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

Abstract.
In this paper, we provide the first masking scheme for floating-point number multiplication and addition to defend against recent side-channel attacks on Falcon's pre-image vector computation. Our approach involves a masked nonzero check gadget that securely identifies whether a shared value is zero. This gadget can be utilized for various computations such as rounding the mantissa, computing the sticky bit, checking the equality of two values, and normalizing a number. To support the masked floating-point number addition, we also developed a masked shift and a masked normalization gadget. Our masking design provides both first- and higher-order mask protection, and we demonstrate the theoretical security by proving the (Strong)-Non-Interference properties in the probing model. To evaluate the performance of our approach, we implemented unmasked, first-order, and second-order algorithms on an Arm Cortex-M4 processor, providing cycle counts and the number of random bytes used. We also report the time for one complete signing process with our countermeasure on an Intel-Core CPU. In addition, we assessed the practical security of our approach by conducting the test vector leakage assessment (TVLA) to validate the effectiveness of our protection. Specifically, our TVLA experiment results for second-order masking passed the test in 100,000 measured traces.

Keywords: Falcon, Floating-Point Arithmetic, Masking, Post-Quantum Cryptography, Side-Channel Analysis

1 Introduction

The rapid development of quantum computers has posed a potential threat to public key cryptography systems. Due to Shor's algorithm [Sho97], public key cryptographic schemes based on the hardness of integer factorization and discrete logarithm, including RSA [RSA78], Diffie-Hellman key agreement [DH76], ElGamal encryption [Elg85], and ECDSA [JMV01], are vulnerable to large-scale quantum computing. To defend against such a menace, post-quantum cryptography, which studies algorithms that are considered secure against quantum computing, has been widely researched. In 2016, the National Institute of Standards and Technology (NIST) initiated a standardization process for post-quantum cryptography [oSTa]. Recently, the four selected algorithms, CRYSTALS-Kyber, CRYSTALS-Dilithium, SPHINCS$^+$, and Falcon became part of NIST's post-quantum cryptographic standards, which are expected to be finalized in about two years [oSTb].

Unfortunately, even with conjectured quantum resistance, cryptographic implementations may not be secure. Side-channel analysis considers the threat of information leakage from an electronic device running cryptographic algorithms through its hardware physical behaviors. These could be the running time of different inputs or the electromagnetic emanation and power consumption during the execution. Many works have been

Received: 2023-10-15 Accepted: 2023-12-15 Published: 2024-03-12

done on implementing such attacks against post-quantum algorithms. Therefore, some side-channel resistance in post-quantum cryptography has been proposed in recent years. For example, studies including [BGR+21, FBR+22, HKL+22] provided countermeasures for CRYSTALS-Kyber, and countermeasures for CRYSTALS-Dilithium was presented in [MGTF19]. However, the side-channel resistance of Falcon has lacked discussion.

Related Work The signing process of Falcon faces at least two side-channel vulnerabilities: the Gaussian sampler and the pre-image vector computation. The Gaussian sampler returns numbers following the discrete Gaussian distribution, and previous works [BHLY16, EFGT17, PBY17, MHS+19] have demonstrated the threat of timing attacks on the sampler. An isochronous design as a countermeasure was developed in [HPRR20]. Besides, a masked implementation of the Gaussian sampler is provided in [EFG+22]. Recently, Guerreau et al. [GMRR22] proposed a simple power analysis attack on Falcon and a related light countermeasure, and their work was further improved in [ZLYW23], which provides more potential sources of leakage and reduces the number of required traces by new algorithms.

The pre-image vector computation in Falcon is used for finding the short vector solution as the signature. In [KA21], Karabulut and Aysu first performed an EM attack on such computation in Falcon, whose method in their setting can recover the secret key with a few thousand measured traces. Their attack was later improved by Guerreau et al. [GMRR22], which reduces the guess space complexity from 2^{27} to only 2^{11}. However, the protection of the pre-image vector computation in Falcon has so far seldom been discussed. At the same time, the authors of Falcon [PFH+20] suggest that the masking scheme should be considered to protect Falcon from side-channel attacks.

Our 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. To support functions for our masked design, we devise masked gadgets for the nonzero checks, unsigned right-shift, and 64-bit normalization. We provide the details of our algorithms and show that our approach can be extended to high-order protections.

- We verify the high-order security of our design in the probing model by the concepts of Non-Interference security [BBD+16]. We offer formal proofs via simulation to show they can resist high-order probing attacks.

- To test the practical leakage of our work, we implemented our algorithms on ChipWhisperer-Pro [Inc] with STM32F415 target. We conducted the Test Vector Leakage Assessment (TVLA) [GGJR+11] experiments using 100,000 measured traces to demonstrate the effectiveness of our countermeasure.

- We tested the performance of our work on an Arm Cortex-M4 core and Intel-Core i9-12900KF CPU. The evaluation results are presented for our first- and second-order implementations and compared with the original unmasked design.

Outline We organize our work as follows. In Section 2, we give notations throughout the paper and review some contexts for our work. We introduce in Section 3 our masking algorithms of floating-point number multiplication and addition in detail. The security proofs in the probing model of our design are given in Section 4. The performance evaluation of our implementation and security assessment results of TVLA are given in Section 5.

2 Preliminaries

2.1 Notation

In the rest of the paper, $N = 2^\kappa$ for some integer κ, $M > N$, q is a prime number, and $\phi = x^N + 1$ is the cyclotomic polynomial. For a polynomial $f = \sum_{i=0}^{N-1} f_i x^i$, its adjoint is $f^* = f_0 - \sum_{i=1}^{N-1} f_i x^{N-i}$. We write a vector $\mathbf{v}$ in bold, and a matrix $\mathbf{A}$ is written in bold and uppercase. The adjoint of a vector $\mathbf{v}$ or a matrix $\mathbf{A}$, which is the transpose of adjoint of each of its coefficient, is written as $\mathbf{v}^*$ or $\mathbf{A}^*$. For a polynomial f modulo ϕ, we may write it as an N-by-N matrix, with the ith row representing the coefficients of $(x^i f \bmod \phi)$ for $0 \leq i \leq N-1$. Matrix additions and multiplications will map to polynomial additions and multiplications in the ring of polynomials modulo ϕ in this tradition.

For a variable x, the jth bit of x is written as $x^{(j)}$. The LSB is the first bit, and the MSB is the kth bit if it is stored in a k-bit register. 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) if the sequence length is not our point, or it is obvious from contexts. Without specifying, n will be the sequence length of (x_i).

We use notations $\oplus$, $\wedge$, $\vee$ to denote the bit-wise XOR, AND and OR operations, respectively. For variable x, we denote by $\neg x$ the bit inversion of x. Also, for a negative integer in a register, we consider two's complement representation and write $-x = (\neg x) + 1$. The $\ll$ and $\gg$ are unsigned left- and right-shift of a variable. In addition, for a proposition P, we let $[\![P]\!] = 1$ if P is true and 0 if otherwise.

2.2 Falcon Signature Scheme

In this section, we briefly review the Falcon signature scheme, emphasizing on the pre-image vector computation in the Fourier domain which is strongly related to our work. For more details, we refer the readers to the NIST round-3 submission of Falcon [PFH+20].

The basis of Falcon is the GPV framework [GPV08]. At a high level, the framework uses a full-rank matrix $\mathbf{A} \in \mathbb{Z}_q^{N \times M}$ as public key, and $\mathbf{B} \in \mathbb{Z}_q^{M \times M}$ as secret key where $\mathbf{B}\mathbf{A}^T = 0 \bmod q$. To sign a message m, one first computes $H(\mathsf{m})$ for some hash function $H : \{0,1\}^* \to \mathbb{Z}_q^N$, and the signature is a short vector $\mathbf{s}$, which is derived by trapdoor $\mathbf{B}$, that satisfies $\mathbf{s}\mathbf{A}^T = H(\mathsf{m})$. To verify, one can simply check that the received $\mathbf{s}$ is short and satisfies the above equation.

Falcon instantiates the GPV framework on NTRU lattices. The secret key is essentially 4 polynomials f, g, F, G in $\mathbb{Z}_q[x]/(\phi)$ which satisfy the NTRU equation

$$fG - gF = q \bmod \phi$$

while the public key is now the polynomial $h = gf^{-1} \bmod (\phi, q)$. To sign a message securely, coefficients of f and g are sampled from a discrete Gaussian distribution with a small standard deviation $\sigma_{\{f,g\}} = 1.17\sqrt{q/2N}$, and F, G are generated from f, g by solving the NTRU equation. Matrices $\mathbf{A}, \mathbf{B}$ are formed as

$$\mathbf{A} = \left[\begin{array}{c|c} 1 & h^* \end{array}\right], \quad \mathbf{B} = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$$

The second part of Falcon key generation is the computation of the Falcon tree, which will be used in looking for the short vector in signing. In short, the Falcon tree is a binary tree where each node at height κ is a polynomial in $\mathbb{Q}[x]/(x^{2^\kappa} + 1)$. The Gram matrix $\mathbf{G} = \mathbf{B}\mathbf{B}^*$ is first comptuted, and a Falcon tree is built by recursively calling the LDL decomposition on each element of the diagonal matrix and store the lower-triangular matrix as a node value. The leaf node is then normalized by a constant σ. The whole key generation process is given in Algorithm 1.

Algorithm 1 FALCON.Keygen from [PFH+20]

Input: Polynomial ϕ, prime modulus q
Output: FALCON secret key sk, public key pk

$f, g, F, G \leftarrow \mathsf{NTRUGen}(\phi, q)$
$\hat{\mathbf{B}} \leftarrow \left[\begin{array}{c|c} \mathsf{FFT}(g) & \mathsf{FFT}(-f) \\ \hline \mathsf{FFT}(G) & \mathsf{FFT}(-F) \end{array}\right]$
$\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^*$
$\mathsf{T} \leftarrow \mathsf{ffLDL}(\mathbf{G})$ ▷ generate FALCON tree
for each leaf leaf of T **do**
    $\mathsf{leaf.value} \leftarrow \sigma/\sqrt{\mathsf{leaf.value}}$
$\mathsf{sk} \leftarrow (\hat{\mathbf{B}}, \mathsf{T})$
$h \leftarrow gf^{-1} \bmod q$
$\mathsf{pk} \leftarrow h$
return sk, pk

The signing process of FALCON in Algorithm 2 starts by sampling a random salt r and computing $c = H(\mathsf{r}\|\mathsf{m})$. Next it finds a short vector $\mathbf{s} = (s_1, s_2)$ such that $\mathbf{s}\mathbf{A}^* = s_1 + s_2 h = c$. To do so, the vector $\mathbf{t} = (c, 0) \times \mathbf{B}^{-1}$ is first computed, which we call the *pre-image vector*. Then the *fast Fourier nearest plane* algorithm [DP16] ($\mathsf{ffSampling}$ in Algorithm 2) is applied to find some $\mathbf{z}$ where $\mathbf{s} = (\mathbf{t} - \mathbf{z})\mathbf{B}$ is close to zero in a secure way, in the sense that no information about secret basis $\mathbf{B}$ is leaked. The signature consists of the salt $\mathbf{r}$ and the second coefficient s_2. The verification in Algorithm 3 is relatively easy. One first retrieves s_1 by $H(\mathsf{r}\|\mathsf{m}) - s_2 h$, and then check whether the norm of vector (s_1, s_2) is small.

Algorithm 2 FALCON.Sign from [PFH+20]

Input: Message m, secret key sk and a bound $\lfloor \beta^2 \rfloor$
Output: Signature sig of m

$\mathsf{r} \xleftarrow{\$} \{0,1\}^{320}$
$c \leftarrow H(\mathsf{r}\|\mathsf{m})$
$\mathbf{t} \leftarrow \left(-\frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(F), \frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(f)\right)$ ▷ pre-image vector computation
do
    do
        $\mathbf{z} \leftarrow \mathsf{ffSampling}(\mathbf{t}, \mathsf{T})$
        $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
    while $\|\mathbf{s}\|^2 > \lfloor \beta^2 \rfloor$
    $(s_1, s_2) \leftarrow \mathsf{invFFT}(\mathbf{s})$
    $\mathsf{s} \leftarrow \mathsf{Compress}(s_2)$
while $\mathsf{s} = \perp$
$\mathsf{sig} \leftarrow (\mathsf{r}, \mathsf{s})$
return sig

To use the fast Fourier sampler and increase the speed in the signing process, FALCON applies the fast Fourier transform on polynomials. Let Ω_ϕ be the sets of all complex roots of ϕ, the fast Fourier transform of a polynomial f is

$$\mathsf{FFT}(f) = (f(\xi))_{\xi \in \Omega_\phi}, \quad \Omega_\phi = \left\{ \exp\left(\frac{i\pi(2k+1)}{N}\right) \mid 0 \le k < N \right\}$$

In this representation, polynomial additions and multiplications can be done by performing

Algorithm 3 FALCON.Verify from [PFH+20]

Input: Message m, signature $\mathsf{sig} = (\mathsf{r}, \mathsf{s})$, public key pk, and a bound $\lfloor \beta^2 \rfloor$
Output: Accept or reject

$c \leftarrow H(\mathsf{r} \| \mathsf{m})$
$s_2 \leftarrow \mathsf{Decompress}(\mathsf{s})$
if $s_2 = \perp$ **then**
    reject
$s_1 \leftarrow c - s_2 h \bmod q$
if $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ **then**
    accept
else
    reject

operations on each coefficient, which is a complex number. Therefore, the pre-image vector computation is composed of complex number multiplication of each coefficient.

In practice, a complex number $a + bi$ is represented by two floating-point numbers a, b as its real and imaginary parts. As pointed out in their document [PFH+20], it may be hard to implement Falcon on constrained devices without a floating-point unit. Falcon thus provides an emulation of floating-point operations with 53 bits of precision in their reference implementation. They follow the IEEE-754 standard to store a 64-bit floating-point number by 1-bit sign, 11-bit exponent, and 53-bit mantissa, where the 53rd bit is always 1 and omitted when storing. To make the comparison of values more straightforward, the exponent is biased with a value 1023 to make it unsigned. For convenience, we will represent a 64-bit floating-point number x as a tuple (s, e, m) where

$$x = (-1)^s \cdot 2^{e-1023} \cdot (m \cdot 2^{-52})$$

Compared to the floating-point number architecture, s is the 64th bit, e is the 53rd to 63rd bit, and m is the 1st to 52nd bit of the floating-point number with the omitted one at the beginning. In this representation, m may be viewed as an integer in $[2^{52}, 2^{53})$.

2.3 Floating-Point Number Multiplication and Addition

The pre-image vector computation, or coefficient-wise complex number multiplication, is performed by combinations of floating-point number multiplications and additions. We here briefly introduce how Falcon emulates them in their given reference implementation.

We begin by introducing the last subroutine of both the multiplication and addition — the floating-point number rounding and packing function FPR (Algorithm 4), which receives a 55-bit mantissa, an unbiased exponent, and a sign bit and outputs the floating-point number with a rounded mantissa and biased exponent. It starts by adding the exponent with a constant 1076 for the bias (1023) and mantissa size (53). If the result is smaller than 0, it is considered subnormal, and the mantissa should be turned to zero. Then it zeros the exponent if the mantissa is zero. Next, the sign bit and mantissa are combined together and added with the exponent. Note the exponent will be incremented by the top bit of the mantissa. Finally, the rounding is done by checking the least three significant bits of the mantissa. It follows the round-to-nearest strategy: if they are 011, 110, or 111, a carry 1 is added.

Now we introduce the floating-point number multiplication in Algorithm 5. The multiplication first XORs the sign bits and sums the exponents of both operands, and then does the mantissa multiplication. Note that a constant -2100 is added to the exponent for exponent bias (1023×2), scaling of mantissa (52×2), and shifting of the later multiplication product (-50). The product is a 106-bit raw mantissa, and rounding is performed to

Algorithm 4 FPR

Input: Sign bit s, exponent e, and 55-bit mantissa z
Output: Floating-Point number 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

reduce the mantissa to 55 bits in which the 55th bit is set. Notice that we need to make the sticky bit preserved after the rounding. If either of the exponents is zero, the computations above are invalid, and the mantissa is turned to zero. Finally, the sign bit, unbiased exponent, and 55-bit mantissa are packed into one floating-point number by the FPR in Algorithm 4.

Algorithm 5 FprMul

Input: Floating-Point numbers $x = (sx, ex, mx)$ and $y = (sy, ey, my)$
Output: Floating-Point numbers product of x and y

$s \leftarrow sx \oplus sy$
$e \leftarrow ex + ey - 2100$
$z \leftarrow mx \times my$
$b \leftarrow [\![z^{[50:1]} \neq 0]\!]$
$z \leftarrow z^{[106:51]} \vee b$
$z' \leftarrow (z \gg 1) \vee z^{(1)}$
$w \leftarrow z^{(106)}$
$z \leftarrow z \oplus (z \oplus z') \wedge (-w)$ ▷ round to 55 bits with sticky bit preserved
$e \leftarrow e + w$ ▷ increment if the product carries to 106th bit
$bx \leftarrow [\![ex \neq 0]\!]$, $by \leftarrow [\![ey \neq 0]\!]$
$b \leftarrow bx \wedge by$
$z \leftarrow z \wedge (-b)$
return $\mathsf{FPR}(s, e, z)$

The floating-point number addition in Algorithm 6 first exchanges the operands to make the absolute value of the first one no less than that of the other. Since the exponent is biased, one can compare the absolute values of two floating-point numbers by comparing their least 63 bits. If both operands are only differed by the sign, it lets the first operand be the positive one. Then it extracts both operands' sign bit, exponent, and mantissa, where the mantissa is scaled up to 55 bits for further rounding. The exponent is subtracted by 1078 for the bias (1023) and mantissa scaling (55). Next, it shifts the second operand according to the exponent difference while preserving the sticky bit and adds both mantissa. The sum is first normalized to $[2^{63}, 2^{64})$; that is, the 64th bit is set and then scaled down to $[2^{54}, 2^{55})$ with the sticky bit preserved. Finally, the packing helps to return a 64-bit floating-point number with a 53-bit mantissa, as it does in multiplication.

It should be noticed that the floating-point number multiplication and addition do not follow the associative law and distributive law. In other words, for some floating-point numbers a, b, and c,

$$a + (b + c) \neq (a + b) + c \quad \text{or} \quad a \times (b + c) \neq a \times b + a \times c$$

Algorithm 6 FprAdd

Input: Floating-Point numbers x and y
Output: Floating-Point numbers sum of x and y

$d \leftarrow x^{[63:1]} - y^{[63:1]}$
$cs \leftarrow d^{(64)} \vee ((1 - (-d)^{(64)}) \wedge x^{(64)})$ ▷ $[\![|x| < |y|]\!] \vee ([\![|x| \leq |y|]\!] \wedge [\![x^{(64)} = 1]\!])$
$m \leftarrow (x \oplus y) \wedge (-cs)$
$x \leftarrow x \oplus m, y \leftarrow y \oplus m$ ▷ swap x and y if necessary
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)

This makes it complicated to design a masked implementation of Falcon without constructing new ways to do the multiplication and addition. In this paper, we follow Falcon reference implementation and rewrite the floating-point number multiplication and addition in a masked way. Our masked functions return the same values as the existing functions, and they can apply to both security levels provided by Falcon (i.e., Falcon-512 and Falcon-1024) since they use the same floating-point number arithmetic.

2.4 Masking

Side-channel attacks can extract secret information in cryptographic devices by measuring their physical behavior during the computation, and masking helps avoid leakage by randomizing secret variables in each operation. Essentially, it splits each sensitive value into several shares that are randomized every round. In this way, the attacker cannot gain any information if only a limited number of intermediate variables are seen.

Common masking methods include Boolean mask and arithmetic mask [MOP07]. The Boolean masking method hides a variable x by a length n sequence $(x_1, x_2, \cdots, x_n)$ where $x = x_1 \oplus \cdots \oplus x_n$. Since the value x should only be recovered if all x_i's are known to the attacker, we also call the sequence (x_i) Boolean shares of x. The arithmetic masking method uses arithmetic shares (x_i) where $x = x_1 + \cdots + x_n$. Note the addition is considered to be modulo 2^k for a k-bit variable.

To theoretically evaluate the power of the attacker and the security level protected by masking, Ishai, Sahai, and Wagner [ISW03] introduced the notion of the t-probing model, which assumes an adversary can probe up to t intermediate values during the cryptographic operations. A gadget is said to be secure against t-order attacks if any t intermediate values in operations leak no information about the hidden secret. Their approach for proving this security is based on simulation; namely, to show that any t intermediate values of the gadget can be simulated without the knowledge of the secret.

To prove the security of compositions of gadgets, the concept of (Strong)-Non-Interference was proposed in [BBD+16]. We recall them in the version presented in [SPOG19].

Definition 1 (t-Non-Interference (t-NI) security)**.** 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.

Definition 2 (t-Strong-Non-Interference (t-SNI) security)**.** 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.

If a gadget is itself t-NI secure, and if any set of t shares of its input is independent of the secret, then it can resist t-order attacks. However, compositions of t-NI secure gadgets may not be t-NI secure, and we can use t-SNI to avoid this problem. Note that a t-SNI secure gadget is, by definition, also t-NI, and it can help the simulation since it requires only the same number of input shares as the internal probed values to simulate all the probes.

In section 4, we will show that our algorithms with $n = t + 1$ shares are t-NI or t-SNI secure and provide formal proofs via simulation.

2.5 Test Vector Leakage Assessment

The Test Vector Leakage Assessment (TVLA) methodology [GGJR+11] is used to analyze whether a cryptographic device leaks information from its power/EM trace. The theory behind this is Welch's t-test. Here we introduce the non-specific version with *fixed-versus-random* policy. The tester records a set of power traces where the device runs with a fixed input and another set of traces where the device runs with random inputs. In implementation, the tester records traces for different sets in random order to avoid possible device bias through time. For each point of trace, the t statistic

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

is calculated, where $\bar{X}_f, s_f, n_f$ and $\bar{X}_r, s_r, n_r$ are the sample mean, standard deviation, and number of traces of set *fixed* and *random*, respectively. When the number of recorded traces is large, the t statistic helps recognize whether both sets are sampled from distributions of the same population mean, which is our null hypothesis. In our contexts, this implies adversaries cannot distinguish some particular input. In practice, we reject the hypothesis if the t statistic exceeds the standard threshold ± 4.5, which is set to guarantee a p-value under 0.00001. However, since our measured traces contain many sampled points, we refer to [DZD+17] and alter this threshold to avoid false positives.

3 Masked Floating-Point Number Multiplication and Addition

We now introduce our main algorithms — floating-point number multiplication and addition in Falcon, which rewrites FprMul in Algorithm 5 and FprAdd in Algorithm 6 in a masked design. To support their complicated operations in shares, we design three masked gadgets as subroutines in our main algorithms, including:

- SecNonzero: the masked algorithm which receives input shares and outputs whether the input is zero in bit shares.
- SecFprUrsh: the masked algorithm which receives input Boolean shares (x_i) of x and arithmetic shares (c_i) of c and returns shares of $x \gg c$ with its sticky bit preserved.

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

Algorithm	Description	Security	Reference
SecAnd	AND of Boolean shares	t-SNI	[ISW03, BBD+16]
SecMult	Multiplication of arithmetic shares	t-SNI	[ISW03, BBD+16]
SecAdd	Addition of Boolean shares	t-NI	[CGTV15, BBE+18]
A2B	Arithmetic to Boolean conversion	t-SNI	[SPOG19]
B2A	Boolean to arithmetic conversion	t-SNI	[BCZ18]
$\mathsf{B2A}_{\mathsf{Bit}}$	One-bit B2A conversion	t-SNI	[SPOG19]
RefreshMasks	t-NI refresh of masks	t-NI	[BBD+16, BCZ18]
Refresh	t-SNI refresh of masks	t-SNI	[BBD+16]
SecOr	OR of Boolean shares	t-SNI	Algorithm 7
SecNonzero	Nonzero check of shares	t-SNI	Algorithm 8
SecFprUrsh	Right-shift with sitcky-bit preserved	t-SNI	Algorithm 9
SecFprNorm64	Normalization to $[2^{63}, 2^{64})$	t-NI	Algorithm 10

- SecFprNorm64: the masked algorithm which left-shifts 64-bit Boolean shares (x_i) to make its 64th bit set. It then adds the shift counts to the other input shares (e_i).

We also use several gadgets from previous works. Table 1 lists all the gadgets used in this work and their t-NI and t-SNI security. For those proposed in this work (Algorithm 7, 8, 9, 10), we provide details of them in the following of this section and will prove their t-NI or t-SNI security in Section 4.

3.1 Masked Nonzero Check

We start by introducing our nonzero-check algorithm. Consider that a k-bit number x is zero if bit-wise OR-ing all its bits results in a zero. That is,

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

Let the input be some Boolean shares. As each bit of shares is independent, we can bitslice the shares and use the SecOr gadget to OR all the bits securely. The detail of the SecOr gadget is given in Algorithm 7, which applies De Morgan's law and calls the AND algorithm SecAnd of shares as a subroutine. That is,

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

To increase efficiency, we consider OR-ing not one bit but half of the register size each time, which reduces the complexity from $\mathcal{O}(n^2 k)$ to $\mathcal{O}(n^2 \log k)$ for n-shared k-bit numbers.

For arithmetic shares input (x_i), since

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

We take the last $\frac{n}{2}$ shares, turn them negative, and use two $\frac{n}{2}$-share arithmetic-to-Boolean conversion gadgets A2B to create two Boolean shares, each representing half shares of the input. Followed by the above nonzero check of Boolean shares, we end up getting one-bit Boolean shares indicating if the input is nonzero. The whole algorithm is given in Algorithm 8.

Algorithm 7 SecOr

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

Algorithm 8 SecNonzero

Input: Shares $(x_i)_{1\le i\le n}$ for value x
Output: One-bit Boolean shares $(b_i)_{1\le i\le n}$ where $\bigoplus_i b_i = 0 \iff x = 0$
1: **if** input (x_i) are arithmetic shares **then**
2: $\quad (t_i)_{1\le i\le \frac{n}{2}} \leftarrow \mathsf{A2B}((x_i)_{1\le i\le \frac{n}{2}})$
3: $\quad (t_i)_{\frac{n}{2}+1\le i\le n} \leftarrow \mathsf{A2B}((-x_i)_{\frac{n}{2}+1\le i\le n})$
4: **else**
5: $\quad (t_i)_{1\le i\le n} \leftarrow (x_i)_{1\le i\le n}$
6: $\mathsf{len} \leftarrow \mathsf{bitsize}/2$
7: **while** $\mathsf{len} \ge 1$ **do**
8: $\quad (l_i) \leftarrow \mathsf{Refresh}((t_i^{[2\mathsf{len}:\mathsf{len}]}), \mathsf{len})$
9: $\quad (r_i) \leftarrow (t_i^{[\mathsf{len}:1]})$
10: $\quad (t_i) \leftarrow \mathsf{SecOr}((l_i), (r_i))$
11: $\quad \mathsf{len} \leftarrow \mathsf{len} \gg 1$
12: **return** $(t_i^{(1)})$

3.2 Masked Unsigned Right-Shift

One step in the floating-point number addition is to right-shift the mantissa of the second operand by the difference of exponents (in line 11 of FprAdd). This is used to make the exponents of the two operands equal, and thus we can directly add their mantissa together. While the exponent difference is part of the secret and is represented in shares, we cannot directly unmask them. The SecFprUrsh in Algorithm 9 right-shifts a Boolean-masked number by a value in arithmetic shares while preserving the sticky bit.

The idea goes as follows. For a 64-bit number, rotating it by some value c is equivalent to rotating by value c mod 64. This shows we can rotate by a 6-bit arithmetic-masked value via sequentially rotating by each share of it. To recover the shifted result from the rotated one, we also rotate a constant $(1 \ll 63)$ by the same value. As there is only a single 1 in bit representation of $(1 \ll 63)$, we could sequentially XOR and right-shift the value to set all the valid bits for our desired shifted result. Moreover, the unset bits are the bits to discard, which determine the sticky bit. We use a SecNonzero to find our desired sticky bit and replace the least significant bit of the shifted result with it.

It is noteworthy that we add t-NI secure RefreshMasks in each iteration of the rotation. This is for removing the dependency between shares to achieve its t-SNI security. A formal proof based on simulation and properties of the RefreshMasks gadget will be given in Section 4.

3.3 Masked 64-bit Normalization

Another crucial part of the floating-point number addition is normalizing a number to range $[2^{63}, 2^{64})$ (in line 14 of FprAdd). This is used to set the correct exponent and left-shift

Algorithm 9 SecFprUrsh

Input: 64-bit Boolean shares $(x_i)_{1\le i\le n}$ for value x, 6-bit arithmetic shares $(c_i)_{1\le i\le n}$ for value c

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)

the mantissa sum for a valid floating-point number. In Falcon reference implementation, they sequentially check whether the high-order bits are all zero and left-shift the mantissa by the corresponding value. We follow their implementation and use our SecNonzero gadget to check the shift counts. In addition, to add the shift counts to the exponent, we use the one-bit Boolean-to-arithmetic conversion algorithm $\mathsf{B2A_{Bit}}$ to transform the result of SecNonzero into arithmetic shares. The whole algorithm is given in Algorithm 10.

Algorithm 10 SecFprNorm64

Input: 64-bit Boolean shares $(x_i)_{1\le i\le n}$ for value x, 16-bit arithmetic shares $(e_i)_{1\le i\le n}$ for value e

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

1: $e_1 \leftarrow e_1 - 63$
2: **for** $j = 5$ to 0 **do**
3: $(t_i) \leftarrow (x_i \oplus (x_i \ll 2^j))$
4: $(n_i) \leftarrow (x_i \gg (64 - 2^j))$
5: $(b_i) \leftarrow \mathsf{SecNonzero}((n_i))$
6: $(b'_i) \leftarrow (-b_i)$
7: $(t_i) \leftarrow \mathsf{SecAnd}((t_i), (\neg b'_1, b'_2, \cdots, b'_n))$
8: $(x_i) \leftarrow (x_i \oplus t_i)$
9: $(b_i) \leftarrow \mathsf{B2A_{Bit}}((b_i))$
10: $(e_i) \leftarrow (e_i + (b_i \ll j))$
11: **return** $(x_i), (e_i)$

3.4 Masked Floating-Point Number Packing

Given 1-bit Boolean-shared sign bit (s_i), 16-bit arithmetic-shared exponent (e_i), and 55-bit Boolean-shared mantissa (z_i), the SecFPR in Algorithm 11 packs them into one Boolean shares and round the mantissa to 53-bit with the round-to-nearest strategy.

Similar to Algorithm 4, the procedure starts by adding a constant 1076 to the exponent. Then we turn the mantissa into zero if the result is smaller than 0 (line 4). The comparison

is made by a 16-bit conversion A2B gadget and checking the most significant bit of the result. Next, we zero the exponent if the mantissa is zero, and we check this by the 55th bit of the mantissa (line 5). The 55th bit is also added to the exponent by the Boolean-masked addition gadget SecAdd. We then pack the sign bit, exponent, and mantissa that is shifted right by 2 bits (line 9). The Refresh gadgets for the sign bit and the exponent are used here to satisfy the t-SNI security. Finally, to do the rounding securely. We consider adding the mantissa by the value derived from OR-ing the first and third bit (line 10) and then AND-ing the second bit (line 11). In this way, the least 3 bits 011, 110, or 111 will cause an increment.

Note that the value will be indeterminate if the input exponent is too large, as it will overflow to the sign bit in Algorithm 4. We omit this check and leave the responsibility for not letting it happen to users, which is what FALCON reference implementation suggests also.

Algorithm 11 SecFPR

Input: 1-bit Boolean shares $(s_i)_{1\le i\le n}$ for value s, 16-bit arithmetic shares $(e_i)_{1\le i\le n}$ for value e, 55-bit Boolean shares $(z_i)_{1\le i\le n}$ for value z

Output: Boolean shares $(x_i)_{1\le i\le n}$ representing the floating-point numbers packed by s, e, z

1: $e_1 \leftarrow e_1 + 1076$
2: $(e_i) \leftarrow \mathsf{A2B}((e_i))$
3: $(b_i) \leftarrow (-e_i^{(16)})$
4: $(z_i) \leftarrow \mathsf{SecAnd}((z_i), (\neg b_1, b_2, \cdots, b_n))$ ▷ set $z = 0$ if $e < 0$
5: $(e_i) \leftarrow \mathsf{SecAnd}((e_i), (-z_i^{(55)}))$ ▷ set $e = 0$ if $z = 0 \Leftrightarrow z^{(55)} = 0$
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)

3.5 Masked Floating-Point Number Multiplication

We now introduce the masked floating-point number multiplication SecFprMul in Algorithm 12. To begin with, we consider its input floating-point numbers to be split into three parts of arithmetic shares: one-bit sign bit shares (s_i), 16-bit exponent shares (e_i), and 128-bit mantissa shares (m_i) where

$$s = \bigoplus_i s_i = \sum_i s_i \bmod 2, \quad e = \sum_i e_i \bmod 2^{16}, \quad m = \sum_i m_i^{[128:1]} \bmod 2^{128}$$

We use this form of input to make the operations more straightforward. It should be noted that the pre-image vector computation is the first operation that should be masked in the signing process, and hence this form of input can be derived directly from unmasked values.

To start with, the sign bit XOR and the exponent addition can be simply done by adding the corresponding share of the input (lines 1 and 2). As in Algorithm 5, a constant -2100 is also added. Mantissa multiplication is done by the SecMult gadget (line 3), which

multiplies each share of both operands and adds them together carefully by inserting random mask values.

The next step is to round the mantissa to range $[2^{54}, 2^{55})$ and preserve the sticky bit. We first convert the arithmetic-shared mantissa into Boolean shares (line 4). The product is in the range $[2^{104}, 2^{106})$, so we shift the product by 50 or 51 bits according to the 106th bit (line 6 to line 10). Note that for one-bit shares, the negative of it can be achieved by turning each share negative, which is used in our conditional shift in lines 9 and 10. The 106th bit is then converted to 16-bit arithmetic shares by the one-bit $\mathsf{B2A}_{\mathsf{Bit}}$ and added to the exponent (line 13). To preserve the sticky bit, consider that if shifted by 50, we need to OR the 51st bit of mantissa with the nonzero result of the last 50 bits, while for a 51-bit shift, we OR the 52nd bit with the nonzero result of the last 51 bits. This can be done in both cases by using our SecNonzero on the last 51 bits and OR the result with the shifted mantissa, which is done in lines 5 and 11.

Finally, we turn the mantissa into zero if any exponent of the input is zero (line 14 to line 17), and we also make this by our SecNonzero gadget of the arithmetic-shared version. Now we get the sign bit, exponent, and 55-bit mantissa, which are in Boolean, arithmetic, and Boolean shares, respectively. We pack them into one Boolean-shared floating-point number in SecFPR.

Algorithm 12 SecFprMul

Input: Shares $(sx_i)_{1\le i\le n}, (ex_i)_{1\le i\le n}, (mx_i)_{1\le i\le n}$ for floating-point number x, shares $(sy_i)_{1\le i\le n}, (ey_i)_{1\le i\le n}, (my_i)_{1\le i\le n}$ for floating-point number y

Output: Boolean shares representing the floating-point numbers product.

1: $(s_i) \leftarrow (sx_i \oplus sy_i)$
2: $(e_i) \leftarrow (ex_1 + ey_1 - 2100, ex_2 + ey_2, \cdots, ex_n + ey_n)$
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: $(z_i) \leftarrow (z'_i \oplus z_i)$ ▷ conditional shift
11: $(z_i) \leftarrow \mathsf{SecOr}((z_i), (b_i))$ ▷ preserve the sticky bit
12: $(w_i) \leftarrow \mathsf{B2A}_{\mathsf{Bit}}((w_i))$
13: $(e_i) \leftarrow (e_i + w_i)$ ▷ add exponent by the 106th bit
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: $(z_i) \leftarrow \mathsf{SecAnd}((z_i), (-d_i^{(1)}))$ ▷ set $z = 0$ if exponent of any operand is 0
18: **return** $\mathsf{SecFPR}((s_i), (e_i), (z_i))$

3.6 Masked Floating-Point Number Addition

The SecFprAdd in Algorithm 13 takes in two Boolean-shared floating-point values and adds them in a masked way. We use Boolean shares as its input since it is followed by floating-point number multiplications in the pre-image vector computation.

The algorithm first exchanges the operands to make the first one no less than the second (line 1 to line 9). Thanks to the biased exponent, we can make the comparison by a simple subtraction and check the sign bit. Originally, the subtraction of two Boolean-masked

values could be done by three steps: (1) inverting all the bits of the second operand (2) adding the inverted result with 1 (3) adding the first and second operand, which applies the equation $x - y = x + (\neg y) + 1$ To avoid an additional call to the SecAdd gadget, we only invert bits and consider the boundary conditions. To put it clearly, let u, v be two values stored in 64-bit registers; we use the relation

$$\begin{aligned}[\![u - v < 0]\!] &= [\![u - v - 1 < 0]\!] \oplus [\![u - v - 1 = -1]\!] \oplus [\![u - v - 1 = 2^{63} - 1]\!] \\ &= [\![u + (\neg v) < 0]\!] \oplus [\![u + (\neg v) = -1]\!] \oplus [\![u + (\neg v) = 2^{63} - 1]\!] \\ &= [\![u + (\neg v) < 0]\!] \oplus [\![u + (\neg v) \neq -1]\!] \oplus [\![u + (\neg v) \neq 2^{63} - 1]\!]\end{aligned}$$

The first and second equalities come from the two's complement representation, where an increment of $2^{63} - 1$ results in -2^{63} and $\neg v = -v - 1$. The third equality is for the output of the SecNonzero gadget. To evaluate the range check of the value $u + (\neg v)$, we also use both of the facts in our algorithm (lines 4 and 5)

$$u + (\neg v) \neq -1 \Longleftrightarrow \neg(u + (\neg v)) \neq 0$$

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

After the conditional swap, we extract the sign bit, exponent, and mantissa from the input shares. Like in Algorithm 6, the mantissa is scaled up 3 bits for further rounding precision, and the exponent is turned to arithmetic shares and subtracted by 1078. Then we use the SecFprUrsh gadget (Algorithm 9) we introduced in Section 3.2 to right-shift the Boolean-masked mantissa of the second operand with sticky bit preserved to make both operands have identical exponents. Note that we set the value to zero if the exponent difference is larger than 59 before shifting, as indicated in lines 9 and 10 of unmasked Algorithm 6 and lines 15 and 16 of Algorithm 13.

After the proper shift, we add/subtract the result to/from the mantissa of the first operand (line 24). The sum has a wide range, so we normalize it to $[2^{63}, 2^{64})$ by the SecFprNorm64 gadget (Algorithm 10) in Section 3.3 and right-shift the result by 9 bits (line 27). Finally, we get the sign bit and exponent of the first operand and the shifted mantissa sum in the range $[2^{54}, 2^{55})$, and the result floating-point number is given by calling SecFPR as the case of multiplication.

4 Security Proof

In this section, we sequentially prove that our design with $n = t + 1$ shares is secure regarding t-NI and t-SNI security.

Lemma 1. *The gadget* SecOr *(Algorithm 7) is t-SNI secure.*

Proof. This is a direct result that the SecAnd gadget is t-SNI secure and the negation is operated share-by-share. □

Lemma 2. *The gadget* SecNonzero *(Algorithm 8) is t-SNI secure.*

Proof. Since the A2B gadget is t-SNI secure, we only need to show that the following loop is itself t-SNI. An abstract diagram of each iteration is given in Figure 1. Since SecOr is t-SNI secure, for any probing set $\mathcal{P}_1$ in SecOr gadget and $\mathcal{O}$ of its output shares, one can use some set $\mathcal{S}_1^1$ of outputs of Refresh and set $\mathcal{S}_1^2$ of shares of (r_i) to simulate both $\mathcal{P}_1$ and $\mathcal{O}$ with $|\mathcal{S}_1^1|, |\mathcal{S}_1^2| \leq \mathcal{P}_1$. Also, since Refresh is t-SNI secure, for any probing set $\mathcal{P}_2$ in Refresh gadget, one can use some set $\mathcal{S}_2$ of shares of (t_i) to simulate both $\mathcal{P}_2$ and $\mathcal{S}_1^1$ with $|\mathcal{S}_2| \leq \mathcal{P}_2$. In summary, for probed output shares $\mathcal{O}$ and internal values $\mathcal{P}_1, \mathcal{P}_2$, one can use $\mathcal{S}_1^2$ and $\mathcal{S}_2$ to simulate all of them with $|\mathcal{S}_1^2 \cup \mathcal{S}_2| \leq |\mathcal{P}_1| + |\mathcal{P}_2|$. This shows each iteration is t-SNI secure, and the whole loop is thus t-SNI secure.

□

Algorithm 13 SecFprAdd

Input: Boolean shares $(x_i)_{1\le i\le n}$ and $(y_i)_{1\le i\le n}$ representing floating-point numbers x and y

Output: Boolean shares representing the floating-point numbers 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: $(d_i) \leftarrow \mathsf{SecAdd}((xm_i), (ym_i))$ ▷ $d = xm - ym - 1$
4: $(b_i) \leftarrow \mathsf{SecNonzero}(\neg d_1, d_2, \cdots, d_n)$ ▷ $[\![d \neq -1]\!]$
5: $(b'_i) \leftarrow \mathsf{SecNonzero}(\neg(d_1 \oplus (1 \ll 63)), d_2, \cdots, d_n)$ ▷ $[\![d \neq 2^{63} - 1]\!]$
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: $(x_i) \leftarrow (x_i \oplus m_i),\ (y_i) \leftarrow (y_i \oplus m_i)$ ▷ swap x and y if necessary
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: $(my_i) \leftarrow \mathsf{SecAnd}((my_i), (-(c_i'^{(16)})))$ ▷ set my to 0 if the exponent difference > 60
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: $(my'_i) \leftarrow \mathsf{SecAdd}((my'_i), (1, 0, \cdots, 0))$ ▷ $-my$
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: $(my_i) \leftarrow (my_i \oplus my'_i)$ ▷ $(-1)^{sx \oplus sy} my$
24: $(z_i) \leftarrow \mathsf{SecAdd}((mx_i), (my_i))$ ▷ $z = mx + (-1)^{sx \oplus sy} my$
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: $(z_i^{(1)}) \leftarrow (b_i)$ ▷ preserve the sticky bit
29: $ex_1 \leftarrow ex_1 + 9$
30: **return** $\mathsf{SecFPR}(\mathsf{Refresh}((sx_i)), (ex_i), (z_i))$

Lemma 3. *The gadget* SecFprUrsh *(Algorithm 9) is t-SNI secure.*

Proof. We first show that the loop of rotation is itself t-SNI secure. Note that since there are n iterations, at least one of them is not probed. Let it be the iteration when $j = j^*$. Since any set of output shares of RefreshMasks with size $\le n - 1$ is uniformly distributed ([BCZ18], Lemma 1), all the probes after j^*, including probes of outputs, can be simulated with fresh randomness. Thus we only need to show that one can simulate probes before j^* with no more number of shares.

Since the rotation is done share-by-share, one can simulate probes of (x_i) and (m_i) with the same number of input shares. As for the simulation of c_j, if in some iteration $j = j'$ the rotation is probed, one then adds $c_{j'}$ into the simulation set. Also, if consecutive RefreshMasks in iterations $j = j' - 1, j'$ are probed, one adds $c_{j'}$ into the simulation set. Note that if RefreshMasks are not consecutively probed, one can simulate c_j with fresh randomness thanks to the uniformity of RefreshMasks's outputs. In this way, the size of the simulation set of c_j is no more than the number of probes.

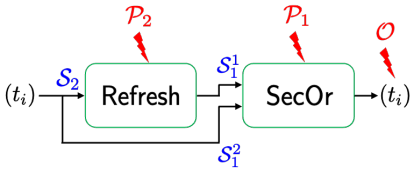


Figure 1: An abstract diagram of each iteration in SecNonzero (Algorithm 8). The probing sets $\mathcal{O}$, $\mathcal{P}_1$, $\mathcal{P}_2$ are colored in red, and the simulations sets $\mathcal{S}_1^1$, $\mathcal{S}_1^2$, $\mathcal{S}_2$ are colored in blue. Gadgets with t-NI and t-SNI security are marked in black and green, respectively.

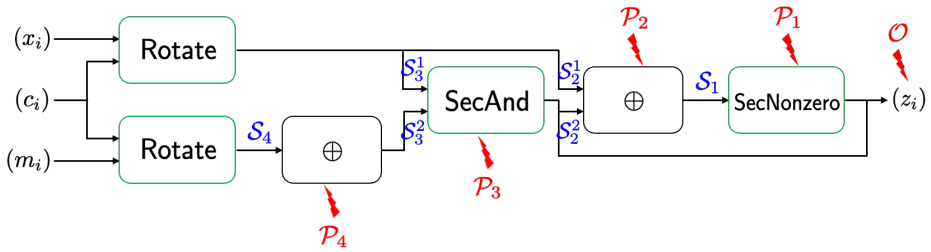


Figure 2: An abstract diagram of SecFprUrsh (Algorithm 9). The probing sets $\mathcal{O}$ and $\mathcal{P}_i$ for some i are colored in red, and the simulations sets $\mathcal{S}_i$ and $\mathcal{S}_i^j$ for some i, j are colored in blue. Gadgets with t-NI and t-SNI security are marked in black and green, respectively.

Now we show the operations following the rotation loop are t-SNI secure, and therefore the whole gadget is t-SNI secure. An abstract diagram of SecFprUrsh is given in Figure 2. Let the adversary probe the intermediate values sets $\mathcal{O}$ of the output shares, $\mathcal{P}_1$ of the SecNonzero, $\mathcal{P}_2$ of the XOR following SecAnd, $\mathcal{P}_3$ of the SecAnd, and $\mathcal{P}_4$ of the XOR before SecAnd. First, by the t-SNI security of SecNonzero, one can use some sets $\mathcal{S}_1$ of output shares of the XOR operation to simulate $\mathcal{P}_1$ and the first 63 bits of $\mathcal{O}$ with size no more than $|\mathcal{P}_1|$. The XOR operation is done share-by-share, so there are some sets $\mathcal{S}_2^1, \mathcal{S}_2^2$ of output shares of the rotation and SecAnd, respectively, that can simulate $\mathcal{P}_2$ and $\mathcal{S}_1$. Note only $|\mathcal{O} \cup \mathcal{S}_2^2| \leq |\mathcal{O}| + |\mathcal{P}_2| + |\mathcal{P}_1| \leq t$ output shares of SecAnd are used. Since SecAnd is t-SNI secure, one can use some sets $\mathcal{S}_3^1, \mathcal{S}_3^2$ to simulate $\mathcal{P}_3$ and $\{\text{the last bit of } \mathcal{O}\} \cup \mathcal{S}_2^2$ with sizes no more than $\mathcal{P}_3$. Finally, one can simulate the probing set $\mathcal{P}_4$ in the XOR and $\mathcal{S}_3^2$ with output shares $\mathcal{S}_4$ of the rotation of (m_i). All the probes are now simulated with output shares $\mathcal{S}_2^1 \cup \mathcal{S}_3^1$ of the rotation of (x_i) and output shares $\mathcal{S}_4$ of the rotation of (m_i). $|\mathcal{S}_2^1 \cup \mathcal{S}_3^1| \leq |\mathcal{P}_2| + |\mathcal{P}_1| + |\mathcal{P}_3|$ and $|\mathcal{S}_4| \leq |\mathcal{P}_4| + |\mathcal{P}_3|$. They, along with the internal probes into the rotation loop, can be simulated by input shares due to the t-SNI security we showed at first.

□

Lemma 4. *The gadget* SecFprNorm64 *(Algorithm 10) is t-NI secure.*

Proof. An abstract diagram of each iteration in SecFprNorm64 is given in Figure 3. Let the adversary probe in iteration j the intermediate values set $\mathcal{P}_1^{(j)}$ of the addition, $\mathcal{P}_2^{(j)}$ of the XOR, $\mathcal{P}_3^{(j)}$ of the $\mathsf{B2A}_{\mathsf{Bit}}$, $\mathcal{P}_4^{(j)}$ of the SecAnd, and $\mathcal{P}_5^{(j)}$ of the SecNonzero. We show that all probes in iteration j can be simulated with no more number of shares of (e_i) and (x_i) as the input of the iteration. If this is the case, all probes across different iterations can be simulated with no more number of input shares.

First, since the addition is done share-by-share, one can use some sets $\mathcal{S}_1^1$ and $\mathcal{S}_1^2$ of shares of (e_i) and $\mathsf{B2A}_{\mathsf{Bit}}$, respectively, to simulate $\mathcal{P}_1^{(j)}$. One can also use some sets $\mathcal{S}_2^1$ and $\mathcal{S}_2^2$ of shares of SecAnd and (x_i), respectively, to simulate $\mathcal{P}_2^{(j)}$. The t-SNI security of

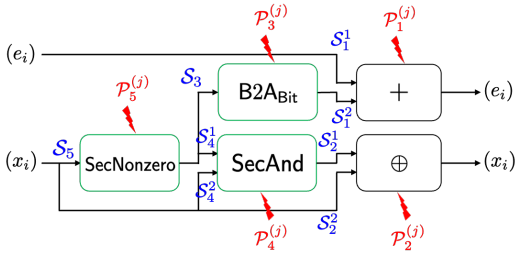


Figure 3: An abstract diagram of iteration j in SecFprNorm64 (Algorithm 10). The probing sets $\mathcal{P}_i^{(j)}$ for some i are colored in red, and the simulation sets $\mathcal{S}_i$ and $\mathcal{S}_i^k$ for some i, k are colored in blue. Gadgets with t-NI and t-SNI security are marked in black and green, respectively.

B2A$_{\text{Bit}}$ assures that one can use some set $\mathcal{S}_3$ of the output of the SecNonzero with the size no more than $|\mathcal{P}_3^{(j)}|$ to simulate $\mathcal{P}_3^{(j)}$ and $\mathcal{S}_1^2$. Similarly, one can use some sets $\mathcal{S}_4^1, \mathcal{S}_4^2$ of the SecNonzero and (x_i), respectively, to simulate $\mathcal{P}_4^{(j)}$ and $\mathcal{S}_2^1$ with sizes no more than $|\mathcal{P}_4^{(j)}|$. Note only $|\mathcal{S}_3 \cup \mathcal{S}_4^1| \leq |\mathcal{P}_3^{(j)}| + |\mathcal{P}_4^{(j)}|$ shares of the SecNonzero are used, and they, along with $\mathcal{P}_5^{(j)}$, can be simulated by some set $\mathcal{S}_5$ of (x_i) with size no more than $|\mathcal{P}_5^{(j)}|$. Now all the probes are simulated by shares $\mathcal{S}_1^1$ of (e_i) and shares $\mathcal{S}_2^2, \mathcal{S}_4^2$ and $\mathcal{S}_5$ of (x_i). As $|\mathcal{S}_1^1| \leq \mathcal{P}_1^{(j)}$ and $|\mathcal{S}_2^2 \cup \mathcal{S}_4^2 \cup \mathcal{S}_5| \leq |\mathcal{P}_2^{(j)}| + |\mathcal{P}_4^{(j)}| + |\mathcal{P}_5^{(j)}|$, we show that all the probes in iteration j can be simulated with no more number of shares of (e_i) and (x_i).

□

We now give proofs that our main algorithms SecFPR (Algorithm 11), SecFprMul (Algorithm 12), and SecFprAdd (Algorithm 13) are t-SNI secure.

Theorem 1. *The gadget* SecFPR *(Algorithm 11) is t-SNI secure.*

Proof. We use an abstract diagram of SecFPR in Figure 4 to demonstrate our proof. Assume the adversary probes t values, including output shares set $\mathcal{O}$ and sets P_i in each gadget for $i = 1, 2, \cdots, 10$. Also, assume we use sets S_i or S_i^j for some j to simulate the necessary values for the corresponding gadgets. For example, we use $\mathcal{S}_1^1, \mathcal{S}_1^2$ to simulate $\mathcal{P}_1$ and $\mathcal{O}$, and $\mathcal{S}_2$ to simulate $\mathcal{P}_2$ and $\mathcal{S}_1$, just to name a few. Our goal is to show that if the size of all the probing sets P_i is $t_I \leq t$, and if the size of values we require to simulate in each gadget is smaller than t, then the simulation sets of input shares (that is, $\mathcal{S}_8, \mathcal{S}_{10}$, and $\mathcal{S}_9^2$) have sizes no more than t_I.

Since gadget SecAdd is t-NI, we have $|\mathcal{S}_1^1|, |\mathcal{S}_1^2| \leq |\mathcal{P}_1| + |\mathcal{O}|$. Due to the t-SNI security of Refresh and SecAnd, we have $|\mathcal{S}_2| \leq |\mathcal{P}_2|$ and $|\mathcal{S}_3^1|, |\mathcal{S}_3^2| \leq |\mathcal{P}_3|$. Similarly, we can sequentially derive

- $|\mathcal{S}_4^1|, |\mathcal{S}_4^2| \leq |\mathcal{P}_4| + |\mathcal{S}_2|$
- $|\mathcal{S}_5^1|, |\mathcal{S}_5^2| \leq |\mathcal{P}_5|$
- $|\mathcal{S}_6^1|, |\mathcal{S}_6^2| \leq |\mathcal{P}_6|$
- $|\mathcal{S}_7^1| \leq |\mathcal{P}_7|$
- $|\mathcal{S}_8| \leq |\mathcal{P}_8|$
- $|\mathcal{S}_9^1|, |\mathcal{S}_9^2| \leq |\mathcal{P}_9|$
- $|\mathcal{S}_{10}| \leq |\mathcal{P}_{10}|$

Based on the above inequalities, one can check that the number of values to simulate in each gadget is no more than $t_I + |\mathcal{O}| = t$. Finally we derive $|\mathcal{S}_8| \leq |\mathcal{P}_8|, |\mathcal{S}_{10}| \leq |\mathcal{P}_{10}|$, and $|\mathcal{S}_9^2| \leq |\mathcal{P}_9|$, which are all no more than t_I.

□

Theorem 2. *The gadget* SecFprMul *(Algorithm 12) is t-SNI secure.*

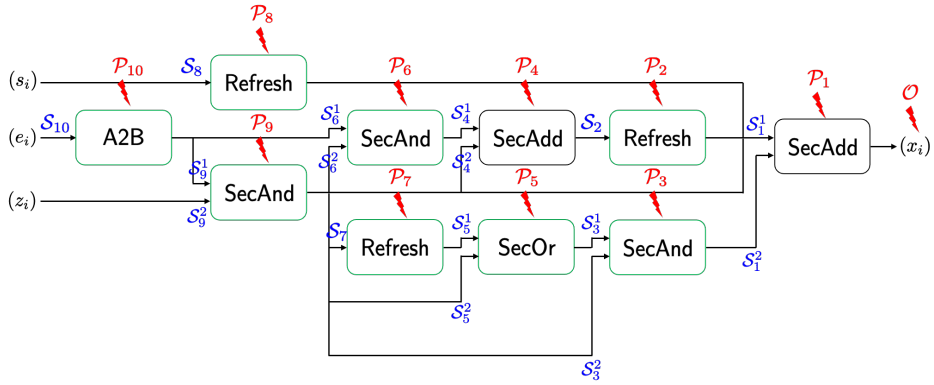


Figure 4: An abstract diagram of SecFPR (Algorithm 11). The probing sets $\mathcal{O}$ and $\mathcal{P}_i$ for some i are colored in red, and the simulation sets $\mathcal{S}_i$ and $\mathcal{S}_i^j$ for some i, j are colored in blue. Gadgets with t-NI and t-SNI security are marked in black and green, respectively.

Proof. We use an abstract diagram of SecFprMul in Figure 5 to demonstrate our proof. Assume the adversary probes t values, including output shares set $\mathcal{O}$ and sets P_i in each gadget for $i = 1, 2, \cdots, 15$. Also, assume we use sets S_i or S_i^j for some j to simulate the necessary values for the corresponding gadgets. Similarly, we need to show that if the size of all the probing sets P_i is $t_I \leq t$, and if the size of values we require to simulate in each gadget is smaller than t, then the simulation sets of input shares (that is, $\mathcal{S}_9^1, \mathcal{S}_9^2, \mathcal{S}_{11}^1, \mathcal{S}_{11}^2, \mathcal{S}_{15}^1$, and $\mathcal{S}_{15}^2$) have sizes no more than t_I.

Theorem 1 shows that the SecFPR gadget is t-SNI secure, so we have $|\mathcal{S}_1^1|, |\mathcal{S}_1^2|, |\mathcal{S}_1^3| \leq |\mathcal{P}_1|$. We then sequentially write down all the inequalities based on the t-NI and t-SNI properties of the gadgets.

- $|\mathcal{S}_2^1|, |\mathcal{S}_2^2| \leq |\mathcal{P}_2|$
- $|\mathcal{S}_3^1|, |\mathcal{S}_3^2| \leq |\mathcal{P}_3|$
- $|\mathcal{S}_4^1|, |\mathcal{S}_4^2| \leq |\mathcal{P}_4| + |\mathcal{S}_3^1|$
- $|\mathcal{S}_5^1|, |\mathcal{S}_5^2| \leq |\mathcal{P}_5|$
- $|\mathcal{S}_6| \leq |\mathcal{P}_6|$
- $|\mathcal{S}_7| \leq |\mathcal{P}_7|$
- $|\mathcal{S}_8| \leq |\mathcal{P}_8|$
- $|\mathcal{S}_9^1|, |\mathcal{S}_9^2| \leq |\mathcal{P}_9|$
- $|\mathcal{S}_{10}^1|, |\mathcal{S}_{10}^2| \leq |\mathcal{P}_{10}| + |\mathcal{S}_1^2|$
- $|\mathcal{S}_{11}^1|, |\mathcal{S}_{11}^2| \leq |\mathcal{P}_{11}| + |\mathcal{S}_{10}^1|$
- $|\mathcal{S}_{12}^1|, |\mathcal{S}_{12}^2| \leq |\mathcal{P}_{12}|$
- $|\mathcal{S}_{13}| \leq |\mathcal{P}_{13}|$
- $|\mathcal{S}_{14}| \leq |\mathcal{P}_{14}|$
- $|\mathcal{S}_{15}^1|, |\mathcal{S}_{15}^2| \leq |\mathcal{P}_{15}| + |\mathcal{S}_1^1|$

Now we derive

$$|\mathcal{S}_9^1|, |\mathcal{S}_9^2| \leq |\mathcal{P}_9|$$
$$|\mathcal{S}_{11}^1|, |\mathcal{S}_{11}^2| \leq |\mathcal{P}_{11}| + |\mathcal{S}_{10}^1| \leq |\mathcal{P}_{11}| + |\mathcal{P}_{10}| + |\mathcal{S}_1^2| \leq |\mathcal{P}_{11}| + |\mathcal{P}_{10}| + |\mathcal{P}_1|$$
$$|\mathcal{S}_{15}^1|, |\mathcal{S}_{15}^2| \leq |\mathcal{P}_{15}| + |\mathcal{S}_1^1| \leq |\mathcal{P}_{15}| + |\mathcal{P}_1|$$

Sizes of all of them are no more than t_I.

One can also show that the number of values to simulate in each gadget is no more than t. Here we show the case for the A2B gadget since it is the most complicated one. The simulation set $\mathcal{S}_8$ needs to simulate $\mathcal{P}_8$ and $\mathcal{S}_{10}^2 \cup \mathcal{S}_6 \cup \mathcal{S}_5^2 \cup \mathcal{S}_4^2 \cup \mathcal{S}_7$. Note we have $|\mathcal{S}_{10}^2| \leq |\mathcal{P}_{10}| + |\mathcal{S}_1^2| \leq |\mathcal{P}_{10}| + |\mathcal{P}_1|$, $|\mathcal{S}_6| \leq |\mathcal{P}_6|$, $|\mathcal{S}_5^2| \leq |\mathcal{P}_5|$, $|\mathcal{S}_4^2| \leq |\mathcal{P}_4| + |\mathcal{S}_3^1| \leq |\mathcal{P}_4| + |\mathcal{P}_3|$, and $|\mathcal{S}_7| \leq |\mathcal{P}_7|$. The number of values $\mathcal{S}_8$ needs to simulate is smaller than

$$|\mathcal{P}_8| + |\mathcal{P}_{10}| + |\mathcal{P}_1| + |\mathcal{P}_6| + |\mathcal{P}_5| + |\mathcal{P}_4| + |\mathcal{P}_3| + |\mathcal{P}_7|$$

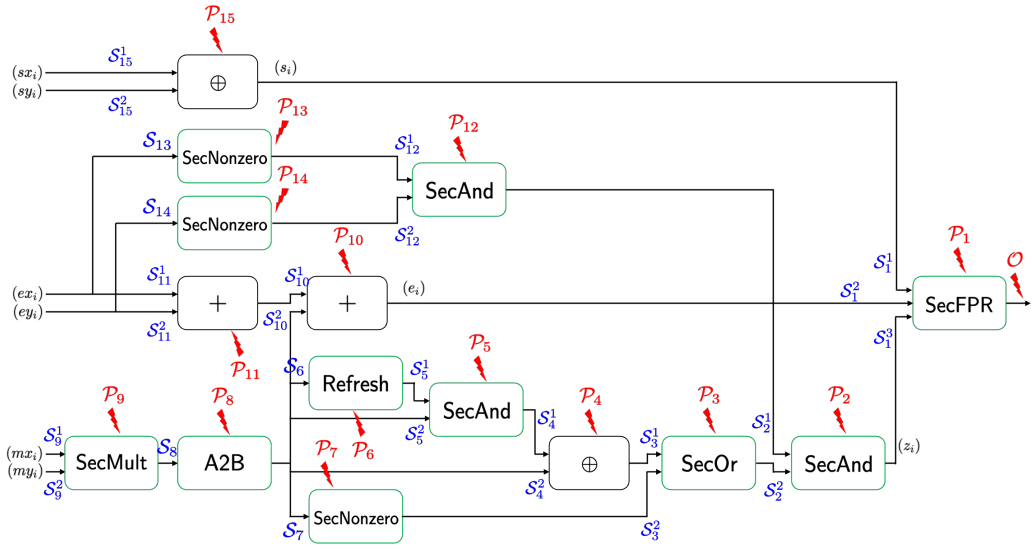


Figure 5: An abstract diagram of SecFprMul (Algorithm 12). The probing sets $\mathcal{O}$ and $\mathcal{P}_i$ for some i are colored in red, and the simulation sets $\mathcal{S}_i$ and $\mathcal{S}_i^j$ for some i, j are colored in blue. Gadgets with t-NI and t-SNI security are marked in black and green, respectively.

which is guaranteed to be no more than t_I.

□

Theorem 3. *The gadget* SecFprAdd *(Algorithm 13) is t-SNI secure.*

Proof. We first show that for any probing sets in the swap part of the algorithm, one can use no more number of shares of each input to simulate. An abstract diagram of the swap part is given in Figure 6. Similar to our proof in Theorem 1 and 2, we write down the inequalities that the t-NI and t-SNI gadgets provide.

- $|\mathcal{S}_1^1|, |\mathcal{S}_1^2| \leq |\mathcal{P}_1|$
- $|\mathcal{S}_2^1|, |\mathcal{S}_2^2| \leq |\mathcal{P}_2|$
- $|\mathcal{S}_3^1|, |\mathcal{S}_3^2| \leq |\mathcal{P}_3|$
- $|\mathcal{S}_4^1|, |\mathcal{S}_4^2| \leq |\mathcal{P}_4|$
- $|\mathcal{S}_5^1|, |\mathcal{S}_5^2| \leq |\mathcal{P}_5|$
- $|\mathcal{S}_6^1|, |\mathcal{S}_6^2| \leq |\mathcal{P}_6| + |\mathcal{S}_4^1| \leq |\mathcal{P}_6| + |\mathcal{P}_4|$
- $|\mathcal{S}_7^1|, |\mathcal{S}_7^2| \leq |\mathcal{P}_7| + |\mathcal{S}_6^2| \leq |\mathcal{P}_7| + |\mathcal{P}_6| + |\mathcal{P}_4|$
- $|\mathcal{S}_8| \leq |\mathcal{P}_8|$
- $|\mathcal{S}_9| \leq |\mathcal{P}_9|$
- $|\mathcal{S}_{10}^1|, |\mathcal{S}_{10}^2| \leq |\mathcal{P}_{10}| + |\mathcal{S}_6^1| + |\mathcal{S}_8| + |\mathcal{S}_9| \leq |\mathcal{P}_{10}| + |\mathcal{P}_6| + |\mathcal{P}_4| + |\mathcal{P}_8| + |\mathcal{P}_9|$
- $|\mathcal{S}_{11}^1|, |\mathcal{S}_{11}^2| \leq |\mathcal{P}_{11}| + |\mathcal{S}_3^1| \leq |\mathcal{P}_{11}| + |\mathcal{P}_3|$

Therefore we can use at most

$$|\mathcal{S}_2^2 \cup \mathcal{S}_5^2 \cup \mathcal{S}_{10}^2 \cup \mathcal{S}_{11}^2| \leq |\mathcal{P}_2| + |\mathcal{P}_5| + |\mathcal{P}_{10}| + |\mathcal{P}_6| + |\mathcal{P}_4| + |\mathcal{P}_8| + |\mathcal{P}_9| + |\mathcal{P}_{11}| + |\mathcal{P}_3|$$

of shares of (x_i), and

$$|\mathcal{S}_1^1 \cup \mathcal{S}_{10}^1 \cup \mathcal{S}_{11}^1| \leq |\mathcal{P}_1| + |\mathcal{P}_{10}| + |\mathcal{P}_6| + |\mathcal{P}_4| + |\mathcal{P}_8| + |\mathcal{P}_9| + |\mathcal{P}_{11}| + |\mathcal{P}_3|$$

of shares of (y_i) to simulate all the probed values.

Now we show that the operations after the swap are t-SNI secure, implying that the whole algorithm is t-SNI secure. An abstract diagram of this part is given in Figure 7.

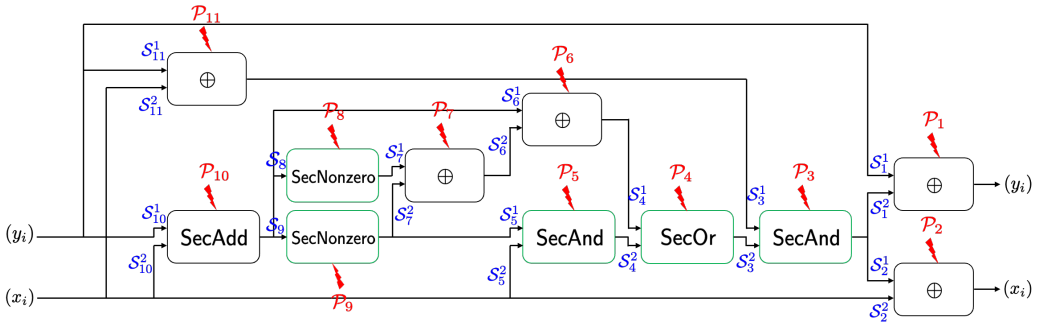


Figure 6: An abstract diagram of the swap part in SecFprAdd (Algorithm 13). The probing sets $\mathcal{P}_i$ for some i are colored in red, and the simulation sets $\mathcal{S}_i$ and $\mathcal{S}_i^j$ for some i, j are colored in blue. Gadgets with t-NI and t-SNI security are marked in black and green, respectively.

From the figure, one can list all the inequalities similarly and check that for each gadget, there are no more number of input shares than the probes used in the simulation. In particular, shares of (x_i) used to simulate include $\mathcal{S}_5^2$, $\mathcal{S}_{15}$, and $\mathcal{S}_{18} \cup \mathcal{S}_{17}^1$, and

$$|\mathcal{S}_5^2| \le |\mathcal{P}_5| + |\mathcal{S}_4^1| \le |\mathcal{P}_5| + |\mathcal{P}_4| + |\mathcal{S}_2^1 \cup \mathcal{S}_3| \le |\mathcal{P}_5| + |\mathcal{P}_4| + |\mathcal{P}_1| + |\mathcal{P}_2| + |\mathcal{P}_3|$$

$$|\mathcal{S}_{15}| \le |\mathcal{P}_{15}|$$

$$|\mathcal{S}_{18} \cup \mathcal{S}_{17}^1| \le |\mathcal{P}_{18}| + |\mathcal{P}_{17}| + |\mathcal{S}_7^1| \le |\mathcal{P}_{18}| + |\mathcal{P}_{17}| + |\mathcal{P}_7|$$

For shares of (y_i), $\mathcal{S}_{12}^2$, $\mathcal{S}_{16}$, and $\mathcal{S}_{17}^2$ are used, and

$$|\mathcal{S}_{12}^2| \le |\mathcal{P}_{12}|$$

$$|\mathcal{S}_{16}| \le |\mathcal{P}_{16}|$$

$$|\mathcal{S}_{17}^2| \le |\mathcal{P}_{17}| + |\mathcal{S}_7^1| \le |\mathcal{P}_{17}| + |\mathcal{P}_7|$$

Both sums are smaller than the number of internal probes.

□

5 Implementation and Evaluation

In this section, we provide our masked implementation's performance and security evaluations. Our experiments were mainly implemented in plain-C code, but we rewrote some segments of the 2-shared version by assembly to reduce some observed leakages in security evaluation, which is discussed in Section 5.2 and 5.3. We first tested the performance of our design on an Arm Cortex-M4 processor, and then we used the program from FALCON reference code [PFH+20] to test the speed of one complete signing process on an Intel-Core i9-12900KF CPU, a general-purpose processor. For security evaluation, we ran our experiments on ChipWhisperer-Pro Level 3 Starter Kit [Inc], which includes a main control board and a target board to run the main program. The control board clocks the target board at 7.38MHz and measures its power consumption during execution at the same frequency. The target board STM32F415 (CW308T-STM32F4) with an Arm Cortex-M4 MCU was used.

In our implementation, the 128-bit multiplication in Algorithm 12 was realized by combining four 32-bit registers in C. We generated the randomness for our masked implementation beforehand and fill them in a table to be read off. We list the number of used randomness in bytes for each algorithm in the performance evaluation subsection.

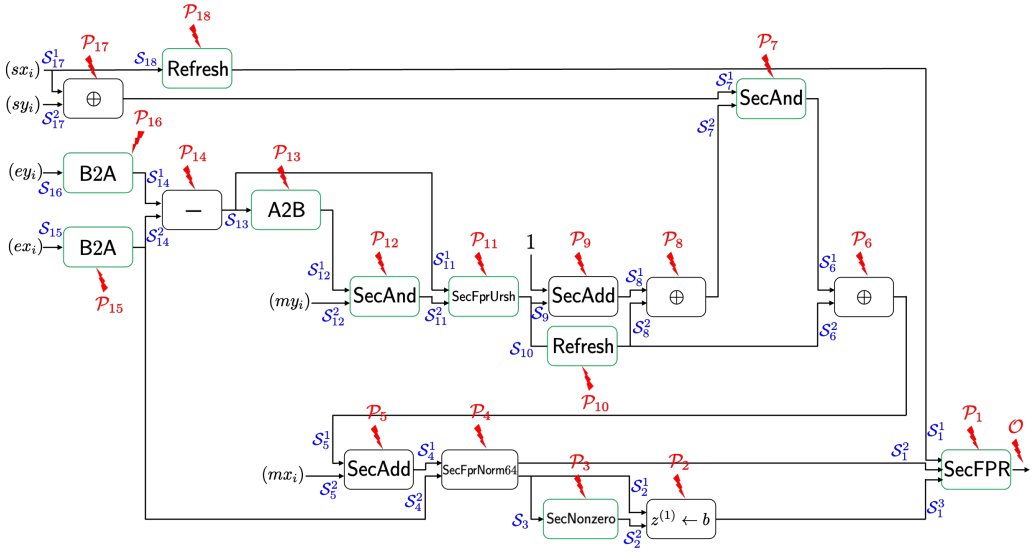


Figure 7: An abstract diagram of the operations following the swap in SecFprAdd (Algorithm 13). The probing sets $\mathcal{O}$ and $\mathcal{P}_i$ for some i are colored in red, and the simulation sets $\mathcal{S}_i$ and $\mathcal{S}_i^j$ for some i, j are colored in blue. Gadgets with t-NI and t-SNI security are marked in black and green, respectively.

5.1 Performance Evaluation

We first evaluate the performance of our masked implementation on the Arm Cortex-M4 processor and compare them with the unmasked version from Falcon reference code in NIST round-3 submission [PFH+20], which is a re-implementation of floating-point arithmetic also written in plain-C. The cycle counts of floating-point number multiplication and addition are given in Table 2. For higher-order mask evaluation, we provide the results for the second-order mask (3 shares). All our code was compiled by `arm-none-eabi-gcc 10.3.1` with optimization level `-O3`.

The masked floating-point number multiplication takes about 23× overhead of the unmasked version for 2 shares and 118× overhead for 3 shares. It shows that our nonzero check gadget SecNonzero only causes a small amount of overhead compared to the whole multiplication and addition algorithm. For the 2-shared design, the bottleneck is the packing SecFPR, in which a 64-bit SecAdd is used. For the 3-shared design, the heaviest overhead comes from the 3-shared 128-bit A2B, which internally calls a 128-bit SecAdd.

The masked floating-point number addition takes about 35× cycles for our first-order masked version than the unmasked one and 99× cycles for our second-order version. It shows that the main overhead is caused by the four 64-bit SecAdd functions, which is also the case in multiplication. Although it costs much in our implementation, it seems unlikely to avoid the Boolean masked addition gadgets or the mask conversion gadgets somewhere since the mantissa needs to be rounded in different stages and the sticky bit needs to be preserved.

In Table 3, we provide the speed for signing one message on the general-purpose Intel-Core i9-12900KF CPU with our masking countermeasure on the pre-image vector computation to show the amortized performance result in the whole Falcon. For this, we first replaced the floating-point arithmetic in the pre-image vector computation (line 3) with our masked multiplication and addition, and then we unmasked the result after all the computations were done. It shows that compared with the unmasked design, one signing process takes about 7.7× for 2 shares (about 1.9 ms for Falcon-512) and about

Table 2: Performance of each component in Algorithm 11, 12, and 13 on Arm Cortex-M4. We count the cycles for subroutines and the total random numbers used in bytes.

Algorithm		Cycles		
		Unmasked	**2 Shares**	**3 Shares**
SecFprMul	**Total**	**308**	**7134**	**36388**
	128-bit A2B in line 4	-	1619	19253
	64-bit SecNonzero in line 5	-	389	1350
	Two 16-bit SecNonzero in line 14, 15	-	662	2012
	SecFPR in Algorithm 11	-	3362	10813
	#randombytes	-	333	2005
SecFprAdd	**Total**	**487**	**17154**	**48291**
	Three 64-bit SecAdd in line 3, 19, 24	-	6990	16956
	Two 16-bit B2A in line 12	-	88	332
	16-bit A2B in line 15	-	146	2267
	SecFprUrsh in Algorithm 9	-	1112	3214
	SecFprNorm64 in Algorithm 10	-	2846	7270
	SecFPR in Algorithm 11	-	3362	10813
	#randombytes	-	849	2691

Table 3: Time (in microseconds) for signing a message on Intel-Core i9-12900KF CPU.

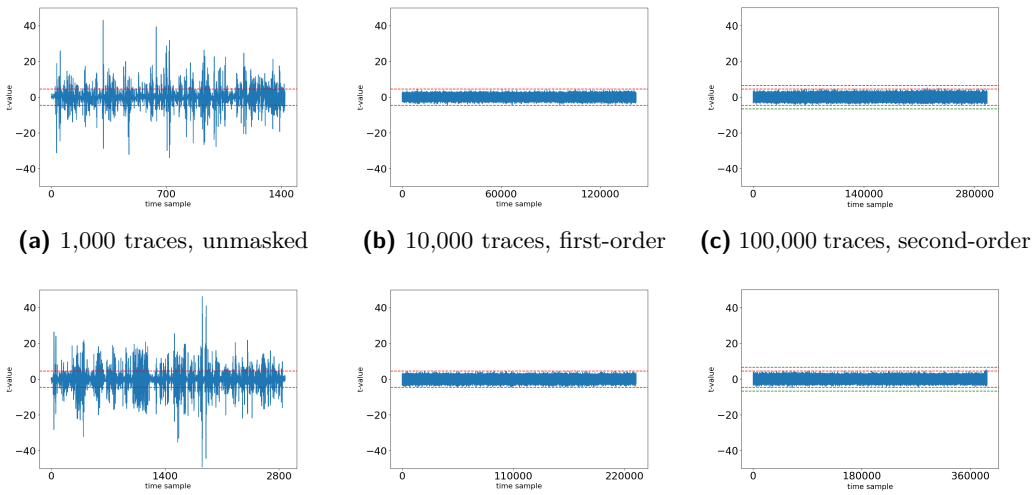
Security Level	Unmasked	2 Shares	3 Shares
Falcon-512	246.56	1905.55	6137.25
Falcon-1024	501.62	3819.76	12287.29

24.9× for 3 shares (about 6.1 ms for Falcon-512).

5.2 Security Evaluation

For practical security evaluation, we conducted the leakage assessment via the TVLA methodology, which we introduced in Section 2.5. We performed TVLA on our first-order (2 shares) and second-order (3 shares) masked floating-point number multiplication and addition, comparing them with the unmasked ones. Figure 8 shows our results. From left to right are the t-value statistics for unmasked, first-order, and second-order implementation. A threshold of ± 4.5 is provided as red dotted lines, while for second-order traces, we offer green dotted lines as the recommended threshold in [DZD+17]. For multiplication, the traces have a length of 295727, and we set the threshold to ± 6.628; while for addition with traces length of 387764, the threshold is set to ± 6.668.

For the unmasked function, we measured a total of 1,000 traces, and it turns out that almost every point exceeds the threshold. The results are improved for first-order implementation, but still some points exceed the threshold. By rewriting part of the code by assembly, adding redundant operations, and rearranging the order of independent instructions, every point is within the threshold in 10,000 traces. However, for tests with 100,000 traces, some values crossed the thresholds. It shows that the device may implicitly leak first-order information in the 2-shared implementation. We discuss this problem more thoroughly in Section 5.3. For second-order implementation, almost all the points are within the threshold ± 4.5 in 100,000 traces, and all the points pass the test with the adapted thresholds. We see that the second-order implementation eliminates the leakages that appeared in the first-order design.



(a) 1,000 traces, unmasked **(b)** 10,000 traces, first-order **(c)** 100,000 traces, second-order

(d) 1,000 traces, unmasked **(e)** 10,000 traces, first-order **(f)** 100,000 traces, second-order

Figure 8: TVLA results of floating-point number multiplication (top row) and addition (bottom row) for (a)(d) unmasked implementation with 1,000 traces, (b)(e) first-order (2 shares) mask with 10,000 traces, and (c)(f) second-order (3 shares) mask with 100,000 traces

5.3 Discussion about the Leakage of the First-Order Design

The first-order implementation without further optimization still showed leakage with 10,000 power consumption traces in our experiment. Similar results were also found in previous works [BGR+21, BC22]. On the other hand, our second-order implementation can pass the TVLA test of 100,000 traces, which indicates that leakage in the first-order design might be caused by unexpected equipment behavior. As pointed out and organized in [GD23], probing security cannot capture the physical defaults of devices. Glitches and transition-based leakage concerning the Hamming distance between two consecutive values written in a memory cell can cause power consumption related to the unmasked secret.

The leakage in [BGR+21, BC22] was eliminated or mitigated through assembly optimization. In our experiment, we used a defensive approach similar to that in [BC22], such as adding a dummy load and store operation before and after each consecutive share-wise operation where leakage appeared in the first-order TVLA result. We also found that shift operations could induce leakage, so we inserted redundant shift operations around the true ones. Besides, we separated dependent instructions to avoid potential leakage from Hamming distance or hidden buffer. With these revisions, we removed all the high values in the tests with 10,000 traces. Nevertheless, our first-order implementation failed to pass the test in 100,000 traces.

Two approaches can be taken to improve the result. The first one is a thorough assembly rewriting. With programs written in assembly, one can manipulate each register to avoid the potential transition-based leakage caused by compiling from a high-level language. However, hidden registers and other memory units in the processor [GOP22, MPW21] cannot be directly accessed. They may still induce transition-based leakage or even recombination of shares. Another concern with this strategy is that the design can vary for different devices. For example, we found different leakage patterns and locations when executing the same optimization method on STM32F303 and STM32F415 target boards; by contrast, the TVLA results of the second-order implementation were similar on both. The second approach is a secure design in the robust probing model [FGP+18, MMSS19],

which considers typical physical defaults like glitches. Unfortunately, a glitch-resistant model-based design of SecAdd, A2B, and B2A gadgets is still unknown to the best of our knowledge, and the procedure can require more than two shares and reduce efficiency.

6 Conclusion

In this work, we provide a masking scheme for Falcon's floating-point number multiplication and addition. To round the mantissa and compute the sticky bit efficiently, we design a masked nonzero check algorithm to find whether a shared value is nonzero, which can also be used to check the equality of two values and normalize a number. In addition, a masked right-shift and a masked normalization algorithm are proposed to add two floating-point numbers securely. The former can securely shift a value by some arithmetic shares while preserving the sticky bit, and the latter helps normalize a 64-bit number to the specific range $[2^{63}, 2^{64})$.

We provide formal proofs to show our design is secure in the t-probing model. Specifically, we apply the t-NI and t-SNI definitions and prove the security of our gadgets based on simulation. In terms of practical leakage on board, we conducted the leakage assessment experiments via TVLA. The first-order countermeasure with part of the functions rewritten by assembly, adding redundant operations, and rearrangement of execution orders can pass the test in 10,000 traces. With second-order masking, there is no significant leakage in 100,000 measured traces.

For performance evaluation, we compare cycle counts among unmasked, first-order, and second-order implementations on an Arm Cortex-M4 core. To achieve complete masking for the procedures containing both Boolean and arithmetic operations, our algorithms call arithmetic-to-Boolean mask conversion gadgets A2B and Boolean-masked addition gadgets SecAdd several times. It turns out that they cause the most considerable overhead in our design. We also tested the speed on an Intel-Core CPU, and it shows that a complete signing process with our countermeasure can be finished within a few milliseconds.

Throughout the signing algorithm of Falcon, the attack and defense of the Gaussian sampler have been discussed. However, the pre-image vector computation and other parts of the fast Fourier sampler are still at risk of being attacked, even though no works on the fast Fourier sampler have been proposed. A complete masking of Falcon can be constructed by combining our works with a masked design of the sampler. With our implementations and evaluations of the masking scheme for Falcon, it can resist known attacks on the pre-image vector computation, allowing Falcon to be used more securely.

Acknowledgement

The authors would like to thank professor Ho-Lin Chen for his contributions through extensive discussions and insightful feedback on this work; Tzu-Hsien Chang, Yi-Lin Hung, and Yu-Cheng Su for their guidance in algorithm designs and implementations. The authors are also grateful to professor Bo-Yin Yang for the support of this research and the assistance in submission, as well as all the reviewers' efforts during the review process, which greatly improve this paper in all aspects.

References

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl,

Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM Press, October 2016.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 354–384. Springer, Heidelberg, April / May 2018.

[BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit: with application to lattice-based kems. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):553–588, Aug. 2022.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from Boolean to arithmetic masking. *IACR TCHES*, 2018(2):22–45, 2018. https://tches.iacr.org/index.php/TCHES/article/view/873.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First- and higher-order implementations. *IACR TCHES*, 2021(4):173–214, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9064.

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 323–345. Springer, Heidelberg, August 2016.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to Boolean masking with logarithmic complexity. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 130–149. Springer, Heidelberg, March 2015.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

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

[DZD+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2017.

[EFG+22] Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: A simpler, parallelizable, maskable variant of falcon. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 222–253. Springer, Heidelberg, May / June 2022.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and

Dongyan Xu, editors, *ACM CCS 2017*, pages 1857–1874. ACM Press, October / November 2017.

[Elg85] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[FBR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR TCHES*, 2022(1):414–460, 2022.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR TCHES*, 2018(3):89–120, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7270.

[GD23] John Gaspoz and Siemen Dhooghe. Threshold implementations in software: Micro-architectural leakages in algorithms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):155–179, Mar. 2023.

[GGJR+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*, volume 7, pages 115–136, 2011.

[GMRR22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR TCHES*, 2022(3):141–164, 2022.

[GOP22] Si Gao, Elisabeth Oswald, and Dan Page. Towards micro-architectural leakage simulators: Reverse engineering micro-architectural leakage features is practical. In *Advances in Cryptology – EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 – June 3, 2022, Proceedings, Part III*, page 284–311, Berlin, Heidelberg, 2022. Springer-Verlag.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206. ACM Press, May 2008.

[HKL+22] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Amber Sprenkels. First-order masked kyber on ARM cortex-M4. Cryptology ePrint Archive, Report 2022/058, 2022. https://eprint.iacr.org/2022/058.

[HPRR20] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous gaussian sampling: From inception to implementation. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 53–71. Springer, Heidelberg, 2020.

[Inc] NewAE Technology Inc. Chipwhisperer-pro (complete level 3 starter kit). https://store.newae.com/chipwhisperer-pro-complete-level-3-starter-kit/.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[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)*, pages 691–696, 2021.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium - efficient implementation and side-channel evaluation. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 344–362. Springer, Heidelberg, June 2019.

[MHS+19] Sarah McCarthy, James Howe, Neil Smyth, Seamus Brannigan, and Máire O'Neill. BEARZ attack FALCON: Implementation attacks with countermeasures on the FALCON signature scheme. Cryptology ePrint Archive, Report 2019/478, 2019. https://eprint.iacr.org/2019/478.

[MMSS19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited. *IACR TCHES*, 2019(2):256–292, 2019. https://tches.iacr.org/index.php/TCHES/article/view/7392.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MPW21] Ben Marshall, Dan Page, and James Webb. Miracle: Micro-architectural leakage evaluation: A study of micro-architectural power leakage across many devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):175–220, Nov. 2021.

[oSTa] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[oSTb] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongSwan's implementation of post-quantum signatures. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1843–1855. ACM Press, October / November 2017.

[PFH+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, oct 1997.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 534–564. Springer, Heidelberg, April 2019.

[ZLYW23] Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved power analysis attacks on falcon. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part IV*, volume 14007 of *LNCS*, pages 565–595. Springer, Heidelberg, April 2023.