Masking Floating-Point Number Multiplication and Addition of Falcon

Keng-Yu Chen

Advisor: Jiun-Peng Chen, Ho-Lin Chen

December 29th, 2023

Table of Contents

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

Introduction

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

Theoretical Security - Hardness of Mathematical Problems

In theory, these algorithms can base their security on problems that are considered still hard given the advantage of quantum computing.

1. CRYSTALS-Kyber: Module Learning With Errors (MLWE)
2. CRYSTALS-Dilithium: Module Short Integer Solution (MSIS)
3. Falcon: NTRU Problem and SIS on NTRU lattices
4. SPHINCS+: Security of the used hash function families

Real-World Security – Side-Channel Attacks

In practice, the implementations of these algorithms can suffer side-channel attacks. Fortunately, there are countermeasures for them.

1. CRYSTALS-Kyber: [Bos+21; Fri+22; Hei+22]
2. CRYSTALS-Dilithium: [Mig+19]
3. FALCON: [How+20; Gue+22; Zha+23]
4. SPHINCS+: [Ber+10; Bel+13]

Unfortunately, there are attacks on FALCON that have not been addressed.

Attacks on FALCON

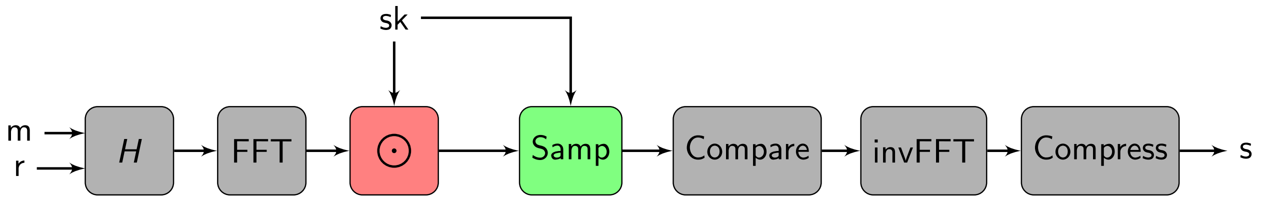

Figure: A graphical overview of FALCON.Sign.

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

Our Contributions

In this paper, we present the following contributions:

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

Notation

Throughout the presentation, we assume

- 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.

Introduction to FALCON

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

KeyGen

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

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

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

Sign(m)

A short $\mathbf{s}$ s.t.

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

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

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

Check

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

Introduction to FALCON

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

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

Introduction to Falcon

In Falcon,

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

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

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

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

Introduction to Falcon

Sign (Simplified)

```
Input: Message m, secret key sk, bound $\lfloor \beta^2 \rfloor$
Output: Signature sig
Sample salt r $\leftarrow \{0,1\}^{320}$ uniformly
$c \leftarrow H(\mathrm{r} \| \mathrm{m})$
Compute the pre-image vector $\mathbf{t} \leftarrow \left[\begin{array}{c|c} c & 0 \end{array}\right] \cdot \mathbf{B}^{-1}$
repeat
    $\mathbf{z} = \mathrm{ffSampling}(\mathbf{t}, \mathrm{sk})$
    $\mathbf{s} = \left[\begin{array}{c|c} s_1 & s_2 \end{array}\right] = (\mathbf{t} - \mathbf{z})\mathbf{B}$
until $\|\mathbf{s}\|^2 \leq \lfloor \beta^2 \rfloor$
sig $\leftarrow (\mathrm{r}, s_2)$
```

Verify (Simplified)

```
Input: Message m, signature sig
Input: Bound $\lfloor \beta^2 \rfloor$
Output: Accept or Reject
$c \leftarrow H(\mathrm{r} \| \mathrm{m})$
$s_1 \leftarrow c - s_2 h \bmod q$
if $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ then
    Accept
else
    Reject
```

Fast-Fourier Transform

The pre-image vector computation includes polynomial multiplications

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

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

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

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

Floating-Point Number

A complex number is represented by two 64-bit floating-point numbers (FPNs). An FPN is composed of sign bit s, exponent e, and mantissa $\tilde{m}$

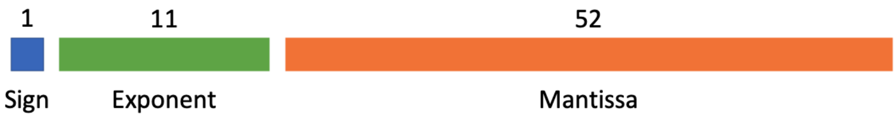

Figure: A 64-bit Floating-Point Number

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

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

FPN addition (FprAdd) is proceeded by

1. Making the first operand $\geq$ the second
2. Right-shifting the second operand
3. Mantissa Addition / Subtraction
4. Normalizing the sum to $[2^{54}, 2^{55})$
5. Combining the results and rounding (FPR)

Sticky Bit

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

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

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

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

Power Analysis Attacks

- Power consumption during the execution of programs depends on intermediate values.
- Power analysis attacks, the side-channel attacks on the pre-image vector computation, leverage this fact to find the secret key.

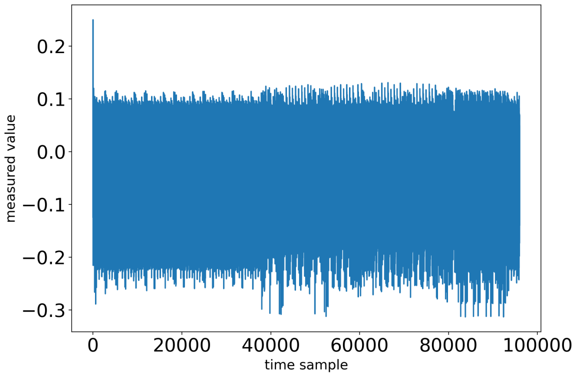

Figure: An Example of a Power Trace

Masking

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

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

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

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

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

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.

Overview of Our Approach

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

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

and use the following gadgets if necessary:

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

Overview of Our Approach

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

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

Overview of Our Approach

We design novel gadgets for these three operations, including:

- SecNonzero (Algorithm 4): securely checking whether a secret value is nonzero.
- SecFprUrsh (Algorithm 5): securely right-shifting a secret value by another secret value
- SecFprNorm64 (Algorithm 6): securely normalizing a secret value to $[2^{63}, 2^{64})$

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

Gadgets Used in Our Work

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]
$\mathrm{B2A}_{\mathrm{Bit}}$	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

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 $z = x \vee y$
$(t_i)_{1\le i\le n} \leftarrow (\neg x_1, x_2, \cdots, x_n)$
$(s_i)_{1\le i\le n} \leftarrow (\neg y_1, y_2, \cdots, y_n)$
$(z_i) \leftarrow \mathsf{SecAnd}((s_i), (t_i))$
$z_1 \leftarrow \neg z_1$
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

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

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
    $(t_i)_{1 \le i \le n} \leftarrow (x_i)_{1 \le i \le n}$
len $\leftarrow$ bitsize$/2$
while len $\ge 1$ do
    $(l_i) \leftarrow \mathsf{Refresh}((t_i^{[2\mathsf{len}:\mathsf{len}]}), \mathsf{len})$
    $(r_i) \leftarrow (t_i^{[\mathsf{len}:1]})$
    $(t_i) \leftarrow \mathsf{SecOr}((l_i), (r_i))$
    len $\leftarrow$ 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

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

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 mod 64.

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

```
$(m_i)_{1 \leq i \leq n} \leftarrow ((1 \ll 63), 0, \cdots, 0)$
for $j = 1$ to $n$ do
    Right-rotate $(x_i)$ by $c_j$
    $(x_i) \leftarrow$ RefreshMasks($(x_i)$)
    Right-rotate $(m_i)$ by $c_j$
    $(m_i) \leftarrow$ RefreshMasks($(m_i)$)
len $\leftarrow 1$
while len $\leq 32$ do
    $(m_i) \leftarrow (m_i \oplus (m_i \gg \text{len}))$
    len $\leftarrow$ len $\ll 1$
$(y_i) \leftarrow$ SecAnd($(x_i), (m_i)$)
$(z_i) \leftarrow (y_i \oplus x_i \oplus y_i^{(1)})$
$(b_i) \leftarrow$ SecNonzero($(z_i)$)
$(z_i) \leftarrow (y_i^{[64:2]} \vee b_i)$
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

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

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

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

```
$e_1 \leftarrow e_1 - 63$
for $j = 5$ to $0$ do
    $(t_i) \leftarrow (x_i \oplus (x_i \ll 2^j))$
    $(n_i) \leftarrow (x_i \gg (64 - 2^j))$
    $(b_i) \leftarrow \mathsf{SecNonzero}((n_i))$
    $(b'_i) \leftarrow (-b_i)$
    $(t_i) \leftarrow \mathsf{SecAnd}((t_i), (\neg b'_1, b'_2, \cdots, b'_n))$
    $(x_i) \leftarrow (x_i \oplus t_i)$
    $(b_i) \leftarrow \mathsf{B2A_{Bit}}((b_i))$
    $(e_i) \leftarrow (e_i + (b_i \ll j))$
return $(x_i), (e_i)$
```

Wrapping-up

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

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

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

Probing Model

To theoretically evaluate the security of our design, 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 t-probing model, $n \geq t + 1$, and any share cannot be combined with each other.
- It is complicated to prove t-probing security for a large composition of small gadgets. The concept of non-interference is convenient in this case.

Non-Interference Security

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

A gadget is t-Non-Interference (t-NI) secure if every set of t intermediate values can be simulated by no more than t shares of each of its inputs.

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

A gadget is t-Strong-Non-Interference (t-SNI) secure if for every set of t_I internal intermediate values and t_O of its output shares with $t_I + t_O \leq t$, they can be simulated by no more than t_I shares of each of its inputs.

Non-Interference Security

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

Moreover,

- 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.

All the gadgets/algorithms in our paper are proven either t-NI or t-SNI secure.

Gadgets/Algorithms 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$_{\text{Bit}}$	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)

Probing model validates the security theoretically.
In practice, the Test Vector Leakage Assessment (TVLA) methodology [GJR+11] can be applied.
A tester records two sets of traces where

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

The Welch's t-test is then applied on the two sets.
By convention, we consider the leakage is significant if the t-value exceeds ± 4.5.
For traces with a large number of points, we refer to [Din+17] alter this threshold to avoid false positives.

Experiment Setup

We implement our algorithms in the following setting:

- Plain-C code
- Compiled by `arm-none-eabi-gcc 10.3.1`
- Using ChipWhisperer with target board STM32F303 with an ARM Cortex-M4 MCU
- We compare the result with the reference implementation of the NIST Round-3 Submission of FALCON [Pre+20].

TVLA

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

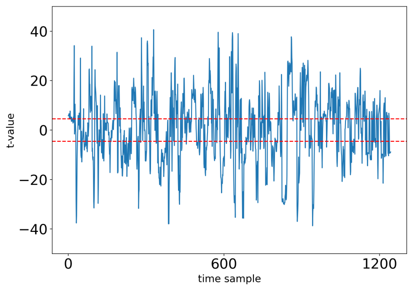

Figure: 1,000 traces, unmasked FprMul

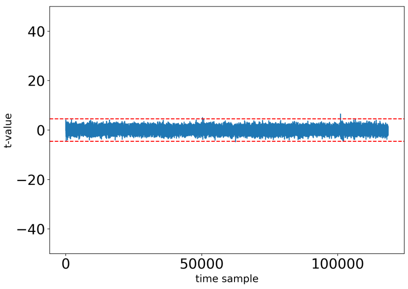

Figure: 10,000 traces, 2-shared SecFprMul

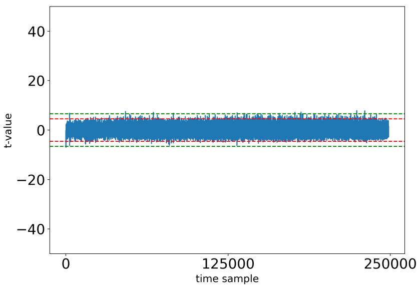

Figure: 100,000 traces, 3-shared SecFprMul

TVLA

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

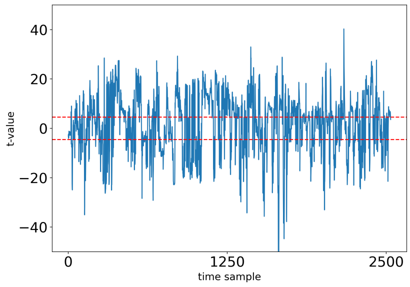

Figure: 1,000 traces, unmasked FprAdd

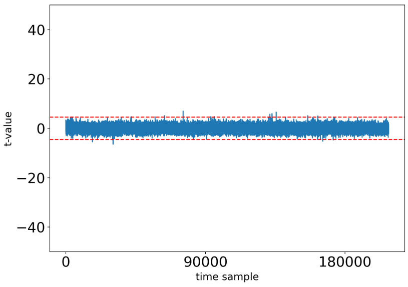

Figure: 10,000 traces, 2-shared SecFprAdd

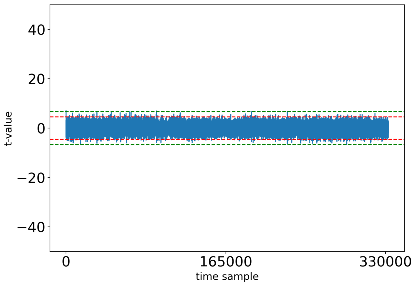

Figure: 100,000 traces, 3-shared SecFprAdd

Performance Evaluation on ARM Cortex-M4

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

Performance Evaluation on General Purpose CPU

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

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.

Conclusion

In this paper,

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

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.

[Ber+10] Guido Bertoni et al. "Building power analysis resistant implementations of Keccak". In: *Second SHA-3 candidate conference*. Vol. 142. Citeseer. 2010.

[GJR+11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. "A testing methodology for side-channel resistance validation". In: *NIST non-invasive attack testing workshop*. Vol. 7. 2011, pp. 115–136.

[Bel+13] Sonia Belaïd et al. "Differential power analysis of HMAC SHA-2 in the Hamming weight model". In: *2013 International Conference on Security and Cryptography (SECRYPT)*. 2013, pp. 1–12.

[Cor+15] Jean-Sébastien Coron et al. "Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity". In: *FSE 2015*. Ed. by Gregor Leander. Vol. 9054. LNCS. Springer, Heidelberg, Mar. 2015, pp. 130–149. DOI: 10.1007/978-3-662-48116-5_7.

[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.

Reference II

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

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

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

[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.

[Mig+19] Vincent Migliore et al. "Masking Dilithium - Efficient Implementation and Side-Channel Evaluation". In: *ACNS 19*. Ed. by Robert H. Deng et al. Vol. 11464. LNCS. Springer, Heidelberg, June 2019, pp. 344–362. DOI: 10.1007/978-3-030-21568-2_17.

Reference III

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

[How+20] James Howe et al. "Isochronous Gaussian Sampling: From Inception to Implementation". In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Springer, Heidelberg, 2020, pp. 53–71. DOI: 10.1007/978-3-030-44223-1_5.

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

[Bos+21] Joppe W. Bos et al. "Masking Kyber: First- and Higher-Order Implementations". In: *IACR TCHES* 2021.4 (2021). https://tches.iacr.org/index.php/TCHES/article/view/9064, pp. 173–214. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.173-214.

[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.

[Fri+22] Tim Fritzmann et al. "Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography". In: *IACR TCHES* 2022.1 (2022), pp. 414–460. DOI: 10.46586/tches.v2022.i1.414-460.

Reference IV

[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.

[Hei+22] Daniel Heinz et al. *First-Order Masked Kyber on ARM Cortex-M4*. Cryptology ePrint Archive, Report 2022/058. https://eprint.iacr.org/2022/058. 2022.

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

[Inc] NewAE Technology Inc. *ChipWhisperer-Pro (Complete Level 3 Starter Kit)*. https://store.newae.com/chipwhisperer-pro-complete-level-3-starter-kit/. URL: https://store.newae.com/chipwhisperer-pro-complete-level-3-starter-kit/.

Floating-Point Number Packing and Rounding

FPR

```
Input: Sign bit $s$, exponent $e$, and 55-bit mantissa $z$
Output: FPN $x$ packed by $s, e, z$
$e \leftarrow e + 1076$
$b \leftarrow [\![ e < 0 ]\!]$
$z \leftarrow z \wedge (b - 1)$
$b \leftarrow [\![ z \neq 0 ]\!]$
$e \leftarrow e \wedge (-b)$
$x \leftarrow ((s \ll 63) \vee (z \gg 2)) + e \ll 52$
$f \leftarrow \texttt{0XC8} \gg z^{[3:1]}$
$x \leftarrow x + f^{(1)}$ {increment if $z^{[3:1]}$ is 011,110 or 111}
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 $y$
$s \leftarrow sx \oplus sy$
$e \leftarrow ex + ey - 2100$
$z \leftarrow mx \times my$
$b \leftarrow [\![ z^{[50:1]} \neq 0 ]\!]$
$z \leftarrow z^{[106:51]} \vee b$
$z' \leftarrow (z \gg 1) \vee z^{(1)}$
$w \leftarrow z^{(106)}$
$z \leftarrow z \oplus (z \oplus z') \wedge (-w)$
$e \leftarrow e + w$
$bx \leftarrow [\![ ex \neq 0 ]\!]$, $by \leftarrow [\![ ey \neq 0 ]\!]$
$b \leftarrow bx \wedge by$
$z \leftarrow z \wedge (-b)$
return FPR$(s, e, z)$
```

Floating-Point Number Addition

FprAdd

Input: FPNs x and y

Output: FPN sum of x and y

```
$d \leftarrow x^{[63:1]} - y^{[63:1]}$
$cs \leftarrow d^{(64)} \vee ((1 - (-d)^{(64)}) \wedge x^{(64)})$
$m \leftarrow (x \oplus y) \wedge (-cs)$
$x \leftarrow x \oplus m, y \leftarrow y \oplus m$
Extract $(sx, ex, mx)$ and $(sy, ey, my)$ from $x, y$, respectively.
$mx \leftarrow mx \ll 3, my \leftarrow my \ll 3$
$ex \leftarrow ex - 1078, ey \leftarrow ey - 1078$
$c \leftarrow ex - ey$
$b \leftarrow [\![c < 60]\!]$
$my \leftarrow my \wedge (-b)$
$my \leftarrow (my \gg c) \vee [\![my^{[c:1]} \neq 0]\!]$
$s \leftarrow sx \oplus sy$
$z \leftarrow mx + (-1)^s my$
Normalize $z, ex$ to make the 64th bit of $z$ set
$z \leftarrow (z \gg 9) \vee [\![z^{[9:1]} \neq 0]\!]$
$ex \leftarrow ex + 9$
return FPR$(sx, ex, z)$
```

SecFPR: Secure FPR

SecFPR

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

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

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

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

```
$e_1 \leftarrow e_1 + 1076$
$(e_i) \leftarrow \mathsf{A2B}((e_i))$
$(b_i) \leftarrow (-e_i^{(16)})$
$(z_i) \leftarrow \mathsf{SecAnd}((z_i), (\neg b_1, b_2, \cdots, b_n))$
$(e_i) \leftarrow \mathsf{SecAnd}((e_i), (-z_i^{(55)}))$
$(e_i) \leftarrow \mathsf{SecAdd}((e_i), (z_i^{(55)}))$
$(e_i) \leftarrow \mathsf{Refresh}((e_i))$
$(s_i) \leftarrow \mathsf{Refresh}((s_i))$
$(x_i) \leftarrow ((s_i^{(1)} \ll 63) \vee (e_i^{[11:1]} \ll 52) \vee (z_i^{[54:3]})$
$(f_i) \leftarrow \mathsf{SecOr}(\mathsf{Refresh}(z_i^{(1)}), (z_i^{(3)}))$
$(f_i) \leftarrow \mathsf{SecAnd}((f_i), (z_i^{(2)}))$
$(x_i) \leftarrow \mathsf{SecAdd}((x_i), (f_i))$
return $(x_i)$
```

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.
$(s_i) \leftarrow (sx_i \oplus sy_i)$
$(e_i) \leftarrow (ex_1 + ey_1 - 2100, ex_2 + ey_2, \cdots)$
$(p_i) \leftarrow \mathsf{SecMult}((mx_i), (my_i))$
$(p_i) \leftarrow \mathsf{A2B}((p_i))$
$(b_i) \leftarrow \mathsf{SecNonzero}((p_i^{[51:1]}))$
$(z_i) \leftarrow (p_i^{[105:51]})$
$(z_i') \leftarrow (p_i^{[105:51]} \oplus p_i^{[106:52]})$
$(w_i) \leftarrow (p_i^{(106)})$
$(z_i') \leftarrow \mathsf{SecAnd}((z_i'), \mathsf{Refresh}((-w_i)))$
$(z_i) \leftarrow (z_i' \oplus z_i)$
$(z_i) \leftarrow \mathsf{SecOr}((z_i), (b_i))$
$(w_i) \leftarrow \mathsf{B2A_{Bit}}((w_i))$
$(e_i) \leftarrow (e_i + w_i)$
$(bx_i) \leftarrow \mathsf{SecNonzero}((ex_i))$
$(by_i) \leftarrow \mathsf{SecNonzero}((ey_i))$
$(d_i) \leftarrow \mathsf{SecAnd}((bx_i), (by_i))$
$(z_i) \leftarrow \mathsf{SecAnd}((z_i), (-d_i^{(1)}))$
return $\mathsf{SecFPR}((s_i), (e_i), (z_i))$
```

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
$(xm_i) \leftarrow (x_i^{[63:1]})$
$(ym_i) \leftarrow (\neg y_1^{[63:1]}, y_2^{[63:1]}, \cdots, y_n^{[63:1]})$
$(d_i) \leftarrow \mathsf{SecAdd}((xm_i), (ym_i))$
$(b_i) \leftarrow \mathsf{SecNonzero}(\neg d_1, d_2, \cdots, d_n)$
$(b_i') \leftarrow \mathsf{SecNonzero}(\neg(d_1 \oplus (1 \ll 63)), d_2, \cdots, d_n)$
$(cs_i) \leftarrow \mathsf{SecAnd}((\neg b_1, b_2, \cdots, b_n), (x_i^{(64)}))$
$(cs_i) \leftarrow \mathsf{SecOr}((cs_i), (d_i^{(64)} \oplus b_i \oplus b_i'))$
$(m_i) \leftarrow \mathsf{SecAnd}((x_i \oplus y_i), (-cs_i))$
$(x_i) \leftarrow (x_i \oplus m_i)$, $(y_i) \leftarrow (y_i \oplus m_i)$
Extract $(sx_i), (ex_i), (mx_i)$ and $(sy_i), (ey_i), (my_i)$ from
    $(x_i)$ and $(y_i)$, respectively.
$(mx_i) \leftarrow (mx_i \ll 3)$, $(my_i) \leftarrow (my_i \ll 3)$
$(ex_i) \leftarrow \mathsf{B2A}((ex_i))$, $(ey_i) \leftarrow \mathsf{B2A}((ey_i))$
$ex_1 \leftarrow ex_1 - 1078$, $ey_1 \leftarrow ey_1 - 1078$.
$(c_i) \leftarrow (ex_i - ey_i)$
$(c_i') \leftarrow \mathsf{A2B}((c_1 - 60, c_2, \cdots, c_n))$
$(my_i) \leftarrow \mathsf{SecAnd}((my_i), (-(c_i'^{(16)})))$
$(my_i) \leftarrow \mathsf{SecFprUrsh}((my_i), (c_i^{[6:1]}))$
$(my_i') \leftarrow (\neg my_1, my_2, \cdots, my_n)$
$(my_i') \leftarrow \mathsf{SecAdd}((my_i'), (1, 0, \cdots, 0))$
$(s_i) \leftarrow (-(sx_i \oplus sy_i))$
$(my_i) \leftarrow \mathsf{Refresh}((my_i))$
$(my_i') \leftarrow \mathsf{SecAnd}((my_i \oplus my_i'), (s_i))$
$(my_i) \leftarrow (my_i \oplus my_i')$
$(z_i) \leftarrow \mathsf{SecAdd}((mx_i), (my_i))$
$(z_i), (ex_i) \leftarrow \mathsf{SecFprNorm64}((z_i), (ex_i))$
$(b_i) \leftarrow \mathsf{SecNonzero}((z_i^{[10:1]}))$
$(z_i) \leftarrow (z_i \gg 9)$
$(z_i^{(1)}) \leftarrow (b_i)$
$ex_1 \leftarrow ex_1 + 9$
return $\mathsf{SecFPR}(\mathsf{Refresh}((sx_i)), (ex_i), (z_i))$
```