

Masking Floating-Point Number Multiplication and Addition of Falcon

Keng-Yu Chen, Jiun-Peng Chen

Conference on Cryptographic Hardware and Embedded Systems

September 5th, 2024

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
- 5 Conclusion

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
- 5 Conclusion

Introduction

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

Introduction

- To defend the potential threat from large-scale quantum computers, the US National Institute of Standards and Technology (NIST) initiated standardization process for post-quantum cryptography in 2016.
- In 2022, four selected algorithms – CRYSTALS-Kyber, CRYSTALS-Dilithium, FALCON, and SPHINCS+ were expected to be part of NIST's post-quantum cryptographic standards.

Introduction

- To defend the potential threat from large-scale quantum computers, the US National Institute of Standards and Technology (NIST) initiated standardization process for post-quantum cryptography in 2016.
- In 2022, four selected algorithms – CRYSTALS-Kyber, CRYSTALS-Dilithium, FALCON, and SPHINCS+ 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 the potential threat from large-scale quantum computers, the US National Institute of Standards and Technology (NIST) initiated standardization process for post-quantum cryptography in 2016.
- In 2022, four selected algorithms – CRYSTALS-Kyber, CRYSTALS-Dilithium, FALCON, and SPHINCS+ 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

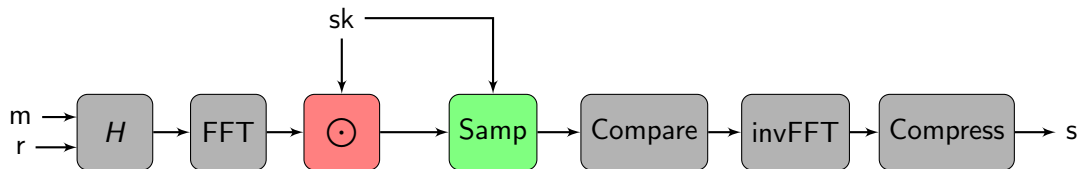


Figure: A graphical overview of FALCON.Sign.

| | Attack | Countermeasure |
|--------------------------------|------------------|------------------|
| Pre-image Vector Computation | [KA21; Gue+22] | None |
| Gaussian Sampler over Lattices | [Gue+22; Zha+23] | [Gue+22; Zha+23] |

Side-channel Attacks on FALCON

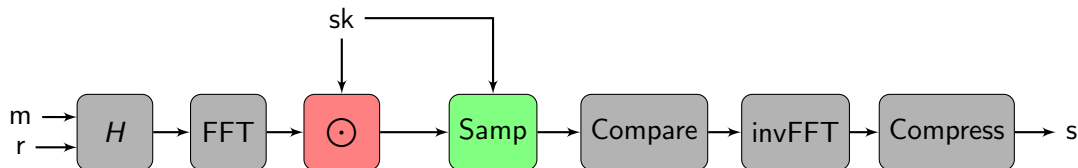


Figure: A graphical overview of FALCON.Sign.

| | Attack | Countermeasure |
|--------------------------------|------------------|------------------|
| Pre-image Vector Computation | [KA21; Gue+22] | None |
| Gaussian Sampler over Lattices | [Gue+22; Zha+23] | [Gue+22; Zha+23] |

Our Contributions

In this paper, we present the following contributions:

Our Contributions

In this paper, we present the following contributions:

- We propose the first masking scheme on the floating-point number multiplication and addition in the pre-image vector computation of `FALCON` as a countermeasure.

Our Contributions

In this paper, we present the following contributions:

- We propose the first masking scheme on the floating-point number multiplication and addition in the pre-image vector computation of `FALCON` as a countermeasure.
- We verify higher-order security of our design in the probing model.

Our Contributions

In this paper, we present the following contributions:

- We propose the first masking scheme on the floating-point number multiplication and addition in the pre-image vector computation of `FALCON` as a countermeasure.
- We verify higher-order security of our design in the probing model.
- To test the practical leakage of our work, we conduct the Test Vector Leakage Assessment (TVLA) [[GJR+11](#)] experiments.

Our Contributions

In this paper, we present the following contributions:

- We propose the first masking scheme on the floating-point number multiplication and addition in the pre-image vector computation of `FALCON` as a countermeasure.
- We verify higher-order security of our design in the probing model.
- To test the practical leakage of our work, we conduct the Test Vector Leakage Assessment (TVLA) [[GJR+11](#)] experiments.
- We also test the performance by comparing with the reference implementation of `FALCON` [[Pre+20](#)].

Notation

Throughout the presentation, we assume

Notation

Throughout the presentation, we assume

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

Notation

Throughout the presentation, we assume

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

Notation

Throughout the presentation, we assume

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

Notation

Throughout the presentation, we assume

- For a variable x , the j th bit of x is written as $x^{(j)}$.
- The i th bit to j th bit ($j \geq i$) of x is represented by $x^{[j:i]}$.
- A sequence of n variables (x_1, x_2, \dots, 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 , $\llbracket P \rrbracket = 1$ if and only if P is true and 0 if otherwise.

Table of Contents

- 1 Introduction
- 2 Preliminaries
 - FALCON
 - Floating-Point Number Arithmetic
 - Power Analysis and Masking
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
- 5 Conclusion

Table of Contents

- 1 Introduction
- 2 Preliminaries
 - FALCON
 - Floating-Point Number Arithmetic
 - Power Analysis and Masking
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
- 5 Conclusion

Introduction to FALCON

- A NIST-standardized digital signature
- Use the Gentry-Peikert-Vaikuntanathan (GPV) framework [GPV08] with NTRU lattices

KeyGen

Public Key: $\mathbf{A} \in \mathbb{Z}_q^{N \times M}$

Secret Key: Short $\mathbf{B} \in \mathbb{Z}_q^{M \times M}$

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

Sign(m)

A short \mathbf{s} s.t.

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

$$H : \{0, 1\}^* \rightarrow \{0, 1\}^N$$

Verify(m, s)

Check

① \mathbf{s} is short

② $\mathbf{sA}^T = H(m) \bmod q$

Introduction to FALCON

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

Introduction to FALCON

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

- Compute $H(\mathbf{m})$

Introduction to FALCON

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

- Compute $H(\mathbf{m})$
- Find a solution \mathbf{c} (not short) where $\mathbf{c}\mathbf{A}^T = H(\mathbf{m}) \bmod q$

Introduction to FALCON

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

- Compute $H(\mathbf{m})$
- Find a solution \mathbf{c} (not short) where $\mathbf{c}\mathbf{A}^T = H(\mathbf{m}) \bmod q$
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$

Introduction to FALCON

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

- Compute $H(\mathbf{m})$
- Find a solution \mathbf{c} (not short) where $\mathbf{c}\mathbf{A}^T = H(\mathbf{m}) \bmod q$
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$
- Apply the nearest plane algorithm to find an integer 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 first

- Compute $H(\mathbf{m})$
- Find a solution \mathbf{c} (not short) where $\mathbf{c}\mathbf{A}^T = H(\mathbf{m}) \bmod q$
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$
- Apply the nearest plane algorithm to find an integer vector \mathbf{z} such that $(\mathbf{t} - \mathbf{z})\mathbf{B}$ is short.
- $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\mathbf{B}$. Note that $\mathbf{s}\mathbf{A}^T = H(\mathbf{m}) \bmod q$

Introduction to FALCON

In FALCON,

Introduction to FALCON

In FALCON,

- Short secret polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^N + 1)$ where

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

Introduction to FALCON

In FALCON,

- Short secret polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^N + 1)$ where

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

- Public polynomial $h = gf^{-1} \bmod q$ and $\mathbf{A}^T = \left[\begin{array}{c} 1 \\ h \end{array} \right]$

Introduction to FALCON

In FALCON,

- Short secret polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^N + 1)$ where

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

- Public polynomial $h = gf^{-1} \bmod q$ and $\mathbf{A}^T = \left[\begin{array}{c} 1 \\ h \end{array} \right]$
- $\mathbf{c} = \left[\begin{array}{c|c} c & 0 \end{array} \right]$, where $c = H(r\|m)$ for the message m and a random salt r .

Introduction to FALCON

In FALCON,

- Short secret polynomials $f, g, F, G \in \mathbb{Z}[x]/(x^N + 1)$ where

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

- Public polynomial $h = gf^{-1} \bmod q$ and $\mathbf{A}^T = \left[\begin{array}{c} 1 \\ h \end{array} \right]$
- $\mathbf{c} = \left[\begin{array}{c|c} c & 0 \end{array} \right]$, where $c = H(r\|m)$ for the message m and a random salt r .

Moreover, FALCON applies the fast Fourier nearest plane algorithm [DP16] to speed up the signing process.

Introduction to FALCON

Sign (Simplified)

Input: Message m , secret key sk , bound $\lfloor \beta^2 \rfloor$

Output: Signature sig

- 1: Sample salt $r \leftarrow \{0, 1\}^{320}$ uniformly
- 2: $c \leftarrow H(r \| m)$
- 3: Compute the pre-image vector $\mathbf{t} \leftarrow \begin{bmatrix} c & 0 \end{bmatrix} \cdot \mathbf{B}^{-1}$
- 4: **repeat**
- 5: $\mathbf{z} = \text{ffSampling}(\mathbf{t}, sk)$
- 6: $\mathbf{s} = \begin{bmatrix} s_1 & s_2 \end{bmatrix} = (\mathbf{t} - \mathbf{z})\mathbf{B}$
- 7: **until** $\|\mathbf{s}\|^2 \leq \lfloor \beta^2 \rfloor$
- 8: $sig \leftarrow (r, s_2)$

Verify (Simplified)

Input: Message m , signature sig

Input: Bound $\lfloor \beta^2 \rfloor$

Output: Accept or Reject

- 1: $c \leftarrow H(r \| m)$
- 2: $s_1 \leftarrow c - s_2 h \bmod q$
- 3: **if** $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ **then**
- 4: Accept
- 5: **else**
- 6: Reject

Introduction to FALCON

Sign (Simplified)

Input: Message m , secret key sk , bound $\lfloor \beta^2 \rfloor$

Output: Signature sig

- 1: Sample salt $r \leftarrow \{0, 1\}^{320}$ uniformly
- 2: $c \leftarrow H(r \| m)$
- 3: **Compute the pre-image vector** $\mathbf{t} \leftarrow [c \mid 0] \cdot \mathbf{B}^{-1}$
- 4: **repeat**
- 5: $\mathbf{z} = \text{ffSampling}(\mathbf{t}, sk)$
- 6: $\mathbf{s} = [s_1 \mid s_2] = (\mathbf{t} - \mathbf{z})\mathbf{B}$
- 7: **until** $\|\mathbf{s}\|^2 \leq \lfloor \beta^2 \rfloor$
- 8: $sig \leftarrow (r, s_2)$

Verify (Simplified)

Input: Message m , signature sig

Input: Bound $\lfloor \beta^2 \rfloor$

Output: Accept or Reject

- 1: $c \leftarrow H(r \| m)$
- 2: $s_1 \leftarrow c - s_2 h \bmod q$
- 3: **if** $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ **then**
- 4: Accept
- 5: **else**
- 6: Reject

Table of Contents

- 1 Introduction
- 2 Preliminaries
 - FALCON
 - Floating-Point Number Arithmetic
 - Power Analysis and Masking
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
- 5 Conclusion

Fast-Fourier Transform

The pre-image vector computation includes polynomial multiplications

$$\mathbf{t} = \left[\begin{array}{c|c} c & 0 \end{array} \right] \cdot \mathbf{B}^{-1} = \frac{1}{q} \left[\begin{array}{c|c} c \cdot -F & c \cdot f \end{array} \right]$$

Fast-Fourier Transform

The pre-image vector computation includes polynomial multiplications

$$\mathbf{t} = \begin{bmatrix} c & | & 0 \end{bmatrix} \cdot \mathbf{B}^{-1} = \frac{1}{q} \begin{bmatrix} c \cdot -F & | & c \cdot f \end{bmatrix}$$

To speed up and apply the fast Fourier nearest plane algorithm, the pre-image vector computation is performed in the Fourier domain:

$$\frac{1}{q} \begin{bmatrix} \text{FFT}(c) \odot \text{FFT}(-F) & | & \text{FFT}(c) \odot \text{FFT}(f) \end{bmatrix}$$

Fast-Fourier Transform

The pre-image vector computation includes polynomial multiplications

$$\mathbf{t} = \left[c \mid 0 \right] \cdot \mathbf{B}^{-1} = \frac{1}{q} \left[c \cdot -F \mid c \cdot f \right]$$

To speed up and apply the fast Fourier nearest plane algorithm, the pre-image vector computation is performed in the Fourier domain:

$$\frac{1}{q} \left[\text{FFT}(c) \odot \text{FFT}(-F) \mid \text{FFT}(c) \odot \text{FFT}(f) \right]$$

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



Figure: A 64-bit Floating-Point Number

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

For convenience, we may use (s, e, m) to represent an FPN.

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 FPN addition (FprAdd) is proceeded by

- 1 Sign bit XOR

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 mantissa to $[2^{54}, 2^{55})$

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 mantissa to $[2^{54}, 2^{55})$
- 5 Combining the results 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 mantissa to $[2^{54}, 2^{55})$
- 5 Combining the results 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 mantissa to $[2^{54}, 2^{55})$
- 5 Combining the results 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 mantissa to $[2^{54}, 2^{55})$
- 5 Combining the results 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 mantissa to $[2^{54}, 2^{55})$
- ⑤ Combining the results 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 sum to $[2^{54}, 2^{55})$

Floating-Point Number Arithmetic

FPN multiplication (FprMul) is proceeded by

- 1 Sign bit XOR
- 2 Exponent Addition
- 3 Mantissa Multiplication
- 4 Right-shifting the mantissa to $[2^{54}, 2^{55})$
- 5 Combining the results 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 sum to $[2^{54}, 2^{55})$
- 5 Combining the results and rounding (FPR)

Sticky Bit

In floating-point arithmetic, when shifted right, the mantissa maintains a sticky bit

$$1001\textcolor{green}{0}\textcolor{red}{0100} \ggg 4 \rightarrow 1001 \underbrace{1}_{\text{Sticky}}$$

It indicates whether there exists any 1 after the least significant bit. In the above example,

$$\text{sticky bit} = \textcolor{green}{0} \vee \llbracket (\textcolor{red}{0100}) \neq 0 \rrbracket = \llbracket (\textcolor{green}{0}\textcolor{red}{0100}) \neq 0 \rrbracket$$

Table of Contents

- 1 Introduction
- 2 Preliminaries
 - FALCON
 - Floating-Point Number Arithmetic
 - Power Analysis and Masking
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
- 5 Conclusion

Power Analysis and Electromagnetic Attacks

- Power consumption and electromagnetic radiation during the execution of programs depend on intermediate values.

Power Analysis and Electromagnetic Attacks

- Power consumption and electromagnetic radiation during the execution of programs depend on intermediate values.
- Side-channel attacks on the pre-image vector computation [[KA21](#); [Gue+22](#)], leverage this fact to find the secret key.

Power Analysis and Electromagnetic Attacks

- Power consumption and electromagnetic radiation during the execution of programs depend on intermediate values.
- Side-channel attacks on the pre-image vector computation [[KA21](#); [Gue+22](#)], leverage this fact to find the secret key.

Masking

Masking resists such threats by secret-sharing the sensitive variables.

Masking

Masking resists such threats by secret-sharing the sensitive variables.

- Boolean Masking: A variable x is split into n shares (x_i) such that

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

Masking

Masking resists such threats by secret-sharing the sensitive variables.

- Boolean Masking: A variable x is split into n shares (x_i) such that

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

- Arithmetic Masking: A variable x is split into n shares (x_i) (when stored in a k -bit register) such that

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

In each run, all x_i 's are randomized so that any $n - 1$ shares of them are independently and uniformly distributed.

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
 - Overview of Our Approach
 - SecNonzero
 - SecFprUrsh
 - SecFprNorm64
- 4 Evaluation and Implementation
- 5 Conclusion

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
 - Overview of Our Approach
 - SecNonzero
 - SecFprUrsh
 - SecFprNorm64
- 4 Evaluation and Implementation
- 5 Conclusion

Overview of Our Approach

We now show how we mask FPR, FprMul, and FprAdd.

Overview of Our Approach

We now show how we mask FPR, FprMul, and FprAdd.
An intuitive approach to mask any algorithm:

Overview of Our Approach

We now show how we mask FPR, FprMul, and FprAdd.

An intuitive approach to mask any algorithm:

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

Overview of Our Approach

We now show how we mask FPR, FprMul, and FprAdd.

An intuitive approach to mask any algorithm:

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

Overview of Our Approach

We now show how we mask FPR, FprMul, and FprAdd.

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:

Overview of Our Approach

We now show how we mask FPR, FprMul, and FprAdd.

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

We now show how we mask FPR, FprMul, and FprAdd.

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) , 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) , 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) and (c_i) , right-shifting (x_i) by (c_i)

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) , 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) and (c_i) , right-shifting (x_i) by (c_i)
- Normalizing a secret value to $[2^{63}, 2^{64})$
 - Given (x_i) , left-shifting (x_i) 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 (Algorithm 6): securely checking whether a secret value is nonzero.

Overview of Our Approach

We design novel gadgets for these three operations, including:

- SecNonzero (Algorithm 6): securely checking whether a secret value is nonzero.
- SecFprUrsh (Algorithm 7): securely right-shifting a secret value by another secret value

Overview of Our Approach

We design novel gadgets for these three operations, including:

- SecNonzero (Algorithm 6): securely checking whether a secret value is nonzero.
- SecFprUrsh (Algorithm 7): securely right-shifting a secret value by another secret value
- SecFprNorm64 (Algorithm 8): securely normalizing a secret value to $[2^{63}, 2^{64})$

Overview of Our Approach

We design novel gadgets for these three operations, including:

- SecNonzero (Algorithm 6): securely checking whether a secret value is nonzero.
- SecFprUrsh (Algorithm 7): securely right-shifting a secret value by another secret value
- SecFprNorm64 (Algorithm 8): securely normalizing a secret value to $[2^{63}, 2^{64})$

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

Gadgets Used in Our Work

| Gadget | Description | Reference |
|--------------------|-------------------------------------|------------------|
| SecAnd | AND of Boolean shares | [ISW03; Bar+16] |
| SecMult | Multiplication of arithmetic shares | [ISW03; Bar+16] |
| SecAdd | Addition of Boolean shares | [Cor+15; Bar+18] |
| A2B | Arithmetic to Boolean conversion | [Sch+19] |
| B2A | Boolean to arithmetic conversion | [BCZ18] |
| B2A _{Bit} | One-bit B2A conversion | [Sch+19] |
| RefreshMasks | t -NI refresh of masks | [Bar+16; BCZ18] |
| Refresh | t -SNI refresh of masks | [Bar+16] |

Table: List of used gadgets in our work

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
 - Overview of Our Approach
 - SecNonzero
 - SecFprUrsh
 - SecFprNorm64
- 4 Evaluation and Implementation
- 5 Conclusion

SecNonzero

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

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

SecNonzero

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

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

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

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

SecNonzero

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

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

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

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

Now we turn to a gadget for secure OR operations.

SecOr: OR of Boolean Shares

SecOr

Input: Boolean shares $(x_i)_{1 \leq i \leq n}$ for value x

Input: Boolean shares $(y_i)_{1 \leq i \leq n}$ for value y

Output: Boolean shares $(z_i)_{1 \leq i \leq n}$ for value $z = x \vee y$

1: $(t_i)_{1 \leq i \leq n} \leftarrow (\neg x_1, x_2, \dots, x_n)$

2: $(s_i)_{1 \leq i \leq n} \leftarrow (\neg y_1, y_2, \dots, y_n)$

3: $(z_i) \leftarrow \text{SecAnd}((s_i), (t_i))$

4: $z_1 \leftarrow \neg z_1$

5: **return** (z_i)

It applies De Morgan's law and calls the AND algorithm SecAnd of shares as a subroutine.

$$x \vee y = \neg [(\neg x) \wedge (\neg y)]$$

SecNonzero

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

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

SecNonzero

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

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

So we apply two $n/2$ -shared A2Bs to the first $n/2$ shares and negative of the second $n/2$ shares and use the same idea.

SecNonzero

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

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

So we apply two $n/2$ -shared A2Bs to the first $n/2$ shares and negative of the second $n/2$ shares and use the same idea.

In this way, we replace one n -shared A2B with two $n/2$ -shared A2Bs, which is usually more efficient.

SecNonzero

SecNonzero

Input: Shares $(x_i)_{1 \leq i \leq n}$ for value x , bitsize

Output: One-bit Boolean shares $(b_i)_{1 \leq i \leq n}$ where $\bigoplus_i b_i = 0 \Leftrightarrow x = 0$

```

1: if input  $(x_i)$  are arithmetic shares then
2:    $(t_i)_{1 \leq i \leq \frac{n}{2}} \leftarrow \text{A2B}((x_i)_{1 \leq i \leq \frac{n}{2}})$ 
3:    $(t_i)_{\frac{n}{2}+1 \leq i \leq n} \leftarrow \text{A2B}((-x_i)_{\frac{n}{2}+1 \leq i \leq n})$ 
4: else
5:    $(t_i)_{1 \leq i \leq n} \leftarrow (x_i)_{1 \leq i \leq n}$ 
6:  $\text{len} \leftarrow \text{bitsize}/2$ 
7: while  $\text{len} \geq 1$  do
8:    $(l_i) \leftarrow \text{Refresh}((t_i^{[2\text{len}:\text{len}]}) , \text{len})$ 
9:    $(r_i) \leftarrow (t_i^{[\text{len}:1]})$ 
10:   $(t_i) \leftarrow \text{SecOr}((l_i), (r_i))$ 
11:   $\text{len} \leftarrow \text{len} \gg 1$ 
12: return  $(t_i^{(1)})$ 

```

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
 - Overview of Our Approach
 - SecNonzero
 - SecFprUrsh
 - SecFprNorm64
- 4 Evaluation and Implementation
- 5 Conclusion

SecFprUrsh

Given 64-bit shares $(x_i)_{i=1}^n$ and 6-bit $(c_i)_{i=1}^n$, we need to derive shares $(z_i)_{i=1}^n$ such that

$$\bigoplus_{i=1}^n z_i = \left(\left(\bigoplus_{i=1}^n x_i \right) \ggg \left(\sum_{i=1}^n c_i \bmod 2^6 \right) \right) \vee \left[\bigoplus_{i=1}^n x_i^{[c:1]} \neq 0 \right]$$

SecFprUrsh

Given 64-bit shares $(x_i)_{i=1}^n$ and 6-bit $(c_i)_{i=1}^n$, we need to derive shares $(z_i)_{i=1}^n$ such that

$$\bigoplus_{i=1}^n z_i = \left(\left(\bigoplus_{i=1}^n x_i \right) \ggg \left(\sum_{i=1}^n c_i \bmod 2^6 \right) \right) \vee \left[\bigoplus_{i=1}^n x_i^{[c:1]} \neq 0 \right]$$

We observe that

SecFprUrsh

Given 64-bit shares $(x_i)_{i=1}^n$ and 6-bit $(c_i)_{i=1}^n$, we need to derive shares $(z_i)_{i=1}^n$ such that

$$\bigoplus_{i=1}^n z_i = \left(\left(\bigoplus_{i=1}^n x_i \right) \ggg \left(\sum_{i=1}^n c_i \bmod 2^6 \right) \right) \vee \left[\bigoplus_{i=1}^n x_i^{[c:1]} \neq 0 \right]$$

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 $(x_i)_{i=1}^n$ and 6-bit $(c_i)_{i=1}^n$, we need to derive shares $(z_i)_{i=1}^n$ such that

$$\bigoplus_{i=1}^n z_i = \left(\left(\bigoplus_{i=1}^n x_i \right) \ggg \left(\sum_{i=1}^n c_i \bmod 2^6 \right) \right) \vee \left[\bigoplus_{i=1}^n x_i^{[c:1]} \neq 0 \right]$$

We observe that

- Right-shifting and right-rotating by a value c only differ by the most c significant bits.
- Both shifting and rotating can be operated share-wise.

SecFprUrsh

Given 64-bit shares $(x_i)_{i=1}^n$ and 6-bit $(c_i)_{i=1}^n$, we need to derive shares $(z_i)_{i=1}^n$ such that

$$\bigoplus_{i=1}^n z_i = \left(\left(\bigoplus_{i=1}^n x_i \right) \ggg \left(\sum_{i=1}^n c_i \bmod 2^6 \right) \right) \vee \left[\bigoplus_{i=1}^n x_i^{[c:1]} \neq 0 \right]$$

We observe that

- Right-shifting and right-rotating by a value c only differ by the most c significant bits.
- Both shifting and rotating can be operated share-wise.
- Right-rotating x by a value c is equal to right-rotating x by a value $c \bmod 64$.

SecFprUrsh

Hence, our idea is to right-rotate all x'_i 's by c_1, c_2, \dots, c_n sequentially.

SecFprUrsh

Hence, our idea is to right-rotate all x'_i s by c_1, c_2, \dots, c_n sequentially.

Some high bits are redundant, so we use an index $m = (1 \lll 63)$ 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, \dots, 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. To clear the redundant high bits, consider

$$m' := m \ggg c = \underbrace{(0, \dots, 0, 1, 0, \dots, 0)}_{c \text{ bits}}$$

SecFprUrsh

Hence, our idea is to right-rotate all x'_i s by c_1, c_2, \dots, 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. To clear the redundant high bits, consider

$$m' := m \ggg c = (\underbrace{0, \dots, 0}_{c \text{ bits}}, 1, 0, \dots, 0)$$

$$m'' := m' \oplus (m' \ggg 1) \oplus \dots \oplus (m' \ggg 63) = (\underbrace{0, \dots, 0}_{c \text{ bits}}, 1, 1, \dots, 1)$$

SecFprUrsh

Hence, our idea is to right-rotate all x'_i s by c_1, c_2, \dots, 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. To clear the redundant high bits, consider

$$m' := m \ggg c = \underbrace{(0, \dots, 0, 1, 0, \dots, 0)}_{c \text{ bits}}$$

$$m'' := m' \oplus (m' \ggg 1) \oplus \dots \oplus (m' \ggg 63) = \underbrace{(0, \dots, 0, 1, 1, \dots, 1)}_{c \text{ bits}}$$

By an AND operation with m'' , we can clear useless bits. Moreover, these redundant bits actually form the sticky bit.

SecFprUrsh

SecFprUrsh

Input: 64-bit Boolean shares $(x_i)_{1 \leq i \leq n}$

Input: 6-bit arithmetic shares $(c_i)_{1 \leq i \leq n}$

Output: Boolean shares $(z_i)_{1 \leq i \leq n}$ for value
 $z = x \ggg c$ with the sticky bit preserved

1: $(m_i)_{1 \leq i \leq n} \leftarrow ((1 \lll 63), 0, \dots, 0)$

2: **for** $j = 1$ to n **do**

3: Right-rotate (x_i) by c_j

4: $(x_i) \leftarrow \text{RefreshMasks}((x_i))$

5: Right-rotate (m_i) by c_j

6: $(m_i) \leftarrow \text{RefreshMasks}((m_i))$

7: $\text{len} \leftarrow 1$

8: **while** $\text{len} \leq 32$ **do**

9: $(m_i) \leftarrow (m_i \oplus (m_i \ggg \text{len}))$

10: $\text{len} \leftarrow \text{len} \lll 1$

11: $(y_i) \leftarrow \text{SecAnd}((x_i), (m_i))$

12: $(z_i) \leftarrow (y_i \oplus x_i \oplus y_i^{(1)})$

13: $(b_i) \leftarrow \text{SecNonzero}((z_i))$

14: $(z_i) \leftarrow (y_i^{[64:2]} \vee b_i)$

15: **return** (z_i)

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
 - Overview of Our Approach
 - SecNonzero
 - SecFprUrsh
 - SecFprNorm64
- 4 Evaluation and Implementation
- 5 Conclusion

SecFprNorm64

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

Let c be the smallest integer such that $((\oplus_{i=1}^n x_i) \ll c) \in [2^{63}, 2^{64})$

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

SecFprNorm64

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

Let c be the smallest integer such that $((\oplus_{i=1}^n x_i) \ll c) \in [2^{63}, 2^{64})$

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

We can repeatedly check whether $(x_i^{(64)})$ is 0, then conditionally shift by 1 bit, and then decrease (e_i) by $\llbracket (x_i^{(64)}) = 0 \rrbracket$.

SecFprNorm64

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

Let c be the smallest integer such that $((\oplus_{i=1}^n x_i) \ll c) \in [2^{63}, 2^{64})$

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

We can repeatedly check whether $(x_i^{(64)})$ is 0, then conditionally shift by 1 bit, and then decrease (e_i) by $\llbracket (x_i^{(64)}) = 0 \rrbracket$.

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

SecFprNorm64

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

Let c be the smallest integer such that $((\oplus_{i=1}^n x_i) \ll c) \in [2^{63}, 2^{64})$

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

We can repeatedly check whether $(x_i^{(64)})$ is 0, then conditionally shift by 1 bit, and then decrease (e_i) by $\llbracket (x_i^{(64)}) = 0 \rrbracket$.

To improve efficiency, we consider sequentially checking $x^{[64:64-2^j]} = 0$ for $j = 5, 4, \dots, 0$. In addition, we first decrease (e_i) by 63 and later add $\llbracket (x_i^{[64:64-2^j]}) \neq 0 \rrbracket \cdot 2^j$ to it.

SecFprNorm64

SecFprNorm64

Input: 64-bit Boolean shares $(x_i)_{1 \leq i \leq n}$

Input: 16-bit arithmetic shares $(e_i)_{1 \leq i \leq n}$

Output: Normalized $(x_i)_{1 \leq i \leq n}$ in $[2^{63}, 2^{64})$ and $(e_i)_{1 \leq i \leq n}$ with shift added

- 1: $e_1 \leftarrow e_1 - 63$
- 2: **for** $j = 5$ to 0 **do**
- 3: $(t_i) \leftarrow (x_i \oplus (x_i \ll 2^j))$
- 4: $(n_i) \leftarrow (x_i \gg (64 - 2^j))$
- 5: $(b_i) \leftarrow \text{SecNonzero}((n_i))$
- 6: $(b'_i) \leftarrow (-b_i)$
- 7: $(t_i) \leftarrow \text{SecAnd}((t_i), (\neg b'_1, b'_2, \dots, b'_n))$
- 8: $(x_i) \leftarrow (x_i \oplus t_i)$
- 9: $(b_i) \leftarrow \text{B2ABit}((b_i))$
- 10: $(e_i) \leftarrow (e_i + (b_i \ll j))$
- 11: **return** $(x_i), (e_i)$

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 by masking.

Wrapping-up

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

- SecFPR: Secure FPR by masking.
- SecFprMul: Secure FprMul by masking.

Wrapping-up

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

- SecFPR: Secure FPR by masking.
- SecFprMul: Secure FprMul by masking.
- SecFprAdd: Secure FprAdd by masking.

Wrapping-up

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

- SecFPR: Secure FPR by masking.
- SecFprMul: Secure FprMul by masking.
- SecFprAdd: Secure FprAdd by masking.

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

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation**
 - Security
 - Performance
- 5 Conclusion

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
 - Security
 - Performance
- 5 Conclusion

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, $n \geq t + 1$, 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, $n \geq t + 1$, and any share cannot be combined with each other.
- It is 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.

Non-Interference Security of Gadgets in Our Work

| Gadget | Security | Gadget | Security |
|--------------------|----------|--------------|----------|
| SecAnd | t -SNI | SecOr | t -SNI |
| SecMult | t -SNI | SecNonzero | t -SNI |
| SecAdd | t -NI | SecFprUrsh | t -SNI |
| A2B | t -SNI | SecFprNorm64 | t -NI |
| B2A | t -SNI | SecFPR | t -SNI |
| B2A _{Bit} | t -SNI | SecFprMul | t -SNI |
| RefreshMasks | t -NI | SecFprAdd | t -SNI |
| Refresh | t -SNI | | |

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

Practical Security – Test Vector Leakage Assessment (TVLA)

To test the leakage of a real device, the Test Vector Leakage Assessment (TVLA) methodology [[GJR+11](#)] can be applied.

Practical Security – Test Vector Leakage Assessment (TVLA)

To test the leakage of a real device, the Test Vector Leakage Assessment (TVLA) methodology [[GJR+11](#)] can be applied.

A tester records two sets of traces where

Practical Security – Test Vector Leakage Assessment (TVLA)

To test the leakage of a real device, the Test Vector Leakage Assessment (TVLA) methodology [GJR+11] can be applied.

A tester records two sets of traces where

- Set 1: fixed input

Practical Security – Test Vector Leakage Assessment (TVLA)

To test the leakage of a real device, the Test Vector Leakage Assessment (TVLA) methodology [GJR+11] can be applied.

A tester records two sets of traces where

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

Practical Security – Test Vector Leakage Assessment (TVLA)

To test the leakage of a real device, the Test Vector Leakage Assessment (TVLA) methodology [GJR+11] can be applied.

A tester records two sets of traces where

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

The Welch's t -test is then applied on the two sets.

Practical Security – Test Vector Leakage Assessment (TVLA)

To test the leakage of a real device, the Test Vector Leakage Assessment (TVLA) methodology [GJR+11] can be applied.

A tester records two sets of traces where

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

The Welch's t -test is then applied on the two sets.

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

Practical Security – Test Vector Leakage Assessment (TVLA)

To test the leakage of a real device, the Test Vector Leakage Assessment (TVLA) methodology [GJR+11] can be applied.

A tester records two sets of traces where

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

The Welch's t -test is then applied on the two sets.

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

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

Experiment Setup

We implemented our design in the following setting:

Experiment Setup

We implemented our design in the following setting:

- Plain-C code

Experiment Setup

We implemented our design in the following setting:

- Plain-C code
- Compiled by `arm-none-eabi-gcc 10.3.1`

Experiment Setup

We implemented our design in the following setting:

- Plain-C code
- Compiled by `arm-none-eabi-gcc 10.3.1`
- Using ChipWhisperer with target board STM32F415 with an ARM Cortex-M4 MCU

Experiment Setup

We implemented our design in the following setting:

- Plain-C code
- Compiled by `arm-none-eabi-gcc 10.3.1`
- Using ChipWhisperer with target board STM32F415 with an ARM Cortex-M4 MCU
- We compare the result with the reference implementation of the NIST Round-3 Submission of FALCON [[Pre+20](#)].

TVLA

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

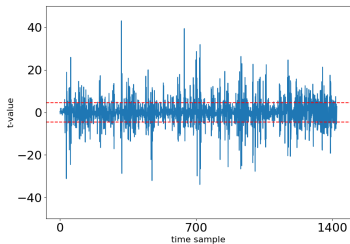


Figure: 1,000 traces, unmasked FprMul

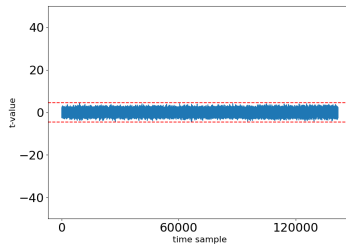


Figure: 10,000 traces, 2-shared SecFprMul

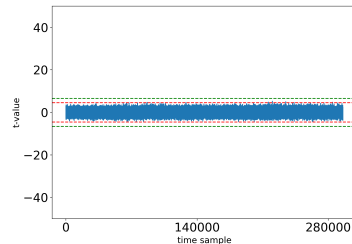


Figure: 100,000 traces, 3-shared SecFprMul

TVLA

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

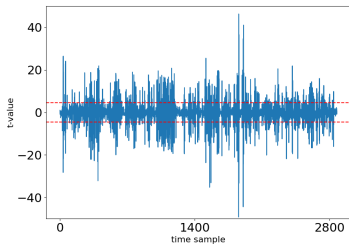


Figure: 1,000 traces, unmasked FprAdd

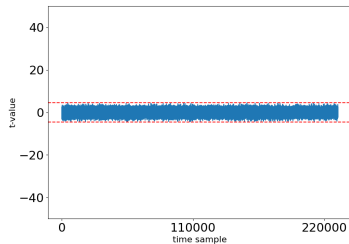


Figure: 10,000 traces, 2-shared SecFprAdd

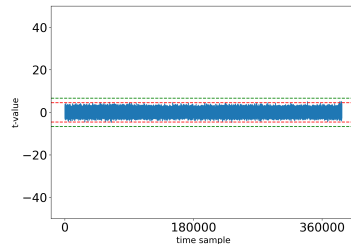


Figure: 100,000 traces, 3-shared SecFprAdd

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
 - Security
 - Performance
- 5 Conclusion

Performance Evaluation on ARM Cortex-M4

| Gadget | Cycle | | |
|-----------|----------|----------------------|-----------------------|
| | Unmasked | 2 Shares | 3 Shares |
| SecFprMul | 308 | 7134 (23 \times) | 36388 (118 \times) |
| SecFprAdd | 487 | 17154 (35 \times) | 48291 (99 \times) |

Table: Performance evaluation of SecFprMul and SecFprAdd

Performance Evaluation on ARM Cortex-M4

| Gadget | | Cycle | | |
|-----------|-----------------------|------------|--------------------|---------------------|
| | | Unmasked | 2 Shares | 3 Shares |
| SecFprMul | Total | 308 | 7134 (23×) | 36388 (118×) |
| | 128-bit A2B | - | 1619 | 19253 |
| | 64-bit SecNonzero | - | 389 | 1350 |
| | Two 16-bit SecNonzero | - | 662 | 2012 |
| | SecFPR | - | 3362 | 10813 |
| | #randombytes | - | 333 | 2005 |
| SecFprAdd | Total | 487 | 17154 (35×) | 48291 (99×) |
| | Three 64-bit SecAdd | - | 6990 | 16956 |
| | Two 16-bit B2A | - | 88 | 332 |
| | 16-bit A2B | - | 146 | 2267 |
| | SecFprUrsh | - | 1112 | 3214 |
| | SecFprNorm64 | - | 2846 | 7270 |
| | SecFPR | - | 3362 | 10813 |
| | #randombytes | - | 849 | 2691 |

Performance Evaluation on ARM Cortex-M4

| Gadget | | Cycle | | |
|-----------|-----------------------|------------|--------------------|---------------------|
| | | Unmasked | 2 Shares | 3 Shares |
| SecFprMul | Total | 308 | 7134 (23×) | 36388 (118×) |
| | 128-bit A2B | - | 1619 | 19253 |
| | 64-bit SecNonzero | - | 389 | 1350 |
| | Two 16-bit SecNonzero | - | 662 | 2012 |
| | SecFPR | - | 3362 | 10813 |
| | #randombytes | - | 333 | 2005 |
| SecFprAdd | Total | 487 | 17154 (35×) | 48291 (99×) |
| | Three 64-bit SecAdd | - | 6990 | 16956 |
| | Two 16-bit B2A | - | 88 | 332 |
| | 16-bit A2B | - | 146 | 2267 |
| | SecFprUrsh | - | 1112 | 3214 |
| | SecFprNorm64 | - | 2846 | 7270 |
| | SecFPR | - | 3362 | 10813 |
| | #randombytes | - | 849 | 2691 |

Performance Evaluation on General Purpose CPU

We also test the time for signing one message on an Intel-Core i9-12900KF.

| Security Level | Unmasked | 2 Shares | 3 Shares |
|----------------|----------|-------------------------|---------------------------|
| Falcon-512 | 246.56 | 1905.55 ($7.7\times$) | 6137.25 ($24.9\times$) |
| Falcon-1024 | 501.62 | 3819.76 ($7.6\times$) | 12287.29 ($24.5\times$) |

Table: Time (in microseconds) for signing a message on Intel-Core i9-12900KF CPU.

Table of Contents

- 1 Introduction
- 2 Preliminaries
- 3 Masked Floating-Point Number Multiplication and Addition
- 4 Evaluation and Implementation
- 5 Conclusion

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 much slower. Improved designs of SecAdd and A2B can reduce the costs.

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](https://doi.org/10.1007/978-3-540-45146-4_27).
- [GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. “Trapdoors for hard lattices and new cryptographic constructions”. In: *40th ACM STOC*. Ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, May 2008, pp. 197–206. DOI: [10.1145/1374376.1374407](https://doi.org/10.1145/1374376.1374407).
- [Ber+10] Guido Bertoni et al. “Building power analysis resistant implementations of Keccak”. In: *Second SHA-3 candidate conference*. Vol. 142. Citeseer. 2010.
- [GJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. “A testing methodology for side-channel resistance validation”. In: *NIST non-invasive attack testing workshop*. Vol. 7. 2011, pp. 115–136.
- [Bel+13] Sonia Belaïd et al. “Differential power analysis of HMAC SHA-2 in the Hamming weight model”. In: *2013 International Conference on Security and Cryptography (SECRYPT)*. 2013, pp. 1–12.
- [Cor+15] Jean-Sébastien Coron et al. “Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity”. In: *FSE 2015*. Ed. by Gregor Leander. Vol. 9054. LNCS. Springer, Heidelberg, Mar. 2015, pp. 130–149. DOI: [10.1007/978-3-662-48116-5_7](https://doi.org/10.1007/978-3-662-48116-5_7).
- [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](https://doi.org/10.1145/2976749.2978427).

Reference II

- [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](https://doi.org/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](https://doi.org/10.1007/978-3-319-78375-8_12).
- [BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. “Improved High-Order Conversion From Boolean to Arithmetic Masking”. In: *IACR TCHES 2018.2* (2018). <https://tches.iacr.org/index.php/TCHES/article/view/873>, pp. 22–45. ISSN: 2569-2925. DOI: [10.13154/tches.v2018.i2.22-45](https://doi.org/10.13154/tches.v2018.i2.22-45).
- [Mig+19] Vincent Migliore et al. “Masking Dilithium - Efficient Implementation and Side-Channel Evaluation”. In: *ACNS 19*. Ed. by Robert H. Deng et al. Vol. 11464. LNCS. Springer, Heidelberg, June 2019, pp. 344–362. DOI: [10.1007/978-3-030-21568-2_17](https://doi.org/10.1007/978-3-030-21568-2_17).

Reference III

- [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](https://doi.org/10.1007/978-3-030-17259-6_18).
- [How+20] James Howe et al. “Isochronous Gaussian Sampling: From Inception to Implementation”. In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Springer, Heidelberg, 2020, pp. 53–71. DOI: [10.1007/978-3-030-44223-1_5](https://doi.org/10.1007/978-3-030-44223-1_5).
- [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.
- [Bos+21] Joppe W. Bos et al. “Masking Kyber: First- and Higher-Order Implementations”. In: *IACR TCHES 2021.4* (2021). <https://tches.iacr.org/index.php/TCHES/article/view/9064>, pp. 173–214. ISSN: 2569-2925. DOI: [10.46586/tches.v2021.i4.173-214](https://doi.org/10.46586/tches.v2021.i4.173-214).
- [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](https://doi.org/10.1109/DAC18074.2021.9586131).
- [Fri+22] Tim Fritzmann et al. “Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography”. In: *IACR TCHES 2022.1* (2022), pp. 414–460. DOI: [10.46586/tches.v2022.i1.414-460](https://doi.org/10.46586/tches.v2022.i1.414-460).

Reference IV

- [Gue+22] Morgane Guereau 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](https://doi.org/10.46586/tches.v2022.i3.141-164).
- [Hei+22] Daniel Heinz et al. *First-Order Masked Kyber on ARM Cortex-M4*. *Cryptology ePrint Archive*, Report 2022/058. <https://eprint.iacr.org/2022/058>. 2022.
- [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](https://doi.org/10.1007/978-3-031-30634-1_19).

Table of Contents

6 Appendix - Algorithms of Floating-Point Number Arithmetic

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

SecFPR: Secure FPR

SecFPR

Input: 1-bit Boolean shares $(s_i)_{1 \leq i \leq n}$

Input: 16-bit arithmetic shares $(e_i)_{1 \leq i \leq n}$

Input: 55-bit Boolean shares $(z_i)_{1 \leq i \leq n}$

Output: Boolean shares $(x_i)_{1 \leq i \leq n}$

1: $e_1 \leftarrow e_1 + 1076$

2: $(e_i) \leftarrow \text{A2B}((e_i))$

3: $(b_i) \leftarrow (-e_i^{(16)})$

4: $(z_i) \leftarrow \text{SecAnd}((z_i), (\neg b_1, b_2, \dots, b_n))$

5: $(e_i) \leftarrow \text{SecAnd}((e_i), (-z_i^{(55)}))$

6: $(e_i) \leftarrow \text{SecAdd}((e_i), (z_i^{(55)}))$

7: $(e_i) \leftarrow \text{Refresh}((e_i))$

8: $(s_i) \leftarrow \text{Refresh}((s_i))$

9: $(x_i) \leftarrow ((s_i^{(1)} \ll 63) \vee (e_i^{[11:1]} \ll 52) \vee (z_i^{[54:3]}))$

10: $(f_i) \leftarrow \text{SecOr}(\text{Refresh}(z_i^{(1)}), (z_i^{(3)}))$

11: $(f_i) \leftarrow \text{SecAnd}((f_i), (z_i^{(2)}))$

12: $(x_i) \leftarrow \text{SecAdd}((x_i), (f_i))$

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: $b \leftarrow \llbracket z^{[50:1]} \neq 0 \rrbracket$

5: $z \leftarrow z^{[106:51]} \vee b$

6: $z' \leftarrow (z \ggg 1) \vee z^{(1)}$

7: $w \leftarrow z^{(106)}$

8: $z \leftarrow z \oplus (z \oplus z') \wedge (-w)$

9: $e \leftarrow e + w$

10: $bx \leftarrow \llbracket ex \neq 0 \rrbracket, by \leftarrow \llbracket ey \neq 0 \rrbracket$

11: $b \leftarrow bx \wedge by$

12: $z \leftarrow z \wedge (-b)$

13: **return** FPR(s, e, z)

SecFprMul: Secure FprMul

SecFprMul

Input: Shares $(sx_i)_{1 \leq i \leq n}, (ex_i)_{1 \leq i \leq n}, (mx_i)_{1 \leq i \leq n}$

Input: Shares $(sy_i)_{1 \leq i \leq n}, (ey_i)_{1 \leq i \leq n}, (my_i)_{1 \leq i \leq n}$

Output: Boolean shares for the FPN product.

$$1: (s_i) \leftarrow (sx_i \oplus sy_i)$$

$$2: (e_i) \leftarrow (ex_1 + ey_1 - 2^{100}, ex_2 + ey_2, \dots)$$

$$3: (p_i) \leftarrow \text{SecMult}((mx_i), (my_i))$$

$$4: (p_i) \leftarrow \text{A2B}((p_i))$$

$$5: (b_i) \leftarrow \text{SecNonzero}((p_i^{[51:1]}))$$

$$6: (z_i) \leftarrow (p_i^{[105:51]})$$

$$7: (z'_i) \leftarrow (p_i^{[105:51]} \oplus p_i^{[106:52]})$$

$$8: (w_i) \leftarrow (p_i^{(106)})$$

$$9: (z'_i) \leftarrow \text{SecAnd}((z'_i), \text{Refresh}((-w_i)))$$

$$10: (z_i) \leftarrow (z'_i \oplus z_i)$$

$$11: (z_i) \leftarrow \text{SecOr}((z_i), (b_i))$$

$$12: (w_i) \leftarrow \text{B2ABit}((w_i))$$

$$13: (e_i) \leftarrow (e_i + w_i)$$

$$14: (bx_i) \leftarrow \text{SecNonzero}((ex_i))$$

$$15: (by_i) \leftarrow \text{SecNonzero}((ey_i))$$

$$16: (d_i) \leftarrow \text{SecAnd}((bx_i), (by_i))$$

$$17: (z_i) \leftarrow \text{SecAnd}((z_i), (-d_i^{(1)}))$$

$$18: \textbf{return SecFPR}((s_i), (e_i), (z_i))$$

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: $b \leftarrow \llbracket c < 60 \rrbracket$
- 10: $my \leftarrow my \wedge (-b)$
- 11: $my \leftarrow (my \ggg c) \vee \llbracket my^{[c:1]} \neq 0 \rrbracket$
- 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: $z \leftarrow (z \ggg 9) \vee \llbracket z^{[9:1]} \neq 0 \rrbracket$
- 16: $ex \leftarrow ex + 9$
- 17: **return** FPR(sx, ex, z)

SecFprAdd: Secure FprAdd

SecFprAdd

Input: Boolean shares $(x_i)_{1 \leq i \leq n}$

Input: Boolean shares $(y_i)_{1 \leq i \leq n}$

Output: Boolean shares for the FPN sum

- 1: $(xm_i) \leftarrow (x_i^{[63:1]})$
- 2: $(ym_i) \leftarrow (\neg y_1^{[63:1]}, y_2^{[63:1]}, \dots, y_n^{[63:1]})$
- 3: $(d_i) \leftarrow \text{SecAdd}((xm_i), (ym_i))$
- 4: $(b_i) \leftarrow \text{SecNonzero}(\neg d_1, d_2, \dots, d_n)$
- 5: $(b'_i) \leftarrow \text{SecNonzero}(\neg(d_1 \oplus (1 \ll 63)), d_2, \dots, d_n)$
- 6: $(cs_i) \leftarrow \text{SecAnd}((\neg b_1, b_2, \dots, b_n), (x_i^{(64)}))$
- 7: $(cs_i) \leftarrow \text{SecOr}((cs_i), (d_i^{(64)} \oplus b_i \oplus b'_i))$
- 8: $(m_i) \leftarrow \text{SecAnd}((x_i \oplus y_i), (\neg cs_i))$
- 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: $(c'_i) \leftarrow \text{A2B}((c_1 - 60, c_2, \dots, c_n))$
- 16: $(my_i) \leftarrow \text{SecAnd}((my_i), (\neg(c'_i)^{(16)})))$
- 17: $(my_i) \leftarrow \text{SecFprUrsh}((my_i), (c_i^{[6:1]}))$
- 18: $(my'_i) \leftarrow (\neg my_1, my_2, \dots, my_n)$
- 19: $(my'_i) \leftarrow \text{SecAdd}((my'_i), (1, 0, \dots, 0))$
- 20: $(s_i) \leftarrow (\neg(sx_i \oplus sy_i))$
- 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** $\text{SecFPR}(\text{Refresh}((sx_i)), (ex_i), (z_i))$