Noname manuscript No.
(will be inserted by the editor)

Masking ring-LWE

Oscar Reparaz · Sujoy Sinha Roy · Ruan de Clercq · Frederik Vercauteren · Ingrid Verbauwhede

Received: date / Accepted: date

Abstract In this paper, we propose a masking scheme to protect ring-LWE decryption from first-order side-channel attacks. In an unprotected ring-LWE decryption, the recovered plaintext is computed by first performing polynomial arithmetic on the secret key, and then decoding the result. We mask the polynomial operations by arithmetically splitting the secret key polynomial into two random shares; the final decoding operation is performed using a new bespoke masked decoder. The output of our masked ring-LWE decryption are Boolean shares suitable for derivation of a symmetric key. Thus, the masking scheme keeps all intermediates, including the recovered plaintext, in the masked domain. We have implemented the masking scheme on both hardware and software. On a Xilinx Virtex-II FPGA, the masked ring-LWE processor requires around 2000 LUTs, a 20% increase in the area with respect to the unprotected architecture. A masked decryption operation takes 7478 cycles, which is only a factor $\times 2.6$ larger than the unprotected decryption. On a 32-bit ARM Cortex-M4F processor, the masked software implementation costs around $\times 5.2$ more cycles than the unprotected implementation.

This journal version is based on a paper appeared at the CHES 2015 conference [27]. Sections 6, 7.3 and 8.2 carry substantial differences.

O. Reparaz
Kasteelpark Arenberg 10 bus 2452
Tel.: +32-45-678910
Fax: +123-45-678910
E-mail: oscar.reparaz@esat.kuleuven.be

1 Introduction

Once the quantum computer is built, Shor's algorithm will make most current public-key cryptographic algorithms obsolete. In particular, public-key cryptosystems that rely on number-theoretic hardness assumptions such as integer factorization (RSA) or discrete logarithms, either in $\mathbb{Z}_p^*$ (Diffie-Hellman) or in elliptic curves over finite fields, will be insecure. On the bright side, there is an entire branch of post-quantum cryptography that is believed to resist mathematical attacks running on quantum computers.

There are three main branches of post-quantum cryptosystems: based on codes, on multivariate quadratic equations or on lattices [2]. Lattice-based cryptographic constructions, founded on the *learning with errors* (LWE) problem [24] and its ring variant known as ring-LWE problem [18], have become a versatile tool for designing asymmetric encryption schemes [18], digital signatures [10] and homomorphic encryption schemes [11,4]. Several hardware and software implementations of such schemes have appeared in the literature. So far, the reported implementations have focused mainly on efficient implementation strategies, and very little research work has appeared in the area of side channel security of the lattice-based schemes.

It comes as no surprise that implementations of post-quantum algorithms are vulnerable to side-channel attacks. Side-channel attacks, as introduced by Kocher [16], exploit timing, power consumption or the electromagnetic emanation from a device executing a cryptographic implementation to extract secrets, such as cryptographic keys. A particularly powerful side-channel technique is Differential Power Analysis (DPA), introduced by Kocher et. al. [17]. In a typical DPA attack, the adversary measures the instantaneous power consumption

of a device, places hypotheses on subkeys and applies statistical tests to confirm or reject the hypotheses. DPA attacks can be surprisingly easy to mount even with low-end equipment, and hence it is important to protect against them.

There are plenty of countermeasures against DPA. Most notably, masking [7,14] is provably sound and popular in industry. Masking effectively randomizes the computation of the cryptographic algorithm by splitting each intermediate into several shares, in such a way that each share is independent from any secret. This property is preserved through the entire computation. Thus, observing any single intermediate (for example, by a side-channel, be it known or unknown) reveals nothing about the secret. However, there are not many masking schemes specifically designed for post-quantum cryptography. In [5] Brenner et. al. present a masked FPGA implementation of the post-quantum pseudo-random function SPRING.

In the rest of the paper, we focus on protecting the ring-LWE decryption operation against side-channel attacks with masking. The decryption algorithm is considerably exposed to DPA attacks since it repeatedly uses long-term private keys. In contrast, the encryption or key-generation procedures use ephemeral secrets only [28].

Our contribution. In this paper we present a compact masked implementation of the ring-LWE decryption function. The masking countermeasure adds a small overhead when compared to the other previous approaches, thanks to a bespoke probabilistic masked decoder designed specifically for our implementation. We implemented the masked ring-LWE decryption on a Virtex-II FPGA and on an ARM Cortex-M4F processor, and tested the side-channel security with practical experiments.

Organization. The paper is structured as follows: we provide a brief mathematical background of the ring-LWE encryption scheme in Section 2 and describe a high-level overview of the proposed masked ring-LWE decryption in Section 3. Next, in Section 4 we construct the masked decoder. In 5 we describe our hardware implementation, and in the next Section 6 we describe our software implementation. We analyze the error rates of the decryption operation in Section 7. We dedicate Section 8 for the side-channel evaluation of both hardware and software implementations and draw conclusions in the last section.

2 Preliminaries

Notation. The Latin letters r, c_i indicate polynomials. When we want to explicitly access a coefficient of the polynomial we write $r[i]$. Multiplication of polynomials is written as $r * c_1$. Coefficient-wise multiplication is denoted as $r \cdot c_1$. The letter m denotes a string of message bits, and q is an integer. Letters with prime x' or double prime x'' represent shares of variable x. Depending on the context, these shares are split either arithmetically $x = x' + x'' \pmod q$ or Boolean $x = x' + x'' \pmod 2$. A polynomial r is shared into (r', r'') by additively sharing each of its coefficients $r[i]$ such that $r = r' + r''$.

Ring-LWE. For completeness, we give in this section a description of the three major algorithms of the ring-LWE public-key cryptosystem [18]: key-generation, encryption and decryption.

The ring-LWE encryption scheme works with polynomials in a ring $R_q = \mathbb{Z}_q[\mathbf{x}]/(f(x))$, where $f(x)$ is an irreducible polynomial of degree n. During the key generation, encryption and decryption operations, polynomial arithmetic such as polynomial addition, subtraction and multiplication are performed. In addition, the key-generation and encryption operations require sampling of error polynomials from an error distribution (typically a discrete Gaussian.)

The ring-LWE encryption scheme is described in this way:

- In the key generation phase, two error polynomials r_1 and r_2 are sampled from the discrete Gaussian distribution. The secret key is the polynomial r_2 and the public key is the polynomial $p = r_1 - g * r_2$. After key generation, there is no use of the polynomial r_1. The polynomial g is globally known.
- In the encryption operation of a binary message vector m of length n, the message is first lifted to a ring element $\bar{m} \in R_q$ by multiplying the message bits by $q/2$. The ciphertext is computed as a pair of polynomials (c_1, c_2) where $c_1 = g * e_1 + e_2$ and $c_2 = p * e_1 + e_3 + \bar{m} \in R_q$. The encryption operation requires generation of three error polynomials e_1, e_2 and e_3.
- The decryption operation uses the private key r_2 to compute the message as $m = \mathrm{th}(c_1 * r_2 + c_2)$. The decoding function th is a simple threshold decoder that is applied coefficient-wise and is defined as

$$\mathrm{th}(x) = \begin{cases} 0 & \text{if } x \in (0, q/4) \cup (3q/4, q) \\ 1 & \text{if } x \in (q/4, 3q/4) \end{cases} \quad (1)$$

Efficiency improvements. To achieve an efficient implementation of the encryption scheme, the irreducible polynomial $f(x)$ is taken as $x^n + 1$ where n is a power of two, and the modulus q is chosen as a prime number satisfying $q \equiv 1 \mod 2n$ [21,29]. In this setting, polynomial multiplications can be efficiently performed in $O(n \log n)$ time using the Number Theoretic Transform (NTT).

Following [29], we keep the ciphertext polynomials c_1 and c_2 in the NTT domain to reduce the computation cost of the decryption operation. The decryption operation thus computes the decrypted message as

$$m = \mathrm{th}\big(\mathrm{INTT}(\tilde{c}_1 \cdot \tilde{r}_2 + \tilde{c}_2)\big). \tag{2}$$

Here the symbol $\tilde{r}$ represents the NTT of a polynomial r, and $\mathrm{INTT}(\cdot)$ represents the inverse NTT operation. The multiplication of $\tilde{c}_1 \cdot \tilde{r}_2$ is thus performed coefficient-wise (as well as the addition $\tilde{c}_1 \cdot \tilde{r}_2 + \tilde{c}_2$.) For convenience, we drop the tildes in the rest of the paper and work with c_1, c_2 and r_2 in the NTT domain. We furthermore refer to $\tilde{r}_2$ simply as r. (We recall that the INTT is a linear transformation applied to the n coefficients of $a = r \cdot c_1 + c_2$.) The decoding function th applies a threshold function to each coefficient of a as defined in Equation 1 to output n recovered message bits.

3 High-level overview

In this section, we give a high-level view of the masked ring-LWE implementation. The most natural way to split the computation of the decryption as Equation 2 is to split the secret polynomial r additively into two shares r' and r'' such that $r[i] = r'[i] + r''[i] \pmod q$ for all i. The n coefficients of r' are chosen uniformly at random in $\mathbb{Z}_q$ in each execution of the decryption.

The bulk of the computation from Equation 2 is amenable to this splitting, since by linearity of the multiplication and INTT operation, we have that $\mathrm{INTT}(r \cdot c_2 + c_1) = \mathrm{INTT}(r' \cdot c_2 + c_1) + \mathrm{INTT}(r'' \cdot c_2)$. Thus, we can split almost the entire computation from Equation 2 into two branches, as drawn in Figure 1. The first branch computes on r' to determine the polynomial

$$a' = \mathrm{INTT}(r' \cdot c_1 + c_2) \tag{3}$$

and the second branch operates on r'' to determine

$$a'' = \mathrm{INTT}(r'' \cdot c_1). \tag{4}$$

The advantage of such a high-level masking is that the operations of Equation 3 and 4 can be performed on an arithmetic processor without any particular protection against DPA. (This is because any intermediate appearing in either branch is independent of the secret r. This situation is very similar to, for example, base point blinding in elliptic curve scalar multiplication.) We can reuse an existing ring-LWE processor for these operations, and leverage the numerous optimizations carried out for this block [21,29,9].

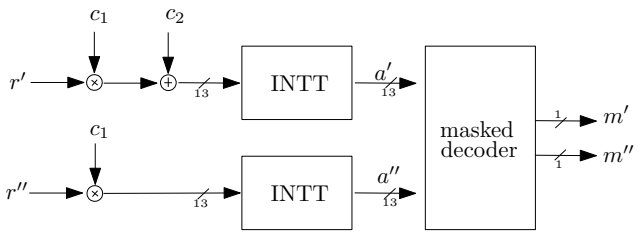

Fig. 1: General data flow of the masked ring-LWE decryption. r' and r'' are the arithmetic shares of the private key r; c_1 and c_2 are the input unmasked ciphertext; m' and m'' are the Boolean shares of the recovered plaintext.

The final threshold $\mathrm{th}(\cdot)$ operation of Equation 2 is obviously non-linear in the base field $\mathbb{F}_q$, and hence cannot be independently applied to each branch (Equation 3 and 4). There are generic approaches to mask arbitrary functions. For instance, in [5] an approach based on masked tables was used. However, these generic approaches are usually quite expensive in terms of area or randomness. In the following Section 4, we pursue another direction. We design a bespoke masked decoder that results in a compact implementation.

4 Masked decoder

In this section we describe a compact, probabilistic masked decoder. In the sequel, a denotes a single coefficient and (a', a'') its shares such that $a' + a'' = a \pmod q$. The decoder computes the function $\mathrm{th}(a)$ from the shares (a', a''). We also drop the symbol $\pmod q$ when obvious.

First crack. The key idea of the efficient masked decoder is that we do not need to know the exact values of the shares a' and a'' of a coefficient a in order to compute $\mathrm{th}(a)$. For example, if $0 < a' < q/4$ and $q/4 < a'' < q/2$ then $a = a' + a''$ is bounded by $q/4 < a < 3q/4$, and thus $\mathrm{th}(a) = 1$. That is, we learnt $\mathrm{th}(a)$ from only a few most significant bits from a' and a''. We can use this idea to substantially simplify the complexity of the masked th function.

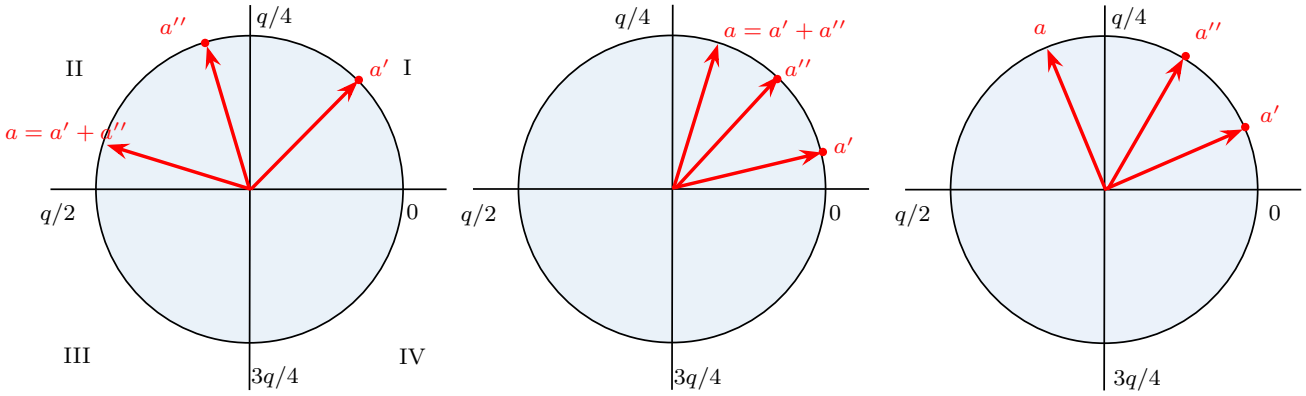

Fig. 2: Idea for the masked decoder. Elements in $\mathbb{Z}_q$ are shown in a circle. Adding two elements translates into adding their respective angles. Left: case $0 < a' < q/4$, $q/4 < a'' < q/2$, and therefore $\mathrm{th}(a) = 1$. Center and right: case $0 < a' < q/4$, $0 < a'' < q/4$, which does not allow to infer $\mathrm{th}(a)$.

4.1 Rules

Figure 2, left, illustrates the situation from the last paragraph. In this case, $0 < a' < q/4$ and $q/4 < a'' < q/2$ so obviously a can range only from $q/4$ to $3q/4$, and hence $\mathrm{th}(a) = 1$. Analogously to this rule, we can formulate 3 other rules:

- If $q/2 < a' < 3q/4$ and $3q/4 < a'' < q$ then $q/4 < a < 3q/4$ and thus $\mathrm{th}(a) = 1$.
- If $q/4 < a' < q/2$ and $q/2 < a'' < 3q/4$ then a belongs to $(0, q/4) \cup (3q/4, q)$ and thus $\mathrm{th}(a) = 0$ (quadrants I and IV, left half of the circle).
- If $3q/4 < a' < q$ and $0 < a'' < q/4$ then a belongs to $(0, q/4) \cup (3q/4, q)$ and thus $\mathrm{th}(a) = 0$.

There are 4 other rules that result from interchanging a' with a'' in the above expressions. (This follows straight from the symmetry of the additive splitting.) Essentially, with the only information of the *quadrant* of each share a' and a'' we can, in half of the cases, deduce the output of $\mathrm{th}(a)$. (For the explanation simplicity, we obviated what happens in the boundaries of the quadrant intervals. Similar conclusions hold when including them.)

What if no rule is hit? In roughly half of the cases, we can apply one of the 8 rules previously described to deduce the value of $\mathrm{th}(a)$. However, in the other half of the cases, none of the rules applies. A representative case of this event is shown in Figure 2, center and right. In both cases, $0 < a' < q/4$ and $0 < a'' < q/4$. This situation is not covered by any of the 8 rules previously described. We see that in the center sub figure $\mathrm{th}(a) = 0$ while in the right sub figure $\mathrm{th}(a) = 1$, so in this case the quadrants of each share a' and a'' do not allow us to infer $\mathrm{th}(a)$.

The solution in this case is to refresh the splitting (a', a''), that is, update $a' \leftarrow a' + \Delta_1$ and $a'' \leftarrow a'' - \Delta_1$ for certain Δ_1. (This refreshing[1] naturally preserves the unshared value $a = a' + a''$.) After the refreshing, the 8 rules can be checked again. If still no rule applies, the process is repeated with a different refreshing value Δ_i. Note that in each iteration of the step, roughly half of the possible values of $(a', a'') \in \mathbb{Z}_q \times \mathbb{Z}_q$ are successfully decoded, and thus the amount of pairs (a', a'') that do not get decoded shrinks exponentially with the number of iterations. In our implementation, $N = 16$ iterations produces a satisfactory result. This will be studied in detail in Section 7.1.

Optimal cooked values for Δ_i. One can determine a sequence of Δ_i values that maximizes the number of pairs successfully decoded after N iterations. We performed a first-order search for such a sequence of Δ_i values. Each Δ_i maximizes the number of successfully decoded pairs after $i - 1$ iterations. For $q = 7681$ the sequence of Δ_i shown in Appendix A was found.

Architecture. The hardware architecture for the masked decoder follows from the previous working principle description. Our implementation is shown in Figure 3. From left to right, we see the first refreshing step by the constants Δ_i. The constants Δ_i vary from iteration to iteration. After the refreshing step, the quadrant function is applied to each share a', a''. This quadrant function outputs x if a belongs to the x-th quadrant, and thus the output consists of 2 bits. These blocks

[1] We use here the term "refresh" to refer to the process of modifying the masked representation (a', a'') of a without modifying the unshared value a, but, contrary to other contexts in the literature, we do not imply that we are pumping new randomness in the new representation.

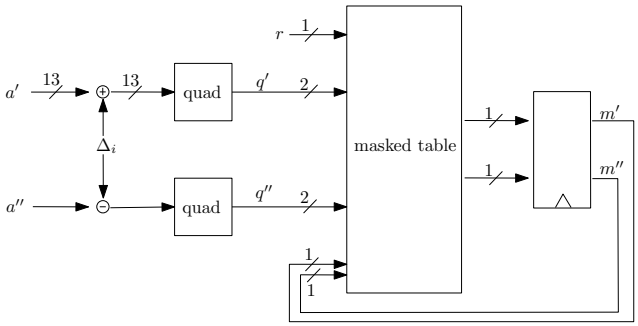

Fig. 3: Data flow for the masked decoder.

are essentially 13-bit comparators, and thus relatively inexpensive in logic.[2] The subsequent rule checking on (q', q'') is performed by a masked table lookup that is described in the following section. The whole process is repeated $N = 16$ iterations, and this number of iterations stays fixed even if the decoding is successful after the few first iterations.

4.2 Masked table lookup

The final step in the masked decoder is a masked table lookup. This table implements the rules described in Section 4.1, and essentially maps the output of each quadrant q'_i and q''_i (2 bits each) after the i-the iteration ($i \in [1, N]$) to a (Boolean) masked output bit value (m'_i, m''_i). In our specific implementation, we have other inputs: the result of the decoding from the previous iteration (m'_{i-1}, m''_{i-1}) and an extra randomness bit r (fresh at each of the N iterations for each of the n coefficients).

This is a well-studied problem that arises in other situations (for instance, when masking the sbox lookup in a typical block cipher) and there are plenty of approaches here to implement such masked table lookup.

Hardware. In hardware, we opted for the approach of masked tables as in [31]. We set $m'_i \leftarrow r$ and we compute $m''_i \leftarrow f(r, q'_i, q''_i, m'_{i-1}, m''_{i-1})$. The function f essentially bypasses the previous decoded value when no rule applies to q'_i, q''_i by setting the output m''_i to $r + m'_{i-1} + m''_{i-1}$ (refreshing the content of the output registers). If a rule applies to q'_i, q''_i, it sets the output m''_i accordingly. By doing this, we can register always the output of this table and no control logic to enable such output register is needed (it is implicitly integrated into this masked table.) This is the reason why the table sees also the previous decoded value m'_{i-1} and m''_{i-1}.

[2] Note that in the special case that q is a prime close to a power of two the construction of the quadrant block can be further simplified.

The usual precautions are applied when implementing f. For our target FPGA platform, we carefully split the 7-bit input to 1-bit output function f into a balanced tree of 4-bit input LUTs, in such a way that any intermediate input or output of LUTs does not leak in the first order. Note that here we are assuming that each LUT is an atomic operation. If stronger security guarantees are needed, other approaches to implement such function f should be followed. When implemented in an ASIC, it may be preferable to store this masked table in ROM (since the contents of the table are immutable and the size is small.)

The output of this table is (Boolean) masked, and thus no unmasked value lives within the implementation. This is suited for consumption of a masked AES module (say) after some preprocessing as will be detailed later. We stress that we use masked tables on the *output* of the quadrants. This is the key for our reduced area requirements, as will be explained in Section 5.

Software. For the software implementation of the masked table lookup we base our approach on the previous hardware description. We first write an unmasked decoder in a (software) bitsliced way, and then apply the method of [1] to provide "gate-level" masking to the bitsliced software implementation. More details are given in Section 6.

5 Hardware implementation

We implemented the fully masked ring-LWE decryption system with the proof-of-concept parameter set $(n, q, s) = (256, 7681, 11.32)$ first introduced in [13], corresponding to a medium-term security level. Note that these concrete choice of parameters is not meant to be deployed. The target platform is a Xilinx Virtex-II xc2vp7 FPGA. The HDL files were synthesized within Xilinx ISE v8.2 with optimization settings set to balanced and `KEEP HIERARCHY` flag when appropriate to prevent optimization of security-critical components. We base our arithmetic processor on the design from [29].

5.1 Area

In our case, a single arithmetic coprocessor performs serially the computations of Equation 3 and then that of Equation 4. This incurs in a very slight area overhead (only the control microcode is slightly modified, plus the masked decoder), at the obvious cost of an increased execution time. In comparison to the unprotected version, our protected decryption scheme consumes more memory as now we store two shares r' and r'' of the

	LUTs/FFs/DSPs	$f_{\max}$ [MHz]	cycles
unprotected	1713/830/1	120	2.8k
protected	2014/959/1	100	7.5k

Table 1: Performance and Comparison on Xilinx Virtex-II xc2vp7 FPGA. Note that these results are not directly comparable with [29], since the latter were obtained from a more advanced Virtex-6 FPGA, which has 6-bit input LUTs and superior routing mechanisms in comparison to our target FPGA.

secret polynomial r, and the two output polynomials a' and a'' from the two INTT operations.

In Table 1, we can see that the proposed masking of the ring-LWE architecture incurs an additional area overhead of only 301 LUTs and 129 FFs in comparison to the unprotected version. This additional area cost is mostly due to a pair of masked decoders. Due to its low area overhead, we chose to keep two masked decoders in parallel, decoding two coefficients simultaneously. (This nicely fits with the memory organization of the arithmetic coprocessor, since it fits two 13-bit coefficients in each memory word.) Thus, we use two addition and subtraction circuits for the refreshing with Δ_i (accounting for 160 LUTs) and two masked tables (90 LUTs in total).

We note that we could straightforward reduce the additional area cost by reusing the 13-bit addition and subtraction circuits present in the arithmetic coprocessor. Since during a decoding operation, the arithmetic coprocessor remains idle, reusing of the addition and subtraction circuits do not cause any increase in the cycle count. For simplicity, we did not implement this approach.

5.2 Cycle count

The cycle count for our approach is decomposed in the computation of Equation 3, Equation 4 and the masked decoder. Equation 3 takes 2840 cycles (one unprotected ring-LWE decryption), Equation 4 takes 2590 cycles, slightly less than Equation 3 since there is no addition present in the second branch.

The two-way parallel masked decoder takes $\frac{1}{2} \times n \times N + \epsilon$ cycles to decode all the coefficients into message bits. In our case with $n = 256$, $N = 16$ the masked decoder takes 2048 cycles. Thus in total, a masked decryption operation requires 7478 cycles. The arithmetic coprocessor and the masked decoder run in constant time and constant flow.

5.3 Comparison with an elliptic-curve cryptosystem

We compare our protected decryption scheme with the unprotected high-speed elliptic curve scalar multiplier architecture proposed by Rebeiro et al. in [23]. The architecture for the field GF(2^{233}) consumes 23 147 LUTs and computes an unprotected scalar multiplication in 12.5μs on a more advanced Virtex-4 FPGA. Thus the scalar multiplier has an area $\times$ time product of approximately 289 337. Our protected ring-LWE decryption (for a similar security) achieves an area $\times$ time product of approximately 151 452 on a Virtex-2 FPGA; thus achieving at least 1.9 times better figure of merit.

5.4 Trade-offs

The previous figures are subject to trade-offs. If smaller latency is desired instead of a compact implementation, two coprocessors can perform the two computations of Equation 3 and 4 in parallel. Trade-offs also apply to the masked decoder, and the parallelization could be extended easily to reduce latency in this stage. Since the BRAMs present in the Xilinx FPGAs support reading of multiple consecutive words, we could keep more pairs of masked decoders in parallel and reduce the number of cycles. Another alternative is to keep the masked decoder in pipeline with the polynomial arithmetic block. Such type of setting is suitable for systems where many decryption operations are performed in a chain. While the masked decoder works on the coefficients of a previous computation, the polynomial arithmetic unit processes new ciphertexts. Since the masked decoder is faster than the polynomial arithmetic unit, the cycle count of the masked decoder is not an overhead in such type of setting. But of course, in this situation we could not reuse the arithmetic circuitry of the arithmetic coprocessor for the refreshing operation of the masked decoder.

5.5 Maximum frequency

We note that the arithmetic coprocessor is a very optimized unit with a complex pipeline organization. We thus insert two pipeline stages in the masked decoder to match the maximum frequency of the whole system to that of the arithmetic coprocessor. In this way, the design can run up to almost 100 MHz. The critical path is inside the arithmetic multiplier.

6 Software implementation

We wrote a software implementation of the complete system for an ARM Cortex-M4F with the same parameter

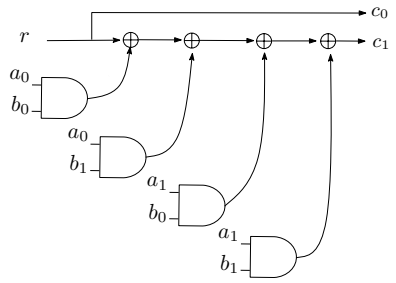

Fig. 4: Trichina AND gate. This masked computes the unshared function $c = ab$. Each variable a is shared into two shares a_1, a_2. This is the construction we use for our secure `AND` instruction in our software implementation.

set as previous section. This Cortex-M4F is a popular and powerful embedded platform. It has a 32-bit word size, 13 general-purpose registers, its instruction set supports single-cycle 32-bit multiplications and 16-bit SIMD arithmetic.

6.1 Arithmetic operations

Our implementation for the arithmetic part (the two branches from Eq. 3 and Eq. 4) follows the lines of de Clercq et al. [9]. We remind here the key ideas of the software implementation. Each coefficient requires 13-bits of storage for $q = 7681$, and we therefore store two coefficient in every processor word. We use the negative-wrapped NTT along with computational optimizations from [29] to implement the polynomial multiplication. We can reduce the number of memory accesses, pointer operations, and loop overhead by 50% by performing a two-fold unrolling of the inner loop of the NTT transformation. The expensive calculation of twiddle factors can be avoided by storing precomputed twiddle factors, and inverse twiddle factors in a lookup table. The code is constant time and constant flow (SPA resistant.) Since each branch operates on only one share, no special protection against DPA is required.

6.2 Masked decoder

Quadrants. The quadrant operation is implemented in a constant-time and constant-flow way. It relies on arithmetic substraction to perform successive comparisions against $q/4$, $q/2$ and $3q/4$. From these comparisions, the quadrant result is constructed by bitmasks. As in the previous paragraph, since each quadrant operates on a single share, no further DPA protection is required.

Table 2: Timings for major operations in software.

Operation	**kCycles**
Equation 3	43
Inverse NTT transform	39
Masked decoder	168

Table lookup. The table lookup is the most sensitive part since it sees both shares q' and q''. We mask the table lookup following [1]. This approach takes as input an unprotected software bitsliced implementation written as a straight-line sequence of `XOR` and `AND` instructions. Then, the input data is shared in a Boolean fashion and the instructions are replaced by its secure equivalent. The masked `XOR` operation is very easy to derive; the masked `AND` instruction is more ellaborate due to the non-linearity of the operation. The dataflow for the `AND` instruction is represented in Figure 4. It is essentially Trichina's masked AND gate.

We wrote the unmasked function that applies the rules of Section 4 (including output feedback) in a bitsliced fashion. We then used espresso [30] and MisII (part of Octtools) for logic minimization and synthesis into `XOR` and `AND` "gates" = instructions. We then substituted the `XOR` and `AND` instructions for its secure equivalents. We perform 32 table lookups (for 32 different coefficients) concurrently, and the decoder always performs 16 iterations. This part (a series of XOR and AND) was prototyped in C and the assembly output carefully inspected.

6.3 Timings

In Table 2 we can see an overview of the time required for each major operation. Note that while the arithmetic part is heavily optimized, we did not focus on achieving the fastest implementation in the masked decoder implementation. The most expensive part of arithmetic computation is the inverse NTT, requiring 39k cycles. The computation of Eq. 3 takes around 43k cycles. The masked decoder takes around 168k cycles. (Most of the time goes to computing the quadrant functions. An assembler version for these functions would greatly benefit the overall timing.) The overhead in cycles for the masked version is around 5.8 times more cycles.

7 Discussion

7.1 Error rates

Cryptosystems based on ring-LWE are inherently probabilistic. This means that there is a non-zero probability

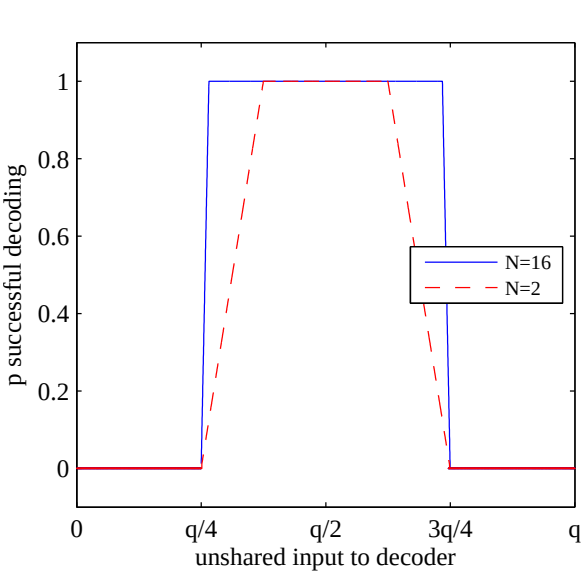

Fig. 5: Empirical success distribution for the masked decoder

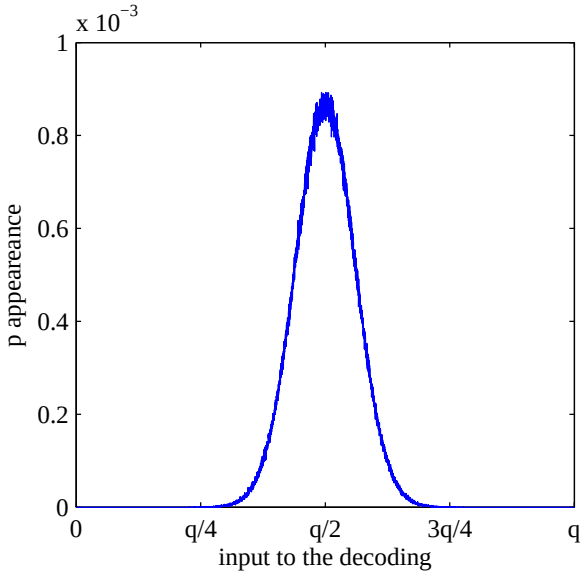

Fig. 6: Distribution of a when plaintext is 1

that the recovered plaintext after ring-LWE decryption is not exactly the plaintext before encryption. In our case, due to the probabilistic nature of our masked decoder approach, there is a second source of noise. Since the number of iterations of the masked decoder is finite, there are some pair values (a', a'') that will not get decoded within the fixed finite number of iterations. In this section, we first explain the error rate of the probabilistic decoding in isolation, and then we switch to the global system error rate and point out strategies to mitigate it.

Errors due to the probabilistic decoding. In this section, we assume that the plaintext bit is 1 and the unmasked input a to the masked decoder is in $(q/4, 3q/4)$. The additional error due to the probabilistic masked decoder is the probability p_e that (a', a'') does not get successfully decoded. Let us write $p_s = 1 - p_e$.

This probability p_s is influenced by two distributions. We have that

$$p_s = \sum \Pr[\text{successful decode}|a] \cdot \Pr[a] \tag{5}$$

where the sum is taken over $a \in (q/4, 3q/4)$. On the one hand, $\Pr[\text{successful decode}|a]$ is the probability that the decoder successfully decodes a. On the other, $\Pr[a]$ is the probability with which a takes various values in $(q/4, 3q/4)$.

The distribution of the decoder success probability $\Pr[\text{successful decode}|a]$ as a function of the unshared input value a to the decoder can be easily computed by averaging over all possible pairs (a', a'') such that $a' + a'' = a$. Since for any given value of a, its shares a' or a'' are (individually) equiprobable, we compute $\Pr[\text{successful decode}|a]$ as $\Pr[\text{successful decode}|a] = \frac{1}{q}\sum_{a'+a''=a} \Pr[\text{successful decode of } (a', a'')]$.

The distribution $\Pr[\text{successful decode}|a]$ is shown in Figure 5. We see that the decoder performs best when $a \approx q/2$, in which case all possible inputs get decoded correctly. Only when the input value a approaches $q/4$ or $3q/4$, the performance degrades. When using a larger number of iterations $N = 16$ this effect is less pronounced when compared to $N = 2$ iterations, as Figure 5 shows.

On the other hand, it is easy to see that not all unshared inputs a to the decoder are equally likely. By the construction of the ring-LWE decryption function, the unshared input to the decoder a is either centered around $q/2$ (resp. 0) when the message bit is 1 (resp. 0). This distribution $\Pr[a]$ is plotted in Figure 6.

These two observations combined produce a nice interaction between the *prior* distribution $\Pr[a]$ of a (given by the ring-LWE decryption) and the success distribution of the masked decoder $\Pr[\text{successful decode}|a]$ as in Equation 5. Namely, values of a that are difficult to decode (those with low $\Pr[\text{successful decode}|a]$) are quite unlikely to appear as input to the masked decoder (their $\Pr[a]$ is also low). This positive interaction keeps the global error rate of the system quite low. This is precisely quantified in the next paragraph.

Global error rate and number of iterations. We performed simulations to estimate the global error rate and

Iterations	p_g $[\times 10^{-5}]$	p_g/p_{baseline}
N =2	332.24	91.41
3	178.44	49.09
4	25.36	6.97
5	20.77	5.71
6	16.22	4.46
8	6.97	1.91
16	4.32	1.19
24	4.06	1.11
30	3.87	1.06

Fig. 7: Global error rates with the probabilistic decoder.

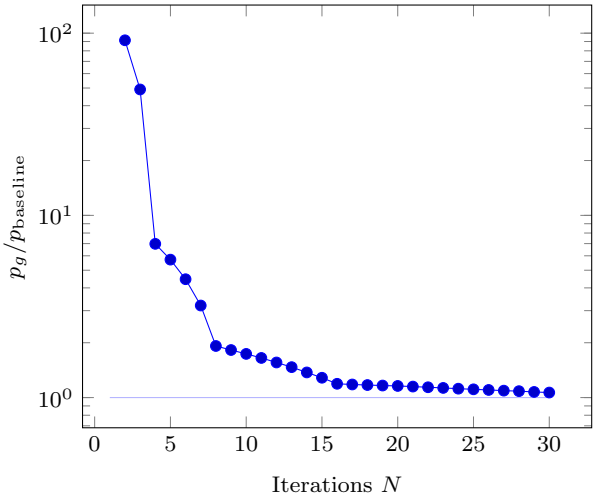

Fig. 8: Evolution of the ratio p_g/p_{baseline} as the number of iterations N grows.

determine the required number of iterations N in our design. Over 10^6 bits, the average error per bit using a deterministic decoder was $p_{\text{baseline}} = 3.634375 \times 10^{-5}$. This is a baseline error intrinsic to the ring-LWE construction. When we plug in the probabilistic decoder, the global, end-to-end, error rate per bit p_g increases. (We have $p_g = p_{\text{baseline}} + p_e$.) In Figure 7, we can find the global error rate for different values of the number of iterations N of the decoding. At $N = 3$, for instance, the error rate is $p_g = 1.7844 \times 10^{-3}$, which is ≈ 49 times larger than p_{baseline}. As already hinted, the error rate quickly decreases with N (roughly exponentially, as can be see in Figure 8). In our design, we set $N = 16$ (we iterate 16 times per coefficient) as a balanced tradeoff between cycle count and error rate. The impact of the masked probabilistic decoder on the global error rate is quite low, adding less than 20% to the intrinsic error rate when compared to a deterministic decoder, as it can be see in Figure 7. We note that one could generalize the masked decoder to trade area for less number of iterations. For details, see Appendix C.

7.2 Comparison with other decoding strategies

We are only aware of a similar masked decoder, the one presented in [5]. There the authors resort to a generic masking method, namely masked tables, to perform the decoding. Translating the ideas of [5] in our context, we would need two tables of 2^{13} bits (one of them random). For a smaller group $\mathbb{Z}_d$ with $d = 257$ the authors report an utilization of 1331 slices on a Virtex 6 FPGA. While the results in slices are not directly comparable with ours, we point out that the size of the masked table following the approach of [5] grows linearly in the group size q, while for our solution the size of the masked table stays constant (independent of q), and the quadrant blocks grow only logarithmically in q. The cycle count, however, is larger in our solution. The critical observation of our masked decoder is that we can *compress* the input coefficient shares a' and a'' to a mere two bit per share (the output of each quadrant) and then perform the decoding based on the information of the two quadrants (4 bits.)

7.3 Post-processing

Albeit the computation from Equation (2) is commonly referred as the "ring-LWE decryption", the decryption process should include a post-processing on the recovered message m. This post-processing consists of error correction and padding verification.

Linear codes with masking. One approach to deal with the probabilistic nature of the ring-LWE decryption system is to use forward error correcting codes (FEC). The message prior to encryption is encoded using a FEC and the resulting composite is ring-LWE encrypted. The output of the ring-LWE decryption should be corrected for errors, preferably in the masked domain. For syndrome decoding of linear codes, this can easily be done by masking the syndrome table. A clever choice of the linear code (for example, perfect codes) can allow very easy masked implementation. (The only perfect linear codes are repetition, Hamming and Golay codes.)

Padding schemes. As presented, the ring-LWE system is malleable. CCA security can be achieved with a padding mechanism. The Fujisaki-Okamoto [12] padding scheme is known to work with ring-LWE [20]. This padding scheme makes use of standard symmetric cryptographic constructions whose masked implementations are well studied. We point out that key-encapsulation mechanisms may result in a more compact and simpler implementation.

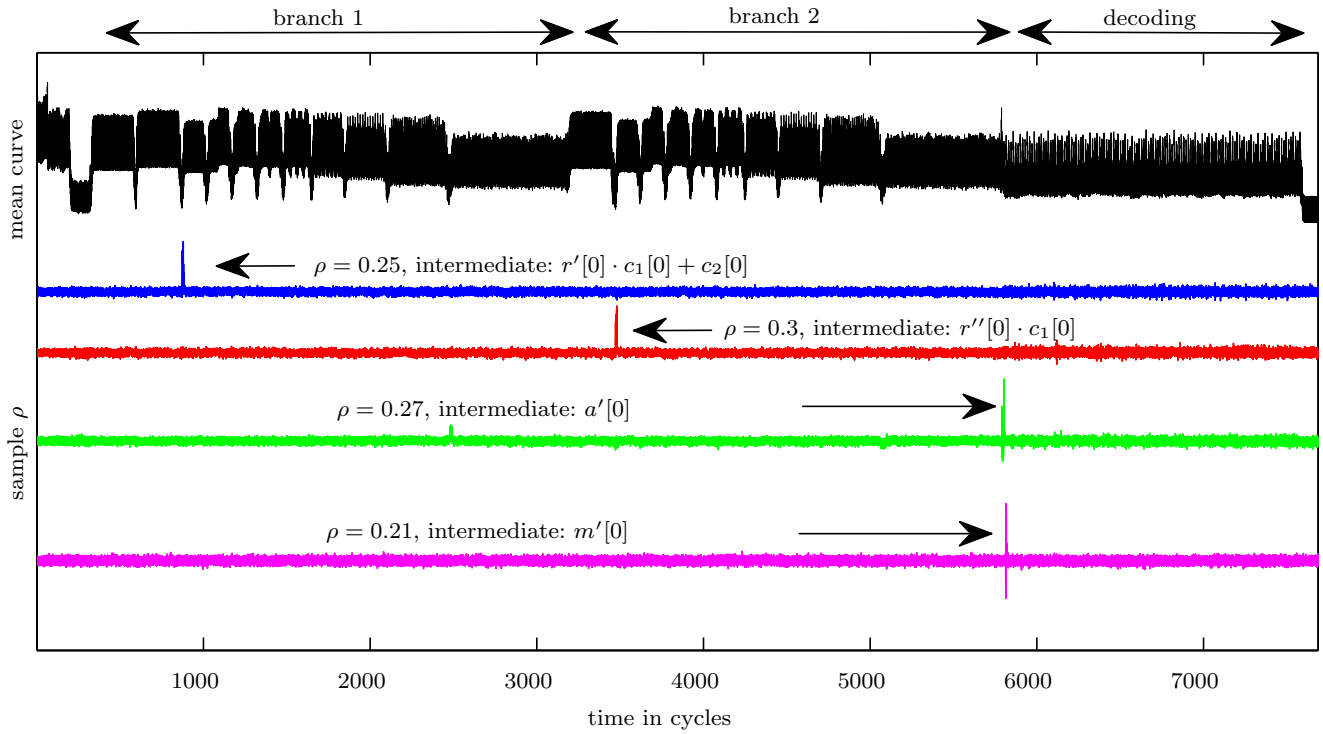

Fig. 9: Hardware implementation. PRNG off. On top, black, one power consumption trace. The different computational stages can be distinguished: first branch, second branch and decoding. Next, in blue, the correlation trace for the value $r'[0] \cdot c_1[0] + c_2[0]$. The correlation achieves a maximum value of $\rho = 0.25$. Below, in red, correlation for $r'' \cdot c_1$ (max $\rho \approx 0.3$); in green: correlation for the input of the masked decoder $a'[0]$. At the bottom: correlation with one message bit $m'[0]$.

We remind that the Fujisaki-Okamoto padding scheme requires a negligible decryption error rate for honestly generated ciphertexts,[3] as explained by Peikert [20]. Thus, the designer must ensure that the global error rate due to the intrisic noise of ring-LWE and the probabilistic decoder is negligible. This can be achieved with FEC as previously described.

A formal generic analysis to choose a FEC code that sets the error rate to, say, 2^{-80} or 2^{-128} is not straightforward. The analysis is greatly simplified if one chooses (n, p, s) parameters such that there is no error contribution due to those parameters and at the same time a required bit security level is maintained. We leave this as future work.

7.4 Extension to higher-order security

We point out that the approach laid out in Section 3 scales quite well with the security order. To achieve security at level $d+1$, one would need to split the computation of Equation 2 into d branches analogously to Equation 3. The masked decoder can follow the same principles with the appropriate modifications. The complexity of this decoder obviously grows. Generic approaches to perform this computation have been discussed in [8, 3, 25].

[3] We would like to thank the anonyomus reviewer for bringing this important issue to our attention.

8 Evaluation

In this section we evaluate both the hardware and the software implementations described above.

We provide a very advantageous setting for the adversary: we assume that the evaluator knows the details about the implementation (for example, pipeline stages and register allocation). In addition, we assume that while guessing a subkey, the adversary knows the rest of the key. These assumptions allow to comfortably place predictions on intermediates arbitrarily deep into the computation. While this may represent a very powerful attacker and somewhat unrealistic, the algebraic structure of such cryptosystem may help the attacker to predict deep intermediates with relatively low effort. In the Appendix B we describe an attack on half-masked

ring-LWE decryption that uses these ideas. This stresses the necessity of masking the decoding function entirely.

The evaluation methodology to test if the masking is sound is as follows. We first proceed with first-order key-recovery attacks when the randomness source (PRNG) is switched off. We demonstrate that in that situation the attacks are successful, indicating that the setup and procedure is sound. Then we switch on the PRNG and repeat the attacks. If the masking is sound, the first-order attacks shall not succeed. In addition, we perform second-order attacks to confirm that the previous first-order analyses were carried out with enough traces.

We modeled the power consumption as the Hamming distance between two consecutive values held in a register, and used Pearson's correlation coefficient to compare predictions with measurements [6].

8.1 Hardware implementation evaluation

Measurement setup. We implemented the full design on a SASEBO G board. The design was clocked at 18.75 MHz and the power consumption was sampled at 500 MS/s. This platform is very low noise.

We test 4 different points which covers all the relevant parts of the computation. The targets are the first 13-bit coefficient of $r' \cdot c_1 + c_2$, the first 13-bit coefficient of $r'' \cdot c_1$, the first input coefficient to the shared decoder and the first output bit.

PRNG off. We first begin the experiments when the PRNG is off. That is, the sharing of r into r' and r'' on each execution is deterministic. This would not happen in practice, as an active PRNG would randomize the representation of r in each execution. In our setting, this would mean that the masking is switched off.

In Figure 9 we draw the result of correlating against the 4 intermediates with 10 000 traces. On top, we draw a mean trace for orientation. The correlation values are, from top to bottom, 0.25, 0.3, 0.27 and 0.21, respectively. This means that the attacks are successful, and confirms the soundness of our setting. In Figure 10 we can see the evolution of the correlation coefficient as the number of traces increases for the first two intermediates. We can see that starting from hundred traces the attack is successful. Similar behavior was observed for other intermediates.

PRNG on. In Figure 11 we draw the result of the previous analysis when the masks are switched on. This corresponds to the situation that an adversary would face in reality. We can see that the correct key guess is no longer distinguishable, even when using 10 000 traces.

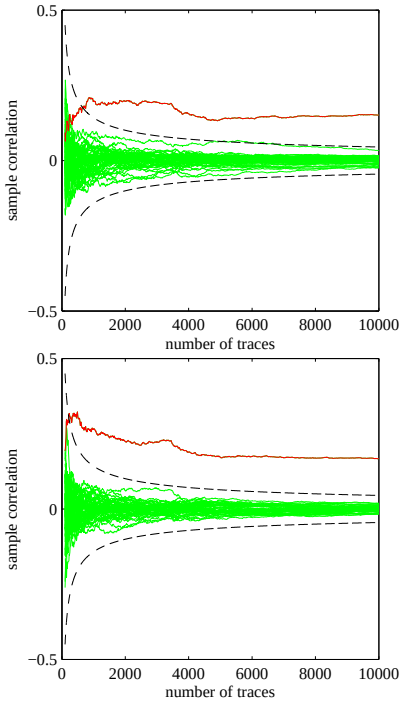

Fig. 10: Hardware implementation. PRNG off. Evolution of the correlation coefficient as the number of traces increases for the intermediates $r'[0] \cdot c_1[0] + c_2[0]$ (left) and $r''[0] \cdot c_1[0]$ (right). Correct subkey guess in red, all other guesses in green. A 99.99 % confidence interval for $\rho = 0$ is plotted in black discontinuous line. We can see that starting from hundred measurements the attacks are successful.

We repeated the same experiments for other intermediates and other intermediate positions with identical results.

Second-order attacks. To confirm that we used enough traces in our previous analyses, we perform here second-order attacks on the masked implementation with the PRNG on. We will focus on the masked decoder. In Figure 12 we draw on top a mean curve in the region of 7 400 to 7 700 cycles, corresponding to the end of the masked decoding. We target one output bit of the decoding: $m[254]$.

In Figure 12 we first begin by correlating against masks and masked values. This is a test scenario, since for this attack we need to know the masks, something that would not happen in a real deployment. Correlation with masks or masked value yield high correlation as expected ($\rho = 0.32$ and $\rho = 0.34$, respectively). In contrast, when correlating against the unshared value (in light blue), the correlation coefficient does not traverse the confidence interval for $\rho = 0$. This indicates that

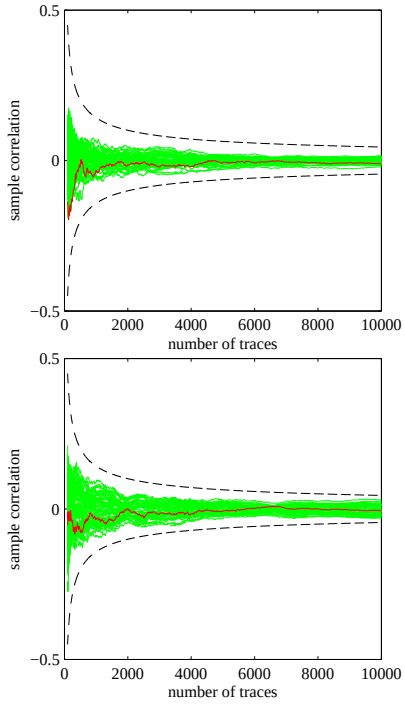

Fig. 11: Hardware implementation. Analogous to Figure 10, but with PRNG on. The correct subkey is no longer identifiable. This is expected and means that the masking is effective.

the masking is effective. We can repeat the same attack against centered and squared traces [7, 22]. This is effectively a second-order attack, and is expected to work. It is shown in magenta in Figure 12, and we can see that the attack succeeds. Using the centered absolute value to pre-process traces also works as expected, as shown in yellow.

In Figure 13 we can see the evolution as a function of the number of traces. We can see that starting from ≈ 2000 measurements this second-order attack is successful. This confirms that the first-order attacks from above were carried out with enough traces, since a second-order attack is already successful starting from ≈ 2000 measurements.

We remark that the relatively low number of traces required for the second-order attack is due to the very friendly scenario for the evaluator. The platform is low noise and no other countermeasure except than masking was implemented. In practice, masking needs a source of noise to be effective, and consequently the higher-order attacks would be harder to mount, requiring more traces [7] and more computation [26].

8.2 Software implementation evaluation

Measurement setup. We deployed the masked software implementation on a 32-bit ARM STM32F407VG Cortex-M4. The MCU operates at 168 MHz and has 192 kB of SRAM. We take contactless power measurements from a decoupling capacitor in the power loop with a Langer LF2-5 H-field probe and 20 dB amplification. This lab setup is very low-noise. DPA on an unprotected byte-oriented AES succeeds with 20 traces. We focus the evaluation on the most challenging part: the masked decoding operation.

Masks off. Figure 15 shows successful correlations when the adversary knows the secret PRNG seed. This serves to confirm that our setup is sound. We selected many different intermediates within the table lookup operation and used 20k traces to produce a good-looking picture. The maximum absolute value for the correlation against the correct key hypothesis is around $|\rho| \approx 0.71$. In Figure 16, top, we see the evolution of sample correlation coefficient as the number of curves at timesample 1390. We can see that starting from less than hundred traces the attack is successful, since the correct subkey stands out from all other competing key hypotheses.

Masks on. When the PRNG output is unknown, first-order attacks are expected not to work. This is the case in our implementation. In Figure 16, middle, the evolution of the correlation coefficient is plotted at the same timesample 1390. The correct subkey is indistinguishable among competing ones. Similar observations apply to the entire timespan.

Second-order attacks. We also performed second-order attacks. Note that our implementation does not claim second-order security. One can see from Figure 16, bottom, that second-order attacks begin to work from a couple hundred measurements. This means that the previous analyses were carried out with enough number of measurements (up to 20k measurements.) Similar observations apply here: our software setting is very friendly towards the evaluator since there is no additional noise present in the measurements. In reality, one would always implement masking along with a source of noise to be effective.

8.3 Horizontal DPA attacks

During the decoder operation, the input coefficients are refreshed $N - 1 = 15$ times with publicly known offsets Δ_i. The device thus handles consecutively the values a', $a' + \Delta_1$, ..., $a' + \Delta_1 + \ldots + \Delta_{15}$. This may enable

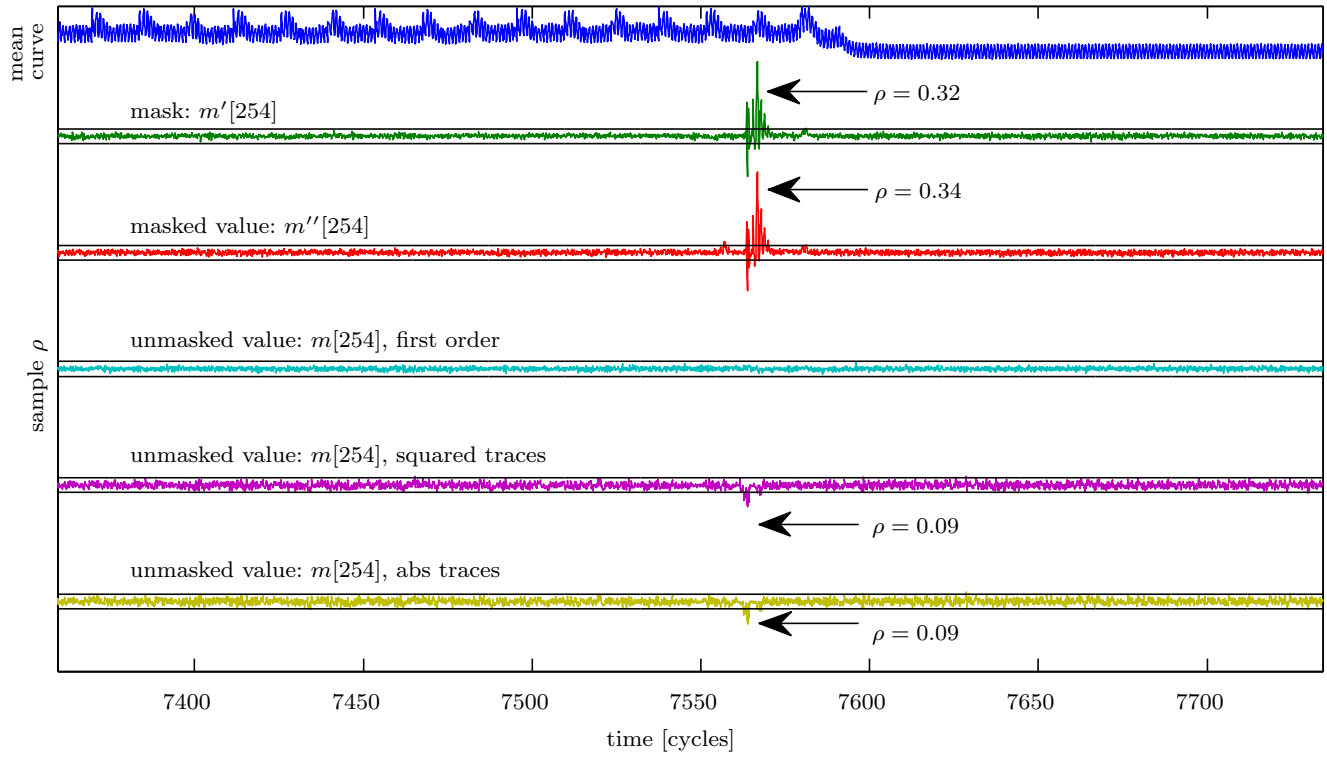

Fig. 12: Hardware implementation. Correlation traces for intermediates within the shared decoder. On top, a power measurement trace showing the last 15 decodings. Below, correlation traces. The first two (masks and masked values) assume that the adversary knows the masks. The third one, in light blue, is a first-order attack without knowing the attack, and is unsuccessful. In contrast, the second-order attack against the same intermediate is successful, as the traces in magenta and yellow show.

a horizontal DPA attack [19] during the operation: the adversary may collect a single trace, split it into 16 chunks and then perform a DPA on these 16 chunks to recover the mask a'. Once the masks from all traces are discovered, a first-order, vertical DPA applies.

There are two factors that mitigate this threat. First, we note the adversary is given a very limited number of traces to recover each mask (namely, $N = 16$). Secondly, this attack can be easily prevented by shuffling the public coefficients Δ_i. This randomizes the order of execution of each refreshing with Δ_i, and thus the exposure to horizontal DPA attacks is minimized.

9 Conclusion

In this paper we described a practical side-channel protected implementation of the lattice-based ring-LWE asymmetric decryption. Our solution is based on the sound principles of masking and incurs in a manageable overhead (in cycles and area). A key component of our solution is a bespoke masked decoder. Our implementation performs the entire ring-LWE decryption computation in the masked domain.

Acknowledgements. The authors would like to thank the CHES 2015 reviewers for their valuable comments. This work has been supported in part by the European Commission through the ICT programme under contracts H2020-ICT-645622 PQCRYPTO, H2020-ICT-644209 HEAT and FP7-ICT-2013-10-SEP-210076296 PRACTICE; by the Research Council KU Leuven TENSE (GOA/11/007); by the Flemish Government FWO G.0550.12N, G.00130.13N and G.0876.14N; and by the Hercules Foundation AKUL/11/19. Oscar Reparaz is funded by a PhD fellowship of the Fund for Scientific Research - Flanders (FWO). Sujoy Sinha Roy was supported by Erasmus Mundus PhD Scholarship.

References

1. Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. Dpa, bitslicing and masking at 1 ghz. In Güneysu and Handschuh [15], pages 599–619.
2. Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post Quantum Cryptography*. Springer, 1st edition, 2008.

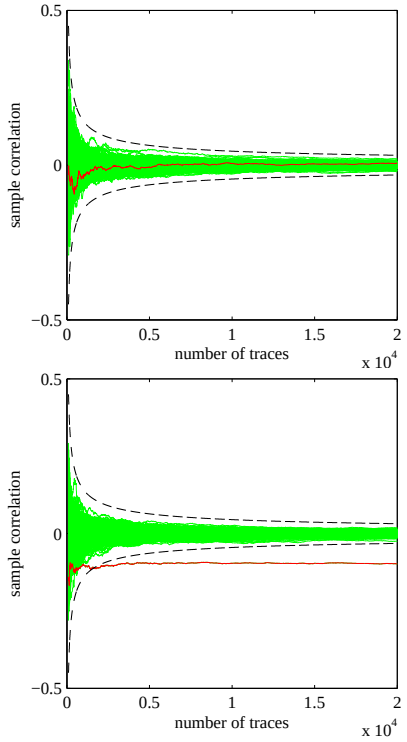

Fig. 13: Hardware implementation. Top: correlation as the number of traces increases for the first-order attack (PRNG on), around clock cycle 7560. Bottom: correlation for the second-order attack with masks on. The attack begins to be successful with 2 000 measurements.

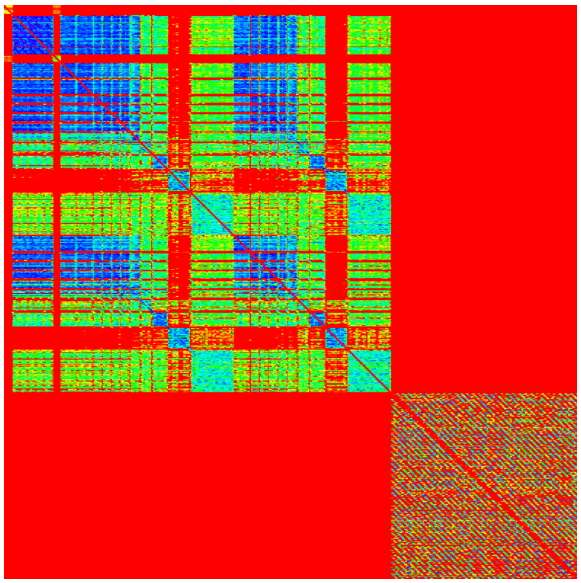

Fig. 14: Hardware implementation. Crosscorrelation trace. The x and y axes represent time, flowing from the upper left hand side corner to the lower right. The entire figure spans 7500 cycles (as Figure 9). It is possible to distinguish the two branch computations (including its components) and the decoding. Colors enhanced to improve contrast.

3. Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-order threshold implementations. In *ASIACRYPT*, volume 8874 of *LNCS*, pages 326–343. Springer, 2014.
4. Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding*, volume 8308 of *LNCS*, pages 45–64. Springer, 2013.
5. Hai Brenner, Lubos Gaspar, Gaëtan Leurent, Alon Rosen, and François-Xavier Standaert. FPGA implementations of SPRING - and their countermeasures against side-channel attacks. In *CHES*, volume 8731 of *LNCS*, pages 414–432. Springer, 2014.
6. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, 2004.
7. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
8. Jean-Sébastien Coron. Higher order masking of look-up tables. In *EUROCRYPT*, volume 8441 of *LNCS*, pages 441–458. Springer, 2014.
9. Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Efficient software implementation of ring-LWE encryption. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 339–344. ACM, 2015.
10. Léo Ducas, Alain Durmus, Tancréde Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *CRYPTO*, volume 8042 of *LNCS*, pages 40–56. Springer, 2013.
11. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `http://eprint.iacr.org/`.
12. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, 2013.
13. Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *CHES*, volume 7428 of *LNCS*, pages 512–529. Springer, 2012.
14. Louis Goubin and Jacques Patarin. DES and differential power analysis the duplication method. In *CHES*, volume 1717 of *LNCS*, pages 158–172. Springer, 1999.
15. Tim Güneysu and Helena Handschuh, editors. *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*. Springer, 2015.
16. Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
17. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
18. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EU-*

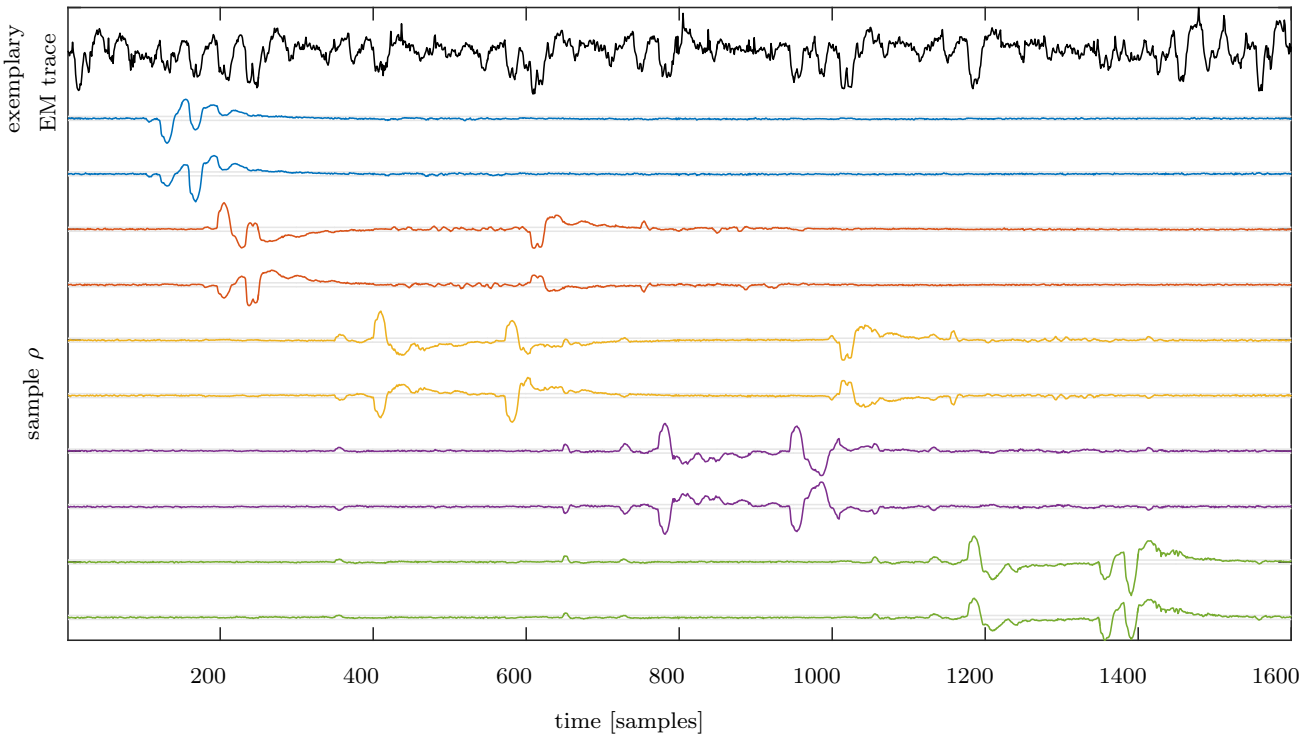

Fig. 15: Software implementation. PRNG off. On top, black, one EM consumption trace. The selected region comprises one masked bitsliced table lookup. Below, different correlation traces for various intermediates. Correlation for different shares of the same intermediate are drawn in the same color. The 99.99 % confidence interval for $\rho = 0$ is drawn in light grey.

ROCRYPT, volume 6110 of *LNCS*, pages 1–23. Springer, 2010. Full Version available at Cryptology ePrint Archive, Report 2012/230.

19. J. Pan, J.I. den Hartog, and Jiqiang Lu. You cannot hide behind the mask: Power analysis on a provably secure s-box implementation. In *Information Security Applications*, volume 5932 of *LNCS*, pages 178–192. Springer, 2009.
20. Chris Peikert. Lattice cryptography for the internet. In *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, pages 197–219, 2014.
21. Thomas Pöppelmann and Tim Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In *Selected Areas in Cryptography – SAC 2013*, volume 8282 of *LNCS*, pages 68–85. Springer, 2014.
22. E. Prouff, M. Rivain, and R. Bevan. Statistical analysis of second order differential power analysis. *Computers, IEEE Transactions on*, 58(6):799–811, June 2009.
23. Chester Rebeiro, Sujoy Sinha Roy, and Debdeep Mukhopadhyay. Pushing the limits of high-speed GF(2^m) elliptic curve scalar multiplication on fpgas. In *CHES*, volume 7428 of *LNCS*, pages 494–511. Springer, 2012.
24. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.
25. Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *CRYPTO*, volume 9215 of *LNCS*, pages 764–783. Springer, 2015.
26. Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Selecting time samples for multivariate DPA attacks. In *CHES*, volume 7428 of *LNCS*, pages 155–174. Springer, 2012.
27. Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. A masked ring-lwe implementation. In Güneysu and Handschuh [15], pages 683–702.
28. Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Compact and side channel secure discrete gaussian sampling. *IACR Cryptology ePrint Archive*, 2014:591, 2014.
29. Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact ring-lwe cryptoprocessor. In *CHES*, volume 8731 of *LNCS*, pages 371–391. Springer, 2014.
30. Richard L Rudell. Multiple-valued logic minimization for pla synthesis. Technical report, DTIC Document, 1986.
31. E.V. Trichina. Table lookup operation on masked data, 2013. US Patent 8,422,668.
32. Michael Tunstall, Neil Hanley, Robert P McEvoy, Claire Whelan, Colin C Murphy, and William P Marnane. Correlation power analysis of large word sizes. *IET Irish Signals and Systems Conference (ISSC) 2007*, Septem-

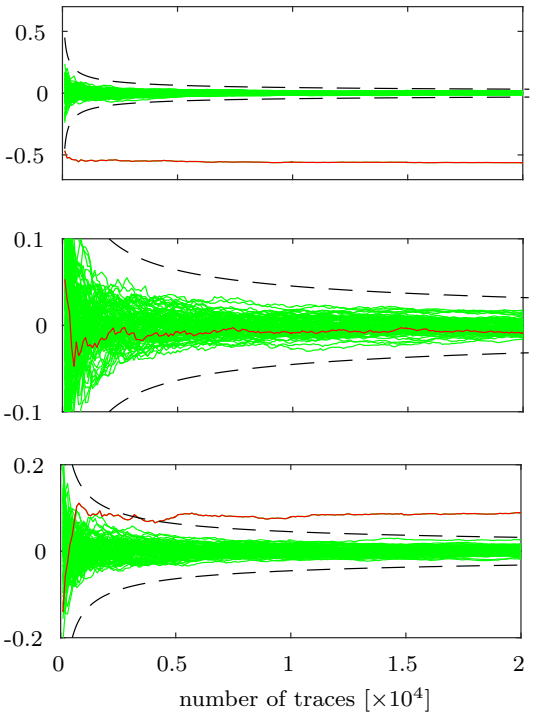

Fig. 16: Software implementation. Evolution of Pearson's correlation coefficient with number of traces for different attacks at timesample 1390. On top: (successful) first-order attack with PRNG off. Middle: (unsuccessful) first-order attack with PRNG on. Bottom: (successful) second-order attack with PRNG on. Correct subkey in red, incorrect in green. We also plot the 99.99 % confidence interval for $\rho = 0$ in dashed line.

ber 2007. Available at `http://www.cs.bris.ac.uk/home/tunstall/papers/THMWMM.pdf`.

a'	a''							
	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
$I_1 = (0, q/8)$	✓	×	✓	✓	✓	×	✓	✓
$I_2 = (q/8, 2q/8)$	×	✓	✓	✓	×	✓	✓	✓
$I_3 = (2q/8, 3q/8)$	✓	✓	✓	×	✓	✓	✓	×
$I_4 = (3q/8, 4q/8)$	✓	✓	×	✓	✓	✓	×	✓
$I_5 = (4q/8, 5q/8)$	✓	×	✓	✓	✓	×	✓	✓
$I_6 = (5q/8, 6q/8)$	×	✓	✓	✓	×	✓	✓	✓
$I_7 = (6q/8, 7q/8)$	✓	✓	✓	×	✓	✓	✓	×
$I_8 = (7q/8, 8q/8)$	✓	✓	×	✓	✓	✓	×	✓

Table 3: The rules for octant-decoding. The cases where no rule is hit are marked with ×.

A Optimal values of Δ_i for $q = 7681$

$$\begin{aligned}\Delta(i) = (&960, 1440, 480, 1680, 240, 720, 1200, 1800, && (6)\\ &120, 360, 600, 840, 1080, 1320, 1560, 1860, && (7)\\ &60, 180, 300, 420, 540, 660, 780, 900, 1020, && (8)\\ &1140, 1260, 1380, 1500, 1620, 1740, 1890, && (9)\\ &30, 90, 150, 210, 270, 330, 390, 450, 510, && (10)\\ &570, 630, 690, 750, 810, 870, 930, 990, 1050, && (11)\\ &1110, 1170, 1230) && (12)\end{aligned}$$

These values were found by exhaustive first-order search. The value Δ_i is chosen so that it maximizes the number of pairs that get decoded after i iterations.

B Attack on half-masked variant

In this section, we analyze the security of a masked ring-LWE variant where the intermediates just before decoding are unmasked, and the decoding is performed in the unmasked domain. This alternative is definitely cheaper than full masking. In the following, we provide evidence to show that this clearly does not provide enough security in our case.

(A seemingly similar situation appears in [5]. However, there are important differences—namely it is not possible to choose ciphertexts. In the following, we are not analyzing the variant of [5] but only the half-masked ring-LWE.)

A common argument is that after key-diffusion is complete, prediction of the intermediates is not possible and hence standard DPA attacks to the half-masked ring-LWE do not apply. We will see that this is not strictly true, if the attacker can choose ciphertexts.

Assume that the coefficients of the polynomial $a = \mathrm{INTT}(r \cdot c_1 + c_2)$ appear unmasked in the implementation. Let the adversary collect measurements with chosen ciphertext. The ciphertext c_1 has the following structure: all the coefficients fixed except $c_1[0]$ randomly varying. The ciphertext c_2 has the same structure. Then observe that due to linearity of the INTT operation, $a[0]$ can be written as $a[0] = \alpha(r[0] \cdot c_1[0] + c_2[0]) + \beta$, where

- α is a public constant determined by the INTT transformation.
- β is a secret constant that is a function of the other (unknown) key coefficients $r[1], \ldots, r[255]$. Note that by construction β is constant within the set of collected traces.

Thus, an attacker can perform a DPA attack targeting the intermediate $a[0]$ and placing predictions on $(r[0], \beta)$. The adversary recovers $r[0]$ and proceeds to recover other key coefficients. We have verified this attack in simulations, even when using $\mathrm{th}(a[i])$ as intermediate.

(It may seem that the high number of hypotheses, 2^{26} may produce a cumbersome attack. However, one can apply techniques of partial correlation [6] to alleviate the computational effort of DPA on large word sizes [32]. And we have experimented that in practice it makes sense to first recover $r[0]$ (this is easier due to larger non-linearity of the modular multiplication) and then β (which may be harder due to the low non-linearity of the modular addition), splitting the 2^{26} effort in two 2^{13} steps.)

C Generalization of the decoding scheme

The probability of not hitting any rule can be reduced by increasing the number of rules, i.e. by splitting the domain of decoding into more than four sections. For example, in Table 3 the rules are shown for the case when the decoding domain is split into eight sections or octant. As seen from the table, the probability of not hitting a rule has reduced to 1/4. Hence to meet a same decryption failure rate, an octant decoder needs almost half the number of iterations as required by a quad decoder. However there are overheads associated with an octant decoder when it is compared to a quad decoder: the number of comparisons to locate the position of a coefficient in the octant chart doubles and the sizes of the tables quadruples.