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 condidate 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). Falcon is a lattice-based signature which relies of rational numbers which is unusual in the cryptography field. While recent work proposed a solution to mask the addition and the multiplication, some roadblocks remains, most noticeably how to protect the floor function. We propose in this work to complete the existing first trials of hardening Falcon againt 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 and 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 breach, new cryptographic primitves 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 of accessing 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 implementations of new primitives and the ways to protect them is an active field of research.

While they have been many works focusing on CRYSTALS Dilithium and CRYSTALS Kyber, summed up by Ravi et al. [26], FALCON is noticeably harder to protect. Indeed, the algorithm relies on floating-point arithmetic, for which there is little litterature 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, it was proved vulnerable by Karabulut and Aysu [17] using an ElectroMagnetic (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 delved 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. However, 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 re-write 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. We discuss the application of this method 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].

Finally, 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 i7-11850H CPU.

2 Notation and Background

2.1 Notation

- 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.

2.2 Diagram Legend

The following diagrams (Figures 4.1, 4,4) 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, please 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] focuses on masking the pre-image vector computation. In this work we intend to mask the Gaussian Sampler. The signature algorithm is detailled in Algorithm **??**.

Gaussian Sampler The Gaussian Sampler is the composition of 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 ? 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 proba $ccs \times e^{-x}$:

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

SamplerZ. The gaussian Sampler:

Algorithm 3: SamplerZ(μ,σ') [25]

```
Data: Foating-point values $\mu,\sigma' \in \mathcal{R}$ such that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
Result: An integer $z \in \mathbb{Z}$ 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
    $z_0 \leftarrow \textsc{BaseSampler}()$;
    $b \leftarrow \textsc{UniformBits}(8)\ \&\ 0\textsc{x}1$;
    $z \leftarrow b + (2 \cdot b - 1)z_0$;
    $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}}$;
    if $BerExp(x, ccs) = 1$ then
        return $z + \lfloor \mu \rfloor$;
```

Algorithm 4: BaseSampler() [25]

```
Data: –
Result: An integer $z_0 \in \{0, \cdots, 18\}$ such that $z \sim \chi$
$u \leftarrow \textsc{UniformBits}(72)$;
$z_0 \leftarrow 0$;
for i from 0 to 17 do
    $z_0 \leftarrow z_0 + [\![u < \mathrm{RCDT}[i]]\!]$;
return $z_0$;
```

where RCDT is define 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 $z \leq x$.*

There are several ways of computing the floor function. One relies on floating-point arithmetic [16].

Binary64 Encoding A floating-point is encoded[4] with a sign bit s, a 11-bits long exponent e and a 52-bits long mantissa m such as:

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

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 of 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

$$\begin{aligned} e > 1074 \implies x &= 2^{e-1023} + m \times 2^{e-1023-52} && (4)\\ &= (2^{e-1023}) \in \mathbb{N}^* + (m \times 2^{e-1075}) \in \mathbb{N} \implies x \in \mathbb{N}^*. && (5) \end{aligned}$$

- 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

$$\begin{aligned} 1023 \leq e \leq 1074 \implies x &= 2^{e-1023} + m^{[64:1075-e]} \times 2^{52-(e-1023)+e-1023-52} && (6)\\ &= (2^{e-1023}) \in \mathbb{N}^* + (m^{[64:1075-e]}) \in \mathbb{N}. && (7) \end{aligned}$$

However, while this method is fast and efficient, it requires the use of the exponent and the sign. We propose here a method to perform this truncation in a secure manner.

2.5 Masking

Masking is a generic countermeasure to SCA at the software level. Instead of processing a sensitive data, it is split in 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 introduce 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.

[4] Binary64 Standard – IEEE 754 [16]

Definition 2. *(*t-NI*) security). A gadget is said* t-*Non Interference (*t-NI*) secure if every set of* t *interemediate 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). A gadget is* 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 will use those models in Section 4 to demonstrate the security of our design.

3 Masking The Floor Function

In this part we denote by f as one of these three funtions : floor, truncature and round. Differences will be discussed when necessary.

3.1 Overview

Recall that we're working with floating-point using the Binary64 – IEEE 754 [16] standard representation. Its encoding is defined as follows: Let $x \in \mathbb{R}$. x 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 such as:

Lemma 1. *Let f be floor or truncate or rounding.Computing f mainly depends on the exponent e, with three different cases:*

1. *If $e - Zero_f < 0$, then f(x) = 0. $Zero_f$ depends on f. For f= floor or truncate, we take $Zero_f = 1023$. For f=rounding we take $Zero_f = 1022$.*
2. *If $e - 1075 \geq 0$, then f(x) = x;*
3. *If $0 \leq e - Zero_f \leq 51$,the result must be computed by removing the $52-(e-Zero_f)$ decimals from x depending on f.*

Proof. 1. $e - Zero_\mathrm{f} < 0$. This case covers $x \in (-1; 1)$ According to Equation 1 we have for $Zero_\mathrm{f} = 1023$:

$$|x| = 2^{e-1023} \times (1 + m \times 2^{-52}) \leq 2^{-1} \times (1 + m \times 2^{-52}) < 1. \tag{8}$$

Hence floor(x) = truncate(x) = 0. For $Zero_\mathrm{f} = 1022$, we have:

$$|x| = 2^{e-1023} \times (1 + m \times 2^{-52}) \leq 2^{-2} \times (1 + m \times 2^{-52}) < 0.5. \tag{9}$$

Hence rounding(x) = 0.

2. $e - 1075 \geq 0$. This case covers $x \in \mathbb{Z}$According to Equation 1 we have:

$$|x| = 2^{e-1023} \times (1 + m \times 2^{-52}) \geq 2^{52} \times (1 + m \times 2^{-52}) \geq 2^{52} + m \in \mathbb{N}. \tag{10}$$

Hence f(x) = $x \in \mathbb{Z}$.

3. The last case is function specific but covers $x \in \mathbb{R} \backsim (-1;1) \backsim \mathbb{Z}$.

□

These three distinguished cases provide a fixed structure shown in algorithm 5 for f. We must compute securely all the checks on masked values, and to change in consequence our mantissa, exponent and eventually our sign bit.

3.2 Masked Floor Function

The function SecFprBaseInt is the main function to compute masked floor, masked truncature or masked round. Its result depends on the inputed function: gadgets and Zero_f are different.

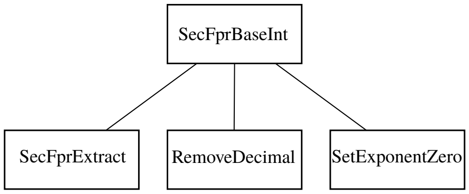

Fig. 1. SecFprBaseInt and its gadgets

In this subsection, suppose f = floor. We will first present the main structure and then present the gadgets. Gadgets from trunc and round function can be found in annexe (the little differences between gadgets will be explain too):

- SetFprExtract;
- $\text{RemoveDecimal}_{\text{trunc}}$ and $\text{SetExponentZero}_{\text{trunc}}$;
- $\text{RemoveDecimal}_{\text{round}}$ and $\text{SetExponentZero}_{\text{round}}$.

The particularity of our shift method for calculating floor function is that it requires the gadgets proposed by Chen and Chen [6]. So by proposing this method, we are extending their work to mask the Gaussian sampler by using only the gadgets of their creation. By way of summary, we propose a table showing the functions used to create our gadgets and the sources from which they come.

$\text{SecFprBaseInt}_{\mathbf{f}}$ – algorithm 5 We present a global structure for all three functions f = floor/round/truncate.

Algorithm	**Description**	**Security**	**Reference**
SecAnd	AND of Boolean shares	t-SNI	[15], [1]
SecAdd	Addition of Boolean shares	t-SNI	[8], [2]
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 keeping sticky bit updated	t-SNI	[6]
SecFprNorm64	Normalization to $[2^{63}, 2^{64})$	t-NI	[6]

Algorithm 5: $\mathrm{SecFprBaseInt_f(x)}$

```
Data: 64-bit boolean shares $(x_i)_{1 \leq i \leq n}$ for value x
Result: 64-bit boolean shares $(y_i)_{1 \leq i \leq n}$ for mantissa value y = f(x).
$((my_i), (ey_i), (sy_i)) \leftarrow \mathrm{SecFprExtract}((x_i))$;
$(cx_i) \leftarrow (ey_i)$;
$cx_1 \leftarrow ey_1 - \mathrm{Zero}_f$;
$(c_i) \leftarrow \mathrm{A2B}((cx_i^{(16)}))$;
$\mathrm{Refresh}((cx_i))$;
$(my_i) \leftarrow \mathrm{SecAnd}((my_i), (\neg(-c_i)))$;
$(my_i), (ey_i), (Rnd_i) \leftarrow \mathrm{RemoveDecimal}_f((my_i), (ey_i), (sy_i), (cx_i))$;
$(my_i), (ey_i) \leftarrow \mathrm{SecFprNorm64}((my_i), (ey_i))$;
$(my_i) \leftarrow (my_i >> 11)$;
$(my_i) \leftarrow (my_i^{[52:1]})$;
$ey_1 \leftarrow ey_1 + 11$;
$(ey_i), (sy_i) \leftarrow \mathrm{SetExponentZero}_f((ey_i), (\neg(-c_i)), (s_i), (Rnd_i))$;
$(y_i^{(64)}) \leftarrow (sy_i)$;
$(y_i^{[63:53]}) \leftarrow (ey_i)$;
$(y_i^{[53:1]}) \leftarrow (my_i)$;
return $(y_i)$;
```

As explained in overview, these three functions can be summarised in a global structure. What will differentiate the three functions will in fact be the gadgets and the Zero_f parameter. To perform the floor function, we first extract the data from the encoding and place it in three variables s_y, e_y and m_y, which will be linked directly to the output of the algorithm. By extracting this data[5], we carry out a few operations on the mantissa and the exponent to match the conventions of Chen's implementation. Indeed, the input x is a 64-bit boolean

[5] Pseudo-code in appendix: SecFprExtract – algorithm 9

share, consequently we'll have to transform the exponent into a 16-bit arithmetic share by performing a boolean to arithmetic conversion. Another important thing to add is the bit implicit in the mantissa. This bit which is not present in the x information, in order to save one bit and gain in precision, is nevertheless very important. In particular, it enables us to normalise our shares correctly. Once these changes have been made, we can start working on the exponent.

The first step consists of checking whether the result is an obvious zero or not. To do so, we compare e_y and Zero_f – which is the correponding exponent to $1 = 2^0$ – by calcultating $c_x = e_y - \text{Zero}_f$. If cx is negative, $\mid x \mid< 1$ and all decimals can be removed by putting $my = 0$. We just need to be carefull at the end of the algorithm because of some particular cases[6]. This step without taking care of particular case corresponds in reality to the truncature. Otherwise, the m_y mantissa will remain unchanged to be adjusted during the next steps.

A mathematical explanation of Zero_f's choice can be summurize in a few simple calculus. In case $f =$ floor or trunc, $\text{Zero}_f = 1023$. Indeed if $e_y - 1023 < 0$,

$$2^{e_x-1023} \times (1 + m_x \times 2^{-52}) \quad \leq \quad 2^{-1} \times (1 + m_x \times 2^{-52}) \quad < \quad 1$$

We remark a difference for $f =$ round. The main reason of this difference is that we want to avoid to reject the case where the binary decomposition of x contains a 1 at indices $2^{-1} = 0.5$, since we will no longer be able to round off the final result. This justify the choice of $\text{Zero}_f = 1022$ for rounding function. This time, if $e_y - 1022 < 0$:

$$2^{e_x-1023} \times (1 + m_x \times 2^{-52}) \quad \leq \quad 2^{-2} \times (1 + m_x \times 2^{-52}) \quad < \quad 0.5$$

To sum up, after the first step, only my can change. The two values it can take are my it-self (no changes) or 0. The second step removes the decimals by shifting them to the right, using RemoveDecimal – algorithm 6. As this algorithm does not normalise the mantissa, we then apply the SecFprNorm64 function, !!!!!!!! followed by a refresh (SecFprNorm64 is a t-NI gadgets) and !!!!!!!!!!! a computation of 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, is to apply the specific encoding of "0" if this is the result to be returned. To do this, we apply the SetExponentZero_f function – algorithm 8.

RemoveDecimal$_{\textbf{floor}}$ – algorithm 6 As we explained at the beginning of this section, we need to remove the decimals from the input number. After checking a first result, the mantissa is equal to 0, if $e - \text{Zero}_f < 0$, or has remained unchanged. This leaves us with the case where a shift is needed. To do so, we write $cx = e - 1023$ as the shift to be performed. The case where $cx < 0$ has already been treaten by modifying (or not) the mantissa, whatever shift is made, we would have $0 >> cx = 0$. If $cx =\geq 52$, x is already in the correct form. In fact, we have :

$$\mid (-1)^s \times 2^{cx} \times (1 + m \times 2^{-52}) \mid \quad \geq \quad 2^{52} \times (1 + m \times 2^{-52}) \quad = \quad 2^{52} + m \in \mathbb{N}$$

[6] For example, if $-1 < x < 0$, we need to return encoding -1, not zero.

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

To avoid removing information if it isn't needed, we replace cx by 0. Be careful if we want to round a number : 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 is the result was 0 or not. So here we need to add 1. If $0 \leq e_y - 1023 \leq 51$, we need to modify the mantissa to remove – if necessary– the decimals. Remarks that even in this case, the mantissa can be in the good format. For example if $x = 5$, we have $0 \leq e_y - 1023 = 1025 - 1023 = 2 \leq 51$ and x is an integer.

After verifying is x is or isn't a "big" number, i.e. without decimal in its representation, we can shift[7] my by $cd = 52 - cx$ by applying a modified [8] SecFprUrsh$_f$ – algorithm 7: we don't want to keep the sticky bit, so we just removed this part. We had an extra output, returning the part we removed. Shift a value is a good thing, but that's not exactly the floor function. After this step we just have the truncature. When x positive there's no problem at all, floor function is equal to truncature. But when x is negative there is one. It

[7] In pseudo-code the subtraction is implicite due to the format of a masked value.

[8] If we want to use Chen's algorithm, it's possible. We just need to compute one shift less then described in our algorithm and then compute an extra shift manually. The disadvantges of using it, is that you need to check if cx is different from 0 – to not compute a shift if it isn't needed – and the cost of computing the sticky bit.

cames from floor properties. Indeed if $x < 0$:

$$\text{floor}(-x) = \begin{cases} \text{trunc}(-x) & \text{if } x \in \mathbb{N} \\ \text{trunc}(-x) - 1 & \text{if } x \in \mathbb{R}/\mathbb{N} \end{cases} \tag{11}$$

On one hand, checking if x is negative is quite easy : we just have to take a look at if sy is equal to 1. On the other hand, checking that x is an integer is trickier. This is where the second modification to SecFprUrsh_f comes in handy. As this function performs a rotation, the bit information is retained until a mask is applied to calculate the shift. We therefore decided to invert the mask in order to retain the suppressed information. If this information has zero Hamming weight then x is an integer. We can then use the SecNonZero gadget on this information, which will return 0 if x is an integer, and 1 if not. We denote this result $b = \text{SecNonZero}((my_i^{[52-c_x:1]}))$. In the following truth table (Figure 2) if the result $cp = s \wedge b$ is 1, the mantissa must be changed. Another question is now

sy	b	$cp = sy \wedge b$	Interpretation
0	b	0	Positive number
1	0	0	x is an integer
1	1	1	Negative real

Fig. 2. $cp = sy \wedge b$ truth table and interpretation

: how to subtract s from the mantissa if the mantissa is always positive? The sign of x is only define by sy, so $\mid x \mid = 2^{ey-1023} \times (1 + my \times 2^{-52})$. We already know that if x is positive or an integer,$my = my \pm 0 = my \pm cp$. Now let's focus on x a non integer negative number. Let x be a positive real, from Equation 11, we can write :

$$my = \mid \text{floor}(-x) \mid = \mid \text{trunc}(-x) - 1 \mid \tag{12}$$

By using $\mid \text{trunc}(-x) \mid = \mid -\text{trunc}(x) \mid$, we can transform Equation 12 :

$$my = \mid \text{floor}(-x) \mid = \mid -\text{trunc}(x) - 1 \mid = \mid \text{trunc}(x) + 1 \mid = my + cp \tag{13}$$

From this, we only have to compute a secure addition between cp and my. The last step of this algorithm is to add the shift cd to ey to keep this data update.

$\textbf{SetExponentZero}_{\textbf{floor}}$ – algorithm 8 This last function is useful 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. It must therefore be possible to update ey and sy if necessary. For the floor function, we need the sy sign bit and the b mantissa zero condition. The desired result is zero only if $\mid x \mid < 1$ and $sy = 0$. Recall that if $sy = 1$ and $\mid x \mid < 1$, $\text{floor}(x) = -1$. Minus 1's encoding is $sy = 1$, $ey = 1023$ and $my = 0$.

```
Algorithm 7: SecFprUrsh_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$((my_i), (m_i))$;
  $(m_i) \leftarrow (\neg(m_i))$;
  $(rot_i) \leftarrow$ SecAnd$((my_i), (m_i))$;
  return $((my'_i), (rot_i))$;
```

```
Algorithm 8: SetExponentZero_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 $((ey_i), (sy_i))$;
```

$-sy$	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 $sy = sy$

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

4 Security Proof

In this section we will cover the *t-SNI* security of our design with $n = t + 1$ shares.

Lemma 2. *The gadget* ***SetExponentZero$_{floor}$*** *(Algorithm 8) is* t-SNI *secure.*

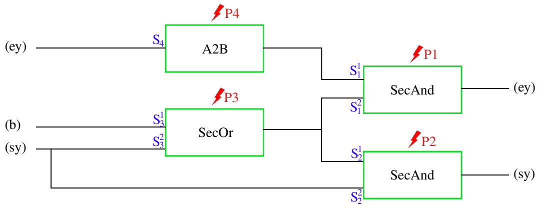

Fig. 4. Abstract diagram of SetExponentZero

Proof. We use an abstract diagram in Figure 4 for our demonstration. The gadget only contains *t-SNI* gadgets and the input shares are distinct from the output shares for each gadgets. By composition of *t-SNI* gadgets and independence of input and output shares, **SetExponentZero**$_{floor}$ is itself *t-SNI*.

Lemma 3. *The gadget* ***SecFprUrsh$_{floor}$*** *(Algorithm 7) is* t-SNI *secure.*

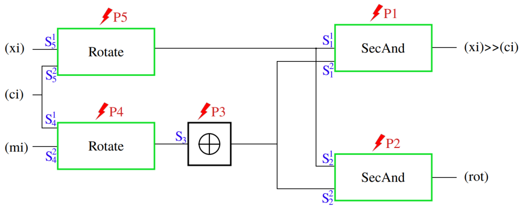

Fig. 5. Abstract diagram of SecFprUrsh$_{floor}$

Proof. The gadget **SecFprUrsh$_{floor}$** is a slight modification of the gadget **SecFprUrsh** from [6]. Ours does not compute the sticky bit but retains the rotated out information. We rely on their proof regarding the *t-SNI* security of the gadget **Rotate** (see [6], Lemma 3 and Figure 2). We now show that the operations following the rotation loop are *t-SNI* secure. We use an abstract diagram in Figure 4 for the demonstration. Let an adversary probe the intermediate values sets P_1 of **SecAnd**, P_2 of **SecAnd** and P_3 of **XOR**. As **SecAnd** is *t-SNI* secure, one can use the sets S_2^1,S_2^2 (resp. S_1^1,S_1^2) to simulate P_2 (resp. P_1) and the ouput shares of (rot) (resp. $(xi) >> (ci)$) with sizes no more than P_2 (resp. P_1). One can simulate the probing set of P_3 in the **XOR** and the simulation sets S_2^2 and S_1^2 with the output shares S_3 of the rotation of (mi). 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 into the rotation loop, can be simulated by input shares due to the *t-SNI* security showed at first in ([6], Lemma 3).

4.1 RemoveDecimal$_{\mathbf{floor}}$

FAIRE LA PREUVE

Lemma 4. *The gadget* ***RemoveDecimal$_{floor}$*** *(Algorithm 6) is* t-SNI *secure.*

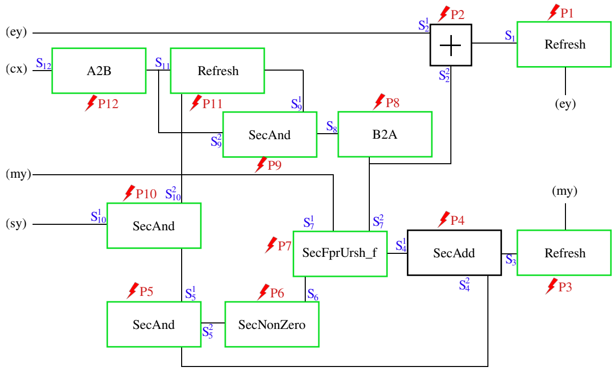

Fig. 6. Abstract diagram of RemoveDecimal$_{\text{floor}}$

Proof. We use an abstract diagram in Figure 4.1 for the demonstration. Let assume an adversary probe the intermediate values sets of the output shares O and P_i in each gadget for $i \in [\![1;12]\!]$. We use simulation sets S_i^j to simulate the values for each gadget. *t-SNI* security implies that if the size of all probing sets P_i is $t_I \leq t$ and if the size of values required to simulate in each gadget is smaller than t, then the simulation sets linked to the input shares are not bigger than t_I. *t-SNI* gadgets imply $|S| \leq |P|$ while *t-NI* gadgets imply $|S| \leq |P| + |O|$. As **Refresh**, **SecAnd**, **SecNonZero**, **SecFprUrsh**$_{floor}$, **B2A** and **A2B** are all *t-SNI* secure while **SecAdd** and + are *t-NI* secure, we can sequentially derive the following:

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

Based on the previous inequalities, one can check that no gadget requires more than t_I values to be simulated. This apply to the input shares as well, with $|S_{10}^1| \le |P_10|$ for (sy), $|S_7^1| \le |P_7|$ for (my), $|S_12| \le |P_12|$ for (cx) and $|S_2^1| \le |P_2| + |S_1| \le |P_2| + |P_1|$ for (ey), no sizes being more than t_I.

Theorem 1. *The gadget* ***SecBaseInt$_{\mathit{floor}}$*** *(Algorithm 5) is* t-SNI *secure.*

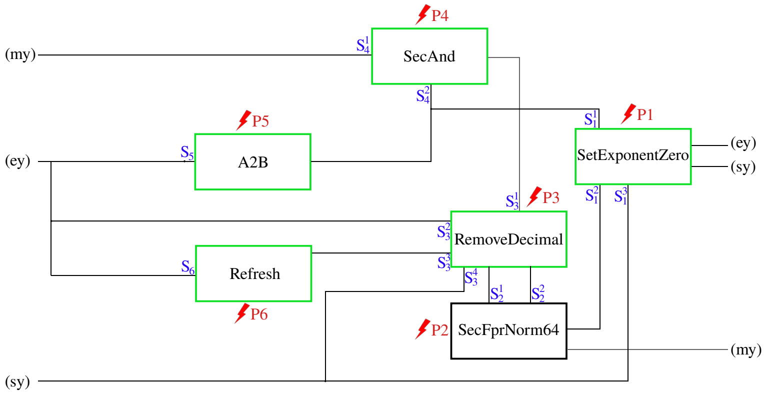

Fig. 7. Abstract diagram of SecBaseInt$_{\text{floor}}$

Proof. We use the same method as for the demonstration of Lemma 4. We use an abstract diagram in Figure 1 for the demonstration. Let assume an adversary probe the intermediate values sets of the output shares O and P_i in each gadget for $i \in [\![1; 6]\!]$. We use simulation sets S_i^j to simulate the values for each gadget. *t-SNI* security implies that if the size of all probing sets P_i is $t_I \le t$ 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:

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

Based on the previous inequalities, one can check that no gadget requires more than t_I values to be simulated. This also apply 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 Application to FALCON

5.1 Masked Floor Function For FALCON

5.2 Masking the Gaussian Sampler

6 Performances

7 Conclusion

Acknowlegdments

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)

Appendix

Extract

Algorithm 9: SecFprExtract(x)

```
Data: 64-bit boolean shares $(x_i)_{1\leq i\leq n}$ for value x
Result: 64-bit boolean shares $(mx_i)_{1\leq i\leq n}$ for mantissa value mx;
16-bit arithmetic shares $(ex_i)_{1\leq i\leq n}$ for exponent value ex;
1-bit boolean shares $(sx_i)_{1\leq i\leq n}$ for sign value s.
$(mx_i) \leftarrow (x_i^{[52:1]})$;
$(mx_i) \leftarrow \mathrm{SecAdd}((mx_i), (2^{52}, 0, \cdots, 0))$; // add implicit bit in the
  mantissa
$(ex_i) \leftarrow (x_i^{[63:53]})$;
$(ex_i) \leftarrow \mathrm{B2A}((ex_i))$;
$(sx_i) \leftarrow (x_i^{(64)})$;
return $((mx_i), (ex_i), (sx_i))$;
```

Truncature Function: Gadgets

```
Algorithm 10: 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 $ey + (52 - cx)$;
  $cx_1 \leftarrow cx_1 - 52$;                                  // check if 0≤c<51
  $(c_i) \leftarrow$ A2B$((cx_i))$;
  $(cp_i) \leftarrow (c_i^{(16)})$;
  $(cp_i) \leftarrow$ SecNonZero$(cp_i)$;
  $(c'_i) \leftarrow (-cp_i)$;                  // if cp = 0 cx = 0. if not cx = cx
  $(c_i) \leftarrow$ SecAnd$((c_i), (cp_i))$;
  $(cx_i) \leftarrow$ B2A$((c_i))$;
  $(cd_i) \leftarrow (-cx_i)$;
  $(my_i) \leftarrow$ SecFprUrsh$_f((my_i), (cd_i))$;                // $my >> 52 - cx$
  $(ey_i) \leftarrow (ey_i + cd_i)$;
  return $((my_i), (ey_i))$;
```

```
Algorithm 11: 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.
  $(ey_i) \leftarrow$ A2B$((ey_i))$;
  $(ey_i) \leftarrow$ SecAnd$((ey_i, b_i))$;
  $(sy_i) \leftarrow$ SecAnd$((sy_i, b_i))$;
  return $((ey_i), (sy_i))$;
```

Round Function: Gadgets

```
Algorithm 12: 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.
  $cx_1 \leftarrow cx_1 - 53$;                                 // check if 0≤c<51
  $(c_i) \leftarrow \text{A2B}((cx_i))$;
  $(cp_i) \leftarrow (c_i^{(16)})$;
  $(cp_i) \leftarrow \text{SecNonZero}(cp_i)$;
  $(rshORnot_i) \leftarrow (\neg cp[i])$;
  $(c'_i) \leftarrow (\neg cp_i)$;                // if cp = 0 cx = 0. if not cx = cx
  $cx_1 \leftarrow cx_1 + 1$;
  $(c_i) \leftarrow \text{A2B}((cx_i))$;
  $(c_i) \leftarrow \text{SecAnd}((c_i),(cp_i))$;
  $(cx_i) \leftarrow \text{B2A}((c_i))$;
  $(cd_i) \leftarrow (-cx_i)$;
  $(my_i) \leftarrow \text{SecFprUrsh}((my_i),(cd_i))$;                 // $my >> 53 - cx$
  $(Rnd_i) \leftarrow (my_i^{(1)})$;
  $(Rnd_i) \leftarrow \text{SecNonZero}(Rnd_i)$;
  $(Rnd_i) \leftarrow \text{SecAnd}((Rnd_i),(rshORnot_i))$;
  $(my1_i) \leftarrow (my_i) >> 1$ ,   $(e1_i) \leftarrow (1,0,\cdots,0)$;
  $(my2_i) \leftarrow (my_i)$        ,   $(e2_i) \leftarrow (0,\cdots,0)$;
  $(my1_i) \leftarrow \text{SecAnd}((my1_i),(rshORnot_i))$  , $(e1_i) \leftarrow$
    $\text{SecAnd}((e1_i),(rshORnot_i))$;
  $(rshORnot_i) \leftarrow (\neg rshORnot_i)$;
  $(my2_i) \leftarrow \text{SecAnd}((my2_i),(rshORnot_i))$  , $(e2_i) \leftarrow$
    $\text{SecAnd}((e2_i),(rshORnot_i))$;
  $(my_i) \leftarrow \text{SecOr}((my1_i),(my2_i))$;
  $(my_i) \leftarrow \text{SecAdd}((my_i),(Rnd_i))$;
  $(e1_i) \leftarrow \text{SecOr}((e1_i),(e2_i))$;
  $(e1_i) \leftarrow \text{B2A}((e1_i))$;
  $(ey_i) \leftarrow (ey_i + e1_i)$;
  $(ey_i) \leftarrow (ey_i + cd_i)$;
  return $((my_i),(ey_i),(Rnd_i))$;
```

```
Algorithm 13: 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.
  $(ey_i) \leftarrow$ A2B$((ey_i))$;
  $(b'_i) \leftarrow (Rnd_i)$;
  $(b'_i) \leftarrow$ SecOr$((b'_i), (b_i))$;
  $(ey_i) \leftarrow$ SecAnd$((ey_i, b'_i))$;
  $(sy_i) \leftarrow$ SecAnd$((sy_i, b'_i))$;
  return $((ey_i), (sy_i))$;
```