Roll. No. A016	Name: Varun Khadayate
Class B. Tech CsBs	Batch: 1
Date of Experiment: 11-09-2022	Subject: Cryptology

Aim

Study any one quantum cryptology cipher

Falcon Quantum Cipher

FALCON

Fast-Fourier Lattice-based Compact Signatures over NTRU

About Falcon

FALCON is a cryptographic signature algorithm submitted to NIST Post-Quantum Cryptography Project on November 30th, 2017. It has been designed by: Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang.

The point of a post-quantum cryptographic algorithm is to keep on ensuring its security characteristics even faced with quantum computers. Quantum computers are deemed feasible, according to our current understanding of the laws of physics, but some significant technological issues remain to be solved in order to build a fully operational unit. Such a quantum computer would very efficiently break the usual asymmetric encryption and digitial signature algorithms based on number theory (RSA, DSA, Diffie-Hellman, ElGamal, and their elliptic curve variants).

FALCON is based on the theoretical framework of Gentry, Peikert and Vaikuntanathan for lattice-based signature schemes. We instantiate that framework over NTRU lattices, with a trapdoor sampler called "fast Fourier sampling". The underlying hard problem is the short integer solution problem (SIS) over NTRU lattices, for which no efficient solving algorithm is currently known in the general case, even with the help of quantum computers.

Algorithm Highlights

FALCON offers the following features:

- **Security:** a true Gaussian sampler is used internally, which guarantees negligible leakage of information on the secret key up to a practically infinite number of signatures (more than 2^{64}).
- **Compactness:** thanks to the use of NTRU lattices, signatures are substantially shorter than in any lattice-based signature scheme with the same security guarantees, while the public keys are around the same size.
- **Speed:** use of fast Fourier sampling allows for very fast implementations, in the thousands of signatures per second on a common computer; verification is five to ten times faster.
- **Scalability:** operations have cost $O(n \log n)$ for degree n, allowing the use of very long-term security parameters at moderate cost.
- **RAM Economy:** the enhanced key generation algorithm of FALCON uses less than 30 kilobytes of RAM, a hundredfold improvement over previous designs such as NTRUSign. FALCON is compatible with small, memory-constrained embedded devices.

Performance

While resistance to quantum computers is the main drive for the design and development of FALCON, the algorithm may achieve significant adoption only if it is also reasonably efficient in our current world, where quantum computers do not really exist. Using the reference implementation on a common desktop computer (Intel® Core® i5-8259U at 2.3 GHz, TurboBoost disabled), FALCON achieves the following performance:

variant	keygen (ms)	keygen (RAM)	sign/s	verify/s	pub size	sig size
FALCON-512	8.64	14336	5948.1	27933.0	897	666
FALCON-1024	27.45	28672	2913.0	13650.0	1793	1280

Sizes (key generation RAM usage, public key size, signature size) are expressed in bytes. Key generation time is given in milliseconds. Private key size (not listed above) is about three times that of a signature, and it could be theoretically compressed down to a small PRNG seed (say, 32 bytes), if the signer accepts to run the key generation algorithm every time the key must be loaded.

To give a point of comparison, FALCON-512 is roughly equivalent, in classical security terms, to RSA-2048, whose signatures and public keys use 256 bytes each. On the specific system on which these measures were taken, OpenSSL's thoroughly optimized assembly implementation achieves about 1140 signatures per second; thus, FALCON's reference implementation, which is portable and uses no inline assembly on x86 CPUs, is already more than five times faster, and it scales better to larger sizes (for long-term security).

An implementation in Python

You can find the code here

Content

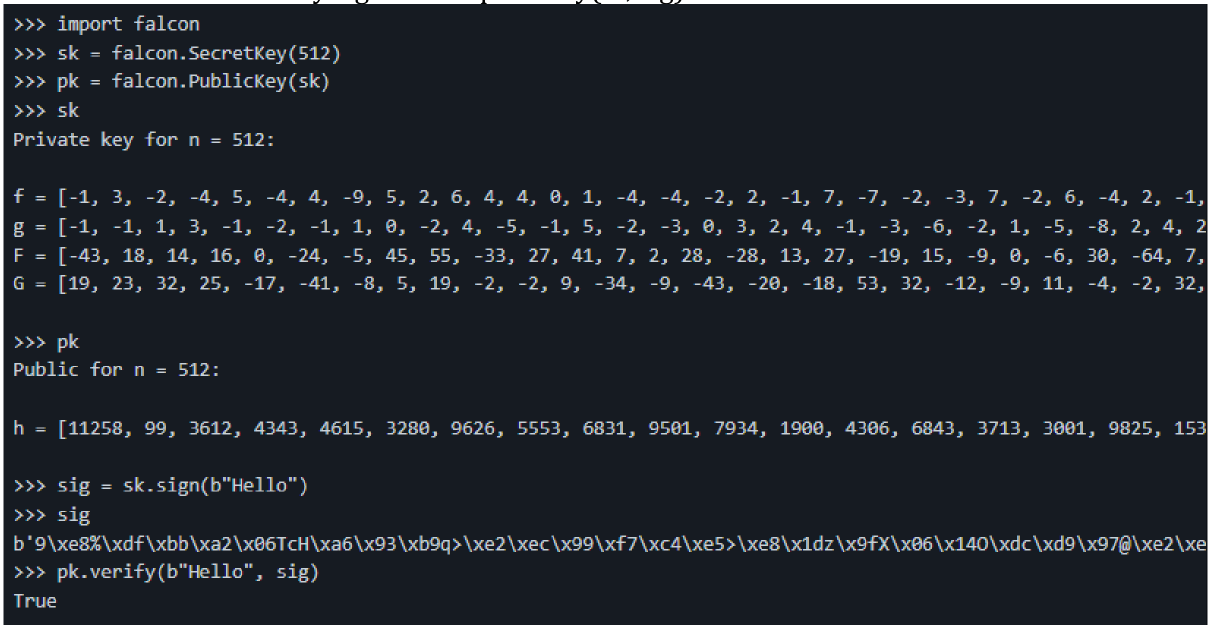
This repository contains the following files (roughly in order of dependency):

1. common.py contains shared functions and constants
2. rng.py implements a ChaCha20-based PRNG, useful for KATs (standalone)
3. samplerz.py implements a Gaussian sampler over the integers (standalone)
4. fft_constants.py contains precomputed constants used in the FFT

5. ntt_constants.py contains precomputed constants used in the NTT
6. fft.py implements the FFT over R[x] / (xn + 1)
7. ntt.py implements the NTT over Zq[x] / (xn + 1)
8. ntrugen.py generate polynomials f,g,F,G in Z[x] / (xn + 1) such that f G - g F = q
9. ffsampling.py implements the fast Fourier sampling algorithm
10. falcon.py implements Falcon
11. test.py implements tests to check that everything is properly implemented

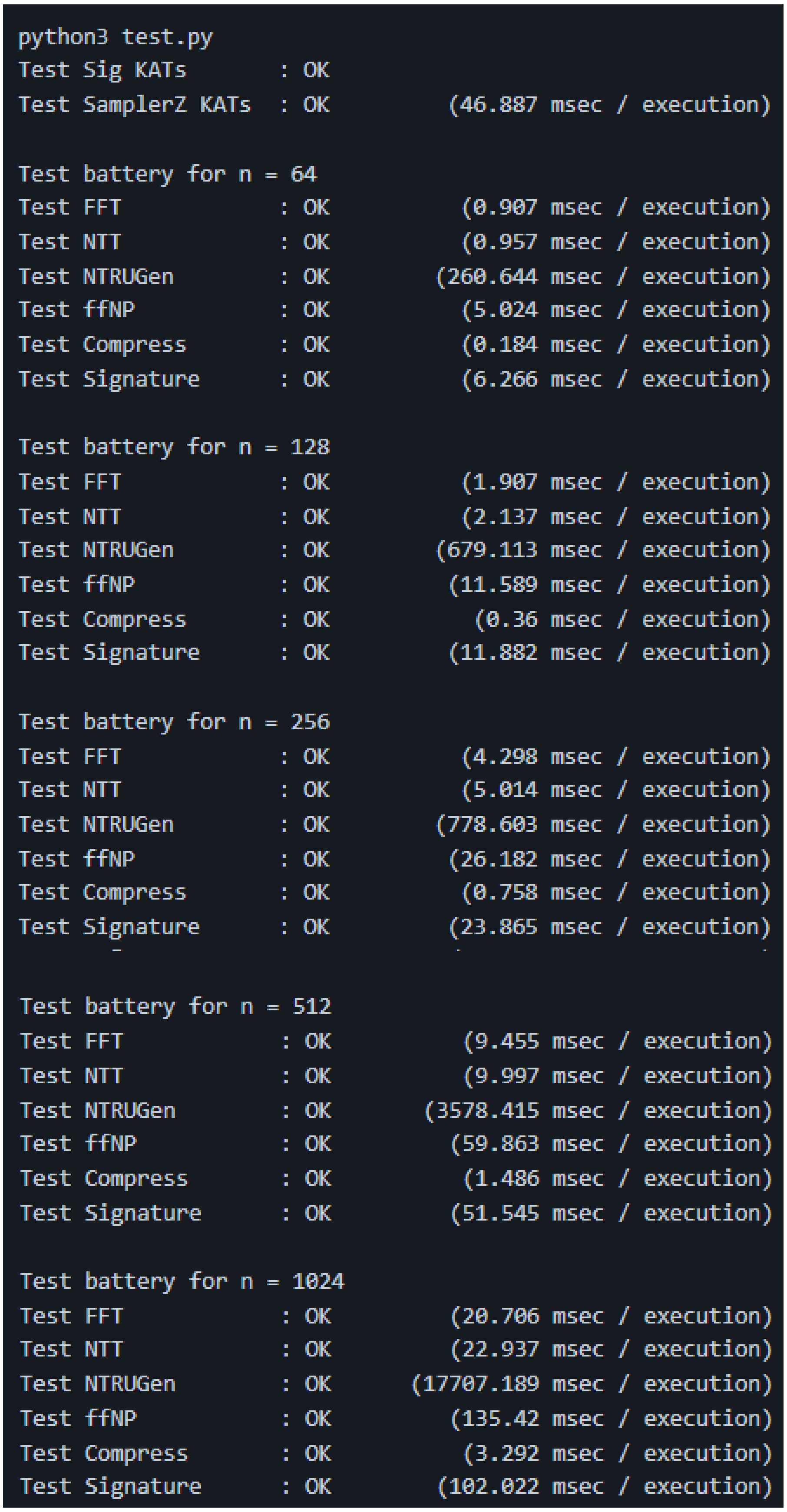
How to use

1. Generate a secret key sk = SecretKey(n)
2. Generate the corresponding public key pk = PublicKey(sk)
3. Now we can sign messages:
    a. To plainly sign a message m: sig = sk.sign(m)
    b. To sign a message m with a pre-chosen 40-byte salt: sig = sk.sign(m, salt) Note that the message MUST be a byte array or byte string.
4. We can also verify signatures: pk.verify(m, sig)



```
>>> import falcon
>>> sk = falcon.SecretKey(512)
>>> pk = falcon.PublicKey(sk)
>>> sk
Private key for n = 512:

f = [-1, 3, -2, -4, 5, -4, 4, -9, 5, 2, 6, 4, 4, 0, 1, -4, -4, -2, 2, -1, 7, -7, -2, -3, 7, -2, 6, -4, 2, -1,
g = [-1, -1, 1, 3, -1, -2, -1, 1, 0, -2, 4, -5, -1, 5, -2, -3, 0, 3, 2, 4, -1, -3, -6, -2, 1, -5, -8, 2, 4, 2
F = [-43, 18, 14, 16, 0, -24, -5, 45, 55, -33, 27, 41, 7, 2, 28, -28, 13, 27, -19, 15, -9, 0, -6, 30, -64, 7,
G = [19, 23, 32, 25, -17, -41, -8, 5, 19, -2, -2, 9, -34, -9, -43, -20, -18, 53, 32, -12, -9, 11, -4, -2, 32,

>>> pk
Public for n = 512:

h = [11258, 99, 3612, 4343, 4615, 3280, 9626, 5553, 6831, 9501, 7934, 1900, 4306, 6843, 3713, 3001, 9825, 153

>>> sig = sk.sign(b"Hello")
>>> sig
b'9\xe8%\xdf\xbb\xa2\x06TcH\xa6\x93\xb9q>\xe2\xec\x99\xf7\xc4\xe5>\xe8\x1dz\x9fX\x06\x14O\xdc\xd9\x97@\xe2\xe
>>> pk.verify(b"Hello", sig)
True
```

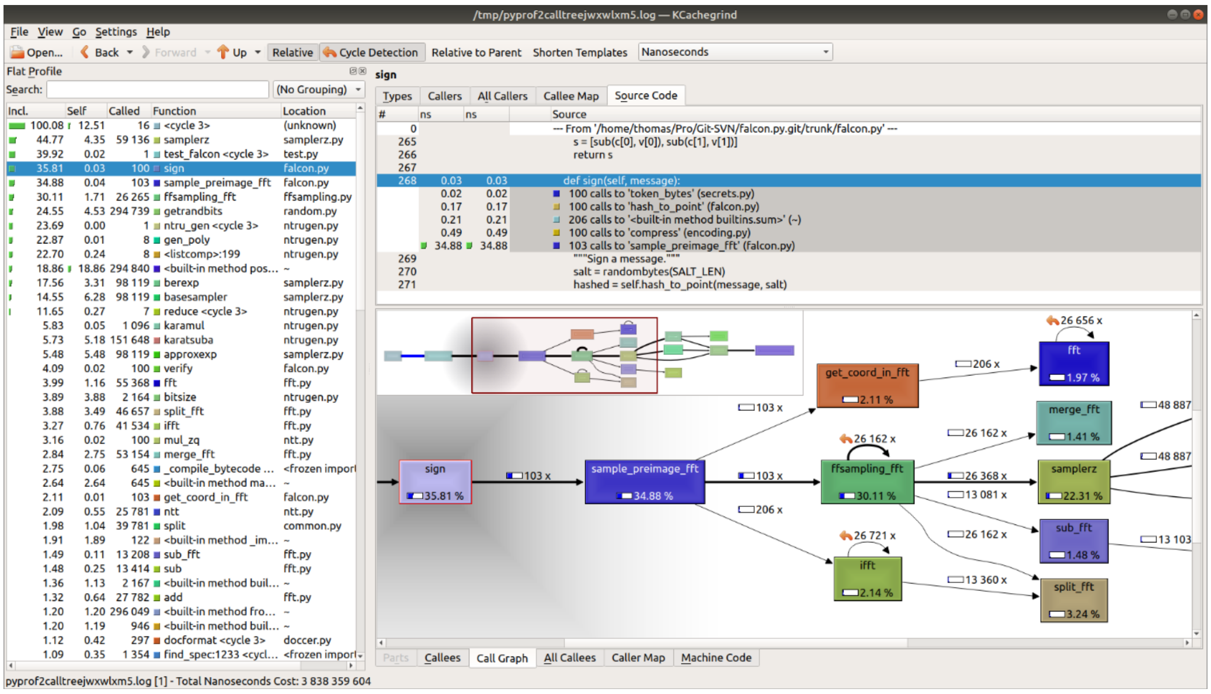
Upon first use, consider running make test to make sure that the code runs properly on your machine. You should obtain (the test battery for n = 1024 may take a few minutes):



```
python3 test.py
Test Sig KATs        : OK
Test SamplerZ KATs   : OK          (46.887 msec / execution)

Test battery for n = 64
Test FFT             : OK           (0.907 msec / execution)
Test NTT             : OK           (0.957 msec / execution)
Test NTRUGen         : OK         (260.644 msec / execution)
Test ffNP            : OK           (5.024 msec / execution)
Test Compress        : OK           (0.184 msec / execution)
Test Signature       : OK           (6.266 msec / execution)

Test battery for n = 128
Test FFT             : OK           (1.907 msec / execution)
Test NTT             : OK           (2.137 msec / execution)
Test NTRUGen         : OK         (679.113 msec / execution)
Test ffNP            : OK          (11.589 msec / execution)
Test Compress        : OK            (0.36 msec / execution)
Test Signature       : OK          (11.882 msec / execution)

Test battery for n = 256
Test FFT             : OK           (4.298 msec / execution)
Test NTT             : OK           (5.014 msec / execution)
Test NTRUGen         : OK         (778.603 msec / execution)
Test ffNP            : OK          (26.182 msec / execution)
Test Compress        : OK           (0.758 msec / execution)
Test Signature       : OK          (23.865 msec / execution)

Test battery for n = 512
Test FFT             : OK           (9.455 msec / execution)
Test NTT             : OK           (9.997 msec / execution)
Test NTRUGen         : OK        (3578.415 msec / execution)
Test ffNP            : OK          (59.863 msec / execution)
Test Compress        : OK           (1.486 msec / execution)
Test Signature       : OK          (51.545 msec / execution)

Test battery for n = 1024
Test FFT             : OK          (20.706 msec / execution)
Test NTT             : OK          (22.937 msec / execution)
Test NTRUGen         : OK       (17707.189 msec / execution)
Test ffNP            : OK          (135.42 msec / execution)
Test Compress        : OK           (3.292 msec / execution)
Test Signature       : OK         (102.022 msec / execution)
```

Profiling

If you type make profile on a Linux machine, you should obtain something along these lines:


/tmp/pyprof2calltreejwxwlxm5.log — KCachegrind
File View Go Settings Help
Open... Back Forward Up Relative Cycle Detection Relative to Parent Shorten Templates Nanoseconds
Flat Profile
Search:
(No Grouping)
sign
Types Callers All Callers Callee Map Source Code
Parts Callees Call Graph All Callees Caller Map Machine Code
fft 1.97 %
26 656 x
206 x
get_coord_in_fft 2.11 %
103 x
merge_fft 1.41 %
48 887
26 162 x
ffsampling_fft 30.11 %
26 162 x
sign 35.81 %
103 x
sample_preimage_fft 34.88 %
103 x
26 368 x
samplerz 22.31 %
48 887
13 081 x
206 x
26 721 x
26 162 x
sub_fft 1.48 %
13 103
ifft 2.14 %
13 360 x
split_fft 3.24 %

```
Incl.    Self    Called  Function                   Location
100.08   12.51       16  <cycle 3>                  (unknown)
 44.77    4.35   59 136  samplerz                   samplerz.py
 39.92    0.02        1  test_falcon <cycle 3>      test.py
 35.81    0.03      100  sign                       falcon.py
 34.88    0.04      103  sample_preimage_fft        falcon.py
 30.11    1.71   26 265  ffsampling_fft             ffsampling.py
 24.55    4.53  294 739  getrandbits                random.py
 23.69    0.00        1  ntru_gen <cycle 3>         ntrugen.py
 22.87    0.01        8  gen_poly                   ntrugen.py
 22.70    0.24        8  <listcomp>:199             ntrugen.py
 18.86   18.86  294 840  <built-in method pos...    ~
 17.56    3.31   98 119  berexp                     samplerz.py
 14.55    6.28   98 119  basesampler                samplerz.py
 11.65    0.27        7  reduce <cycle 3>           ntrugen.py
  5.83    0.05    1 096  karamul                    ntrugen.py
  5.73    5.18  151 648  karatsuba                  ntrugen.py
  5.48    5.48   98 119  approxexp                  samplerz.py
  4.09    0.02      100  verify                     falcon.py
  3.99    1.16   55 368  fft                        fft.py
  3.89    3.88    2 164  bitsize                    ntrugen.py
  3.88    3.49   46 657  split_fft                  fft.py
  3.27    0.76   41 534  ifft                       fft.py
  3.16    0.02      100  mul_zq                     ntt.py
  2.84    2.75   53 154  merge_fft                  fft.py
  2.75    0.06      645  _compile_bytecode ...      <frozen import
  2.64    2.64      645  <built-in method ma...     ~
  2.11    0.01      103  get_coord_in_fft           falcon.py
  2.09    0.55   25 781  ntt                        ntt.py
  1.98    1.04   39 781  split                      common.py
  1.91    1.89      122  <built-in method _im...    ~
  1.49    0.11   13 208  sub_fft                    fft.py
  1.48    0.25   13 414  sub                        fft.py
  1.36    1.13    2 167  <built-in method buil...   ~
  1.32    0.64   27 782  add                        fft.py
  1.20    1.20  296 049  <built-in method fro...    ~
  1.20    1.19      946  <built-in method buil...   ~
  1.12    0.42      297  docformat <cycle 3>        doccer.py
  1.09    0.35    1 354  find_spec:1233 <cycl...    <frozen import
```

```
#      ns      ns      Source
  0                    --- From '/home/thomas/Pro/Git-SVN/falcon.py.git/trunk/falcon.py' ---
265                          s = [sub(c[0], v[0]), sub(c[1], v[1])]
266                          return s
267
268     0.03    0.03       def sign(self, message):
        0.02    0.02       100 calls to 'token_bytes' (secrets.py)
        0.17    0.17       100 calls to 'hash_to_point' (falcon.py)
        0.21    0.21       206 calls to '<built-in method builtins.sum>' (~)
        0.49    0.49       100 calls to 'compress' (encoding.py)
       34.88   34.88       103 calls to 'sample_preimage_fft' (falcon.py)
269                          """Sign a message."""
270                          salt = randombytes(SALT_LEN)
271                          hashed = self.hash_to_point(message, salt)
```

pyprof2calltreejwxwlxm5.log [1] - Total Nanoseconds Cost: 3 838 359 604

Make sure you have pyprof2calltree and kcachegrind installed on your machine, or it will not work.

Resources

Warning (2021-11-01): an issue in the external API was detected by David Lazar and Chris Peikert (of Algorand, Inc.): the functions shake256_init_prng_from_seed() and shake256_init_prng_from_system() were not behaving properly. The NIST API, test vectors and benchmarks used as part of the NIST PQC standardization project are not affected; neither are the implementations of Falcon in the PQClean and pqm4 libraries. The reference source code linked below (both the Zip archive and the browsable version) have been fixed.

- FALCON submission package [zip] (specification, source code, scripts and test vectors)
- FALCON specification [pdf]
- FALCON reference implementation:
    - Source code archive [zip]
    - Browse source code online [html]
- An implementation in Python [html]
- For reference, submission packages for previous rounds: Round 1 [zip] and Round 2 [zip]
- Presentations at the NIST PQC Standardization Conferences: Round 1 [pdf] and Round 2 [pdf]

Related works

- Thomas Pornin and Thomas Prest More Efficient Algorithms for the NTRU Key Generation using the Field Norm [pdf]
- Xingye Lu, Man Ho Au and Zhenfei Zhang Raptor: A Practical Lattice-Based (Linkable) Ring Signature [pdf]

- Thomas Pornin New Efficient, Constant-Time Implementations of Falcon [pdf]
- Pierre-Alain Fouque, Paul Kirchner, Mehdi Tibouchi, Alexandre Wallet and Yang Yu
  Key Recovery from Gram-Schmidt Norm Leakage in Hash-and-Sign Signatures over NTRU Lattices [pdf]
- James Howe, Thomas Prest, Thomas Ricosset and Mélissa Rossi
  Isochronous Gaussian Sampling: From Inception to Implementation [pdf]