Formal Verification of Emulated Floating-Point Arithmetic in Falcon

Vincent Hwang(✉)

Max Planck Institute for Security and Privacy, Bochum, Germany
vincentvbh7@gmail.com

Abstract. We show that there is a discrepancy between the emulated floating-point multiplication in the submission package of the digital signature Falcon and the claimed behavior. In particular, we show that some floating-point products with absolute values the smallest normal positive floating-point number are incorrectly zeroized. However, we show that the discrepancy doesn't affect the complex fast Fourier transform in the signature generation of Falcon by modeling the floating-point addition, subtraction, and multiplication in CryptoLine. We later implement our own floating-point multiplications in Armv7-M assembly and Jasmin and prove their equivalence with our model, demonstrating the possibility of transferring the challenging verification task (verifying highly-optimized assembly) to the presumably more readable code base (Jasmin).

Keywords: Falcon · Floating-point arithmetic · Formal verification · CryptoLine

1 Introduction

Falcon [Pre+20] is one of the recently selected digital signatures for standardization by the National Institute of Standards and Technology. Essentially the signature is sampled with a probability approximated by floating-point numbers. Since floating-point arithmetic is not always constant-time, [Por19] implemented a series of constant-time floating-point arithmetic with software emulation. We show that

- the emulated floating-point multiplication does not honor its behavior claimed by [Por19];
- the discrepancy does not affect the complex fast Fourier transform in the signature generation of Falcon; and
- how to prove the equivalence between emulated floating-point addition/subtraction/multiplication implementations.

Our source code is publicly available at
https://github.com/vincentvbh/Float_formal.

K. Minematsu and M. Mimura (Eds.): IWSEC 2024, LNCS 14977, pp. 125–141, 2024.
https://doi.org/10.1007/978-981-97-7737-2_7

2 Preliminaries

2.1 Falcon

Falcon is a lattice-based hash-and-sign digital signature based on fast Fourier sampling over an NTRU lattice [Pre+20]. The NTRU lattice is determined by four integer polynomials f, g, F, G satisfying

$$fG - gF = q \bmod (x^n + 1)$$

where $q = 12289$ and $n = 512, 1024$. The lattice is generated by the basis $\mathbf{B} := \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$.

For the key generation, the four polynomials f, g, F, G form the secret key $\mathtt{sk}$ and hence must have small coefficients, and the public key $\mathtt{pk}$ is the polynomial $h := gf^{-1} \bmod (x^n + 1, q)$. See Algorithm 1 for an illustration.

For the signature generation, we generate a nonce r and hash it with the message m. We then start sampling two small polynomials s_1 and s_2 satisfying $s_1 + s_2 h = c \bmod (x^n + 1, q)$ where c is the hash. The signature is defined as (r, s_2). Falcon adopts the so-called fast Fourier sampling based on a randomized variant of fast Fourier nearest plane [DP16, Pre+20]. The idea essentially goes as follows: We compute $\hat{\mathbf{B}} = \mathtt{FFT}(\mathbf{B})$ and $\hat{c} = \mathtt{FFT}(c)$ with complex fast Fourier transform, compute $\mathbf{t} = \left(-\frac{\hat{c}\hat{F}}{q}, \frac{\hat{c}\hat{f}}{q}\right)$, construct the corresponding Falcon tree $\mathbf{T}$ from the LDL decomposition of $\hat{\mathbf{B}}\hat{\mathbf{B}}^*$, and apply fast Fourier nearest plane where the nearest plane part at the leaf level is replaced by a discrete Gaussian sampling with secret center constructed serially from $\mathbf{t}$ and prior samples and secret deviation constructed from $\mathbf{T}$. We refer to Algorithm 2 for an overview of the signature generation and [Pre+20, Algorithm 11] for a more detailed explanation of the fast Fourier sampling.

For the signature verification, we compute $s_1 = c - s_2 h \bmod (x^n + 1, q)$ and accept the signature if $||(s_1, s_2)||^2$ is small enough (reject otherwise). See Algorithm 3 for an illustration.

Algorithm 1: Falcon key generation from the reference implementation.

```
Outputs: a public key pk and a secret key sk
1: $(f, g) = \mathtt{mkgauss()}$        ▷ Generate $f, g$ from a discrete gaussian distribution.
2: $(F, G) = \mathtt{solve\_NTRU}(f, g, x^n + 1, q)$        ▷ $fG - gF = q \bmod (x^n + 1)$
3: $h = gf^{-1} \bmod (x^n + 1, q)$
4: $\mathtt{sk} = (f, g, F, G)$
5: $\mathtt{pk} = h$
6: return pk, sk
```

Algorithm 2: Falcon signature generation from the reference implementation.

```
Inputs: A message $m$ and a secret key sk.
Outputs: A signature sig.
1: $r \leftarrow \{0,1\}^{320}$ uniformly                ▷ Salt.
2: $c = \texttt{HashToPoint}\,(r||m)$
3: $\hat{c} = \texttt{FFT}(c)$
4: $\mathbf{B} = \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$
5: $\hat{\mathbf{B}} = \begin{pmatrix} \hat{g} & -\hat{f} \\ \hat{G} & -\hat{F} \end{pmatrix} = \texttt{FFT}(\mathbf{B})$
6: $\mathbf{T} = \texttt{ffLDL}^*\left(\hat{\mathbf{B}}\hat{\mathbf{B}}^*\right)$
7: $\mathbf{T} = \texttt{Normalize}\,(\mathbf{T})$
8: $\mathbf{t} = \left(\frac{-\hat{c}\hat{F}}{q}, \frac{\hat{c}\hat{f}}{q}\right)$                ▷ $\mathbf{t} = (\hat{c}, 0)\,\hat{\mathbf{B}}^{-1}$
9: do
10:     do
11:         $\mathbf{z} = \texttt{ffSampling}\,(\mathbf{t}, \mathbf{T})$
12:         $\mathbf{s} = (\mathbf{t} - \mathbf{z})\,\hat{\mathbf{B}}$
13:     while $||\mathbf{s}||^2 > \lfloor \beta^2 \rfloor$
14:     $(s_1, s_2) = \texttt{iFFT}\,(\mathbf{s})$
15:     $s = \texttt{Compress}\,(s_2, 8 \cdot \texttt{sbytelen} - 328)$
16: while $s == \perp$
17: $\texttt{sig} = (r, s)$
18: return sig
```

2.2 Fast Fourier Transform

Fast Fourier transform (FFT) is a popular approach in signal processing, polynomial multiplication, and sampling. For a power of two n and the primitive $2n$-th root of unity $\omega_{2n} \in \mathbb{C}$, the negacyclic Cooley–Tukey FFT transforms the polynomial ring $\mathbb{C}[x]/\langle x^n + 1\rangle$ into $\prod_{i=0,\ldots,n-1} \mathbb{C}[x]/\langle x - \omega_{2n}^{1+2i}\rangle$ in $O(n \log_2 n)$ operations in $\mathbb{C}$ up to the bitreversal permutation. In Falcon, since the input coefficients are integers, [Por19] implemented an optimized variant of the complex Cooley–Tukey FFT with $\mathbb{C} = \mathbb{R}[z]/\langle z^2 + 1\rangle$. They also approximated the real number arithmetic by floating-point arithmetic in the signature generation.

2.3 Emulated Floating-Point Arithmetic

In Falcon, the real arithmetic in the signature generation is implemented as floating-point arithmetic. We briefly review the IEEE 754 double-precision floating-point specification.

A double-precision floating-point number is a 64-bit element consists of three parts (most significant bits first): a 1-bit `s` for the sign, an 11-bit `e` for the biased exponent, and a 52-bit `m` for the mantissa. We denote a floating-point number as `s|e|m` with the sign `s`, the biased exponent `e`, and the mantissa `m`. When the biased exponent satisfies $0 < \texttt{e} < 2047$, the floating-point number corresponds to the following real number:

Algorithm 3: Falcon signature verification.

```
Inputs: a message $m$, a signature sig, and a public key $\texttt{pk} = h$
1: $c = \texttt{HashToPoint}\,(r||m)$
2: $s_2 = \texttt{Decompress}\,(s, 8 \cdot \texttt{sbytelen} - 328)$
3: if $s_2 == \perp$ then
4:     reject
5: $s_1 = c - s_2 h$
6: if $||\,(s_1, s_2)\,||^2 > \lfloor \beta^2 \rfloor$ then
7:     reject
8: accept
```

$$(-1)^{\mathtt{s}}\, 2^{\mathtt{e}-1075} \left(2^{52} + \mathtt{m}\right).$$

We call such a floating-point number normal. In addition to the normal values, we also have the following special values:

- $\mathtt{e} = 0, \mathtt{m} = 0$: This corresponds to a zero value. Notice that there are two zeros ± 0 distinguished by the sign $\mathtt{s}$.
- $\mathtt{e} = 0, \mathtt{m} \neq 0$: This corresponds to the denormalized number $(-1)^{\mathtt{s}}\, 2^{\mathtt{e}-1074}\mathtt{m}$.
- $\mathtt{e} = 2047, \mathtt{m} = 0$: This corresponds to an infinity. Notice that there are also two infinities $\pm\infty$ distinguished by the sign $\mathtt{s}$.
- $\mathtt{e} = 2047, \mathtt{m} \neq 0$: This corresponds to a NaN (not-a-number) value.

In IEEE 754, "rounding to the nearest even" is adopted by default for rounding the real number result to a floating-point number. In Falcon, the authors claimed that infinites, NaNs, and denormalized numbers are not used and implemented a set of functions emulating the elementary floating-point arithmetic where the results are, according to their claim, correctly rounded for all normal values and zeros with "rounding to the nearest even" rule [Por19, Section 3.3]. We show that the latter doesn't hold, but it doesn't impact the complex fast Fourier transform in the signature generation of Falcon.

2.4 CryptoLine

CryptoLine is a domain specific language for modeling straightline cryptographic programs. It was introduced by [TWY17,PTWY18] for verifying elliptic-curve arithmetic with assembly programs optimized "in the wild." In other words, assembly optimized programs were first delivered by experts in assembly programming without considerations on verification, and verification effort was later devoted to verifying the resulting programs. CryptoLine was extended by [LSTWY19] for verifying elliptic-curve C implementations, and by [FLSTWY19] for signed arithmetic. Recently, [Hwa+22] extended CryptoLine with compositional reasoning for verifying large dimensional number-theoretic transforms, and [LLSTWY23] extended CryptoLine with logical equivalence checking for the stream cipher ChaCha20 [Ber08] and the cryptographic hash functions SHA-2 and SHA-3.

In CryptoLine, there are various instructions implementing basic arithmetic, including signed/unsigned addition/subtraction/multiplication, logical/arithmetic shift, bit-wise or/exclusive-or/and/not, bit-field splitting/concatenation, signed/unsigned extension, and conditional move. These instructions effectively capture the commonly used assembly instructions in cryptographic programs. We translate the target assembly programs into strings of CryptoLine instructions, and argue the properties of the strings of CryptoLine instructions.

There are two classes of predicates in CryptoLine for modeling the properties of strings of CryptoLine instructions: the algebraic predicates and the range predicates. An algebraic predicate is a conjunction of equations and modular equations, and a range predicate is a boolean formula with comparisons, equations, and modular equations. We have the assertion `assert` and the assumption `assume` annotations for imposing properties on the predicates. For an algebraic predicate `P` and a range predicate `Q`, `assert P && Q` asks the backend to verify `P` with the associated computer algebra system and `Q` with the associated SMT solver, and `assume P && Q` adds `P` and `Q` to the corresponding backend tools.

Assertions are used alone for verifying properties, and assumptions are commonly used in conjunction with assertions for transferring predicates between the backend tools. For example, we first verify an algebraic predicate `P` by imposing `assert P && true` and pass it to the SMT solver by imposing `assume true && P`.

For verifying a program as a whole, we specify pre-conditions on the variables, insert the string of CryptoLine instructions translated from the target program, annotate it with assertions and assumptions at proper locations, and finally specify the post-conditions. The most difficult part is the insertions of annotations, which, if ignored, results in non-responseness of the verification process in our context.

2.5 Jasmin

Jasmin is a programming language serving as a vehicle correlating assembly programs and their high-level abstractions. It was introduced by [Alm+17] for verifying the memory safety and constant-timeness of elliptic-curve arithmetic implementations. Jasmin was extended by [Alm+19] for verifying implementation correctness and the security of SHA3 implementations with EasyCrypt, and [Alm+20] revisited the compiler, memory model, and EasyCrypt embedding for verifying the ChaCha20 stream cipher, the Poly1305 message-authentication code [Ber05], and the Gimli permutation [Ber+17]. Recently, [Alm+23] extended Jasmin with function calls, pointers to the stack memory, and the system call `randombytes`, and proved the implementation correctness of the key encapsulation mechanism Kyber recently selected by the National Institute of Standards and Technology as one of the to-be-standardized algorithms for post-quantum cryptography.

Programmers write Jasmin programs with similar control of the computational flow as in assembly, and compile the programs into assembly programs with the certified compiler `jasminc`. For verification purpose, we extract the

Jasmin programs to EasyCrypt according to the Jasmin model in EasyCrypt, and verify the desired properties with EasyCrypt. Compared to CryptoLine, verification in EasyCrypt requires much more effort by explicitly applying various lemmas instead of simply imposing properties in a declarative fashion in CryptoLine, but one can argue more properties in Easycrypt, for example, the indifferentiability of SHA3 from random oracle as shown in [Alm+19].

3 Incorrect Zeroization

3.1 The Problem of Floating-Point Multiplication

We point out an incorrect zeroization in the emulated floating-point multiplications in Falcon. We illustrate the issue in the C reference implementation, and our finding also applies to the Armv7-M assembly optimized implementation.

We briefly review the C reference implementation of the emulated floating-point multiplication in the submission package of Falcon as follows:

1. The inputs are two 64-bit integers with each representing a double-precision floating-point number.
2. Extract the mantissas and add them with 2^{52} as if the floating-point inputs are non-zero.
3. Compute the product of mantissas with radix-25 arithmetic.
4. Normalize the product to a 55-bit value.
5. Compute the exponent field as the sum of input exponent fields with a corrective subtraction.
6. Compute the sign field as the exclusive-or of the input sign fields.
7. Zeroize the product if any of the input exponent fields is zero.
8. Zeroize the product if the resulting exponent is too small.
9. Zeroize the exponent field if the product is zero.
10. Assemble the sign field, exponent field, and the upper 53 bits of the 55-bit product.
11. Increment the resulting floating-point as an unsigned 64-bit integer if the 55-bit product should be rounded.

The issue is that the zeroization due to the smallness of the exponent field should be the last operation since the increment from rounding may results in an exponent field that is slightly above the zeroization threshold. We refer to Algorithm 4 for a more detailed illustration where the line in red (blue) corresponds to the line in red(blue) of the above.

3.2 Extracting Witnesses

We show how to find inputs triggering the incorrect zeroization. For a floating-point number with exponent field e and mantissa m, we find that if $1 \leq e \leq 1022$, $1 \leq m \leq 2^{52} - 2$, and $\left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor (2^{52} + m) \geq 2^{105} - 2^{51}$, then a floating-point with exponent field $1023 - e$ and mantissa $\left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor$ leads to incorrect zeroization in

Algorithm 4: Emulated C implementation (with some high-level syntax for the irrelevant parts for readability) of floating-point multiplication in Falcon.

```
1: uint64_t xu, yu, zu, z;
2: uint32_t z0, z1, sticky, round;
3: int ex, ey, e, d, s;
4: xu = $2^{52}$ | x & ($2^{52}-1$);
5: yu = $2^{52}$ | y & ($2^{52}-1$);
6: z0 + z1 * $2^{25}$ + zu * $2^{50}$ = xu * yu;
7: sticky = ((z0 | z1) + $2^{25}-1$ ) ≫ 25;  ▷ sticky = 0 if z0 = z1 = 0, otherwise 1.
8: zu = zu | (uint64_t)sticky;
9: ex = (int)((x ≫ 52) & ($2^{11}-1$));
10: ey = (int)((y ≫ 52) & ($2^{11}-1$));
11: e = ex + ey - 2100;
12: (zu, e) = normalize(zu, e, 55, sticky);
13: s = (int)((x ^ y) ≫ 63);
14: d = ((ex + $2^{11}-1$) & (ey + $2^{11}-1$)) ≫ 11;  ▷ d = 0 if ex = 0 or ey = 0, otherwise 1.
15: zu = zu & (uint64_t)-d;  ▷ zu = 0 if d = 0, otherwise unchanged.
16: m = zu & ( ((uint32_t)(e + 1076) ≫ 31) - 1);  ▷ m = 0 if e < −1076, otherwise unchanged.
17: e = e + 1076;
18: e = e & -((int)(uint32_t)(m ≫ 54) );  ▷ e = 0 if $m = 0$, otherwise unchanged.
19: z = ( ((uint64_t)s ≪ 63) | (m ≫ 2) ) + ( (uint64_t)(uint32_t) e ) ≪ 52;
20: round = (0xc8 ≫ ((uint32_t)m & 7) ) & 1;  ▷ round = 1 if m & 7 = 3, 6, 7, otherwise 0.
21: z = z + (uint64_t)round;
22: return (fpr)z;  ▷ fpr is defined as uint64_t.
```

Algorithm 4 where the correct result is a floating-point number with absolute value the smallest normal positive floating-point number.

Recall that the issue of Algorithm 4 is that the product is zeroized due to the smallness of the sum of exponents prior to the rounding at the end. We seek for conditions triggering both lines (if-conditions are taken) while the floating-point product is large enough after the rounding.

For simplicity, we first assume that the product of mantissas is an unsigned 105-bit integer (we will explain how this condition is satisfied shortly) so Line 12 changes nothing. We then choose e as the largest value, −1077, triggering Line 16 in Algorithm 4:

$$\texttt{m} = \texttt{zu \& (((uint32\_t)(e + 1076) ≫ 31) - 1)}.$$

This leads to the exponent fields ex $= e$ and ey $= 1023 - e$ after tracing the code (cf. Line 11). It remains to choose mantissas with a 105-bit product triggering Line 20:

$$\texttt{round} = \texttt{(0xc8 ≫ ((uint32\_t)m \& 7)) \& 1}.$$

This leads to the mantissas xu $= 2^{52}+m$ and yu $= \left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor$ with an m satisfying

- $1 \leq m \leq 2^{52} - 2$, and
- $\left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor \left(2^{52}+m\right) \geq 2^{105} - 2^{51}$.

This implies that we have $2^{55} - 2$ or $2^{55} - 1$ after normalizing to a 55-bit value (cf. Line 12), whose rounded value is 2^{55} if we round it prior to the zeroization in Line 16. Since the correct mantissa is 2^{55}, we have to increment the exponent by 1, removing the need of zeroization from the smallness of the exponent.

Listing 1.2 is our program testing if we can find a floating-point number b from the input floating-point number a whose floating-point product leads to an incorrect zeroization in Algorithm 4, and Listing 1.1 is an auxiliary function.

Listing 1.1: Our C program testing if the input is small enough. We return 1 if x is small enough, and 0 otherwise.

```c
int test_smallness(fpr x){

    fpr e = (x >> 52) & 0x7ff;
    fpr m = x & 0xfffffffffffff;

    if( (1 <= e) && (e <= 1022) )
        if( (1 <= m) && (m <= 0xffffffffffffe) )
            return 1;

    return 0;

}
```

Listing 1.2: Our C program testing if there is an input leading to incorrect zeroization. If we find a floating-point value such that its floating-point product with a leads to incorrect zeroization, the floating-point value is stored in *b and 1 is returned. Otherwise, -1 is returned.

```c
int retrieve_zeroization(fpr *b, fpr a){

    uint64_t t;

    __uint128_t a128, b128, t128;

    if(test_smallness(a) == 0)
        return -1;

    a128 = (1ULL << 52) + (a & 0xfffffffffffff);
    t128 = 1; t128 <<= 105;
    b128 = t128 / a128;

    if( a128 * b128 + (1ULL << 51) < t128)
```

```
        return -1;

    t = ( 1023 - ((a >> 52) & 0x7ff) ) << 52;
    t |= b128 - (1ULL << 52);
    *b = t;

    return 1;

}
```

3.3 An Example in Falcon

In Falcon, we need to approximate the real number $\frac{1}{\sqrt{2}}$ for representing the complex number $e^{\frac{\pi i}{4}} = \frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}$. The real number $\frac{1}{\sqrt{2}}$ is approximated by the floating-point number $\mathtt{s}|\mathtt{e}|\mathtt{m} = \mathtt{0}|\mathtt{1022}|\mathtt{1865452045155277}$. Since $1 \leq \mathtt{e} \leq 1022$, $1 \leq \mathtt{m} \leq 2^{52} - 2$, and $\left\lfloor \frac{2^{105}}{2^{52}+\mathtt{m}} \right\rfloor \left(2^{52} + \mathtt{m}\right) = 6369051672525772 \left(2^{52} + \mathtt{m}\right) \geq 2^{105} - 2^{51}$, we know that if the other operand is the floating-point number $\mathtt{0}|\mathtt{1}|\mathtt{6369051672525772}$, the result is incorrectly zeroized. One can pass the pair $\left(1022 \cdot 2^{52} + 1865452045155277, 2^{52} + 6369051672525772\right)$ as arguments of the emulated floating-point multiplication in Falcon and compare the result with the native floating-point multiplication to see the difference.

4 Is it Relevant to Falcon?

In previous section, we demonstrate that the emulated floating-point multiplication doesn't honor its claim where some non-zero floating-point numbers are zeroized. An immediate question is its impact to Falcon implementations. Among the functions in Falcon, we are interested in the complex FFT in the signature generation where the inputs are polynomials with integer coefficients in $\left[-2^{15}, 2^{15}\right)$. After going through the tests for all the floating-point constants in the complex FFT, we find that 692 out of 2048 floating-point constants admit floating-point operands leading to incorrect zeroization. Nevertheless, we model the floating-point addition, subtraction, and multiplication in CryptoLine, and show that all non-zero intermediate floating-point numbers have absolute values lie in

$$\left[2^{-476}, 2^{27}(2^{52} + 605182448294568)\right],$$

far away from triggering incorrect zeroizations.

4.1 Modeling with CryptoLine Instructions

We first model our own strings of CryptoLine instructions and start annotating CryptoLine programs with assertions and assumptions to transfer predicates between backend tools. The main difficulties are as follows:

- When to declare statements that should be proved by the backend proof systems?
- Which statements should be transferred between proof systems at a given point?

We do not know of any systematic approaches resolving the two difficulties. Nevertheless, we find the following constructions of intermediate symbols and annotations sufficient for verifying the range:

1. Construct the 128-bit product r of mantissas with the long multiplication.
2. Split the input into radix-25 representation with bitfield arithmetic, verify the correctness of the spliting with the SMT solver, and add the corresponding algebraic identities to the computer algebra system.
3. Compute the multi-limb product, verify its algebraic correctness with r in the computer algebra system, and add the corresponding boolean identities to the SMT solver.
4. Verify the remaining operations (zeroization, rounding, assembling) entirely with the SMT solver.

If we remove Steps 2. and 3., the SMT solver doesn't return a result (it doesn't find an instance disproving the properties, but it doesn't finish verifying over all the possible inputs).

4.2 Range-Checking

We develop our own range arithmetic in C++ computing the pre- and post-conditions to be verified. Once the pre- and post-conditions are computed for all the possible floating-point additions/subtractions/multiplications, we verify the correctness with CryptoLine. Typically, range-checking of floating-point arithmetic focus on upper-bounding the floating-point errors[1]. However, we need to derive non-trivial lower bounds of floating-point numbers for proving the non-smallness of the absolute values of non-zero floating-point numbers.

For two non-negative floating-point numbers $a.l \leq a.u$, we represent the subset $\{0\} \cup [a.l, a.u] \cup [-a.u, -a.l]$ as a structure with lower bound $a.l$ and upper bound $a.u$. Since the definition is symmetric for the positive and negative sides, we only store the positive bounds, and update the positive bounds throughput the entire computation. The zero values are included implicitly and we do not store its existence (it always exists in all the ranges). The range arithmetic of floating-point multiplication is straightforward as shown in Algorithm 5. For the floating-point addition/subtraction with the ranges a and b, we distinguish between two cases:

[1] For example, Frama-C [CKKPSY12] only shows that the floating-point number is upper-bounded by a floating-point number and lower-bounded by 0, which is useless for proving the non-smallness of the absolute values of non-zero floating-point numbers.

1. Case $a \cap b = \{0\}$: The upper bound is computed as the sum of upper bounds, and the lower bound is defined as the minimum of the absolute values of the differences between an upper bound and a lower bound from different ranges. In other words, the lower bound is defined as $\min(|a.u - b.l|, |b.u - a.l|)$.
2. Case $a \cap b = t \neq \{0\}$: The upper bound is also computed as the sum of upper bounds, and the lower bound is defined as the floating-point value with mantissa 0 and exponent field 52 smaller than the exponent field of $t.l$, since the smallest value occurs when subtracting two values with the real value difference $2^{\mathsf{e}-1075}$ where e is the smallest exponent field of the two and choosing e as the exponent field of $t.l$ results in a worse case analysis. Since we have to shift the leading bit of mantissa to the 52-th bit position, the exponent field is subtracted by 52 and the mantissa becomes 2^{52}. By the definition of floating-point numbers, the leading bit of mantissa is stored implicity. This is why we set the mantissa to 0 in the floating-point number representation.

Algorithm 6 is an illustration of the range arithmetic of floating-point addition/subtraction. After replacing all the floating-point arithmetic with the range arithmetic in the FFT of Falcon, we transform all the input-output tuples into pre- and post-conditions for the corresponding CryptoLine model. We then run CryptoLine to verify the conditions. Our CryptoLine verification shows that

- All the range arithmetic are correct within our modeling of floating-point addition, subtraction, and multiplication.
- All non-zero intermediate floating-point numbers have absolute values lie in

$$\left[2^{-476}, 2^{27}(2^{52} + 605182448294568)\right]$$

  when the input coefficients of FFT are integers in $\left[-2^{15}, 2^{15}\right)$.

Table 1 summarizes the verification time of the range conditions of floating-point additions and multiplications in Falcon's size-1024 complex FFT.

Algorithm 5: Range arithmetic of floating-point multiplication.

```
Inputs: $a = (a.l, a.u), b = (b.l, b.u)$
Output: $c = (c.l, c.u)$
1: $c.l = a.l \cdot b.l$
2: $c.u = a.u \cdot b.u$
3: return $c$
```

Algorithm 6: Range arithmetic of floating-point addition/subtraction.

```
Inputs: $a = (a.l, a.u), b = (b.l, b.u)$
Output: $c = (c.l, c.u)$
 1: $t = a \cap b$.
 2: if $t = \{0\}$ then
 3:     $(d_0, d_1) = (|a.u - b.l|, |b.u - a.l|)$
 4:     $c.l = \min(d_0, d_1)$
 5:     $c.u = a.u + b.u$
 6:     return $c$
 7: $c.u = a.u + b.u$
 8: $\texttt{s|e|m} = t.l$
 9: $c.l = \texttt{s|(e} - 52\texttt{)|0}$
10: return $c$
```

Table 1. Verification time of range conditions for a size-1024 complex FFT with $\mathbb{C} \cong \mathbb{R}[z]/\langle z^2 + 1\rangle$ and input polynomials drawn from $\mathbb{Z} \cap [-2^{15}, 2^{15})$. Floating-point subtractions are regarded as floating-point additions in our interval arithmetic. FP stands for "floating-point."

Operation	Number of instances	Verification time (avr./total in seconds)
FP addition	767	0.297 886/228.478 732
FP multiplication	511	2.589 009/1 322.983 371

5 Equivalence Proofs

In this section, we briefly describe our implementations of floating-point multiplication and their equivalence proofs.

5.1 Our Implementations and The Claimed Behavior

Since there is a discrepancy between the emulated floating-point multiplications in Falcon and the claimed behavior, we implement our own assembly implementation honoring the following rules:

- It rounds the values correctly by experiment.
- Its output range is always zeros or normal floating-point values by formal verification. If the real number product is too small in absolute value, it returns a zero. If the real number product is too large in absolute value, the largest possible normal value is returned when the result is positive (smallest possible normal value is returned in the negative case).

We start with the assembly implementation in Falcon, which is much more optimized compared to the C reference implementation, and implement the above rules. This ensures that the output range is always a zero or a normal floating-point value when the inputs are zeros or normal floating-point values.

Comparisons to [Por19]. In the emulated floating-point multiplications in Falcon by [Por19], since the program does not handle infinities, one has to verify the correctness within a certain input range avoiding infinity outputs. The former forbids us to argue the correctness of the full range of zeros and normal floating-point values.

In addition, we also implement an emulated floating-point multiplication in Jasmin essentially following the more readable (but slower) C reference implementation. In the follow-up section, we explain how to verify the equivalences of emulated floating-point multiplication implementations.

5.2 Equivalence Proofs in CryptoLine

We start with our CryptoLine model used for range-checking and add more annotations. Essentially, the majority of the effort is still about verifying the multi-limb arithmetic and transferring its correctness to the SMT solver. In principle, whenever we issue a multiplication, we prove its correctness in the computer algebra system, and add the corresponding boolean identities to the SMT solver. We apply the idea to proving the equivalence of our CryptoLine model and our assembly implementation, and the equivalence of our CryptoLine model and our Jasmin implementation. See Table 2 for an overview of verification time of the equivalences. Since equivalence is transitive, we have an equivalence between our assembly optimized implementation and our Jasmin implementation where the former is more optimized and the latter is more readable.

Table 2. Verification time of equivalence proofs between Armv7-M implementations and our CryptoLine model.

Programming langauge	Verification time (in seconds)
Floating-point addition	
Jasmin	53.946 560
Assembly	59.863 976
Floating-point multiplication	
Jasmin	57.108 668
Assembly	5.333 913

6 Discussions

6.1 How the Discrepancy Was Found?

The core of this paper is about modeling floating-point addition, subtraction, and multiplication with the domain specific language CryptoLine, and its application in proving the lower bound and upper bound of non-zero intermediate floating-point numbers and the equivalences between implementations via software emulation. The whole paper is written in a way with concise logical reasoning so readers can follow more easily. However, the true story of the discovery is more disorganized than the story told in the paper.

The true story is that, we first wrote a model in CryptoLine and proved its equivalence with the emulated floating-point multiplication by [Por19]. With a much more readable model at hand, we were confounded by its correctness since it was inconsistent with our understanding of floating-point arithmetic. Our careful examinations eventually led to the C program extracting witnesses with incorrect zeroization, in the sense that the results were different from the native floating-point multiplication on our laptop and the emulated floating-point multiplication by the Arm's toolchain for Cortex-M4. After contacting the author of [Por19], we knew that experimentally, there were no such floating-point numbers but there was no formal proof. We later fixed our model, simplified it for range-checking, and verified the absence of non-zero floating-point numbers with absolute values the smallest normal positive floating-point number throughout the complex FFT in the signature generation of Falcon. The model was finally used for verifying the equivalence of implementations. We hope the true story will give more insights on how to use the tools.

6.2 The Validity of This Paper After Recent Uses of Fixed-Point Arithmetic

Recently, a fixed-point implementation for the complex FFT in the key generation was proposed by [Por23]. An immediate question is the validity of our findings in the emulated floating-point arithmetic. We would like to stress that, the roles of the complex FFTs are quite different in key generation and signature generation.

Key Generation. We review the uses of complex FFT in the key generation of Falcon as follows. We first generate short integer polynomials f and g, and solve for integer polynomials F and G satisfying

$$fG - gF = q \bmod (x^n + 1) \, .$$

Since the coefficients of F and G could be too large for efficient computation for the follow-up computation, we need to reduce the bit-size of the pair (F, G) with respect to the pair (f, g). This can be achieved by the Babai's reduction: we compute $k = \left\lfloor \frac{Ff^* + Gg^*}{ff^* + gg^*} \right\rceil$ and subtract (kf, kg) from (F, G) where $f^* := f_0 - \sum_{i=1}^{n-1} f_i x^{n-i}$ is the adjoint of $f = \sum_{i=0}^{n-1} f_i x^i$. Obviously, if $fG - gF = q \bmod (x^n + 1)$, then $f(G - kg) - g(F - kf) = fG - gF = q \bmod (x^n + 1)$ and $(F - kf, G - kg)$ is a valid solution for the NTRU equation. For the quotient $\frac{Ff^* + Gg^*}{ff^* + gg^*}$ in $\mathbb{Q}[x]/\langle x^n + 1\rangle$, we instead compute them with the aid of complex FFT in $\mathbb{C}[x]/\langle x^n + 1\rangle$. In [Por23], the author implemented the complex FFT with scaled 64-bit fixed-point arithmetic and reduced the pair (F, G) several times instead of reducing it once with high-precision complex FFT.

Signature Generation. In the signature generation, the role of the complex FFT is quite different. Essentially, the sampler in Falcon converts the sampling task over the NTRU lattice into several one-dimensional sampling task and the complex FFT is involed in this conversion. If one wants to replace the floating-point FFT with scaled fixed-point arithmetic, one has to thoroughly revise the range analysis of the scaling, potentially use a much higher precision, and revise the security analysis from the implementational perspective. We have not seen effort from the community deploying the scaled fixed-point arithmetic and analyzing the accompanied security impact.

6.3 Possible Future Extensions

We briefly outline several possible future extensions of this paper.

Verifying Additional Constant-Time Emulations of Floating-Point Arithmetic. This paper demonstrates the formal verification of the software emulation of floating-point addition, subtraction, and multiplication with respect to our CryptoLine model. Our approach extends to several interesting floating-point arithmetic, including negation, halving and fused multiply-add/sub. Our approach also applies to other rounding rules. As for the floating-point division, it will be interesting to explore the formal verification of the bit-by-bit division by [Por19].

Applications to `ffLDL`* and `ffSampling`. In this paper, we verify the range of the complex FFT computation with input integer polynomials. An immediate question is the applicability of our verification approach to the operations `ffLDL`* and `ffSampling` in the signature generation. For `ffLDL`*, it is a straightline program with floating-point divisions so we can only verify the computation once floating-point division is verified. For `ffSampling`, it is built upon the one-dimensional discrete Gaussian sampler with a rejection loop. Therefore, CryptoLine along cannot verify this operation. We believe CryptoLine should be used as a plug-in of formal verification tools handling the rejection loop.

6.4 Applications to Other Lattice-Based Schemes

Our formal verification approach applies to several digital signature schemes. For ModFalcon [CPSWX20], since it also relies on the fast Fourier sampling from [DP16], one needs to apply FFT in a similar fashion as in Falcon's signature generation. For Mitaka [Esp+22], there are two samplers proposed by [Esp+22]: the hybrid sampler built upon the Gram-Schmidt orthogonalization with the aid of complex FFT and the integer arithmetic friendly sampler built upon the integral Gram decomposition by [DGPY20]. For the former, our verification approach applies since one needs to apply complex FFT. For the latter, integral Gram decomposition reduces to writing a positive integer as a sum of four squared integers and the fastest know algorithms are the randomized ones [PT18]. It seems difficult to find an unconditional deterministic algorithm for the problem [PT18, Section 5]. Therefore, it is unclear to us whether the integral version of Mitaka can be implemented securely and efficiently.

Acknowledgements. The author would like to thank Tiago Oliveira for a tutorial of Jasmin, Thomas Pornin for providing the experimental range of the intermediate floating-point numbers, eventually motivating the author to develope the range-checking, and Academia Sinica for the hospitality under grant AS-GCS-113-M07.

References

[Alm+17] Almeida, J.B., et al.: Jasmin: high-assurance and high-speed cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1807–1823 (2017). https://dl.acm.org/doi/10.1145/3133956.3134078

[Alm+19] Almeida, J.B., et al.: Machine-checked proofs for cryptographic standards: indifferentiability of sponge and secure high-assurance implementations of SHA-3. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1607–1622 (2019). https://dl.acm.org/doi/10.1145/3319535.3363211

[Alm+20] Almeida, J.B., et al.: The last mile: high-assurance and highspeed cryptographic implementations. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 965–982. IEEE (2020)

[Alm+23] Almeida, J.B., et al.: Formally verifying Kyber episode IV: implementation correctness. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2023**(3), 164–193 (2023). https://tches.iacr.org/index.php/TCHES/article/view/10960

[Ber+17] Bernstein, D.J., et al.: Gimli: a cross-platform permutation. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 299–320. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_15

[Ber05] Bernstein, D.J.: The Poly1305-AES message-authentication code. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 32–49. Springer, Heidelberg (2005). https://doi.org/10.1007/11502760_3

[Ber08] Bernstein, D.J.: ChaCha, a variant of Salsa20. In: Workshop record of The State of the Art of Stream Ciphers, pp. 273–278 (2008). https://www.ecrypt.eu.org/stvl/sasc2008/SASCRecord.zip.Citeseer

[CKKPSY12] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16

[CPSWX20] Chuengsatiansup, C., Prest, T., Stehlé, D., Wallet, A., Xagawa, K.: ModFalcon: compact signatures based on module-NTRU lattices. In: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, pp. 853–866 (2020)

[DGPY20] Ducas, L., Galbraith, S., Prest, T., Yu, Y.: Integral matrix gram root and lattice gaussian sampling without floats. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12106, pp. 608–637. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45724-2_21

[DP16] Ducas, L., Prest, T.: Fast Fourier Orthogonalization. In: Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, pp. 191–198 (2016). https://doi.org/10.1007/978-3-031-15777-6_7

[Esp+22] Espitau, T., et al.: Mitaka: a simpler, parallelizable, maskable variant of falcon. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022. LNCS, vol. 13277, pp. 222–253. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-07082-2_9

[FLSTWY19] Fu, Y.F., Liu, J., Shi, X., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y.: Signed cryptographic program verification with typed cryptoline. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 1591–1606 (2019). https://dl.acm.org/doi/abs/10.1145/3319535.3354199

[Hwa+22] Hwang, V., et al.: Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU. IACR Trans. Cryptogr. Hardw. Embed. Syst. 718–750 (2022). https://tches.iacr.org/index.php/TCHES/article/view/9838

[LLSTWY23] Lai, L.-C., Liu, J., Shi, X., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y.: Automatic verification of cryptographic block function implementations with logical equivalence checking. Cryptology ePrint Archive, Paper 2023/1861 (2023). https://eprint.iacr.org/2023/1861

[LSTWY19] Liu, J., Shi, X., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y.: Verifying arithmetic in cryptographic C programs. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 552–564. IEEE (2019). https://ieeexplore.ieee.org/document/8952256

[Por19] Pornin, T.: New efficient, constant-time implementations of falcon (2019). https://eprint.iacr.org/2019/893

[Por23] Pornin, T.: Improved key pair generation for falcon, BAT and Hawk (2023). https://eprint.iacr.org/2023/290

[Pre+20] Prest, T., et al.: Falcon. Submission to the NIST Post-Quantum Cryptography Standardization Project [NISTPQC] (2020). https://falcon-sign.info/

[PT18] Pollack, P., Treviño, E.: Finding the four squares in Lagrange's theorem. Integers **18A**, A15 (2018). https://api.semanticscholar.org/CorpusID:203588112

[PTWY18] Polyakov, A., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y.: Verifying arithmetic assembly programs in cryptographic primitives (invited talk). In: 29th International Conference on Concurrency Theory (CONCUR 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018). https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2018.4

[TWY17] Tsai, M.-H., Wang, B.-Y., Yang, B.-Y.; Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1973–1987 (2017). https://dl.acm.org/doi/abs/10.1145/3133956.3134076