Masking Floating-Point Number Multiplication and Addition of Falcon

Keng-Yu Chen, Jiun-Peng Chen

October 16th, 2024

IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 2, pp. 276–303. DOI:10.46586/tches.v2024.i2.276-303

Masking Floating-Point Number Multiplication and Addition of Falcon

First- and Higher-order Implementations and Evaluations

Keng-Yu Chen[1] and Jiun-Peng Chen[1,2]

[1] National Taiwan University, Taipei, Taiwan, r11921066@ntu.edu.tw
[2] Academia Sinica, Taipei, Taiwan, jpchen@ieee.org

[CC24]: Accepcted by Cryptographic Hardware and Embedded Systems (CHES) 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

Theoretical and Real-World Security

- In 2022, the US National Institute of Standards and Technology (NIST) selected four algorithms for its post-quantum cryptography standardization process.

Theoretical and Real-World Security

- In 2022, the US National Institute of Standards and Technology (NIST) selected four algorithms for its post-quantum cryptography standardization process.
- In theory, these algorithms can base their security on problems that are considered still hard given large-scale quantum computing.

Theoretical and Real-World Security

- In 2022, the US National Institute of Standards and Technology (NIST) selected four algorithms for its post-quantum cryptography standardization process.
- In theory, these algorithms can base their security on problems that are considered still hard given large-scale quantum computing.
- In practice, the implementations of these algorithms need side-channel countermeasures.

Theoretical and Real-World Security

- In 2022, the US National Institute of Standards and Technology (NIST) selected four algorithms for its post-quantum cryptography standardization process.
- In theory, these algorithms can base their security on problems that are considered still hard given large-scale quantum computing.
- In practice, the implementations of these algorithms need side-channel countermeasures.
- There exist attacks on FALCON that have not been addressed (until our paper).

Attacks on FALCON

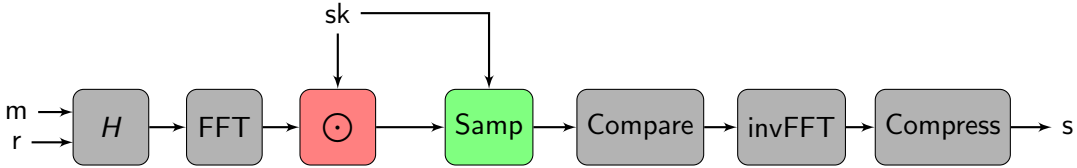


Figure: A graphical overview of FALCON.Sign.

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

Attacks on FALCON

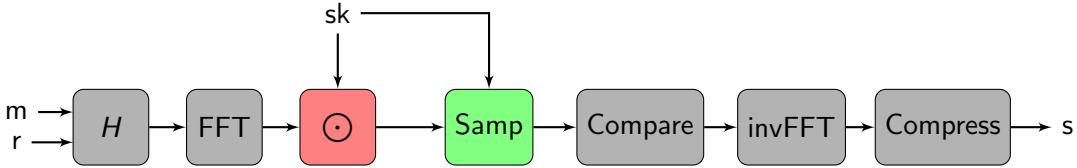


Figure: A graphical overview of FALCON.Sign.

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

Notation

Throughout the presentation, we assume

Notation

Throughout the presentation, we assume

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

Notation

Throughout the presentation, 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

Throughout the presentation, 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

Throughout the presentation, 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

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

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, $\mathbf{s}$)

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)

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

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

- H: hash function
- r: random salt

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)

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

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

- H: hash function
- r: random salt

Verify(m, s)

Check

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

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 $c \leftarrow H(\mathsf{r} \| \mathsf{m})$.

Introduction to Falcon

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

- 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 $\mathbf{c}\mathbf{A}^T = H(\mathsf{r} \| \mathsf{m}) \bmod q$.

Introduction to FALCON

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

- 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 $\mathbf{c}\mathbf{A}^T = H(\mathsf{r} \| \mathsf{m}) \bmod q$.
- 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 $c \leftarrow H(\mathsf{r} \| \mathsf{m})$.
- Find a solution $\mathbf{c} \leftarrow \left[\begin{array}{c|c} c & 0 \end{array}\right]$ such that $\mathbf{c}\mathbf{A}^T = H(\mathsf{r} \| \mathsf{m}) \bmod q$.
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$.
- Apply the nearest plane algorithm (Appendix – Nearest-Plane Algorithm) 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 first

- 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 $\mathbf{c}\mathbf{A}^T = H(\mathsf{r} \| \mathsf{m}) \bmod q$.
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$.
- Apply the nearest plane algorithm (Appendix – Nearest-Plane Algorithm) 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 $\mathbf{s}\mathbf{A}^T = H(\mathsf{r} \| \mathsf{m}) \bmod q$.

Introduction to Falcon

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

- 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 $\mathbf{c}\mathbf{A}^T = H(\mathsf{r} \| \mathsf{m}) \bmod q$.
- Compute the pre-image vector $\mathbf{t} \leftarrow \mathbf{c}\mathbf{B}^{-1}$.
- Apply the nearest plane algorithm (Appendix – Nearest-Plane Algorithm) 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 $\mathbf{s}\mathbf{A}^T = H(\mathsf{r} \| \mathsf{m}) \bmod q$.

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

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

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

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

Fast-Fourier Transform

The pre-image vector computation includes polynomial multiplications

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

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

$$\frac{1}{q} \left[\begin{array}{c|c} \mathsf{FFT}(c) \odot \mathsf{FFT}(-F) & \mathsf{FFT}(c) \odot \mathsf{FFT}(f) \end{array}\right]$$

The pre-image vector computation is 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

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

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

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 mantissa

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
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 mantissa
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 mantissa
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 mantissa
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 mantissa
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 mantissa
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)

Sticky Bit

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

$$100100100 \gg 4 \rightarrow$$

Sticky Bit

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

$$100100100 \gg 4 \rightarrow 1001 \underbrace{1}_{\text{Sticky}}$$

Sticky Bit

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

$$100100100 \gg 4 \rightarrow 1001 \underbrace{1}_{\text{Sticky}}$$

It indicates whether there exists any 1 after the least significant bit.

Sticky Bit

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

$$100100100 \gg 4 \to 1001 \underbrace{1}_{\text{Sticky}}$$

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

$$\text{sticky bit } 1 = 0 \vee [\![(0100) \neq 0]\!] = [\![(00100) \neq 0]\!]$$

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

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** FPR(s, e, z)

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 $z \in [2^{63}, 2^{64})$
15: $z \leftarrow (z \gg 9) \vee [\![z^{[9:1]} \neq 0]\!]$
16: $ex \leftarrow ex + 9$
17: **return** FPR(sx, ex, z)

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 Attacks

- Power consumption during the execution of programs depends on intermediate values.

Power Analysis Attacks

- Power consumption during the execution of programs depends on intermediate values.
- Power analysis attacks leverage this fact to find the secret key.

Power Analysis Attacks

- Power consumption during the execution of programs depends on intermediate values.
- Power analysis attacks leverage this fact to find the secret key.

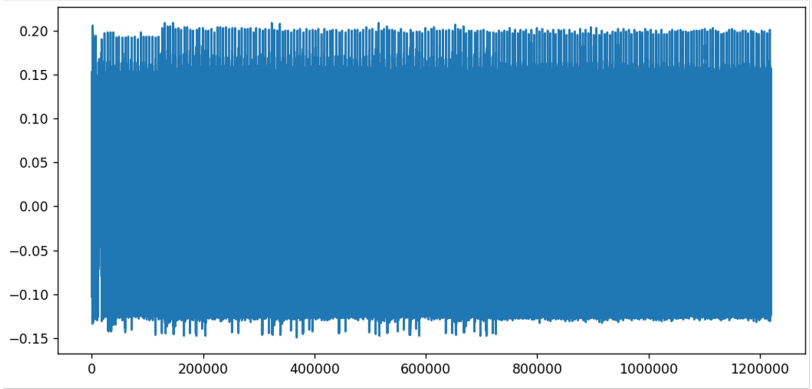


Figure: An Example of a Power Trace

Masking

Masking defends against such threats by secret-sharing the sensitive variables.

Masking

Masking defends against such threats 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 defends against such threats by secret-sharing the sensitive variables.

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

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

- Arithmetic Masking: A variable x is split into n shares $(x_i)_{1 \le i \le n}$ (when stored in a k-bit register) such that

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

Masking

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

Masking

- In each run, all x_i's are randomized so that any $n-1$ shares of them are independently and uniformly distributed.
- All operations need to be operated via shares.

Masking

- In each run, all x_i's are randomized so that any $n-1$ shares of them are independently and uniformly distributed.
- All operations need to be operated via shares.

For example, if x is a secret variable, the operation $y \leftarrow \mathsf{pt} \oplus x$ will become

$$\begin{cases} y_1 \leftarrow \mathsf{pt} \oplus x_1 \\ y_2 \leftarrow x_2 \end{cases} \quad \text{where } x_1, x_2 \text{ are sampled uniformly such that } x_1 \oplus x_2 = x$$

The variables with secret information are splitted into shares.

Table of Contents

1 Introduction

2 Preliminaries

3 Masked Floating-Point Number Multiplication and Addition
- Overview of Our Approach
- Tricks to Removing Branches

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
- Tricks to Removing Branches

4 Evaluation and Implementation

5 Conclusion

Overview of Our Approach

An intuitive approach to mask an algorithm:

Overview of Our Approach

An intuitive approach to mask an algorithm:

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

Overview of Our Approach

An intuitive approach to mask an algorithm:

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

Overview of Our Approach

An intuitive approach to mask an 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

An intuitive approach to mask an algorithm:

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

and use the following gadgets if necessary:

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

Overview of Our Approach

An intuitive approach to mask an algorithm:

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

and use the following gadgets if necessary:

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

Overview of Our Approach

For example, if x is a secret variable, the operation $y \leftarrow \mathsf{pt} \oplus x; \quad z \leftarrow y \times x$ will become

Overview of Our Approach

For example, if x is a secret variable, the operation $y \leftarrow \mathsf{pt} \oplus x$; $z \leftarrow y \times x$ will become

Input: pt, x
Output: (z_1, z_2) such that $z_1 + z_2 = (\mathsf{pt} \oplus x) \times x$
1: Sample x_1, x_2 uniformly such that $x_1 \oplus x_2 = x$
2: $y_1 \leftarrow \mathsf{pt} \oplus x_1$
3: $y_2 \leftarrow x_2$
4: $(x_1', x_2') \leftarrow \mathsf{B2A}((x_1, x_2))$ // $x_1' + x_2' = x_1 \oplus x_2 = x$
5: $(y_1', y_2') \leftarrow \mathsf{B2A}((y_1, y_2))$ // $y_1' + y_2' = y_1 \oplus y_2 = y$
6: $z_1 \leftarrow y_1 \times x_1 + y_2 \times x_2$
7: $z_2 \leftarrow y_1 \times x_2 + y_2 \times x_1$

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:

- Check whether a secret value is nonzero
  - Given (x_i), check 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:

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

Overview of Our Approach

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

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

Overview of Our Approach

Let's see where these operations are:

- Check whether a secret value is nonzero
- Right-shift a secret value by another secret value
- Normalize a secret value to $[2^{63}, 2^{64})$

FprMul:

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

FprAdd:

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)

Overview of Our Approach

Let's see where these operations are:

- Check whether a secret value is nonzero
- Right-shift a secret value by another secret value
- Normalize a secret value to $[2^{63}, 2^{64})$

FprMul:

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

FprAdd:

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)

Overview of Our Approach

Let's see where these operations are:

- Check whether a secret value is nonzero
- Right-shift a secret value by another secret value
- Normalize a secret value to $[2^{63}, 2^{64})$

FprMul:

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

FprAdd:

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)

Overview of Our Approach

Let's see where these operations are:

- Check whether a secret value is nonzero
- Right-shift a secret value by another secret value
- Normalize a secret value to $[2^{63}, 2^{64})$

FprMul:

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

FprAdd:

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)

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 check whether a secret value is nonzero

SecNonzero

Given shares (x_i), output one-bit shares (b_i) such that

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

Overview of Our Approach

We design novel gadgets for these three operations, including:

- SecNonzero: securely check whether a secret value is nonzero
- SecFprUrsh: securely right-shift a secret value by another secret value

SecFprUrsh

Given 64-bit shares (x_i) and 6-bit shares (c_i), output shares (z_i) such that

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

Overview of Our Approach

We design novel gadgets for these three operations, including:

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

SecFprNorm64

Given 64-bit shares (x_i) and 16-bit shares (e_i), output shares (x'_i) and (e'_i) such that

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

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

Gadgets from Previous Works

Algorithm	**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]
$\mathsf{B2A}_{\mathsf{Bit}}$	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
- Tricks to Removing Branches

4 Evaluation and Implementation

5 Conclusion

Why Removing Branch

- For cryptographic operations, we need constant-time implementations.

Why Removing Branch

- For cryptographic operations, we need constant-time implementations.
- Branch is usually not allowed in a constant-time implementation.
  - Different operations can cause different running times (and power consumption patterns)
  - Branch prediction

Tricks to Removing Branches

If we want to run the following operations:

1: **if** $a = 0$ **then**
2: $\quad b \leftarrow 0$

Tricks to Removing Branches

If we want to run the following operations:

1: **if** $a = 0$ **then**
2: $\quad b \leftarrow 0$

Suppose a is either 0 or 1, we can write it as

1: $b \leftarrow b \wedge (-a)$

Tricks to Removing Branches

If we want to run the following operations:

1: **if** $a = 0$ **then**
2: $\quad b \leftarrow 0$

Suppose a is either 0 or 1, we can write it as

1: $b \leftarrow b \wedge (-a)$

Now, for Boolean-shared values in our design

1: $(b_i) \leftarrow \mathsf{SecAnd}((b_i), (-a_i))$

We utilize that $\bigoplus_{i=1}^{n} -a_i = -\bigoplus_{i=1}^{n} a_i = -a$, which is not true for a general k-bit a.

Tricks to Removing Branches

Similarly, for operations

1: **if** $a = 1$ **then**
2: $\quad b \leftarrow 0$

Tricks to Removing Branches

Similarly, for operations

1: **if** $a = 1$ **then**
2: $\quad b \leftarrow 0$

Suppose a is either 0 or 1, we can write it as

1: $b \leftarrow b \wedge (\neg(-a))$

Tricks to Removing Branches

Similarly, for operations

1: **if** $a = 1$ **then**
2: $\quad b \leftarrow 0$

Suppose a is either 0 or 1, we can write it as

1: $b \leftarrow b \wedge (\neg(-a))$

For Boolean-shared values,

1: $(c_i) \leftarrow (-a_i)$
2: $(b_i) \leftarrow \mathsf{SecAnd}((b_i), (\neg c_1, c_2, \cdots, c_n))$

We utilize that $\neg \left(\bigoplus_{i=1}^{n} c_i\right) = (\neg c_1) \oplus \left(\bigoplus_{i=2}^{n} c_i\right)$.

Tricks to Removing Branches

Moreover, for operations,

1: **if** $a = 1$ **then**
2: $\quad b \leftarrow c$

Tricks to Removing Branches

Moreover, for operations,

1: **if** $a = 1$ **then**
2: $\quad b \leftarrow c$

Suppose a is either 0 or 1, we may write it as

1: $d \leftarrow b \oplus c$
2: $b \leftarrow b \oplus (d \wedge (-a))$

Tricks to Removing Branches

Moreover, for operations,

1: **if** $a = 1$ **then**
2: $\quad b \leftarrow c$

Suppose a is either 0 or 1, we may write it as

1: $d \leftarrow b \oplus c$
2: $b \leftarrow b \oplus (d \wedge (-a))$

For Boolean-shared values,

1: $(d_i) \leftarrow (b_i \oplus c_i)$
2: $(d_i) \leftarrow \mathsf{SecAnd}((d_i), (-a_i))$
3: $(b_i) \leftarrow (b_i \oplus d_i)$

Tricks in Masking FPR

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

Tricks in Masking FprMul

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** FPR(s, e, z)

Tricks in Masking FprAdd

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)

Wrapping-up

Utilizing new gadgets and the tricks, we design the following gadgets:

Wrapping-up

Utilizing new gadgets and the tricks, we design the following gadgets:

- SecFPR: Secure FPR by masking.

Wrapping-up

Utilizing new gadgets and the tricks, we design the following gadgets:

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

Wrapping-up

Utilizing new gadgets and the tricks, we design the following gadgets:

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

Wrapping-up

Utilizing new gadgets and the tricks, 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 concrete implementations and several tricks for improvements in Appendix – Details of Our Design.

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

Probing Model

To theoretically evaluate the security, we consider the probing model [ISW03].

Probing Model

To theoretically evaluate the security, we consider the probing model [ISW03].

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

Probing Model

To theoretically evaluate the security, we consider the probing model [ISW03].

- The t-probing model assumes that an adversary is able to peek any t intermediate values in the algorithm.
- To be secure in the t-probing model (t-probing secure), $n \geq t + 1$.

Probing Model

To theoretically evaluate the security, we consider the probing model [ISW03].

- The t-probing model assumes that an adversary is able to peek any t intermediate values in the algorithm.
- To be secure in the t-probing model (t-probing secure), $n \geq t + 1$.
- It is complicated to prove t-probing security directly, so we apply the concept of *non-interference security*.

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.

Appendix - Examples of Non-Interference Security

Non-Interference Security

Takeaway:

Non-Interference Security

Takeaway:

- If a gadget is t-(S)NI secure for $t = n - 1$, and if any $n - 1$ input shares are independent to the secret, then the gadget is t-probing secure.

Non-Interference Security

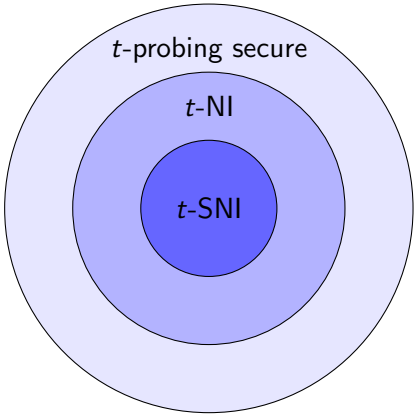
Takeaway:

- If a gadget is t-(S)NI secure for $t = n - 1$, and if any $n - 1$ input shares are independent to the secret, then the gadget is t-probing secure.
- t-SNI is stronger than t-NI by definition.

Non-Interference Security

Takeaway:

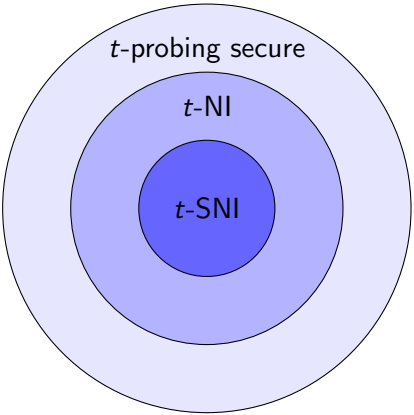
- If a gadget is t-(S)NI secure for $t = n - 1$, and if any $n - 1$ input shares are independent to the secret, then the gadget is t-probing secure.
- t-SNI is stronger than t-NI by definition.



Non-Interference Security

Takeaway:

- If a gadget is t-(S)NI secure for $t = n - 1$, and if any $n - 1$ input shares are independent to the secret, then the gadget is t-probing secure.
- t-SNI is stronger than t-NI by definition.
- A composition of t-NI gadgets may not be t-NI, so we insert t-SNI gadgets to make it t-NI or t-SNI.



Gadgets in Our Work

Algorithm	**Security**	**Algorithm**	**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/algorithms in our work with $n = t + 1$ shares

Test Vector Leakage Assessment (TVLA)

For practical security validation, we apply the Test Vector Leakage Assessment (TVLA) [GJR+11].

Test Vector Leakage Assessment (TVLA)

For practical security validation, we apply the Test Vector Leakage Assessment (TVLA) [GJR+11].

A tester records two sets of traces where

Test Vector Leakage Assessment (TVLA)

For practical security validation, we apply the Test Vector Leakage Assessment (TVLA) [GJR+11].

A tester records two sets of traces where

- Set 1: fixed input

Test Vector Leakage Assessment (TVLA)

For practical security validation, we apply the Test Vector Leakage Assessment (TVLA) [GJR+11].

A tester records two sets of traces where

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

Test Vector Leakage Assessment (TVLA)

For practical security validation, we apply the Test Vector Leakage Assessment (TVLA) [GJR+11].

A tester records two sets of traces where

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

The Welch's t-test is then applied.

Test Vector Leakage Assessment (TVLA)

For practical security validation, we apply the Test Vector Leakage Assessment (TVLA) [GJR+11].

A tester records two sets of traces where

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

The Welch's t-test is then applied.

$$t = \frac{\bar{x}_f - \bar{x}_r}{\sqrt{\frac{s_f^2}{n_f} + \frac{s_r^2}{n_r}}}$$

- $\bar{x}_f, \bar{x}_r$: Sample means.
- s_f^2, s_r^2: Sample variances.
- n_f, n_r: Sample sizes.

Test Vector Leakage Assessment (TVLA)

For practical security validation, we apply the Test Vector Leakage Assessment (TVLA) [GJR+11].

A tester records two sets of traces where

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

The Welch's t-test is then applied.

$$t = \frac{\bar{x}_f - \bar{x}_r}{\sqrt{\frac{s_f^2}{n_f} + \frac{s_r^2}{n_r}}}$$

- $\bar{x}_f, \bar{x}_r$: Sample means.
- s_f^2, s_r^2: Sample variances.
- n_f, n_r: Sample sizes.

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

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
FprMul/SecFprMul	308	7134 (23×)	36388 (118×)
FprAdd/SecFprAdd	487	17154 (35×)	48291 (99×)

Table: Performance evaluation of SecFprMul and SecFprAdd

Performance Evaluation on Intel-Core i9-12900KF

We also test the time for signing one message on a general-purpose CPU.

Security Level	**Unmasked**	**2 Shares**	**3 Shares**
Falcon-512	246.56	1905.55 (7.7×)	6137.25 (24.9×)
Falcon-1024	501.62	3819.76 (7.6×)	12287.29 (24.5×)

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 of FALCON.

Conclusion

In this paper,

- We present the first masking scheme for floating-point number multiplication and addition to protect the pre-image vector computation of FALCON.
- 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 of FALCON.
- We design novel gadgets SecNonzero, SecFprUrsh, and SecFprNorm64.
- All our masked gadgets are proven either 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 of FALCON.
- We design novel gadgets SecNonzero, SecFprUrsh, and SecFprNorm64.
- All our masked gadgets are proven either t-NI or t-SNI secure.
- Our design pass the TVLA test in 10,000 (for 2-shared) or 100,000 (for 3-shared) 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 of FALCON.
- We design novel gadgets SecNonzero, SecFprUrsh, and SecFprNorm64.
- All our masked gadgets are proven either t-NI or t-SNI secure.
- Our design pass the TVLA test in 10,000 (for 2-shared) or 100,000 (for 3-shared) traces.
- Our countermeasure compared to the unmasked reference implementation is slow.

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.

[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.

[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.

[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.

[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.

Reference II

[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.

[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.

[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.

[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.

Reference III

[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.

[CC24] Keng-Yu Chen and Jiun-Peng Chen. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024.2 (Mar. 2024), pp. 276–303. DOI: 10.46586/tches.v2024.i2.276-303. URL: https://tches.iacr.org/index.php/TCHES/article/view/11428.

Table of Contents

6 Appendix – Nearest-Plane Algorithm

7 Appendix – Details of Our Design

8 Appendix - Examples of Non-Interference Security

Randomized Nearest-Plane Algorithm [GPV08]

Randomized Nearest-Plane Algorithm [GPV08]

Input: $\mathbf{t} = \mathbf{c}\mathbf{B}^{-1}, \mathbf{B}$ where $\mathbf{B} = \tilde{\mathbf{B}}\mathbf{U}$ is the Gram-Schmidt Orthogonalization, constant $\sigma > 0$

Output: $\mathbf{z} = (z_1, z_2, \cdots, z_{2N})$

1: **for** $i = 2N$ **to** 1 **do**

2: $\quad t'_i \leftarrow t_i + \sum_{j>i} \mathbf{U}_{ij}(t_j - z_j)$

3: $\quad \sigma_i \leftarrow \frac{\sigma}{||\tilde{\mathbf{b}}_i||}$ // $\tilde{\mathbf{b}}_i$ is the i-th row vector of $\tilde{\mathbf{B}}$

4: $\quad z_i \leftarrow_{\$} D_{\mathbb{Z},\sigma_i,t'_i}$ // Sample a value z_i from a discrete Gaussian distribution

Randomized Nearest-Plane Algorithm [GPV08]

Randomized Nearest-Plane Algorithm [GPV08]

Input: $\mathbf{t} = \mathbf{c}\mathbf{B}^{-1}, \mathbf{B}$ where $\mathbf{B} = \tilde{\mathbf{B}}\mathbf{U}$ is the Gram-Schmidt Orthogonalization, constant $\sigma > 0$
Output: $\mathbf{z} = (z_1, z_2, \cdots, z_{2N})$

1: **for** $i = 2N$ **to** 1 **do**
2: $\quad t'_i \leftarrow t_i + \sum_{j>i} \mathbf{U}_{ij}(t_j - z_j)$
3: $\quad \sigma_i \leftarrow \frac{\sigma}{\|\tilde{\mathbf{b}}_i\|}$ // $\tilde{\mathbf{b}}_i$ is the i-th row vector of $\tilde{\mathbf{B}}$
4: $\quad z_i \leftarrow\!\!\!\!\$\ D_{\mathbb{Z},\sigma_i,t'_i}$ // Sample a value z_i from a discrete Gaussian distribution

Lemma 4.5 in [GPV08]

If $\sigma \geq \|\tilde{\mathbf{B}}\| \cdot \omega(\sqrt{\log(N)}) = \max_i \|\tilde{\mathbf{b}}_i\| \cdot \omega(\sqrt{\log(N)})$, then $\mathbf{z}\mathbf{B} \overset{\Delta}{\sim} D_{\mathcal{L}(\mathbf{B}),\sigma,\mathbf{c}}$.

Randomized Nearest-Plane Algorithm [GPV08]

Randomized Nearest-Plane Algorithm [GPV08]

Input: $\mathbf{t} = \mathbf{c}\mathbf{B}^{-1}, \mathbf{B}$ where $\mathbf{B} = \tilde{\mathbf{B}}\mathbf{U}$ is the Gram-Schmidt Orthogonalization, constant $\sigma > 0$
Output: $\mathbf{z} = (z_1, z_2, \cdots, z_{2N})$

1: **for** $i = 2N$ **to** 1 **do**
2: $\quad t_i' \leftarrow t_i + \sum_{j>i} \mathbf{U}_{ij}(t_j - z_j)$
3: $\quad \sigma_i \leftarrow \frac{\sigma}{\|\tilde{\mathbf{b}}_i\|}$ // $\tilde{\mathbf{b}}_i$ is the i-th row vector of $\tilde{\mathbf{B}}$
4: $\quad z_i \leftarrow_{\$} D_{\mathbb{Z},\sigma_i,t_i'}$ // Sample a value z_i from a discrete Gaussian distribution

Lemma 4.5 in [GPV08]

If $\sigma \geq \|\tilde{\mathbf{B}}\| \cdot \omega(\sqrt{\log(N)}) = \max_i \|\tilde{\mathbf{b}}_i\| \cdot \omega(\sqrt{\log(N)})$, then $\mathbf{z}\mathbf{B} \overset{\Delta}{\sim} D_{\mathcal{L}(\mathbf{B}),\sigma,\mathbf{c}}$.

FALCON uses fast Fourier nearest plane algorithm [DP16] to further speed up.

Table of Contents

6 Appendix – Nearest-Plane Algorithm

7 Appendix – Details of Our Design
- New Gadgets
- SecFPR: Secure FPR
- SecFprMul: Secure FprMul
- SecFprAdd: Secure FprAdd

8 Appendix - Examples of Non-Interference Security

Table of Contents

6 Appendix – Nearest-Plane Algorithm

7 Appendix – Details of Our Design
- New Gadgets
- SecFPR: Secure FPR
- SecFprMul: Secure FprMul
- SecFprAdd: Secure FprAdd

8 Appendix - Examples of Non-Interference Security

SecNonzero

We need a gadget that, given shares (x_i), can derive one-bit shares (b_i) 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), can derive one-bit shares (b_i) 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 considering 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), can derive one-bit shares (b_i) 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 considering OR-ing all the bits.

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

Now we turn to a gadget for secure OR operations.

SecOr: OR of Boolean Shares

SecOr

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

1: $(t_i)_{1 \le i \le n} \leftarrow (\neg x_1, x_2, \cdots, x_n)$
2: $(s_i)_{1 \le i \le n} \leftarrow (\neg y_1, y_2, \cdots, y_n)$
3: $(z_i) \leftarrow \mathsf{SecAnd}((s_i), (t_i))$
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.

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

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 $(x_i)_{1\le i\le n}$ for value x, bitsize

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

if input (x_i) are arithmetic shares **then**
  $(t_i)_{1\le i\le \frac{n}{2}} \leftarrow \mathsf{A2B}((x_i)_{1\le i\le \frac{n}{2}})$
  $(t_i)_{\frac{n}{2}+1\le i\le n} \leftarrow \mathsf{A2B}((-x_i)_{\frac{n}{2}+1\le i\le n})$
else
  $(t_i)_{1\le i\le n} \leftarrow (x_i)_{1\le i\le n}$
$\mathsf{len} \leftarrow \mathsf{bitsize}/2$
while $\mathsf{len} \ge 1$ **do**
  $(l_i) \leftarrow \mathsf{Refresh}((t_i^{[2\mathsf{len}:\mathsf{len}]}), \mathsf{len})$
  $(r_i) \leftarrow (t_i^{[\mathsf{len}:1]})$
  $(t_i) \leftarrow \mathsf{SecOr}((l_i), (r_i))$
  $\mathsf{len} \leftarrow \mathsf{len} \gg 1$
return $(t_i^{(1)})$

SecFprUrsh

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

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

SecFprUrsh

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

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

We observe that

SecFprUrsh

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

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

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 (c_i), we need to derive shares (z_i) such that

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

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

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

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) by $c_1, c_2, \cdots, c_n$ sequentially.

SecFprUrsh

Hence, our idea is to right-rotate all (x_i) 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) 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. To clear the redundant high bits, consider

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

SecFprUrsh

Hence, our idea is to right-rotate all (x_i) 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. To clear the redundant high bits, consider

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

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

SecFprUrsh

Hence, our idea is to right-rotate all (x_i) 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. To clear the redundant high bits, consider

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

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

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 $(x_i)_{1\le i\le n}$
Input: 6-bit arithmetic shares $(c_i)_{1\le i\le n}$
Output: Boolean shares $(z_i)_{1\le i\le n}$ for value $z = x \gg c$ with the sticky bit preserved

1: $(m_i)_{1\le i\le n} \leftarrow ((1 \ll 63), 0, \cdots, 0)$
2: **for** $j = 1$ to n **do**
3: Right-rotate (x_i) by c_j
4: $(x_i) \leftarrow \mathsf{RefreshMasks}((x_i))$
5: Right-rotate (m_i) by c_j
6: $(m_i) \leftarrow \mathsf{RefreshMasks}((m_i))$
7: $\mathsf{len} \leftarrow 1$
8: **while** $\mathsf{len} \le 32$ **do**
9: $(m_i) \leftarrow (m_i \oplus (m_i \gg \mathsf{len}))$
10: $\mathsf{len} \leftarrow \mathsf{len} \ll 1$
11: $(y_i) \leftarrow \mathsf{SecAnd}((x_i), (m_i))$
12: $(z_i) \leftarrow (y_i \oplus x_i \oplus y_i^{(1)})$
13: $(b_i) \leftarrow \mathsf{SecNonzero}((z_i))$
14: $(z_i) \leftarrow (y_i^{[64:2]} \vee b_i)$
15: **return** (z_i)

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

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

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

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

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

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

We can repeatedly check whether $(x_i^{(64)}) = 0$, conditionally shift 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 new (x_i') and (e_i') such that

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

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

We can repeatedly check whether $(x_i^{(64)}) = 0$, conditionally shift by 1 bit, and then decrease (e_i) by $[\![(x_i^{(64)}) = 0]\!]$.
To improve efficiency, we consider sequentially checking $x^{[64:64-2^j]} = 0$ for $j = 5, 4, \cdots, 0$.

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

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

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

We can repeatedly check whether $(x_i^{(64)}) = 0$, conditionally shift by 1 bit, and then decrease (e_i) by $[\![(x_i^{(64)}) = 0]\!]$.
To improve efficiency, we consider sequentially checking $x^{[64:64-2^j]} = 0$ for $j = 5, 4, \cdots, 0$.
In addition, we first decrease (e_i) by 63 and later add $[\![(x_i^{[64:64-2^j]}) \neq 0]\!] \cdot 2^j$ to it.

SecFprNorm64

SecFprNorm64

Input: 64-bit Boolean shares $(x_i)_{1\le i\le n}$
Input: 16-bit arithmetic shares $(e_i)_{1\le i\le n}$
Output: Normalized $(x_i)_{1\le i\le n}$ in $[2^{63}, 2^{64})$ and $(e_i)_{1\le i\le n}$ with shift added

1: $e_1 \leftarrow e_1 - 63$
2: **for** $j = 5$ to 0 **do**
3: $\quad (t_i) \leftarrow (x_i \oplus (x_i \ll 2^j))$
4: $\quad (n_i) \leftarrow (x_i \gg (64 - 2^j))$
5: $\quad (b_i) \leftarrow \mathsf{SecNonzero}((n_i))$
6: $\quad (b'_i) \leftarrow (-b_i)$
7: $\quad (t_i) \leftarrow \mathsf{SecAnd}((t_i), (\neg b'_1, b'_2, \cdots, b'_n))$
8: $\quad (x_i) \leftarrow (x_i \oplus t_i)$
9: $\quad (b_i) \leftarrow \mathsf{B2A}_{\mathsf{Bit}}((b_i))$
10: $\quad (e_i) \leftarrow (e_i + (b_i \ll j))$
11: **return** $(x_i), (e_i)$

Table of Contents

6 Appendix – Nearest-Plane Algorithm

7 Appendix – Details of Our Design
- New Gadgets
- SecFPR: Secure FPR
- SecFprMul: Secure FprMul
- SecFprAdd: Secure FprAdd

8 Appendix - Examples of Non-Interference Security

SecFPR: Secure FPR

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: $x \leftarrow x + f^{(1)}$
9: **return** x

We now show how we mask the floating-point number rounding and packing algorithm FPR.

SecFPR: Secure FPR

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: $x \leftarrow x + f^{(1)}$
9: **return** x

Recall that FPR is the last subroutine of FprMul and FprAdd.
By our masking design of FprMul and FprAdd, (s_i) is Boolean-masked, (e_i) is 16-bit arithmetic-masked, and (z_i) is Boolean-masked.

SecFPR: Secure FPR

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: $x \leftarrow x + f^{(1)}$
9: **return** x

This is by adding to any one share.

SecFPR: Secure FPR

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: $x \leftarrow x + f^{(1)}$
9: **return** x

This is equivalent to

1: **if** $e < 0$ **then**
2: $\quad z \leftarrow 0$

and is done by an A2B, taking the MSB, and the simple trick.

SecFPR: Secure FPR

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: $x \leftarrow x + f^{(1)}$
9: **return** x

This is done by SecNonzero and the simple trick.

SecFPR: Secure FPR

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: $x \leftarrow x + f^{(1)}$
9: **return** x

Shift, OR, and a SecAdd.
We add (e_i) and the 55th bit of (z_i) in advance instead of adding (e_i) to a 64-bit value. That is, use a 16-bit SecAdd to save a 64-bit SecAdd

SecFPR: Secure FPR

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: $x \leftarrow x + f^{(1)}$
9: **return** x

If the least 3 bits of (z_i) are 011, 110, and 111, $f^{(1)} = 1$.
We OR $(z_i^{(1)})$ and $(z_i^{(3)})$ by SecOr, and then AND $(z_i^{(2)})$ by SecAnd. The result is then added to (x_i) by SecAdd.

SecFPR: Secure FPR

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: $x \leftarrow x + f^{(1)}$
9: **return** x

Done.
The returned value is a 64-bit Boolean-masked (x_i).

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

Table of Contents

6 Appendix – Nearest-Plane Algorithm

7 Appendix – Details of Our Design
- New Gadgets
- SecFPR: Secure FPR
- SecFprMul: Secure FprMul
- SecFprAdd: Secure FprAdd

8 Appendix - Examples of Non-Interference Security

SecFprMul: Secure FprMul

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** FPR(s, e, z)

We show how we mask the floating-point number multiplication algorithm FprMul.

SecFprMul: Secure FprMul

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** FPR(s, e, z)

We assume (sx_i) and (sy_i) are Boolean shares, (ex_i) and (ey_i) are 16-bit arithmetic shares, and (mx_i) and (my_i) are 128-bit arithmetic shares, which can load the product of two 53-bit values.

SecFprMul: Secure FprMul

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** FPR(s, e, z)

These can be operated share-wise.

SecFprMul: Secure FprMul

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** $\mathrm{FPR}(s, e, z)$

This is done by SecMult. For further operations, we then apply an A2B to turn them to Boolean shares.

SecFprMul: Secure FprMul

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** $\mathrm{FPR}(s, e, z)$

Conditional shift by 50 bits and 51 bits, depending on $z^{(106)}$, while preserving the sticky bit. These can be done by SecNonzero and SecOr.

SecFprMul: Secure FprMul

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** FPR(s, e, z)

We observe that we can save one SecOr.

- When shifted by 50 bits, we OR the last bit with $z^{[50:1]}$.
- When shifted by 51 bits, we OR the last bit with $z^{[51:1]}$.

We can simply OR the the last bit with $z^{[51:1]}$, regardless of the conditional shift result.

SecFprMul: Secure FprMul

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** FPR(s, e, z)

This is by adding to any share.

SecFprMul: Secure FprMul

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** FPR(s, e, z)

This is by SecNonzero and SecAnd, and applying the tricks.

SecFprMul: Secure FprMul

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** FPR(s, e, z)

Now it calls FPR to return a 64-bit Boolean-masked FPN.

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: $(p_i) \leftarrow \mathsf{SecMult}((mx_i), (my_i))$
4: $(p_i) \leftarrow \mathsf{A2B}((p_i))$
5: $(b_i) \leftarrow \mathsf{SecNonzero}((p_i^{[51:1]}))$
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: $(z_i') \leftarrow \mathsf{SecAnd}((z_i'), \mathsf{Refresh}((-w_i)))$
10: $(z_i) \leftarrow (z_i' \oplus z_i)$
11: $(z_i) \leftarrow \mathsf{SecOr}((z_i), (b_i))$
12: $(w_i) \leftarrow \mathsf{B2A}_{\mathsf{Bit}}((w_i))$
13: $(e_i) \leftarrow (e_i + w_i)$
14: $(bx_i) \leftarrow \mathsf{SecNonzero}((ex_i))$
15: $(by_i) \leftarrow \mathsf{SecNonzero}((ey_i))$
16: $(d_i) \leftarrow \mathsf{SecAnd}((bx_i), (by_i))$
17: $(z_i) \leftarrow \mathsf{SecAnd}((z_i), (-d_i^{(1)}))$
18: **return** $\mathsf{SecFPR}((s_i), (e_i), (z_i))$

Table of Contents

6 Appendix – Nearest-Plane Algorithm

7 Appendix – Details of Our Design
- New Gadgets
- SecFPR: Secure FPR
- SecFprMul: Secure FprMul
- SecFprAdd: Secure FprAdd

8 Appendix - Examples of Non-Interference Security

SecFprAdd: Secure FprAdd

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)

We show how we mask the floating-point number addition algorithm FprAdd.

SecFprAdd: Secure FprAdd

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)

By the output of SecFprMul, we assume the input shares (x_i) and (y_i) are 64-bit Boolean-masked FPNs.

SecFprAdd: Secure FprAdd

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)

The subtraction of two Boolean-masked values can be operated by considering $x^{[63:1]} - y^{[63:1]} = x^{[63:1]} + (\neg y^{[63:1]}) + 1$, which takes two SecAdds.

SecFprAdd: Secure FprAdd

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)

But since we only need $(x^{[63:1]} - y^{[63:1]})^{(64)}$, we only compute $x^{[63:1]} + (\neg y^{[63:1]})$ and then check the boundary conditions. This saves us one SecAdd.

SecFprAdd: Secure FprAdd

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)

That is, $[\![u - v < 0]\!] = [\![u + (\neg v) < 0]\!] \oplus [\![u + (\neg v) = -1]\!] \oplus [\![u + (\neg v) = 2^{63} - 1]\!]$

SecFprAdd: Secure FprAdd

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)

Moreover, we apply $u + (\neg v) \neq -1 \Leftrightarrow \neg(u + (\neg v)) \neq 0$ and
$u + (\neg v) \neq 2^{63} - 1 \Leftrightarrow (u + (\neg v)) \oplus (1 \ll 63) \neq -1 \Leftrightarrow \neg((u + (\neg v)) \oplus (1 \ll 63)) \neq 0$

SecFprAdd: Secure FprAdd

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)

Therefore, these operations can be computed by SecNonzero, SecAnd, and SecOr.

SecFprAdd: Secure FprAdd

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)

Share-wise operations, two B2As to convert (ex_i) and (ey_i) to arithmetic shares, and subtractions to any shares.

SecFprAdd: Secure FprAdd

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)

Subtraction to any share of c by 60 and an A2B to get the MSB of c. Then apply the tricks.

SecFprAdd: Secure FprAdd

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)

This is by our gadget SecFprUrsh.

SecFprAdd: Secure FprAdd

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)

A share-wise operation.

SecFprAdd: Secure FprAdd

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)

Use the tricks with $my + (my \oplus (-my)) \wedge -s$, where $-my = (\neg my) + 1$ is derived by an SecAdd. Then add the result to mx.

SecFprAdd: Secure FprAdd

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)

This is by our gadget SecFprNorm64.

SecFprAdd: Secure FprAdd

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)

A share-wise operation and a SecNonzero. Add 9 to any share of ex.

SecFprAdd: Secure FprAdd

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)

Finally, it calls FPR to return a 64-bit Boolean-masked FPN.

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: $(d_i) \leftarrow \mathsf{SecAdd}((xm_i), (ym_i))$
4: $(b_i) \leftarrow \mathsf{SecNonzero}(\neg d_1, d_2, \cdots, d_n)$
5: $(b_i') \leftarrow \mathsf{SecNonzero}(\neg(d_1 \oplus (1 \ll 63)), d_2, \cdots, d_n)$
6: $(cs_i) \leftarrow \mathsf{SecAnd}((\neg b_1, b_2, \cdots, b_n), (x_i^{(64)}))$
7: $(cs_i) \leftarrow \mathsf{SecOr}((cs_i), (d_i^{(64)} \oplus b_i \oplus b_i'))$
8: $(m_i) \leftarrow \mathsf{SecAnd}((x_i \oplus y_i), (-cs_i))$
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: $(ex_i) \leftarrow \mathsf{B2A}((ex_i))$, $(ey_i) \leftarrow \mathsf{B2A}((ey_i))$
13: $ex_1 \leftarrow ex_1 - 1078$, $ey_1 \leftarrow ey_1 - 1078$.
14: $(c_i) \leftarrow (ex_i - ey_i)$
15: $(c_i') \leftarrow \mathsf{A2B}((c_1 - 60, c_2, \cdots, c_n))$
16: $(my_i) \leftarrow \mathsf{SecAnd}((my_i), (-(c_i'^{(16)})))$
17: $(my_i) \leftarrow \mathsf{SecFprUrsh}((my_i), (c_i^{[6:1]}))$
18: $(my_i') \leftarrow (\neg my_1, my_2, \cdots, my_n)$
19: $(my_i') \leftarrow \mathsf{SecAdd}((my_i'), (1, 0, \cdots, 0))$
20: $(s_i) \leftarrow (-(sx_i \oplus sy_i))$
21: $(my_i) \leftarrow \mathsf{Refresh}((my_i))$
22: $(my_i') \leftarrow \mathsf{SecAnd}((my_i \oplus my_i'), (s_i))$
23: $(my_i) \leftarrow (my_i \oplus my_i')$
24: $(z_i) \leftarrow \mathsf{SecAdd}((mx_i), (my_i))$
25: $(z_i), (ex_i) \leftarrow \mathsf{SecFprNorm64}((z_i), (ex_i))$
26: $(b_i) \leftarrow \mathsf{SecNonzero}((z_i^{[10:1]}))$
27: $(z_i) \leftarrow (z_i \gg 9)$
28: $(z_i^{(1)}) \leftarrow (b_i)$
29: $ex_1 \leftarrow ex_1 + 9$
30: **return** $\mathsf{SecFPR}(\mathsf{Refresh}((sx_i)), (ex_i), (z_i))$

Table of Contents

6 Appendix – Nearest-Plane Algorithm

7 Appendix – Details of Our Design

8 Appendix - Examples of Non-Interference Security

t-Non-Interference - Example (XOR of Boolean Shares)

XOR of Boolean Shares

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \oplus y = z_1 \oplus \cdots \oplus z_n$

1: **for** $i = 1$ to n **do**
2: $\quad z_i \leftarrow x_i \oplus y_i$

t-Non-Interference - Example (XOR of Boolean Shares)

XOR of Boolean Shares

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \oplus y = z_1 \oplus \cdots \oplus z_n$
1: **for** $i = 1$ to n **do**
2: $\quad z_i \leftarrow x_i \oplus y_i$

- Intermediate Variables: (z_i).

t-Non-Interference - Example (XOR of Boolean Shares)

XOR of Boolean Shares

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \oplus y = z_1 \oplus \cdots \oplus z_n$
1: **for** $i = 1$ to n **do**
2: $\quad z_i \leftarrow x_i \oplus y_i$

- Intermediate Variables: (z_i).
- Simulator: For any probing set $\{z_i \mid i \in I\}$ of size $|I| = t$, let the simulation set be $\{x_i \mid i \in I\}, \{y_i \mid i \in I\}$. The simulator receives the simulation set and outputs $\{(x_i \oplus y_i) \mid i \in I\}$.

t-Non-Interference - Example (XOR of Boolean Shares)

XOR of Boolean Shares

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \oplus y = z_1 \oplus \cdots \oplus z_n$
1: **for** $i = 1$ to n **do**
2: $\quad z_i \leftarrow x_i \oplus y_i$

- Intermediate Variables: (z_i).
- Simulator: For any probing set $\{z_i \mid i \in I\}$ of size $|I| = t$, let the simulation set be $\{x_i \mid i \in I\}, \{y_i \mid i \in I\}$. The simulator receives the simulation set and outputs $\{(x_i \oplus y_i) \mid i \in I\}$.
- Distribution of the probing set is identical to the distribution of the simulator's output $\{(x_i \oplus y_i) \mid i \in I\}$.

t-Non-Interference - Example (XOR of Boolean Shares)

XOR of Boolean Shares

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \oplus y = z_1 \oplus \cdots \oplus z_n$
1: **for** $i = 1$ to n **do**
2: $\quad z_i \leftarrow x_i \oplus y_i$

- Intermediate Variables: (z_i).
- Simulator: For any probing set $\{z_i \mid i \in I\}$ of size $|I| = t$, let the simulation set be $\{x_i \mid i \in I\}, \{y_i \mid i \in I\}$. The simulator receives the simulation set and outputs $\{(x_i \oplus y_i) \mid i \in I\}$.
- Distribution of the probing set is identical to the distribution of the simulator's output $\{(x_i \oplus y_i) \mid i \in I\}$.
- The size of the simulation set for each input is $|I| \leq t$, no more than the probing set.

t-Non-Interference - Example (XOR of Boolean Shares)

XOR of Boolean Shares

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \oplus y = z_1 \oplus \cdots \oplus z_n$
1: **for** $i = 1$ to n **do**
2: $\quad z_i \leftarrow x_i \oplus y_i$

- Intermediate Variables: (z_i).
- Simulator: For any probing set $\{z_i \mid i \in I\}$ of size $|I| = t$, let the simulation set be $\{x_i \mid i \in I\}, \{y_i \mid i \in I\}$. The simulator receives the simulation set and outputs $\{(x_i \oplus y_i) \mid i \in I\}$.
- Distribution of the probing set is identical to the distribution of the simulator's output $\{(x_i \oplus y_i) \mid i \in I\}$.
- The size of the simulation set for each input is $|I| \leq t$, no more than the probing set.

Therefore, this gadget is t-NI secure for any t.

t-Non-Interference - Example (AND of Boolean Shares)

AND of Boolean Shares

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \wedge y = z_1 \oplus \cdots \oplus z_n$

1: Initialize all z_i to 0.
2: **for** $i = 1$ to n **do**
3: **for** $j = 1$ to n **do**
4: $z_i \leftarrow z_i \oplus (x_i \wedge y_j)$

t-Non-Interference - Example (AND of Boolean Shares)

AND of Boolean Shares

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \wedge y = z_1 \oplus \cdots \oplus z_n$

1: Initialize all z_i to 0.
2: **for** $i = 1$ to n **do**
3: **for** $j = 1$ to n **do**
4: $z_i \leftarrow z_i \oplus (x_i \wedge y_j)$

This gadget is not even 1-NI. If the adversary probes the intermediate value

$$(x_1 \wedge y_1) \oplus (x_1 \wedge y_2)$$

One cannot simulate this without knowing x_1 (1 share of x) and y_1, y_2 (2 shares of y).

t-Strong-Non-Interference - Example (SecAnd)

SecAnd

Input: (x_i) where $x = x_1 \oplus \cdots \oplus x_n$, (y_i) where $y = y_1 \oplus \cdots \oplus y_n$
Output: (z_i) where $x \wedge y = z_1 \oplus \cdots \oplus z_n$

1: **for** $i = 1$ to n **do**
2: $\quad z_i \leftarrow x_i \wedge y_i$
3: **for** $i = 1$ to n **do**
4: $\quad$ **for** $j = i + 1$ to n **do**
5: $\quad\quad r \leftarrow\$ \{0,1\}^k$
6: $\quad\quad z_i \leftarrow z_i \oplus ((x_i \wedge y_j \oplus r) \oplus x_j \wedge y_i)$
7: $\quad\quad z_j \leftarrow z_j \oplus r$

This gadget is t-NI. It is actually also t-SNI.