The Spinning Coin and the Stolen Key: What Shor's Algorithm Actually Threatens

Dr Neal Aggarwal · 22 September 2026 · 38 min read

Every conference I go to lately, somebody wants to talk to me about whether AI is about to take over the world. Nobody has ever once asked me whether the same era of computing might quietly read every patient record I have spent this year designing MedLattice to protect. It should be the other way around. AI takeover is speculative — disputed timelines, disputed probabilities, no working example. Shor's algorithm is not speculative. It has existed, proven, since 1994; the only open question is how many years of engineering stand between now and a machine large enough to run it. This post is the explanation I wish existed before I had to go and build cryptography around the problem myself: what encryption actually is and how a record like MedLattice's gets locked in the first place, why a classical computer cannot pick that lock no matter how large it gets, what a quantum computer is actually doing differently — with superposition explained properly, not just named and left — Euclid's algorithm worked by hand because it turns out to be the quiet backbone of the whole attack, Shor's algorithm itself walked through on numbers small enough to check on paper, and exactly how a quantum computer would derive a private key from a public one. It closes with why AES-256 is not the part that's actually in danger, and why the lattice mathematics behind ML-KEM resists this attack in a structurally different way than RSA or elliptic curves ever could.

Contents

> *I write here in a personal capacity. This post is a companion to Not Shown Is Not Locked, which describes MedLattice's cryptography in full and asserts, in one paragraph, that it is built to survive quantum computers. This post is the whole argument behind that paragraph, aimed at a reader with no physics or number theory background — because I don't think anyone should have to take my word for a claim like that.*

Download this essay as a PDF

Twenty-two pages walking through quantum superposition, Euclid's algorithm, and Shor's algorithm by hand — the full argument behind the one paragraph in MedLattice's specification that claims to survive a quantum computer.

📄 **Aggarwal N. *The Spinning Coin and the Stolen Key: What Shor's Algorithm Actually Threatens.* 2026.**

⬇ Download PDF (381 KB, 22 pages, 8 figures)

Every conference I go to lately, somebody corners me about whether AI is about to take over the world. It's usually well-meant, and it's usually the same conversation: an agent got a little too autonomous in a demo, somebody read a paper about deceptive alignment, and now the question arrives already dressed as an existential one.

Nobody has ever once asked me whether the same decade of computing might quietly read every patient record I have spent this year designing MedLattice to protect. I think that's backwards, and I want to explain carefully why, because "quantum computers will break encryption" is a sentence people nod at and then file next to "AI will take over the world" — another abstract, someday, science-fiction risk. It isn't the same kind of claim at all. One is a disputed prediction about the behaviour of a system nobody has built yet. The other is a solved mathematics problem, proven correct in 1994, waiting only on enough physical qubits to run it. The internet, the banking system, and every blockchain including the one under MedLattice's own consortium chain are all standing on cryptography that this already-proven algorithm defeats outright, given a big enough machine.

That's the claim. The rest of this post is me actually justifying it, from the ground up, in the way I wish somebody had done for me before I had to go and design cryptography around the problem myself for a patient record.

Why I think this is the risk that should be keeping people up

I want to be honest about the shape of this argument before I make it, because I think overclaiming it would undermine the part that's actually solid.

I am not saying AI safety doesn't matter, and I'm not going to pretend serious people don't disagree with me on the relative ranking. Some very well-informed people rank the possibility of a misaligned advanced AI system above almost everything else on the list of things worth worrying about, and I don't think that position is foolish. What I'm saying is narrower: as things stand today, the quantum threat to cryptography gets a fraction of the attention, is much closer to certain, and — this is the part I actually care about — already has a concrete, buildable, deployable answer. AI alignment is a research problem nobody has solved. Post-quantum cryptography is an engineering problem that is, right now, solved, standardised, and sitting unused in almost every system that should already have migrated to it.

Here is what makes the quantum case different in kind, not just degree. Shor's algorithm is not a forecast. Peter Shor proved, mathematically, in 1994, that a sufficiently large quantum computer factors integers and solves the discrete logarithm problem in polynomial time — meaning the time it takes grows manageably as the numbers get bigger, rather than exploding the way it does for every classical method we know of. That proof hasn't been disputed since; there's no expert disagreement about whether the algorithm works, only about how many years of engineering separate today's noisy, error-prone quantum processors from a machine with enough clean, error-corrected qubits to run it on numbers the size actually used in RSA or elliptic-curve cryptography. Estimates for that engineering timeline genuinely vary — I've seen serious people argue for something like a decade, and others argue for several decades, and I'll come back to that honestly later in this post. But the mathematics at the end of that timeline is not in question. It is one of the few places in this entire discussion where the endpoint is certain and only the calendar is uncertain.

> **An honest caveat before I go further** I'm not neutral here — MedLattice is built around the assumption that this threat is real and near enough to matter, so I have every incentive to argue it forcefully. I've tried to keep every specific technical claim in this post checkable against the maths rather than against my own conviction, and I've flagged the places where the maths runs out and only an estimate is left. Where I'm giving you my opinion rather than a proof, I've said so.

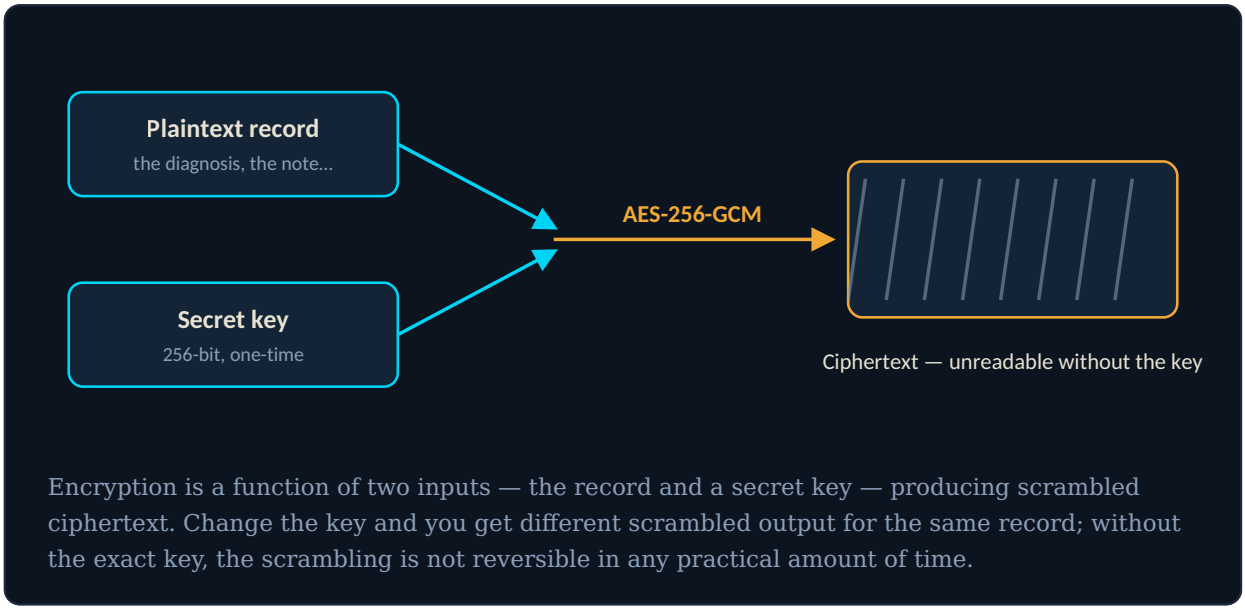
The specific reason patient records make this urgent rather than merely interesting is a pattern security people call **harvest now, decrypt later**. An adversary doesn't need a working quantum computer today to start benefiting from one. They need to copy your encrypted traffic today — which is cheap, passive, and already routinely

done at national scale — and simply keep it in storage until the machine that can open it exists. For a credit card number, that's not much of a threat, because the card will have expired long before anyone can read it. For a patient's HIV status, genetic sequence, or psychiatric history, there is no expiry date. Kenya's Digital Health Act alone requires twenty years of retention as a floor, not a ceiling, and a record opened for a child today has to keep its secrecy into the 2090s regardless of when the machine that could break it actually arrives. If harvesting is happening now — and there is no serious reason to think it isn't — then the deadline that matters for that data has already passed. Migrating to quantum-resistant cryptography today isn't precautionary. For health data specifically, it's closer to overdue.

What encryption actually is, and how a record like a patient's actually gets locked

Before any of this can make sense, I need to be precise about what "encryption" means, because the word gets used loosely and the looseness is exactly where confusion about the quantum threat creeps in.

Encryption is a mathematical function that takes your data and a secret number — the **key** — and produces scrambled output, called **ciphertext**, that is computationally infeasible to turn back into the original data without that same key, or one mathematically linked to it. That's the whole idea. Nothing about "quantum-safe" or "quantum-vulnerable" changes that basic structure; what changes is which specific mathematical trick is used to manage and protect the key, and that trick is exactly what a quantum computer can attack.

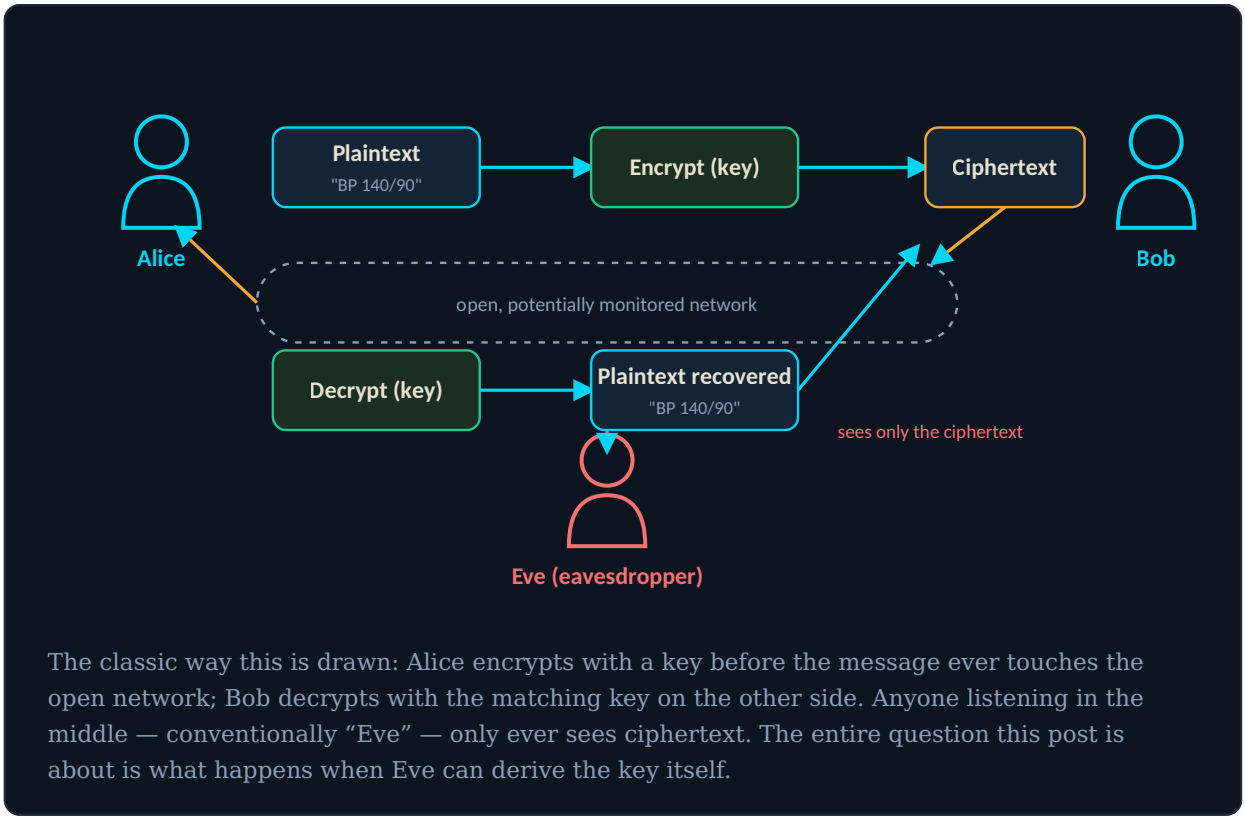


Encryption is a function of two inputs — the record and a secret key — producing scrambled ciphertext. Change the key and you get different scrambled output for the same record; without the exact key, the scrambling is not reversible in any practical amount of time.

There are two fundamentally different families of encryption, and a real system like MedLattice — like almost every secure system on the internet — uses both together, for different jobs.

Symmetric encryption uses one secret key that both locks and unlocks the data. MedLattice uses AES-256-GCM for this: the actual bytes of your medical record — the diagnosis, the note, the lab result — are encrypted with a 256-bit secret key, and whoever holds that exact key can decrypt it, and nobody else can. It's fast, it's mathematically simple to reason about, and — as I'll get to properly later — it is not the part of this story that a quantum computer meaningfully threatens.

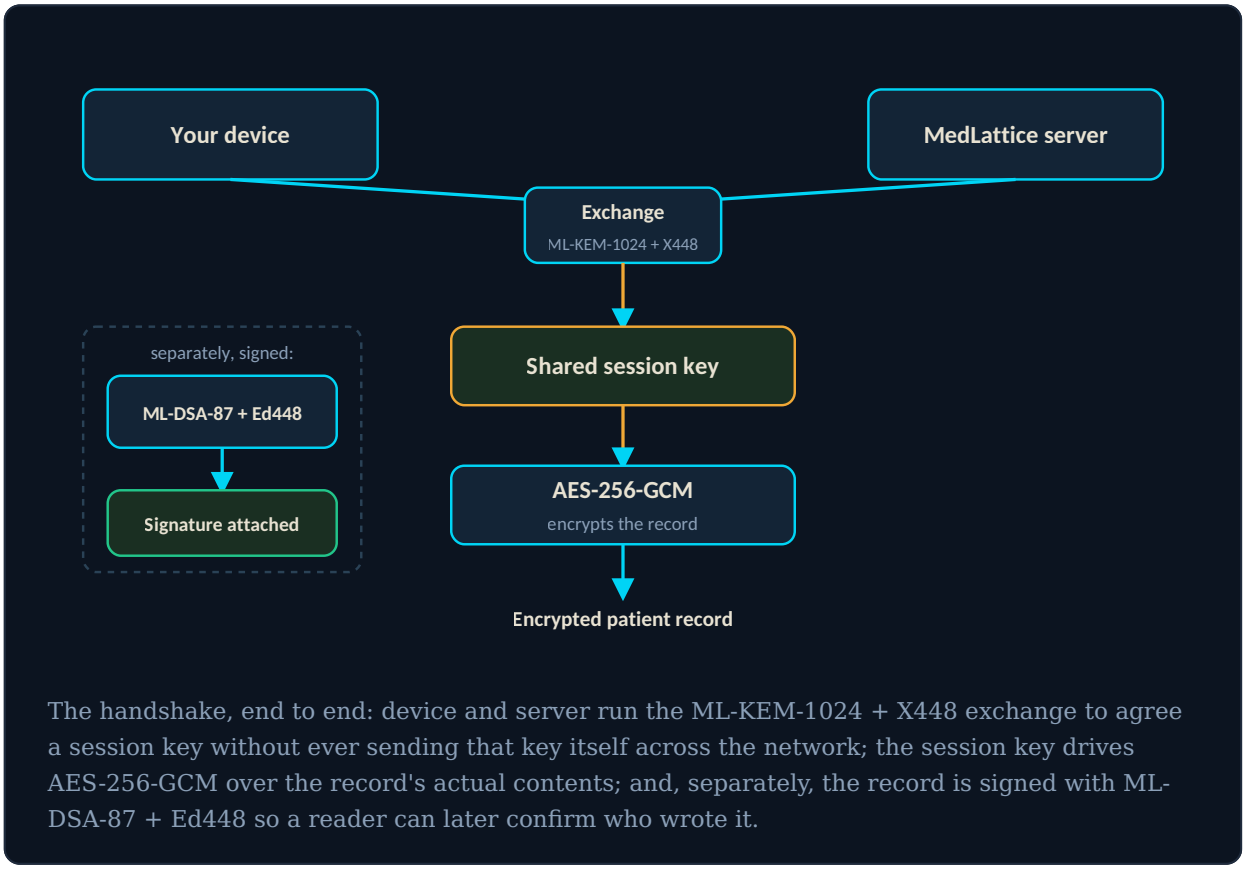
Asymmetric encryption, also called public-key cryptography, is the clever bit, and it's what makes the whole thing work without everyone having met in person to exchange secrets first. You generate a mathematically linked *pair* of numbers: a **public key**, which you can hand to anyone, publish on a certificate, or write into a blockchain transaction with no loss of security, and a **private key**, which you keep forever and never disclose. The entire scheme rests on one asymmetry, and I want to state it as plainly as I can because everything later in this post is really just an explanation of how that asymmetry gets destroyed: it is trivially easy to compute the public key from the private key, and — for a classical computer — it is almost unimaginably hard to compute the private key from the public key, even though the two are locked together by a precise mathematical relationship that anyone can see in full.



The classic way this is drawn: Alice encrypts with a key before the message ever touches the open network; Bob decrypts with the matching key on the other side. Anyone listening in the middle — conventionally “Eve” — only ever sees ciphertext. The entire question this post is about is what happens when Eve can derive the key itself.

So here is how your record actually moves, concretely, in a system like MedLattice or in any ordinary hospital TLS connection. Your device and the server first use asymmetric cryptography to agree, over an open and potentially monitored network, on a fresh one-time secret — a **session key**. Historically this step used RSA or Elliptic-Curve Diffie–Hellman; MedLattice uses ML-KEM-1024 combined with the classical curve X448, for reasons I'll get to in why ML-KEM resists this further down.

A quick gloss on those four names, so you're not left guessing: **ML-KEM** (the Module-Lattice-Based Key-Encapsulation Mechanism, standardised by NIST as FIPS 203; background on Wikipedia) is the new, lattice-based algorithm actually doing the quantum-resistant work of agreeing the session key. **X448** (Wikipedia) is an older, extensively-studied classical elliptic curve, run alongside ML-KEM purely as a safety net. **ML-DSA** (FIPS 204) and **Ed448** (background on EdDSA) are the equivalent lattice-plus-classical pairing, doing the same job for signatures instead of key exchange.



The handshake, end to end: device and server run the ML-KEM-1024 + X448 exchange to agree a session key without ever sending that key itself across the network; the session key drives AES-256-GCM over the record's actual contents; and, separately, the record is signed with ML-DSA-87 + Ed448 so a reader can later confirm who wrote it.

That session key is then used, symmetrically, with AES-256-GCM to actually encrypt your record's contents. Separately, a digital signature — again built from asymmetric cryptography, in MedLattice's case ML-DSA-87 combined with the classical curve Ed448 — is attached so that anyone verifying the record later can confirm exactly which clinician or system produced it, and that it hasn't been altered since. Three

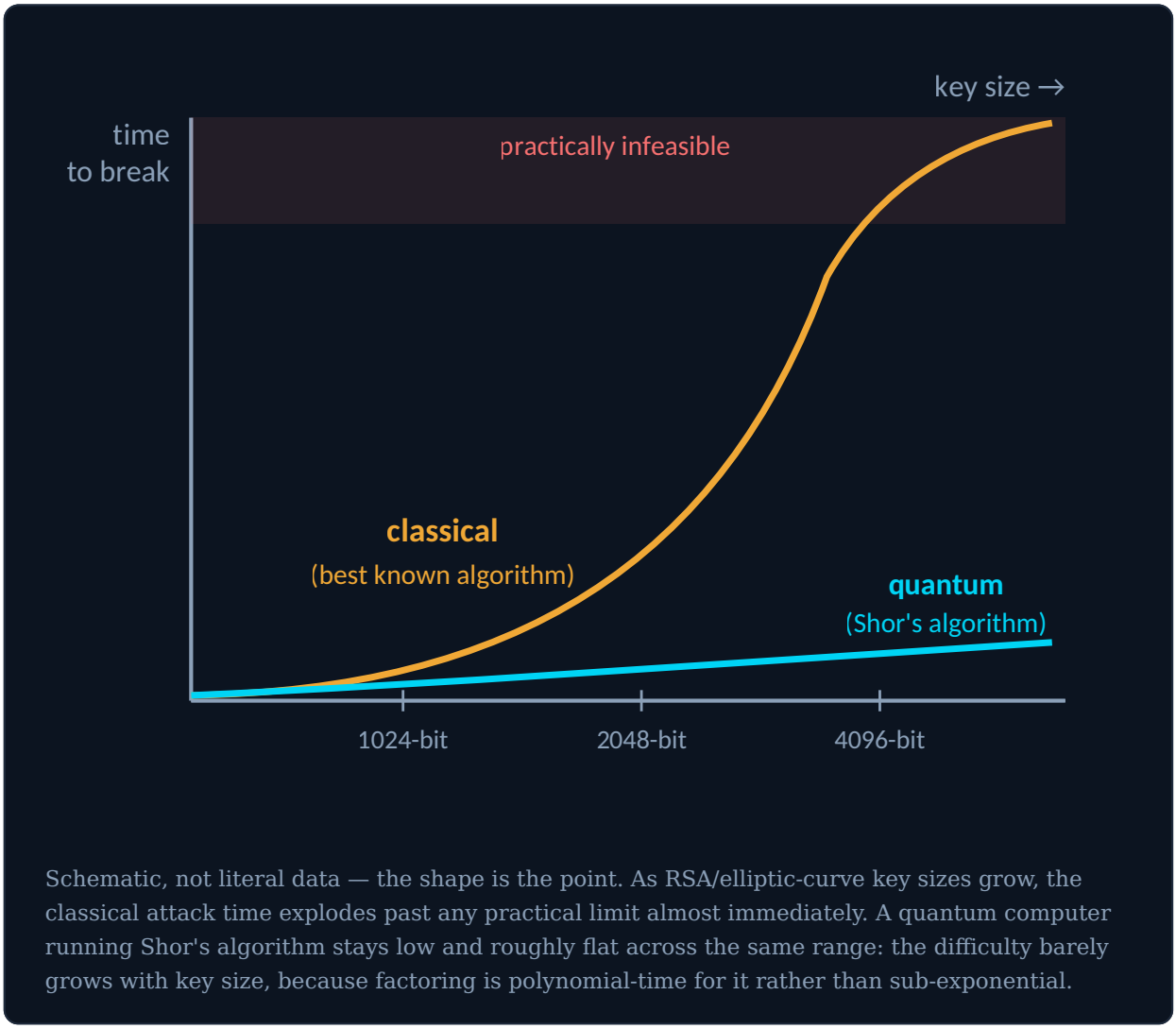
cryptographic jobs, two of them asymmetric, one symmetric, working together every single time a record is written or read.

> **The one distinction to hold onto for the rest of this post** Symmetric encryption (AES-256) protects the record itself and has no public/private key split — there is nothing to "derive." Asymmetric encryption (RSA, elliptic curves, or ML-KEM) protects the handshake that establishes the symmetric key, and the signature that proves who wrote the record — and it is built entirely on the assumption that deriving a private key from a public one is computationally out of reach. Shor's algorithm is an attack on that second assumption. It does not touch the first.

Why a classical computer cannot do this, no matter how big you build it

The specific mathematical trapdoors in use today come in two families, and MedLattice's own default blockchain layer touches both of them, which is a useful thing to notice.

RSA relies on **integer factorisation**: multiplying two enormous prime numbers together is fast, but given only the product — a number with hundreds of digits — recovering the two original primes is, for every classical algorithm known, ferociously slow. The best classical method, the general number field sieve, still takes time that grows *sub-exponentially* in the size of the number, which sounds almost reasonable until you see what that means in practice: widely-cited illustrative estimates put cracking a 2,048-bit RSA key, using every classical computer humanity could plausibly build, somewhere past 300 trillion years. That figure isn't meant to be exact — nobody actually expects to run the experiment — it's meant to communicate "not in any sense relevant to a human lifetime, or to civilisation."



Schematic, not literal data — the shape is the point. As RSA/elliptic-curve key sizes grow, the classical attack time explodes past any practical limit almost immediately. A quantum computer running Shor's algorithm stays low and roughly flat across the same range: the difficulty barely grows with key size, because factoring is polynomial-time for it rather than sub-exponential.

Elliptic-curve cryptography relies on a different hard problem: the **discrete logarithm** problem over an elliptic curve. secp256k1 — the specific curve Bitcoin uses, and, not coincidentally, the default curve Ethereum and every ordinary EVM (ethereum virtual machine) chain use for their transaction signatures — is exactly this kind of scheme. Given a starting point on the curve and the result of "adding" that point to itself some secret number of times, working out how many times classically requires, again, a number of steps that grows so fast with key size that 256-bit curves are considered entirely safe against any classical computer that could ever plausibly be built. This is worth sitting with for a moment: secp256k1 (there's no dedicated Wikipedia article for the curve itself, but the Bitcoin Wiki entry is a solid technical reference) is literally the curve a permissioned EVM chain like MedLattice's would use by default for every transaction signature, which is exactly why the MedLattice specification deliberately swaps it out.

I want to underline what these two hard problems have in common, because it's the detail that makes everything downstream of it make sense. Both factorisation and the discrete logarithm are, underneath the surface, the same *kind* of problem: they both

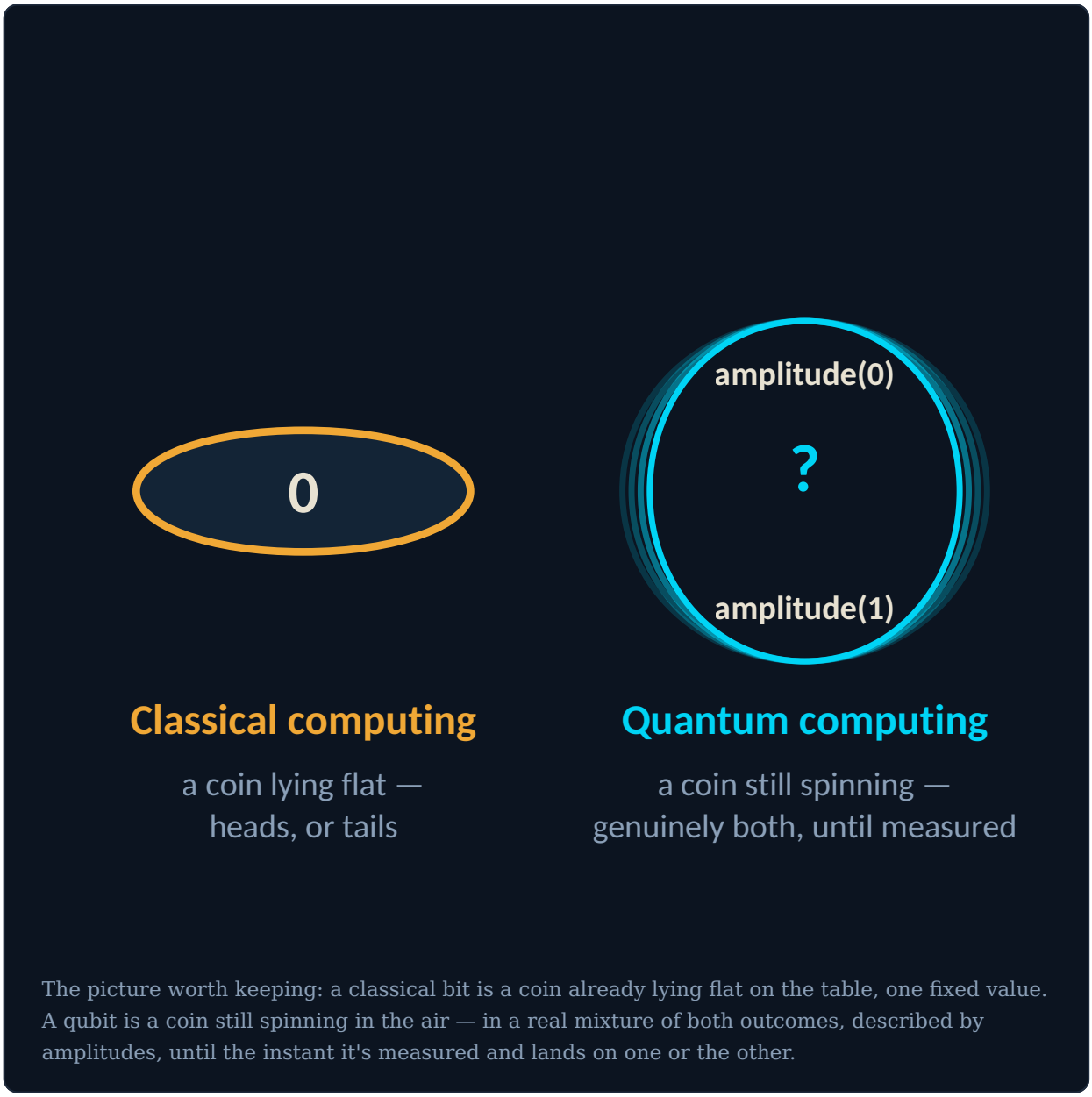
reduce to finding a hidden **period** — a repeating pattern — inside a mathematical structure built from modular arithmetic. Classical computers are catastrophically bad at finding that period once the numbers get large, because the only way they know to look is to try candidate answers more or less one at a time, or in clever but still fundamentally sequential ways, and the search space grows exponentially. That shared periodic structure is not a coincidence, and it's the exact thing a quantum computer turns out to be extraordinarily good at exploiting. It's also why a single algorithm — Shor's — breaks both RSA and elliptic curves, including secp256k1, using the same underlying trick aimed at two different but structurally similar targets.

What a quantum computer is actually doing differently — superposition, properly explained

This is the part that gets skipped. Every explainer I watched while putting this post together — including the ones I'm drawing on here — says the words "quantum superposition," treats the concept as self-evident, and moves straight on to the algorithm. It isn't self-evident, and skipping it is exactly how you end up with the popular but wrong idea that a quantum computer "just tries every possible answer at once and reads off the right one," which is not what happens, and which will actively mislead you about why Shor's algorithm needs the extra machinery it has. I want to build this properly, because the rest of the post depends on it.

Start with a classical bit. Think of it as a coin lying flat on a table: heads, or tails. One fixed, definite value, and it stays that value until you deliberately change it. Every operation a classical computer performs — every AND, every OR, every addition — takes coins in some arrangement of heads and tails and produces a new arrangement of heads and tails. At every single instant, the whole machine's state is one specific, fully determined pattern.

A **qubit** is not a coin lying flat. It's closer to a coin *while it is spinning* in the air. While it's spinning, it isn't meaningfully "heads" or "tails" — it's in a genuine mixture of both possibilities, described by two numbers called **amplitudes**, one attached to the heads-outcome and one to the tails-outcome. This mixture is what "superposition" means: not "we don't yet know which it is," in the way a tossed coin is definitely already heads-or-tails and merely hidden under your hand, but a physically real state that is neither one nor the other until it's measured. The moment you *do* measure it — the moment the coin lands — the superposition collapses, instantly and irreversibly, into one definite classical outcome, heads or tails, with a probability set by those amplitudes. Squared, technically, but the detail that matters here is just: bigger amplitude, more likely outcome.



The picture worth keeping: a classical bit is a coin already lying flat on the table, one fixed value. A qubit is a coin still spinning in the air — in a real mixture of both outcomes, described by amplitudes, until the instant it's measured and lands on one or the other.

Here's the part that turns this from a curiosity into a computational resource. With **one** spinning coin, you have a mixture of two possibilities. With **two**, you have a mixture of four (heads-heads, heads-tails, tails-heads, tails-tails) held simultaneously, all at once, in one physical system. With n qubits prepared this way, you are holding a superposition over 2^n possibilities simultaneously — with 300 qubits, that's a superposition over more numbers than there are atoms in the observable universe, all represented at once in one coherent quantum state. This is genuinely what people mean by "quantum parallelism," and it's real.

Superposition compounds. One qubit holds a mixture of 2 possibilities; two qubits, held together in one physical system, hold a mixture of 4; n qubits hold a mixture of 2^n — every one of them present simultaneously, though (as the next section covers) not all simultaneously readable.

But — and this is the part almost every popular explanation glosses over — you cannot simply *read out* all 2^n answers. The instant you measure, the superposition collapses to exactly **one** of those possibilities, chosen randomly according to the amplitudes, and every other possibility that was "computed" alongside it vanishes, unobserved, forever. If a quantum algorithm were nothing more than "prepare a superposition over all possible answers, then measure," it would be useless — a very expensive way of picking one random guess.

The actual trick — the whole reason quantum algorithms are hard to design and only a handful of genuinely useful ones exist — is **interference**. Think of the amplitudes the way you'd think of ripples spreading across the surface of water. Where two ripples' peaks arrive at the same point together, they add up: a taller wave, a higher chance of that outcome being the one you measure. Where a peak from one ripple arrives at the same point as a trough from another, they cancel out: the wave flattens there, and that outcome becomes correspondingly unlikely, potentially all the way down to zero. A quantum algorithm is, at its mathematical core, a carefully engineered recipe of operations that arranges the amplitudes across your superposition so that the *wrong* answers interfere destructively — their ripples cancel — and the *right* answer, or answers close to it, interfere constructively — their ripples reinforce. Do this well, and by the time you finally measure, the probability has been concentrated onto the answer you actually want, even though you never inspected any of the 2^n possibilities individually along the way.

That is what a quantum computer brings to the table that a classical one categorically cannot: not brute-force parallelism in the sense of trying every answer and keeping the best one, but the ability to hold an exponentially large superposition and then sculpt it, through interference, so that measurement is far more likely to land on a structurally meaningful answer than chance alone would predict. It also

explains something important about the limits of quantum computing that I think gets lost in the AI-adjacent hype around it: this trick only works when the problem has enough internal *structure* for interference to exploit. It doesn't give you an exponential speed-up on arbitrary hard problems — most NP-hard problems get no meaningful quantum advantage at all. It works spectacularly well on problems with hidden periodic structure, which is precisely, and not coincidentally, the shape shared by integer factorisation and the discrete logarithm.

Two terms worth pinning down there. **NP-hard** is computer science's label for a class of problems believed to have no efficient — polynomial-time — solution at all, by any method, classical or quantum; a lot of famously hard problems fall in this class and get no help whatsoever from a quantum computer simply existing. **Periodic structure** means a function that repeats itself at some fixed, hidden interval as its input grows — exactly the property factoring and the discrete logarithm turn out to have buried inside them (via the modular exponentiation function this post gets to shortly), and exactly the property a quantum Fourier transform is built to detect. That combination — hard for anything classical, but with a hidden period a quantum computer can exploit — is a narrow target, which is precisely why only a short list of problems get this kind of speed-up.

> **The analogy I keep coming back to** Superposition alone is a coin spinning in the air, genuinely undecided. A useful quantum algorithm is a room full of those spinning coins wired together so that, at the moment they all land, the physics itself has been arranged to make them overwhelmingly likely to land in a pattern that spells out the answer to your question — even though no single coin, and no observer, ever looked at any of them while they were still spinning.

The quiet backbone: Euclid's algorithm

Before I can walk through Shor's algorithm properly, I need one piece of purely classical mathematics that is nearly 2,300 years old and does more of the actual work than most explanations let on. Euclid described it in Book VII of the *Elements*, around 300 BCE, as a method for finding the **greatest common divisor** of two whole numbers — the largest whole number that divides into both of them with nothing left over. Take 12 and 18 as a small warm-up: both divide evenly by 6 (12 ÷ 6 = 2, 18 ÷ 6 = 3, no remainder either time), and no bigger number does that trick for both, so 6 is their greatest common divisor.

The method is: divide the larger number by the smaller, keep the remainder, then repeat the whole process using the smaller number and that remainder, over and over, until the remainder is zero. The last non-zero remainder you produced is the greatest common divisor.

Let me actually do it, on paper, the way the algorithm is meant to be followed. Take 1071 and 462.

1071 divided by 462 goes 2 times, with 147 left over — because $2 \times 462 = 924$, and $1071 - 924 = 147$.

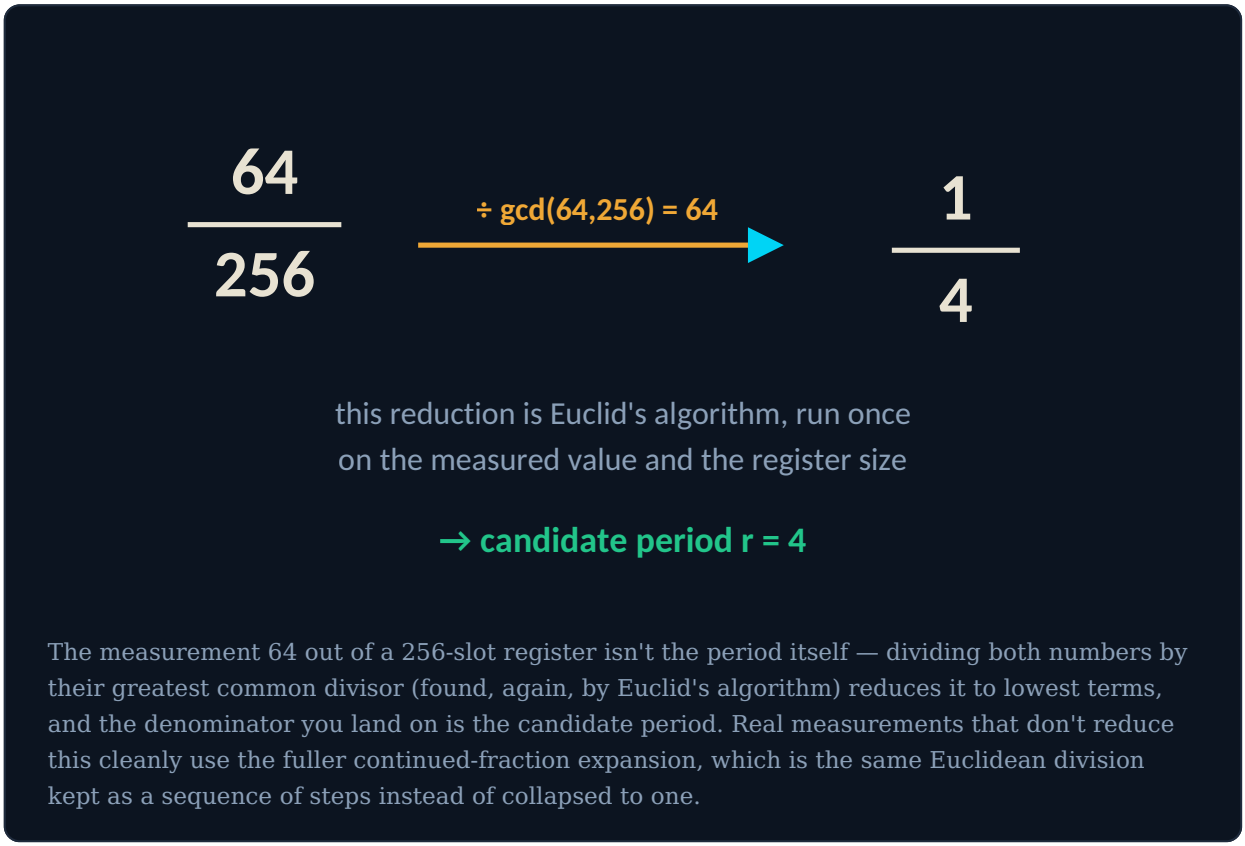
Now repeat, using 462 and 147. 462 divided by 147 goes 3 times, with 21 left over — $3 \times 147 = 441$, and $462 - 441 = 21$.

Repeat again, using 147 and 21. 147 divided by 21 goes exactly 7 times, with nothing left over — $7 \times 21 = 147$ exactly.

The remainder just hit zero, so we stop, and the answer is the last non-zero remainder: **21**. You can check this yourself — $1071 = 21 \times 51$, and $462 = 21 \times 22$, and 51 and 22 share no common factor, so 21 really is the largest number that divides both.

Each step divides the previous pair, carries the remainder down, and repeats. The moment a remainder hits zero, the last non-zero remainder — here, 21 — is the answer.

Two things about this method matter enormously for what comes next. First, it is startlingly fast — the number of steps it takes grows only in proportion to the number of *digits* in the input, not the size of the numbers themselves, so even for numbers with hundreds of digits, the kind actually used in RSA keys, Euclid's algorithm finishes in a fraction of a second on an ordinary laptop. It has nothing in common, difficulty-wise, with factoring or the discrete logarithm — it is one of the *easy* problems in this whole story. Second — and this is the part I want you to hold onto — Euclid's algorithm is exactly the same procedure used to compute a **continued fraction**, which is a way of expressing any fraction as a nested sequence of whole-number divisions. Run Euclid's algorithm on the numerator and denominator of a fraction and keep the whole-number *quotients* at each step, instead of throwing them away and keeping only the final remainder, and you have just computed its continued fraction expansion. Same arithmetic, two different uses. Remember this, because it's about to reappear twice inside Shor's algorithm, doing two completely different jobs.



The measurement 64 out of a 256-slot register isn't the period itself — dividing both numbers by their greatest common divisor (found, again, by Euclid's algorithm) reduces it to lowest terms, and the denominator you land on is the candidate period. Real measurements that don't reduce this cleanly use the fuller continued-fraction expansion, which is the same Euclidean division kept as a sequence of steps instead of collapsed to one.

Shor's algorithm, worked by hand

The textbook way to introduce Shor's algorithm is with a number small enough to check by hand. That's a genuine teaching tool here, not just a simplification for its own sake: the algorithm's steps don't change shape as the numbers grow, only their size does. A quantum computer factoring a 600-digit RSA modulus runs exactly the same sequence of steps as one factoring 15 — it just needs a much bigger register to

hold a much bigger superposition, the same way a calculator adds 7+8 and a 600-digit sum using the same procedure, just with more digits to carry. So let's factor **N = 15**, using the base **a = 7**, and check every single step by hand as we go. (You already know the answer is 3 × 5 — that's deliberate. It means you can verify each step actually did its job, rather than having to take it on trust.)

Step one, entirely classical. Before touching a quantum computer at all, check whether a and N already share a common factor, using — yes — Euclid's algorithm: gcd(7, 15). 15 divided by 7 goes 2 times with 1 left over; 7 divided by 1 goes 7 times with 0 left over. gcd = 1. No shared factor, so we can't get lucky and skip the rest; we genuinely need to find the **order** of 7 modulo 15 — the smallest positive whole number r such that 7 raised to the power r, divided by 15, leaves a remainder of exactly 1.

Classically, the only way to find r is to compute 7^1, 7^2, 7^3, and so on, each time reducing modulo 15, until you hit a remainder of 1 — and for a 15-digit or 600-digit modulus, that search becomes exactly as hard as factoring itself. This is the one step in the entire algorithm that a quantum computer actually accelerates, and everything else here — before and after it — is classical arithmetic a laptop finishes instantly.

Step two, the quantum part. A quantum computer prepares a register of qubits in a superposition over every possible value of x at once — every spinning coin, landing on every value from 0 up to some large power of two simultaneously — and then, using that superposition as an input, computes 7^x mod 15 for *all* of those values of x **at the same time**, in one pass, storing the results in a second register. Because 7^x mod 15 is a periodic function — it repeats every r steps, by definition of what "order" means — the amplitudes across that huge superposition now carry a hidden periodic pattern, still invisible, still inaccessible by direct measurement.

Think of it in three steps, with an audio analogy.

1. You have a hidden repeating pattern. After the step above, the quantum register is in a superposition where the "right" states repeat every r values. You can't see r directly — if you measured right now, you'd get an essentially random number and learn almost nothing about it.

2. Use a "frequency finder" on that pattern. An ordinary Fourier transform takes a sound wave and answers "what frequencies are present in this wave?" A sound with a clear repeating pattern at some frequency produces a sharp spike at that frequency and near-zero everywhere else. The quantum Fourier transform does the same kind of thing, but to the amplitudes of the quantum states rather than to a sound wave: it looks at the repeating structure hidden in the superposition and reshapes it so that states connected to the true period get large amplitude, while every other state's amplitude cancels toward zero through interference. In short, it turns "there is a hidden period r somewhere in this superposition" into "almost all of the probability is now sitting on the handful of outcomes that encode r."

3. Measure, and get real information about r. Before the quantum Fourier transform, measuring gives you an almost uniformly random number — useless. After it, you're very likely to measure a number close to a whole-number multiple of (register size) ÷ *r*. That measured number isn't *r* itself, but it's tightly linked to it, and a short piece of classical math — the continued-fraction reduction, the same Euclidean division from a moment ago — turns it into the actual period.

The one-sentence version: the quantum Fourier transform is a frequency detector. It takes a hidden repeating pattern buried in the quantum state and turns it into a measurement outcome that strongly reveals the period *r*.

This is exactly where interference earns its keep. The machine applies a **quantum Fourier transform** to the register — mathematically the same idea as an ordinary Fourier transform, the tool that takes a sound wave and tells you which pitches are present in it, applied here to the pattern of amplitudes instead of to a sound wave. Just as a Fourier transform turns "a repeating wave" into "a sharp spike at the frequency it repeats at," the quantum Fourier transform arranges the interference across the superposition so that amplitudes belonging to values connected to the true period *r* reinforce each other, and everything else cancels out. Measure the register now, and instead of a uniformly random guess across an astronomical range of possibilities, you get a value that is, with good probability, closely related to *r* — specifically, an integer very close to some whole-number multiple of (register size) ÷ *r*.

For N = 15, the standard textbook choice of register size is 256 (eight qubits' worth, comfortably larger than N^2). Suppose the measurement returns **64**.

Step three, classical again — and this is where Euclid's algorithm reappears. 64 out of 256 isn't the period itself; it's a noisy fingerprint of it, and you recover the period by reducing that fraction to its lowest terms — which is precisely Euclid's algorithm run on 64 and 256. gcd(64, 256): 256 divided by 64 goes exactly 4 times with 0 remainder, so gcd = 64. Divide both numerator and denominator of 64⁄256 by that gcd, and you get 1⁄4. (When a real measurement doesn't reduce this cleanly, you run the fuller continued-fraction expansion I described a moment ago — the same Euclidean division, just kept as a sequence of quotients instead of a single final remainder, converging on the best small-denominator approximation to the fraction you measured.) Either way, the denominator you land on is your candidate period: **r = 4**.

Check it by hand, classically, in a second: $7^1 = 7$. $7^2 = 49$, which mod 15 is 4. $7^3 = 343$, which mod 15 is 13. $7^4 = 2401$, which mod 15 is 1. There it is — the remainder hits 1 exactly at the fourth power, confirming r = 4.

Step four, classical, and Euclid's algorithm a third and final time. Because r = 4 is even (this method needs it to be, and there are standard fallbacks when it isn't), compute 7 to the power r⁄2, which is $7^2 = 49$, or 4 modulo 15. Then take the greatest

common divisor of (4 − 1) and 15, and separately of (4 + 1) and 15. gcd(3, 15): 15 ÷ 3 goes exactly 5 times, remainder 0, so gcd = 3. gcd(5, 15): 15 ÷ 5 goes exactly 3 times, remainder 0, so gcd = 5.

3 and 5. Exactly the factors of 15, recovered without ever trying them directly.

> **What actually happened, stripped of the arithmetic** The quantum computer did exactly one thing a classical computer structurally cannot do at this scale: it found the hidden period of a modular exponentiation function, using superposition to hold every candidate at once and interference — via the quantum Fourier transform — to make the true period the overwhelmingly likely thing you'd measure. Every other step — checking for a lucky shared factor, turning the noisy measurement into an exact period, and turning that period into the actual prime factors — is ordinary classical arithmetic, and every one of those classical steps is, underneath, Euclid's algorithm from 300 BCE. A single, narrow, specialised quantum subroutine, wrapped in un-glamorous, millennia-old classical glue.

Scale this up from N = 15 to an RSA modulus with 600 decimal digits, and nothing about the *method* changes — only the size of the quantum register and the number of physical qubits needed to keep it stable and error-corrected for long enough to complete the quantum Fourier transform, which is the genuinely hard *engineering* problem standing between today's hardware and a working attack, and the reason nobody has factored a real RSA key with a quantum computer yet.

How this actually reaches your encrypted patient record

Here's the question I think matters most, and the one I want to answer without hand-waving: if the patient record itself is encrypted, how does breaking a *different* algorithm — one that finds prime factors — let anyone read it? Is it really, as it sounds, deriving the private key from the public key?

Yes. Precisely that, and I want to walk through exactly how, because "deriving a key" can sound abstract in a way that undersells how total the compromise is.

Recall the structure from earlier: your record's actual contents are protected by a symmetric AES-256 key, and that symmetric key was itself established using an asymmetric handshake — historically RSA key transport or elliptic-curve Diffie–Hellman, secured by a public/private key pair. The public key in an RSA system *is*, almost literally, the product N = p × q of two secret primes, published openly as part of the key. The private key is derived mathematically from p and q individually. Shor's algorithm, run against that public N, factors it — recovers p and q directly, using exactly the procedure worked through above, just on a number with hundreds of

digits instead of 15 — and from p and q, computing the matching private key is fast, ordinary, classical arithmetic, the kind any laptop finishes instantly. There's no side channel, no software bug, no stolen password involved anywhere in this. The private key was never hidden in some inaccessible location; it stood in a precise, publicly inspectable mathematical relationship to a number everyone could already see. Shor's algorithm simply reverses a relationship that used to only run one way.

Elliptic-curve schemes — including secp256k1, sitting under Bitcoin, Ethereum, and any ordinary EVM chain's transaction signatures — work the same way in spirit, using a variant of Shor's algorithm aimed at the discrete logarithm instead of factorisation, but the conclusion is identical: given only the public key, a large enough quantum computer recovers the private key directly.

And once an attacker holds that private key, they are — cryptographically, and for every practical purpose — indistinguishable from the legitimate party. They can decrypt every session key that public key was ever used to protect, and through it, every patient record that session key encrypted. They can produce a digital signature on a forged or altered record that will pass every verification check MedLattice's software — or any hospital's — ever runs on it, because the signature really was produced by the matching private key; there's no way for the software to tell the difference between a legitimately generated signature and one produced by an attacker who has derived the same key mathematically. It's worth being precise about what the AES-256-encrypted payload itself experiences through all of this: nothing. The quantum computer never touches it, never attacks it directly, never needs to. It forges the master key that was used to hand the combination over in the first place, and everything downstream of that key falls with it.

What about AES-256? Why the symmetric layer is a different story

I've described ML-KEM in earlier posts as the thing that replaces our old public-key cryptography, and I want to head off a natural but mistaken way of hearing that: it isn't a straight upgrade over AES-256, as if the two were competing on the same scale. ML-KEM and AES-256 aren't competing solutions to the same problem — they defend two entirely different stages of the same handshake, and only one of those stages is what Shor's algorithm threatens.

Symmetric ciphers like AES-256 have no public/private key structure for Shor's algorithm to exploit — there's no hidden periodic relationship to find, because there's no asymmetric mathematical relationship there at all, just one secret number and a well-mixed cipher. The only quantum attack that applies to a symmetric cipher is a completely different algorithm, Grover's, and it does a fundamentally weaker thing: it accelerates *unstructured search* — brute-forcing every possible key — giving a **quadratic** speed-up rather than the exponential one Shor's gives against factoring. Concretely: brute-forcing AES-256 classically means trying, in the worst case, up to

2^{256} keys. Grover's algorithm on a quantum computer would need on the order of 2^{128} operations to find the key — which sounds dramatic until you notice that 2^{128} is still roughly 3.4×10^{38}, a number so large that even generous, decades-forward assumptions about quantum hardware don't bring it into practical reach. This is why cryptographers already treat AES-256 as quantum-resistant in practice, with no fundamental redesign needed — the existing 256-bit key size already builds in the margin Grover's algorithm would eat into, landing you back at a comfortable, effectively unbreakable 128-bit security level. There's a further honest wrinkle worth knowing: Grover's algorithm needs its operations to stay coherent, one after another, for a very long, uninterrupted sequence, which most cryptographers now think makes it considerably harder to actually run at scale than the equivalent-strength version of Shor's algorithm — so even the comfortable 2^{128} figure is arguably pessimistic about the real-world risk.

So: AES-256-GCM, already the choice MedLattice makes for the record itself, needed no replacement at all. What needed replacing was the RSA- or elliptic-curve-based handshake that established and signed around it — and that's specifically the job ML-KEM and ML-DSA do.

Why ML-KEM resists this, and RSA and secp256k1 never could

This is the part I actually find intellectually satisfying, because the answer isn't "bigger keys." Making an RSA key longer doesn't help against Shor's algorithm in any fundamental sense — the algorithm's advantage over classical factoring only grows more overwhelming as the key gets bigger, since the classical difficulty grows sub-exponentially while the quantum difficulty grows only polynomially. You cannot out-run an exponential gap by adding more digits; you can only postpone it briefly, at real cost to performance, while the underlying vulnerability stays exactly as total as it always was.

ML-KEM — standardised by NIST as FIPS 203, and the primary key-establishment mechanism in MedLattice's design — is built on a completely different family of mathematics: the **Module Learning With Errors** problem, a lattice-based construction. The intuitive picture is genuinely different in kind, not just in degree, from factoring or discrete logarithms. Imagine an enormous, perfectly regular grid of points stretching out in hundreds of dimensions at once — a lattice. The secret is hidden by taking a point on that lattice and nudging it slightly, with carefully calibrated random noise, so what's public is a point *near* a lattice point, not the lattice point itself. Recovering the secret means finding the nearest actual lattice grid-point to that nudged target — and unlike factoring or the discrete logarithm, this problem has no known periodic or repeating structure buried inside it for a quantum Fourier transform to lock onto. There is nothing for interference to sculpt towards, because there's no hidden period there to find. In three decades of dedicated

cryptanalytic effort — including serious, well-funded attempts specifically motivated by the arrival of quantum computing — nobody has found a quantum algorithm that gives an exponential speed-up against well-parameterised lattice problems, which is precisely why NIST selected ML-KEM and its signature counterpart ML-DSA (FIPS 204) as the general-purpose standards for the post-quantum era.

I want to be honest about the exact shape of that confidence, though, because it matters and because pretending otherwise would be intellectually sloppy. "No known quantum algorithm breaks this" is not the same statement as "this is mathematically proven unbreakable" — and, worth remembering, that's *also* true of RSA and elliptic curves; their forty-plus years of trusted use rested on exactly the same kind of claim, an absence of a known attack, right up until Shor found one. Lattice-based cryptography is newer and has had less collective scrutiny than RSA's four decades. That's precisely why MedLattice doesn't bet everything on it alone: ML-KEM-1024 is combined with the classical curve X448 through what's called an X-Wing hybrid combiner, so that breaking the connection requires defeating *both* the new, less battle-tested lattice assumption *and* the old, extensively-studied classical curve, simultaneously. The signature side does the same thing, pairing ML-DSA-87 with Ed448. And for the one signature the system relies on to prove the record's history hasn't been silently rewritten, MedLattice reaches for SLH-DSA — a hash-based scheme, resting on a third and mathematically unrelated foundation again, precisely so that even an unforeseen future attack on lattice mathematics specifically wouldn't take down every layer of the system at once. Belt, braces, and a third strap nobody's thought to cut yet.

The honest caveats

A few things I've deliberately simplified, in the spirit of the honesty sections I try to always include.

I gave you a working period of r recovered cleanly from a single measurement, because 64⁄256 happens to reduce exactly to 1⁄4. Real runs are noisier than that, and the full algorithm typically needs several repeated measurements and the general continued-fraction machinery to converge reliably on the correct period, especially as N grows. I skipped the substantial engineering reality of quantum error correction — today's quantum processors are noisy enough that thousands of imperfect physical qubits are needed to synthesise each single clean, reliable "logical" qubit the algorithm actually runs on, and building a machine with enough logical qubits to factor a real RSA-2048 key remains, by every credible estimate I've seen, a project of years to decades, not months. Where exactly on that timeline the real machine lands is genuinely disputed among serious people, and I don't think anyone honest claims to know the year with any precision.

None of that softens the underlying argument, though, and I think it's worth saying plainly why. The mathematics at the end of the timeline is settled, not merely probable, in a way that very little else in the current wave of technology anxiety can claim. And because of harvest-now-decrypt-later, the uncertainty in the hardware timeline doesn't actually buy patient data any safety margin — anything encrypted with vulnerable cryptography today can simply be stored and opened later, whenever that timeline resolves. For data that has to stay confidential for decades, the safe assumption was always going to be the pessimistic one.

Where this leaves the argument I opened with

I think this is why I find the quantum threat harder to set aside than the AI-takeover conversation, even though I take that conversation seriously too. Shor's algorithm isn't a forecast about how a future system might choose to behave. It's a proof, thirty years old, about what happens to two specific, extremely widely deployed mathematical assumptions the moment enough physical qubits exist in one machine — assumptions that the entire internet's trust layer, most blockchains including secp256k1-based ones, and, until I designed around it, my own patient record system all quietly depend on. The uncertainty left in this story isn't whether it happens. It's only when, and — because of harvesting — for data with a long enough shelf life, the "when" barely matters anyway.

That's the argument behind one paragraph in MedLattice's specification. If you want the fuller cryptographic picture of how the record itself is built around ML-KEM-1024, ML-DSA-87, and SLH-DSA, Not Shown Is Not Locked is the companion piece, and the full specification is linked from there.

Further viewing

Three videos prompted this post, in the sense that all three throw the word "superposition" into a Shor's algorithm explanation without ever building it, which is the gap I've tried to close above. Worth watching once you've read this, in roughly this order: How Quantum Computers Break Encryption | Shor's Algorithm Explained, minutephysics's fast, visual run through the algorithm; a second explainer at this link; and Quantum Expert Insight: Peter Shor, a short interview with Shor himself on what motivated the algorithm and how he thinks about its consequences. None of the three build superposition up from first principles the way I've tried to here — which was rather the point of writing this.