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

Justine Paillet[1,3][0009−0009−6056−7766], Pierre-Augustin Berthet[2,3][0009−0005−5065−2730], and Cédric Tavernier[3][0009−0007−5224−492X]

[1] Université Jean-Monnet, Saint-Étienne, France,
justine.paillet@univ-st-etienne.fr
[2] Télécom Paris, Palaiseau, France, berthet@telecom-paris.fr
[3] Hensoldt SAS FRANCE, Plaisir, France,
<pierre-augustin.berthet,justine.paillet,cedric.tavernier>@hensoldt.net

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.

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 [28]. 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 [7]. The finalists are CRYSTALS Kyber [5,22], CRYSTALS Dilithium [9,21], SPHINCS+ [3,23] and FALCON [25].

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 [18] 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 works have been done on CRYSTALS Dilithium and CRYSTALS Kyber, summed up by Ravi et al. [26], FALCON has been less covered. Indeed, the algorithm relies on floating-point arithmetic, for which there is little literature on how to protect it.

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 [17] using an Electro-Magnetic (EM) attack. Their work was later improved by Guerreau et al. [13]. To counter those attacks, Chen and Chen [6] 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 [12,10,20,24]. A isochronous design was proposed by Howe et al. [14] to counter those attacks. Nonetheless, a successful single power analysis (SPA) was proposed by Guerreau et al. [13] and further improved by Zhang et al. [29]. There is currently no masking countermeasure for FALCON's Gaussian Sampler. Existing work [11] tends to rewrite the Gaussian Sampler to remove the use of floating arithmetic, thus avoiding the challenge of masking the floor function.

Our Contribution In this work, we further expand the countermeasure from Chen and Chen [6] 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 [6], 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. [1].

We provide some performances of our methods and compare them with the reference unmasked implementation and the previous work of Chen and Chen [6]. The implementation is tested on a personal computer with an Intel-Core i7-11800H CPU and is not optimized.

2 Notation and Background

2.1 Notation

– 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 1-bit Boolean shares for value b, we denote $(-b_i)$ as the 64-bit Boolean shares for $2^{64} - b$. It means that if $b = 0$, $(-b_i)$ is a 64-bit boolean shares for 0, and $b = 1$, $(-b_i)$ is a 64-bit boolean shares for 0xFFFFFFFFFFFFFFFF.

For algorithmic extracts of FALCON [25], refer to 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 [25] 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[25].

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 $\boldsymbol{s} = (s_1, s_2)$ such as $s_1 + s_2 h = c$. To do so, the need to compute $\boldsymbol{s} = (\boldsymbol{t} - \boldsymbol{z})\mathbf{B}$, with $\boldsymbol{t}$ a pre-image vector and $\boldsymbol{z}$ provided by a Gaussian Sampler. Chen and Chen [6] focus on masking the pre-image vector computation. In this work, we intend to mask the Gaussian Sampler. The signature algorithm is detailed in [25] in the corresponding section.

Gaussian Sampler The Gaussian Sampler is the composition built from the following functions:

ApproxExp. This function return $2^{63} \times ccs \times e^{-x}$ and depends of a matrix C defined in page 42 of [25]:

Algorithm 1: ApproxExp(x,ccs) [25]

Data: Foating-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) [25]

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

$s \leftarrow \lfloor x / \ln(2) \rfloor$; `// Compute the unique decomposition` $x = \ln(2^s) + r$ `with` $(r, s) \in [0, \ln(2)) \times \mathbb{Z}^+$
$r \leftarrow x - s \cdot \ln(2)$;
$s \leftarrow \min(s, 63)$;
$z \leftarrow (2 \cdot \text{ApproxExp}(r, ccs) - 1) >> s$;
$i \leftarrow 64$;
do
  $i \leftarrow i - 8$;
  $w \leftarrow \text{UniformBits}(8) - ((z >> i)\ \&\ 0\text{xff})$;
while $((w = 0)\ and\ (i > 0))$;
return $[\![w < 0]\!]$;

SamplerZ. The Gaussian Sampler:

Algorithm 3: SamplerZ(μ,σ') [25]

Data: Foating-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 $D_{\mathbb{Z},\mu,\sigma'}$

$r \leftarrow \mu - \lfloor \mu \rfloor$;
$ccs \leftarrow \sigma_{min}/\sigma'$;
while *1* **do**
  $z_0 \leftarrow \text{BaseSampler}()$;
  $b \leftarrow \text{UniformBits}(8)\ \&\ 0\text{x1}$;
  $z \leftarrow b + (2 \cdot b - 1)z_0$;
  $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}}$;
  if $\text{BerExp}(x, ccs) = 1$ **then**
    return $z + \lfloor \mu \rfloor$;

Algorithm 4: BaseSampler() [25]

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 [25].

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.

Binary64 Encoding A floating-point [16] is encoded with a sign bit s, a 11-bits long exponent e and a 52-bits long mantissa m such as:

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

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 us to properly compute the floor function from the truncated one in this case.

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

This method requires the use of the exponent and the sign, which are both sensitive values. In this work, we propose a method to perform this truncation securely.

2.5 Masking

Masking is a generic countermeasure to 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 [19]. Masking security can be evaluated thanks to the *t-probing* model, first introduced in [15]. A gadget is then 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 [1]: the (Strong) Non-Interference model.

Definition 2. *(*t*-Non Interference (*t*-NI) security [1]). 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 [1]). 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 use those models in Section 5 to demonstrate the security of our design. We rely on existing gadgets and propose new ones, as shown in Table 2.5.

3 Masking the Floor Function

In Section 2.4 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. As only the floor is required to protect FALCON, we provide their pseudo-codes in Appendix B.

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

Algorithm	Description	Security	Reference
SecAnd	AND of Boolean shares	*t-SNI*	[1],[15]
SecAdd	Addition of Boolean shares	*t-SNI*	[2],[8]
A2B	Arithmetic to Boolean conversion	*t-SNI*	[27]
B2A	Boolean to Arithmetic conversion	*t-SNI*	[4]
RefreshMasks	t-NI refresh of masks	*t-NI*	[1], [4]
Refresh	t-SNI refresh of masks	*t-SNI*	[1]
SecOr	OR of Boolean shares	*t-SNI*	[6]
SecNonZero	NonZero check of shares	*t-SNI*	[6]
SecFprUrsh	Right-shift with sticky bit	*t-SNI*	[6]
SecFprNorm64	Normalization to $[2^{63}, 2^{64})$	*t-NI*	[6]
SecFprAdd	Floating addition	*t-SNI*	[6]
SecFprMul	Floating multiplication	*t-SNI*	[6]
SetExponentZero	Set exponent to zero	*t-SNI*	Algorithm 8
SecFprUrsh$_f$	Right-shift without sticky bit	*t-SNI*	Algorithm 7
RemoveDecimal	Truncate the mantissa	*t-SNI*	Algorithm 6
SecFprBaseInt	Compute the floor	*t-SNI*	Algorithm 5
SecFprScalePow2	Multiplies by a power of 2	*t-SNI*	Algorithm 11
SecFprComp	Compares two values	*t-SNI*	Algorithm 10
SecFprInv	Inversion	*t-SNI*	Algorithm 9

SecFprBaseInt$_\mathbf{f}$: The gadget SecFprBaseInt$_\mathrm{f}$ (Algorithm 5) is the main function of the masked floor, the masked *truncate*, and the masked *rounding*. Gadgets and Zero$_f$ are parametered[4] by these functions.

We now focus on f = floor. We first extract[5] the data from the encoding used by [6] and place into three variables s_y, e_y, and m_y, which are directly linked to the output of the algorithm.

We first check if $c_x = e_y - \mathrm{Zero}_f < 0$, corresponding to Equation 2. If cx is negative, $\mid x \mid < 1$ and all decimals can be removed by putting $my = 0$. The case $-1 < x < 0$ is solved by SetExponentZero (Algorithm 8) at the end of the algorithm. For the other cases, the mantissa is unchanged and we can cover the two remaining cases using RemoveDecimal (Algorithm 6).

[4] $\mathrm{Zero}_{\mathrm{floor}} = \mathrm{Zero}_{\mathrm{trunc}} = 1023$ and $\mathrm{Zero}_{\mathrm{round}} = 1022$
[5] Pseudo-code in Appendix: SecFprExtract – Algorithm 12

Algorithm 5: 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).

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

As the algorithm RemoveDecimal does not normalize the mantissa, then we apply SecFprNorm64 (see [6] Algorithm 10 page 286) and compute a shifted my and 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 applying the specific encoding of "0" if it is the expected result. To do this, we apply the SetExponentZero$_f$ function (Algorithm 8).

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

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

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

RemoveDecimal$_{\text{floor}}$: $cx = e_y - 1023 < 0$ (Equation 2) being covered, we use RemoveDecimal$_{floor}$ (Algorithm 6) for the two remaining cases[6], as described in Section 2.4. If $cx \geq 52$, then x is an integer as shown in Equation 4. To avoid information loss during the remainder of SecFprBaseInt, we replace cx by 0. Finally, if $0 \leq c_x \leq 51$, the mantissa must be truncated accordingly.

We use a modification of the SecFprUrsh method from [6] (Algorithm 9 page 286) to shift the mantissa my by $cd = 52 - cx$. Our method, SecFprUrsh$_{\text{f}}$ (Algorithm 7, does not keep the sticky bit but the removed part. Once the mantissa is shifted, we have consequently performed the $truncate(x)$ function. As described in Section 2.4, for the floor we also have to check whether the sign sy is 1. In that case, we check by applying SecNonZero, with result denoted b, if the removed part is 0. If so, we apply the floor function to a negative integer. Else, we have to retrieve 1 to the result in accordance with Section 2.4. We do so 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

SetExponentZero$_{\text{floor}}$: This last function (Algorithm 8) is necessary in the algorithm and uses the data collected throughout the calculations of the whole algorithm to modify ey and sy if the expected result is 0. The encoding of 0 is special because it is encoded by itself. The desired result is zero only if $\mid x \mid < 1$ and $sy = 0$. We remind that $floor(x) = -1$ if $sy = 1$ and $\mid x \mid < 1$. -1 is encoded as $sy = 1$, $ey = 1023$ and $my = 0$.

4 Application to Falcon : Gaussian Sampler

The floor function has been described above and we now address the SamplerZ function (Algorithm 3 or see [25] Algorithm 15 page 43). In the algorithms SamplerZ and BerExp (Algorithm 2 or see [25] 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, it is necessary to design a way to perform a division by an arbitrary value securely. To do so we choose to invert x and then compute a multiplication in

[6] First case is not affected by RemoveDecimal as mantissa is set to 0.

Algorithm 7: 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 $((my'_i),(rot_i))$;

Table 3. Encoding 0, minus 1 or others: Truth table

$-sy$	b	$-sy \vee b$	Interpretation
$0\cdots0$	$0\cdots0$	$0\cdots0$	"Small" positive number : $ey = 0$ and $sy = 0$
$1\cdots1$	$0\cdots0$	$1\cdots1$	"Small" negative number : $ey = 1023$ and $sy = 1$
$-sy$	$1\cdots1$	$01\cdots1$	Non zero number : $ey = ey$ and $sy = sy$

Algorithm 8: SetExponentZero$_{\text{floor}}((ey_i),(sy_i),(b_i))$

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

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

order to divide. Computing the inverse involves performing a Euclidean division until obtaining sufficient precision (55 bits) to construct it.

Division : Remind that a binary64 value is represented by a tuple of three elements (s, e, m), where s is a 1-bit sign, e a 11-bit exponent and m a 52-bit mantissa. We construct the result tuple (s_y, e_y, m_y) considering each element separately. Inverse operation preserves the sign, so $s_y = s_x$. Then we consider the exponent. It consists in finding how many shift we require before starting to subtract information, hence we calculate $c_x = e_x - 1023$. If the mantissa is zero, then x is a power of 2 and $1 << c_x = x$, enabling us to perform the only subtraction necessary for the inverse calculation. However, if $m_x \neq 0$, $(1 << c_x) < x$, an additional shift is required. This is the reason for which we compute b = SecNonZero(m_x) and write the final formula for c_x as $c_x = e_x - 1023 + b$. By inversing this operation, we finally get:

$$e_{\text{inv}} = 1023 - c_x = 2046 - e_x - b$$

The last step consists in finding the 52 bit-length mantissa. This part essentially corresponds to the Euclidean division: first, we compare our dividend $d = (1 << cx)$ to x, by computing $comp = \text{SecFprComp}(d, x)$ (Algorithm 10). If $x < d$, then comp = 1, and this value needs to be carried over to the new mantissa. We then add $-x$ (just a sign modification) to d. If $comp = 0$, then we replace x by 0 applying SecAnd$(x, comp)$ and performing an addition with d, which does not alter the result. Finally, we shift d one time to the left to continue the division. After computing 53 bits (52 plus implicit bit) we calculate two additional bits, totaling 55 bits, to preserve the sticky bit. The method for calculating the sticky bit is available in the pseudo-code and is directly derived from SecFpr (see [6] Algorithm 11 page 287).

Comparison. In the inversion process, we must compare two masked binary64 values. We adapt the swap part of the SecFprAdd function (see [6] Algorithm 13 page 290). We add refresh operations to make the gadget t-SNI.

Minimum and Shift. BerExp (Alogirthm 2 or see [25] Algorithm 14 page 43) must be briefly described. It requires to multiply a number by 2^{-s}, with s masked after taking the minimum between s and 63. s must not be greater than 63 to retain its value, we can simply check if the exponent of s is less than 1029, same as checking if $(s_i^{(63)})$ is equal to 1.

To perform a shift, we extract the integer value of the masked s, then it is injected into the $\text{SecFprUrsh}_\text{f}$ function (Algorithm 7). More precisely we extract the mantissa of s, to which we add the implicit bit (unless $s = 0$), and shift it by 46 to obtain the bits (information is in 6 bits). Then we shift again using $\text{SecFprUrsh}_\text{f}$ by $es - 1023$($1023 <= es <= 1029$ or $es = 0$). Finally we subtract the integer s from the exponent of the z that must be shifted.

Algorithm 9: 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($(xcpy_i), (d_i)$);
  $(d_i) \leftarrow$ SecFprScalPtwo($(d_i), 1$);
$(my_i) \leftarrow$ 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);

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

Our function SecFprScalPow2 (Algorithm 11) handles the case of shifting by an unmasked integer value by simply performing a left shift if pow_2 is positive and a right shift if it is negative, then we remove the useless information applying the truncation function [7]. Using this algorithm, division and multiplication by two are cheaper to perform.

Algorithm 11: SecFprScalPow2$((x_i), p)$

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 $x \times 2^p$
1 $(sx_i), (ex_i), (mx_i) \leftarrow$ SecFprExtract$((x_i))$;
2 $(b_i) \leftarrow$ SecNonZero$((x_i))$;
3 $(ex_i) \leftarrow$ B2A$((ex_i))$;
4 $ex_1 \leftarrow ex_1 + p$;
5 $(ex_i) \leftarrow$ A2B$((ex_i))$;
6 $(ey_i) \leftarrow$ SecAnd$((ex_i), -(b_i))$;
7 $(y_i^{(64)}) \leftarrow (sy_i), (y_i^{[63:53]}) \leftarrow (ey_i), (y_i^{[53:1]}) \leftarrow (my_i)$;
8 **return** *Refresh*(y_i);

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 [6] for our proofs. We aim to propose only *t-SNI* secure gadgets as they are composable. It limits the risks of compositional flaws. We are aware that it leads to performance overheads and more demanding randomness requirements.

5.1 Floor Function

Lemma 1. *The gadget* ***SetExponentZero$_{floor}$*** *(Algorithm 8) 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$_{floor}$*** *(Algorithm 7) is* t-SNI *secure.*

Proof. The gadget **SecFprUrsh$_{floor}$** is a slight modification of the gadget **SecFprUrsh** from [6]. 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

[7] The pseudo-code is provided in Appendix B

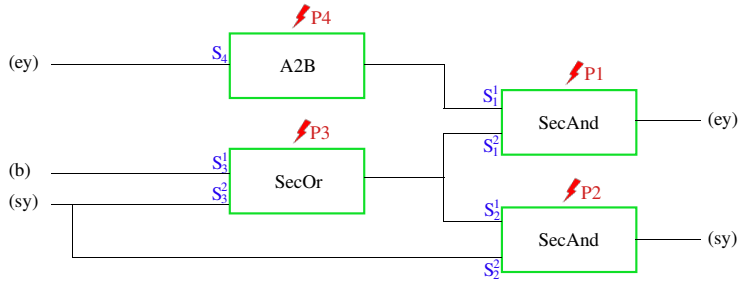

Fig. 1. Abstract diagram of SetExponentZero$_{floor}$

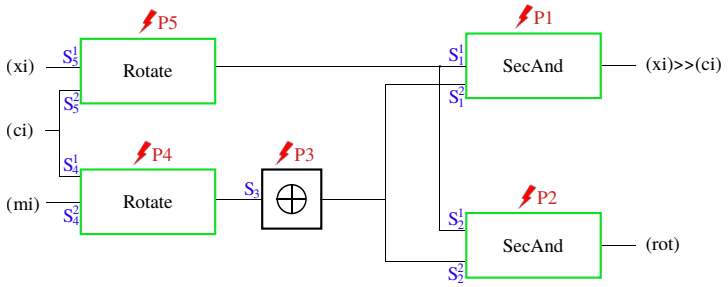

Fig. 2. Abstract diagram of SecFprUrsh$_{floor}$

the gadget **Rotate** (see [6], Lemma 3 and Figure 2). We now show that the operations below the rotation loop are *t-SNI* secure. We use an abstract diagram in Figure 2 for the demonstration. Let an adversary probes 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 ([6], 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

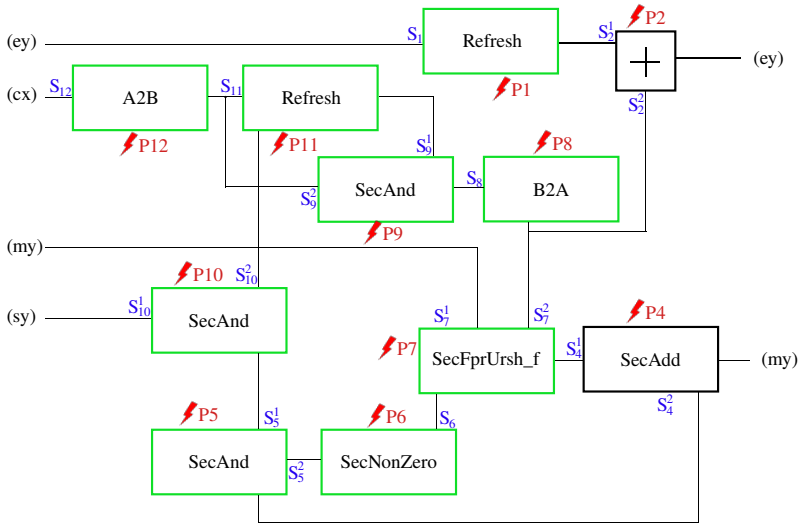

Fig. 3. Abstract diagram of $\text{RemoveDecimal}_{\text{floor}}$

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**, $\textbf{SecFprUrsh}_{floor}$, **B2A** and **A2B** are all *t-SNI* secure whereas **SecAdd** and "+" are *t-NI* secure, we can sequentially derive the following:

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

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* $\textbf{SecFprBaseInt}_{\textbf{floor}}$ *(Algorithm 5) is* t-SNI *secure.*

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

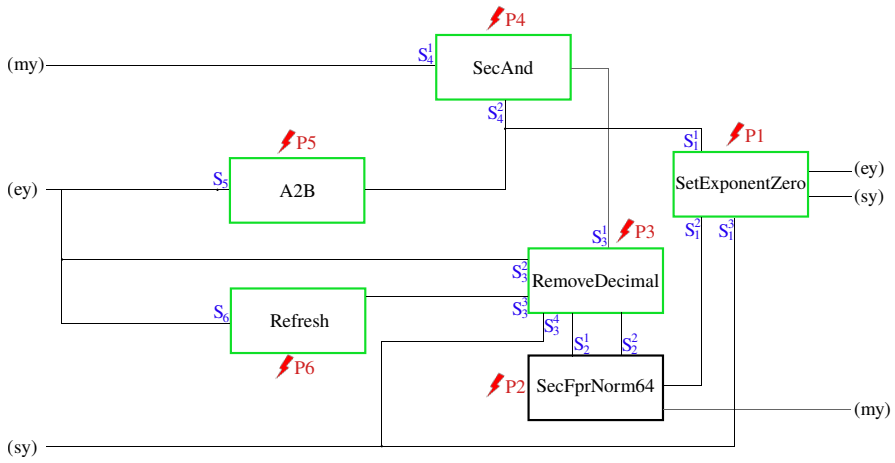

Fig. 4. Abstract diagram of SecFprBaseInt$_{\text{floor}}$

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.*

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 [6] (Theorem 3, first part of the proof). We add some **Refresh** to ensure the *t-SNI* property. Note that 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:

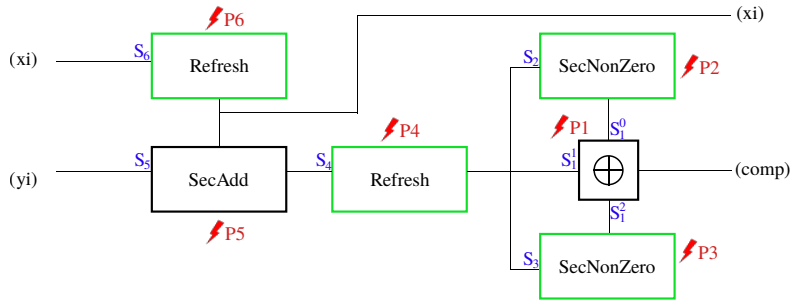

Fig. 5. Abstract diagram of SecFprComp

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

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.*

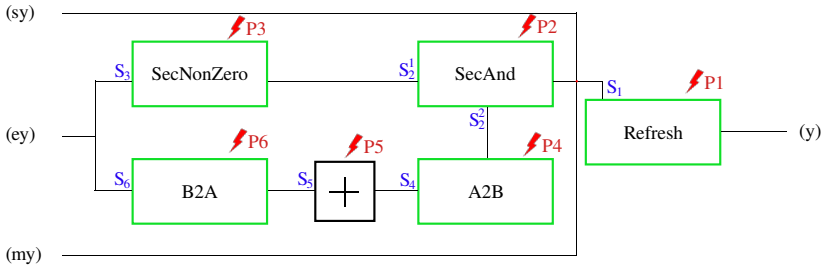

Fig. 6. Abstract diagram of SecFprScalePow2

Proof. We use 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 9) is* t-SNI *secure.*

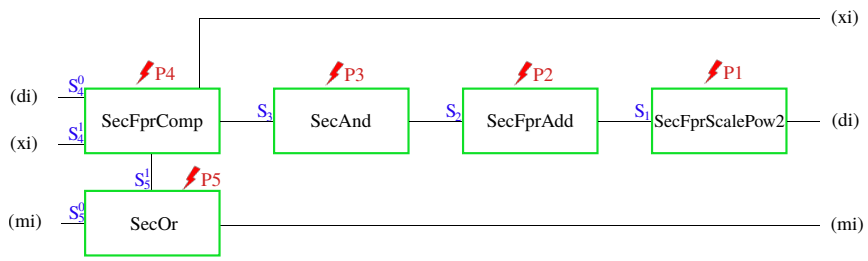

Fig. 7. Abstract diagram of LOOP

Proof. We use an abstract diagram in Figure 8 for our demonstration. 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 $|S_6^2| \leq |P_6| \leq t_I$. □

6 Performances

Some results are shown in Table 4. This implementation is not optimized and is realized with a personal 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 [6], as they used a different hardware (Intel

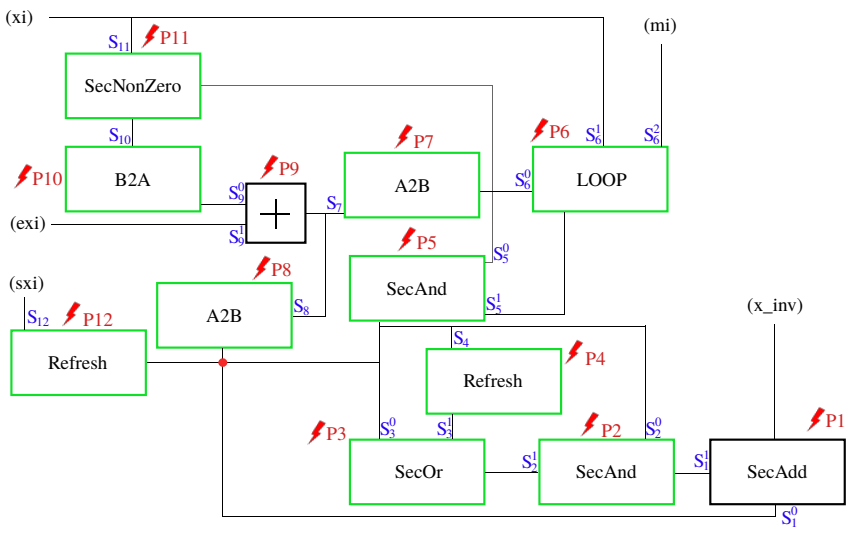

Fig. 8. Abstract diagram of SecFprInv

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	[25]	**2 Shares**	**3 Shares**
SecFprAdd [6]	0.000 11	7.533	13.552
SecFprMul [6]	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
SecFprScalPwo2	-	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 [8]	122.962	1 850 633	4 382 602
2048 SamplerZ [9]	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 SecFprBaseInt$_{\text{floor}}$ gadget is no more costly than one SecFprAdd.

7 Conclusion

In this paper we have extended the work of Chen and Chen [6] 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 personal computer equipped with an Intel Core CPU (Section 6).
Future works could lead to a complete masked implementation of the FALCON signature relying both on Chen and Chen [6] and our work. Improving the division 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. 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

1. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P.A., Grégoire, B., Strub, P.Y., Zucchini, R.: Strong non-interference and type-directed higher-order masking. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. p. 116–129. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978427, https://doi.org/10.1145/2976749.2978427
2. Barthe, G., Belaïd, S., Espitau, T., Fouque, P.A., Grégoire, B., Rossi, M., Tibouchi, M.: Masking the glp lattice-based signature scheme at any order. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018. pp. 354–384. Springer International Publishing, Cham (2018)
3. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs+ signature framework. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 2129–2146. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363229, https://doi.org/10.1145/3319535.3363229
4. Bettale, L., Coron, J.S., Zeitoun, R.: Improved high-order conversion from boolean to arithmetic masking. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(2), 22–45 (May 2018). https://doi.org/10.13154/tches.v2018.i2.22-45, https://tches.iacr.org/index.php/TCHES/article/view/873
5. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehle, D.: Crystals - kyber: A cca-secure module-lattice-based kem. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 353–367 (April 2018). https://doi.org/10.1109/EuroSP.2018.00032
6. Chen, K.Y., Chen, J.P.: 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). https://doi.org/10.46586/tches.v2024.i2.276-303, https://tches.iacr.org/index.php/TCHES/article/view/11428
7. Chen, L., Chen, L., Jordan, S., Liu, Y.K., Moody, D., Peralta, R., Perlner, R.A., Smith-Tone, D.: Report on post-quantum cryptography, vol. 12. US Department of Commerce, National Institute of Standards and Technology ... (2016)
8. Coron, J.S., Großschädl, J., Tibouchi, M., Vadnala, P.K.: Conversion from arithmetic to boolean masking with logarithmic complexity. In: Leander, G. (ed.) Fast Software Encryption. pp. 130–149. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
9. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(1), 238–268 (Feb 2018). https://doi.org/10.13154/tches.v2018.i1.238-268, https://tches.iacr.org/index.php/TCHES/article/view/839
10. Espitau, T., Fouque, P.A., Gérard, B., Tibouchi, M.: 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. p. 1857–1874. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134028, https://doi.org/10.1145/3133956.3134028
11. Espitau, T., Fouque, P.A., Gérard, F., Rossi, M., Takahashi, A., Tibouchi, M., Wallet, A., Yu, Y.: Mitaka: A simpler, parallelizable, maskable variant of falcon. In:

Dunkelman, O., Dziembowski, S. (eds.) Advances in Cryptology – EUROCRYPT 2022. pp. 222–253. Springer International Publishing, Cham (2022)
12. Groot Bruinderink, L., Hülsing, A., Lange, T., Yarom, Y.: Flush, gauss, and reload – a cache attack on the bliss lattice-based signature scheme. In: Gierlichs, B., Poschmann, A.Y. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2016. pp. 323–345. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
13. Guerreau, M., Martinelli, A., Ricosset, T., Rossi, M.: 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). https://doi.org/10.46586/tches.v2022.i3.141-164, https://tches.iacr.org/index.php/TCHES/article/view/9697
14. Howe, J., Prest, T., Ricosset, T., Rossi, M.: Isochronous gaussian sampling: From inception to implementation. In: Ding, J., Tillich, J.P. (eds.) Post-Quantum Cryptography. pp. 53–71. Springer International Publishing, Cham (2020)
15. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003. pp. 463–481. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
16. Kahan, W.: Ieee standard 754 for binary floating-point arithmetic. Lecture Notes on the Status of IEEE **754**(94720-1776), 11 (1996)
17. Karabulut, E., Aysu, A.: Falcon down: Breaking falcon post-quantum signature scheme through side-channel attacks. In: 2021 58th ACM/IEEE Design Automation Conference (DAC). pp. 691–696 (Dec 2021). https://doi.org/10.1109/DAC18074.2021.9586131
18. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96. pp. 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
19. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks: Revealing the secrets of smart cards, vol. 31. Springer Science & Business Media (2008)
20. McCarthy, S., Howe, J., Smyth, N., Brannigan, S., O'Neill, M.: 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, https://eprint.iacr.org/2019/478
21. NIST: Module-lattice-based digital signature standard. NIST FIPS (2024). https://doi.org/10.6028/NIST.FIPS.204.ipd
22. NIST: Module-lattice-based key-encapsulation mechanism standard. NIST FIPS (2024). https://doi.org/10.6028/NIST.FIPS.203.ipd
23. NIST: Stateless hash-based digital signature standard. NIST FIPS (2024). https://doi.org/10.6028/NIST.FIPS.205.ipd
24. Pessl, P., Bruinderink, L.G., Yarom, Y.: 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. p. 1843–1855. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134023, https://doi.org/10.1145/3133956.3134023
25. Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon. Post-Quantum Cryptography Project of NIST (2020)
26. Ravi, P., Chattopadhyay, A., D'Anvers, J.P., Baksi, A.: 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). https://doi.org/10.1145/3603170, https://doi.org/10.1145/3603170

27. Schneider, T., Paglialonga, C., Oder, T., Güneysu, T.: Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In: Lin, D., Sako, K. (eds.) Public-Key Cryptography – PKC 2019. pp. 534–564. Springer International Publishing, Cham (2019)

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

29. Zhang, S., Lin, X., Yu, Y., Wang, W.: Improved power analysis attacks on falcon. In: Hazay, C., Stam, M. (eds.) Advances in Cryptology – EUROCRYPT 2023. pp. 565–595. Springer Nature Switzerland, Cham (2023)

A SecFprExtract

Algorithm 12: 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.

1 $(mx_i) \leftarrow (x_i^{[52:1]})$;
2 $(mx_i) \leftarrow \text{SecAdd}((mx_i), (2^{52}, 0, \cdots, 0))$; `// add implicit bit in the mantissa`
3 $(ex_i) \leftarrow (x_i^{[63:53]})$;
4 $(ex_i) \leftarrow \text{B2A}((ex_i))$;
5 $(sx_i) \leftarrow (x_i^{(64)})$;
6 **return** $((mx_i), (ex_i), (sx_i))$;

This algorithm provides us the extraction of (sx, ex, mx) by converting the 11-bit boolean shares ex into 16-bit arithmetic shares and adding the implicit bit to the mantissa mx. This bit which is not present in the x information (in order to save one bit and gain in precision), is nevertheless very important. It especially enables us to normalize our shares correctly.

B Alternate Algorithms For Truncate and Rounding

B.1 Truncature Function: Gadgets

Algorithm 13: RemoveDecimal$_{\text{trunc}}((my_i),(ey_i),(sy_i),(cx_i))$

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

1 $cx_1 \leftarrow cx_1 - 52$; // check if 0≤c<51
2 $(c_i) \leftarrow \text{A2B}((cx_i))$;
3 $(cp_i) \leftarrow (c_i^{(16)})$;
4 $\text{Refresh}((c_i))$;
5 $(c'_i) \leftarrow (-cp_i)$; // if cp = 0 cx = 0. if not cx = cx
6 $(c_i) \leftarrow \text{SecAnd}((c_i),(cp_i))$;
7 $(cx_i) \leftarrow \text{B2A}((c_i))$;
8 $(cd_i) \leftarrow (-cx_i)$;
9 $(my_i) \leftarrow \text{SecFprUrsh}_f((my_i),(cd_i))$; // $my >> 52-cx$
10 $(ey_i) \leftarrow (\text{Refresh}(ey_i)+cd_i)$;
11 **return** $((my_i),(ey_i))$;

Algorithm 14: SetExponentZero$_{\text{trunc}}((ey_i),(sy_i),(b_i))$

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

1 $(ey_i) \leftarrow \text{A2B}((ey_i))$;
2 $(ey_i) \leftarrow \text{SecAnd}((ey_i,b_i))$;
3 $(sy_i) \leftarrow \text{SecAnd}((sy_i,b_i))$;
4 **return** $((ey_i),(sy_i))$;

For truncature function, 0 is the result if $ex-1023<0$. So SetExponentZero$_{\text{trunc}}$ sets ey and sy to 0 depending only on (b_i), the zero condition.

B.2 Rounding Function: Gadgets

Algorithm 15: RemoveDecimal$_{\text{round}}((my_i),(ey_i),(sy_i),(cx_i))$

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

1 $cx_1 \leftarrow cx_1 - 53$; // check if 0≤c<51
2 $(c_i) \leftarrow \text{A2B}((cx_i))$;
3 $(cp_i) \leftarrow (c_i^{(16)})$;
4 $(rshORnot_i) \leftarrow$Refresh$(-cp[i])$;
5 $(c'_i) \leftarrow (-cp_i)$; // if cp = 0 cx = 0. if not cx = cx
6 $cx_1 \leftarrow cx_1 + 1$;
7 $(c_i) \leftarrow \text{A2B}((cx_i))$;
8 $(c_i) \leftarrow \text{SecAnd}((c_i),(cp_i))$;
9 $(cx_i) \leftarrow \text{B2A}((c_i))$;
10 $(cd_i) \leftarrow (-cx_i)$;
11 $(my_i) \leftarrow \text{SecFprUrsh}((my_i),(cd_i))$; // $my >> 53 - cx$
12 $(Rnd_i) \leftarrow (my_i^{(1)})$;
13 $(Rnd_i) \leftarrow \text{SecAnd}((Rnd_i),(rshORnot_i))$;
14 $(my1_i) \leftarrow (my_i) >> 1$, $(e1_i) \leftarrow (1, 0, \cdots, 0)$;
15 $(my2_i) \leftarrow (my_i)$, $(e2_i) \leftarrow (0, \cdots, 0)$;
16 $(my1_i) \leftarrow \text{SecAnd}((my1_i),(rshORnot_i))$;
17 $(e1_i) \leftarrow \text{SecAnd}((e1_i),(rshORnot_i))$;
18 $(rshORnot_i) \leftarrow (-rshORnot_i)$;
19 $(my2_i) \leftarrow \text{SecAnd}((my2_i),(rshORnot_i))$;
20 $(e2_i) \leftarrow \text{SecAnd}((e2_i),(rshORnot_i))$;
21 $(my_i) \leftarrow \text{SecOr}((my1_i),(my2_i))$; // choice of my
22 $(my_i) \leftarrow \text{SecAdd}((my_i),(Rnd_i))$;
23 $(e1_i) \leftarrow \text{SecOr}((e1_i),(e2_i))$; // choice of ey
24 $(e1_i) \leftarrow \text{B2A}((e1_i))$;
25 $(ey_i) \leftarrow (ey_i + e1_i)$;
26 $(ey_i) \leftarrow (ey_i + cd_i)$;
27 **return** $((my_i),(ey_i),(Rnd_i))$;

RemoveDecimal$_{\text{round}}$ provides us an algorithm able to remove decimals and round up or down if necessary. It's important to compare cx to 53 instead of 52. The reason is that we first subtract to ey 1022 instead of 1023 when we were checking if the result was 0 in Algorithm 5.

Remark 3. Line 12-25 can be replace by using the last bit of removed values, after checking if a rounding must be done (check if $rshOrNot = 1 \cdots 1$).

Algorithm 16: SetExponentZero$_{\text{round}}((ey_i),(sy_i),(b_i),(Rnd_i))$

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

1 $(ey_i) \leftarrow \text{A2B}((ey_i))$;
2 $(b'_i) \leftarrow (Rnd_i)$;
3 $(b'_i) \leftarrow \text{SecOr}((b'_i),(b_i))$;
4 $(ey_i) \leftarrow \text{SecAnd}((ey_i, b'_i))$;
5 $(sy_i) \leftarrow \text{SecAnd}((sy_i, b'_i))$;
6 **return** $((ey_i),(sy_i))$;

B.3 Performances

Table 5. Time in microseconds

Algorithm	**2 Shares**	**3 Shares**
SecFprAdd [6]	7.533467	13.552070
SecFprMul [6]	5.563748	11.622864
SecFprBaseInt$_{\text{floor}}$	7.084196	13.284748
SecFprBaseInt$_{\text{truncate}}$	5.502315	11.367418
SecFprBaseInt$_{\text{rounding}}$	6.960874	13.404681
SecFprUrsh$_{\text{trunc}}$	0.095197	0.172693
SecFprUrsh$_{\text{floor}}$	0.113149	0.219650