.org/2026/1032. Cited in §1.2, §1.2, §3.2, §7.4.1, §9.2.

[86] Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (editors), *CCS '20: 2020 ACM SIGSAC conference on computer and communications security, virtual event, USA, November 9–13, 2020*, ACM, 2020. ISBN 978-1-4503-7089-9. DOI: 10.1145/3372297. See [114].

[87] Vadim Lyubashevsky, *OFFICIAL COMMENT: CRYSTALS-DILITHIUM* (2018). URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/aWxC2ynJDLE/m/YOsMJ2ewAAAJ. Cited in §1.2, §1.2, §1.2, §1.3, §3, §3, §3, §10.1, §A.1.

[88] Vadim Lyubashevsky, *Basic lattice cryptography: the concepts behind Kyber (ML-KEM) and Dilithium (ML-DSA)* (2024). URL: https://eprint.iacr.org/2024/1287. Cited in §1.2.

[89] Alfred Menezes, *A gentle introduction to lattice-based cryptography* (2026). URL: https://eprint.iacr.org/2026/1098. Cited in §1.2.

[90] MITRE Corporation, *CRYSTALS-DILITHIUM (in Post-Quantum Cryptography Selected Algorithms 2022) in PQClean d03da30 may allow universal forgeries of digital signatures via a template side-channel attack because of intermediate data leakage of one vector* (2023). URL: https://www.cve.org/CVERecord?id=CVE-2023-24025. Cited in §B.

[91] MITRE Corporation, *An issue in Open Quantum Safe liboqs v.10.0 allows a remote attacker to escalate privileges via the crypto_sign_signature parameter in the /pqcrystals-dilithium-standard_ml-dsa-44-ipd_avx2/sign.c component* (2024). URL: https://www.cve.org/CVERecord?id=CVE-2024-31510. Cited in §B.

[92] MITRE Corporation, *Libgcrypt before 1.12.2 mishandles Dilithium signing. Writes to a static array lack a bounds check but do not use attacker-controlled data* (2026). URL: https://www.cve.org/CVERecord?id=CVE-2026-41990. Cited in §B.

[93] Michele Mosca and Marco Piani, *Quantum threat timeline report 2024* (2024). URL: https://globalriskinstitute.org/publication/2024-quantum-threat-timeline-report/. Cited in §8.1.

[94] Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne (editors), *Progress in cryptology—AFRICACRYPT 2023—14th international conference on cryptology in Africa, Sousse, Tunisia, July 19–21, 2023, proceedings*, Lecture Notes in Computer Science, 14064, Springer, 2023. ISBN 978-3-031-37678-8. See [63].

[95] National Institute of Standards and Technology, *NIST SP 800-57 part 1 rev. 5: recommendation for key management: part 1 – general* (2020). URL: https://csrc.nist.gov/pubs/sp/800/57/pt1/r5/final. Cited in §7.6.

[96] National Institute of Standards and Technology, *FIPS 204 (draft): Module-lattice-based digital signature standard* (2023). URL: https://web.archive.org/web/20230824124107/https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.ipd.pdf. Cited in §1.2.

[97] National Institute of Standards and Technology, *FIPS 204: Module-lattice-based digital signature standard* (2024). URL: https://web.archive.org/web/20240813093957/https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf. Cited in §1.2, §3.1.

[98] National Security Agency, *There is more than one way to QUANTUM* (2014). URL: https://www.eff.org/files/2014/04/09/20140312-intercept-multiple_methods_of_quantum.pdf. Cited in §7.7, §10.6.

[99] National Security Agency, *Commercial Solutions for Classified (CSfC) threat prevention* (2018). URL: https://web.archive.org/web/20220524232250/https://www.nsa.gov/Portals/75/documents/resources/everyone/csfc/threat-prevention.pdf. Cited in §10.3.

[100] Matús Nemec, Marek Sýs, Petr Svenda, Dusan Klinec, and Vashek Matyas, *The return of Coppersmith's attack: practical factorization of widely used RSA moduli*, in CCS 2017 [**121**] (2017), 1631–1648. DOI: 10.1145/3133956.3133969. Cited in §7.7.2.

[101] Tabitha Ogilvie, *On the concrete hardness gap between MLWE and LWE* (2026). URL: https://eprint.iacr.org/2026/279. Cited in §10.2.

[102] OpenSSL Software Foundation, *BN_mod_exp may produce incorrect results on x86_64* (2017). URL: https://www.cve.org/CVERecord?id=CVE-2017-3732. Cited in §1.1.

[103] OpenSSL Software Foundation, *'openssl dgst' one-shot codepath silently truncates inputs >16MB* (2026). URL: https://www.cve.org/CVERecord?id=CVE-2025-15469. Cited in §B.

[104] Bart Preneel and Tsuyoshi Takagi (editors), *Cryptographic Hardware and Embedded Systems—CHES 2011—13th international workshop, Nara, Japan, September 28–October 1, 2011, proceedings*, Lecture Notes in Computer Science, 6917, Springer, 2011. ISBN 978-3-642-23950-2. See [38].

[105] Tirumaleswar Reddy.K, Tim Hollebeek, John Gray, Scott Fluhrer, and Daniel Van Geest, *Use of composite ML-DSA in TLS 1.3* (2026). URL: https://web.archive.org/web/20260528193432/https://datatracker.ietf.org/doc/draft-reddy-tls-composite-mldsa/. Cited in §10.3.

[106] Reuters, *IBM to invest $10 billion for large-scale quantum computer by 2029* (2026). URL: https://www.reuters.com/technology/ibm-plans-10-billion-investment-large-scale-quantum-computer-by-2029-2026-05-28/. Cited in §8.2.

[107] RustCrypto, *Signatures has timing side-channel in ML-DSA decomposition* (2026). URL: https://www.cve.org/CVERecord?id=CVE-2026-22705. Cited in §B.

[108] RustCrypto, *ML-DSA signature verification accepts signatures with repeated hint indices* (2026). URL: https://www.cve.org/CVERecord?id=CVE-2026-24850. Cited in §B.

[109] Keegan Ryan, *Return of the hidden number problem: a widespread and novel key extraction attack on ECDSA and DSA*, IACR Transactions on Cryptographic Hardware and Embedded Systems **2019.1** (2019), 146–168. DOI: 10.13154/tches.v2019.i1.146-168. Cited in §8.3.1.

[110] Kazue Sako and Palash Sarkar (editors), *Advances in cryptology—ASIACRYPT 2013—19th international conference on the theory and application of cryptology and information security, Bengaluru, India, December 1–5, 2013, proceedings, part II*, Lecture Notes in Computer Science, 8270, Springer, 2013. ISBN 978-3-642-42044-3. DOI: 10.1007/978-3-642-42045-0. See [35].

[111] Birger Schacht and Peter Kieseberg, *An analysis of 5 million OpenPGP keys*, Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications **11** (2020), 107–140. URL: https://web.archive.org/web/20201230023235id_/http://isyou.info/jowua/papers/jowua-v11n3-6.pdf. Cited in §7.7.2.

[112] Sophie Schmieg, *ML-KEM mythbusting* (2025). URL: https://web.archive.org/web/20251209094138/https://keymaterial.net/2025/11/27/ml-kem-mythbusting/. Cited in §10.3.

[113] Sophie Schmieg, *Re: Composite ML-DSA* (2026). URL: https://archive.cr.yp.to/2026-04-18/01:02:48/W8nSJPe7jXlIn4GvxoMUH21oxZacQoiuITYjaHF2E2k/https/mailarchive.ietf.org/arch/msg/tls/dgtr2zMoHxiwasmgL1PvoR_oLuo/. Cited in §10.6.

[114] Peter Schwabe, Douglas Stebila, and Thom Wiggers, *Post-quantum TLS without handshake signatures*, in CCS 2020 [**86**] (2020), 1461–1480. DOI: 10.1145/3372297.3423350. Cited in §7.6.

[115] Peter W. Shor, *Algorithms for quantum computation: discrete logarithms and factoring*, in FOCS 1994 [**73**] (1994), 124–134; see also newer version [**116**]. MR 1489242. Cited in §8.

[116] Peter W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer* (1995); see also older version [**115**]; see also newer version [**117**]. URL: https://arxiv.org/abs/quant-ph/9508027v2.

[117] Peter W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing **26** (1997), 1484–1509; see also older version [**116**]. MR 98i:11108.

[118] Marc Stevens, *CWI cryptanalyst discovers new cryptographic attack variant in Flame spy malware* (2012). URL: https://web.archive.org/web/20120610003117/https://www.cwi.nl/news/2012/cwi-cryptanalist-discovers-new-cryptographic-attack-variant-in-flame-spy-malware. Cited in §1.1.

[119] Jakob Stühn, Jan-Niclas Hilgert, and Martin Lambertz, *The hidden threat: analysis of Linux rootkit techniques and limitations of current detection tools*, Digital Threats: Research and Practice **5** (2024), 28:1–28:24. DOI: 10.1145/3688808. Cited in §10.6.

[120] Simon Tatham, *PuTTY bug eddsa-overlarge-s* (2026). URL: https://www.chiark.greenend.org.uk/~sgtatham/putty/wishlist/eddsa-overlarge-s.html. Cited in §C.

[121] Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (editors), *Proceedings of the 2017 ACM SIGSAC conference on computer and*

communications security, CCS 2017, Dallas, TX, USA, October 30–November 03, 2017, ACM, 2017. ISBN 978-1-4503-4946-8. DOI: 10.1145/3133956. See [100].

[122] United States Supreme Court, *Motor Vehicle Mfrs. Ass'n v. State Farm Mutual Automobile Ins. Co.* (1983), 463 U.S. 29. URL: https://www.courtlistener.com/opinion/110991/motor-vehicle-mfrs-assn-of-united-states-inc-v-state-farm-mut/. Cited in §1.4.

[123] Filippo Valsorda, *Avoid the randomness from the sky* (2023). URL: https://web.archive.org/web/20230718094446/https://words.filippo.io/dispatches/avoid-the-randomness-from-the-sky/. Cited in §3.2.5.

[124] Filippo Valsorda, *Accumulated test vectors* (2024). URL: https://web.archive.org/web/20250708015905/https://words.filippo.io/accumulated/. Cited in §3.2.3.

[125] Filippo Valsorda, *A cryptography engineer's perspective on quantum computing timelines* (2026). URL: https://web.archive.org/web/20260511083730/https://words.filippo.io/crqc-timeline/. Cited in §1.2, §2, §8.1, §10.1, §10.1, §10.1, §10.1, §10.3, §10.5, §10.5, §10.5.

[126] Jeffrey Scott Vitter (editor), *Proceedings of the thirtieth annual ACM symposium on the theory of computing, Dallas, Texas, USA, May 23–26, 1998*, ACM, 1998. ISBN 978-0-89791-962-3. DOI: 10.1145/276698. See [10].

[127] Bas Westerbaan, *Cloudflare targets 2029 for full post-quantum security* (2026). URL: https://blog.cloudflare.com/post-quantum-roadmap/. Cited in §8.1.

[128] Carey Williamson, Aditya Akella, and Nina Taft (editors), *Proceedings of the 2014 Internet measurement conference, IMC 2014, Vancouver, BC, Canada, November 5–7, 2014*, ACM, 2014. ISBN 978-1-4503-3213-2. URL: http://dl.acm.org/citation.cfm?id=2663716. See [67].

[129] wolfSSL Inc., *Fault injection attack with ML-DSA and ML-KEM on ARM* (2026). URL: https://www.cve.org/CVERecord?id=CVE-2026-3503. Cited in §B.

[130] wolfSSL Inc., *wolfSSL ECDSA certificate verification* (2026). URL: https://www.cve.org/CVERecord?id=CVE-2026-5194. Cited in §B.

[131] Paul Wouters, *[TLS] forwarding draft-ietf-tls-mlkem-05 use case* (2026). URL: https://web.archive.org/web/20260311152511/https://archive.cr.yp.to/2026-02-21/18:04:50/g3QdEISLDFLsAFawzKSvm0CazLoSXkdd8Dy6urq0qvY/https/mailarchive.ietf.org/arch/msg/tls/YZT5IzoumhTt3C531QR2W0ZNvBU/. Cited in §1.4.

A Attack goals

This views some of the attack goals considered in the signature-system literature.

A.1 Universal forgery

The attack goal highlighted in Section 1 is what the literature typically calls "universal forgery", meaning that the attacker can forge signatures on new attacker-chosen messages, where a new message means a message that the legitimate signer never signed. Severe signature vulnerabilities in this paper mean low-cost universal forgeries.

An attack recovering the original secret key implies a universal-forgery attack, but the concept of universal forgery doesn't require key recovery. For example, this paper's attacks are universal forgeries, even though the attacks don't recover the ML-DSA secret key—they recover only part of it. The claim from [**87**] to recover "the secret key" is unsubstantiated but is also not relevant here.

A.2 Existential forgery

Signature systems are normally designed to avoid not just "universal forgery" but also "existential forgery". Cryptographic jargon: these systems are designed to provide "EUF-CMA".

If the attacker manages to come up with a single signed message that passes verification, where the legitimate signer never signed that message, then that's an existential forgery. The concept of existential forgery doesn't require the attacker to be able to choose the message to be signed; even a tiny modification to a message counts as an existential forgery if the legitimate signer never signed the modified message.

Focusing on universal forgeries could be excluding some important existential-forgery attacks. I don't mean to suggest that it is acceptable for a signature system to allow existential forgeries. However, I checked a sample of the existential-forgery attacks surveyed in [**24**], and all of them allowed universal forgeries. All of the specific severe vulnerabilities considered in this paper allow universal forgeries. So, for purposes of this paper, considering existential forgeries seems unnecessary.

A.3 Forgeries for one verifier

The definition of universal forgeries is relative to a specified verification algorithm. There isn't a standard definition of universal forgeries (or of existential forgeries) when there's a variety of verifiers. For this paper, any verification bug that allows universal forgeries is counted, although this could be quantitatively overestimating impact in cases where other verifiers checking signatures under the same keys don't have the same bug.

A.4 Signature malleability

Sometimes signature systems are designed to avoid "signature malleability". Cryptographic jargon: these systems are designed to provide "SUF-CMA".

Violating SUF-CMA means forging a new signature-message pair, where new means that the legitimate signer never produced that signature-message pair. The cases where this is different from EUF-CMA are cases where the attacker is replacing one signature with another signature on the same message.

As noted in Section 1.2, signature malleability doesn't matter for applications that simply want to ensure that every message passing verification is a message that was signed by the legitimate signer. A cryptosystem such as ECDSA

allowing signature malleability, or a bug allowing signature malleability in a signature system that's supposed to stop signature malleability, isn't treated in this paper as a severe vulnerability.

For applications where signature malleability *is* a problem, ML-DSA claims to be suitable, but one of the vulnerabilities in [**81**] would then have to be treated as a severe vulnerability. Also, if the same applications deploy ECC+ML-DSA by concatenating an ECC signature and an ML-DSA signature on the same message, then any ECC signature malleability (as an extreme case, a quantum break of the ECC key) allows ECC+ML-DSA signature malleability.

On the other hand, if the same applications instead sign with ECC and then *sign that signed message* with ML-DSA, then malleability of the ECC signature would require ML-DSA forgery; i.e., it's easy to ensure that ECC+PQ achieves SUF-CMA whenever the PQ part achieves SUF-CMA. This two-step ECC+PQ mechanism was already recommended in, e.g., [**20**, Section 3.2] in 2018. See also [**76**] and [**64**] for ECC+PQ constructions that achieve SUF-CMA whenever the ECC part *or* the PQ part achieves SUF-CMA.

B CVEs mentioning Dilithium or ML-DSA

I'm not counting CVE-2026-22705 [**107**] ("a timing side-channel was discovered in the Decompose algorithm which is used during ML-DSA signing") or CVE-2026-24850 [**108**] ("the ML-DSA signature verification implementation in the RustCrypto 'ml-dsa' crate incorrectly accepts signatures with repeated (duplicate) hint indices") as severe vulnerability announcements for Dilithium. These are CVEs for issues in Dilithium software, but I don't see any analysis of exploitability.

CVE-2026-41990 [**92**] ("Libgcrypt before 1.12.2 mishandles Dilithium signing. Writes to a static array lack a bounds check but do not use attacker-controlled data") is another CVE for a bug in Dilithium software, but appears to say that it isn't exploitable.

CVE-2023-24025 [**90**] ("CRYSTALS-DILITHIUM (in Post-Quantum Cryptography Selected Algorithms 2022) in PQClean d03da30 may allow universal forgeries of digital signatures via a template side-channel attack because of intermediate data leakage of one vector") refers to an attack demo from [**47**], but the cost of the demo (a day of direct physical inspection of a target device) is too high for this paper to count it as a severe vulnerability.

For a different reason, I'm not counting CVE-2025-15469 [**103**] as a Dilithium software vulnerability. It's a severe vulnerability, and the vulnerability report mentions various algorithms including ML-DSA, but the bug is in the higher-level code for the `openssl dgst` command-line tool rather than in the Dilithium software per se. The bug had to be fixed in that tool. See also [**32**] regarding the root cause of this bug.

Similarly, CVE-2026-5194 [**130**] should not be blamed on Dilithium.

Finally, I'm not counting fault-injection attacks such as CVE-2024-31510 [**91**] or CVE-2026-3503 [**129**].

C CVEs mentioning Ed25519 or EdDSA

Section 8.3.1 looks at CVE-2017-9526, CVE-2021-20305, and CVE-2026-3562. Here are the other CVEs that I found mentioning Ed25519 or EdDSA:

- Signature malleability (see Appendix A.4): CVE-2020-36843. CVE-2023-44273. CVE-2024-42459. CVE-2024-45193. CVE-2024-48949. CVE-2025-57801. CVE-2026-3706. CVE-2026-4115 (see [120] for details). CVE-2026-4541. CVE-2026-33895.
- Bugs in software for other cryptosystems (such as ECDSA or "threshold Ed25519"): CVE-2022-47930. CVE-2022-47931. CVE-2023-26556. CVE-2023-26557. CVE-2024-23342. CVE-2025-69277. CVE-2026-5194.
- Physical side channels or fault injection: CVE-2024-2881. CVE-2025-3301.
- Actual or potential denial-of-service attacks: CVE-2022-38178. CVE-2024-30172. CVE-2026-33936. CVE-2026-40092. CVE-2026-46598.
- Vulnerabilities in the application code, not in code for the signature system per se; see list below.

The vulnerabilities in application code were as follows:

- CVE-2015-9258 (application sometimes called the wrong signature system).
- CVE-2019-15545 (application used a verification wrapper that always returned success).
- CVE-2022-50237 (maybe some applications pass wrong inputs to signing).
- CVE-2024-1631 (application did not provide randomness to keygen).
- CVE-2024-32482 (application overread the data to sign, exposing subsequent data in memory).
- CVE-2024-40640 (application used variable-time code to load private keys).
- CVE-2025-15469 (application truncated messages passed to signing and verification, while thinking it had verified complete messages; also mentioned in Appendix B).
- CVE-2025-34198 (application used "hardcoded SSH host private keys in the appliance image" so "The same private host keys (RSA, ECDSA, and ED25519) are present across installations, rather than being uniquely generated per appliance").
- CVE-2025-62705 (application "did not appropriately redact fields . . . would emit public keys to the audit log").
- CVE-2026-26319 (application did not verify signatures "when telnyx.publicKey is not configuredwhen telnyx.publicKey is not configured").
- CVE-2026-39831 (application "did not check the User Presence flag" so passed wrong data to the signature system).
- CVE-2026-41564 (application did not "reseed the Crypt::PK PRNG state after forking").

Note that application vulnerabilities are important to the end user and of interest for engineers designing safer interfaces for cryptographic libraries, as [32] illustrates. However, these vulnerabilities are orthogonal to the evaluation of the relative risks of PQ and ECC+PQ.