Masking Floating-Point Number Multiplication and Addition of Falcon

Keng-Yu Chen, Jiun-Peng Chen

Conference on Cryptographic Hardware and Embedded Systems

September 6th, 2024

Table of Contents

Table of Contents

Introduction

- To defend against the potential threat from large-scale quantum computers, the US National Institute of Standards and Technology (NIST) initiated a standardization process for post-quantum cryptography in 2016.

Introduction

- To defend against the potential threat from large-scale quantum computers, the US National Institute of Standards and Technology (NIST) initiated a standardization process for post-quantum cryptography in 2016.
- In 2022, FALCON [Pre+20] became one of the selected schemes that were expected to be part of NIST's post-quantum cryptographic standards.

Introduction

- To defend against the potential threat from large-scale quantum computers, the US National Institute of Standards and Technology (NIST) initiated a standardization process for post-quantum cryptography in 2016.
- In 2022, Falcon [Pre+20] became one of the selected schemes that were expected to be part of NIST's post-quantum cryptographic standards.
- In theory, these algorithms can base their security on problems that are considered still hard given the advantage of quantum computing.

Introduction

- To defend against the potential threat from large-scale quantum computers, the US National Institute of Standards and Technology (NIST) initiated a standardization process for post-quantum cryptography in 2016.
- In 2022, FALCON [Pre+20] became one of the selected schemes that were expected to be part of NIST's post-quantum cryptographic standards.
- In theory, these algorithms can base their security on problems that are considered still hard given the advantage of quantum computing.
- In practice, the implementations of these algorithms can suffer side-channel attacks.

Side-channel Attacks on FALCON

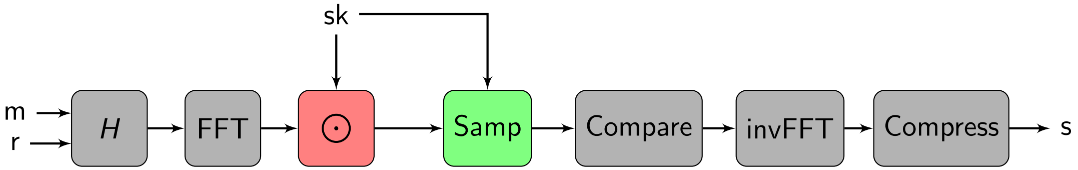

Figure: A graphical overview of FALCON.Sign.

	Attack	Countermeasure
Pre-image Vector Computation	[KA21; Gue+22]	
Gaussian Sampler over Lattices	[Gue+22; Zha+23]	[Gue+22; Zha+23]

Side-channel Attacks on FALCON

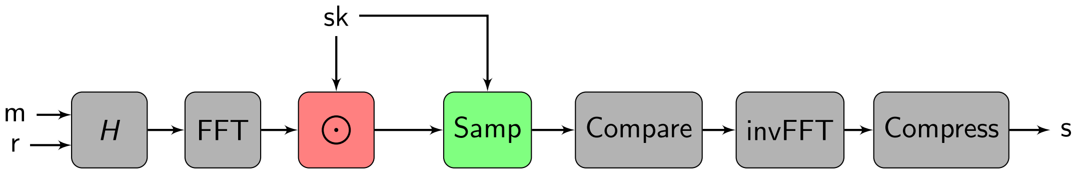

Figure: A graphical overview of FALCON.Sign.

	Attack	Countermeasure
Pre-image Vector Computation	[KA21; Gue+22]	**This Paper**
Gaussian Sampler over Lattices	[Gue+22; Zha+23]	[Gue+22; Zha+23]

Our Contribution: The first masking scheme on the the pre-image vector computation as a countermeasure against current attacks.

Notation

We assume

Notation

We assume

- For a variable x, the jth bit of x is written as $x^{(j)}$.

Notation

We assume

- For a variable x, the jth bit of x is written as $x^{(j)}$.
- The ith bit to jth bit $(j \geq i)$ of x is represented by $x^{[j:i]}$.

Notation

We assume

- For a variable x, the jth bit of x is written as $x^{(j)}$.
- The ith bit to jth bit ($j \geq i$) of x is represented by $x^{[j:i]}$.
- A sequence of n variables $(x_1, x_2, \cdots, x_n)$ (e.g. shares of variable x) is written as $(x_i)_{1 \leq i \leq n}$, or simply (x_i).

Notation

We assume

- For a variable x, the jth bit of x is written as $x^{(j)}$.
- The ith bit to jth bit ($j \geq i$) of x is represented by $x^{[j:i]}$.
- A sequence of n variables $(x_1, x_2, \cdots, x_n)$ (e.g. shares of variable x) is written as $(x_i)_{1 \leq i \leq n}$, or simply (x_i).
- For a proposition P, $[\![P]\!] = 1$ if and only if P is true and 0 if otherwise.

Table of Contents

Table of Contents

Introduction to FALCON

KeyGen

Sign(m)

Verify(m, **s**)

Introduction to FALCON

KeyGen

Secret Key: Short polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^N + 1)$ such that $fG - gF = q$ and

$$\mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$$

Sign(m)

Verify(m, $\mathbf{s}$)

Introduction to FALCON

KeyGen

Secret Key: Short polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^N + 1)$ such that $fG - gF = q$ and

$$\mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$$

Public Key: Polynomial $h = gf^{-1} \bmod q$ and

$$\mathbf{A} = \left[\begin{array}{c|c} 1 & h \end{array}\right]$$

Note that

$$\mathbf{B}\mathbf{A}^T = \mathbf{0} \bmod q$$

Sign(m)

Verify(m, **s**)

Introduction to FALCON

KeyGen

Secret Key: Short polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^N + 1)$ such that $fG - gF = q$ and

$$\mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$$

Public Key: Polynomial $h = gf^{-1} \bmod q$ and

$$\mathbf{A} = \left[\begin{array}{c|c} 1 & h \end{array}\right]$$

Note that

$$\mathbf{B}\mathbf{A}^T = \mathbf{0} \bmod q$$

Sign(m)

A short signature $\mathbf{s}$ such that

$$\mathbf{s}\mathbf{A}^T = H(\mathrm{r}\|\mathrm{m}) \bmod q$$

where H is a hash function and r is the random salt

Verify(m, **s**)

Introduction to FALCON

KeyGen

Secret Key: Short polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^N + 1)$ such that $fG - gF = q$ and

$$\mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$$

Public Key: Polynomial $h = gf^{-1} \bmod q$ and

$$\mathbf{A} = \left[\begin{array}{c|c} 1 & h \end{array}\right]$$

Note that

$$\mathbf{BA}^T = \mathbf{0} \bmod q$$

Sign(m)

A short signature **s** such that

$$\mathbf{sA}^T = H(\mathrm{r}\|\mathrm{m}) \bmod q$$

where H is a hash function and r is the random salt

Verify(m, **s**)

Check

1. **s** is short
2. $\mathbf{sA}^T = H(\mathrm{r}\|\mathrm{m}) \bmod q$

Introduction to FALCON

To find such a short **s**, one can

Introduction to FALCON

To find such a short **s**, one can

- Compute $c \leftarrow H(\mathrm{r} \| \mathrm{m})$.

Introduction to FALCON

To find such a short **s**, one can

- Compute $c \leftarrow H(\mathrm{r}\|\mathrm{m})$.
- Find a solution $\mathbf{c} \leftarrow \left[\begin{array}{c|c} c & 0 \end{array}\right]$ such that $\mathbf{cA}^T = H(\mathrm{r}\|\mathrm{m}) \bmod q$.

Introduction to FALCON

To find such a short $\mathbf{s}$, one can

- Compute $c \leftarrow H(\mathsf{r}\|\mathsf{m})$.
- Find a solution $\mathbf{c} \leftarrow \left[\begin{array}{c|c} c & 0 \end{array}\right]$ such that $\mathbf{cA}^T = H(\mathsf{r}\|\mathsf{m}) \bmod q$.
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{cB}^{-1}$.

Introduction to FALCON

To find such a short $\mathbf{s}$, one can

- Compute $c \leftarrow H(\mathrm{r} \| \mathrm{m})$.
- Find a solution $\mathbf{c} \leftarrow \left[\begin{array}{c|c} c & 0 \end{array}\right]$ such that $\mathbf{c}\mathbf{A}^T = H(\mathrm{r} \| \mathrm{m}) \bmod q$.
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$.
- Apply the nearest plane algorithm [DP16] to find a vector $\mathbf{z}$ such that $(\mathbf{t} - \mathbf{z})\mathbf{B}$ is short.

Introduction to FALCON

To find such a short $\mathbf{s}$, one can

- Compute $c \leftarrow H(\mathrm{r}\|\mathrm{m})$.
- Find a solution $\mathbf{c} \leftarrow \left[\begin{array}{c|c} c & 0 \end{array}\right]$ such that $\mathbf{c}\mathbf{A}^T = H(\mathrm{r}\|\mathrm{m}) \bmod q$.
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$.
- Apply the nearest plane algorithm [DP16] to find a vector $\mathbf{z}$ such that $(\mathbf{t} - \mathbf{z})\mathbf{B}$ is short.
- $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\mathbf{B}$. Note that $\mathbf{s}\mathbf{A}^T = H(\mathrm{r}\|\mathrm{m}) \bmod q$.

Introduction to FALCON

To find such a short $\mathbf{s}$, one can

- Compute $c \leftarrow H(\mathrm{r}\|\mathrm{m})$.
- Find a solution $\mathbf{c} \leftarrow \left[\begin{array}{c|c} c & 0 \end{array}\right]$ such that $\mathbf{c}\mathbf{A}^T = H(\mathrm{r}\|\mathrm{m}) \bmod q$.
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$.
- Apply the nearest plane algorithm [DP16] to find a vector $\mathbf{z}$ such that $(\mathbf{t} - \mathbf{z})\mathbf{B}$ is short.
- $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\mathbf{B}$. Note that $\mathbf{s}\mathbf{A}^T = H(\mathrm{r}\|\mathrm{m}) \bmod q$.

Table of Contents

Fast-Fourier Transform

The pre-image vector computation includes polynomial multiplications

$$\mathbf{t} = \mathbf{c}\mathbf{B}^{-1} = \frac{1}{q}\left[\ c \cdot -F \ \middle|\ c \cdot f\ \right]$$

Fast-Fourier Transform

The pre-image vector computation includes polynomial multiplications

$$\mathbf{t} = \mathbf{c}\mathbf{B}^{-1} = \frac{1}{q}\left[\ c \cdot -F \ \middle|\ c \cdot f\ \right]$$

To speed up, the pre-image vector computation is performed after the Fourier transform:

$$\frac{1}{q}\left[\ \mathrm{FFT}(c) \odot \mathrm{FFT}(-F) \ \middle|\ \mathrm{FFT}(c) \odot \mathrm{FFT}(f)\ \right]$$

Fast-Fourier Transform

The pre-image vector computation includes polynomial multiplications

$$\mathbf{t} = \mathbf{cB}^{-1} = \frac{1}{q}\left[\ c \cdot -F \ \middle|\ c \cdot f\ \right]$$

To speed up, the pre-image vector computation is performed after the Fourier transform:

$$\frac{1}{q}\left[\ \mathrm{FFT}(c) \odot \mathrm{FFT}(-F) \ \middle|\ \mathrm{FFT}(c) \odot \mathrm{FFT}(f)\ \right]$$

Therefore, the pre-image vector computation is essentially coefficient-wise complex number multiplications.

Floating-Point Number

A complex number is represented by two 64-bit floating-point numbers (FPNs), its real and imaginary parts.

Floating-Point Number

A complex number is represented by two 64-bit floating-point numbers (FPNs), its real and imaginary parts.

An FPN is composed of sign bit s, exponent e, and mantissa $\tilde{m}$

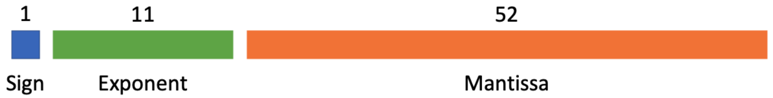

Figure: A 64-bit Floating-Point Number

Floating-Point Number

A complex number is represented by two 64-bit floating-point numbers (FPNs), its real and imaginary parts.

An FPN is composed of sign bit s, exponent e, and mantissa $\tilde{m}$

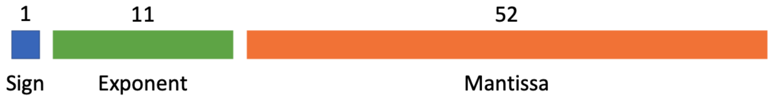

Figure: A 64-bit Floating-Point Number

The value is $(-1)^s \cdot 2^{e-1023} \cdot \underbrace{(1 + \tilde{m} \cdot 2^{-52})}_{\times 2^{52} = m}$

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

FPN addition (FprAdd) is proceeded by

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR

FPN addition (FprAdd) is proceeded by

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition

FPN addition (FprAdd) is proceeded by

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition
3. Mantissa multiplication

FPN addition (FprAdd) is proceeded by

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition
3. Mantissa multiplication
4. Right-shifting the result

FPN addition (FprAdd) is proceeded by

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition
3. Mantissa multiplication
4. Right-shifting the result
5. Packing and rounding (FPR)

FPN addition (FprAdd) is proceeded by

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition
3. Mantissa multiplication
4. Right-shifting the result
5. Packing and rounding (FPR)

FPN addition (FprAdd) is proceeded by

1. Making the first operand $\geq$ the second

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition
3. Mantissa multiplication
4. Right-shifting the result
5. Packing and rounding (FPR)

FPN addition (FprAdd) is proceeded by

1. Making the first operand $\geq$ the second
2. Right-shifting the second operand

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition
3. Mantissa multiplication
4. Right-shifting the result
5. Packing and rounding (FPR)

FPN addition (FprAdd) is proceeded by

1. Making the first operand $\geq$ the second
2. Right-shifting the second operand
3. Mantissa addition / subtraction

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition
3. Mantissa multiplication
4. Right-shifting the result
5. Packing and rounding (FPR)

FPN addition (FprAdd) is proceeded by

1. Making the first operand $\geq$ the second
2. Right-shifting the second operand
3. Mantissa addition / subtraction
4. Normalizing the result

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

1. Sign bit XOR
2. Exponent addition
3. Mantissa multiplication
4. Right-shifting the result
5. Packing and rounding (FPR)

FPN addition (FprAdd) is proceeded by

1. Making the first operand $\geq$ the second
2. Right-shifting the second operand
3. Mantissa addition / subtraction
4. Normalizing the result
5. Packing and rounding (FPR)

Table of Contents

Masking

Masking resists side-channel attacks by secret-sharing the sensitive variables.

Masking

Masking resists side-channel attacks by secret-sharing the sensitive variables.

- Boolean Masking: A variable x is split into n shares $(x_i)_{1 \leq i \leq n}$ such that

$$x = \bigoplus_{i=1}^{n} x_i$$

Masking

Masking resists side-channel attacks by secret-sharing the sensitive variables.

- Boolean Masking: A variable x is split into n shares $(x_i)_{1\leq i\leq n}$ such that

$$x = \bigoplus_{i=1}^{n} x_i$$

- Arithmetic Masking: A variable x is split into n shares $(x_i)_{1\leq i\leq n}$ (in k-bit registers) such that

$$x = \sum_{i=1}^{n} x_i \pmod{2^k}$$

Masking

Masking resists side-channel attacks by secret-sharing the sensitive variables.

- Boolean Masking: A variable x is split into n shares $(x_i)_{1\leq i\leq n}$ such that

$$x = \bigoplus_{i=1}^{n} x_i$$

- Arithmetic Masking: A variable x is split into n shares $(x_i)_{1\leq i\leq n}$ (in k-bit registers) such that

$$x = \sum_{i=1}^{n} x_i \pmod{2^k}$$

In each run, all x_i's are freshly randomized.

Table of Contents

Table of Contents

Overview of Our Approach

An intuitive approach to mask any algorithm:

Overview of Our Approach

An intuitive approach to mask any algorithm:

- For operations like $\wedge, \oplus$: Boolean masking

Overview of Our Approach

An intuitive approach to mask any algorithm:

- For operations like $\wedge, \oplus$: Boolean masking
- For operations like $+, \times$: arithmetic masking

Overview of Our Approach

An intuitive approach to mask any algorithm:

- For operations like $\land, \oplus$: Boolean masking
- For operations like $+, \times$: arithmetic masking

and use the following gadgets if necessary:

Overview of Our Approach

An intuitive approach to mask any algorithm:

- For operations like $\wedge, \oplus$: Boolean masking
- For operations like $+, \times$: arithmetic masking

and use the following gadgets if necessary:

- A2B: $(x_i)_{1\leq i\leq n} \mapsto (y_i)_{1\leq i\leq n}$ such that $\sum_{i=1}^{n} x_i = \bigoplus_{i=1}^{n} y_i$

Overview of Our Approach

An intuitive approach to mask any algorithm:

- For operations like $\wedge, \oplus$: Boolean masking
- For operations like $+, \times$: arithmetic masking

and use the following gadgets if necessary:

- A2B: $(x_i)_{1\leq i\leq n} \mapsto (y_i)_{1\leq i\leq n}$ such that $\sum_{i=1}^{n} x_i = \bigoplus_{i=1}^{n} y_i$
- B2A: $(y_i)_{1\leq i\leq n} \mapsto (x_i)_{1\leq i\leq n}$ such that $\bigoplus_{i=1}^{n} y_i = \sum_{i=1}^{n} x_i$

Overview of Our Approach

However, some operations in floating-point number arithmetic cannot be easily implemented in this way:

Overview of Our Approach

However, some operations in floating-point number arithmetic cannot be easily implemented in this way:

- Checking whether a secret value is nonzero
  - Given $(x_i)_{1\leq i\leq n}$, checking whether $\bigoplus_{i=1}^{n} x_i \neq 0$ or $\sum_{i=1}^{n} x_i \neq 0$

Overview of Our Approach

However, some operations in floating-point number arithmetic cannot be easily implemented in this way:

- Checking whether a secret value is nonzero
    - Given $(x_i)_{1\leq i\leq n}$, checking whether $\bigoplus_{i=1}^{n} x_i \neq 0$ or $\sum_{i=1}^{n} x_i \neq 0$
- Right-shifting a secret value by another secret value
    - Given $(x_i)_{1\leq i\leq n}$ and $(c_i)_{1\leq i\leq n}$, right-shifting $(x_i)_{1\leq i\leq n}$ by $(c_i)_{1\leq i\leq n}$

Overview of Our Approach

However, some operations in floating-point number arithmetic cannot be easily implemented in this way:

- Checking whether a secret value is nonzero
  - Given $(x_i)_{1\leq i\leq n}$, checking whether $\bigoplus_{i=1}^{n} x_i \neq 0$ or $\sum_{i=1}^{n} x_i \neq 0$
- Right-shifting a secret value by another secret value
  - Given $(x_i)_{1\leq i\leq n}$ and $(c_i)_{1\leq i\leq n}$, right-shifting $(x_i)_{1\leq i\leq n}$ by $(c_i)_{1\leq i\leq n}$
- Normalizing a secret value to $[2^{63}, 2^{64})$
  - Given $(x_i)_{1\leq i\leq n}$, left-shifting $(x_i)_{1\leq i\leq n}$ until its 64th bit is set

Overview of Our Approach

We design novel gadgets for these three operations, including

Overview of Our Approach

We design novel gadgets for these three operations, including

- SecNonzero: securely checking whether a secret value is nonzero

Overview of Our Approach

We design novel gadgets for these three operations, including

- SecNonzero: securely checking whether a secret value is nonzero
- SecFprUrsh: securely right-shifting a secret value by another secret value

Overview of Our Approach

We design novel gadgets for these three operations, including

- SecNonzero: securely checking whether a secret value is nonzero
- SecFprUrsh: securely right-shifting a secret value by another secret value
- SecFprNorm64: securely normalizing a secret value to $[2^{63}, 2^{64})$

Overview of Our Approach

We design novel gadgets for these three operations, including

- SecNonzero: securely checking whether a secret value is nonzero
- SecFprUrsh: securely right-shifting a secret value by another secret value
- SecFprNorm64: securely normalizing a secret value to $[2^{63}, 2^{64})$

In addition, we make several improvements to reduce the costs.

Table of Contents

SecNonzero

We need a gadget that, given shares $(x_i)_{i=1}^n$, outputs one-bit shares $(b_i)_{i=1}^n$ such that

$$\left[\!\left[\bigoplus_{i=1}^{n} x_i \neq 0\right]\!\right] = \bigoplus_{i=1}^{n} b_i \quad \text{or} \quad \left[\!\left[\sum_{i=1}^{n} x_i \neq 0\right]\!\right] = \bigoplus_{i=1}^{n} b_i$$

SecNonzero

We need a gadget that, given shares $(x_i)_{i=1}^n$, outputs one-bit shares $(b_i)_{i=1}^n$ such that

$$\left[\!\!\left[\bigoplus_{i=1}^{n} x_i \neq 0\right]\!\!\right] = \bigoplus_{i=1}^{n} b_i \quad \text{or} \quad \left[\!\!\left[\sum_{i=1}^{n} x_i \neq 0\right]\!\!\right] = \bigoplus_{i=1}^{n} b_i$$

For Boolean shares, our method is by OR-ing all the bits.

$$x = 0 \iff x^{(k)} \vee x^{(k-1)} \vee \cdots \vee x^{(1)} = 0$$

SecNonzero

We need a gadget that, given shares $(x_i)_{i=1}^n$, outputs one-bit shares $(b_i)_{i=1}^n$ such that

$$\left[\!\!\left[\bigoplus_{i=1}^{n} x_i \neq 0\right]\!\!\right] = \bigoplus_{i=1}^{n} b_i \quad \text{or} \quad \left[\!\!\left[\sum_{i=1}^{n} x_i \neq 0\right]\!\!\right] = \bigoplus_{i=1}^{n} b_i$$

For Boolean shares, our method is by OR-ing all the bits.

$$x = 0 \iff x^{(k)} \vee x^{(k-1)} \vee \cdots \vee x^{(1)} = 0$$

Then we need a gadget for secure OR operations, which can be constructed by applying the De Morgan's law and a SecAnd gadget [ISW03; Bar+16].

SecNonzero

For arithmetic shares, instead of applying an n-shared A2B, we consider that

$$\sum_{i=1}^{n} x_i = 0 \Longleftrightarrow \sum_{i=1}^{\frac{n}{2}} x_i \oplus \sum_{i=\frac{n}{2}+1}^{n} (-x_i) = 0$$

SecNonzero

For arithmetic shares, instead of applying an n-shared A2B, we consider that

$$\sum_{i=1}^{n} x_i = 0 \Longleftrightarrow \sum_{i=1}^{\frac{n}{2}} x_i \oplus \sum_{i=\frac{n}{2}+1}^{n} (-x_i) = 0$$

$$\mathsf{SecNonzero}_{\mathsf{arith}}(x_1, \cdots, x_n) = \mathsf{SecNonzero}_{\mathsf{Bool}}(\mathsf{A2B}(x_1, \cdots, x_{\frac{n}{2}}),\ \mathsf{A2B}(x_{\frac{n}{2}+1}, \cdots, x_n))$$

SecNonzero

For arithmetic shares, instead of applying an n-shared A2B, we consider that

$$\sum_{i=1}^{n} x_i = 0 \Longleftrightarrow \sum_{i=1}^{\frac{n}{2}} x_i \oplus \sum_{i=\frac{n}{2}+1}^{n} (-x_i) = 0$$

$$\text{SecNonzero}_{\text{arith}}(x_1, \cdots, x_n) = \text{SecNonzero}_{\text{Bool}}(\text{A2B}(x_1, \cdots, x_{\frac{n}{2}}),\ \text{A2B}(x_{\frac{n}{2}+1}, \cdots, x_n))$$

In this way, we replace one n-shared A2B with two $n/2$-shared A2Bs.

Table of Contents

SecFprUrsh

Given 64-bit shares (x_i) and 6-bit shares (c_i), we need to derive shares (z_i) such that

$$\bigoplus_{i=1}^{n} z_i = \left(\bigoplus_{i=1}^{n} x_i\right) \gg \left(\sum_{i=1}^{n} c_i \bmod 64\right)$$

SecFprUrsh

Given 64-bit shares (x_i) and 6-bit shares (c_i), we need to derive shares (z_i) such that

$$\bigoplus_{i=1}^{n} z_i = \left(\bigoplus_{i=1}^{n} x_i\right) \gg \left(\sum_{i=1}^{n} c_i \bmod 64\right)$$

We observe that

SecFprUrsh

Given 64-bit shares (x_i) and 6-bit shares (c_i), we need to derive shares (z_i) such that

$$\bigoplus_{i=1}^{n} z_i = \left(\bigoplus_{i=1}^{n} x_i\right) \gg \left(\sum_{i=1}^{n} c_i \bmod 64\right)$$

We observe that

- Right-shifting and right-rotating by a value c only differ by the most c significant bits.

SecFprUrsh

Given 64-bit shares (x_i) and 6-bit shares (c_i), we need to derive shares (z_i) such that

$$\bigoplus_{i=1}^{n} z_i = \left(\bigoplus_{i=1}^{n} x_i\right) \gg \left(\sum_{i=1}^{n} c_i \bmod 64\right)$$

We observe that

- Right-shifting and right-rotating by a value c only differ by the most c significant bits.
- Right-rotating x by a value c is equal to right-rotating x by c mod 64.

SecFprUrsh

Hence, our idea is to right-rotate all $x_i's$ by $c_1, c_2, \cdots, c_n$ sequentially.

SecFprUrsh

Hence, our idea is to right-rotate all $x_i's$ by $c_1, c_2, \cdots, c_n$ sequentially.

Some high bits are redundant, so we use an index $m = (1 \ll 63)$ to indicate the first meaningful bit of the result.

SecFprUrsh

Hence, our idea is to right-rotate all $x_i's$ by $c_1, c_2, \cdots, c_n$ sequentially.

Some high bits are redundant, so we use an index $m = (1 \ll 63)$ to indicate the first meaningful bit of the result. Consider

$$m \gg c = (\underbrace{0 \cdots 0}_{c \text{ bits}} 10 \cdots 0)_2$$

SecFprUrsh

Hence, our idea is to right-rotate all $x_i's$ by $c_1, c_2, \cdots, c_n$ sequentially.

Some high bits are redundant, so we use an index $m = (1 \ll 63)$ to indicate the first meaningful bit of the result. Consider

$$m \gg c = (\underbrace{0 \cdots 0}_{c \text{ bits}} 10 \cdots 0)_2$$

$$m' := m \gg c \oplus (m \gg c) \gg 1 \oplus \cdots \oplus (m \gg c) \gg 63 = (\underbrace{0 \cdots 0}_{c \text{ bits}} 11 \cdots 1)_2$$

SecFprUrsh

Hence, our idea is to right-rotate all $x_i's$ by $c_1, c_2, \cdots, c_n$ sequentially.

Some high bits are redundant, so we use an index $m = (1 \ll 63)$ to indicate the first meaningful bit of the result. Consider

$$m \gg c = (\underbrace{0 \cdots 0}_{c \text{ bits}} 10 \cdots 0)_2$$

$$m' := m \gg c \oplus (m \gg c) \gg 1 \oplus \cdots \oplus (m \gg c) \gg 63 = (\underbrace{0 \cdots 0}_{c \text{ bits}} 11 \cdots 1)_2$$

By an AND operation with m', we can clear useless bits.

Table of Contents

SecFprNorm64

Given 64-bit shares (x_i) and 16-bit shares (e_i), we need to derive shares (x_i') and (e_i') such that

$$\bigoplus_{i=1}^{n} x_i' = \bigoplus_{i=1}^{n} x_i \ll c \text{ and } \sum_{i=1}^{n} e_i' = (\sum_{i=1}^{n} e_i) - c$$

where c is the smallest integer such that $\bigoplus_{i=1}^{n} x_i \ll c \in [2^{63}, 2^{64})$

SecFprNorm64

Given 64-bit shares (x_i) and 16-bit shares (e_i), we need to derive shares (x'_i) and (e'_i) such that

$$\bigoplus_{i=1}^{n} x'_i = \bigoplus_{i=1}^{n} x_i \ll c \text{ and } \sum_{i=1}^{n} e'_i = (\sum_{i=1}^{n} e_i) - c$$

$$\text{where } c \text{ is the smallest integer such that } \bigoplus_{i=1}^{n} x_i \ll c \in [2^{63}, 2^{64})$$

We repeatedly check whether $(x_i^{(64)})$ is 0 or not, then conditionally shift it by 1 bit, and then decrease (e_i) by $[\![(x_i^{(64)}) = 0]\!]$.

SecFprNorm64

Given 64-bit shares (x_i) and 16-bit shares (e_i), we need to derive shares (x_i') and (e_i') such that

$$\bigoplus_{i=1}^{n} x_i' = \bigoplus_{i=1}^{n} x_i \ll c \text{ and } \sum_{i=1}^{n} e_i' = (\sum_{i=1}^{n} e_i) - c$$

$$\text{where } c \text{ is the smallest integer such that } \bigoplus_{i=1}^{n} x_i \ll c \in [2^{63}, 2^{64})$$

We repeatedly check whether $(x_i^{(64)})$ is 0 or not, then conditionally shift it by 1 bit, and then decrease (e_i) by $[\![(x_i^{(64)}) = 0]\!]$.

To improve efficiency, we sequentially check $x^{[64:64-2^j]} = 0$ for $j = 5, 4, \cdots, 0$.

Wrapping-up

Utilizing SecNonzero, SecFprUrsh, and SecFprNorm64, we design the following gadgets:

Wrapping-up

Utilizing SecNonzero, SecFprUrsh, and SecFprNorm64, we design the following gadgets:

- SecFPR: Secure FPR (FPN packing and rounding) by masking.

Wrapping-up

Utilizing SecNonzero, SecFprUrsh, and SecFprNorm64, we design the following gadgets:

- SecFPR: Secure FPR (FPN packing and rounding) by masking.
- SecFprMul: Secure FprMul (FPN multiplication) by masking.

Wrapping-up

Utilizing SecNonzero, SecFprUrsh, and SecFprNorm64, we design the following gadgets:

- SecFPR: Secure FPR (FPN packing and rounding) by masking.
- SecFprMul: Secure FprMul (FPN multiplication) by masking.
- SecFprAdd: Secure FprAdd (FPN addition) by masking.

Wrapping-up

Utilizing SecNonzero, SecFprUrsh, and SecFprNorm64, we design the following gadgets:

- SecFPR: Secure FPR (FPN packing and rounding) by masking.
- SecFprMul: Secure FprMul (FPN multiplication) by masking.
- SecFprAdd: Secure FprAdd (FPN addition) by masking.

We leave the details of the implementations and several tricks for improvements in our paper.

Table of Contents

Table of Contents

Theoretical Security – Probing Model

For a positive integer t,

- The t-probing model [ISW03] assumes that an adversary is able to peek any t intermediate values in the algorithm.

Theoretical Security – Probing Model

For a positive integer t,

- The t-probing model [ISW03] assumes that an adversary is able to peek any t intermediate values in the algorithm.
- To be secure in t-probing model, the number of shares $n \geq t + 1$, and any share cannot be combined with each other.

Theoretical Security – Probing Model

For a positive integer t,

- The t-probing model [ISW03] assumes that an adversary is able to peek any t intermediate values in the algorithm.
- To be secure in t-probing model, the number of shares $n \geq t + 1$, and any share cannot be combined with each other.
- It can be complicated to prove t-probing security for a large composition of gadgets. We apply the concept of non-interference.

Non-Interference Security

t-Non-Interference (t-NI) Security (from [Bar+16])

A gadget is 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.

Non-Interference Security

t-Non-Interference (t-NI) Security (from [Bar+16])

A gadget is 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-Strong Non-Interference (t-SNI) Security (from [Bar+16])

A gadget is t-Strong-Non-Interference (t-SNI) secure if for every set of t_I 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.

Non-Interference Security

- For $t = n - 1$, if a gadget is t-NI or t-SNI secure, and if any $n - 1$ input shares are uniformly and independently distributed, then it is t-probing secure.

Non-Interference Security

- For $t = n - 1$, if a gadget is t-NI or t-SNI secure, and if any $n - 1$ input shares are uniformly and independently distributed, then it is t-probing secure.
- All the gadgets in our paper are proven either t-NI or t-SNI secure.

Gadget	**Security**	**Gadget**	**Security**
SecOr	t-SNI	SecNonzero	t-SNI
SecFprUrsh	t-SNI	SecFprNorm64	t-NI
SecFPR	t-SNI	SecFprMul	t-SNI
SecFprAdd	t-SNI		

Table: List of gadgets in our work with $n = t + 1$ shares

Practical Security – Test Vector Leakage Assessment (TVLA)

In TVLA, one records two sets of power or electromagnetic traces where

Practical Security – Test Vector Leakage Assessment (TVLA)

In TVLA, one records two sets of power or electromagnetic traces where

- Set 1: fixed input

Practical Security – Test Vector Leakage Assessment (TVLA)

In TVLA, one records two sets of power or electromagnetic traces where

- Set 1: fixed input
- Set 2: random inputs

Practical Security – Test Vector Leakage Assessment (TVLA)

In TVLA, one records two sets of power or electromagnetic traces where

- Set 1: fixed input
- Set 2: random inputs

and then applies the Welch's *t*-test on the two sets.

Practical Security – Test Vector Leakage Assessment (TVLA)

In TVLA, one records two sets of power or electromagnetic traces where

- Set 1: fixed input
- Set 2: random inputs

and then applies the Welch's t-test on the two sets.

By convention, we consider the leakage significant if the t-value exceeds ± 4.5.

Practical Security – Test Vector Leakage Assessment (TVLA)

In TVLA, one records two sets of power or electromagnetic traces where

- Set 1: fixed input
- Set 2: random inputs

and then applies the Welch's t-test on the two sets.

By convention, we consider the leakage significant if the t-value exceeds ± 4.5.

For long traces, we refer to [Din+17] to alter this threshold to avoid false positives.

TVLA results of floating-point number multiplication (FprMul, SecFprMul)

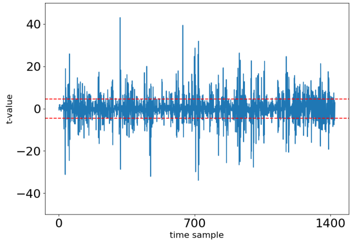

Figure: 1,000 traces, unmasked

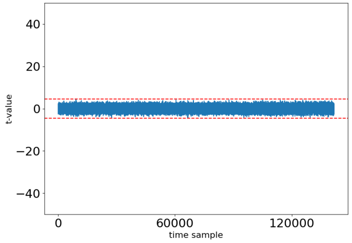

Figure: 10,000 traces, 2-shared

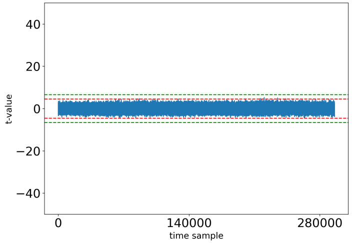

Figure: 100,000 traces, 3-shared

TVLA results of floating-point number addition (FprAdd, SecFprAdd)

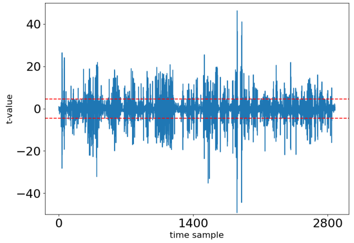

Figure: 1,000 traces, unmasked

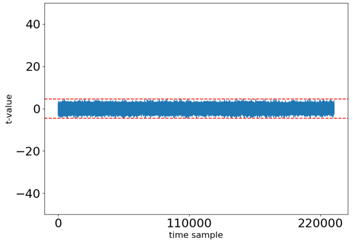

Figure: 10,000 traces, 2-shared

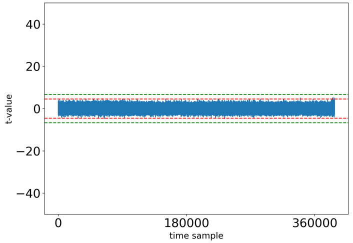

Figure: 100,000 traces, 3-shared

Table of Contents

Performance Evaluation on ARM Cortex-M4

Gadget	**Cycle**		
	Unmasked	**2 Shares**	**3 Shares**
FprMul/SecFprMul	308	7134 (23×)	36388 (118×)
FprAdd/SecFprAdd	487	17154 (35×)	48291 (99×)

Table: Performance evaluation of SecFprMul and SecFprAdd

Performance Evaluation on ARM Cortex-M4

Gadget		Cycle	
		2 Shares	3 Shares
SecFprMul	**Total**	**7134**	**36388**
	128-bit A2B [Sch+19]	1619 (23%)	19253 (53%)
	SecFPR	3362 (47%)	10813 (30%)
SecFprAdd	**Total**	**17154**	**48291**
	64-bit SecAdd [Bar+18]	6990 (41%)	16956 (35%)
	SecFPR	3362 (20%)	10813 (22%)

Table: Performance evaluation of each component in SecFprMul and SecFprAdd.

Table of Contents

Conclusion

In this paper,

Conclusion

In this paper,

- We present the first masking scheme for floating-point number multiplication and addition to protect the pre-image vector computation.

Conclusion

In this paper,

- We present the first masking scheme for floating-point number multiplication and addition to protect the pre-image vector computation.
- We design novel gadgets SecNonzero, SecFprUrsh, and SecFprNorm64.

Conclusion

In this paper,

- We present the first masking scheme for floating-point number multiplication and addition to protect the pre-image vector computation.
- We design novel gadgets SecNonzero, SecFprUrsh, and SecFprNorm64.
- All our gadgets are proven t-NI or t-SNI secure.

Conclusion

In this paper,

- We present the first masking scheme for floating-point number multiplication and addition to protect the pre-image vector computation.
- We design novel gadgets SecNonzero, SecFprUrsh, and SecFprNorm64.
- All our gadgets are proven t-NI or t-SNI secure.
- The TVLA result shows no leakage in the 2-shared version in 10,000 traces, and no leakage in the 3-shared version in 100,000 traces.

Conclusion

In this paper,

- We present the first masking scheme for floating-point number multiplication and addition to protect the pre-image vector computation.
- We design novel gadgets SecNonzero, SecFprUrsh, and SecFprNorm64.
- All our gadgets are proven *t*-NI or *t*-SNI secure.
- The TVLA result shows no leakage in the 2-shared version in 10,000 traces, and no leakage in the 3-shared version in 100,000 traces.
- Our countermeasure when compared to the unmasked reference implementation is slow. Improved designs of SecAdd and A2B can reduce the costs.

Thank You

Any question?

Reference I

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Springer, Heidelberg, Aug. 2003, pp. 463–481. DOI: 10.1007/978-3-540-45146-4_27.

[Bar+16] Gilles Barthe et al. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl et al. ACM Press, Oct. 2016, pp. 116–129. DOI: 10.1145/2976749.2978427.

[DP16] Léo Ducas and Thomas Prest. "Fast fourier orthogonalization". In: *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*. 2016, pp. 191–198.

[Din+17] A. Adam Ding et al. "Towards Sound and Optimal Leakage Detection Procedure". In: *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*. Ed. by Thomas Eisenbarth and Yannick Teglia. Vol. 10728. Lecture Notes in Computer Science. Springer, 2017, pp. 105–122. DOI: 10.1007/978-3-319-75208-2_7. URL: https://doi.org/10.1007/978-3-319-75208-2%5C_7.

[Bar+18] Gilles Barthe et al. "Masking the GLP Lattice-Based Signature Scheme at Any Order". In: *EUROCRYPT 2018, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. LNCS. Springer, Heidelberg, Apr. 2018, pp. 354–384. DOI: 10.1007/978-3-319-78375-8_12.

Reference II

[Sch+19] Tobias Schneider et al. "Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto". In: *PKC 2019, Part II*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11443. LNCS. Springer, Heidelberg, Apr. 2019, pp. 534–564. DOI: 10.1007/978-3-030-17259-6_18.

[Pre+20] Thomas Prest et al. *FALCON*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. National Institute of Standards and Technology, 2020.

[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)*. 2021, pp. 691–696. DOI: 10.1109/DAC18074.2021.9586131.

[Gue+22] Morgane Guerreau et al. "The Hidden Parallelepiped Is Back Again: Power Analysis Attacks on Falcon". In: *IACR TCHES* 2022.3 (2022), pp. 141–164. DOI: 10.46586/tches.v2022.i3.141-164.

[Zha+23] Shiduo Zhang et al. "Improved Power Analysis Attacks on Falcon". In: *EUROCRYPT 2023, Part IV*. Ed. by Carmit Hazay and Martijn Stam. Vol. 14007. LNCS. Springer, Heidelberg, Apr. 2023, pp. 565–595. DOI: 10.1007/978-3-031-30634-1_19.

Table of Contents

Floating-Point Number Packing and Rounding

FPR

Input: Sign bit s, exponent e, and 55-bit mantissa z
Output: FPN x packed by s, e, z

```
1: $e \leftarrow e + 1076$
2: $b \leftarrow [\![e < 0]\!]$
3: $z \leftarrow z \wedge (b - 1)$
4: $b \leftarrow [\![z \neq 0]\!]$
5: $e \leftarrow e \wedge (-b)$
6: $x \leftarrow ((s \ll 63) \vee (z \gg 2)) + e \ll 52$
7: $f \leftarrow \texttt{0XC8} \gg z^{[3:1]}$
8: $x \leftarrow x + f^{(1)}$ {increment if $z^{[3:1]}$ is 011,110 or 111}
9: **return** $x$
```

SecFPR: Secure FPR

SecFPR

Input: 1-bit Boolean shares $(s_i)_{1\le i\le n}$
Input: 16-bit arithmetic shares $(e_i)_{1\le i\le n}$
Input: 55-bit Boolean shares $(z_i)_{1\le i\le n}$
Output: Boolean shares $(x_i)_{1\le i\le n}$

```
1: $e_1 \leftarrow e_1 + 1076$
2: $(e_i) \leftarrow \mathrm{A2B}((e_i))$
3: $(b_i) \leftarrow (-e_i^{(16)})$
4: $(z_i) \leftarrow \mathrm{SecAnd}((z_i), (\neg b_1, b_2, \cdots, b_n))$
5: $(e_i) \leftarrow \mathrm{SecAnd}((e_i), (-z_i^{(55)}))$
6: $(e_i) \leftarrow \mathrm{SecAdd}((e_i), (z_i^{(55)}))$
7: $(e_i) \leftarrow \mathrm{Refresh}((e_i))$
8: $(s_i) \leftarrow \mathrm{Refresh}((s_i))$
9: $(x_i) \leftarrow ((s_i^{(1)} \ll 63) \vee (e_i^{[11:1]} \ll 52) \vee (z_i^{[54:3]})$
10: $(f_i) \leftarrow \mathrm{SecOr}(\mathrm{Refresh}(z_i^{(1)}), (z_i^{(3)}))$
11: $(f_i) \leftarrow \mathrm{SecAnd}((f_i), (z_i^{(2)}))$
12: $(x_i) \leftarrow \mathrm{SecAdd}((x_i), (f_i))$
13: **return** $(x_i)$
```

Floating-Point Number Multiplication

FprMul

```
Input: FPN $x = (sx, ex, mx)$
Input: FPN $y = (sy, ey, my)$
Output: FPN product of $x$ and $y$
 1: $s \leftarrow sx \oplus sy$
 2: $e \leftarrow ex + ey - 2100$
 3: $z \leftarrow mx \times my$
 4: $b \leftarrow [\![z^{[50:1]} \neq 0]\!]$
 5: $z \leftarrow z^{[106:51]} \vee b$
 6: $z' \leftarrow (z \gg 1) \vee z^{(1)}$
 7: $w \leftarrow z^{(106)}$
 8: $z \leftarrow z \oplus (z \oplus z') \wedge (-w)$
 9: $e \leftarrow e + w$
10: $bx \leftarrow [\![ex \neq 0]\!]$, $by \leftarrow [\![ey \neq 0]\!]$
11: $b \leftarrow bx \wedge by$
12: $z \leftarrow z \wedge (-b)$
13: return $\mathrm{FPR}(s, e, z)$
```

SecFprMul: Secure FprMul

SecFprMul

```
Input: Shares $(sx_i)_{1\le i\le n}, (ex_i)_{1\le i\le n}, (mx_i)_{1\le i\le n}$
Input: Shares $(sy_i)_{1\le i\le n}, (ey_i)_{1\le i\le n}, (my_i)_{1\le i\le n}$
Output: Boolean shares for the FPN product.
 1: $(s_i) \leftarrow (sx_i \oplus sy_i)$
 2: $(e_i) \leftarrow (ex_1 + ey_1 - 2100, ex_2 + ey_2, \cdots)$
 3: $(p_i) \leftarrow \text{SecMult}((mx_i), (my_i))$
 4: $(p_i) \leftarrow \text{A2B}((p_i))$
 5: $(b_i) \leftarrow \text{SecNonzero}((p_i^{[51:1]}))$
 6: $(z_i) \leftarrow (p_i^{[105:51]})$
 7: $(z'_i) \leftarrow (p_i^{[105:51]} \oplus p_i^{[106:52]})$
 8: $(w_i) \leftarrow (p_i^{(106)})$
 9: $(z'_i) \leftarrow \text{SecAnd}((z'_i), \text{Refresh}((-w_i)))$
10: $(z_i) \leftarrow (z'_i \oplus z_i)$
11: $(z_i) \leftarrow \text{SecOr}((z_i), (b_i))$
12: $(w_i) \leftarrow \text{B2A}_{\text{Bit}}((w_i))$
13: $(e_i) \leftarrow (e_i + w_i)$
14: $(bx_i) \leftarrow \text{SecNonzero}((ex_i))$
15: $(by_i) \leftarrow \text{SecNonzero}((ey_i))$
16: $(d_i) \leftarrow \text{SecAnd}((bx_i), (by_i))$
17: $(z_i) \leftarrow \text{SecAnd}((z_i), (-d_i^{(1)}))$
18: return $\text{SecFPR}((s_i), (e_i), (z_i))$
```

Floating-Point Number Addition

FprAdd

```
Input: FPNs $x$ and $y$
Output: FPN sum of $x$ and $y$
1: $d \leftarrow x^{[63:1]} - y^{[63:1]}$
2: $cs \leftarrow d^{(64)} \vee ((1 - (-d)^{(64)}) \wedge x^{(64)})$
3: $m \leftarrow (x \oplus y) \wedge (-cs)$
4: $x \leftarrow x \oplus m, y \leftarrow y \oplus m$
5: Extract $(sx, ex, mx)$ and $(sy, ey, my)$ from $x, y$, respectively.
6: $mx \leftarrow mx \ll 3, my \leftarrow my \ll 3$
7: $ex \leftarrow ex - 1078, ey \leftarrow ey - 1078$
8: $c \leftarrow ex - ey$
9: $b \leftarrow [\![c < 60]\!]$
10: $my \leftarrow my \wedge (-b)$
11: $my \leftarrow (my \gg c) \vee [\![my^{[c:1]} \neq 0]\!]$
12: $s \leftarrow sx \oplus sy$
13: $z \leftarrow mx + (-1)^s my$
14: Normalize $z, ex$ to make the 64th bit of $z$ set
15: $z \leftarrow (z \gg 9) \vee [\![z^{[9:1]} \neq 0]\!]$
16: $ex \leftarrow ex + 9$
17: return FPR$(sx, ex, z)$
```

SecFprAdd: Secure FprAdd

SecFprAdd

```
Input: Boolean shares $(x_i)_{1\le i\le n}$
Input: Boolean shares $(y_i)_{1\le i\le n}$
Output: Boolean shares for the FPN sum
 1: $(xm_i) \leftarrow (x_i^{[63:1]})$
 2: $(ym_i) \leftarrow (\neg y_1^{[63:1]}, y_2^{[63:1]}, \cdots, y_n^{[63:1]})$
 3: $(d_i) \leftarrow \text{SecAdd}((xm_i), (ym_i))$
 4: $(b_i) \leftarrow \text{SecNonzero}(\neg d_1, d_2, \cdots, d_n)$
 5: $(b'_i) \leftarrow \text{SecNonzero}(\neg(d_1 \oplus (1 \ll 63)), d_2, \cdots, d_n)$
 6: $(cs_i) \leftarrow \text{SecAnd}((\neg b_1, b_2, \cdots, b_n), (x_i^{(64)}))$
 7: $(cs_i) \leftarrow \text{SecOr}((cs_i), (d_i^{(64)} \oplus b_i \oplus b'_i))$
 8: $(m_i) \leftarrow \text{SecAnd}((x_i \oplus y_i), (-cs_i))$
 9: $(x_i) \leftarrow (x_i \oplus m_i)$, $(y_i) \leftarrow (y_i \oplus m_i)$
10: Extract $(sx_i), (ex_i), (mx_i)$ and $(sy_i), (ey_i), (my_i)$ from
    $(x_i)$ and $(y_i)$, respectively.
11: $(mx_i) \leftarrow (mx_i \ll 3)$, $(my_i) \leftarrow (my_i \ll 3)$
12: $(ex_i) \leftarrow \text{B2A}((ex_i))$, $(ey_i) \leftarrow \text{B2A}((ey_i))$
13: $ex_1 \leftarrow ex_1 - 1078$, $ey_1 \leftarrow ey_1 - 1078$.
14: $(c_i) \leftarrow (ex_i - ey_i)$
15: $(c'_i) \leftarrow \text{A2B}((c_1 - 60, c_2, \cdots, c_n))$
16: $(my_i) \leftarrow \text{SecAnd}((my_i), (-(c_i'^{(16)})))$
17: $(my_i) \leftarrow \text{SecFprUrsh}((my_i), (c_i^{[6:1]}))$
18: $(my'_i) \leftarrow (\neg my_1, my_2, \cdots, my_n)$
19: $(my'_i) \leftarrow \text{SecAdd}((my'_i), (1, 0, \cdots, 0))$
20: $(s_i) \leftarrow (-(sx_i \oplus sy_i))$
21: $(my_i) \leftarrow \text{Refresh}((my_i))$
22: $(my'_i) \leftarrow \text{SecAnd}((my_i \oplus my'_i), (s_i))$
23: $(my_i) \leftarrow (my_i \oplus my'_i)$
24: $(z_i) \leftarrow \text{SecAdd}((mx_i), (my_i))$
25: $(z_i), (ex_i) \leftarrow \text{SecFprNorm64}((z_i), (ex_i))$
26: $(b_i) \leftarrow \text{SecNonzero}((z_i^{[10:1]}))$
27: $(z_i) \leftarrow (z_i \gg 9)$
28: $(z_i^{(1)}) \leftarrow (b_i)$
29: $ex_1 \leftarrow ex_1 + 9$
30: return $\text{SecFPR}(\text{Refresh}((sx_i)), (ex_i), (z_i))$
```