IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 4, pp. 483–508. DOI:10.46586/tches.v2024.i4.483-508

Masking FALCON's Floating-Point Multiplication in Hardware

Emre Karabulut[1] and Aydin Aysu[1]

North Carolina State University, Raleigh, USA, ekarabu@ncsu.edu,aaysu@ncsu.edu

Abstract. Floating-point arithmetic is a cornerstone in a wide array of computational domains, and it recently became a building block for the FALCON post-quantum digital signature algorithm. As a consequence, the side-channel security of these operations became under scrutiny. Recent works unveiled the first side-channel attack specifically targeting floating-point multiplication to steal secret cryptographic keys. Despite these new attacks on floating point arithmetic, there is no secure hardware design for side-channel leakage to date. A concurrent work has applied masking of floating-point multiplication in software [CC24], but their empirical validation still demonstrated significant first-order leakages. This paper presents the first hardware masking scheme for floating-point multiplication to mitigate side-channel attacks. Our technique extends the cryptographic masking principles that split all intermediate computations into multiple, random shares while preserving the output functionality. Our innovation also provides a design-time configurable first-order masked multiplier gadget that carries out integer multiplication, which can support future designs. To that end, we propose new hardware gadgets including Integer Multiplier, Carry Calculator, Secure MUX, Zero Check, and Mantissa Selection, and we prove their security in the PINI model. Moreover, we validate the desired first-order side-channel security of our implementation on a Sakura-X FPGA board using 10 million measurements. We explore the design space with different architectural choices to trade-off performance for the area. Our implementation results show that masking overhead ranges between 5.42×-43.31× in the area and 2×-440× in throughput.

Keywords: Masking · Floating-point · Side-channel Analysis · FALCON

1 Introduction

Floating-point multiplication, despite its computational expense, remains a crucial operation in numerous computing applications. Floating-point multiplication is not just a convenience; it is necessary for achieving accuracy and fidelity in AI/ML applications. On the cryptographic front, the new NIST-approved quantum-secure digital signature standard (FALCON) performs its polynomial multiplication in the FFT domain, requiring floating-point multiplications [PFH+20].

Floating-point multiplication, however, might be vulnerable to side-channel attacks. Such attacks pose a significant risk as they can inadvertently leak sensitive information processed during floating-point arithmetic operations, including secret keys in cryptographic algorithms [KA21]. For example, differential power analysis (DPA) on floating-point multiplication can extract the operands with just 1,000 trials and leak the secret key in the FALCON algorithm [KA21]. Subsequent improvements to this attack technique have further reduced the complexity of key recovery, emphasizing the need for robust side-channel protection mechanisms for FALCON [GMRR22]. Such attacks highlight

Licensed under Creative Commons License CC-BY 4.0.
Received: 2024-04-15 Accepted: 2024-06-15 Published: 2024-09-05

the need for side-channel protection for FALCON, one of the four finalists of NIST's standardization.

Masking is one such technique that can address side-channel leakage and is a popular choice for protecting cryptography hardware [CGF21]. Therefore, previous research efforts introduced various masking schemes designed to protect NIST's post-quantum cryptography standard implementations. This includes proposed masking schemes specifically for CRYSTALS-Kyber [BGR+21, FVBR+21a, HKL+22] and countermeasures developed for CRYSTALS-Dilithium [MGTF19, ABC+22, CGTZ23].

Despite these emerging attacks and aforementioned efforts of masking against side-channel leakage, floating-point multiplication has no implemented defense to date. FALCON, therefore, stands out as a candidate within NIST's finalist that lacks a side-channel protection mechanism. A concurrent work has recently attempted to provide a first-order software masking for FALCON, yet the practical demonstration has shown first-order leakage after 100,000 measurements [CC24], which was attributed but not explicitly substantiated, to potential micro-architectural leakages. A secure extension of masking schemes to floating-point multiplication is non-trivial. This complexity arises from arithmetic operations with large operands interfacing with Boolean operations.

In this work, we aim to bridge this existing gap of side-channel secure hardware for floating-point multiplication. We set two principle goals to develop a hardware-centric solution to mitigate the vulnerabilities associated with floating-point multiplication against side-channel attacks. First, our goal is to introduce a *secure* first-order hardware masking scheme for FALCON's floating-point multiplication, aiming to protect both operands and their associated computations from potential threats. Our foremost goal is to design and implement a solution that does not have first-order leakages either in practice or in theory. Our second goal is to expose the design challenges, given this is the first hardware masking of floating point multiplication, and to explore the design space with novel architectural decisions that can lead to different area-performance trade-offs.

We claim the following contributions that stand out in terms of our approach and its application:

- We introduce the first-ever hardware masking technique tailored for floating-point multiplication. This hardware is the first hardware-based defense against the DPA attack on FALCON.
- We introduce new hardware gadgets including Integer Multiplication, Carry Calculator, Secure MUX, Zero Check, and Mantissa Selection, and we prove their security in the PINI model.
- We explore the design space of hardware architectures and propose three different solutions depending on the parallelization level.
- We conduct the Test Vector Leakage Assessment (TVLA) experiments using 10 million traces to demonstrate the effectiveness of our countermeasure and quantify the overhead of masking. The results show that we have achieved the first solution that does not leak first-order leakages in practice.
- We introduce a new fixed-vs-fixed TVLA test setting specifically configured to evaluate corner cases in floating-point mantissa multiplication.

The rest of the paper is organized as follows. Section 2 provides the basic background information on masking, reviews previously masked gadget multiplications, explains the used leakage detection method, and breaks down FALCON's floating-point multiplication algorithm. In Section 3, we explore the challenges associated with masking mantissa multiplication and detail our hardware design, covering the proposed masked gadgets, operational sub-modules, and our overarching implementation approach. Section 5 presents our implementation results, including the hardware resource utilization, performance analysis, and our solution's side-channel evaluation. In Section 6, we provide a discussion

on the limitations of our work and the evaluation method. Finally, we conclude our paper and define possible future research directions in Section 7.

2 Preliminaries

Before we describe our novel ideas and applied techniques, we first introduce the relevant background in this section.

2.1 Masking

Masking is an effective countermeasure against side-channel attacks. The principle of masking is based on secure multi-party computation and secret sharing [ISW03]. A masked scheme splits sensitive variables into multiple randomized shares and then operates on these shares to prevent correlating physical side-channel information with the original secret variables. The security order of the masked scheme is determined with 'd'. The d-order masking splits the secret x into $d+1$ randomized shares $x_0, x_1, ..., x_d$. These shares satisfy the equation $x = \oplus_i x_i \bmod p$, ensuring that the secret x is reconstructed through $\oplus$ operation under modulo p condition. This randomized splitting ensures that each share (x_i) is statistically independent of the secret x; hence, this splitting is performed with random r, varying from 0 to $p-1$. When p is larger than 2, the scheme is referred to as arithmetic masking, where $\oplus$ corresponds to arithmetic addition over modulo p. Otherwise, it is called Boolean masking and $\oplus$ becomes exclusive-OR. It is more efficient to mask arithmetic operations using arithmetic masking and Boolean operations with Boolean masking [DMRB18]. Several masking schemes have been proposed in the literature as well as mathematical models to prove their effectiveness [BBD+16, CS20].

2.2 Composability and Probing Models

Security proofs for large designs is often unfeasible, leading to a common approach where security is proven theoretically for smaller circuits, which then serve as building blocks for more complex designs. These smaller subcircuits are known as "gadgets". A gadget is represented as a Directed Acyclic Graph (DAG) $G = \{V, E\}$ where V and E denote the set of vertices and edges, respectively. In the circuit concept, vertices represent combinatorial gates, while the edges metaphorically stand for wires, forging connections between logic elements. The inputs of a gadget are denoted by I, and its outputs are represented by O.

Combining individually secure gadgets does not necessarily result in a secure circuit. The term "composability" describes a gadget's ability to retain its side-channel security when incorporated into larger circuits and interacting with other gadgets or circuits. In order to analyze the composability of gadgets, several theoretical models have been proposed including Non-Interference (NI) [BBD+16], Strong Non-Interference (SNI) [BBD+16], Multiple-Output Strong Non-Interference (MO-SNI) [CS20], and PINI [CS20].

Simulatability [BBP+16]. Simulation is a fundamental technique in probing models used for the theoretical security proof of a gadget. In this process, a gadget is subjected to a simulation under an adversary model where an attacker is assumed to have knowledge of 't' wires (probes). Upon knowing these shares, the attacker attempts to simulate a subset I of the input shares of G. The gadget is considered secure if the attacker can correctly execute the simulation without necessitating additional probes given by the rules of the simulation. This assessment is based on the premise that the simulated shares should exhibit statistical independence from the probed shares. Therefore, probing specified shares does not offer any advantage to the attacker, as their ability to simulate shares remains equivalent to an individual who does not have probing knowledge. Since shares

are randomized, both attackers with probing knowledge and those without have an equal probability of accurately guessing the simulated shares.

There are different types of adversary models and therefore the rule of simulation varies based on the definition of adversary models. Nonetheless, we can formulate a generalized term for security proof applicable across these models: a gadget is deemed probing-secure if, and only if, the selected probe—defined within the constraints of the adversary model—can be accurately simulated without access to any shares other than the ones model specifies.

Probe-Isolating Non-Interference [CS20]. *"Given a gadget G, let I be a set of at most t_1 probes on its internal wires and O a set of probes on its output shares. Let A be the set of the share indexes of the shares in O, and $t_2 = |A|$. Let I and O be chosen such that $t_1 + t_2 \leq t$. The gadget G is t-PINI iff for all I and O there exists a set of at most t_1 share indexes B such that observations corresponding to I and O can be simulated using only the shares with indexes $A \cup B$ of each input sharing."*

2.3 Hardware Private Circuit (HPC) gadget

HPC1 and HPC2 are utilized to implement 2-input composable AND gadgets in line with the PINI concept within the glitch-extended probing model [CGLS20]. Here, we summarize them and refer interested readers to the original paper [CGLS20]. Notably, HPC1 combines a Domain-oriented masked AND gate (DOM-AND) with a refresh gadget, wherein one input sharing of the DOM-AND is refreshed with randomness. Conversely, HPC2 offers an alternative architecture for a 2-input AND gate with less randomness requirement. Although HPC2 needs less fresh randomness compared to HPC1, it consumes more area due to its additional logical operations. Our design strategy is to provide an area-optimized masked design and thereby we use HPC1 architecture over HPC2 in our gadget designs. Since HPC1 AND gate is PINI secure [CGLS20], the composition of HPC1 AND gate and other PINI gadgets are PINI. Thus, we do not provide additional proof for those gadgets. Moreover, HPC architecture can also be applied in arithmetic masking. In this context, the 2-input AND gates are replaced with p-bit multipliers, while the 2-input XOR gates are substituted with p-bit adders/subtractors. These operations are performed over modulo p rather than modulo 2, extending the applicability of HPC gadgets to arithmetic operations.

2.4 Test Vector Leakage Assessment

TVLA [SM16] is an effective and commonly used technique for leakage detection. It identifies statistically significant differences between the means of two groups of side-channel measurements corresponding to different inputs, using the t-score defined as:

$$t = \frac{\mu_1 - \mu_2}{\sqrt{\frac{\sigma_1^2}{N_1} + \frac{\sigma_2^2}{N_2}}} \tag{1}$$

where μ_i, σ_i and N_i are the mean, standard deviation, and number of traces respectively, of group $i \in \{0, 1\}$. $|t| > 4.5$ signifies that the two groups are statistically different with high confidence, indicating the presence of leakage. There are different ways to apply the TVLA test. The first-order TVLA involves applying an ordinary t-test to each sample point of the traces, which is known as a first-order univariate test. Second-order univariate TVLA requires preprocessing the traces by making each trace mean-free and then squaring each sample point. Following this preprocessing step, the same procedure as the first-order univariate test is applied on the preprocessed traces to detect leakage at the second-order.

Algorithm 1 FALCON Floating-point Multiplication Algorithm [PFH+20]

Input: two 64-bit floating-point numbers x ($\{s_x, e_x, m_x\}$) and y ($\{s_y, e_y, m_y\}$)
Output: a raw floating-point multiplication product xy

1: $z \leftarrow m_x \times m_y$
2: $zu \leftarrow z >> 50$
3: $zl \leftarrow z \wedge (2^{50} - 1)$
4: $zu \leftarrow zu \vee (zl \mathrel{!}= 0)$
5: $zv \leftarrow (zu >> 1) \vee (zu \wedge 1)$
6: $w \leftarrow z >> 55$
7: **if** $(w == 0)$ **then**
8: $\quad zu \leftarrow zv$
9: **end if**
10: **if** $(e_x == 0 \vee e_y == 0)$ **then**
11: $\quad zu \leftarrow 0$
12: **end if**
13: $e \leftarrow e_x + e_y - 2100 + w$
14: $s \leftarrow s_x \oplus s_y$
15: **return** $xy = (s, e, zu)$

2.5 Floating-point Multiplication in FALCON

FALCON executes its polynomial multiplication in the FFT domain. Given that the FFT domain mandates the coefficients of FALCON polynomials to be represented as complex numbers, floating-point operations become necessary. Therefore, FALCON's reference implementation works with emulated floating-point operations, and floating-point multiplication is one of the most crucial parts of FALCON implementation. This emulation follows the IEEE-754 standard for double precision. It first multiplies two operands with the IEEE-754 standard and then performs a rounding operation.

Algorithm 1 presents FALCON's floating-point multiplication pseudo-code. The algorithm begins with receiving the two 64-bit floating-point numbers, x and y. Each of these numbers has a sign bit (s_x and s_y), an 11-bit exponent (e_x and e_y), and a 53-bit mantissa (m_x and m_y). The first operation is to multiply these two mantissa values and produce a raw 106-bit product. This product undergoes rounding to compress it to 56 bits. This rounding is an OR operation between the least significant bit (LSB) 50 bit of this product (zl) and its LSB 51st-bit (see Step 5 in Algorithm 1). Then, there is a conditional right-shift operation. This condition depends on the most significant bit (MSB) of this raw product (w). If this MSB is set to 1, the resulting 56-bit product is right-shifted, and its least significant bit undergoes an OR operation with the second bit. The last operation on the mantissa is corrective actions for zeros. if either of the exponents is zero, the mantissa is set to 0. Otherwise, the obtained mantissa is preserved.

The sign-bit and the exponent additions are straightforward. The exponents of the two operands are added together with an offset of -2100 and then increased by 0 or 1 depending on the MSB of mantissa raw multiplication (w). For the sign-bit, the MSB bits of x and y are simply XORed to determine the sign of the result. Finally, the sign bit, exponent, and the 55-bit mantissa are returned. Since the generated mantissa is 55-bit, FALCON implementation provides an additional step to round it to 53-bit: if the last three bits of the mantissa are either 011, 110, or 111, the algorithm discards the last two bits and adds 1 to the result. In all other scenarios, it simply discards the last two bits.

In this paper, our primary focus is on securing the mantissa multiplication, as it represents the most challenging aspect of floating-point multiplication. Masking the XOR operations for the sign-bit and the additions for the exponent is relatively straightforward, as these consist of basic linear operations such as XOR and addition. Although we have

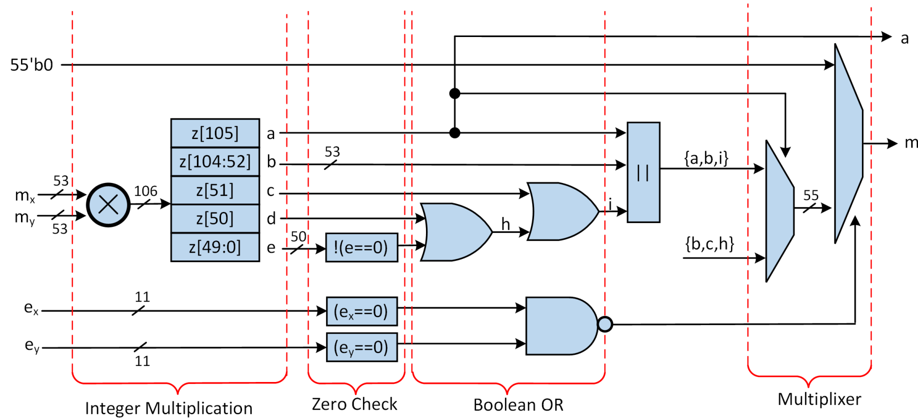

Figure 1: Baseline hardware diagram of the mantissa multiplication. The inputs are 53-bit mantissa operands (m_x and m_y) and 11-bit exponent operands (e_x and e_y), while the outputs are an exponent carry-bit (a) and the 55-bit unrounded mantissa (m).

implemented a one-bit masked XOR and an 11-bit masked adder for the sign-bit XORing and exponent addition operations, respectively, these components are not discussed in detail within this document. Additionally, the specifics of the rounding procedure are omitted from this discussion, as this aspect of the implementation is tailored specifically to FALCON emulation and does not readily extend to AI/ML models.

3 Hardware Design

Floating-point mantissa multiplication involves several core operations: integer multiplication, addition, logical shifts, and bitwise operations (AND, OR, XOR). Figure 1 displays the hardware design tailored for the mantissa multiplication in FALCON's floating-point operation. Note that some intermediate variable labels (a, b, c, e, h, i) are not explicitly denoted in Algorithm 1 because the hardware diagram shows further operational details.

Two 53-bit mantissa inputs, m_x and m_y, are processed using an integer multiplier, represented by the '×' symbol. This results in a 106-bit product denoted as 'z', which is subsequently segmented into different bit-lengths (a, b, c, d, e). The "Zero Check" phase evaluates two 11-bit exponent inputs (e_x, e_y) and the LSB 50 bits of z. The outputs from this phase are the direct input to the "Booleans" stage. Here, a NAND operation checks if either e_x or e_y is zero. If either exponent is zero, the corresponding mantissa is set to zero. Two additional OR operations produce intermediate values: h and i, which inform the creation of two potential mantissas $(a, b, i$ and $b, c, h)$.

These generated mantissa candidates are selected in "Mantissa Selection" step based on the following criteria. If the MSB of 'z' is 1, the MSB 56 bits of the product are right-shifted (a, b, c), with the least significant bit (c) undergoing an OR operation with h. If the MSB is 0, then b, c, h becomes the mantissa. If any of the exponents is zero, the mantissa defaults to zero. The finalized design emits two principal outputs: the 55-bit unrounded mantissa 'm' and a carry-bit 'a', which will be used in exponent addition. Our implementation target is a Kintex-7 FPGA. To make our solution efficient on this target, we propose several circuit-level optimizations such as efficient use of multiplier blocks in Xilinx FPGAs. But our optimizations are not fundamentally limited to Xilinx FPGAs as the same concepts can be extended (or resized) for other FPGAs or for ASIC designs.

3.1 Masking Challenges of Mantissa Multiplication

The mantissa multiplication is a challenging operation because it mixes arithmetic and Boolean operations. An efficient implementation of mantissa multiplication would need to mask multiplication with arithmetically masked gadgets and use Boolean-masked gadgets for logical and bitwise operations. Due to the inherent interplay between arithmetic and Boolean operations, conversion gadgets—specifically arithmetic to Boolean (A2B) and Boolean to arithmetic (B2A)—become essential [Gou01, BC22, LZP+24].

A further notable challenge arises from the operand sizes in integer multiplication and addition, which surpass the capacity of DSP48E1—the standard Xilinx Digital Signal Processing hardmacro circuit. A DSP48E1 block is equipped to handle multiplications of up to signed 18-bit with 25-bit variables or unsigned 17-bit with 24-bit variables [Xil18]. As a consequence, executing large integer multiplications necessitates the cascaded use of multiple DSP48E1 blocks. For instance, a baseline mantissa multiplication, denoted as $m_x * m_y$ in HDL, demands the use of 12 cascaded DSP48E1 blocks.

There are two challenges in this multiplication. The first challenge is the extension of integer arithmetic to the masking scheme and its effect on the DSP48E1 block utilization. The integer multiplication in the mantissa is not performed over a modulo field. Masking does not support direct extension to integer arithmetic and requires modulo arithmetic, the operand inputs and the output must be in the same field. Since the multiplication output is 106-bit, input operands needs to be correspondingly within the field $\mathbb{F}_{(2^{106})-1}$. Consequently, the multiplication operation now requires the 35 DSP48E1 blocks as the input operands are 106 bits rather than 53 bits. Moreover, a masked arithmetic multiplier demands roughly quadruple the multiplier components compared to a baseline design [GMK16], which results in the utilization of 140 DSP48E1 blocks. Such expansive multiplier circuits inevitably require tailored optimization strategies to address the long critical paths. Furthermore, the multiplication operation adds significant overhead on randomness since the processed operands are large and the latency is short.

The second challenge is due to glitches [NRR06], which are the physical flaws that occur when the secret shares fail to propagate uniformly during execution. Since the direct masked mantissa multiplication cascades several DSP48E1 blocks, this implementation is inherently susceptible to glitches, and resolving would lead to excessive register utilization.

3.2 Masked Integer Multiplication

We propose an integer multiplier circuit that tackles the described issues in the integer multiplication of floating-point multiplication. Our circuit breaks down large multiplications, ensuring efficient utilization of DSP resources. One of the standout features of our design is its versatility. This integer multiplication unit stands out as the first parametric and masked integer multiplier. As for the randomness throughput, the design demands a consistent 32-bit randomness per cycle, making the randomness need consistent irrespective of operand size. Although the proposed design's ability is beyond the mantissa integer multiplication, we configure the design to perform the mantissa multiplication, which is 53-bit by 53-bit. Figure 2 outlines our masked hardware solution, illustrating its four core components: the Multiplier, the Carry Calculator, the Secure MUX, and the Stage Adder.

The Multiplier operates as the initial computation unit, processing the mantissa in 8-bit byte chunks and storing results in two BRAMs. Each mantissa has 7 chunks thereby there are 49 partial products. Notably, we do not convert these arithmetic partial products to the Boolean domain to avoid resource-intensive A2B conversions. However, summing these partial products necessitates transferring the multiplication products from a smaller modulo field to a larger modulo field, leading to a carry-bit issue. We address this carry-bit issue using the Carry Calculator, Stage Adder, and Secure MUX circuits. The Carry Calculator detects carry-bit errors induced during this modulo field transfer. The Secure

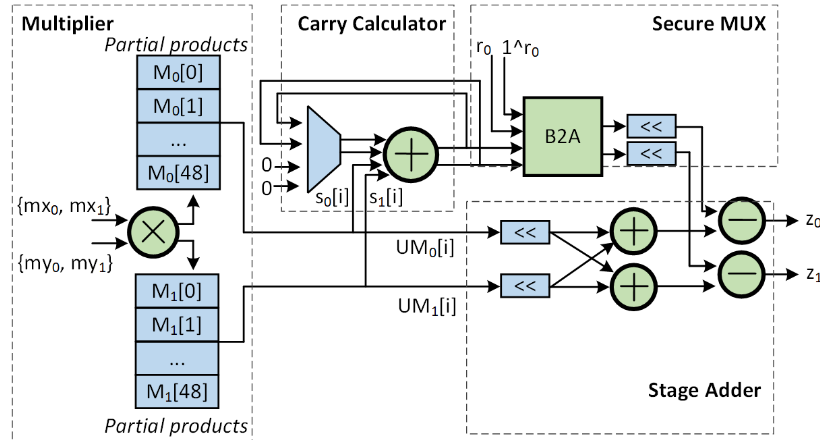

Figure 2: Block diagram of the integer multiplication in floating-point multiplication. Green elements represent the masked gadgets, whereas blue elements denote other logic components. The architecture comprises four pivotal blocks: (i) Multiplier, which processes the mantissa in 8-bit byte chunks and stores results in two BRAMs; (ii) a carry calculator, analyzing each chunk's result bit-by-bit; (iii) a secure MUX, detecting a carry-bit, (iv) and a stage adder for the masked cumulative addition of the product. The outputs are integer multiplication products z_0 and z_1.

MUX checks and handles the carry-bit effects by rebalancing the shares for each partial product. Finally, the Stage Adder aggregates the outputs of the multiplication, applying the necessary left-shift operations. It then consolidates these results and adjusts for any carry-bits by integrating outputs from the Secure MUX. Collectively, these modules are instrumental in the precise realization of integer multiplication, effectively navigating its complexities. We next describe how to mask each component in the following subsections and provide an example to illustrate how the carry-bit issue arises and is addressed.

Proof. This gadget is the composition of PINI gadgets that are proven with Proofs given in Section 3.2.1, Section 3.2.2, Section 3.2.3, and Section 3.2.4; therefore, this gadget is PINI secure. □

3.2.1 Masked Multiplier

Our masked Multiplier unit aims a balance between security and DSP resource optimization. Instead of performing mantissa multiplications at once, our design divides the mantissa operand into seven distinct chunks, where each chunk comprises 8-bit values in the integer arithmetic domain, with the exception of the last chunk, which is 5-bit. By adopting this approach, the multiplication process is broken down into 49 steps, as each of the seven mantissa chunks is multiplied by every chunk of the opposing mantissa. This approach paves the way for our design to operate within a smaller modulo field.

This choice relies on the intrinsic abilities of the DSP48E1, which handles multiplications up to 17-by-24 bits. Given that modulo arithmetic dictates both the input and output of an operation to remain in the same field, a solitary DSP48E1 can effectively execute multiplication operations within $\mathbb{F}_{(2^{17})-1}$. Although a 17-bit configuration might appear optimal at first glance, its nature poses challenges for crafting customizable hardware, motivating us to choose the nearest even number, 16.

Although our multiplier can work with 16-bit operands in modulo arithmetic, the operands can be a maximum of 8-bit in the integer arithmetic domain. Multiplications exceeding 8 bits could induce results overflowing the 16-bit field. Therefore, our multiplier unit is precisely designed to perform 8-bit integer multiplications, albeit with operands

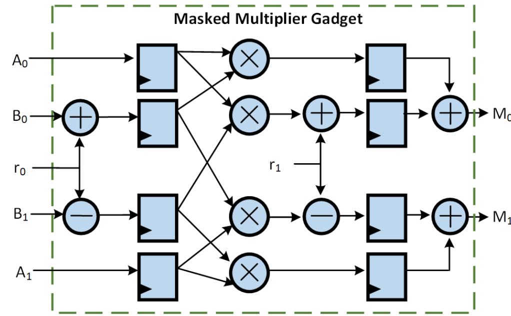

Figure 3: The first-order masked arithmetic multiplier based on HPC1 architecture. This gadget multiplies 8-bit secrets (A, B) in $\mathbb{F}_{(2^{16})-1}$. Therefore, the gadget operands are 16-bit including the random numbers (r_0, r_1).

randomized within a 16-bit. As a result, a masked multiplier works with 16-bit operands that are indeed the secret shares of 8-bit chunks of mantissa.

Figure 3 shows our multiplication gadget designed based on HPC1 architecture [CGLS20]. Therefore, it satisfies the PINI probing security requirement. The design's functionality unfolds in two distinct phases: the refreshing phase followed by the domain-oriented multiplier phase. To sustain the security, this gadget needs 32-bit randomness.

In the initial refreshing phase, shares of the second operand (B_0 and B_1) undergo a refreshment process provided by a 16-bit randomness (r_0). Post this step, all shares are synchronized within the first register blocks. During the domain-oriented multiplier phase, the randomized shares go through arithmetic multiplication, and then the independent domains use the second 16-bit randomness (r_1) to refresh the multiplication output. This usage of second randomness is called resharing and that allows the compressing of the 4 shares into 2 shares (M_0, M_1) by using the same randomness. Overall, the operation has 2-cycle latency and has throughput of multiplication per cycle.

Proof. Due to the similarities between the HPC1 AND gate and the proposed multiplier, we can rely on the proof from [CGLS20]. We build a PINI simulator where the set of probes is P, input and output shares are I and O and their coordinates are denoted with B. Any outputs $S_o \subset O$ that can simulate the set of probes $S_p \subset P$ with two conditions: (i) S_p has connections coordinates $i \subset B$ that reaches $S_i \subset I$, (ii) number of used probes and output in the simulation cannot be larger than $d+1$, which is 2 in our case as the gadget is first order masked. Since all possible S_p are statically independent of their primary input shares within $i \subset B$ coordinates, they are simulatable and the gadget is PINI secure. □

To prevent potential transition-based masking flaws, our Multiplier design strategically partitions the multiplication outputs, storing them in two separate BRAMs. Following this segregation, the next procedural step entails the summation of these outputs to derive the final product of the 53-bit multiplication. However, our Multiplier design works with a smaller modulo field $\mathbb{F}_{(2^{16})-1}$, adding up these chunked multiplication results in $\mathbb{F}_{(2^{106})-1}$ might give the incorrect output. Assume that a secret s has two shares s_0 and s_1, randomized in $\mathbb{F}_{(2^{16})-1}$ and this shares thereby sastify the equation $s = s_0 + s_1 \bmod 2^{16}$. To obtain the same result within $\mathbb{F}_{(2^{106})-1}$, the equation is updated to $s = s_0 + s_1 - c \times 2^{16} \bmod 2^{106}$ where 'c' symbolizes the carry bit extracted from the MSB of raw $s_0 + s_1$ summation. This underlines the necessity for an auxiliary step in the multiplication output summation process: the calculation of the carry bit associated with paired shares. A significant challenge arises here: directly combining two shares to find out the carry bit stands against the main principle of masking schemes.

A plausible solution is to convert multiplication output to the Boolean domain and mask it in Boolean to avoid the carry-bit issue. However, the subsequent step requires

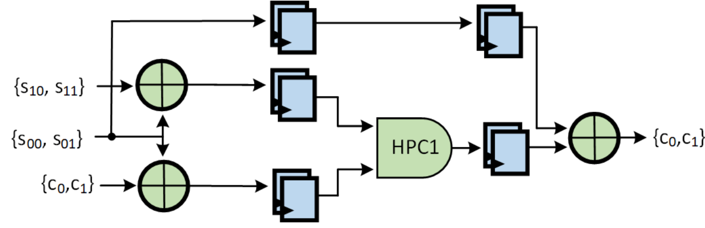

Figure 4: Hardware block diagram of the Carry Calculator. This gadget utilizes a single masked AND gate to compute the carry-bit.

adding all multiplication results to obtain the final multiplication output. Implementing this arithmetic addition in the Boolean domain is more costly given that it is more efficient to mask arithmetic operations using arithmetic masking and Boolean operations with Boolean masking [DMRB18]. Therefore, we do not convert multiplication output to the Boolean domain but just calculate the carry-bits in the Boolean domain.

3.2.2 Masked Carry Calculator

We propose a solution with a carry-bit calculator gadget, and this efficient design eliminates the need for resource-intensive A2B conversions. Figure 4 presents our carry-bit calculator gadget design. The proposed masked carry-bit calculator gadget works only with one PINI secure HPC1 AND gate gadget [CGLS20], thereby minimizing the proposed countermeasure's area overhead. The initial idea for this design came from methods used to convert arithmetic shares to Boolean shares [BC22]. However, our main goal is different: we want to keep the shares in the arithmetic domain and still calculate the carry-bit.

This design performs the functionality of a full adder with one HPC1 AND gate and 3 three masked XOR gates. A traditional full adder receives three one-bit operands and produces two outputs: the sum and $carry - out$. However, our gadget's main purpose is to calculate only the carry-bit and therefore it does not calculate sum bit but its carry-bit. The process starts with the LSBs of the share pairs s_0 and s_1 and they are in the form: $\{s_{00}, s_{01}\}$, $\{s_{10}, s_{11}\}$. For the first addition, the third operand of the full adder, $carry - in$, is set to 0. After this step, the resulting $carry - out$, represented as shares c_0 and c_1, are sent back to the gadget with the next bit from shares s_0 and s_1. This process continues until the final bit of shares s_0 and s_1 is used to determine the final values of c_0 and c_1.

Proof. The proposed carry calculator is a composition of HPC1 AND gadget and refreshed XORs. Since the refreshed XORs allow the propagation of a probe simulation while maintaining statistical independence, the carry-bit calculator is PINI secure. □

3.2.3 Masked Secure MUX

Our third main block is Secure MUX. This module serves a critical purpose – to ensure the carry-bit effect does not distort the expected results of our multiplication process, especially given the varying effects of carry-bits due to the chunk-based multiplication approach. We illustrate the varying effect with the following integer multiplication example. Given two 16-bit operands, V and C, if we break them into 8-bit chunks represented as $\{V1, V0\}$, and $\{C1, C0\}$. The multiplication of V and C can be executed with 4 chunk multiplications with the following calculation.

$$V0 \times C0 + (V1 \times C0) << 8 + (V0 \times C1) << 8 + (V1 \times C1) << 16 \tag{2}$$

Each carry-bit belonging to a chunk multiplication has a different influence on the result. For instance, the product $(V0 \times C1) << 8$ contributes to the 24-bit position with

Algorithm 2 Masked Secure MUX Algorithm

Input: a carry-bit c ($\{c_0, c_1\}$), a scalar k, and r_0, r_1 in $\mathbb{F}_{(2^{106})-1}$
Output: a carry-bit adjustment value adj ($\{adj_0, adj_1\}$) in $\mathbb{F}_{(2^{106})-1}$
1: $cext_0 \leftarrow Reg(c_0 \ << \ k)$
2: $cext_1 \leftarrow Reg(c_1 \ << \ k)$
3: $cBool_0 \leftarrow Reg(cext_0 \oplus r_0)$
4: $cBool_1 \leftarrow Reg(cext_1 \oplus r_0)$
5: $\{cArith_0, cArith_1\} \leftarrow B2A(cBool_0, cBool_1)$
6: $adj_0 \leftarrow cArith_0 - r_1$
7: $adj_1 \leftarrow cArith_1 + r_1$
8: **return** $adj(adj_0, adj_1)$

its carry-bit due to its shift factor, whereas $V0 \times C0$ directly affects the 16-bit position. These distinctions are vital because the carry-bit associated with each multiplication stage has a specific impact based on its positional value in the overall calculation. To address this potential discrepancy, we introduced the Secure MUX module. Its primary function is to check the carry-bit within the rules of masking and rebalance the shares if the multiplication results have a carry-bit effect. This adjustment ensures that the carry-bit effect is avoided, preserving the correctness of our calculations.

Algorithm 2 presents implementation details of the proposed Secure MUX. The Secure MUX operation starts by left-shifting the constant value carry-bit ($\{c_0, c_1\}$) by k times, where k is determined by the multiplication stage. As explained in the given example, for the ($V0 \times C1$) scenario, k is 24 because the resultant product has 24 significant bits. However, k is 16 for $V0 \times C0$ since there's no shift for this multiplication stage. The shifted value is refreshed with randomness r_0. Finally, the obtained value is then converted into the Arithmetic domain in $\mathbb{F}_{(2^{106})-1}$ with a B2A converter. This conversion ensures the outcome remains the same field where we keep the final multiplication output. Since our innovation does not include designing a novel B2A or A2B domain, we do not discuss any implementation details of these converters. Interested readers can refer to related papers on such converters [BC22, LZP+24].

Proof. Algorithm 2 is PINI secure if and only if B2A is PINI secure. The inputs of the Secure MUX gadget get refreshed and become statistically independent and the outputs of the B2A converter are also refreshed before being returned. Therefore, B2A PINI simulation can be propagated in both the input and the output directions of the gadget if and only if B2A is PINI secure. □

3.2.4 Masked Stage Adder

The final component of our integer multiplication system is the masked Stage Adder. This module undertakes the task of collating all the outputs from the multiplication process and adjusting the results in light of the carry-bit modifications. This module mainly has three sub-operations: left shift, addition, and subtraction. Similar to the Secure MUX module, the Stage Adder applies a left-shift operation to the multiplication output as exemplified in Equation 2. Subsequent to the left-shift operation, the outputs are aggregated. This entails adding together the shifted multiplication results to form a consolidated output.

In cases where a particular chunk multiplication results in a carry-bit, the final operation involves adjusting the aggregated output. This is achieved by subtracting the output derived from the Secure MUX module from the addition result. This ensures the overall computation reflects any changes or anomalies introduced due to carry-bits. By encapsulating these three operations, the Stage Adder ensures that the final result of the integer multiplication is both accurate and takes into consideration the intricate nuances

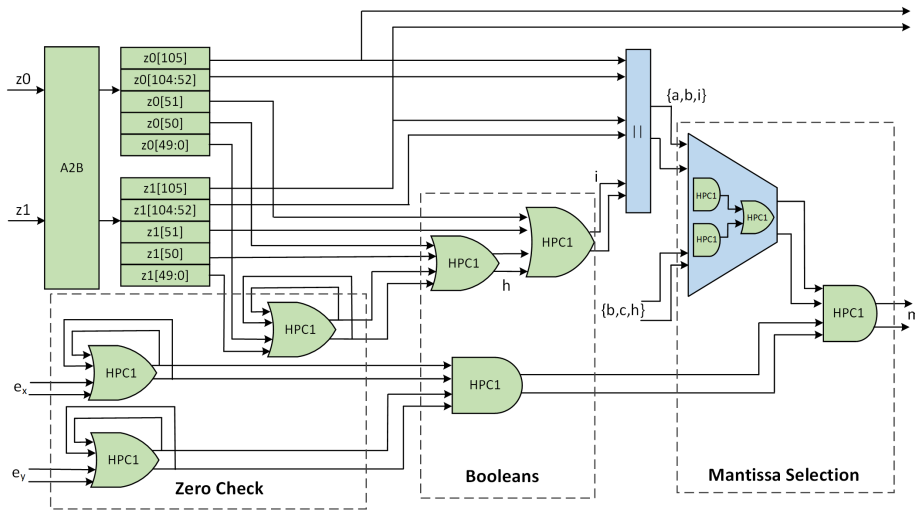

Figure 5: Baseline hardware design diagram of masked Boolean League. Green elements represent the masked gadgets, whereas blue elements denote other logic components. The inputs are 106-bit integer multiplication product operand (z_0 and z_1) and 11-bit exponent operands e_x and e_y, while the outputs are a carry-bit (a) and the 55-bit unrounded mantissa (m). Although the figure shows the masked AND gates have feedback, it does not represent combinatorial loops. Instead, these are HPC1-masked gadgets, which inherently prevent combinatorial loops through their architecture.

of the chunk-based multiplication and carry-bit adjustments.

Since the arithmetic masked addition or masked subtraction are well-known implementations and do not contribute to the novelty of the paper, we do not discuss the details of these operations. Note that Figure 2 has the same sign (+) in both Carry Calculator and Stage Adder modules. However, the addition performed in Carry Calculator is different and it represents the full adder gadget that we discussed in Section 3.2.2.

Proof. This gadget is composed of two paths each having refreshed linear operations that are randomized for each probe. Therefore, this gadget is PINI secure. □

3.3 Boolean League: Zero Check, Booleans, Mantissa Selection

After integer multiplication, the Mantissa multiplication incorporates additional post-processing steps to calculate the final result. Figure 1 illustrates the processes succeeding integer multiplication: zero checks, Booleans, and mantissa selection. These operations start with a 106-bit Integer multiplication product along with 2 exponents each having 11-bit and then generate the 55-bit result.

The zero check operation is the initial checkpoint where three operands independently are checked whether they are zero or not. The simplest method to perform this comparison is by performing a bitwise OR operation, which can easily be obtained from an AND gate. The next stage encompasses a series of logic operations, specifically two OR and one NAND. These operations are used to generate two mantissa candidates. After obtaining zero check and Boolean operations, the final step is the selection of generated mantissa candidates. This requires a multiplexer functionality. A multiplexer functionality can be implemented with an AND and an OR gate.

The series of operations described above underscores a pivotal decision: performing the remaining operations in the Boolean masking domain. The inherent Boolean nature of

Algorithm 3 Zero Check Algorithm

Input: n-bit Boolean sharings m ($\{m_0, m_1\}$), randomness r
Output: Boolean sharings x ($\{x_0, x_1\}$) such that $x = (m == 0)$

1: $x_0 \leftarrow m_0[0] \oplus r[0]$
2: $x_1 \leftarrow m_1[0] \oplus r[0]$
3: **for** i from 1 to n **do**
4: $y_0 \leftarrow Reg(m_0[i] \oplus r[2i+1])$
5: $y_1 \leftarrow Reg(m_1[i] \oplus r[2i+1])$
6: $x_0 \leftarrow Reg(x_0 \oplus r[2i])$
7: $x_1 \leftarrow Reg(x_1 \oplus r[2i])$
8: $x_0, x_1 \leftarrow SecOR(x_0, x_1, y_0, y_1)$
9: **end for**
10: **return** x_0, x_1

Algorithm 4 *SecOR* Algorithm

Input: Boolean sharings m ($\{m_0, m_1\}$), n ($\{n_0, n_1\}$)
Output: Boolean sharings p ($\{p_0, p_1\}$) such that $p = m \vee n$

1: $x_0 \leftarrow \neg m_0$
2: $x_1 \leftarrow m_1$
3: $y_0 \leftarrow \neg n_0$
4: $y_1 \leftarrow n_1$
5: $z_0, z_1 \leftarrow SecAND(x_0, x_1, y_0, y_1)$
6: **return** $z_0, \neg z_1$

these operations makes Boolean masking a logical choice, ensuring a synergy between the data and the domain. Hence, we convert the arithmetic multiplication product generated by the integer multiplication module to the Boolean masking domain. Then, the Boolean shares are processed within zero checks, Booleans, and mantissa selection modules.

Figure 5 depcits the masked operations of post-integer multiplication. First, arithmetic shares $z0$ and $z1$, which represent the 106-bit integer multiplication results, are converted to the Boolean domain with an A2B circuit. Second, they are segmented into different bit ranges, showing the granularity of the data being processed. Third, The exponents (e_x and e_y) and lower bits of integer multiplication product are fed into the zero-check module. The fourth step is the transition from the zero check module to the Booleans module. The Boolean operations help process and shape the multiplication results into a truncated form and create two mantissa candidates. A series of HPC1 operations assist in determining these mantissa candidates. Fifth, the flow ends with the Mantissa Selection module. Here, the mantissa candidates from the Booleans module are selected based on the MSB of the multiplication product and the zero check result of the exponents. This final step leads to the final mantissa result m and an auxiliary output a.

Proof. This gadget is the composition of PINI gadgets, therefore it is PINI secure. □

3.3.1 Masked Zero Check

During mantissa multiplication, the initial operation involves an integer multiplication that results in a 106-bit product. Rounding is then applied to its LSB 50 bits in order to truncate 106 bits to 56 bits. A preliminary assessment determines if these 50 bits are equal to zero. If they are not, the product's 51st bit undergoes an OR operation with 1, implying that the entire 50-bit section contributes a single bit to the final result. Furthermore, a parallel check is executed on the exponents (e_x, e_y). Here, the task is to determine if either exponent equals zero. Unlike the previous mechanism, the outcome here doesn't influence rounding. Instead, it functions as a corrective action. The identification of a zero among the operands flags an error in mantissa computation, making the mantissa's reset to zero.

To functionally carry out these zero-check processes, we introduce our masked zero-check gadget. Bitwise OR operation is a foundational mechanism in the zero-check process. We obtained this operational capability from the previously described masked HPC1 AND gate. Algorithm 4 illustrates our proposed method where the OR gate functionality is obtained from the masked AND gate ($secAND$) and three logical NOT operations

Algorithm 5 Mantissa 1-to-2 Multiplixer Algorithm

Input: n-bit Boolean sharings x ($\{x_0, x_1\}$), y ($\{x_0, x_1\}$)
Input: Boolean sharings of a select bit s ($\{s_0, s_1\}$)
Output: n-bit Boolean sharings z ($\{z_0, z_1\}$) such that $z = s \; ? \; x : y$
1: **for** i from 0 to n **do**
2: $t_0[i], t_1[i] \leftarrow SecAND(x_0, x_1, s_0, s_1)$
3: $w_0[i], w_1[i] \leftarrow SecAND(y_0, y_1, \neg s_0, s_1)$
4: $z_0[i], z_1[i] \leftarrow SecOR(t_0, t_1, w_0, w_1)$
5: **end for**
6: **return** z

(see Steps 1, 3, and 5). We show the implementation strategy of our zero-check gadget in Algorithm 3 by utilizing masked OR gate ($SecOR$). The algorithm starts with the initialization of x (x_0 and x_1) with the LSB of the operand m, which is subject to the zero check. Subsequently, a loop executes a series of logical OR operations with $SecOR$. In each iteration, one input to the OR operation is x, while the other draws from the subsequent bit of m. This iterative process continues until every bit of m is participated in an OR operation. In addition to the OR operation, there are also share refreshing steps between Steps 4 and 7 in order to simplify the probing model.

Proof. This gadget is the composition of PINI gadgets, therefore it is PINI secure. □

3.3.2 Masked Booleans

Following the zero check, the next computational phase encompasses Boolean operations. The integer multiplication product truncated to 56 bits after the zero check, combined with the exponent's zero-check outcomes, serves as inputs to this stage. As illustrated in Figure 5, the Booleans segment houses three masked logic gates.

The two HPC1 OR gates primarily facilitate the generation of two distinct mantissa candidates. The 51st bit undergoes a dual-stage OR operation: initially with the zero-check result of the integer multiplication product and subsequently with the 52nd bit (c_0 and c_1). The initial OR operation outputs shares of h (h_0 and h_1), while the second OR generates the shares of i (i_0 and i_1). These shares feed into two separate mantissa candidates. The first combines the shares of a_0, b_0, i_0 and a_1, b_1, i_1; the second combines the shares of b_0, c_0, h_0 and b_1, c_1, h_1. Within this framework, a stands for the MSB of the integer multiplication product, while b denotes the following 53-bit of a. In parallel, another Boolean operation is performed on the zero check results of the exponent with HPC1 AND gate. This operation indicates if either exponent is equal to zero.

Proof. This gadget is the composition of PINI gadgets, therefore it is PINI secure. □

3.3.3 Masked Mantissa Selection

Upon the successful completion of the zero check and Boolean operations, the subsequent task is to select the correct mantissa candidates. Within this context, three candidates emerge. The initial two originate from the integer multiplication product, while the last one is zero and selected only if one of the exponents matches zero. We introduce our masked Mantissa Selection design capable of executing the selection with one 1-to-2 multiplexer and an HPC1 AND gate.

Algorithm 5 provides the implementation methodology for 1-to-2 multiplexer. This algorithm is a masked multiplexer, ensuring that the selection between x and y based on the select bit s is done without revealing any other intermediate information. The algorithm first computes the AND operation of x and s to obtain t. Similarly, the same

step is applied on y but with the negation of s. As a consequence of this negation, the outcome of one of these AND operations inevitably stands at 0. The last operation is the computation of the OR operation between the results of the previous two steps (t and w), determining the values of $z0$ and $z1$. Figure 5 depicts the masked Mantissa Selection implementation details. In our implementation, the select-bit is the MSB of the integer multiplication, which is a. The initial two mantissa candidates are a, b, i and b, c, h. After this selection, the next selection is among the first selection output and the zero. Since an AND operation with any select bit and zero consistently yields zero, a 1-to-2 multiplexer becomes unnecessary as it makes the use of two AND gates and an OR gate redundant. Therefore, a single AND gate, connecting the exponent's zero check to the initial mantissa selection output, suffices for the subsequent selection.

Proof. This gadget is the composition of PINI gadgets, therefore it is PINI secure. □

4 Design Space Exploration with Novel Design Elements

This section delves into the comprehensive exploration of the design space to enhance the performance of masked floating-point mantissa multiplication. Specifically, we discuss our optimization strategies to overcome the bottlenecks identified in integer multiplication and the time-intensive processes within the zero check module. Subsequently, we introduce and detail two primary innovations that significantly improve efficiency: the parallelization of the masked carry calculator unit and the masked zero check module.

The initial design aims to minimize the DSP and LUT consumption at the expense of performance—referred to herein as the "low-area design". This configuration employed a single masked carry calculator unit calculating the carry bit for 49 partial products, each comprising two shares. The calculation of the carry-bit is the most time-consuming routine of the integer multiplication phase in the floating-point mantissa multiplication process. Another time-consuming operation is the zero check operation in masked Boolean League operation. This module checks if exponents (e_x and e_y) and 50 lower bits of integer multiplication product are zero. Both bottlenecks (zero check and carry calculator) emphasize the iterative execution flow's inefficiency in the low-area design.

Parallelization of the carry calculator. This optimization employs multiple instances of the carry calculator module, enabling parallelization and pipelining in integer multiplication. Our masked multiplier circuit breaks down the integer multiplication into 49 steps, generating 49 partial products, each with two shares. One carry calculator module needs 80 cycles to calculate the carry bit of one partial product pair (two 16-bit shares). As a result, the low-area design, having one carry calculator module, spends $3,920$ cycles for 49 partial products. Since these 49 products are independent, an alternative design, referred balanced design, can calculate the carry-bit of these products in parallel execution of 49 carry calculator module. This balanced design requires 80 cycles to calculate carry-bits of all 49 partial products. This significant enhancement not only accelerates the calculation process but also supports pipelining, further optimizing throughput.

Checking zeros in a tree structure. Our initial design choice utilizes one masked OR gadget to check if 50 lower bits of integer multiplication product is zero. This process inherently introduces latency due to the sequential passing of bits through four registers, including two within the masked OR gate. Given that the exponent shares (e_x and e_y) are only 11 bits each, the majority of latency (200 cycles) is spent to check the 50 lower bits of integer multiplication. To address this, we introduce the novel architecture that employs a tree structure to streamline the zero-check process for the 50 lower bits of integer multiplication. This tree comprises two layers: the first layer employs five masked OR

gadgets, each responsible for processing 10 bits of the 50-bit randomized shares, thereby generating five outputs with two shares each. The second layer employs a single masked OR gadget that iterates five times to execute a logical OR operation on the outputs from the first layer. This innovative approach significantly enhances performance, reducing the latency of the zero-check operation by a factor of 4.5 compared to the low-area design.

Input width adjustment and multiplier gadget parallelization. An additional optimization method involves modifying the input widths for our multiplier gadget, directly impacting the latency of the carry-bit calculation in the partial products. We call this design "high-performance design". By default, the carry-bit calculator processes each partial product, composed of two shares, within 80 clock cycles. This latency is inherently tied to the size of the partial product; reducing the bit length of each partial product can decrease the latency per partial product carry-bit calculation.

To maintain the multiplier gadget's overall latency while reducing the carry-bit calculation time, we propose the introduction of multiple instances of the multiplier gadget module. This approach, albeit at the expense of additional randomness requirements, enables the processing of smaller chunks in parallel, effectively reducing the carry-bit calculation latency. Concurrently, the Boolean League circuit, tasked with handling the masked operations, must be adapted to accommodate the increased speed of the input generation from the integer multiplication circuit. This design strategy ensures a more efficient, albeit more complex, pipeline for integer multiplication. Notably, this optimization does not merely affect the integer multiplication unit but extends its impact to the Boolean League circuit as well. The necessity to handle a higher volume of partial products with smaller sizes demands adjustments in both the multiplier gadget and the Boolean League circuit.

Area overhead and performance gain. Although providing an exact formula is challenging, we can describe a general relationship between parallelism, performance, and area overhead. The carry calculator circuit utilizes 211 LUTs, which accounts for approximately 3% of the entire design. A single carry calculator requires $3,920$ cycles to compute the carry bits for 49 partial products, which constitutes about 52% of the total execution latency. Thus, n carry calculators take roughly $3920/n$ cycles and cost about $n \times 211$ LUTs. A similar case applies to the zero-check process. However, the parallelism in the zero-check process comes with a binary-tree architecture, which requires $\lfloor \log_2(u) \rfloor$ stages, and each stage requires $(\log_2(k))^2$ masked OR gates, where u is the input bit length and k is the stage number. In addition to the masked OR gates, this technique needs a binary-tree architecture and its pipeline registers. Therefore, this technique reduces the zero-checking process from 200 to 44 cycle latency but increases LUT utilization from 293 to $1,405$ LUTs.

5 Implementation Results

This section provides details about the implementation details of our solution. This includes the main environmental setup, hardware resource allocation, and side-channel security evaluation. We implemented the proposed designs on the Sakura-X FPGA board that hosts a Xilinx Kintex-7 XC7K160T-1FBG676C for testing hardware designs. The FPGA EDA tool is Xilinx Vivado 2020.2 which provides an FPGA implementation flow infrastructure that covers the synthesis, placing, routing, bitstream generation, and FPGA programming. We used SystemVerilog to implement our solution at the HDL level. We did not use FPGA-dependent hardware primitive but we used KEEP_HIERARCHY implementation directive attribute in order to preserve the gadget architectures that we discussed in Section 3. We also did not set a specific synthesis and implementation strategy flag on the EDA tool. We used the default flag, which is 'Vivado Implementation Defaults'.

Table 1: Comparative analysis of hardware area utilization, performance, and countermeasure overhead among various design strategies and the prior work [CC24].

Design	LUT/FF DSP/BRAM	Cycle Count[a]	Area Overhead[b]	Perf. Overhead[c]	Max Freq.	1st Order Security
Unprotected	1,140/1,550 4/0	16	-	-	180 MHz	✗
Low-area Protected	6,183/8,916 4/0.5	7,049	5.42×	440×	208 MHz	✓
Balanced Protected	18,807/22,632 4/0.5	204	16.94×	12.75×	212 MHz	✓
High Perf. Protected	49,383/61,290 16/0.5	32	43.31×	2×	212 MHz	✓
SW-Protected [CC24]	-	3,772	-	12.24×[d]	-	✗

[a]: This is throughput cycle count per floating-point multiplication without including rounding.
[b]: Area Overhead refers to the increase in the number of LUTs used compared to the unprotected design.
[c]: Performance Overhead is the relative decrease in throughput compared to the unprotected design.
[d]: Performance Overhead for the software (SW) Protected design refers specifically to the decrease in throughput when compared to the corresponding unprotected software implementation.

5.1 Area, Performance and Overhead Results

Table 1 presents five distinct designs for floating-point multiplication: the unprotected, low-area protected, balanced protected, high-performance protected hardware designs, and the recent software-based masking design [CC24]. Each hardware design is evaluated based on the LUT and FF counts, DSP and BRAM utilization, throughput, area overhead, performance overhead, maximum frequency, and whether mitigates first-order side-channel leakage. The prior work, being software-based, does not report area metrics like memory footprint, hence we only listed their cycle count and performance overhead results. Since hardware and software designs run on different platforms, performance overhead is calculated on cycle count and excludes the frequency from this calculation.

The baseline, an unprotected design, utilizes minimal resources with a LUT/FF count of 1,140/1,550 and a DSP/BRAM count of 4/0. It achieves a throughput of one operation per 16 cycles, serving as the reference model for evaluating the overheads of protected configurations. The low-area protected configuration markedly increases the area overhead to 5.42× that of the unprotected design, consuming 6,183 LUTs and 8,916 FFs, with minimal DSP and BRAM use. This design incurs the highest performance overhead, reflecting its resource-constrained strategy. Also, our low-area protected design works with 1.15× higher frequency than the baseline design since we register the combinatorial logics in order to prevent glitches.

The balanced protected design offers a compromise between performance and area overhead. This configuration requires 18,807 LUTs and 22,632 FFs, however, the DSP and BRAM utilization is the same with low-area configuration, resulting in an area overhead of 16.94× the baseline. The throughput is optimized to one operation per 204 cycles, with a more manageable performance overhead of 12.75×. Prioritizing performance, the high-performance design is anticipated to exhibit the highest area overhead at 43.31×, with a LUT/FF utilization of 49,383/61,290 and an increased DSP count. It is expected to improve throughput to one operation per 32 cycles, potentially reducing the performance overhead to just 2× the baseline, thus offering a high-performance option. Notably, both the balanced and high-performance protected designs achieve the highest maximum frequency of 212 MHz. We provide these area and performance metrics for the high-performance design in an estimative capacity, as the design itself has not been implemented. FALCON has one hardware implementation designed with High-Level Synthesis (HLS) [SAW+23]. However, this implementation does not provide hardware utilization or performance metrics at the sub-module level. Therefore, we cannot project the area and performance overhead of our countermeasure on the full implementation of FALCON.

Lastly, the software-based design [CC24] does not utilize hardware resources like LUTs or DSPs, as indicated by the absence of these metrics. This prior work also does not provide a memory footprint and therefore does not have an area-overhead metric. It achieves a throughput of one operation per 3,772 cycles, with a performance overhead of 23 times compared to the unprotected software implementation. Although this design includes rounding operations, its computational contribution to throughput has been excluded from our reported performance results. Additionally, empirical validation has shown significant first-order leakages, indicating that this approach does not provide first-order security.

Breakdown of randomness overhead. Our low-area protected design is optimized to reduce the randomness overhead by requiring only 32 bits of fresh randomness per cycle. The design breaks down the large integer multiplication into 49 chunks and performs each chunk's multiplication with a 16-bit masked arithmetic multiplier based on the HPC1 architecture, which necessitates 32-bit of fresh randomness for each multiplication. The masked carry calculator, however, requires only 10-bit of fresh randomness. Consequently, the remaining 22 bits of fresh randomness are stored in a BRAM to be used later in the stage adder and masked secure MUX, which require 106 bits of randomness per cycle. A similar approach is employed for the Boolean League. Although the Zero Check modules require 12 bits of randomness, the remaining 20 bits of fresh randomness are stored in a BRAM for later use in the Mantissa Selection, which needs 330 bits of randomness.

5.2 Side-channel evaluation

We use the Sakura-X FPGA board for executing the hardware design for the side-channel evaluation. The FPGA board has a designated SMA port that provides the power drop across a shunt resistor of 1Ω on the main supply line. To minimize potential information loss stemming from aliasing across clock cycles, we operated the design at a deliberately low frequency (12 MHz). The Picoscope 3206D oscilloscope is used to monitor voltage fluctuations during the execution of implemented mantissa multiplication. The oscilloscope was configured to sample at a rate of 125 MHz, translating to an acquisition of 10 data points for each clock cycle. Such a setup aligns with standard configurations previously employed in masking leakage evaluations [DCA20a, ABC+23, FVBR+21b]. We set the oscilloscope memory buffer to 170K sample in order to capture a full execution of a mantissa multiplication, the exponent addition and sign XORing.

We adopt the widely-used TVLA methodology to perform the masked design's leakage evaluation (see Section 2.4). We ran the first-order and second-order univariate fixed-vs-random and fixed-vs-fixed t-tests. In the fixed-vs-random test, the setup captures two sets of power traces: one where the input remains constant throughout all set collection, and another where the input varies with each execution. However, the sequence in which these sets are captured is randomized. The reason for setting these sets is that the t-score gives a result of whether the fixed dataset power activity is distinguishable from the ones belonging to the random dataset. Additionally, we perform first-order TVLA for the fixed-vs-fixed setting to cover zero correction cases stemming from the exponent and the lower bits of the mantissa. In this setting, one group has operands that cause zero correction actions, while the other group does not. This comprehensive evaluation ensures that our masked design is thoroughly assessed for side-channel leakage across the exponent and the lower-bits of mantissa zero correction actions.

Figure 6 illustrates the TVLA test results for both the low-area and balanced protected design configurations across six plots. The first four plots, Figures 6(a)-(d), correspond to the low-area protected design. In Figure 6(a), the first-order TVLA results with active randomness over 10 million traces are depicted. The t-scores consistently remain within the threshold (±4.5), demonstrating the empirical security of our masked design. Figure 6(b) presents the second-order TVLA results with 10 million traces, which exhibit the expected

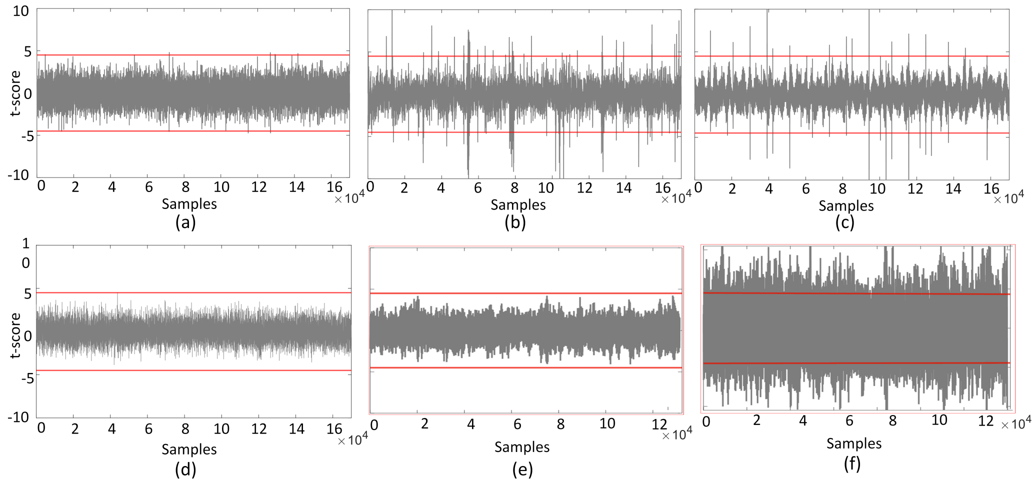

Figure 6: TVLA results for the low-area and balanced protected design configurations across six scenarios. The first four plots correspond to the low-area protected design: (a) first-order fixed-vs-random TVLA with 10 million traces, (b) second-order fixed-vs-random TVLA with 10 million traces, (c) first-order fixed-vs-random TVLA with 5,000 traces with randomness deactivated, and (d) first-order fixed-vs-fixed TVLA with 5 million traces. The final two plots illustrate the TVLA results for the balanced protected design: (e) first-order fixed-vs-random TVLA with 1 million traces, and (f) first-order fixed-vs-random TVLA with 5,000 traces with randomness deactivated.

information leakage. In Figure 6(c), the first-order TVLA results with randomness deactivated show significant peaks well above the t-score threshold (± 4.5), indicating clear leakage even after 5,000 traces. This result demonstrates that the unprotected design's power activity is statistically distinguishable from the random dataset's activity, revealing the presence of side-channel leakage. Additionally, Figure 6(d) presents the first-order TVLA results for the fixed-vs-fixed setting with 5 million traces. The results show no leakage, indicating that the design effectively mitigates leakage, even though the mantissa and exponent operations undergo zero correction actions.

The last two plots, Figures 6(e)-(f), present the TVLA results for the balanced protected design. Figure 6(e) depicts the first-order TVLA results with active randomness over 1 million traces, where the t-scores consistently remain within the threshold (± 4.5), demonstrating the empirical security of our masked design. In contrast, Figure 6(f) shows the first-order TVLA results with randomness deactivated, indicating significant peaks well above the t-score threshold (± 4.5) and clear leakage even after 5,000 traces. Notably, the plots in Figures 6(e)-(f) show fewer samples due to the balanced design's shorter latency relative to the low-area configuration. Since our balanced design uses the same masked gadgets as the low-area protected design, we performed first-order TVLA only with 1 million traces and did not perform second-order TVLA or the fixed-vs-fixed setting.

6 Discussions

Mitigating the attacks on floating-point multiplications in [KA21] and [GMRR22]. We propose a robust masking circuit designed to mitigate the leakages in FALCON's floating-point multiplication, priorly exploited by [KA21] and [GMRR22]. These attacks specifically target the sign-bit XORing, exponent addition, and integer multiplication of the mantissa. Our countermeasure effectively mitigates the leakages identified in the

initial attack by [KA21] as well as its improved version [GMRR22]. Karabulut et al. target integer multiplication and then its following intermediate addition operations in the mantissa multiplication to find the firstly possible guesses and then resolve false positives. However, this attack is avoided with our approach, as both multiplication and stage addition processes are masked.

Similarly, Guerreau et al. describe attack sequences that target the same operations but within reduced search space, which our countermeasure mitigates by masking the operations involved. Additionally, our design masks sign-bit XORing and exponent addition, further securing the sign and exponent values of the floating-point multiplication against side-channel attacks. Thus, our countermeasure addresses side-channel vulnerabilities, particularly identified by [KA21] and [GMRR22]. While our protection scheme addresses these vulnerabilities, it is important to note that we do not claim first-order CPA security for the rounding operation that concludes floating-point multiplication in FALCON implementation. Although no current attacks have successfully exploited rounding operations, we acknowledge the potential risk that secret information might be exposed if the rounding is targeted by a more sophisticated side-channel attack.

Masking Enhancements and Limitations. Our proposal is the first one to mask floating-point multiplication in hardware; hence, it aims to serve as a benchmark for future optimizations. Other masking schemes such as GLM or UMA, can possibly be adapted in the future to reduce the latency and randomness of the masked gadgets [GIB18, GM18, RSM20]. Moreover, there are other available share conversion circuits, which can also be incorporated to potentially reduce the area costs [MTMM07, Deb12, BC22, SPOG19, BCZ18]. Recent research has shown how coupling effects can cause first-order secure FPGA implementations to become susceptible to high temperatures, high voltages, and high clock frequencies [DEM18, MHK+23]. Furthermore, there is another work that showed how the couplings might be enhanced externally to violate the security assumptions [LBS19]. Our threat model considers such attacks out of scope, i.e., our masked designs are not proven either theoretically or practically against such attacks. Note that these are not our unique assumptions and are commonplace in general masking schemes as evidenced in recent TCHES publications [DAP+22].

We only consider first-order masking schemes in this work. Proposed schemes may suffer from local or global compositional flaws [MMSS19] if it is desired to extend them for higher-order masking. Higher-order masking with related challenges and attacks that manipulate the setup is out of scope, i.e., we exclusively focus on first-order masking as in prior recent TCHES publications [ABC+23, DAP+22, CC24]. More maturity is needed on these aspects in the context of AES and other classical schemes before investigating and transitioning them to post-quantum cryptography or ML frameworks.

TVLA vs Other Methods. We do not claim TVLA to be neither perfect nor the best technique to test side-channel leakage for all scenarios and all types of circuits. There are several alternatives to TVLA that can also evaluate side-channel leakage in a proposed hardware/software implementation [KJJ99, BCO04, CRR03, GBTP08, HGD+11]. Typically, there is a need to understand the underlying algorithm to form a hypothesis and establish a power model, or there is a necessity to modify confidential data to create a power profile for the targeted device if such approaches are pursued. We opted for TVLA since it does not come with these constraints. Nonetheless, we recognize the potential pitfalls of TVLA due to its simplistic moment-based analysis, which can result in both false negatives and positives [Sta18].

Extending the Design to Other FPGAs. As in prior TCHES papers [ABC+23, DCV+23, MGTF19, DCA20a, CGF21], we demonstrate the application of our proposed solution on

Xilinx FPGAs. However, our technique is not fundamentally limited to Xilinx FPGAs. For example, if the design were to be extended to Altera Stratix FPGAs, the developer can use our mantissa operand splitting technique but re-size the operads for the 18-by-18 multipliers of Altera DSP blocks. ASIC developers who incorporate third-party multiplier hard macros into their designs can follow the same strategy.

Masking AI/ML hardware. An attack on floating-point multiplication for AI/ML applications can leak the internal model values [BBJP19, NK23], which are trademark secrets, e.g., in Machine-Learning-as-a-Service applications, due to the costs of model training. Therefore, masking of floating point multiplications can find other use cases. Although there are efforts in masking neural networks [DCA20b, DCA20a, DCSA22, DAP+22, DCV+23, AWDF21], they focus on quantized networks without floating-point arithmetic. Our proposed techniques can be used in non-quantized neural networks that preserve full precision.

Addressing Masking Flaws in Practical Hardware Implementations. In practical applications, most hardware implementations are not glitch-free, and transient values can leak additional information [MPG05, NRR06]. The robust probing model addresses this by assuming adversaries can observe all inputs a wire depends on in combinatorial circuits [MPO05]. Although our masking gadgets are proven secure under the PINI probing model within the glitch-extended probing framework, this abstract-level security does not guarantee flaw-free implementations. Therefore, we complement our theoretical proofs with empirical validation methods, such as TVLA, to support our security claims.

7 Conclusions and Future Works

In this research, we ventured into the largely uncharted domain of side-channel protections for floating-point multiplication, an operation integral to numerous computational tasks yet vulnerable to potential attacks. Our findings underscore the significance of protecting this specific arithmetic process. Through the methodologies presented, we have successfully established the novel side-channel defense mechanism tailored for floating-point multiplication. While our results show promising resilience against known side-channel attack techniques, future endeavors should be directed towards enhancing the protection's efficiency, ensuring compatibility with a broader range of devices, and assessing the defense against evolving side-channel attack methodologies. The foundations laid in this study pave the way for a more secure computational landscape in applications where floating-point multiplication is pivotal.

References

[ABC+22] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Markus Schönauer, Tobias Schneider, François-Xavier Standaert, and Christine van Vredendaal. Protecting dilithium against leakage: Revisited sensitivity analysis and improved implementations. *Cryptology ePrint Archive*, 2022.

[ABC+23] Aikata Aikata, Andrea Basso, Gaetan Cassiers, Ahmet Can Mert, and Sujoy Sinha Roy. Kavach: Lightweight masking techniques for polynomial arithmetic in lattice-based cryptography. Cryptology ePrint Archive, Paper 2023/517, 2023. `https://eprint.iacr.org/2023/517`.

[AWDF21] Konstantinos Athanasiou, Thomas Wahl, A Adam Ding, and Yunsi Fei. Masking feedforward neural networks against power analysis attacks. *Proceedings on Privacy Enhancing Technologies*, 2022(1), 2021.

[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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 116–129, 2016.

[BBJP19] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security*, pages 515–532, Santa Clara, CA, 2019.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 616–648. Springer, 2016.

[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*, pages 553–588, 2022.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *CHES 2004*, volume 3156 of *LNCS*, pages 16–29. Springer, Heidelberg, August 2004.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. Improved high-order conversion from boolean to arithmetic masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 22–45, 2018.

[BGR+21] Joppe Willem Bos, Marc Olivier Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking kyber: First-and higher-order implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):173–214, 2021.

[CC24] Keng-Yu Chen and Jiun-Peng Chen. Masking floating-point number multiplication and addition of falcon: First- and higher-order implementations and evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):276–303, Mar. 2024.

[CGF21] Ana Covic, Fatemeh Ganji, and Domenic Forte. Circuit masking: from theory to standardization, a comprehensive survey for hardware security researchers and practitioners. *arXiv preprint arXiv:2106.12714*, 2021.

[CGLS20] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Transactions on Computers*, 70(10):1677–1690, 2020.

[CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4):110–145, 2023.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 13–28. Springer, Heidelberg, August 2003.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020.

[DAP+22] Anuj Dubey, Afzal Ahmad, Muhammad Adeel Pasha, Rosario Cammarota, and Aydin Aysu. Modulonet: Neural networks meet modular arithmetic for efficient hardware masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 506–556, 2022.

[DCA20a] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Bomanet: Boolean masking of an entire neural network. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*, pages 51:1–51:9. IEEE, 2020.

[DCA20b] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Maskednet: The first hardware inference engine aiming power side-channel protection. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 197–208. IEEE, 2020.

[DCSA22] Anuj Dubey, Rosario Cammarota, Vikram Suresh, and Aydin Aysu. Guarding machine learning hardware against physical side-channel attacks. *J. Emerg. Technol. Comput. Syst.*, 18(3), apr 2022.

[DCV+23] Anuj Dubey, Rosario Cammarota, Avinash Varna, Raghavan Kumar, and Aydin Aysu. Hardware-software co-design for side-channel protected neural network inference. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 155–166. IEEE, 2023.

[Deb12] Blandine Debraize. Efficient and provably secure methods for switching from arithmetic to Boolean masking. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 107–121. Springer, Heidelberg, September 2012.

[DEM18] Thomas De Cnudde, Maik Ender, and Amir Moradi. Hardware masking, revisited. *IACR TCHES*, 2018(2):123–148, 2018. https://tches.iacr.org/index.php/TCHES/article/view/877.

[DMRB18] Lauren De Meyer, Oscar Reparaz, and Begül Bilgin. Multiplicative masking for aes in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 431–468, 2018.

[FVBR+21a] 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. *Cryptology ePrint Archive*, 2021.

[FVBR+21b] 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. *Cryptology ePrint Archive*, 2021.

[GBTP08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES 2008*, volume 5154 of *LNCS*, pages 426–442. Springer, Heidelberg, August 2008.

[GIB18] Hannes Gross, Rinat Iusupov, and Roderick Bloem. Generic low-latency masking in hardware. *IACR TCHES*, 2018(2):1–21, 2018. https://tches.iacr.org/index.php/TCHES/article/view/871.

[GM18] Hannes Groß and Stefan Mangard. A unified masking approach. *Journal of Cryptographic Engineering*, 8(2):109–124, June 2018.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *Cryptology ePrint Archive*, 2016.

[GMRR22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: power analysis attacks on falcon. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 141–164, 2022.

[Gou01] Louis Goubin. A sound method for switching between Boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *CHES 2001*, volume 2162 of *LNCS*, pages 3–15. Springer, Heidelberg, May 2001.

[HGD+11] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293–302, December 2011.

[HKL+22] Daniel Heinz, Matthias J Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Daan Sprenkels. First-order masked kyber on arm cortex-m4. *Cryptology ePrint Archive*, 2022.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*, pages 463–481. Springer, 2003.

[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. IEEE, 2021.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Heidelberg, August 1999.

[LBS19] Itamar Levi, Davide Bellizia, and François-Xavier Standaert. Reducing a masked implementation's effective security order with setup manipulations. *IACR TCHES*, 2019(2):293–317, 2019. https://tches.iacr.org/index.php/TCHES/article/view/7393.

[LZP+24] Jiangxue Liu, Cankun Zhao, Shuohang Peng, Bohan Yang, Hang Zhao, Xiangdong Han, Min Zhu, Shaojun Wei, and Leibo Liu. A low-latency high-order arithmetic to boolean masking conversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):630–653, 2024.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium: Efficient implementation and side-channel evaluation. In *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*, pages 344–362. Springer, 2019.

[MHK+23] Dev M Mehta, Mohammad Hashemi, David S Koblah, Domenic Forte, and Fatemeh Ganji. Bake It Till You Make It: Heat-induced Power Leakage from Masked Neural Networks. *Cryptology ePrint Archive*, 2023.

[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.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M Gammel. Side-channel leakage of masked cmos gates. In *Cryptographers' Track at the RSA Conference*, pages 351–365. Springer, 2005.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. Successfully attacking masked aes hardware implementations. In *International workshop on cryptographic hardware and embedded systems*, pages 157–171. Springer, 2005.

[MTMM07] Robert P. McEvoy, Michael Tunstall, Colin C. Murphy, and William P. Marnane. Differential power analysis of HMAC based on sha-2, and countermeasures. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers*, volume 4867 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2007.

[NK23] Hanae NOZAKI and Kazukuni KOBARA. Power analysis of floating-point operations for leakage resistance evaluation of neural network model parameters. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 2023.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *International conference on information and communications security*, pages 529–545. Springer, 2006.

[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: Fast-fourier lattice-based compact signatures over NTRU. Technical report, National Institute of Standards and Technology, 2020.

[RSM20] Aein Rezaei Shahmirzadi and Amir Moradi. Re-consolidating first-order masking schemes: Nullifying fresh randomness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):305–342, Dec. 2020.

[SAW+23] Michael Schmid, Dorian Amiet, Jan Wendler, Paul Zbinden, and Tao Wei. Falcon takes off-a hardware implementation of the falcon signature scheme. *Cryptology ePrint Archive*, 2023.

[SM16] Tobias Schneider and Amir Moradi. Leakage assessment methodology — extended version. *J. Cryptogr. Eng.*, 6(2):85–99, 2016.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *Public-Key Cryptography–PKC 2019: 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II 22*, pages 534–564. Springer, 2019.

[Sta18] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. In Begül Bilgin and Jean-Bernard Fischer, editors, *Smart Card Research and Advanced Applications, 17th International Conference, CARDIS 2018, Montpellier, France, November 12-14, 2018, Revised Selected Papers*, volume 11389 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2018.

[Xil18] Xilinx Inc. *7 Series DSP48E1 Slice*, 3 2018. v1.10.