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

Pierre-Augustin Berthet[1,3], Justine Paillet[2,3], Cédric Tavernier[3], Lilian Bossuet[2] and Brice Colombier[2]

[1] Institut Polytechnique de Paris, Télécom Paris, LTCI, Palaiseau, France
[2] Université Jean-Monnet Saint-Étienne, CNRS, Institut d'Optique Graduate School, Laboratoire Hubert Curien UMR 5516, Saint-Étienne, France
[3] Hensoldt France SAS, Plaisir, France

Abstract. FALCON is a signature selected 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. Although recent work proposed a solution to mask the addition and the multiplication, some roadblocks remain, most noticeably, how to protect the floor function. In this work, we propose to complete the first existing tests of hardening FALCON against SCA. We perform the mathematical proofs of our methods as well as formal security proofs in the probing model by ensuring Multiple Input Multiple Output Strong Non-Interference (MIMO-SNI) security. We provide performances on a laptop computer of our gadgets as well as of a complete masked FALCON. We notice significant overhead in doing so and discuss the deployability of our method in a real-world context.

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

1 Introduction

With the rise of quantum computing, mathematical problems that were hard to solve with current technologies will be easier to break. Among the concerned problems, the Discrete Logarithm Problem (DLP) could be solved in polynomial time 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 standardisation 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-1990s. 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.

E-mail: berthet@telecom-paris.fr (Pierre-Augustin Berthet), justine.paillet@univ-st-etienne.fr (Justine Paillet), cedric.tavernier@hensoldt.net (Cédric Tavernier), lilian.bossuet@univ-st-etienne.fr (Lilian Bossuet), b.colombier@univ-st-etienne.fr (Brice Colombier)

The study of weaknesses in the implementations of new primitives and the way to protect them is an active field of research.

Although many efforts have been made to protect CRYSTALS Dilithium and CRYSTALS Kyber, summarised by Ravi et al. [RCDB24], FALCON has been less covered. The algorithm relies on floating-point arithmetic, for which there is little literature on how to protect it.

1.1 Related Work

Previous works have identified two main weaknesses within the Falcon signing process: the preimage computation and the Gaussian sampler. The latter 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 these attacks, Chen and Chen [CC24] propose an implementation of the addition and multiplication of FALCON using a generic side-channel countermeasure, namely masking. Karabulut and Aysu [KA24] proposed a similar approach to mask the multiplication of FALCON in hardware. However, neither works 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 eliminate the use of floating arithmetic, thus avoiding the challenge of masking the floor function.

1.2 Contributions

In this work, we further expand the countermeasure of 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.

Based on the previous work by Chen and Chen [CC24], we also verify the higher-order security of our method in the probing model. Our formal proofs are based on the Non-Interference (NI) security model first introduced by Barthe et al. [BBD+16]. More specifically, we use the Probe Propagation Framework from Cassiers and Standaert [CS20] to prove the Multiple Inputs Multiple Outputs Strong Non-Interference (MIMO-SNI) security of several of our gadgets.

We provide some performance of our methods and compare them with the unmasked reference 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. However, it could be further optimised.
The source code of this paper is publicly available at the following link:

https://github.com/burak231708/masked_FALCON

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 nonzero 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 notation.

2.2 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. More details on key generation or verification are available in the original FALCON paper[PFH+20].

2.2.1 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 to build an instance of the SIS problem on 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 preimage vector and $\vec{z}$ provided by a Gaussian sampler must be computed. Chen and Chen [CC24] focus on masking the computation of the preimage vector. 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.

2.2.2 Gaussian Sampler

The Gaussian Sampler denoted by SamplerZ can be evaluated from the three following functions, ApproxExp, BerExp, and BaseSampler. In purple are highlighted the operations that are affected by the countermeasures presented in this work.

ApproxExp. This function returns $2^{63} \times ccs \times e^{-x}$ and depends on 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 $2^{63} \cdot ccs \cdot \exp(-x)$

```
1 y ← C[0];          // y and z remain in {0···2^63 − 1} the whole algorithm
2 z ← ⌊2^63 · x⌋;
3 for i from 1 to 12 do
4   y ← C[i] − (z · y) >> 63;
5 z ← ⌊2^63 · ccs⌋;
6 y ← (z · y) >> 63;
7 return y;
```

BerExp. This function returns 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 $\approx ccs \cdot \exp(-x)$

1 $s \leftarrow \lfloor x/\ln(2) \rfloor$; `// Compute the unique decomposition` $x = \ln(2^s) + r$ `with` $(r, s) \in [0, \ln(2)) \times \mathbb{Z}^+$
2 $r \leftarrow x - s \cdot \ln(2)$;
3 $s \leftarrow \min(s, 63)$;
4 $z \leftarrow (2 \cdot \text{ApproxExp}(r, ccs) - 1) >> s$;
5 $i \leftarrow 64$;
6 **do**
7 $\quad i \leftarrow i - 8$;
8 $\quad w \leftarrow \text{UniformBits}(8) - ((z >> i)\ \&\ 0\text{xff})$;
9 **while** $((w = 0)$ *and* $(i > 0))$;
10 **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 $z \sim \chi$

1 $u \leftarrow \text{UniformBits}(72)$;
2 $z_0 \leftarrow 0$;
3 **for** *i from 0 to 17* **do**
4 $\quad z_0 \leftarrow z_0 + [\![u < \text{RCDT}[i]]\!]$;
5 **return** z_0;

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

The Gaussian sampler is constructed as follows:

Algorithm 4: SamplerZ(μ,σ') [PFH+20]

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

1 $r \leftarrow \mu - \lfloor \mu \rfloor$;
2 $ccs \leftarrow \sigma_{min}/\sigma'$;
3 **while** *1* **do**
4 $\quad z_0 \leftarrow \text{BaseSampler}()$;
5 $\quad b \leftarrow \text{UniformBits}(8)\ \&\ 0\text{x}1$;
6 $\quad z \leftarrow b + (2 \cdot b - 1)z_0$;
7 $\quad x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}}$;
8 $\quad$ **if** $\text{BerExp}(x, ccs) = 1$ **then**
9 $\quad\quad$ **return** $z + \lfloor \mu \rfloor$;

2.3 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.

2.3.1 Binary64 Encoding

A floating-point is encoded in the binary64 standard (IEEE 754, [Kah96]) with a sign bit s, a 11-bits long exponent e and a 52-bits long mantissa m such as:

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

2.3.2 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,

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

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

$$\begin{aligned} e > 1074 \implies |x| &= 2^{e-1023} + m \times 2^{e-1023-52} \\ &= (2^{e-1023}) \in \mathbb{N}^* + (m \times 2^{e-1075}) \in \mathbb{N} \implies x \in \mathbb{N}^*. \end{aligned} \tag{3}$$

  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

$$\begin{aligned} 1023 \leq e \leq 1074 \implies x &= 2^{e-1023} + m^{[64:1075-e]} \times 2^{52-(e-1023)+e-1023-52} \\ &= (2^{e-1023}) \in \mathbb{N}^* + (m^{[64:1075-e]}) \in \mathbb{N}. \end{aligned} \tag{4}$$

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

2.4 Masking

Masking is a generic countermeasure against SCA at the algorithmic level. Instead of processing a sensitive datum, it is split into random shares which are processed separately, such as in Boolean and Arithmetic masking [MOP08]. For example, the Boolean masking of a secret bit x in n shares is $Mask(x) = (r_1, \ldots, r_{n-1}, x \bigoplus_{i=1}^{n-1} r_i)$, where the r_i are random bits. Masking security can be evaluated with the *t-probing* model, first introduced in [ISW03]. As a consequence, a gadget is said to be 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.

In this work, we have several gadgets that have multiple inputs and outputs. To formally prove their security, we use the Multiple Inputs Multiple Outputs SNI (t-*MIMO-SNI*) security introduced by Cassiers and Standaert [CS20] to properly compose simple gadgets into more complex but secured gadgets. We have the following definition:

Definition 4. (t-*MIMO-SNI* security [CS20]). Let $\mathcal{O}_i$ be a set of shares indices for $i = 0, \ldots, d-1$. A gadget is t-*MIMO-SNI* if and only if for any set $\mathcal{I}$ of t_1 internal probes and any sets $\mathcal{O}_i$ such that there exists a t_2 that satisfies $t_1 + t_2 \leq t$ and $|\mathcal{O}_i| \leq t_2$ for $i = 0, \ldots, d-1$, the sets of probes $\mathcal{I} \cup y_{\mathcal{O}_0,0} \cup \cdots \cup y_{\mathcal{O}_{d-1},d-1}$ can be simulated with at most t_1 input shares.

Remark 2. In this paper, t-*MIMO-SNI* secure gadgets with a single output are said t-*MI-SNI* secure.

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	t-*SNI*	[SPOG19]
B2A	Boolean to Arithmetic	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-*MIMO-SNI*	Algorithm 5
RemoveDecimal	Truncate the mantissa	t-*MIMO-SNI*	Algorithm 6
SetExponentZero	Set exponent to zero	t-*MIMO-SNI*	Algorithm 7
SecFprBaseInt	Compute the floor	t-*MIMO-SNI*	Algorithm 9
SecFprComp	Compares two values	t-*MI-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

2.4.1 Strategy to mask FALCON

We follow a similar approach to Chen and Chen [CC24]. They take the reference implementation for the addition and multiplication of floating point value in FALCON and

"translated" it in a masked form. We do the same for the floor function (Section 3) and the inverse (Section 4) by taking a naive implementation of each of them in the Binary64 encoding and translating it in a masked form. We then integrate [CC24] and our work in a *proof-of-concept* implementation in C for computer (Section 6) following the reference specification given by the FALCON team to have the performances on the Gaussian Sampler and then on the entire FALCON.

3 Masking the Floor Function

In Section 2.3.2 we have described how to calculate the floor using floating-point arithmetic. We now present the corresponding masking gadgets.

Remark 3. With small modifications, our design can also be used to calculate the functions *truncate* and *rounding*.

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

3.1 SecFprUrsh$_\mathsf{f}$

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

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

1. $(m_i)_{1 \le i \le n} \leftarrow ((1 << 63), 0, \cdots, 0)$;
2. Refresh$((cx_i))$;
3. **for** *j from 1 to n* **do**
4.   Right-Rotate (my_i) by cx_j;
5. $(my_i) \leftarrow$ RefreshMasks$((my_i))$;
6.   Right-Rotate (m_i) by cx_j;
7. $(m_i) \leftarrow$ RefreshMasks$((m_i))$;
8. $len \leftarrow 1$;
9. **while** $len \le 32$ **do**
10. $(m_i) \leftarrow (m_i \oplus (m_i >> len))$;
11. $len \leftarrow len << 1$;
12. $(my'_i) \leftarrow$ SecAnd$((my_i), (m_i))$;
13. $(rot_i) \leftarrow$ SecAnd$((my_i), (\neg(m_i)))$;
14. **return** $((my'_i), (rot_i))$;

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

3.2 RemoveDecimal

The 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 SecFprUrsh$_{\text{floor}}$. Once the mantissa is shifted, we have performed the function $truncate(x)$. As described in Section 2.3.2, for the floor we also have to check

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

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 $ey + (52 - cx)$

1. $cx_1 \leftarrow cx_1 - 52$;
2. $(c_i) \leftarrow \text{A2B}((cx_i))$;
3. $(cp_i) \leftarrow ((c_i^{(16)}))$;
4. $(c_i) \leftarrow \text{SecAnd}(\text{Refresh}((c_i)), (-cp_i))$;
5. $(cx_i) \leftarrow \text{B2A}((c_i))$;
6. $(my'_i), (rot_i) \leftarrow \text{SecFprUrsh}_f((my_i), (-cx_i))$;
7. $(b_i) \leftarrow \text{SecNonZero}((rot_i))$;
8. $(cp_i) \leftarrow \text{SecAnd}((cp_i), (sy_i))$;
9. $(cp_i) \leftarrow \text{SecAnd}((cp_i), (b_i))$;
10. $(my'_i) \leftarrow \text{SecAdd}((my'_i), (cp_i))$;
11. $(ey'_i) \leftarrow (\text{Refresh}(ey_i) - cx_i)$;
12. **return** $((my'_i), (ey'_i), (b_i))$;

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 from the final result in accordance with Section 2.3.2 and proceed by securely adding $cp = s \wedge b$ to the shifted my, as summarised 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 a negative integer
1	1	1	x is a non-integer negative real

3.3 SetExponentZero

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

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.

1. $(ey_i) \leftarrow \text{A2B}((ey_i))$;
2. $(b'_i) \leftarrow (-sy_i)$;
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** $((ey_i), (sy_i))$;

Finally, we have to address the exponent computation. This is done with the SetExponentZero gadget (Algorithm 7). This function handles specific binary64 encoding cases, specifically the encoding of 0 and the one of -1. In fact, if $| x |< 1$ and $sy = 0$, then the expected result is 0 in binary64 form. Otherwise, if $sy = 1$ and $| x |< 1$, then the expected result is -1 in binary64 form. Table 3 highlights the relationship between s_y, b and the expected result.

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 $sy = sy$

3.4 SecFprBaseInt$_\mathrm{f}$:

The gadget SecFprBaseInt$_\mathrm{f}$ (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 focusses 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, 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.

1 $(mx_i) \leftarrow (x_i^{[52:1]})$;
2 $(mx_i) \leftarrow \mathrm{SecAdd}((mx_i), (2^{52}, 0, \cdots, 0))$; `// add implicit bit in the mantissa`
3 $(ex_i) \leftarrow (x_i^{[63:53]})$;
4 $(ex_i) \leftarrow \mathrm{B2A}((ex_i))$;
5 $(sx_i) \leftarrow (x_i^{(64)})$;
6 **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, $| x |< 1$ 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 RemoveDecimal$_{floor}$ (Algorithm 6), as described in Section 2.3.2. The cases are as follows: If $cx \geq 52$, then x is an integer as shown in Equation 3 and no modification of the mantissa is required. Otherwise, if $0 \leq c_x \leq 51$, we truncate the mantissa.

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

Algorithm 9: SecFprBaseInt$_\mathrm{f}$(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).
1 $((my_i),(ey_i),(sy_i)) \leftarrow$ SecFprExtract$((x_i))$;
2 Refresh$((sy_i))$;
3 $(cx_i) \leftarrow (ey_i)$, $cx_1 \leftarrow ey_1 - \mathrm{Zero}_f$;
4 $(c_i) \leftarrow$ A2B$((cx_i^{(16)}))$;
5 $(my_i) \leftarrow$ SecAnd$((my_i),(\neg(-c_i)))$;
6 $(my_i),(ey_i),(Rnd_i) \leftarrow$ RemoveDecimal$_f((my_i),(ey_i),(sy_i)$,Refresh$((cx_i)))$;
7 $(my_i),(ey_i) \leftarrow$ SecFprNorm64$((my_i),(ey_i))$;
8 $(my_i) \leftarrow (my_i^{[63:11]})$;
9 $ey_1 \leftarrow ey_1 + 11$;
10 $(ey_i'),(sy_i') \leftarrow$ SetExponentZero$_f((ey_i),(\neg(-c_i)),(sy_i),(Rnd_i))$;
11 $(y_i^{(64)}) \leftarrow (sy_i)$, $(y_i^{[63:53]}) \leftarrow (ey_i)$, $(y_i^{[52:1]}) \leftarrow (my_i)$;
12 **return** (y_i);

As the algorithm RemoveDecimal does not normalise the mantissa, 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 of computing the specific encoding of "0" if it is the expected result, by applying the 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 precalculate the inverse and perform a multiplication. However, the first division in SamplerZ (line 2) involves a division with secret information. Hence, we must perform a division securely 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 5:

$$e_y = 1023 - (e_x - 1023 + b) = 2046 - e_x - b \tag{5}$$

Computing the mantissa corresponds to the Euclidean division: first, the dividend $d = (1 << c_x)$ is compared to x by computing $comp =$ 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.

If $x < d$, then the comparison algorithm outputs 1. This result is carried over to the new mantissa, and we add $-x$ to d. Otherwise, if $comp = 0$, no addition is performed on d. To continue the Euclidean division, d is shifted one time to the left. Performing this

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 $[\![x < y]\!]$

1 Refresh((x_i));
2 $(mx_i) \leftarrow (x_i^{[63:1]})$, $\quad (my_i) \leftarrow (y_i^{[63:1]})$;
3 $(d_i) \leftarrow$ SecAdd($(mx_i), (\neg my_1, my_2, \cdots, my_n)$);
4 Refresh((d_i));
5 $(b_i) \leftarrow$ SecNonZero($(\neg d_1, d_2, \cdots, d_n)$);
6 $(b'_i) \leftarrow$ SecNonZero($(\neg(d_1 \oplus 2^{63}), d_2, \cdots, d_n)$);
7 $(comp_i) \leftarrow (d_i^{(63)} \oplus b_i \oplus b'_i)$;
8 **return** $(comp_i)$;

shift is done by calling the SecFprScalePow2 (Algorithm 11) function. This function either multiplies by 2 or divides by 2 its input, and truncates the result if necessary.

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 $x \times 2^p$

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

After we compute these 53 bits (52 plus the implicit bit) of the mantissa m_y, two additional bits are calculated to preserve the sticky bit. Consequently, we have the 55 bits of the mantissa m_y.

4.2 Masking BerExp

BerExp (Algorithm 2) requires a secure computation of a minimum and a right-shift by a sensitive value. For the minimum, the comparison is made between a constant equal to 63 and the sensitive value 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 is given by the mask of $((-s\epsilon) \wedge X) \vee ((-(\neg s\epsilon)) \wedge 63)$. The calculations of the minimum are performed in Algorithm 13.

To right-shift a binary64 masked Y to another binary64 masked $X \in [\![0, 63]\!]$, 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 must take into account the possibility that $X = 0$. Thus, when injecting the implicit bit into 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

Algorithm 12: SecFprInv((x_i))

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

1 $(sx_i), (ex_i), (mx_i) \leftarrow$ SecFprExtract((x_i));
2 $(b_i) \leftarrow$ SecNonZero((mx_i));
3 $(ba_i) \leftarrow$ B2A(b_i);
4 $(ed_i) \leftarrow (ex_i + ba_i)$;
5 $(ey_i) \leftarrow (-ed_i)$;
6 $(ey_i) \leftarrow$ A2B((ey_i)), $(ed_i) \leftarrow$ A2B((ed_i));
7 $(d_i) \leftarrow (ed_i << 52)$;
8 $(minusX_i) \leftarrow$ Or($(2^{63}, 0, \cdots, 0), (x_i)$);
9 **for** *j from 1 to 55* **do**
10 $(comp_i) \leftarrow$ SecFprComp($(x_i), (d_i)$);
11 $(my_i) \leftarrow (my_i \oplus (comp_i << (63 - j)))$;
12 $(xcpy_i) \leftarrow$ SecAnd($(minusX_i), -(comp_i)$);
13 $(d_i) \leftarrow$ SecFprAdd($(xcpy_i), (d_i)$);
14 $(d_i) \leftarrow$ SecFprScalePow2($(d_i), 1$);
15 $(my_i) \leftarrow$ SecAnd($(my_i), -(b_i)$);
16 $(y_i^{(64)}) \leftarrow$Refresh((sx_i)), $(y_i^{[63:53]}) \leftarrow (ey_i), (y_i^{[52:1]}) \leftarrow (my_i^{[54:3]})$;
17 $(f_i) \leftarrow$ SecOr(Refresh($my_i^{(1)}$), $(my_i^{(3)})$);
18 $(f_i) \leftarrow$ SecAnd($(f_i), (my_i^{(2)})$);
19 $(y_i) \leftarrow$ SecAdd($(y_i), (f_i)$);
20 **return** (y_i);

with the SecFprUrsh function:

$$m = \text{SecFprUrsh}(mX', 52 - eX + 1023) \tag{6}$$

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 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 shift 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 use the Probe Propagation Framework introduced in [CS20] to both prove the *t-SNI* and *t-MIMO-SNI* security of our gadgets. This framework highlights where the information a probe gathers comes from in a circuit. We first recall their methodology, which relies on graphs.

Definition 5. (*Gadget Composition*, [CS20]) A gadget composition G over n shares is a directed acyclic graph (DAG) whose vertices are composing gadgets (which are gadgets over n shares) or inputs/outputs, and edges are connections between those gadgets. For each composing gadget, there is a one-to-one mapping between its m inputs and the incoming edges of the associated vertex. Furthermore, each outgoing edge is associated to an output of the gadget (there can be multiple edges associated to the same output). Output vertices (resp., input vertices) have one (resp., zero) incoming edge and zero (resp., any number of) outgoing edge(s). A gadget composition can be instantiated by mapping each vertex to the corresponding gadget or n inputs/outputs,and each edge to n wires

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

1. $(sx_i), (ex_i), (mx_i) \leftarrow$ SecFprExtract((x_i));
2. (st_i) is a masking of the value 63;
3. $ex_1 \leftarrow ex_1 - 1029$;
4. $(ex_i) \leftarrow$ A2B((ex_i));
5. $(rA_i) \leftarrow$ SecAnd($(-(ex_i)^{(16)}), (x_i)$);
6. $(rB_i) \leftarrow$ SecAnd($(-(\neg(ex_i)^{(16)})), (st_i)$);
7. $(y_i) \leftarrow$ SecOr($(rA_i), (rB_i)$);
8. **return** (y_i);

(which connect the composing gadgets). The inputs and outputs of the composing gadgets are erased in the instantiation process. We use the term *composite gadget* to refer to the instantiation of a gadget composition.

In their work, Cassiers and Standaert [CS20] introduced a new computation graph model based on the gadget composition DAG. They forbid the connection of more than one edge to an output of a gadget or an input gate. Forbidden cases are addressed with the $Split_j$ gadget, which takes one input and returns j duplicates of it. For simplicity, they assume that all composing gadgets are *t-NI* operation gadgets, *t-SNI* Refresh gadgets or $Split_j$ gadgets. They model *t-SNI* gadgets as *t-NI* ones followed by a *t-SNI* Refresh. We do the same for *t-MIMO-SNI* gadgets. To graphically prove the *t-MIMO-SNI* security of a gadget composition, we use the Simplified Computation Graph (SCG) model:

Definition 6. (*Simplified Computation Graph* (SCG), [CS20]) The simplification of the computation graph G is the graph that is obtained from G by removing all *t-SNI* Refresh vertices and their incident edges.

According to [CS20] (Proposition 7, page 2549), to prove the *t-NI* security of a gadget composition, it is sufficient to verify that its SCG is a Single Path - NI Built gadget (SP-NIB):

Definition 7. (*Single Path - NI Built gadget* (SP-NIB), [CS20]) A composite gadget G is SP-NIB if it is implemented with only NI gadgets and SNI refreshes, and if for any pair of vertices u, v in the corresponding simplified computation graph there exists at most one path from u to v.

The conditions required for a gadget composition to verify *t-MIMO-SNI* security are as follows:

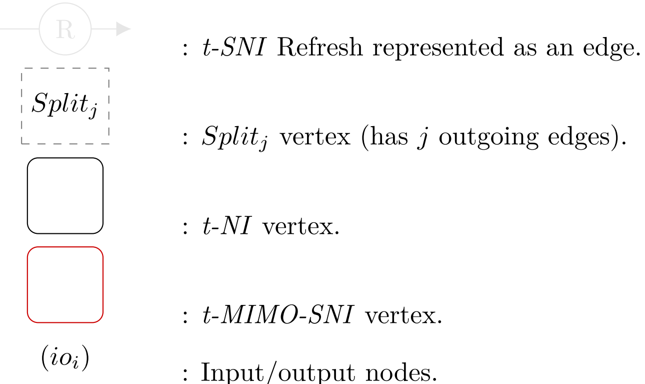
Proposition 1. *A composite gadget G is* t-MIMO-SNI *if it satisfies the three following conditions. (i) G is SP-NIB. (ii) For any pair of output nodes u_1, u_2 there is no node v such that there is a path from v to u_1 and a path from v to u_2. (iii) For any pair of input nodes u_1, u_2 there is no node v such that there is a path from u_1 to v and a path from u_2 to v.*

In this section, we present the SCGs for each one of our gadgets and ensure that they either verify the Proposition 1 (*t-MIMO-SNI*) or are SP-NIB (*t-NI*). For *t-SNI* gadgets, we also use SCGs to first prove that they are SP-NIB and thus *t-NI.* Then we highlight that there is no path from any input node to the output node, thus proving that a probe on the output cannot propagate to the inputs, which is a sufficient condition to prove that a *t-NI* gadget composition is *t-SNI* secure.

5.1 SCG Legend

For readability, we keep *t-SNI* Refreshes on the SCG. They are, however, converted to an edge and grayed out. They are not taken into account when looking at the SCG properties. We also coloured the *t-MIMO-SNI* gadgets differently to highlight them. They will be considered at *t-NI* nodes in the SCG proofs. Finally, the inputs will always be aligned on the left and the outputs on the right of the graphs.

Table 4: SCG legend



5.2 Floor Function

5.2.1 SecFprUrsh

Lemma 1. *The gadget* ***SecFprUrsh***$_{\textbf{floor}}$ *(Algorithm 5) is* t-MIMO-SNI *secure.*

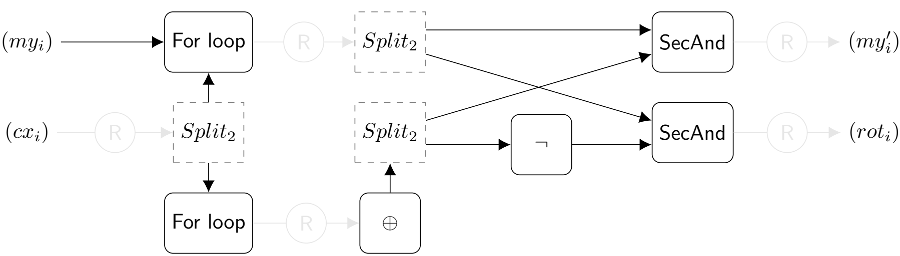


Figure 1: SCG of SecFprUrsh$_{floor}$

Proof. The gadget **SecFprUrsh**$_{\textbf{floor}}$ 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 For loop (see [CC24], Lemma 3 and Figure 2) to justify our representation of the two For loop nodes as *t-NI* ones followed each by a *t-SNI* Refresh in the SCG. The graph in Figure 1 verifies Proposition 1. To ensure that condition (iii) is verified, a *t-SNI* Refresh is performed on the input (cx_i). □

5.2.2 RemoveDecimal

Lemma 2. *The gadget* ***RemoveDecimal***$_{\textbf{floor}}$ *(Algorithm 6) is* t-MIMO-SNI *secure.*

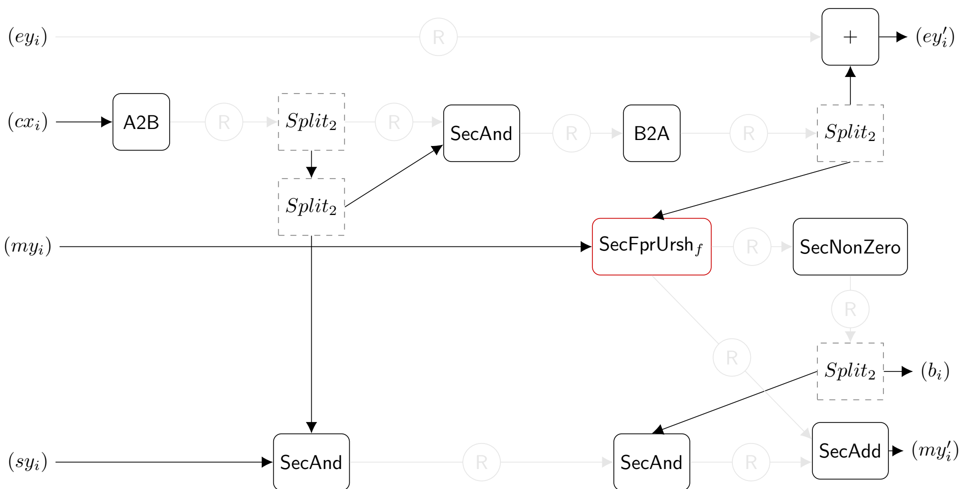


Figure 2: SCG of $\text{RemoveDecimal}_{\text{floor}}$

Proof. The SCG in Figure 2 verifies the Proposition 1. The *t-MIMO-SNI* security of the SecFprUrsh_f node is proven in Lemma 1. We perform a *t-SNI* Refresh on the input (ey_i) to cut it from the output (ey'_i), otherwise the gadget would be *t-NI* secure only. □

5.2.3 SetExponentZero

Lemma 3. *The gadget* $\textbf{\textit{SetExponentZero}}_{\textbf{\textit{floor}}}$ *(Algorithm 7) is* t-MIMO-SNI *secure.*

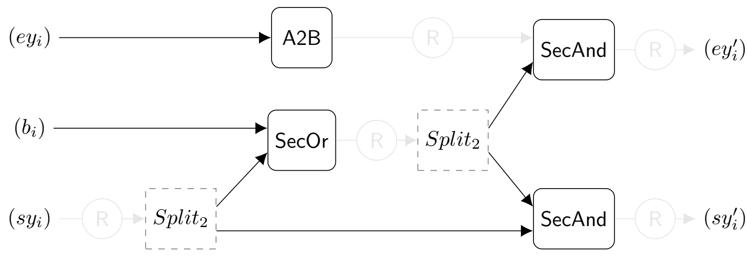


Figure 3: SCG of $\text{SetExponentZero}_{floor}$

Proof. The Proposition 1 is verified in Figure 3. To ensure condition (iii), we perform a *t-SNI* Refresh on the (sy_i) input. □

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

Proof. The Proposition 1 is verified in Figure 4. To ensure condition (iii), we perform a *t-SNI* Refresh on the (sy_i) input. The SecFprNorm64 gadget has multiple outputs, but its *t-NI* security has been proved in [CC24] (Lemma 4, page 291). □

5.3 Inverse

5.3.1 SecFprComp

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

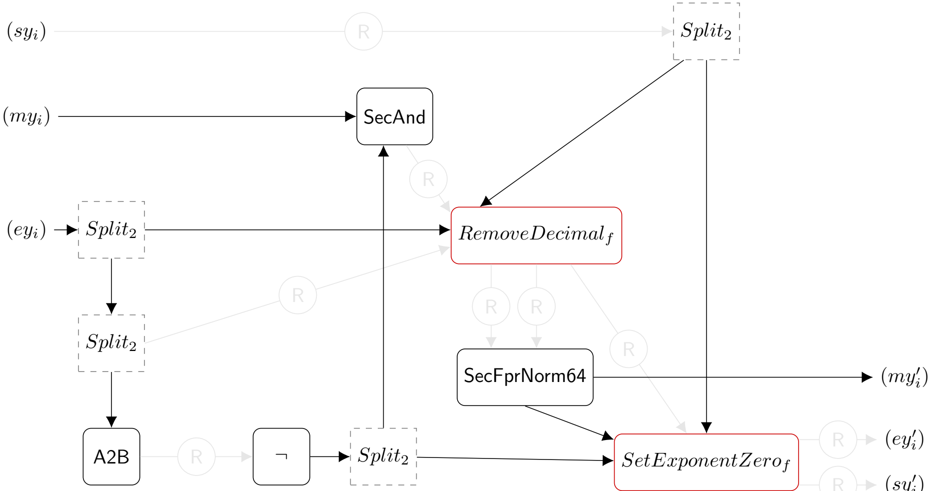


Figure 4: SCG of $\mathrm{SecFprBaseInt_{floor}}$

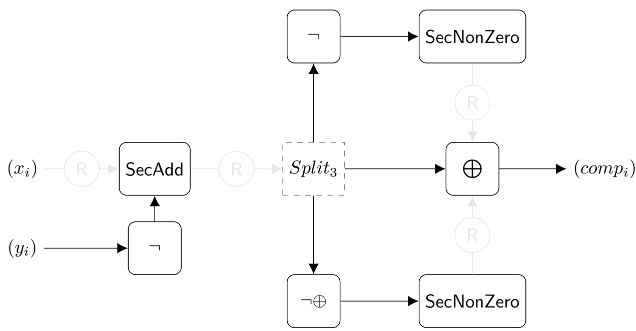


Figure 5: SCG of SecFprComp

Proof. The SCG in Figure 5 verifies Proposition 1. To ensure the third condition of the proposition, a *t-SNI* Refresh is performed on the input (x_i). The gadget has only one output node. Thus, it is *t-MI-SNI* secure (the Multiple Output or MO condition is not required). □

5.3.2 SecFprScalePow2

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

Proof. The SCG in Figure 6 is SP-NIB. The gadget is thus at least *t-NI* secure. As there is no path from any input to the output node, a probe placed on the output cannot propagate to the input gates, and we have a *t-SNI* secure gadget. □

5.3.3 SecFprInv

We first start by proving the following lemma.

Lemma 6. *The **For loop** in gadget **SecFprInv** (Algorithm 12, Line 9 to 14) is* t-NI *secure.*

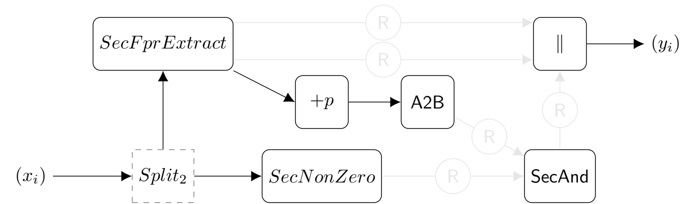


Figure 6: SCG of SecFprScalePow2

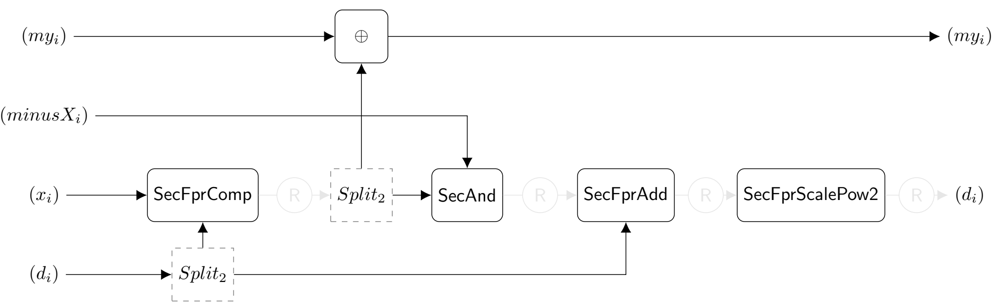


Figure 7: SCG of the For loop in the SecFprInv gadget

Proof. The SCG in Figure 7 is SP-NIB. Thus, an isolated instance of the loop is *t-NI* secure. To ensure that the composition of instances remains *t-NI* secure, we proceed recursively. The outputs (d_i) and (my_i) are used as inputs for the next instance. The propagation of a probe from the output (d_i) is immediately blocked by a *t-SNI* Refresh in the SCG. This is sufficient to ensure the composition of two instances of the loop. Recursively, the composition of all instances of the For loop in the SecFprInv can be modelled as a SCG which verifies SP-NIB. □

Remark 4. As SecFprComp is *t-MI-SNI* secure, all the inputs of the For loop verify the third condition of the Proposition 1. This fact will be used in the proof of the following theorem:

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

Proof. At first glance, the SCG in Figure 8 is not SP-NIB as it seems that two paths are possible from the input (x_i) to the node For loop. However, according to Remark 4, there is no node v within the SCG of the For loop (Figure 7) such that there is a path from an input node to v and also from a different input node to v. Thus, by replacing the For loop with its SCG, the SCG in Figure 8 is SP-NIB. As there is no path from (x_i) to (y_i), the gadget is *t-SNI* secure. □

5.4 Minimum63

Lemma 7. *The gadget* ***Minimum63*** *(Algorithm 13) is* t-SNI *secure.*

Proof. The SCG in Figure 9 is SP-NIB. There is no path from the input node to the output. □

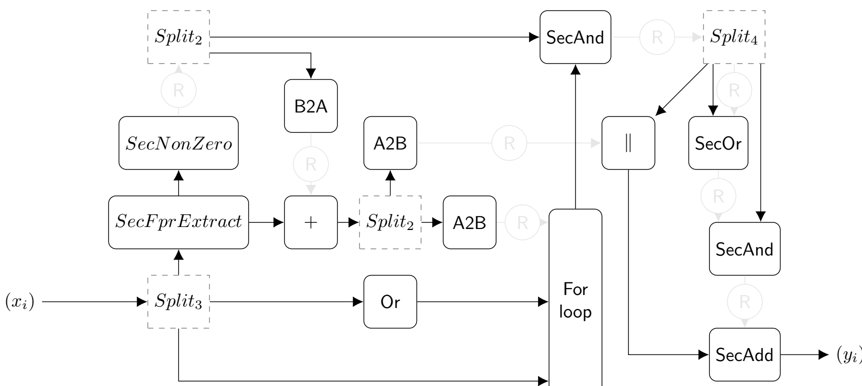


Figure 8: SCG of SecFprInv

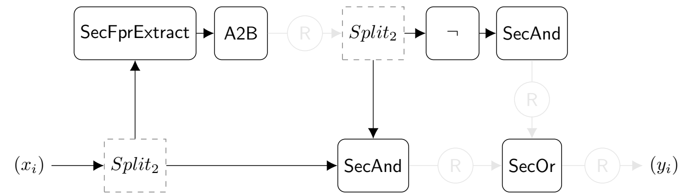


Figure 9: SCG of Minimum63

6 Performances

Some results are shown in Table 5. This implementation is not optimised and is performed on a laptop computer equipped with an Intel Core i7-11800H CPU. The compiler used is *gcc version 9.4.0*. We have considered our performance of SecFprAdd and SecFprMul as reference and compared our work with that 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 optimisations for 2 shares masking have not been implemented.

Table 5: Time in microseconds

Algorithm	[PFH+20]	**2 Shares**	**3 Shares**	**4 Shares**
SecFprAdd [CC24]	0.000 11	3.949	9.469	17.143
SecFprMul [CC24]	0.000 14	2.641	7.096	12.686
$\text{SecFprBaseInt}_{\text{floor}}$	0.000 136	2.976	7.205	13.125
$\text{SecFprUrsh}_{\text{floor}}$	-	0.228	0.330	0.493
SecFprInv	0.000 138	268.209	740.935	1283.108
SecFprComp	-	0.732	1.451	2.519
SecFprScalPwo2	-	0.649	1.673	2.762
ApproxExp	0.000 126	94.236	239.578	441.347
BerExp	0.005 446	112.061	288.597	522.388
SamplerZ	0.114	417.141	1071.943	1974.423

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 5 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. The methodology used to fully mask FALCON is to rely on the high-level view of the signature and use Chen and Chen [CC24] gadgets or the ones presented in this paper when required. Its performances are summarised in Table 6. We do not perform the signature rejection. Thus, in a real-world use case, the performances might be doubled on average. Our results clearly highlight that this masking methodology for FALCON is not ready for deployment.

Table 6: 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 new gadgets to mask the floor function (Section 3). The Gaussian sampler of FALCON (Section 4) has been protected with this floor gadget. Furthermore, to complete this task, we provided a masked implementation of the division (Section 4). We discussed the *t-SNI* properties of our gadgets (Section 5). Finally, we provided some performances 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.

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.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020.

[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.

[KA24] Emre Karabulut and Aydin Aysu. Masking falcon's floating-point multiplication in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):483–508, 2024.

[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.