Fast and Secure Implementations of the Falcon Post-Quantum Cryptography Signature Algorithm

Sep 18, 2019 • Thomas Pornin

Earlier this month I released new, improved implementations of the Falcon post-quantum signature algorithm. The new implementations are available on the Falcon Web Site, along with a descriptive note. They are fast, secure, RAM-efficient, constant-time, portable, and open-source.

Many terms in the above paragraph may need some further explanations, so here they are.

What is a signature algorithm?

A cryptographic signature algorithm tries to mimic the act of signing a piece of paper, transposed into the computer world. In such an algorithm, the signer has a key pair consisting of a public key and a private key; both keys are mathematical objects that share some internal hidden structure. The public key is public and is used for verifying signatures; the private key is kept secret and is used for producing signatures. Formally, the algorithm consists of three operations:

- **Key pair generation.** Using some source of randomness, a brand new public/private key pair is produced.
- **Signature generation.** Using the private key, a signature is computed over a given message; the input message is an arbitrary sequence of bytes, and the signature is itself yet another mathematical object.
- **Signature verification.** Using the public key, a given signature value is verified to match a given message.

Signatures are used in many communication protocols, in particular for authentication purposes. When a web browser connects to an HTTPS Web site and the browser displays the famous padlock icon, this means that some authentication based on signatures has happened.

The hidden mathematical structure that supports the link between a public key and the corresponding private key is leveraged in the signature generation and verification algorithms. Moreover, that structure must be such that knowing the public key is not sufficient to recompute the private key; otherwise, it would not be possible to make the public key public while keeping the private key private. It is important here to notice that such impossibility of recomputing the private key is relative to what can be done practically with existing computers, which are powerful but not infinitely so. With an infinitely powerful computer, it would be easy to just try out all possible private keys until a match is found.

The most well-known signature algorithm is RSA, invented in the late 1970s. In RSA, the public key is a big composite integer, which is the product of two large prime numbers; the private key is knowledge of these prime numbers. Finding the private key from the public key is called integer factorization; it has been studied by mathematicians for at least 2500 years, and while some relatively efficient methods to do so are known, they fail to be practical when integers are too large. The current world record is a 768-bit integer (that's about 231 decimal digits). RSA is routinely used with 2048-bit integers (617 digits), and our best algorithms and computers cannot factor such large integers. Thus, RSA is safe… so far.

What does "post-quantum" mean?

Classical computers cannot break big RSA keys, but quantum computers can. "Quantum computers" are a bit of a misnomer: we may argue that all computers are "quantum" since they use semiconductors whose behavior cannot be adequately explained, except by using the parts of physics that have been developed in the last century or so, i.e. quantum mechanics. However, what is meant by "quantum computer" is a new kind of computing machine that tries to maintain an internal state which cannot be described as a collection of independent bits.

There is no correct intuitive description of a quantum computer. In fact, it is a longstanding problem of quantum physics; they don't make sense, philosophically speaking. That's what Schrödinger was explaining with his half-alive/half-dead cat (it was a "thought experiment"—no actual feline was harmed). The implications of the new physics were just plain ridiculous. Niels Bohr basically responded something along the lines of, "Yes, quantum physics don't make sense, but they work. Deal with it."

However, there is an incorrect description, which is that a quantum computer is in a superposition of many states and thus computes many things in parallel with the same hardware, at the same time, and that we can filter out the right answer at the end. This explanation is wrong, but it can give some correct intuitions about what the deal is with quantum computers. Namely, they can do some operations much more efficiently than classic computers.

Quantum computers can, in theory, break RSA. They can also break most of other signature algorithms that have been designed over the years, in particular systems based on discrete logarithm (e.g., DSA, ElGamal, Schnorr) and elliptic curves (e.g., ECDSA, EdDSA). The main drawback of quantum computers is that they don't exist. Or, at least, they don't exist yet. They can exist without breaking any of the existing laws of physics; actually building a working quantum computer still appears to be very challenging from a technological point of view. We have a few very reduced prototypes with a dozen "qubits," while a thousand would be needed to break RSA. We also have some "not really quantum" computers with thousands of "not really qubits," but these cannot break RSA. Working, full-size quantum computers are often said to be a mere ten years away. In fact, they have been "ten years away" for the last twenty years. Thus, it is unclear whether we will ever see a working quantum computer.

Nevertheless, with their theoretical ability to utterly devastate existing signature algorithms (and also asymmetric encryption and key exchange algorithms), quantum computers have prompted the search for new cryptographic algorithms that would remain unbreakable on quantum computers. This is where the metaphor of the superposition of many computations falls apart: there are still some computations that quantum computers cannot make with such parallelism. Such algorithms

are called post-quantum because they will survive the (possible) future transition to a world where a quantum computer exists.

What is Falcon?

Falcon is a candidate to the NIST Post-Quantum Cryptography Standardization Process.

When cryptographers want to make nice, secure algorithms, they tend to organize competitions. In a competition, several teams propose candidate algorithms, then spending some years trying to find flaws and weaknesses in the other candidates. The algorithms that survive the onslaught are then declared as “probably secure,” or at least “not obviously weak.” Competitions are needed because there is no known positive way to ensure that an algorithm is secure. We can show an algorithm to be insecure by explaining how to break it, but we don’t know how to prove that an algorithm cannot be broken. Sometimes we can prove that an algorithm cannot be broken in a specific way, but that’s not exclusive of other breaking methods that we did not think of.

Some famous past competitions resulted in the AES and SHA-3. As an added bonus, open competitions with international submitters and public analysis and workshops help build trust in the designs.

In 2017, the NIST initiated such a process for post-quantum cryptographic algorithms (in two categories, for key exchange protocols, and for signatures). The NIST is adamant that their project is not a competition; by this, they mean that they don’t want a single “winner”, but a portfolio of several algorithms, which they will proceed to describe into nice, implementable standards.

Falcon is one of the candidates submitted to this not-a-competition within the “signatures” category. The underlying mathematical structure is a lattice — a sort of sub-vector space with only integer coefficients — and breaking Falcon would entail being able to find short vectors in that lattice. This seems to be a hard problem for which no efficient solution is known, even with the help of quantum computers. In the NIST not-a-competition, many candidates use lattices, especially in the key exchange category.

When all candidates appear to be secure (nobody finds practical or even theoretical attacks), they tend to be compared with regards to performance. There are many metrics that may matter for a signature algorithm, in particular:

- Computational cost (CPU and RAM) of generating a new key pair
- Computational cost of generating a signature
- Computational cost of verifying a signature
- Size of private key
- Size of public key
- Size of signature

Indeed, consider a small embedded system such as a smart card. Such systems are constrained in everything, in particular available RAM, but also CPU power and I/O bandwidth. A smart card that contains a private key and uses it to sign messages, presumably for authentication purposes,

should generate its own key pair; the private key should never leave the smart card at all. Private key storage taxes the permanent storage abilities of the card, usually an EEPROM. Public keys and signatures are often exchanged on the wire; for example, a Web server sends its certificate chain to every connecting client, and each certificate contains a public key and a signature. Thus, public keys and signatures should be short.

Cost of signature verification also matters when a constrained system must perform the verification (e.g., a microcontroller checking the signature on its firmware when booting up) or when many signatures must be verified at a high rate, which would be typical of blockchain systems.

Among these metrics, Falcon runs well. It has the shortest signatures and public keys among lattice-based systems for a given security level. Signature generation is efficient; signature verification is very fast. Key pair generation is more expensive, but not awfully so; it can still be done in reasonable time on a small microcontroller. Private keys are also quite compact. There are some other candidates that can do better along one or the other metrics, but they are substantially worse for others. In a nutshell, Falcon is a good trade-off. This is why it has survived the first phase of the not-a-competition: while round 1 had 19 candidates in the “signature” category, only 9 have been selected for round 2.

What do the new implementations provide?

When Falcon was first submitted, it came with a “reference implementation” that was functional and quite efficient in terms of speed of generating signatures on “big” computers (e.g., smartphones, laptops, servers). However, it had some drawbacks:

- The reference implementation was not constant-time: its overall computation time and memory access pattern, depended on some private elements and, if observed through side channels, could potentially lead to private key recovery by outsiders. Indeed, on August 20th, 2019, researchers (Fouque, Kirchner, Tibouchi, Wallet and Yu) announced at CRYPTO 2019’s rump session a method for such key recovery that would theoretically work on Falcon’s old reference implementation.

- The reference implementation was a bit wasteful on RAM, using up to 180 kB for signature generation (with the “high” security level, called “Falcon-1024”).

- The reference implementation used floating-point numbers, leading to portability issues: precision may vary depending on the underlying architecture, and, on small microcontrollers without a FPU, these operations must be emulated with some compiler-provided routines which are typically not constant-time.

The new implementations are really four variants of a common core, which solves these issues:

- **Constant-time throughout.** Key pair generation and signature generation have been carefully modified to ensure constant-time behavior. Notably, a new Gaussian sampler, designed by Prest, Ricosset and Rossi, has been implemented. That sampler was demonstrated not to leak information in sufficient quantity to allow key recovery within the limits of the NIST threat model (which allows the attacker to observe up to 18 billions of billions of signature operations, possibly on messages all chosen by the attacker). This new sampler, in particular, prevents the attack announced at CRYPTO’s rump session.

    - Even signature verification, which nominally uses only public elements, was altered to be optionally constant-time in the unusual situations where public keys are not public, signatures are hidden, and signed messages are low-entropy secrets. This is not the common usage scenario, but in some rare cases, such properties might matter.
- **Efficient.** Some computations have been optimized to replace division operations by faster multiplications. On recent x86 CPU with AVX2 opcodes, Falcon-512 signatures (the "low" security level, still comparable to AES-128 and unbreakable with existing and foreseeable technology) can be computed at a rate of about 7700 per second on a single core of a MacBook Pro laptop CPU. This is about 5 times faster than the best RSA-2048 implementation on the same machine, and Falcon-512 offers an arguably better security than RSA-2048 (even if assuming that quantum computers will never exist).

    - RAM efficiency was improved as well. RAM usage was lowered to less than 40 kB for Falcon-512, and less than 80 kB for Falcon-1024.
- **Portable.** When using the FPU, precision was carefully set on multiple architectures to ensure strict reproducibility of test vectors. This was tested on various systems (32-bit and 64-bit x86, PowerPC and ARM, including some big-endian PowerPC systems).

    - For systems without a FPU, the new implementation embeds its own software emulation routines, which are constant-time and work everywhere there is a C compiler. All test vectors can be reproduced with exact precision, down to the last bit.
- **Compatible with constrained systems.** The FPU-less code was further optimized with assembly versions of some functions for ARM Cortex M4 microcontrollers. On a STM32F4 test board, all combinations of Falcon-512 and 1024 could be tested, including key pair generation. For Falcon-512, key pair generation time was close to 1 second (on average), while signature time was 126 milliseconds.

One last performance metric that I have not outlined explicitly above is "complexity." This is not about the computer science notion of algorithmic complexity, but the much more fuzzy and subjective notion of how hard the developer finds it to make a working, secure, and efficient implementation of the algorithm. I have implemented numerous cryptographic algorithms, including math-heavy things such as pairings on elliptic curves. Falcon is, in my experience, the "hardest" I've had to tackle so far. This shows in the size of the result: about 18000 lines of C code (including comments and whitespace, but not counting tests), which is far more than usual for a cryptographic algorithm. This could, arguably, count as a negative point about Falcon.

On the other hand, difficulty to write code is very dependent on the context. Indeed, one can say that the easiest software to implement is that which has already been implemented by somebody else. These new Falcon implementations are open-source, with a very permissive license (MIT). Another developer faced with the task of implementing Falcon now has a very easy way, which is to simply grab a copy of my code and use it. Thus, we can now claim that this "implementation hardness" issue is mostly solved.

A Final Warning…

The new implementation of Falcon was published on August 2nd, 2019. The text above was the original blog post, ending in a triumphant note about how, once development efforts were done, we could claim that complexity had been vanquished.

Such hubris could not go unpunished. On August 30th, Markku-Juhani O. Saarinen, while looking at the source code, found two distinct bugs in the “Gaussian sampler”. This piece is responsible for “adding noise” during signature generation: each signature is an approximate solution to an equation that involves the public key and the message to sign; finding that solution requires knowledge of the private key. However, if the signature is a “too exact” solution, then it leaks information on the private key. The sampler generates random extra noise to avoid this information leak; the mathematical analysis in the Falcon specification shows that this strategy is effective as long as the sampler faithfully follows that Gaussian distribution.

With the flawed implementation, the wrong distribution was used, leading to signatures that leaked information on the private key. Moreover, it also made the algorithm slightly faster than expected. The code has now been fixed, with a small performance hit (e.g. the 123 milliseconds per signature on the Cortex M4 were originally 117 milliseconds per signature).

An important point here is that bugs in samplers are very hard to detect during development. All produced signatures were “valid”: they passed verification without any error. Reproducible test vectors can help, but only insofar as such vectors exist, which was not the case for this development effort: since the reference implementation was using a different strategy for the sampler, existing test vectors could not be used to verify the correctness of the signature generation. As such, while the current code version *seems* to be correct, there is no real guarantee, and the development methodology (aka, “the developer is super-careful”) has already failed once. Only further external scrutiny will gradually help confirm (or not!) the quality of the current implementation.
