Masked Computation of the Floor Function and Its Application to the FALCON Signature

Pierre-Augustin Berthet[1,3], Justine Paillet[2,3] and Cédric Tavernier[3]

[1] Télécom Paris, LTCI, Palaiseau, France
[2] Université Jean-Monnet, Saint-Étienne, France
[3] Hensoldt France SAS, Plaisir, France

Abstract.
FALCON is candidate for standardization of the new Post Quantum Cryptography (PQC) primitives by the National Institute of Standards and Technology (NIST). However, it remains a challenge to define efficient countermeasures against side-channel attacks (SCA) for this algorithm. FALCON is a lattice-based signature that relies on rational numbers which is unusual in the cryptography field. While recent work proposed a solution to mask the addition and the multiplication, some roadblocks remain, most noticeably how to protect the floor function. We propose in this work to complete the existing first trials of hardening FALCON against SCA. We perform the mathematical proofs of our methods as well as formal security proof in the probing model using the Non-Interference concepts. We provide performances on a laptop computer of our gadgets as well as of a complete masked FALCON.

Keywords: Floor Function · Floating-Point Arithmetic · Post-Quantum Cryptography · FALCON · Side-Channel Analysis · Masking

1 Introduction

With the rise of quantum computing, mathematical problems which were hard to solve with current technologies will be easier to breach. Among the concerned problems, the Discrete Logarithm Problem (DLP) could be solved in polynomial times by the Shor quantum algorithm [Sho99]. As much of the current asymmetric primitives rely on this problem and will be compromised, new cryptographic primitives are studied. The National Institute of Standards and Technology (NIST) launched a post-quantum standardization process [CCJ+16]. The finalists are CRYSTALS Kyber [BDK+18, NIS24b], CRYSTALS Dilithium [DKL+18, NIS24a], SPHINCS+ [BHK+19, NIS24c] and FALCON [PFH+20].

Another concern for the security of cryptographic primitives is their robustness to a Side-Channel opponent. Side-Channel Analysis (SCA) was first introduced by Paul Kocher [Koc96] in the mid-1990. This new branch of cryptanalysis focuses on studying the impact of a cryptosystem on its surroundings. As computations take time and energy, an opponent able to access the variation of one or both could find correlations between its physical observations and the data manipulated, thus resulting in a leakage and a security breach. Thus, the study of weaknesses in the implementations of new primitives and the way to protect them is an active field of research.

While many efforts have been done to protect CRYSTALS Dilithium and CRYSTALS Kyber, summed up by Ravi et al. [RCDB24], FALCON has been less covered. Indeed, the

E-mail: berthet@telecom-paris.fr (Pierre-Augustin Berthet), justine.paillet@univ-st-etienne.fr (Justine Paillet), cedric.tavernier@hensoldt.net (Cédric Tavernier)

algorithm relies on floating-point arithmetic, for which there is little literature on how to protect it.

1.0.1 Related Work

Previous works have identified two main weaknesses within the signing process of Falcon: the pre-image computation and the Gaussian sampler. The latest is proved vulnerable by Karabulut and Aysu [KA21] using an ElectroMagnetic (EM) attack. Their work was later improved by Guerreau et al. [GMRR22]. To counter those attacks, Chen and Chen [CC24] propose a masked implementation of the addition and multiplication of FALCON. However, they did not delve into the second weakness of Falcon, the Gaussian sampler. The Gaussian sampler is vulnerable to timing attacks, as shown by previous work [GBHLY16, EFGT17, MHS+19, PBY17]. An isochronous design was proposed by Howe et al. [HPRR20] to counter those attacks. Nonetheless, a successful single power analysis (SPA) was proposed by Guerreau et al. [GMRR22] and further improved by Zhang et al. [ZLYW23]. There is currently no masking countermeasure for FALCON's Gaussian Sampler. Existing work [EFG+22] tends to rewrite the Gaussian Sampler to remove the use of floating arithmetic, thus avoiding the challenge of masking the floor function.

1.0.2 Our Contribution

In this work, we further expand the countermeasure from Chen and Chen [CC24] and apply it to the Gaussian Sampler. We propose a masking method based on the mantissa truncation to compute the floor function as well as a method to mask the division. We discuss the application of those methods to masking FALCON.

Relying on the previous work of Chen and Chen [CC24], we also verify the higher-order security of our method in the probing model. Our formal proofs rely on the Non-Interference (NI) security model first introduced by Barthe et al. [BBD+16].

We provide some performances of our methods and compare them with the reference unmasked implementation and the previous work of Chen and Chen [CC24]. The implementation is tested on a laptop computer with an Intel-Core i7-11800H CPU and can be further optimized.

2 Notation and Background

2.1 Notations

- We denote by $A \backsim B$ the set A excluding the values of set B, *id est* $(A \backsim B) \bigcap B = \emptyset$. We denote by $\mathbb{K}^-$ the negative values of the set $\mathbb{K}$ and by $\mathbb{K}^*$ its non-zero values.
- For $x \in \mathbb{R}$, we denote the floor function of x by $\lfloor x \rfloor$.
- We will use the dot . as the separator between the integer part i and the fractional part f of a real number $x = i.f$.
- If (b_i) is a 64-bit Boolean sharing for bit value b, we denote $(-b_i)$ a 64-bit Boolean sharing for $2^{64} - b$. It means that if $b = 0$, $(-b_i)$ is a 64-bit boolean sharing for 0, and $b = 1$, $(-b_i)$ is a 64-bit boolean sharing for 0xFFFFFFFFFFFFFFFF.

For algorithmic extracts of FALCON [PFH+20], we use the original paper notations.

2.2 Diagram Legend

The diagrams in Section 5 use the same legend:

- Probing sets are denoted by P_i or O and are colored in red.
- Simulation sets are denoted by S_i^j and are colored in blue.
- *t-SNI* gadgets are colored in green.
- *t-NI* gadgets are colored in black.

2.3 FALCON Sign

FALCON [PFH+20] is a Lattice-Based signature using the GPV framework over the NTRU problem. In this paper, we will focus on the Gaussian Sampler used in the signature algorithm. For more details on the key generation or the verification, refer to the original paper of FALCON[PFH+20].

2.3.1 Signature

The signature follows the Hash-Then-Sign strategy. The message m is salted with a random value r and then hashed into a challenge c. The remainder of the signature aims at building an instance of the SIS problem upon c and a public key h, *id est* finding $\vec{s} = (s_1, s_2)$ such as $s_1 + s_2 h = c$. Hence, $\vec{s} = (\vec{t} - \vec{z})\mathbf{B}$ with $\vec{t}$ a pre-image vector and $\vec{z}$ provided by a Gaussian Sampler must be computed. Chen and Chen [CC24] focus on masking the pre-image vector computation. In this work, we mask the Gaussian Sampler and provide performances for the entire signature algorithm. This algorithm is detailed in [PFH+20] in the corresponding section.

2.3.2 Gaussian Sampler

The Gaussian Sampler denoted by SamplerZ can be evaluated from the three following functions, ApproxExp, BerExp and BaseSampler:

ApproxExp. This function return $2^{63} \times ccs \times e^{-x}$ and depends of a matrix C defined in page 42 of [PFH+20]:

Algorithm 1: ApproxExp(x,ccs) [PFH+20]

Data: floating-point values $x \in [0, \ln(2)]$ and $ccs \in [0, 1]$
Result: An integral approximation of $2^{63} \cdot ccs \cdot \exp(-x)$

$y \leftarrow C[0]$; `// ` y `and` z `remain in` $\{0 \cdots 2^{63} - 1\}$ `the whole algorithm`
$z \leftarrow \lfloor 2^{63} \cdot x \rfloor$;
for i *from 1 to 12* **do**
  $y \leftarrow C[i] - (z \cdot y) >> 63$;
$z \leftarrow \lfloor 2^{63} \cdot ccs \rfloor$;
$y \leftarrow (z \cdot y) >> 63$;
return y;

BerExp. This function return 1 with probability $ccs \times e^{-x}$:

Algorithm 2: BerExp(x,ccs) [PFH+20]

Data: floating-point values x, $ccs \geq 0$
Result: A single bit, equal to 1 with probability $\approx ccs \cdot \exp(-x)$

```
s ← ⌊x/ln(2)⌋ ;          // Compute the unique decomposition x = ln(2^s) + r with
  (r, s) ∈ [0, ln(2)) × Z^+
r ← x − s · ln(2);
s ← min(s, 63);
z ← (2 · ApproxExp(r, ccs) − 1) >> s;
i ← 64;
do
    i ← i − 8;
    w ← UniformBits(8) − ((z >> i) & 0xff);
while ((w = 0) and (i > 0));
return ⟦w < 0⟧;
```

BaseSampler This function samples a random integer between 0 and 18:

Algorithm 3: BaseSampler() [PFH+20]

Data: –
Result: An integer $z_0 \in \{0, \cdots, 18\}$ such that $z \sim \chi$

```
u ← UniformBits(72);
z_0 ← 0;
for i from 0 to 17 do
    z_0 ← z_0 + ⟦u < RCDT[i]⟧;
return z_0;
```

where RCDT is defined in Falcon Specification [PFH+20].

The Gaussian Sampler is constructed as follows:

Algorithm 4: SamplerZ(μ,σ') [PFH+20]

Data: floating-point values μ,$\sigma' \in \mathcal{R}$ such that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
Result: $z \in \mathbb{Z}$ sampled from a distribution very close to $D_{\mathbb{Z},\mu,\sigma'}$

```
r ← μ − ⌊μ⌋;
ccs ← σ_min/σ';
while 1 do
    z_0 ← BaseSampler();
    b ← UniformBits(8) & 0x1;
    z ← b + (2 · b − 1)z_0;
    x ← (z−r)^2/(2σ'^2) − z_0^2/(2σ_max);
    if BerExp(x, ccs) = 1 then
        return z + ⌊μ⌋;
```

2.4 Floor Function

The floor function is defined as follows:

Definition 1. $\forall x \in \mathbb{R}$, the floor function of x, denoted by $\lfloor x \rfloor$, returns the greatest integer z such as $z \leq x$.
$\forall x \in \mathbb{R}$, the truncate function of $x = i.f, (i, f) \in \mathbb{Z} \times \mathbb{N}$, denoted by $truncate(x)$, returns i.

2.4.1 Binary64 Encoding

A floating-point [Kah96] is encoded with a sign bit s, a 11-bits long exponent e and a 52-bits long mantissa m such as:

$$x \in \mathbb{R}, x = (-1)^s \times 2^{e-1023} \times (1 + m \times 2^{-52}). \tag{1}$$

2.4.2 Computing The Floor

Computing the floor function on a floating-point is performed by truncating the mantissa according to the value of the exponent and the sign:

- If $e < 1023$ then if $s = 0$ then $\lfloor x \rfloor = 0$ else $\lfloor x \rfloor = -1$. Indeed,

$$(e < 1023) \wedge (s = 0) \implies 0 \leq x \leq 2^{-1} + m \times 2^{-53} < 1 \tag{2}$$

$$(e < 1023) \wedge (s = 1) \implies 0 > x \geq -2^{-1} + -m \times 2^{-53} \geq -1. \tag{3}$$

- If $e > 1074$ then $\lfloor x \rfloor = x$. We have

$$e > 1074 \implies |x| = 2^{e-1023} + m \times 2^{e-1023-52} \tag{4}$$

$$= (2^{e-1023}) \in \mathbb{N}^* + (m \times 2^{e-1075}) \in \mathbb{N} \implies x \in \mathbb{N}^*. \tag{5}$$

  The sign bit s only changes "$\in \mathbb{N}$" in "$\in \mathbb{Z}^-$".

- If $1023 \leq e \leq 1074$ then we truncate the mantissa m of x and remove its $1074 - e$ last bits $m^{[52-(e-1023):1]}$. That way we have

$$1023 \leq e \leq 1074 \implies x = 2^{e-1023} + m^{[64:1075-e]} \times 2^{52-(e-1023)+e-1023-52} \tag{6}$$

$$= (2^{e-1023}) \in \mathbb{N}^* + (m^{[64:1075-e]}) \in \mathbb{N}. \tag{7}$$

  However, this only provides $truncate(x)$. To get $\lfloor x \rfloor$, one has to take into account the sign bit s. We can rely on the fact that $\forall x \in \mathbb{R}^- \backsim \mathbb{Z}, truncate(x) = \lfloor x \rfloor + 1$ and $\forall x \in \mathbb{R}^+, truncate(x) = \lfloor x \rfloor$. Thus, recovering the sign bit allows to properly compute the floor function from the truncated one in this case.

Remark 1. To compute the $truncate(x)$ function, the same method can be applied but discard the use of the sign. For the case $e < 1023$, the result is always 0.

This method requires the knowledge of the exponent and the sign, which are both some sensitive values. We propose in this work a method to perform this truncation securely.

2.5 Masking

Masking is a generic countermeasure against SCA at the software level. Instead of processing a sensitive data, it is split into random shares which are processed separately, like in Boolean and Arithmetic masking [MOP08]. Masking security can be evaluated with the *t-probing* model, first introduced in [ISW03]. As consequence, a gadget is said secured against t-order attacks if no information can be recovered by any set of t intermediate values. However, for the composition of gadgets we use a stronger model introduced in [BBD+16]: the (Strong) Non-Interference model.

Definition 2. (t-Non Interference (t-*NI*) security [BBD+16]). A gadget is said t-Non Interference (t-*NI*) secure if every set of t intermediate values can be simulated by no more than t shares of each of its inputs.

t-*NI* gadgets composition does not imply t-*NI* security. We need a stronger definition for this:

Definition 3. (t-Strong Non Interference (t-*SNI*) security [BBD+16]). A gadget is said t-Strong Non-Interference (t-*SNI*) secure if for every set of t_I of internal intermediate values and t_O of its output shares with $t_I + t_O \leq t$, they can be simulated by no more than t_I shares of each of its inputs.

We consider these models in Section 5 to demonstrate the security of our design. We rely on existing gadgets and propose new ones, as shown in Table 1.

Table 1: List of gadgets, their security and their reference

Algorithm	Description	Security	Reference
SecAnd	AND of Boolean shares	t-*SNI*	[BBD+16],[ISW03]
SecAdd	Addition of Boolean shares	t-*SNI*	[BBE+18],[CGTV15]
A2B	Arithmetic to Boolean conversion	t-*SNI*	[SPOG19]
B2A	Boolean to Arithmetic conversion	t-*SNI*	[BCZ18]
RefreshMasks	t-NI refresh of masks	t-*NI*	[BBD+16], [BCZ18]
Refresh	t-SNI refresh of masks	t-*SNI*	[BBD+16]
SecOr	OR of Boolean shares	t-*SNI*	[CC24]
SecNonZero	NonZero check of shares	t-*SNI*	[CC24]
SecFprUrsh	Right-shift with sticky bit	t-*SNI*	[CC24]
SecFprNorm64	Normalization to $[2^{63}, 2^{64})$	t-*NI*	[CC24]
SecFprAdd	Floating addition	t-*SNI*	[CC24]
SecFprMul	Floating multiplication	t-*SNI*	[CC24]
SecFprUrsh$_f$	Right-shift without sticky bit	t-*SNI*	Algorithm 5
RemoveDecimal	Truncate the mantissa	t-*SNI*	Algorithm 6
SetExponentZero	Set exponent to zero	t-*SNI*	Algorithm 7
SecFprBaseInt	Compute the floor	t-*SNI*	Algorithm 9
SecFprComp	Compares two values	t-*SNI*	Algorithm 10
SecFprScalePow2	Multiplies by a power of 2	t-*SNI*	Algorithm 11
SecFprInv	Inversion	t-*SNI*	Algorithm 12
Minimum63	Comparison with 63	t-*SNI*	Algorithm 13

3 Masking the Floor Function

In Section 2.4.2 we have described how to compute the floor using floating-point arithmetic. We present now the corresponding masking gadgets.

Remark 2. With small modifications, our design can also be used to compute the *truncate* and the *rounding* functions.

To perform the floor function, we have to truncate the mantissa, modify the exponent as well as address the sign and the special case of having 0 as a result. To do this we introduce several gadgets:

3.0.1 SecFprUrsh$_\mathbf{f}$

This gadget is a modification of the SecFprUrsh gadget from [CC24] (Algorithm 9 page 286). Our method, SecFprUrsh$_\mathrm{f}$ (Algorithm 5), does not keep the sticky bit but returns the removed part instead.

Algorithm 5: $\text{SecFprUrsh}_{\text{floor}}((my_i),(cx_i))$

Data: 6-bit arithmetic shares $(cx_i)_{1\leq i\leq n}$ for value cx;
64-bit boolean shares $(my_i)_{1\leq i\leq n}$ for sign value my.
Result: 64-bit boolean shares $(my'_i)_{1\leq i\leq n}$ for value $my >> cx$
64-bit boolean shares $(rot_i)_{1\leq i\leq n}$ for value $my^{[cx:1]}$.

$(m_i)_{1\leq i\leq n} \leftarrow ((1 << 63), 0, \cdots, 0)$;
for *i from 1 to n* **do**
  Right-Rotate (my_i) by cx_j;
  $(my_i) \leftarrow$ RefreshMasks$((my_i))$;
  Right-Rotate (m_i) by cx_j;
  $(m_i) \leftarrow$ RefreshMasks$((m_i))$;
$len \leftarrow 1$;
while $len \leq 32$ **do**
  $(m_i) \leftarrow (m_i \oplus (m_i >> len))$;
  $len \leftarrow len << 1$;
$(my'_i) \leftarrow$ SecAnd$((my_i),(m_i))$;
$(rot_i) \leftarrow$ SecAnd$((my_i),(\neg(m_i)))$;
return $((my'_i),(rot_i))$;

3.0.2 RemoveDecimal

The $\text{SecFrpUrsh}_{\text{floor}}$ gadget is used within another gadget, RemoveDecimal (Algorithm 6). We use this gadget to truncate the mantissa. We first shift the mantissa my by $cd = 52 - cx$, using $\text{SecFprUrsh}_{\text{floor}}$. Once the mantissa is shifted, we have performed the $truncate(x)$ function. As described in Section 2.4.2, for the floor we also have to check whether the sign sy is 1. In that case, we check by applying SecNonZero on the mantissa part removed by $\text{SecFprUrsh}_{\text{floor}}$, with result denoted b. If the result is 0, we apply the floor function to a negative integer. Otherwise, we have to retrieve 1 to the final result in accordance with Section 2.4.2 and proceed by securely adding $cp = s \wedge b$ to the shifted my, as summed up in Table 2.

Table 2: Truth table of $cp = s \wedge b$ and interpretations

sy	b	$cp = sy \wedge b$	Interpretation
0	b	0	x is a positive real
1	0	0	x is an negative integer
1	1	1	x is an non-integer negative real

3.0.3 SetExponentZero

Finally, we have to address the exponent computation. This is done with the SetExponentZero (Algorithm 7) gadget. This function handles specific Binary64 encoding cases, specifically the encoding of 0 and the one of -1. Indeed, if $\mid x \mid< 1$ and $sy = 0$, then the expected result is 0 in its Binary64 form. Else, if $sy = 1$ and $\mid x \mid< 1$, then the expected result is -1 in its Binary64 form. Table 3 highlights the relation between s_y, b and the expected result.

Algorithm 6: RemoveDecimal$_{\text{floor}}((my_i), (ey_i), (sy_i), (cx_i))$

Data: 64-bit boolean shares $(my_i)_{1 \le i \le n}$ for mantissa value my;
16-bit arithmetic shares $(ey_i)_{1 \le i \le n}$ for exponent value ey;
1-bit boolean shares $(sy_i)_{1 \le i \le n}$ for sign value sy
16-bit arithmetic shares $(cx_i)_{1 \le i \le n}$ for value cx = ex-2013.
Result: 64-bit boolean shares $(my_i)_{1 \le i \le n}$ for mantissa value $my >> (52 - cx)$;
16-bit arithmetic shares $(ey_i)_{1 \le i \le n}$ for exponent value $ey + (52 - cx)$

$cx_1 \leftarrow cx_1 - 52$;
$(c_i) \leftarrow \text{A2B}((cx_i))$;
$(cp_i) \leftarrow ((c_i^{(16)}))$;
$(c_i) \leftarrow \text{SecAnd}(\text{Refresh}((c_i)), (-cp_i))$;
$(cx_i) \leftarrow \text{B2A}((c_i))$;
$(my_i), (rot_i) \leftarrow \text{SecFprUrsh}_f((my_i), (-cx_i))$;
$(b_i) \leftarrow \text{SecNonZero}((rot_i))$;
$(cp_i) \leftarrow \text{SecAnd}((cp_i), (sy_i))$;
$(cp_i) \leftarrow \text{SecAnd}((cp_i), (b_i))$;
$(my_i) \leftarrow \text{SecAdd}((my_i), (cp_i))$;
$(ey_i) \leftarrow (\text{Refresh}(ey_i) - cx_i)$;
return $((my_i),\ (ey_i),\ (b_i))$;

Algorithm 7: SetExponentZero$_{\text{floor}}((ey_i), (sy_i), (b_i))$

Data: 16-bit arithmetic shares $(ey_i)_{1 \le i \le n}$ for exponent value ey;
1-bit boolean shares $(sy_i)_{1 \le i \le n}$ for sign value sy
64-bit boolean shares $(b_i)_{1 \le i \le n}$.
Result: 16-bit boolean shares $(ey_i)_{1 \le i \le n}$ for exponent value $ey + (52 - cx)$;
1-bit boolean shares $(sy_i)_{1 \le i \le n}$ for sign value.

$(ey_i) \leftarrow \text{A2B}((ey_i))$;
$(b'_i) \leftarrow (-sy_i)$;
$(b'_i) \leftarrow \text{SecOr}((b'_i), (b_i))$;
$(ey_i) \leftarrow \text{SecAnd}((ey_i, b'_i))$;
$(sy_i) \leftarrow \text{SecAnd}((sy_i, b'_i))$;
return $((ey_i), (sy_i))$;

Table 3: Encoding 0, -1 or others: Truth table

$-sy$	b	$-sy \vee b$	Interpretation
$0 \cdots 0$	$0 \cdots 0$	$0 \cdots 0$	"Small" positive number : $ey = 0$ and $sy = 0$
$1 \cdots 1$	$0 \cdots 0$	$1 \cdots 1$	"Small" negative number : $ey = 1023$ and $sy = 1$
$-sy$	$1 \cdots 1$	$01 \cdots 1$	Non zero number : $ey = ey$ and $sy = sy$

3.0.4 SecFprBaseInt$_{\text{f}}$:

The gadget SecFprBaseInt$_{\text{f}}$ (Algorithm 9) is the main function of the masked floor, the masked *truncate*, and the masked *rounding*. Gadgets and Zero$_f$ are parameterized[1] by these functions.

This paper focuses on f = floor. The sign, exponent and mantissa are extracted from the masked Binary64 encoding used by [CC24] and place them into three variables s_y, e_y,

[1] $\text{Zero}_{\text{floor}} = \text{Zero}_{\text{trunc}} = 1023$ and $\text{Zero}_{\text{round}} = 1022$

and m_y, which are directly linked to the output of the algorithm. This extraction is performed with the SecFprExtract algorithm (Algorithm 8):

Algorithm 8: SecFprExtract(x)

Data: 64-bit boolean shares $(x_i)_{1 \leq i \leq n}$ for value x
Result: 64-bit boolean shares $(mx_i)_{1 \leq i \leq n}$ for mantissa value mx;
16-bit arithmetic shares $(ex_i)_{1 \leq i \leq n}$ for exponent value ex;
1-bit boolean shares $(sx_i)_{1 \leq i \leq n}$ for sign value s.

$(mx_i) \leftarrow (x_i^{[52:1]})$;
$(mx_i) \leftarrow \mathrm{SecAdd}((mx_i), (2^{52}, 0, \cdots, 0))$; `// add implicit bit in the mantissa`
$(ex_i) \leftarrow (x_i^{[63:53]})$;
$(ex_i) \leftarrow \mathrm{B2A}((ex_i))$;
$(sx_i) \leftarrow (x_i^{(64)})$;
return $((mx_i), (ex_i), (sx_i))$;

The inequality $c_x = e_y - \mathrm{Zero}_f < 0$, corresponding to Equation 2, is checked. If cx is negative, $\mid x \mid < 1$ and we remove the decimals by $my = 0$. The algorithm SetExponentZero (Algorithm 7) is called later in the algorithm to encode the result according to this case. The two remaining cases are dealt with by $\mathrm{RemoveDecimal}_{floor}$ (Algorithm 6), as described in Section 2.4.2. The cases are as follows: If $cx \geq 52$, then x is an integer as shown in Equation 4 and no modification of the mantissa is required. Else, if $0 \leq c_x \leq 51$, we truncate the mantissa consequently.

Algorithm 9: $\mathrm{SecFprBaseInt_f}$(x)

Data: 64-bit boolean shares $(x_i)_{1 \leq i \leq n}$ for value x
Result: 64-bit boolean shares $(y_i)_{1 \leq i \leq n}$ for mantissa value y = f(x).

$((my_i), (ey_i), (sy_i)) \leftarrow \mathrm{SecFprExtract}((x_i))$;
$(cx_i) \leftarrow (ey_i)$, $cx_1 \leftarrow ey_1 - \mathrm{Zero}_f$;
$(c_i) \leftarrow \mathrm{A2B}((cx_i^{(16)}))$;
$(my_i) \leftarrow \mathrm{SecAnd}((my_i), (\neg(-c_i)))$;
$(my_i), (ey_i), (Rnd_i) \leftarrow \mathrm{RemoveDecimal}_f((my_i), (ey_i), \mathrm{Refresh}(sy_i), \mathrm{Refresh}((cx_i)))$;
$(my_i), (ey_i) \leftarrow \mathrm{SecFprNorm64}((my_i), (ey_i))$;
$(my_i) \leftarrow (my_i^{[63:11]})$;
$ey_1 \leftarrow ey_1 + 11$;
$(ey_i), (sy_i) \leftarrow \mathrm{SetExponentZero}_f((ey_i), (\neg(-c_i)), (s_i), (Rnd_i))$;
$(y_i^{(64)}) \leftarrow (sy_i)$, $(y_i^{[63:53]}) \leftarrow (ey_i)$, $(y_i^{[52:1]}) \leftarrow (my_i)$;
return (y_i);

As the algorithm RemoveDecimal does not normalize the mantissa, then SecFprNorm64 (see [CC24] Algorithm 10 page 286) is called and returns a shifted my as well as ey to set the mantissa back to bits $[52 : 1]$ and update ey. Finally, the last step in the algorithm, before reformatting the initial encoding, consists in computing the specific encoding of "0" if it is the expected result, by applying the $\mathrm{SetExponentZero}_f$ function (Algorithm 7).

4 Application to Falcon : Gaussian Sampler

The floor function has been described above and we propose now to address the SamplerZ function (Algorithm 4 or see [PFH+20] Algorithm 15 page 43). In the algorithms SamplerZ and BerExp (Algorithm 2 or see [PFH+20] Algorithm 14 page 43), division operations are used. Most of these divisions involve constants as the divisor, allowing us to pre-calculate the inverse and perform a multiplication. However, the first division in SamplerZ (line 2) involves a division with secret information. Hence, we must perform securely a division by an arbitrary value. To divide by x, we invert it and then compute a multiplication. Computing the inverse involves performing a Euclidean division until obtaining sufficient precision (55 bits) to construct it.

4.1 Division:

Let $x = (s_x, e_x, m_x)$ and $\frac{1}{x} = y = (s_y, e_y, m_y)$. As the inverse operation preserves the sign, $s_y = s_x$. To compute the exponent e_y, we subtract 1023 by $c_x = e_x - 1023 + b$, where b depends on if x is a power of two and cheap to invert in Binary64. This condition is verified when the mantissa is 0. If not, we set $b = 1$ to further subtract 1023 and get the correct exponent e_y. This is obtained by performing $b =$SecNonZero(m_x). The exponent is computed with the following Equation 8:

$$e_y = 1023 - (e_x - 1023 + b) = 2046 - e_x - b \tag{8}$$

Computing the mantissa corresponds to the Euclidean division: first, the dividend $d = (1 << c_x)$ is compared to x by computing $comp = \text{SecFprComp}(d, x)$ (Algorithm 10). The comparison algorithm is an adaptation of the swap part of the SecFprAdd function (see [CC24] Algorithm 13 page 290) where a similar comparison is performed.

Algorithm 10: SecFprComp$((x_i), (y_i))$

Data: 64-bit boolean shares $(x_i)_{1\leq i\leq n}$ for value x;
64-bit boolean shares $(y_i)_{1\leq i\leq n}$ for sign value y.
Result: 1-bit boolean shares $(comp_i)_{1\leq i\leq n}$ for value $[\![x < y]\!]$

Refresh$((x_i))$;
$(mx_i) \leftarrow (x_i^{[63:1]}), \quad (my_i) \leftarrow (y_i^{[63:1]})$;
$(d_i) \leftarrow \text{SecAdd}((mx_i), (\neg my_1, my_2, \cdots, my_n))$;
Refresh$((d_i))$;
$(b_i) \leftarrow \text{SecNonZero}((\neg d_1, d_2, \cdots, d_n))$;
$(b'_i) \leftarrow \text{SecNonZero}((\neg(d_1 \oplus 2^{63}), d_2, \cdots, d_n))$;
$(comp_i) \leftarrow (d_i^{(63)} \oplus b_i \oplus b'_i)$;
return $(comp_i)$;

If $x < d$, then the comparison algorithm outputs 1. This result is carried over to the new mantissa and we add $-x$ to d. Else, if $comp = 0$, no addition is performed on d. To continue the Euclidian divison, d is shifted one time to the left. Performing this shift is done by calling the SecFprScalePow2 (Algorithm 11) function. This function either multiplies by 2 or either divide by 2 its input, and truncates the result if necessary.

After getting by this way 53 bits (52 plus the implicit bit) of the mantissa m_y, two additional bits are computed to preserve the sticky bit. Consequently we get the 55 bits of the mantissa m_y.

4.2 Masking BerExp

BerExp (Algorithm 2) requires to securely compute a minimum as well as perform a right-shift by a sensitive value. For the minimum, the comparison is made between a constant equal to 63 and the sensitive value that we will denote here by $X = (sX, eX, mX)$. We check if $X \geq 64$. To do so we verify that the exponent eX is greater than 1029 and its sign sX is 0. In BerExp, X is always positive and we only check the exponent condition. As eX is a signed integer, we verify it by looking at the sign of the computation of $\epsilon = eX - 1029$. We use an A2B conversion to extract the sign bit $s\epsilon$. The final output

Algorithm 11: SecFprScalePow2($(x_i), p$)

Data: 64-bit boolean shares $(x_i)_{1 \leq i \leq n}$ for value x;
An integer p.
Result: 64-bit boolean shares $(y_i)_{1 \leq i \leq n}$ for value $x \times 2^p$

$(sx_i), (ex_i), (mx_i) \leftarrow$ SecFprExtract$((x_i))$;
$(b_i) \leftarrow$ SecNonZero$((x_i))$;
$(ex_i) \leftarrow$ B2A$((ex_i))$;
$ex_1 \leftarrow ex_1 + p$;
$(ex_i) \leftarrow$ A2B$((ex_i))$;
$(ey_i) \leftarrow$ SecAnd$((ex_i), -(b_i))$;
$(y_i^{(64)}) \leftarrow (sy_i), (y_i^{[63:53]}) \leftarrow (ey_i), (y_i^{[53:1]}) \leftarrow (my_i)$;
return *Refresh*(y_i);

Algorithm 12: SecFprInv((x_i))

Data: 64-bit boolean shares $(x_i)_{1 \leq i \leq n}$ for value x.
Result: 64-bit boolean shares $(y_i)_{1 \leq i \leq n}$ for value $1/x$

$(sx_i), (ex_i), (mx_i) \leftarrow$ SecFprExtract$((x_i))$;
$(b_i) \leftarrow$ SecNonZero$((mx_i))$;
$(ba_i) \leftarrow$ B2A(b_i);
$(ed_i) \leftarrow (ex_i + ba_i)$;
$(ey_i) \leftarrow (-ed_i)$;
$(ey_i) \leftarrow$ A2B$((ey_i))$, $(ed_i) \leftarrow$ A2B$((ed_i))$;
$(d_i) \leftarrow (ed_i << 52)$;
$(minusX_i) \leftarrow$ Or$((2^{63}, 0, \cdots, 0), (x_i))$;
for *j from 1 to 55* **do**
  $(comp_i) \leftarrow$ SecFprComp$((x_i), (d_i))$;
  $(my_i) \leftarrow (my_i \oplus (comp_i << (63 - j)))$;
  $(xcpy_i) \leftarrow$ SecAnd$((minusX_i), -(comp_i))$;
  $(d_i) \leftarrow$ SecFprAdd$((xcpy_i), (d_i))$;
  $(d_i) \leftarrow$ SecFprScalePow2$((d_i), 1)$;
$(my_i) \leftarrow$ SecAnd$((my_i), -(b_i))$;
$(y_i^{(64)}) \leftarrow$Refresh$((sy_i)), (y_i^{[63:53]}) \leftarrow (ey_i), (y_i^{[52:1]}) \leftarrow (my_i^{[54:3]})$;
$(f_i) \leftarrow$ SecOr(Refresh$(my_i^{(1)})$, $(my_i^{(3)})$);
$(f_i) \leftarrow$ SecAnd$((f_i), (my_i^{(2)}))$;
$(y_i) \leftarrow$ SecAdd$((y_i), (f_i))$;
return (y_i);

is given by the mask of $((-s\epsilon) \land X) \lor ((-(\neg s\epsilon)) \land 63)$. The minimum computations are performed in Algorithm 13.

Algorithm 13: Minimum63(x_i)

Data: 64-bit boolean shares $(x_i)_{1\leq i\leq n}$ for positive integer x;
Result: 64-bit boolean shares $(y_i)_{1\leq i\leq n}$ equal to the minimum between 63 and x

$(sx_i), (ex_i), (mx_i) \leftarrow$ SecFprExtract((x_i));
(st_i) is a masking of the value 63;
$ex_1 \leftarrow ex_1 - 1029)$;
$(ex_i) \leftarrow$ A2B((ex_i));
$(rA_i) \leftarrow$ SecAnd($(-(ex_i)^{(16)}), (x_i)$;
$(rB_i) \leftarrow$ SecAnd($(-(\neg(ex_i)^{(16)})), (x_i)$;
$(y_i) \leftarrow$ SecOr($(rA_i), (rB_i)$);
return (y_i);

To right-shift a masked Binary64 Y by another masked Binary64 $X \in [\![0, 63]\!]$, we use SecFprUrsh (Algorithm). However, we first convert X, a 64-bit boolean sharing, into a 6-bit arithmetic sharing. We denote $X = (sX, eX, mX)$. We have to take into account the possibility that $X = 0$. Thus, when injecting the implicit bit on each share, we take the mantissa mX and compute: $mX' = SecNonZero(eX)||mX$. To keep only the integer value, we perform a right-shift of the mantissa mX' by $52 - (eX - 1023)$. This is done with the SecFprUrsh function:

$$m = \text{SecFprUrsh}(mX', 52 - eX + 1023) \tag{9}$$

The result m is a 64-bit boolean sharing. As $X \in [\![0, 63]\!]$, only the 6 lower bits can be masks of 1, all the other bits are known to be masks of 0. Thus, we apply a B2A conversion on those 6 bits to get the masked integer value of X as an arithmetic sharing. The result of the shifting of Y by X is therefore SecFprUrsh(Y,$m^{[6:1]}$).

5 Security Proof

In this section we cover the *t-SNI* security of our design with $n = t + 1$ shares. We follow and rely on the same principles used by Chen and Chen [CC24] for our proofs. We aim to propose only *t-SNI* secure gadgets as the composition of those gadgets is itself *t-SNI*. this limits the risks of compositional flaws at the cost of performance overheads and more demanding randomness requirements.

5.1 Floor Function

Lemma 1. *The gadget* ***SetExponentZero$_{\text{floor}}$*** *(Algorithm 7) is* t-SNI *secure.*

Proof. We use an abstract diagram in Figure 1 for our demonstration. The gadget only contains *t-SNI* gadgets. By composition of *t-SNI* gadgets, this gadget is itself *t-SNI*. □

Lemma 2. *The gadget* ***SecFprUrsh$_{\text{floor}}$*** *(Algorithm 5) is* t-SNI *secure.*

Proof. The gadget **SecFprUrsh$_{\text{floor}}$** is a slight modification of the gadget **SecFprUrsh** from [CC24]. Our gadget does not compute the sticky bit but retains the rotated out information. We rely on their proof regarding the *t-SNI* security of the gadget **Rotate** (see [CC24], Lemma 3 and Figure 2). We now show that the operations below the rotation

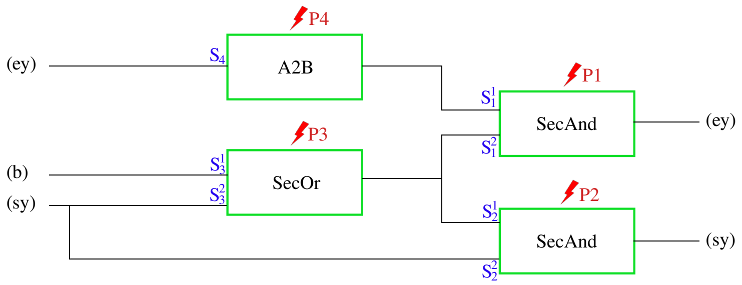

Figure 1: Abstract diagram of SetExponentZero$_{floor}$

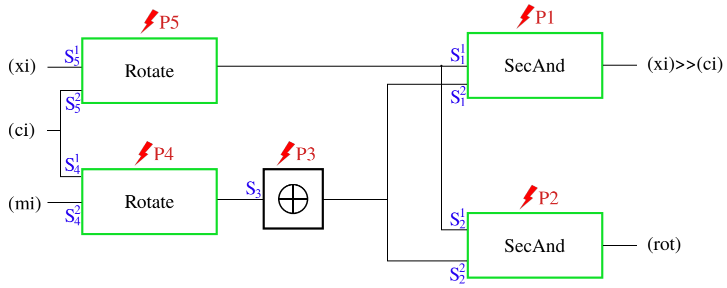

Figure 2: Abstract diagram of SecFprUrsh$_{floor}$

loop are *t-SNI* secure. We use an abstract diagram in Figure 2 for the demonstration. Let an adversary probe the intermediate values sets P_1 of **SecAnd**, P_2 of **SecAnd** and P_3 of **XOR**. As **SecAnd** is *t-SNI* secure, one can use the sets S_2^1,S_2^2 (resp. S_1^1,S_1^2) to simulate P_2 (resp. P_1) and the ouput shares of (rot) (resp. $(xi) >> (ci)$) with sizes no more than P_2 (resp. P_1). One can simulate the probing set of P_3 in the **XOR** and the simulation sets S_2^2 and S_1^2 with the output shares S_3 of the rotation of (mi). Indeed, as the **XOR** is a linear operation performed on each share separately, it is *t-NI* secure. All probes are now simulated with output shares $S_1^1 \cup S_2^1$ of the rotation of (xi) and S_3 of the rotation of (mi). We have $|S_1^1 \cup S_2^1| \leq |P_1| + |P_2|$ and $|S_3| \leq |P_3| + |S_2^2| + |S_1^2| \leq |P_3| + |P_2| + |P_1|$. Along with the internal probes P_5 and P_4 from the rotation loop, all gadgets can be simulated by input shares with no more than t_I values due to the *t-SNI* security showed at first in ([CC24], Lemma 3). □

Lemma 3. *The gadget* ***RemoveDecimal$_{floor}$*** *(Algorithm 6) is* t-SNI *secure.*

Proof. We use an abstract diagram in Figure 3 for the demonstration. We assume an adversary probes the intermediate values sets of the output shares O and P_i in each gadget for $i \in [\![1; 12]\!]$. We use simulation sets S_i^j to simulate the values for each gadget. *t-SNI* security implies that: if the size of all probing sets P_i is $t_I \leq t$ and if the size of values required to simulate in each gadget is smaller than t, then the simulation sets linked to the input shares are not bigger than t_I. The *t-SNI* gadgets imply $|S| \leq |P|$ and the *t-NI* gadgets imply $|S| \leq |P| + |O|$. As **Refresh**, **SecAnd**, **SecNonZero**, **SecFprUrsh**$_{floor}$, **B2A** and **A2B** are all *t-SNI* secure whereas **SecAdd** and "+" are *t-NI* secure, we can sequentially derive the following:

- $|S_1| \leq |P_1|$
- $|S_2^1|, |S_2^2| \leq |P_2| + |O_{(ey)}|$
- $|S_4^1|, |S_4^2| \leq |P_4| + |O_{(my)}|$
- $|S_5^1|, |S_5^2| \leq |P_5|$
- $|S_6| \leq |P_6|$
- $|S_7^1|, |S_7^2| \leq |P_7|$

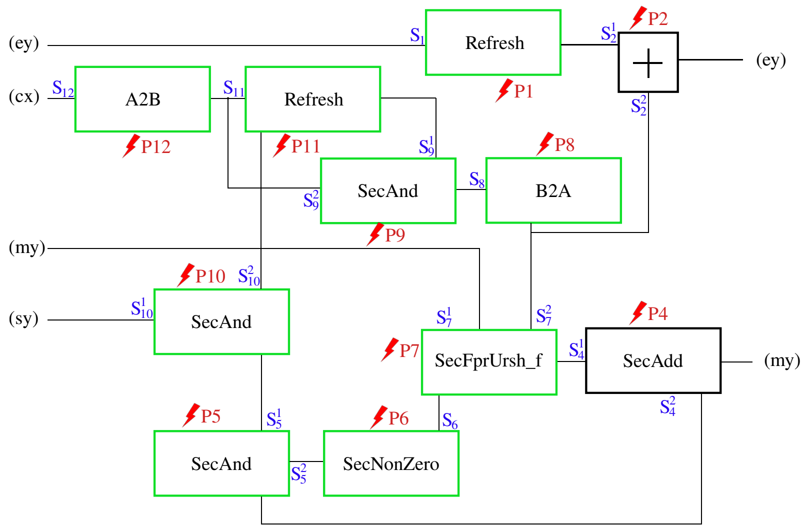

Figure 3: Abstract diagram of $\text{RemoveDecimal}_{\text{floor}}$

- $|S_8| \leq |P_8|$
- $|S_9^1|, |S_9^2| \leq |P_9|$
- $|S_{10}^1|, |S_{10}^2| \leq |P_{10}|$
- $|S_{11}| \leq |P_{11}|$
- $|S_{12}| \leq |P_{12}|$

Based on the previous inequalities, we know that no gadget requires more than $t_I + t_O = t$ values to be simulated. This above method can be applied to the input shares as well, with $|S_{10}^1| \leq |P_{10}|$ for (sy), $|S_7^1| \leq |P_7|$ for (my), $|S_{12}| \leq |P_{12}|$ for (cx) and $|S_2^1| \leq |P_2| + |S_1| \leq |P_2| + |P_1|$ for (ey), no sizes being more than t_I. □

Theorem 1. *The gadget* ***SecFprBaseInt***$_{\text{floor}}$ *(Algorithm 9) is* t-SNI *secure.*

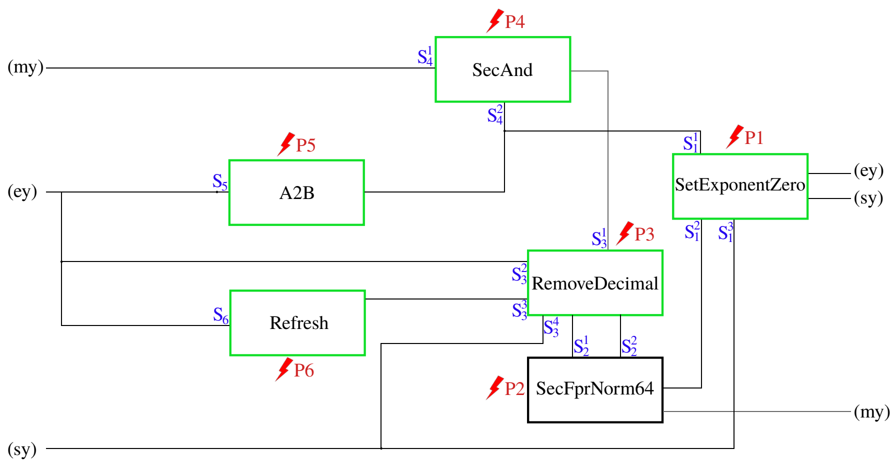

Figure 4: Abstract diagram of $\text{SecFprBaseInt}_{\text{floor}}$

Proof. We use the same method as for the demonstration of Lemma 3. We use an abstract diagram in Figure 4 for the demonstration. Let assume an adversary probes the intermediate values sets of the output shares O and P_i in each gadget for $i \in [\![1; 6]\!]$. We use simulation sets S_i^j to simulate the values for each gadget. *t-SNI* security implies that if the size of all probing sets P_i is $t_I \leq t$ and if the size of values required to simulate in each gadget is smaller than t, then the simulation sets linked to the input shares are not bigger than t_I. As **SetExponentZero**, **RemoveDecimal**, **SecAnd**, **A2B** and **Refresh** are all *t-SNI* secure while **SecFprNorm64** is *t-NI* secure, we can sequentially derive the following:

- $|S_1^1|, |S_1^2|, |S_1^3| \leq |P_1|$
- $|S_2^1|, |S_2^2| \leq |P_2| + |O_{(my)}|$
- $|S_3^1|, |S_3^2|, |S_3^3|, |S_3^4| \leq |P_3|$
- $|S_4^1|, |S_4^2| \leq |P_4|$
- $|S_5| \leq |P_5|$
- $|S_6| \leq |P_6|$

Based on the previous inequalities, we know that no gadget requires more than $t_I + |O_{(my)}| \leq t$ values to be simulated. The above method is also applied to the input shares, with $|S_4^1| \leq |P_4|$ for (my), $|S_5 \cup S_3^2 \cup S_6| \leq |P_5| + |P_3| + |P_6|$ for (ey) and $|S_3^4 \cup S_1^3| \leq |P_3| + |P_1|$ for (sy), none being more than t_I. □

5.2 Inverse

Lemma 4. *The gadget **SecFprComp** (Algorithm 10) is* t-SNI *secure.*

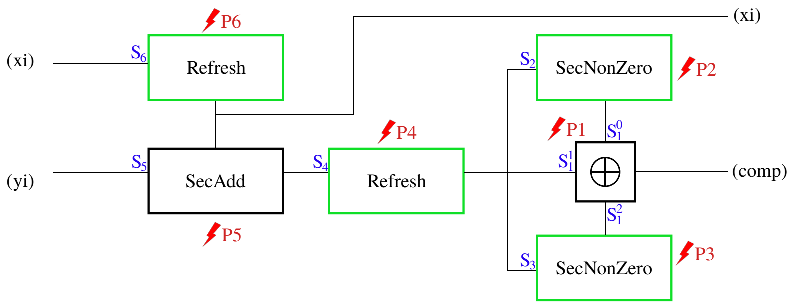

Figure 5: Abstract diagram of SecFprComp

Proof. We use an abstract diagram in Figure 5 for our demonstration. This gadget is similar to the swap part of the **SecFprAdd** gadget from [CC24] (Theorem 3, first part of the proof). We add some **Refresh** to ensure the *t-SNI* property. The **XOR** associated to the probing set P_1 is *t-NI* secure as this linear operation is performed on each share separately. The gadget **SecAdd** associated to the probe P_5 is also *t-NI* secure. The other gadgets are *t-SNI* secure. Hence, we have the following inequalities:

- $|S_1^0|, |S_1^1|, |S_1^2| \leq |P_1| + |O_{(comp)}|$
- $|S_2| \leq |P_2|$
- $|S_3| \leq |P_3|$
- $|S_4| \leq |P_4|$
- $|S_5^0|, |S_5^1| \leq |P_5| + |S_4| \leq |P_5| + |P_4|$
- $|S_6| \leq |P_6|$

According to these inequalities, no gadget requires more than $t_I + |O_{(comp)}| \leq t$ values to be simulated. This method can be applied to the input shares: For (x_i), we have $|S_6| \leq |P_6| \leq t_I$ and for (y_i) we have $|S_5^0| \leq |P_5| + |P_4| \leq t_I$. □

Lemma 5. *The gadget **SecFprScalePow2** (Algorithm 11) is* t-SNI *secure.*

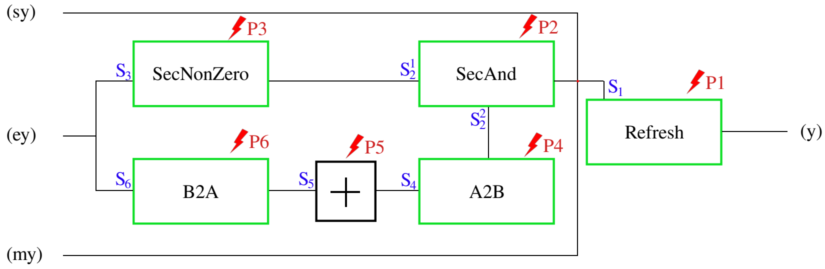

Figure 6: Abstract diagram of SecFprScalePow2

Proof. We consider an abstract diagram in Figure 6 for our demonstration. This gadget mainly affects the exponent shares (ey). Apart from "+" which is *t-NI* as it is simply adding a constant to one share, all other gadgets are *t-SNI*. As the single input of the gadget "+" comes from a *t-SNI* gadget **B2A** and then has its single output fed into another *t-SNI* gadget, the chain **B2A** $\rightarrow$ " + " $\rightarrow$ **A2B** is itself *t-SNI*. By composition, the entire gadget is *t-SNI*. □

Theorem 2. *The gadget **SecFprInv** (Algorithm 12) is* t-SNI *secure.*

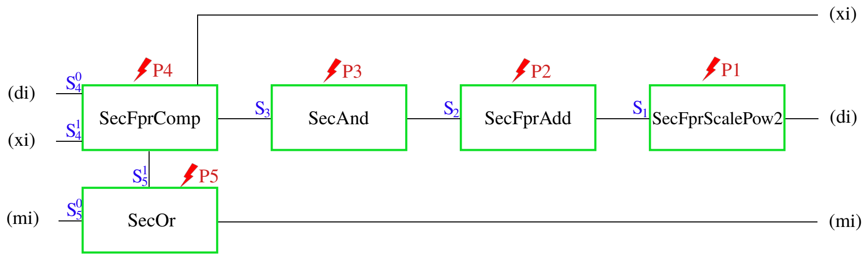

Figure 7: Abstract diagram of LOOP

Proof. We base our demonstration on an abstract diagram in Figure 8. We first prove that the gadget **LOOP** associated to the probes set P_5 is *t-SNI* secure.
We use an abstract diagram in Figure 7 for our demonstration. This gadget composes *t-SNI* gagdets, including **SecFprComp** and **SecFprScalePow2**, proven *t-SNI* in Lemmas 4 and 5. As the first iteration of the loop is *t-SNI* secure by composition, and the loop cycles on itself, all remaining iterations are also *t-SNI* secure. This implies the gadget **LOOP** is itself *t-SNI* secure.
For the rest of the **SecFprInv** gadget, all gadgets are *t-SNI* apart from + associated to the probes set P_7 and **SecAdd** associated to the probes set P_1. We can derive the following:

- $|S_1| \leq |P_1| + |O_{(x\_inv)}|$
- $|S_2^0|, |S_2^1| \leq |P_2|$
- $|S_3^0|, |S_3^1| \leq |P_3|$
- $|S_4| \leq |P_4|$
- $|S_5^0|, |S_5^1| \leq |P_5|$
- $|S_6^0|, |S_6^1|, |S_6^2| \leq |P_6|$
- $|S_7| \leq |P_7|$
- $|S_8| \leq |P_8|$
- $|S_9^0|, |S_9^1| \leq |P_9| + |S_2| + |S_8| \leq |P_9| + |P_2| + |P_8|$
- $|S_{10}| \leq |P_{10}|$
- $|S_{11}| \leq |P_{11}|$
- $|S_{12}| \leq |P_{12}|$

Based on these inequalities, we know that no gadgets requires more than $t_I + |O_{(x\_inv)}| \leq t$ values to be simulated. This method can also be applied to the input shares: For (xi) we have $|S_{11} \cup S_6^1| \leq |P_{11}| + |P_6| \leq t_I$, for (exi) we have $|S_9^1| \leq |P_9| + |P_8| + |P_2| \leq t_I$, for (sxi) we have $|S_{12}| \leq |P_{12}| \leq t_I$ and for (mi) we have $|S_6^2| \leq |P_6| \leq t_I$. □

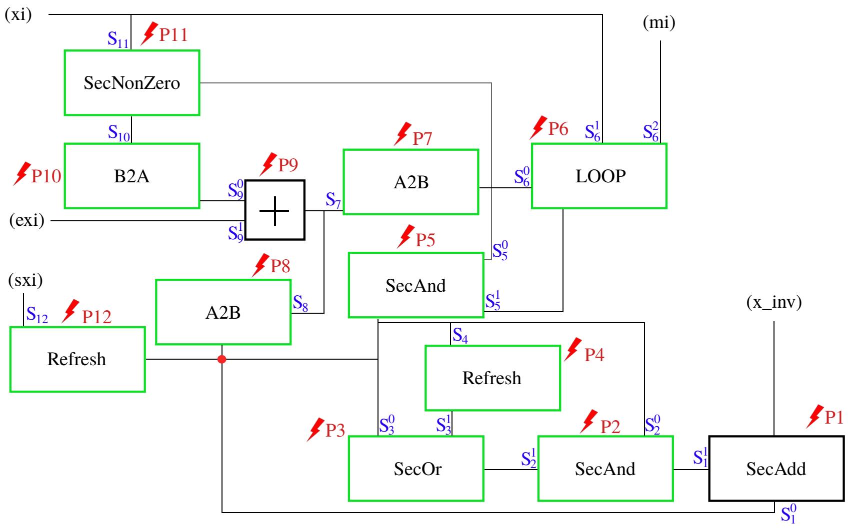

Figure 8: Abstract diagram of SecFprInv

Lemma 6. *The gadget* ***Minimum63*** *(Algorithm 13) is* t-SNI *secure.*

Proof. The Minimum63 algorithm is composed only of *t-SNI* gadgets, namely A2B, SecAnd and SecOr. It is thus itself *t-SNI*. □

6 Performances

Some results are shown in Table 4. This implementation is not optimized and is realized with a laptop computer equipped with an Intel Core i7-11800H CPU. The compiler used is *gcc version 9.4.0* with options *-O3*. We have considered our performances of SecFprAdd and SecFprMul as reference and compare our work with the one of Chen and Chen [CC24], as they used a different hardware (Intel Core i9-12900KF). We have designed our code around 3 shares and some well-known optimizations for 2 shares masking have not been implemented. Hence, we observe that the complexity increases linearly with the number of shares.

Table 4: Time in microseconds

Algorithm	Unmasked [PFH+20]	2 Shares	3 Shares
SecFprAdd [CC24]	0.000 11	7.533	13.552
SecFprMul [CC24]	0.000 14	5.563	11.622
$\text{SecFprBaseInt}_{\text{floor}}$	0.000 136	7.084	13.284
$\text{SecFprUrsh}_{\text{floor}}$	-	0.113	0.219
SecFprInv	0.000 138	559.658	994.416
SecFprComp	-	1.601	2.471
SecFprScalePow2	-	0.943	1.903
ApproxExp	0.000 126	190.207	367.245
BerExp	0.005 446	227.187	441.951
SamplerZ	0.114	1807.353	4205.701
1024 SamplerZ	122.962	1 850 633	4 382 602
2048 SamplerZ	247.902	3 780 432	8 731 953

To replicate the performances of the calls to the Gaussian Sampler by FALCON, we performed SamplerZ by the same amount of iterations required in both FALCON-512 and FALCON-1024. Table 4 highlights the impact of the division computation on SamplerZ. The SecFprInv gadget is the main bottleneck of our design as it involves 55 SecFprAdd. On the other hand, our $\text{SecFprBaseInt}_{\text{floor}}$ gadget is no more costly than one SecFprAdd.

We also tested a masked complete version of FALCON. Its performances are summarized in Table 5. We do not perform the signature rejection. Thus, in a real world use case, the performances might be doubled. Our results clearly highlight that this masking methodology for FALCON is not ready for a deployment.

Table 5: Masked FALCON in seconds

FALCON	FFSampling	Compress	Preimage	Total
FALCON 512 (2 shares)	3.157 130	0.001 258	0.040 156	3.198 545
FALCON 512 (3 shares)	6.284 270	0.002 396	0.081 091	6.367 758
FALCON 1024 (2 shares)	6.825 461	0.002 594	0.080 565	6.908 620
FALCON 1024 (3 shares)	12.759 945	0.004 814	0.162 189	12.926 950

7 Conclusion

In this paper we have extended the work of Chen and Chen [CC24] and have used their gadgets and our new own gadgets to mask the floor function (Section 3). The Gaussian sampler of FALCON (Section 4) has been protected with this floor gadget. Additionally, to reach this task, we provided a masked implementation of the division (Section 4). We discussed about the *t-SNI* properties of our gadgets (Section 5). Finally, we provided some performances got on a laptop computer equipped with an Intel Core CPU (Section 6), highlighting the non-readiness state of this masking methodology for real world deployment. Future works could investigate better masking methodologies and/or algorithmic improvements. For instance, reducing the division's cost should lead to better performances, as it is the main bottleneck in our current design. New masking methods for floating-point arithmetic, less reliant on A2B and B2A conversions, could be studied and offer better performances. Other representations than Binary64 could also be of interest but should first be allowed in the FALCON standard. Finally, fault-injection resilient designs could be of interest.

Acknowlegdments We would like to thank Ken-Yu Chen and Jiun-Peng Chen who responded to our questions regarding their work.
This work thanks grant 2022156 and grant 2023151 from the Appel à projets 2022 and Appel à projets 2023 thèses AID CIFRE-Défense by the Agence de l'Innovation de Défense (AID), Ministère des Armées (French Ministry of Defense).
This paper is also part of the on-going work of Hensoldt SAS France for the Appel à projets Cryptographie Post-Quantique launched by Bpifrance for the Stratégie Nationale Cyber (France National Cyber Strategy) and Stratégie Nationale Quantique (France National Quantum Strategy). In this, Hensoldt SAS France is a part of the X7-PQC project in partnership with Secure-IC, Télécom Paris and Xlim.

References

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 116–129, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978427.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the glp lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 354–384, Cham, 2018. Springer International Publishing.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from boolean to arithmetic masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):22–45, May 2018. URL: https://tches.iacr.org/index.php/TCHES/article/view/873, doi:10.13154/tches.v2018.i2.22-45.

[BDK+18] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. Crystals - kyber: A cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367, April 2018. doi:10.1109/EuroSP.2018.00032.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The sphincs+ signature framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2129–2146, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3363229.

[CC24] Keng-Yu Chen and Jiun-Peng Chen. Masking floating-point number multiplication and addition of falcon: First- and higher-order implementations and evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):276–303, Mar. 2024. URL: https://tches.iacr.org/index.php/TCHES/article/view/11428, doi:10.46586/tches.v2024.i2.276-303.

[CCJ+16] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray A Perlner, and Daniel Smith-Tone. *Report on post-quantum cryptography*, volume 12. US Department of Commerce, National Institute of Standards and Technology ..., 2016.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In Gregor Leander, editor, *Fast Software Encryption*, pages 130–149, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, Feb. 2018. URL: https://tches.iacr.org/index.php/TCHES/article/view/839, doi:10.13154/tches.v2018.i1.238-268.

[EFG+22] Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: A simpler, parallelizable, maskable variant of falcon. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 222–253, Cham, 2022. Springer International Publishing.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on bliss lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1857–1874, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3134028.

[GBHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload – a cache attack on the bliss lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 323–345, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[GMRR22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):141–164, Jun. 2022. URL: https://tches.iacr.org/index.php/TCHES/article/view/9697, doi:10.46586/tches.v2022.i3.141-164.

[HPRR20] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous gaussian sampling: From inception to implementation. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 53–71, Cham, 2020. Springer International Publishing.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[KA21] Emre Karabulut and Aydin Aysu. Falcon down: Breaking falcon post-quantum signature scheme through side-channel attacks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 691–696, Dec 2021. doi:10.1109/DAC18074.2021.9586131.

[Kah96] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[MHS+19] Sarah McCarthy, James Howe, Neil Smyth, Seamus Brannigan, and Máire O'Neill. Bearz attack falcon: Implementation attacks with countermeasures on the falcon signature scheme. Cryptology ePrint Archive, Paper 2019/478, 2019. https://eprint.iacr.org/2019/478. URL: https://eprint.iacr.org/2019/478.

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[NIS24a] NIST. Module-lattice-based digital signature standard. *NIST FIPS*, 2024. doi:10.6028/NIST.FIPS.204.ipd.

[NIS24b] NIST. Module-lattice-based key-encapsulation mechanism standard. *NIST FIPS*, 2024. doi:10.6028/NIST.FIPS.203.ipd.

[NIS24c] NIST. Stateless hash-based digital signature standard. *NIST FIPS*, 2024. doi:10.6028/NIST.FIPS.205.ipd.

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To bliss-b or not to be: Attacking strongswan's implementation of post-quantum signatures. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1843–1855, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3134023.

[PFH+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. *Post-Quantum Cryptography Project of NIST*, 2020.

[RCDB24] Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (kyber, dilithium): Survey and new results. *ACM Trans. Embed. Comput. Syst.*, 23(2), mar 2024. doi:10.1145/3603170.

[Sho99] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999. arXiv:https://doi.org/10.1137/S0036144598347011, doi:10.1137/S0036144598347011.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography – PKC 2019*, pages 534–564, Cham, 2019. Springer International Publishing.

[ZLYW23] Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved power analysis attacks on falcon. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 565–595, Cham, 2023. Springer Nature Switzerland.