Formal Verification of Emulated Floating-Point Arithmetic in Falcon

Vincent Hwang

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

Abstract. We show that there is a discrepancy between the emulated floating-point multiplications in the submission package of Falcon and the claimed behavior. In particular, we show that floating-point products with absolute values the smallest normal positive floating-point number are incorrectly zeroized. However, we show that the discrepancy doesn't effect the complex fast Fourier transform 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 is one of the recently selected digital signatures for standardization by the National Institute of Standards and Technology [Pre+20]. 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 serious of constant-time floating-point arithmetic with software emulation. We show that

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

We will publish our programs shortly.

2 Preliminaries

2.1 Falcon

Falcon is a lattice-based 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 $\mathbf{B}\hat{\mathbf{B}}$, 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, Algorithim 11] for a more detailed explanation of 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 $\mathtt{pk}$ and a secret key $\mathtt{sk}$

1: $(f, g, F, G) = \mathtt{solve\_NTRU}(x^n + 1,\ q)$ ▷ $fG - gF = q \bmod (x^n + 1)$
2: $h = gf^{-1} \bmod (x^n + 1, q)$
3: $\mathtt{sk} = (f, g, F, G)$
4: $\mathtt{pk} = h$
5: **return** $\mathtt{pk}, \mathtt{sk}$

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

Algorithm 2: Falcon signature generation from the reference implementation.

Inputs: a message m and a secret key $\texttt{sk}$
Outputs: a signature $\texttt{sig}$

$r \leftarrow \{0,1\}^{320}$ uniformly
$c = \texttt{HashToPoint}(r||m)$
$\hat{c} = \texttt{FFT}(c)$
$\mathbf{B} = \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$
$\hat{\mathbf{B}} = \begin{pmatrix} \hat{g} & -\hat{f} \\ \hat{G} & -\hat{F} \end{pmatrix} = \texttt{FFT}(\mathbf{B})$
$\mathbf{T} = \texttt{ffLDL}^*\left(\hat{\mathbf{B}}\hat{\mathbf{B}}^*\right)$
$\mathbf{T} = \texttt{Normalize}(\mathbf{T})$
$\mathbf{t} = \left(\frac{-\hat{c}\hat{F}}{q}, \frac{\hat{c}\hat{f}}{q}\right)$ ▷ $\mathbf{t} = (\hat{c}, 0)\,\hat{\mathbf{B}}^{-1}$
do
  do
    $\mathbf{z} = \texttt{ffSampling}(\mathbf{t}, \mathbf{T})$
    $\mathbf{s} = (\mathbf{t} - \mathbf{z})\,\hat{\mathbf{B}}$
  while $||\mathbf{s}||^2 > \beta^2$
  $(s_1, s_2) = \texttt{iFFT}(\mathbf{s})$
  $s = \texttt{Compress}(s_2, 8 \cdot \texttt{sbytelen} - 328)$
while $s == \perp$
$\texttt{sig} = (r, s)$
return $\texttt{sig}$

$2n$-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 $\prod_{i=0,\ldots,n-1} \mathbb{C}[x]/\langle x - \omega_{2n}^{1+2i}\rangle$ up to the bitreversal permutation in $O(n \log_2 n)$ operations in $\mathbb{C}$. 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 $\texttt{s}$ for the sign, an 11-bit $\texttt{e}$ for the biased exponent, and a 52-bit $\texttt{m}$ for the mantissa. When the biased exponent satisfies $0 < \texttt{e} < 2047$, the floating-point number corresponds to the following real number:

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

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

Algorithm 3: Falcon signature verification.

Inputs: a message m, a signature $\texttt{sig}$, 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

- $\texttt{e} = 0, \texttt{m} = 0$: This corresponds to a zero value. Notice that there are two zeros ± 0 distinguished by the sign $\texttt{s}$.
- $\texttt{e} = 0, \texttt{m} \neq 0$: This corresponds to the denormalized number $(-1)^{\texttt{s}}\, 2^{\texttt{e}-1074}\texttt{m}$.
- $\texttt{e} = 2047, \texttt{m} = 0$: This corresponds to an infinity. Notice that there are also two infinities $\pm\infty$ distinguished by the sign $\texttt{s}$.
- $\texttt{e} = 2047, \texttt{m} \neq 0$: 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 later doesn't hold, but it doesn't impact the complex fast Fourier transform in 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 we 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` imposes `P` and `Q` to the corresponding backend tools.

Assertions are used alone for verifying properties, and assumptions can be used in conjunction with assertions for transferring predicates between the two 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.

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. Comparing to CryptoLine, verification in EasyCrypt requires much more effort by explicitly applying various lemmas instead of simply asserting 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. Extract the mantissas and add them with 2^{52} as if the floating-point inputs are non-zero.
2. Compute the product of mantissas with radix-25 arithmetic.
3. Normalize the product to a 55-bit value.
4. Compute the exponent field as the sum of input exponent fields with a corrective subtraction.
5. Compute the sign field as the exclusive-or of the input sign fields.
6. Zeroize the product if any of the input exponent fields is zero.
7. Zeroize the product if the resulting exponent is too small.
8. Zeroize the exponent field if the product is zero.
9. Assemble the sign field, exponent field, and the upper 53 bits of the 55-bit product.
10. Increment the resulting floating-point as an unsigned 64-bit number 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 where the correct result is a floating-point number with absolute value the smallest noraml 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 exponent 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.

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: int32_t ex, exy, e, d;
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 = (x » 52) & (2^11 − 1);
10: ey = (y » 52) & (2^11 − 1);
11: e = ex + ey - 2100;
12: (zu, e) = normalize(zu, e, 55);
13: s = (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 & -((int32_t)(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;
```

For simplicity, we first assume that the product of mantissas is a 105-bit number (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 $\texttt{ex} = e$ and $\texttt{ey} = 1023 - e$ 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 $\texttt{xu} = 2^{52} + m$ and $\texttt{yu} = \left\lfloor \frac{2^{105}}{2^{52}+m} \right\rfloor$ with an m satisfying

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

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 the 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.

```
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.

```
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;
}
```

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 fast Fourier transform where the inputs are polynomials with integer coefficients in $\left[-2^{15}, 2^{15}\right)$. 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 dificulties are as follows:

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

We don't know of any systematic approaches resolving the two difficulties. Nevertheless, we find the following 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. 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. 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 $2^{\mathtt{e}-1075}$ where $\mathtt{e}$ is the smallest exponent field of the two and choosing $\mathtt{e}$ 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 possition, 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 on floating-point addition/subtraction. After replacing all the floating-point arithmetic with the range arithmetic in the FFT of Falcon, we transform all the output-input 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.

[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.

– 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 $\left[-2^{15}, 2^{15}\right)$.

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: $\quad (d_0, d_1) = (|a.u - b.l|, |b.u - a.l|)$
4: $\quad c.l = \min(d_0, d_1)$
5: $\quad c.u = a.u + b.u$
6: $\quad$ **return** c
7: $(d_0, d_1) = (|a.u - b.l|, |b.u - a.l|)$
8: $c.u = a.u + b.u$
9: $(s, e, m) = t.l$
10: $c.l = (s, e - 52, 0)$
11: **return** c

5 Equivalence Proofs

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

5.1 Our Implementations and 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. 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 later is more readable.

6 How the Discrepancy is 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 doesn't follow our understanding of floating-point arithmetic. Our careful examinations eventually led to the C program extracting witnesses leading to 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 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.

References

[Alm+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "Jasmin: High-Assurance and High-Speed Cryptography". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. https://dl.acm.org/doi/10.1145/3133956.3134078. 2017, pp. 1807–1823.

[Alm+19] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. "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*. https://dl.acm.org/doi/10.1145/3319535.3363211. 2019, pp. 1607–1622.

[Alm+20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. "The last mile: High-assurance and high-speed cryptographic implementations". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 965–982.

[Alm+23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. "Formally verifying Kyber Episode IV: Implementation Correctness". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.3 (2023). https://tches.iacr.org/index.php/TCHES/article/view/10960, pp. 164–193.

[Ber+17] Daniel J Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, et al. "Gimli: a cross-platform permutation". In: *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. https://link.springer.com/chapter/10.1007/978-3-319-66787-4_15. Springer. 2017, pp. 299–320.

[Ber05] Daniel J Bernstein. "The Poly1305-AES message-authentication code". In: *International workshop on fast software encryption*. https://doi.org/10.1007/11502760_3. Springer. 2005, pp. 32–49.

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

[CKKPSY12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software Analysis Perspective". In: *Software Engineering and Formal Methods: 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings 10*. Springer. 2012, pp. 233–247.

[DP16] Léo Ducas and Thomas Prest. "Fast Fourier Orthogonalization". In: *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*. https://dl.acm.org/doi/10.1145/2930889.2930923. 2016, pp. 191–198.

[FLSTWY19] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. "Signed Cryptographic Program Verification with Typed Cryptoline". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. https://dl.acm.org/doi/abs/10.1145/3319535.3354199. 2019, pp. 1591–1606.

[Hwa+22] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. "Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022). https://tches.iacr.org/index.php/TCHES/article/view/9838, pp. 718–750.

[LLSTWY23] Li-Chang Lai, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. *Automatic Verification of Cryptographic Block Function Implementations with Logical Equivalence Checking*. Cryptology ePrint Archive, Paper 2023/1861. https://eprint.iacr.org/2023/1861. 2023.

[LSTWY19] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. "Verifying Arithmetic in Cryptographic C Pro-

grams". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://ieeexplore.ieee.org/document/8952256. IEEE. 2019, pp. 552–564.

[NIS] NIST, the US National Institute of Standards and Technology. *Post-Quantum Cryptography Standardization Project*. https://csrc.nist.gov/Projects/post-quantum-cryptography.

[Por19] Thomas Pornin. *New Efficient, Constant-Time Implementations of Falcon*. Cryptology ePrint Archive, Paper 2019/893. https://eprint.iacr.org/2019/893. 2019.

[Pre+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *Falcon*. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS]. https://falcon-sign.info/. 2020.

[PTWY18] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. "Verifying Arithmetic Assembly Programs in Cryptographic Primitives (Invited Talk)". In: *29th International Conference on Concurrency Theory (CONCUR 2018)*. https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2018.4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[TWY17] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. "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*. https://dl.acm.org/doi/abs/10.1145/3133956.3134076. 2017, pp. 1973–1987.