Deterministic Falcon Signatures

David Lazar Chris Peikert

February 3, 2022

Contents

1 Motivation and Overview

Falcon [**?**] is a signature scheme that instantiates the Gentry–Peikert–Vaikuntanathan (GPV) [**?**] '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 formally, a coset of a certain lattice defined by the public key). Then, a 'trapdoor sampler' algorithm is used to randomly 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 inequivalent signatures for the same digest syndrome.

'Inequivalent' means that the signatures represent different short vectors having the same syndrome; signatures representing the same short vector under different valid encodings are equivalent.

Violating the above rule would break a central assumption needed by the GPV unforgeability proof. Moreover, it may even enable an attacker to forge signatures, because it would reveal short vectors in the lattice associated with the public key. GPV recalls three generic options for enforcing the 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.
- **Randomized hashing.** 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 deterministic hash function to map the message to a syndrome, and 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 sampling 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 sampling algorithm, etc.

A further complication for Falcon is the fact that its trapdoor sampler inherently relies on *floating-point* (FP) operations. 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 40-byte (320-bit) 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 computed using a real hash function that is used to instantiate an idealized random oracle. For recommended 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 [**?**]. 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.

1.2.0.1 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

[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.

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.

1.3 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 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 3.

2 Specification

This section specifies a deterministic mode for Falcon. In short, this mode essentially removes the salt from ordinary Falcon signatures, instead using a fixed salt value (for compatibility), and the signer uses deterministic pseudorandom choices in its sampling algorithm.

2.1 Determinism and Versioned Salt

In deterministic mode, the signer first deterministically maps the message to a syndrome digest, by hashing (just as in the randomized mode) the message and a certain *fixed* salt value. It then deterministically generates a signature for this digest, by supplying a trapdoor sampling algorithm with its secret 'random' choices using a pseudorandom function of the message. The precise details of the sampling procedure and the pseudorandom function are left unspecified here, and are determined by the implementation; the goal is for a particular family of implementations to instantiate the same deterministic behavior. It is stressed that

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

Doing so may violate the desired determinism, and thereby the central rule needed for security (see 1.1).

In particular, if the input-output functionality of the sampling procedure ever changes—e.g., due to a bug fix, an optimization, or a port to an different kind of

computing device—then either the private key or the deterministic mapping from messages to syndromes should be completely refreshed. One way to achieve this refreshing is to generate and distribute entirely new keys for use with the new procedure; however, this may be operationally difficult in certain applications.

As an alternative to refreshing keys, this specification includes a versioning mechanism that allows for refreshing the message-to-syndrome map, by slightly changing the salt. More specifically, each signature contains a 'salt version,' which is included as part of the otherwise fixed salt used by the signer and verifier. When modeling the hash function as a random oracle (as is standard with GPV), changing the salt version yields a completely fresh random digest for every message. Therefore, signers can safely use inequivalent sampling procedures with the same secret key, simply by using different salt versions.[3]

2.2 Keys

In deterministic Falcon, public verification keys and private signing 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 any of the (versioned) salt values that the deterministic mode uses.

2.3 Signatures

In deterministic Falcon, signatures have an 'unsalted' format, which is just a slight modification of the salted format for standard (randomized) Falcon. In summary, unsalted format implicitly uses a fixed (versioned) salt value, which is omitted from the signatures themselves, and the header byte is slightly different to indicate the different format.

2.3.1 Salted and Unsalted Formats

2.3.1.1 Salted signatures.

First recall from [**?**, Section 3.11.3] that in randomized Falcon, a signature contains a 'salt' (or 'nonce') value, and has the format

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

where

- $\mathsf{salted\_header\_byte}$ is a header byte that defines the Falcon dimension parameter $n = 2^\ell$ and the signature encoding, as defined in [**?**, Section 3.11.3]. Note that the most-significant bit of $\mathsf{salted\_header\_byte}$ is always 0.

[3] Note that for the SNARKed compact certificate application described in 1.2, it is best if all the signatures in a given certificate use the same salt version, so that the SNARK needs to prove correctness of only a single digest. But the salt version may still change from one certificate and SNARK proof to another.

- 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 core of the signature (i.e., the output of the trapdoor sampler).

2.3.1.2 Unsalted signatures.

An unsalted signature unsalted_sig essentially just replaces the arbitrary salt string r_bytes with a single byte specifying a fixed (versioned) salt string. Specifically, it has the format

$$\mathsf{unsalted\_sig} = \mathsf{unsalted\_header\_byte} \| \mathsf{salt\_version\_byte} \| \mathsf{s\_bytes} \ ,$$

where:

- unsalted_header_byte is exactly as in salted Falcon signatures (see above), except that its most-significant bit is always 1 (not 0).
- salt_version_byte is a byte specifying the version of the fixed salt that was used by the signer. This implicitly selects a specific message-to-syndrome mapping, as describe at the start of the section.
- s_bytes is exactly as in salted Falcon signatures.

2.3.2 Validity and Verification

For a given public key and message, an unsalted signature unsalted_sig as in [eq:unsaltedsig] is defined to be valid if and only if the corresponding salted Falcon signature

$$\mathsf{salted\_sig} = \mathsf{salted\_header\_byte} \| \mathsf{salt\_version\_byte} \| \mathsf{preversioned\_salt}_\ell \| \mathsf{s\_bytes}$$

is valid for the same public key and message (respectively), where:

- salted_header_byte is simply unsalted_header_byte with its most-significant bit set to 0, and
- preversioned_salt$_\ell$ is a certain 39-byte string that depends only on the parameter $\ell = \log_2(n)$ defined by the header byte; this string is defined in 2.3.3 below.

Note that in salted_sig, the 40-byte string r_bytes := salt_version_byte$\|$preversioned_salt$_\ell$ serves as the salt.

Therefore, to verify unsalted_sig for a given public key and message, one can simply construct the corresponding salted_sig and verify it, as follows:

1. check that the most-significant bit of the header byte is 1, change it to 0, and determine the value of ℓ defined by the header;
2. insert preversioned_salt$_\ell$ between salt_version_byte and s_bytes, and

3. verify the resulting salted_sig as an ordinary salted Falcon signature, for the same public key and message (respectively).

2.3.3 Pre-Versioned Salt

For any legal value of $\ell = \log_2(n)$ for Falcon (i.e., $\ell \in \{1, 2, \ldots, 10\}$), the 39-byte string $\mathsf{preversioned\_salt}_\ell$ is defined as:

1. a byte representing the value of ℓ (e.g., `0x09` for $n = 512$, and `0x0A` for $n = 10$)[4], followed by
2. the ASCII representation of `FALCONDET`, followed by
3. the required number of all-zero padding bytes to make the length exactly 39 bytes.

(The string `FALCONDET` is included simply to provide domain separation from other uses of the hash function.)

3 Implementation

This section describes an implementation, called `falcondet1024` [**?**], of a *subset* of the deterministic Falcon signature scheme defined in 2. This implementation is obtained by just a small amount of simple 'wrapper' code around the official, unchanged Falcon interface and implementation.

The functionality implemented by `falcondet1024` is limited to fixed Falcon parameter $n = 2^{10} = 1024$, which targets NIST post-quantum security category 5 (the highest defined level). It is also limited to signatures in either of Falcon's 'compressed' or 'constant-time' (CT) formats, which have variable and fixed lengths, respectively; it does not support the fixed-length 'padded' format. More specifically, all functions are limited to $n = 1024$, and:

- signing creates a signature *in compressed format only*;
- there is an additional function that converts a given signature from compressed format to CT format;
- verification accepts a signature if and only if it is in the caller-specified format—*either compressed or CT*—and it is valid according to the specification from 2.3.

The rationale for the choice of supported (and unsupported) signature formats is given below in 3.5.

[4] Due to the format of the header byte, the byte representing ℓ can be obtained as $\mathsf{unsalted\_header\_byte}$ AND `0x0F`.

3.0.0.1 Strong unforgeability.

Because `falcondet1024` supports transcoding from one signature format to another, it does not, strictly speaking, preserve Falcon's strong unforgeability (if an application ever accepts signatures in CT format).[5] However, `falcondet1024` does still preserve a slightly relaxed form of strong unforgeability, where the set of message-signature pairs revealed by the signer is defined to include both the directly released compressed-form signatures *and* their transcoded CT-form versions.

3.1 Keys

As mentioned in 2.2, randomized and deterministic Falcon have identical keys, and keys are interoperable between the two modes. Therefore, `falcondet1024` 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, `falcondet1024` checks that the signature has the expected parameter $n = 1024$ and caller-specified format (either compressed or CT), then proceeds according to the steps given in 2.3.2: it constructs the corresponding salted signature, and then verifies it using Falcon's verification procedure as a 'black box.'

3.3 Signature Conversion

To convert a signature from compressed format to CT format, `falcondet1024` simply uses Falcon's internal encoding and decoding functions to decompress the given signature and then re-encode it. This transcoding does not require any secret information, nor even any other public information apart from the signature itself.

3.4 Robust Deterministic Signing

The signing procedure in `falcondet1024` is more subtle than the other procedures. However, it is still a straightforward application of Falcon's 'streamed' interface, which:

- externalizes (to the caller) the hashing of the salt and the data to be signed[6], and

[5] Note that Falcon's signatures are inherently transcodable, so it is strongly unforgeable only if the application enforces the use of a single signature format when signing and verifying. However, the envisioned applications of `falcondet1024` will violate this requirement.

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

- allows the client to provide (the state of) a pseudorandom generator, which the sampling procedure uses for its secret random choices.

This interface allows `falcondet1024` to hash the data with a fixed (versioned) salt, and to provide a deterministic stream of secret pseudorandom bits that depends solely on the private key and the data to be signed. Hence, when the same 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 3.4.2 below for caveats).

3.4.1 Implementation Details

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

- a SHAKE context `hashdata` 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 `falcondet1024` signing procedure calls `falcon signdynfinish` with the following SHAKE contexts to obtain a salted signature with a fixed (versioned) salt string, then removes the salt (except for the version byte) and tweaks the header byte to produce the corresponding unsalted signature.

- The `hashdata` context is prepared by injecting the appropriate (versioned) salt string, then the data to be signed.
- The `rng` context is prepared by injecting a byte representing the $\texttt{logn} = 10$ parameter, then the entire private key string `privkey`, then the `data` to be signed. The output of this `rng` context represents the function

  $$F_{\texttt{privkey}}(\texttt{data}) := \texttt{SHAKE}(\texttt{logn} \| \texttt{privkey} \| \texttt{data}) \ .^{7}$$

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

Note that both of the above SHAKE 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.[8]

[7] The `logn` byte is used for domain separation.

[8] 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.

3.4.2 Robustness Across Devices

For fully deterministic signing across different computing devices, it is not 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.

3.4.2.1 Floating-point emulation.

Fortunately, Falcon includes a floating-point emulation mode (`FALCONFPEMU`), 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 it does not use floating-point operations).

By default, `falcondet1024` uses Falcon's floating-point emulation, and also explicitly disables certain optimizations that could potentially yield functional inequivalence.[9] 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 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.

3.4.2.2 Test vectors.

To help 'sanity check' other devices, implementations, compilers, optimizations, etc. for functional (near-)equivalence, `falcondet1024` includes known-answer tests (KATs) for its deterministic mode, along with a `testdeterministic` 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 KATs does not prove equivalence for all possible inputs; for this, one could potentially use an automated theorem prover.

[9] The Falcon implementation already disables most of these optimizations when floating-point emulation is enabled, but `falcondet1024` disables them explicitly as a defensive measure.

The authors have checked the KATs on modern Intel CPUs 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.

3.5 Rationale for Supported Signature Formats

The rationale for the supported signature formats—signing only in compressed format, verifying in either compressed or CT format, and not allowing padded format at all—is as follows.

3.5.0.1 Signing in only one format.

Signing is limited to just one format (i.e., compressed) in order to robustly adhere to the central rule needed for security: the signer must never reveal two *inequivalent* signatures for the same digest syndrome (see 1.1). Allowing the signer to output signatures in two different formats might open the possibility of violating this rule.

Note that the current official Falcon implementation actually *can* sign deterministically in both compressed and CT formats without violating the above rule, because its signing procedure generates *equivalent* signatures in these formats, when all the inputs (other than the requested format) remain the same. However, this is *not* the case for 'padded' format, because Falcon's signer retries whenever a candidate signature does not fit into padded format, even though it may fit into compressed and CT formats. Given these subtleties, the possibility of a future change in implementation, and the low cost of transforming between formats, it seems best to restrict deterministic signing to just one format.

3.5.0.2 Properties of the formats.

Compressed format, which has a variable length, is Falcon's default, and has the shortest average length of all of the formats. Padded format has a fixed length, which is only slightly longer than the average compressed-format length. Clearly, shorter signatures reduce the communication required for sending many signatures over the network to prepare compact certificates, and the size of such certificates. However, compressed and padded formats are quite SNARK-unfriendly, because they require a complicated decompression algorithm that has many data-dependent conditionals and loops.

On the other side of the spectrum, CT format is significantly larger (about 25% more than compressed and padded formats), so using it for network communication and in storage is wasteful. However, it is very SNARK-friendly, because it represents each signed-integer polynomial coefficient with a fixed number of bits,

in two's complement. Therefore, each coefficient can be reconstructed via a fixed linear function of the corresponding block of bits, which is very inexpensive in terms of SNARK constraints.

3.5.0.3 Interchangeability.

CT format is *fully interchangeable* (in both directions) with compressed format, but is only interchangeable with padded format in one direction.

In more detail: any valid, possibly maliciously generated compressed-format signature (for any given public key and message, which may also be maliciously generated) can be efficiently transformed into a valid CT-format signature (for the same public key and message), and vice-versa. The latter direction works because the variable length of compressed format can expand to represent any valid signature, with a proven maximum length. By contrast, any valid padded-format signature can be efficiently transformed into a valid CT-format signature, *but not necessarily vice-versa*. This is because transforming a valid CT-format signature may require more bytes than are allowed by padded format.

3.5.0.4 SNARKs.

Full interchangeability is very important for the correctness of transforming a compact certificate into a SNARK proof, and also for the security argument connecting the soundness of the SNARK to the security of compact certificates, and then to the underlying signature scheme. More specifically:

- Any valid compact certificate—i.e., one that satisfies its verifier—must be efficiently transformable to a valid SNARK proof—i.e., one that satisfies its own verifier. As mentioned above, the SNARKed circuit needs signatures to be given in CT format, but compact certificates (and the network used to create them) may wish to use a more compact format. Therefore, any valid signature in the compact format must always be efficiently transformable into a valid signature in CT format. Fortunately, this is the case for both of Falcon's 'compressed' and 'padded' formats.
- The security argument uses the opposite direction: it requires that any witness that satisfies the SNARKed circuit—i.e., one containing valid CT-format signatures—can be efficiently transformed into a valid compact certificate (or a 'break' of the SNARK or related cryptographic primitive). In addition, any valid compact certificate should be transformable into one or more valid signatures (or a 'break' of a component primitive), etc.

Because all Falcon signature formats can be transformed to CT format, one way to achieve the above goals is to ensure that CT format is accepted by the compact-certificate verifier *and any other subcomponent that verifies Falcon signatures*. However, this is a brittle solution, because it requires every verifier in the chain to accept a format that may be unnatural for its specific purpose, and may even be unused in reality.

A robust approach, which is what `falcondet1024` adopts, is simply to ensure that all signature formats that may be accepted by the various verifiers (the SNARK circuit, the compact-certificate verifier, etc.) are fully interchangeable. This ensures that valid witnesses can always be transformed to include signatures of the desired format(s). This full-interchangeability policy leads `falcondet1024` to support compressed and CT formats, but not padded format.

MRV$^+$21 urlstyle

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/.

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

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

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.