Deterministic Falcon Signatures

David Lazar and Chris Peikert

Algorand, Inc.

November 3, 2021

1 Motivation and Overview

Falcon [FHK$^+$20] is a signature scheme that instantiates the Gentry–Peikert–Vaikuntanathan (GPV) [GPV08] 'hash-and-sign' framework for lattice-based digital signatures. It uses the NTRU family of lattices, and its trapdoor sampler involves several algorithmic innovations and optimizations to obtain good time and space efficiency. It has been selected as one of three finalist signature schemes in the NIST Post-Quantum Cryptography project.

1.1 Re-Signing Strategies

In the GPV framework, each message to be signed is first hashed to a digest 'syndrome' (more precisely, a coset of a certain lattice defined by the public key). Then, a randomized 'trapdoor sampler' algorithm is used to select one of many valid signatures for this syndrome. For security, the following rule must be maintained:

for any given public key,
the signer must never reveal two different signatures for the same syndrome.

Violating this rule would break a central assumption needed by the GPV unforgeability proof, and may even enable an attacker to forge signatures for arbitrary messages (because it would reveal short vectors in the lattice associated with the public key).

The work of GPV recalls three generic options for enforcing the above rule:

- **Statefulness.** The signer can maintain a list of the signed messages and their signatures, and return the same signature if the same message is ever to be re-signed. This approach is very difficult to properly implement in practice, especially (but not only) if multiple devices may need to sign messages using the same private key.

- **Hash randomization.** Whenever signing a message, the signer can choose a random 'salt' (or 'nonce') and incorporate it into the hashing process to obtain a randomized digest. (The salt is included with the signature so that the verifier can compute the same digest.) If the salt has sufficient randomness, then no salt value will ever be repeated, so neither will any digest syndrome (under typical assumptions about hash function). Importantly, this non-repetition property is all that is required by the GPV security proof in the random-oracle model; one could even allow the adversary to choose the salt, as long as no message-salt pair is ever repeated.

- **Derandomization.** When signing a message, the signer can use a pseudorandom function (applied to the message) as the source of the random choices for the trapdoor sampler. So, when re-signing the same message, the same pseudorandom choices are made, yielding the same ultimate output.

  However, for this strategy to work in practice, it is very important that the entire signing procedure be *functionally equivalent* across all the relevant signing devices, where 'device' broadly encompasses the entire relevant computing stack: hardware processors, operating system, compiler (including versions and optimizations), implementation of the signing algorithm, etc.

  For Falcon, what further complicates this picture is the fact that it inherently relies on *floating-point* (FP) operations in its trapdoor sampling procedure, and it is well known that slight deviations in FP computations can arise due to optimizations like reordered or combined instructions, or even slight differences in the implementations of floating-point units (FPUs).

The Falcon specification opts to use hash randomization, using a 320-bit (40-byte) salt value that is prepended to the message before hashing (and included in the signature). This approach provides the flexibility to safely use specialized optimizations (which may yield functionally inequivalent signing procedures) across a variety of computing devices and configurations. However, it also requires signing devices to have a high-quality source of randomness, which is not available in all contexts.

1.2 SNARK (Un)Friendliness

Hash randomization means that the digest syndrome depends on the random salt in the signature. This syndrome is calculated using a hash function that is chosen to instantiate an idealized random oracle. For recommended choices of hash functions like SHAKE, this makes signature verification very 'unfriendly' for SNARKs.[1]

More concretely: for a given message, to prove knowledge of a valid Falcon signature for this message (relative to some public key), the SNARK would need to embed the full computation of the digest syndrome, because it depends on the salt in the signature.[2] More generally, suppose one wishes to succinctly prove knowledge of *many* signatures (relative to respective public keys) for the same message, as in compact certificates [MRV$^+$21]. Then the SNARK must embed the computations of *all* the digest syndromes, which vary from signature to signature due to the different salts. For existing SNARK systems, this would require a prohibitive amount of computation for the SNARK prover.

SNARK-friendliness of derandomization. On the other hand, a *derandomized* GPV signature scheme is much more SNARK-friendly in the above scenarios. This is because the digest syndrome depends *only on the message*, which is known to the verifier as (part of) the statement to be proved. So, the hashing can be done 'outside the SNARK', i.e., both prover and verifier can compute the digest, and the SNARK can simply prove that the signature is valid relative to that digest. For GPV signatures, this part of the signature verification is very SNARK-friendly, consisting essentially of just modular linear equations and enforcement of a norm bound (essentially, a range check). More broadly, in the single-message, many-signatures scenario of compact certificates, the *same* digest syndrome is used for every signature. So, this single hash computation can again be done outside the SNARK, or even inside the SNARK just once, and amortized over all the signatures.

[1]A SNARK is a Succinct Noninteractive ARgument of Knowledge; it allows one to prove complex NP statements, possibly having large witnesses, via short proofs that are often also very fast to verify.

[2]Alternatively, once could keep the salt 'in the clear' and compute the syndrome digest 'outside the SNARK.' However, this requires that the SNARK verifier know the salt (along with the message), and it is not succinct in the multiple-signature context described next, because all the various salts would need to be known by the verifier.

1.3 Deterministic Falcon

Motivated by the above considerations, this document specifies a derandomized—or more precisely, 'deterministic signing'—variant of Falcon, following the third option above. Despite the qualitative differences between this approach and randomized hashing, it turns out that it is straightforward to define a deterministic mode in terms of the randomized mode via minor tweaks; see Section 2. Moreover, the deterministic mode can be implemented easily and robustly using Falcon's existing interface and implementation, with just a small amount of 'wrapper' code; see Section 3.

2 Specification

2.1 Keys

In deterministic Falcon, public verification keys and private signing keys keys are identical to those in randomized Falcon. Moreover, they are safely interoperable between modes: a private key may be used to sign messages in both randomized and deterministic mode, with no loss in security versus using just one of these modes. This is simply because the randomized mode is vanishingly unlikely to choose the fixed salt value that the deterministic mode uses.

It is stressed that for deterministic mode,

a private key should not be used to sign the same message digest using functionally inequivalent signing procedures.

This may violate the desired determinism, and thereby the central rule needed for security (see Section 1.1). In particular, if the input-output functionality of the signing procedure ever changes—e.g., due to a bug fix, an optimization, or a port to a different kind of computing device—then new keys should be generated and used in the new procedure.[3]

2.2 Signatures

In deterministic Falcon, valid signatures are a slight modification of those for standard (randomized) Falcon: they simply use a fixed salt value (which is omitted from the signatures themselves), and have an extra byte prepended to indicate their special type.

Randomized (salted) signatures. First recall from [FHK$^+$20, Section 3.11.3] that in randomized Falcon, a signature contains a 'salt' (or 'nonce') value, and has the format

$$\mathsf{salted\_sig} = \mathsf{header\_byte} \| \mathsf{r\_bytes} \| \mathsf{s\_bytes}\ ,$$

where

- header_byte is a header byte defining the Falcon dimension parameter $n = 2^\ell$ and signature encoding, as defined in [FHK$^+$20, Section 3.11.3];
- r_bytes is a 40-byte salt string;
- s_bytes is a string of bytes (whose length depends on n and the signature encoding) representing the signature's 's value'.

[3]Alternatively, it may be safe just to change the fixed salt used in the new signing procedure, to make each message hash to a different digest syndrome. However, the verifier would need to know which salt to use for a given signature. This could be achieved by letting the prefix_byte (defined in Section 2.2 below) specify the 'version' of the salt that was used.

Deterministic (unsalted) signatures. A deterministic signature $\mathsf{determ\_sig}$ has an additional prefix byte but no salt value, and is defined to have the format

$$\mathsf{determ\_sig} = \mathsf{prefix\_byte} \| \mathsf{header\_byte} \| \mathsf{s\_bytes} \ , \tag{1}$$

where:

- $\mathsf{prefix\_byte} = \texttt{0x80}$, i.e., the byte having value 128 in decimal;
- $\mathsf{header\_byte}$ and $\mathsf{s\_bytes}$ are exactly as in randomized Falcon signatures (see above).

2.2.1 Validity and Verification

For a given public key and message, a deterministic signature $\mathsf{determ\_sig}$ as in Equation (1) is defined to be valid if and only if the corresponding salted Falcon signature

$$\mathsf{salted\_sig} = \mathsf{header\_byte} \| \mathsf{fixed\_salt}_n \| \mathsf{s\_bytes} \tag{2}$$

is valid for the same public key and message (respectively). Here $\mathsf{fixed\_salt}_n$ is a certain fixed 40-byte salt string that depends only on the dimension parameter n defined by $\mathsf{header\_byte}$; this salt is defined in Section 2.2.2 below.

Therefore, to verify $\mathsf{determ\_sig}$ for a given public key and message, one can simply construct the corresponding $\mathsf{salted\_sig}$ and verify it, as follows:

1. check and then remove the initial $\mathsf{prefix\_byte}$,
2. insert $\mathsf{fixed\_salt}_n$ between $\mathsf{header\_byte}$ and $\mathsf{s\_bytes}$ (for the value of n defined by $\mathsf{header\_byte}$), and
3. verify the resulting $\mathsf{salted\_sig}$ as an ordinary 'salted' Falcon signature, for the same public key and message (respectively).

2.2.2 Fixed Salt Values

For any legal value of n for Falcon (i.e., $n \in \{2^1, 2^2, \dots, 2^{10}\}$), the string $\mathsf{fixed\_salt}_n$ is defined to be the ASCII representation of `FALCON_DET`, followed by the ASCII representation of n in decimal (with nonzero leading digit), followed by the required number of all-zero padding bytes to make the length exactly 40 bytes. In particular:

- For $n = 512$, $\mathsf{fixed\_salt}_n$ is the ASCII representation of `FALCON_DET512`, followed by 27 zero bytes.
- For $n = 1024$, $\mathsf{fixed\_salt}_n$ is the ASCII representation of `FALCON_DET1024`, followed by 26 zero bytes.

3 Implementation

This section describes an implementation [LP21], called `falcon_det1024`, of a subset of deterministic Falcon signatures as defined in Section 2. This implementation is just a small amount of simple 'wrapper' code around the official, unchanged Falcon interface and implementation.

The functionality implemented by `falcon_det1024` is limited to fixed Falcon parameter $n = 2^{10} = 1024$, and signatures in 'padded' format (which have a fixed length).[4] More specifically:

[4]Falcon-1024 targets NIST post-quantum security category 5, which is the highest defined level.

- the key-generation procedure creates keys only for $n = 1024$;
- the signing procedure requires a private key having $n = 1024$, and creates only padded signatures; and
- the verification procedure accepts a signature if and only if it has $n = 1024$, is in padded format, and is valid according to the specification from Section 2.2. (It therefore accepts any signature produced by `falcon_det1024`'s signing procedure.)

3.1 Keys

As mentioned in Section 2.1, randomized and deterministic Falcon have identical key formats, and keys are interoperable between the two modes. Therefore, `falcon_det1024` simply invokes Falcon's key-generation function on suitable fixed arguments corresponding to $n = 1024$.

3.2 Signature Verification

To verify a given deterministic signature, `falcon_det1024` checks that the signature has the proper format (padded and $n = 1024$), then proceeds according to the steps given in Section 2.2.1: it constructs the corresponding salted signature, and then verifies it using Falcon's verification procedure as a 'black box.'

3.3 Robust Deterministic Signing

The signing procedure in `falcon_det1024` is more subtle than the other two procedures. However, it is still a straightforward application of Falcon's 'streamed' interface, which:

- externalizes (to the client) the hashing of the salt and the data to be signed[5], and
- allows the client to provide (the state of) a pseudorandom generator, which the signing procedure uses for its secret random choices.

This interface allows `falcon_det1024` to hash the data with a fixed salt, and to provide a deterministic stream of secret pseudorandom bits that depends solely on the private key and the data. Hence, when some data is re-signed under the same private key, the same stream is produced, thus yielding a fully deterministic signing procedure across all conforming computing devices (though see Section 3.3.2 below for caveats).

3.3.1 Implementation Details

Falcon's streamed interface provides a function `falcon_sign_dyn_finish`, which produces a salted signature when given the following arguments (among others that are not relevant to the present discussion):

- a SHAKE context `hash_data` that represents the hashed salt and data, and
- another SHAKE context `rng` that is used to generate a stream of secret pseudorandom bytes for the trapdoor-sampling procedure.

The `falcon_det1024` signing procedure calls `falcon_sign_dyn_finish` with the following SHAKE contexts to obtain a salted signature, then removes the salt and prepends the prefix byte to produce the corresponding unsalted signature:

- The `hash_data` context is prepared by injecting $\mathsf{fixed\_salt}_{1024}$, then the data to be signed.

[5]For consistency with the Falcon implementation, in this section, the term 'data' is used for the (to-be-)signed message.

- The `rng` context is prepared by injecting a zero byte, then a byte representing the $\texttt{logn} = 10$ parameter, then the entire private key string, then the data to be signed. This `rng` context represents the function

  $$F_{\texttt{privkey}}(\texttt{data}) := \texttt{SHAKE}(0\|\texttt{logn}\|\texttt{privkey}\|\texttt{data})\ .^6$$

  Under standard assumptions about SHAKE, for any fixed value of `logn` this is a deterministic, pseudorandom function of just the private key and the data, as desired.

Note that both of these contexts require the injection of the entire data to be signed; for extremely long data, this may result in a noticeable slowdown. In such a case, applications may opt to 'pre-hash' the data using a collision-resistant hash function and instead sign the hash value.[7]

3.3.2 Robustness Across Devices

For fully deterministic signing across different computing devices, it is not necessarily enough to generate a repeatable stream of pseudorandom bytes; the actual signing procedures that *consume* those bytes should ideally be *functionally equivalent*, i.e., they should have the same input-output behavior. Failing that, the procedures should be near-equivalent, i.e., equivalent on 'almost all' inputs, keeping in mind that the attacker may have partial or total control over the data to be signed.

For Falcon, ensuring functional (near-)equivalence can be a delicate matter, because the signing procedure internally uses a sophisticated algorithm that relies on floating-point operations. The presence of different floating-point units and code optimizations, such as 'fused multiply-and-add' (FMA), on different devices can potentially result in slight discrepancies that could result in functional inequivalence.

Floating-point emulation. Fortunately, Falcon includes a floating-point emulation mode (`FALCON_FPEMU`), which implements the required subset of floating-point operations using 32- or 64-bit integers. This mode is fully deterministic and constant time, assuming a C99-compliant compiler and hardware that implements a certain few operations in constant time. On modern desktop/server-class CPUs, the emulation slows down key generation by only about a 2x factor, signature generation by about a 15x factor, and signature verification not at all (because floating-point operations are not used in verification).

By default, `falcon_det1024` uses Falcon's floating-point emulation, and also explicitly disables certain optimizations that could potentially yield functional inequivalence.[8] It is *strongly recommended* that applications retain this emulation, unless performance considerations absolutely require otherwise. In such a case, caution should be exercised to ensure functional (near-)equivalence, and certainly including compliance with the test vectors (see below). Where possible, operational restrictions may also be appropriate, such as restricting the use of every private key to a specific kind of device and configuration.

Test vectors. To help 'sanity check' other devices, implementations, compilers, optimizations, etc. for functional (near-)equivalence, `falcon_det1024` includes 512 known-answer tests (KATs) for its deterministic mode, along with a `test_deterministic` program for checking them (and generating more, if desired). Any deviation from the KATs indicates a lack of the desired equivalence. However, agreement with all the

[6]The `logn` and zero bytes are used for domain separation and potential future versions, respectively.

[7]Alternatively, in just the `rng` context, the `data` input could be replaced by a short collision-resistant hash. However, this change would yield a highly inequivalent signing procedure, and in particular would be incompatible with the known-answer tests for the deterministic mode.

[8]Falcon already disables most of these optimizations when floating-point emulation is enabled, but `falcon_det1024` disables them explicitly as a defensive measure.

KATs does not prove equivalence for all possible inputs; for this, one could potentially use an automated theorem prover.

The authors have checked the KATs for a variety of Falcon options (native versus emulated floating-point operations, AVX2 and FMA optimizations), compilers (`gcc` and `clang`), and compiler optimizations (`-O0` through `-O3`). Under all these variants, *no deviation from the KATs has been detected*. While this does not come close to proving that all the various procedures are equivalent, it does show that the Falcon code is not very sensitive to configuration changes on the available computing platforms of interest.

References

[FHK+20] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. Falcon: Fast-Fourier lattice-based compact signatures over NTRU, 2020. Specification v1.2, `https://falcon-sign.info/`.

[GPV08] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, pages 197–206. 2008.

[LP21] D. Lazar and C. Peikert. Deterministic Falcon-1024 implementation, November 2021. `https://github.com/Algorand/falcon`.

[MRV+21] S. Micali, L. Reyzin, G. Vlachos, R. S. Wahby, and N. Zeldovich. Compact certificates of collective knowledge. In *IEEE Symposium on Security and Privacy*, pages 626–641. 2021.