Lightweight and Fault-Resilient Implementations of Binary Ring-LWE for IoT Devices

Shahriar Ebrahimi and Siavash Bayat-Sarmadi, *Member, IEEE*

***Abstract*—While the Internet of Things (IoT) shapes the future of the Internet, communications among nodes must be secured by employing cryptographic schemes such as public-key encryption (PKE). However, classic PKE schemes, such as RSA and elliptic curve cryptography (ECC) suffer from both high complexity and vulnerability to quantum attacks. During the past decade, post-quantum schemes based on the learning with errors (LWEs) problem have gained high attention due to the lower complexity among PKE schemes. In addition to resistance against theoretical (quantum and classic) attacks, every practical implementation of any cryptosystem must also be evaluated against different side-channel attacks such as power analysis or fault injection ones. In this article, we analyze the vulnerability of binary ring learning with error (Ring-LWE) scheme regarding (first-order) fault attacks, such as randomization, zeroing, and skipping faults. We show that previous implementations can be easily broken by employing such fault attacks. Moreover, we propose fault-resilient software implementations of binary Ring-LWE on 8- and 32-b lightweight microcontrollers, namely, AVR ATxmega128A1 and ARM Cortex-M0 that are ideal for IoT devices. Furthermore, we formally prove the resilience of the proposed implementations against different fault attacks. To the best of our knowledge, this article is the first one to propose fault-resilient binary Ring-LWE implementations on resource-constrained microcontrollers. Our implementations on AVR ATxmega128A1 require only 80 and 120 ms for encryption and decryption, respectively.**

***Index Terms*—Internet of Things (IoT), lattice-based cryptography, post-quantum cryptography, ring learning with errors (Ring-LWEs), software implementation.**

I. Introduction

INTERNET of Things (IoT) expands the architecture of the traditional Internet network by connecting smart devices, equipped with sensors and actuators, to computing systems. Therefore, IoT has found its way into various applications, such as smart city, connected vehicles, and e-Health. As more devices connect to the IoT network, a higher number of security threats appear that can result in even vital tragedies in applications such as e-health [1]. Thus, it is necessary to provide secure communication infrastructure for the IoT devices to preserve the privacy and confidentiality of the user and data against threats.

Manuscript received February 26, 2020; accepted March 2, 2020. Date of publication March 9, 2020; date of current version August 12, 2020. This work was supported by the Sharif University of Technology under Grant G960803. *(Corresponding author: Siavash Bayat-Sarmadi.)*

The authors are with the Department of Computer Engineering, Sharif University of Technology, Tehran 11365-11155, Iran (e-mail: shebrahimi@ce.sharif.edu; sbayat@sharif.edu).

Digital Object Identifier 10.1109/JIOT.2020.2979318

In order to establish secure communication between two parties or provide information security for sensitive data, various protocols are available in different layers of the network. Some protocols are in the physical layer that benefits from low complexity and overhead [2], [3], while more general approaches, such as public-key cryptography (PKE) are employed in building blocks of widely used application-layer protocols, such as TLS, SSL, and HTTPS. Public-key encryption (PKE) schemes have multiple applications and are considered as the main approach for secure key-exchange and channel establishment in different protocols. However, classic PKEs, namely, RSA [4] and elliptic curve cryptography (ECC) [5], have high complexity to be efficiently implemented on resource-constrained IoT devices [6], [7]. On the other hand, with recent advances in quantum computers and the vulnerability of classic PKEs to quantum attacks based on Shor's algorithm [8], it is necessary to consider alternative cryptosystems for post-quantum era [6], [9]–[11].

Among current post-quantum schemes, the lattice-based cryptography and especially learning with errors (LWEs) problem (and its variants such as ring learning with error (Ring-LWE) [12]) have gained high attention in the standardization process held by the National Institute of Standard and Technology (NIST) [13]. This is mainly due to the lower complexity compared to other PKE schemes, which also makes lattice-based PKE a proper candidate for resource-constrained nodes in IoT [6], [9], [10], [12].

Regev [14] proposed the LWE problem and proved that it is reducible to hard lattice problems, such as shortest and closest vectors (SVP and CVP) [15]. The results from [14] suggest that secure cryptosystems can be built based on LWE, which is on average as hard as worst case lattice problems. In 2010, Lyubashevsky *et al.* proposed Ring-LWE by employing ideal lattices in LWE to achieve lower complexity while providing the same level of security. Ring-LWE and its variants are shown to have lower complexity compared to previous PKE schemes [10], [12], [16]. In 2016, a variant of Ring-LWE cryptosystem has been proposed that uses binary error distribution instead of the Gaussian one, which results in even smaller key sizes and more practicality for IoT applications [10]. More details regarding security and complexity analysis of the binary Ring-LWE hereafter referred to as Ring-BinLWE, are discussed in [9] and [10].

Besides theoretical security, every implementation of any cryptosystem should also be evaluated against different side-channel analysis such as fault injection attacks [17] that target not the scheme but the implementation itself [18]–[20].

Such attacks have been a threat to cryptographic implementations since 1970s and it is proven that different LWE-based cryptosystems are vulnerable against simple but efficient fault injection attacks [18]–[20]. On the other hand, various studies indicate that the distributed and remote nature of IoT devices provides the adversaries with the time and physical access to manipulate any remote node [1], [9]. Therefore, depending on the budget and the adversary's expertise, different attacks can be performed on a cryptographic implementation.

In this article, we analyze the possibility of first-order fault attacks on implementations of Ring-BinLWE from [10]. We show that the adversary can easily break the scheme and reveal the secret data by employing only first-order fault attacks. While some of the fault attacks can be encountered by different verification in the implementation itself, there are a few fault attacks that can be mapped into simple and adaptive-chosen ciphertext attacks (CCA and CCA2) that threat the Ring-BinLWE scheme itself [21]. Therefore, to provide fault-resilient Ring-BinLWE, it is necessary to encounter fault attacks in both scheme and implementation levels [18]–[21].

In this article, we aim to propose different fault-resilient software implementations of the Ring-BinLWE scheme. Therefore, we also propose a CCA2-secure variant of the Ring-BinLWE scheme based on the method from [21]. Furthermore, we formally prove the security of our implementations against fault attacks. In terms of efficiency, we have optimized the proposed implementation for two resource-constrained microcontrollers: 1) 8-b AVR ATxmega128A1 and 2) 32-b ARM Cortex-M0. The selected microcontrollers have limited computation resources to assure that our implementations can be practical and scalable to even tiny and low-power IoT development boards, such as Adafruit's Trinket [22], Arduino's Zero [23], and Texas Instruments' TmoteSky [24].

The main contributions of this article are as follows.

1) We evaluate Ring-BinLWE scheme and implementations from [10] regarding fault attacks. We show that an adversary can easily break the cryptosystem using three types of (first-order) fault injection attacks: a) skipping; b) randomization; and c) zeroing.
2) We propose fault-resilient implementations of Ring-BinLWE. Moreover, we provide security analysis and formally prove such a claim.
3) Our fault-resilient implementations of Ring-BinLWE on both AVR and ARM microcontrollers consume acceptable clock cycles based on IoT criteria. The complete execution of the proposed fault-resilient encryption and decryption phases on ATxmega128A1 require only 80 and 120 μs, respectively.

The remainder of this article is organized as follows. We describe the necessary background in Section II. Section III provides the detailed information on practical fault attacks on Ring-BinLWE. The proposed fault-resilient implementations along with the corresponding security proof are described in Section IV. We provide details of our implementations in Section V. Finally, this article is concluded in Section VI.

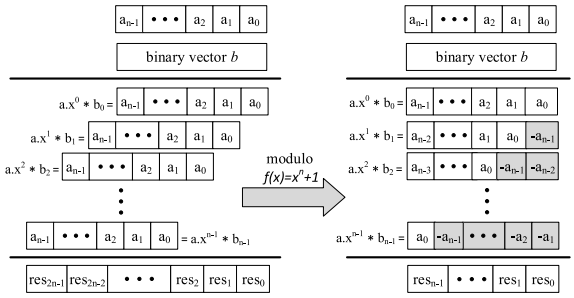

Fig. 1. Normal versus modular polynomial *shift-and-add* multiplication with the modulus $f(x) = x^n + 1$ as used in Ring-BinLWE [10].

II. Preliminaries

Regev [14] defined the hard problem of LWE and proposed the first LWE-based cryptosystem. The problem has two versions (search and decision) that are proven to be as hard as worst case lattice problems on average case. In the search version of LWE, one has to find the secret $s \in \mathbb{Z}_q^n$ in many given pairs of (a, b_i) such that $b_i = as + e_i$, while $a \in \mathbb{Z}_q^n$ is known and uniformly random and $e_i \in \mathbb{Z}_q$ are error vectors driven by a distribution ψ (such as Gaussian [14]) over $\mathbb{Z}_q$. In decision version of the LWE problem, one has to distinguish between uniformly random set $R = \{\bigcup r_i \leftarrow \mathbb{Z}_q^n\}$ and the LWE-based set $B = \{\bigcup bi = a.s + e_i | e_i \leftarrow \psi\}$.

In 2010, a more practical version of LWE, namely, Ring-LWE, has been proposed that utilizes ideal lattices [12]. In Ring-LWE, the coefficients of polynomials are chosen from the ring $\mathbb{Z}_q = (-\lfloor q/2 \rfloor + 1, \lfloor q/2 \rfloor)$. Moreover, each polynomial belongs to the poly ring of $\mathcal{R}_q = \mathbb{Z}_q[x]/f(x)$. In [12], the modulus $f(x)$ has been chosen as $f(x) = x^n + 1$, therefore, every multiplication of an x to the polynomial, would result in an anti-circular rotation of coefficients [12]. This feature makes Ring-LWE and its variants to be more efficient for implementation.

Buchmann *et al.* [10] have shown that instantiating Ring-LWE over binary errors is still secure and results in a more lightweight and compact public-key scheme compared to the standard Ring-LWE over the Gaussian distribution. Table I provides detailed information regarding required computations in both Ring-LWE and Ring-BinLWE, respectively. Moreover, the multiplication in Ring-BinLWE can be implemented by the *shift_and_add* algorithm [10] as shown in Fig. 1. In [9], it is shown that no reduction in the entire scheme operations is required, when Ring-BinLWE instances are presented in 2s complement notation. Utilizing the binary error distribution in Ring-BinLWE decreases key and ciphertext sizes [10], which makes Ring-BinLWE a better match for resource-constrained devices in IoT [6], [9]. According to the latest results in [25], choosing a parameter set of $n = 256$ and $q = 256$ provides 84 and 73 b of classic and quantum security level, respectively.

III. Fault Attacks on Ring-BinLWE

Fault injection attacks can target both data and/or control unit in a system [18]–[20]. During a fault attack, the adversary injects an error into a memory or a signal to exploit the faulty execution of the function and extract the secret data

TABLE I
RING-LWE AND RING-BINLWE PKE SCHEMES

Scheme	Key Generation		Encryption		Decryption
	secret key	public public	c_1	c_2	
Ring-LWE [12] $q = 7681, 12289$	$sk \leftarrow \varphi \in \mathcal{R}_q$	$e \leftarrow \varphi \in \mathcal{R}_q$ $pk = a.sk + e$	$r, e_1 \leftarrow \varphi \in \mathcal{R}_q$ $c_1 = a.r + e_1$	$e_2 \leftarrow \varphi \in \mathcal{R}_q$ $c_2 = pk.r + e_2 + \lfloor q/2 \rceil .m$	$\bar{m} \approx c_2 - c_1.sk \in \mathcal{R}_q$
Ring-BinLWE [10] $q = 256$	$sk \leftarrow \{0,1\}^n$	$e \leftarrow \{0,1\}^n$ $pk = e - a.sk$	$e_1, e_2 \leftarrow \{0,1\}^n$ $c_1 = a.e_1 + e_2$	$e_3 \leftarrow \{0,1\}^n$ $c_2 = pk.e_1 + e_3 + \lfloor q/2 \rceil .m$	$\bar{m} \approx c_2 + c_1.sk \in \mathcal{R}_q$

such as an unencrypted message or a private key. Various techniques exist to inject faults to a running software, ranging from underpowering the entire microcontroller to employing delicate fault injection lasers [19]. Therefore, it is necessary to consider such attacks while implementing any cryptographic scheme. On the other hand, due to the remote and distributed nature of the IoT network, fault injection attacks must be analyzed in order to secure the implementation of any cryptosystem [18]–[20].

In this article, we assume that the adversary can compromise a device by employing different fault attacks as follows.

1) *Randomization:* Setting part of the memory to random values, which are not even known to the adversary.
2) *Skipping:* Skipping certain instructions of the code under execution by avoiding different conditional checks or manipulating loop counters.
3) *Zeroing:* Setting parts of (or the entire) variable to zero.

Note that our attack model does not cover attackers who can:

1) modify the code execution in the processor instruction level. Such an attacker has the power of dumping EEPROM to obtain the secret key without any required fault attack [26];
2) inject permanent fault on *multiple* input elements of functions that break the integrity of any fault countermeasures [26].

Contrary to previous work, such as [18], [19], [26], and [27], in our attack model, the adversary has the power to change the Boolean result of a "conditional check" to make it ineffective. Therefore, we employ *unconditional infections* instead of common *if/else*-like checks to encounter such powerful fault attacks.

A. Fault Attacks on Key Generation

During the key generation phase, one must calculate $pk = r_1 - a.r_2$ as depicted in Table I. The adversary can manipulate the function to produce weak keys by employing zeroing or skipping faults. Randomization faults are not beneficial during the key generation phase; due to the fact that the adversary will not obtain any usable information. Table II shows detailed information regarding fault attacks on three phases of Ring-BinLWE.

1) Zeroing Faults: The adversary can benefit from setting parts of (or the entire) r_1, r_2, or a to zero, which results in $pk = -a.r_2$, $sk = 0$ or $pk = r_1$, respectively.

2) Skipping Faults: The adversary may skip the subtraction or parts of *shift_and_add* multiplication, which result in $pk = a.r_2$, or $pk = r_1$, respectively.

TABLE II
SUMMARY OF FAULT ATTACKS ON RING-BINLWE

Phase	Attack	Result
Key generation	$r_1 = 0$ or skip_add	weak $pk \Rightarrow sk$ recovery
	$r_2 = 0$	$sk = 0$ and weak pk
	skip_mult	weak $pk \Rightarrow sk$ recovery
Encryption	$e_1 = 0$	$\bar{m}$ recovery
	$e_2 = 0$	weak $c_1 \Rightarrow \bar{m}$ recovery
	$pk.e_1 = 0$ or skip_mult	$\bar{m}$ recovery
Decryption	$c_1 = 0$ or skip_add	sk recovery
	zeroing parts of r_2 or c_2	
	randomizing parts of r_2	

B. Fault Attacks on Encryption

The adversary can recover the message from faulty encryption execution. As same as key generation, randomization faults will not directly result in any secret data recovery.

1) Zeroing Faults: The adversary can target any of e_1, e_2, or even pk to inject zeroing attacks, which result in $c_2 = e_3 + \bar{m}$ or $c_1 = a.e_1$ to extract $\bar{m}$ from the faulty ciphertexts.

2) Skipping Faults: The adversary benefits from skipping different calculations during the *encryption* phase to produce faulty ciphertexts, such as $c_2 = e_3 + \bar{m}$, $c_2 = \bar{m}$, or $c_1 = a.e_1$ in order to extract $\bar{m}$ from c_2.

C. Fault Attacks on Decryption

The decryption phase of Ring-BinLWE can be the target of various fault attacks that recover the secret key. Some of the fault attacks on decryption can be mapped as passive and active chosen ciphertext attacks (CCA and CCA2) [18]–[20].

1) Zeroing Faults: During the decryption phase, the attacker can extract the secret key by attempting to set parts of the ciphertext c_1 in order to calculate certain parts of r_2 by differentiating between correct and faulty decryption results.

2) Skipping Faults: The adversary can skip certain cycles during the calculation of $c_1.r_2$ multiplication. This attack works very similar to the above-mentioned zeroing faults on decryption.

3) Randomization Faults: The only phase where randomization attacks are helpful to successfully extract secret data is decryption. The attacker sets a certain bit of r_2 to a random value and evaluates the difference between the faulty and the correct message $\bar{m}$ to guess the corresponding bit of r_2. Furthermore, the attacker repeats the same process for every other bit of r_2 (see differential fault attacks [28]).

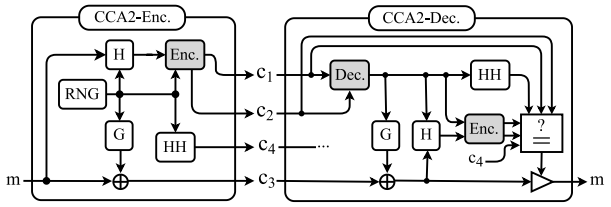

Fig. 2. CCA2-secure Ring-BinLWE scheme derived from [21].

D. Vulnerability of Previous Ring-BinLWE Implementations

We argue that most of the fault attacks are applicable to previous implementations of Ring-BinLWE presented in [10]. This is due to the fact that no fault attack countermeasures have been discussed in its implementations and none of the algorithms contain integrity checks. Moreover, Buchmann *et al.* [10] have implemented the original Ring-BinLWE scheme that is vulnerable against CCA and CCA2 attacks [18]–[20]. Therefore, all of the fault attacks are applicable to implementations presented in [10].

IV. Proposed Fault-Resilient Ring-BlinLWE

In this section, we provide detailed information regarding the implementation of a (first-order) fault-resilient Ring-BinLWE scheme on software. First, we propose a CCA2-Secure variant of Ring-BinLWE, which is required in order to resist some of the fault injection attacks during the decryption phase [18], [19]. In the remaining parts of the section, we analyze C-based implementation of all three phases of CCA2-Secure Ring-BinLWE along with primitive functions such as *shift_and_add* multiplication. In order to achieve better performance in comparison with straight forward implementations, we employ the same scheme optimization as [9]. We handle every coefficient directly in 2s complement notation, which eliminates all of the necessary reduction operations after modular arithmetic [9].

A. CCA2-Secure Scheme for Ring-BinLWE

Some of the fault attacks on the decryption phase can be mapped to CCA and CCA2 attacks [18], [19]. Therefore, it is necessary to have a CCA2-secure implementation to provide resistance against fault attacks on decryption. Here, we propose a CCA2-secure variant of the Ring-BinLWE scheme based on the same approach as proposed in [21] (for standard Ring-LWE). Fig. 2 presents the overall dataflow of the proposed CCA2-Secure Ring-BinLWE. The gray-colored modules show original encryption and decryption functions of Ring-BinLWE.

Three random oracles H:$\{0,1\}^n \rightarrow \{0,1\}^n$, G:$\{0,1\}^n \rightarrow \{0,1\}^n$, and HH:$\{0,1\}^{2n} \rightarrow \{0,1\}^n$ are employed in order to: 1) encrypt message m; 2) generate *pseudo*random vectors; and 3) provide adaptive security checks during the decryption phase. Fig. 2 present detailed information regarding CCA2-Secure encryption and decryption operations for Ring-BinLWE.

B. Primitive Functions

In this section, we describe the primitive functions that are used by scheme operations. Each primitive function is designed such that it is resistant against the proposed fault model. Furthermore, the entire system including all primitives is also designed to be fault resilient. This will be discussed in detail in the following sections. First, we provide implementation details of different functions such as random number generators (RNGs), and one-way oracles that are necessary to resist CCA2 and fault attacks. Later, polynomial arithmetic over the ring $\mathcal{R}_q$, such as addition and multiplication are presented.

1) Random Number Generator: To implement a proper RNG primitive, we employ the same method used in [29] and [30]. To this end, a random initial vector, which is generated by the device, is fed to an AES module to obtain a nonlinear pseudorandom number. We employ AES-256 in order to provide long-term security [29]. More details on the implementation of our employed PRNG can be found in Sections V-A and V-B for ATxmega128A1 and Cortex-M0, respectively.

2) Random Oracles: To provide resistance against CCA2, we have implemented three random oracles G, H, and HH as described in Section IV-A. These random oracles also exploit AES-256. The input to each of these random oracles is a 256-b number, which is served as the AES-256 key. The AES-256 input data (message) is a 128-b number, which is constant but different for each random oracle. Further implementation details on our random oracles can be found in Sections V-A and V-B for ATxmega128A1 and Cortex-M0, respectively.

3) Digest and Verify: To prevent or detect manipulating input/output arguments between function calls, two functions are implemented: 1) *digest:* it obtains a number and provides the unique digest of that number, which is apparently of the same size (implemented in the same manner as random oracles) and 2) *verify:* it compares a vector with a given digest and returns *zero* in case of match and a random number otherwise. These two functions are being called before and after each primitive function.

4) Addition: In Ring-BinLWE operations, at least one addition of an n-bit binary vector with a polynomial ($n \times 8$ b) is required during each phase (note that encryption requires multiple additions). The proposed fault-resilient addition function obtains five inputs as follows: a polynomial element vector $A \in \mathcal{R}_q$, a binary vector $r \in \{0,1\}^n$, two vectors containing the digest of each main input, and the digest of the final calculated result (*digest_res*). The complete code for implementing *poly_bin_add* function is shown in Listing 1.

According to the fault model proposed in Section III, the adversary has the power to change the result or skip any *if/else*-like conditional branch. Therefore, in all of the implementations of this article, we have not used any *if/else*-like statement. Lines 5–13 implement the main functionality of the procedure. Note that lines 7 and 8 perform 256 ($= 8 \times 32$) loop cycles in total. Line 9 performs bit selection of the binary vector r. Instead of evaluating the value of the variable *val* using an *if/else*-like operation, we perform the same round of operations in every loop cycle and add the variable *val* to its corresponding byte from A and store the results to *res* (line 10 from Listing 1).

We check the correctness of the input vectors and loop counters in lines 15–21 before returning the result of addition

```
1  byte * poly_bin_add (byte * A, byte * r, byte *
       digest_A, byte * digest_r, byte * digest_res) {

3    [... initializations ...]

5    /** main loop: Ring-BinLWE addition **/
6    loops = 0;
7    for(i=0; i<n/8; i++) {
8      for (j=0; j<8 ; j++) {
9        val = (r[i] >> j) & 1; // j th bit selection
10       res[loops] = val + A[loops];
11       loops++;
12     } // for
13   } // for

15   /** check in_r and loops to be as expected **/
16   err = gen_rand_bin();
17   chk_A = verify(digest_A, A);
18   chk_r = verify(digest_r, r);
19   A_r_ok = bin_nums_mult(chk_A, chk_r, err);
20   A_r_lps_ok = bin_num_mul_xnor(A_r_ok,loops-256,err);
21   final_res = bin_add(A_r_lps_ok, res);
22   digest_res = digest(final_res);

24   return final_res;
25 }
```

Listing 1. Addition primitive function.

in line 23. Note again that no *if/else*-like operations can be employed during these checks. The main threat to the addition function can be either skipping certain loops or zeroing some parts of (or the entire) the input variables A and/or r. The countermeasures are implemented as follows.

1) *Verifying Correctness of Input Arrays A and r:* This check is completed in line 19 as follows. *chk_A* is zero if the digest of A matches *digest_A*; otherwise, it is set to a random number. *chk_r* is evaluated in the same manner. *A_r_ok* is evaluated as follows:

$$(chk\_A + chk\_r) \times err,$$

where *err* is a 256-b random generated in line 16.

2) *Verifying the Number of Executed Loops:* Line 20 multiplies the variable $loops - 256$ (expected to be equal to zero) to every byte of *err* vector. Moreover, it performs an XNOR operation on the result with the *A_r_ok* vector. In case of no faults, the variable *A_r_lps_ok* is set to a 256-b zero vector. Otherwise, *loops* does not equal to 256 and hence *A_r_lps_ok* is set to a nonzero value. *A_r_lps_ok* is calculated as follows:

$$[(loops - 256) \times err] \odot A\_r\_ok.$$

3) *Masking the Result Before Returning From Function:* Finally, line 21 adds the generated error vector *A_r_lps_ok* to the calculated *res* before returning the result.

Every attempt in skipping any of the verification lines (lines 15–21) will result in errors such as *null pointer* or *segmentation fault*; because by skipping a line at least one local variable will not be allocated and initialized. Hence, in one of the following lines, this variable will surely be used and the error has occurred. Moreover, any fault injection attack presented in Table II will result in a randomly generated vector to be added to the final result. This method will ensure that the attacker cannot obtain any useful information from injecting faults to the primitive functions.

```
1  byte * shift_and_add (byte * A, byte * r, byte *
       digest_A, byte * digest_r, byte * digest_res) {

3    [... initialization ...]

5    /** main loop: Ring-BinLWE multiplication **/
6    loops = 0;
7    for(i=0; i<n/8; i++) { // i: 0→32
8      for (j=0; j<8 ; j++) { // j: 0→8
9        val = (r[i] >> j) & 1; // j-th bit selection
10       k=0;
11       while(k < loops) {
12         A_neg = A[n-loops+k] ^ 255;
13         A_neg = A_neg + 1;
14         res[k] = res[k] + A_neg * val;
15         k++;
16       } // while
17       while(k < n) {
18         res[k] = res[k] + A[k-loops] * val;
19         k++;
20       } // while
21       loops++;
22     } // for
23   } // for

25   [... check A, r, loops and stats ...]
26   // final checks are performed same as lines 15 to 21
         from Listing 1.

28   return final_res;
29 }
```

Listing 2. Fault resiliant multiplication function.

5) Multiplication: Fig. 1 presents the *shift-and-add* multiplication in the Ring-BinLWE scheme. The proposed fault-resilient multiplication function receives five inputs: a polynomial element vector $A \in \mathcal{R}_q$, a binary vector $r \in \{0, 1\}^n$, two vectors containing the digest of each of the main inputs, and the digest of the final calculated result (*digest_res*). Listing 2 presents the implementation details of the proposed *shift_and_add* function.

Lines 5–23 implement the main multiplication function over the ring $\mathcal{R}_q$. Lines 7 and 8 perform 256 ($= 8 \times 32$) loop cycles in total. Line 9 performs the same bit-selection method as used in the addition function. We note that during every main loop cycle, no *if/else*-like operation is employed to branch based on the variable *val*. Lines 14 and 18 of Listing 2 perform a multiplication by *val*, before adding the selected parts of the vector to the results. This way, addition operations are effective only if *val* is equal to 1. As mentioned earlier, the shifting polynomial, which is equivalent with multiplying the polynomial by x, over the ring $\mathcal{R}_q$ maps to an *anticircular* rotation; because the modulus is $x^n + 1$ (see Fig. 1). Lines 11–16 deal with the addition of the *anti-circular*ly rotated inputs and the result; while lines 17–20 implement the remaining normal addition operations.

Similar to the *poly_bin_add* function, we check inputs A and r before returning the result. In order to ensure that no skipping faults are injected during the *main loop*, we also consider the variable *loops* in our final checks in the function. Overall, the final checks are in the same manner as the *poly_bin_add* function.

C. Main Functions

1) Key Generation: Listing 3 presents the proposed implementation for key generation in Ring-BinLWE. First, the

```
1  byte * Key_Gen (byte * A) {

3    [... initialization ...]

5    [... setting random values to r1 and r2 ...]

7    /** main function: Ring-BinLWE Key_Gen **/
8    tmp = shift_and_add(A, r2,dig_A,dig_r2, dig_temp);
9    res = bin_poly_sub(r1,tmp,dig_tmp,dig_r1,dig_res);

11   fltr_res = verify(res, dig_res);
12   final_res = poly_num_xor(res, fltr_res);

14   return final_res;
15 }
```

Listing 3. Fault resiliant key generation function.

```
10  res[loops] = val + ((A[loops] ^ 0xFF) + 1);
```

Listing 4. Subtraction function: binary minus polynomial.

```
1  byte ** Encryption (byte * A, byte * pk, byte * m) {

3    [... initialization ...]

5    v = secure_RNG(dig_v);
6    enc_seed1 = H(v,m);
7    enc_seed2 = HH(enc_seed1);
8    enc_seed3 = HH(enc_seed2);
9    e1 = rblwe_prng(enc_seed1);
10   e2 = rblwe_prng(enc_seed2);
11   e3 = rblwe_prng(enc_seed3);

13   c3 = secure_G_bin_xor(v,dig_v,m);
14   c4 = HH(v);

16   /** normal Ring BinLWE encryption of v **/
17   tmp1 = shift_and_add(A, e1, dig_A, dig_v, dig_tmp1);
18   tmp2 = shift_and_add(pk,e1,dig_pk,dig_v,dig_tmp2);
19   c1 = poly_bin_add(e2,tmp1,dig_tmp1,dig_v,dig_c2);
20   tmp3= poly_bin_add(e3,tmp2,dig_tmp2,dig_v,dig_tmp3);
21   c2 = poly_bin_msg_add(tmp3,v,dig_tmp3,dig_v,dig_c2);
22 }
```

Listing 5. Fault resiliant encryption function.

procedure starts with setting random values to the n-bit binary vectors r_1 and r_2.

After generating binary vectors, the multiplication $a.r_2$ is performed using implemented *shift_and_add* function. The result of multiplication is stored in a temporary variable, namely, *tmp*. The *bin_poly_sub* function is implemented the same as the addition function *poly_bin_add* but with a difference that it performs a 2s complement subtraction instead of addition in the 10th line of Listing 1. Listing 4 presents the only different line of the subtraction function compared to the addition one.

Before returning the public key, we perform final checks (lines 11 and 12) in order to provide resistance against the zeroing attack on *res* variable. Line 11 verifies whether *res* matches its generated digest by the *bin_poly_sub* function. The result of the verification function in line 11, which is a byte, is XORed (line 12) with every byte of calculated *res* in order to mask it in a random manner in case of inconsistency. Note that the result of the verification function will be equal to 0×00 if no faults are injected. Furthermore, each line from 7 to 12 in Listing 3 depends on its previous lines. Therefore, injecting any skipping fault, including skipping multiplication or final subtraction, will result in *null pointer* or "segmentation fault" errors.

2) Encryption: In order to resist differential randomization fault attacks on the decryption phase, it is necessary to provide CCA2 resistance in Ring-LWE schemes [18]–[20]. To this end, we have implemented the CCA2-Secure Ring-BinLWE scheme provided in Fig. 2. Three random oracles (H, G, and HH) are used to provide both classic and quantum attack resistance in this scheme [21], [31].

Listing 5 shows the proposed encryption function in detail. After generation of the binary random vector v and its verifiable digest, we calculate three random seeds derived from the message m and the random vector v in lines 6–8. Then, three binary error vectors e_1, e_2, and e_3 are generated from seeds (lines 9–11). Two additional ciphertexts (c_3 and c_4) are produced in lines 14 and 15 according to Fig. 2. The *secure_G_bin_xor* function in line 14, verifies the consistency between the vector v and its digest, before returning $m \oplus G(v)$.

The standard Ring-BinLWE encryption of the binary vector v is implemented in lines 17–22. The same method of employing temporary variables is used to ensure resistance against skipping faults. Note that applying fault injection attacks to the calculated ciphertexts (c_1, c_2, c_3, and c_4) does not provide any useful information to an adversary. Such attacks are detected during the *decryption* phase due to resistance against CCA2. Therefore, no countermeasures are required after the calculation of ciphertexts.

3) Decryption: The decryption phase starts with obtaining the binary vector v from the ciphertexts c_1 and c_2 (lines 5–8). Furthermore, the message m can be calculated as $m = G(v) \oplus c_3$ (lines 10 and 11). In addition, the error vectors e_1, e_2, and e_3 are regenerated using the same method as in encryption. To this end, ciphertexts c_1, c_2, and c_4 can be recalculated in order to be verified against given ciphertexts to provide classic and quantum CCA2 resistance [21], [31].

The final step in CCA2-secure decryption is to assure the consistency between given and regenerated ciphertexts (without the usage of *if/else*-like checks) before returning the decoded message. Lines 17–19 calculate XNOR of ciphertexts, which will result in an array of 1s in case of no fault. Then, we perform bitwise NAND operation on three calculated vectors that results in a vector of zeros in a normal scenario (no fault). Line 21 adds the decrypted message m to the vector *final_eq* in order to mask the message in case of any fault attack.

We note that skipping any lines or setting any parts of check variables (such as *eq_c2* or *final_eq*) to zero or random values, will be detected in the next function call. This will result in the generation of random outputs from that function. Therefore, the adversary cannot obtain any useful information by employing first-order fault attacks.

D. Security Analysis

In order to formally prove the fault resilience of the proposed functions, we employ the same probability-based

```
1  byte * Decryption (byte * A, byte * pk, byte * sk,
       byte * digest_sk, byte * c1, byte * c2, byte * c3
       , byte * c4) {

3    [... initialization ...]

5    /** normal Ring BinLWE decryption to get v **/
6    tmp1 = shift_and_add(c1,sk,digest_sk);
7    v_bar = poly_poly_add(tmp1, c2);
8    v = decode(v_bar);

10   tmp2 = G(v);
11   m = bin_xor(c3,tmp2);

13   [... re-gen e1, e2, and e3 with H(v,m) ...]
14   [... normal Ring BinLWE encryption of v ...]

16   // check if (C4==C4' && C1==C1' && C2==C2')
17   eq_c4 = poly_xnor(c4, new_c4, dig_eqc4);
18   eq_c1 = poly_xnor(c1, new_c1, dig_eqc1);
19   eq_c2 = poly_xnor(c2, new_c2, dig_eqc2);
20   final_eq = secure_bin_nand(eq_c4, eq_c1, eq_c2,
         dig_eqc4, dig_eqc1, dig_eqc2, dig_final);
21   res = bin_add(final_eq, m, dig_final);

23   return res;
24 }
```

Listing 6. Fault resiliant decryption function.

approach as [26]. It is based on the success probability of passing the verifications in the presence of faults. We review the associated success probability of the fault attack during each function as follows.

1) Primitives (Add/Sub/Mul): Attacking inputs *A* or *r* (*zeroing* or *randomization*) will result in two scenarios to successfully pass the verifications. 1) *verify* function (Listing 1 line 18) fails to return nonzero value; this happens if and only if the underlying AES-256 is malfunctioning, which has the probability of 2^{-256}. In scenario 2) *bin_nums_mult* (Listing 1 line 19) fails and sets every bit of *A_r_ok* variable to zero, which has the probability of $2^{\text{len}(A\_r\_ok)} = 2^{-256}$. The probability is measured as follows: $Pr[r\_ok = 0 \cup A\_ok = 0 \cup A\_r\_ok = 0] = 3 \times 2^{-256}$.

Another scenario is that the attacker skips certain loop cycles during the main calculation process in primitive functions. Same as previous scenarios, the success probability of this attack is equal to 2^{-256}; this is where the *bin_nums_mul_xnor* function (Listing 1 line 20) fails to execute properly.

2) Key Generation: Besides the verification process in every primitive function, a successful fault attack on the final result requires the *verify* function (Listing 3 line 11) to fail with the probability of 2^{-256}.

3) Encryption: Other than skipping and zeroing attacks on calculation of $c1$ and $c2$ ciphertexts, zeroing or skipping faults can target during initialization of the vector v; this results in corrupted $c3$ and $c4$ ciphertexts to exact message m. This attack requires the *secure_G_bin_xor* function (Listing 5 line 13) to fail and has the success probability of 2^{-256}.

4) Decryption: Providing the decryption function with corrupted inputs or changing ciphers by any of three types of faults results in inconsistency between given and newly generated ciphertexts; this is triggered in lines 17–20 of Listing 6. Successfully skipping any of these checks has a probability of 2^{-256}. However, regeneration of the ciphertexts can also fail,

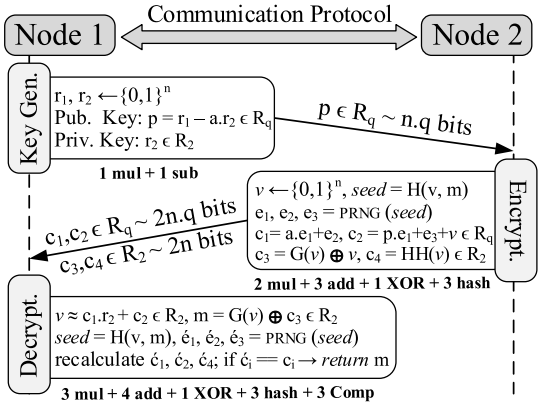

Fig. 3. Communication protocol of the proposed scheme along with scheme-level operations complexity.

TABLE III
TIME COMPLEXITY ANALYSIS OF THE PROPOSED IMPLEMENTATIONS

Phase	Ring-BinLWE [10]†	Proposed method†
Key G.	$T_{mul} + T_{sub}$	$T_{mul} + T_{sub} + 3 \times T_v + T_{xor}$
Enc.	$2 \times T_{mul} + 3 \times T_{add}$	$2 \times T_{mul} + 3 \times T_{add} + 5 \times T_v + T_{xor} + 3 \times T_H$
Dec.	$T_{mul} + T_{add}$	$T_{mul} + T_{add} + 3 \times T_v + T_{encryption}^{\bullet} + 3 \times T_{xnor} + T_{nand} + T_{add}$

† T_{mul}, T_{sub}, T_{add}, T_{xor}, T_{xnor}, T_{nand}, T_H and T_v stand for timing complexity of multiplication, subtraction, addition, XOR, XNOR, NAND, *hash* and *verify* operations, respectively.

• $T_{encryption}$ is equal to timing complexity of the entire encryption phase of the proposed method.

which has the probability of 2^{-84} due to the basic security level of 84 b from the Ring-BinLWE problem.

To summarize, the analysis shows that in order to successfully attack the proposed implementations, the adversary has the success probability of 2^{-84}, which is equal to the nominal security level of the Ring-BinLWE scheme.

E. Complexity Analysis

The complexity analysis and comparison of the original Ring-BinLWE against other Ring-LWE and classic PKE schemes such as ECC are detailed in [10]. Therefore, in this article, we only focus on the complexity analysis of the proposed method against the one presented in [10]. The proposed fault-resilient scheme has obviously higher complexity in comparison with the original scheme from [10]. The overhead is mainly due to the verifications in primitive functions and additional scheme-level operations such as extra calculations related to c_3 and c_4 ciphertexts. The only exposed overhead during the *key generation* phase is caused by the *verify* function within and after primitive function calls. Moreover, in the proposed scheme, a complete execution of *encryption* is done during the *decryption* phase as shown in Figs. 2 and 3, which increases the *decryption* overhead significantly. Table III presents detailed timing complexity analysis of the proposed method in comparison with the original implementations from [10].

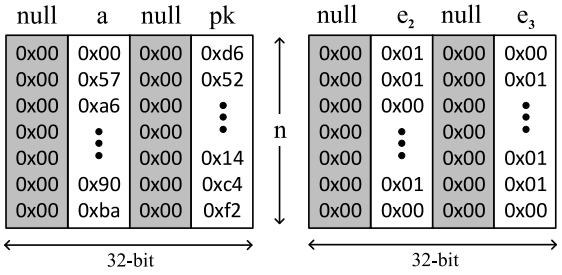

Fig. 4. Optimization for 32-b architectures.

```
11  A_neg = A[n loops+k] ^ 3855  // equal to 0x0F0F
12  A_neg = A_neg + 257; // equal to 0x0101
```

Listing 7. Changelog of 32-b implementation compared to Listing 2.

```
17  /** normal Ring BinLWE encryption of v **/
18  tmp1 = shift_and_add_32bit(A_P, e1, dig_A, dig_P,
        dig_e1, dig_tmp1);
19  c1 = poly_bin_add_sel_high(e2, tmp1, dig_tmp1, dig_e2,
        dig_c1);
20  tmp3 = poly_bin_add_sel_low(e3, tmp1, dig_tmp1, dig_e3
        , dig_tmp3);
21  c2 = poly_bin_message_add(tmp3, v, dig_tmp3, dig_v,
        dig_c2);
```

Listing 8. Changelog of 32-b implementation compared to Listing 5.

V. Implementation Details

In this section, we provide device-specific optimization of the proposed method for two widely used and resource-constrained microcontrollers: 1) 8-b AVR ATxmega128A1 and 2) 32-b ARM Cortex-M0. Moreover, our implementation results are compared against the original Ring-BinLWE.

A. AVR ATxmega128A1

ATxmega128A1 is an 8-b resource-constrained AVR microcontroller that is widely used for lightweight devices in IoT. Due to the fact that Ring-BinLWE scheme operations are modulo 256, they match the 8-b datapath of the ATxmega128A1. Using the same scheme optimization as proposed in [9] and [10], all of the operations match exactly the 2s complement notation of ATxmega128A1's ALU. No computation results exceed 8 b and hence, no overhead will be imposed.

To obtain random binary vectors, the PRNG presented in [30] has been exploited that utilizes an AES. Although the built-in AES accelerator of ATxmega128A1 is efficient, it is limited to a maximum key length of 128 b, which is not proper for long-term security [29]. Therefore, we employ the software AES implementation from [32] that is optimized for AVR microcontrollers. The assembly-based implementation of AES-256 from [32] requires 3521 clock cycles to encrypt a 256-b block.

As discussed in Section IV-A, three random oracles are necessary for CCA2 resistance. They are implemented based on AES-256 from [32] with different static block vectors, while the 256-b inputs are used as the encryption key. The encrypted outputs are used as the returned value of random oracles.

B. ARM Cortex-M0

In order to optimize the implementation of the 32-b architecture of Cortex-M0, Buchman *et al.* [10] proposed a vectorizing method in that reduces the time of each multiplication by 50% in Ring-BinLWE. However, this technique requires compression and decompression of the input vectors before and after every multiplication function call, which results in more vulnerabilities to fault injection attacks. Therefore, we propose a more simplified optimization method, which can only be applied to the encryption phase. Note that the encryption in Ring-BinLWE requires two multiplications (i.e., $pk.e_1$ and $a.e_1$) that have the same multiplicand e_1. Using the *shift_and_add* method for multiplication, we can store both corresponding coefficients of pk and a vectors in the same memory word as shown in Fig. 4. After 256 cycles, the maximum number that can be reached for each vector coefficient is $65\,280 = 255 \times 256$ (i.e., $0xFF00$), which requires only 16 b of memory. Therefore, during the entire multiplication process, the first 16-b part of the word does not produce any carry to be managed in the second 16-b part. Note that the gray columns of the memory shown in Fig. 4 are accumulating the overflows of the 2s complement addition and are not being considered in the final results. Using this technique, two multiplications can be calculated in parallel without any storage or timing overhead. Moreover, e_2 and e_3 are added to the result of the multiplication using the same method as shown in the right side of Fig. 4.

This technique improves normal encryption time by almost 49% but does not affect the multiplications in the key generation and decryption phases. It is worth mentioning that in the proposed CCA2-secure Ring-BinLWE scheme, we require a complete re-encryption of the decrypted message in order to ensure the validity of ciphertexts. Therefore, improving encryption time also affects decryption time.

To implement this technique, lines 18 and 19 from Listing 5 are merged into one function call *shfit_and_add_32bit*. This function call is very similar to Listing 2 with the difference that lines 11 and 12 change according to Listing 7. Moreover, Listing 8 presents the altered lines of code compared to the normal encryption (i.e., Listing 5).

C. Experimental Result

To the best of our knowledge, we are the first to propose fault-resilient Ring-BinLWE implementations on lightweight and resource-constrained 8- and 32-b microcontrollers. Table IV provides detailed information regarding our fault-resilient implementations compared to the original Ring-BinLWE from [10].

One of the advantages of the proposed implementations is to provide resistance against timing attacks due to the constant execution time for every round of multiplication regardless of the secret key value. On the other hand, proposed implementations contain various verification methods to detect fault attacks that impose more overhead compared to [10]. The proposed scheme imposes up to 12.5% more overhead due to the increasing ciphertext size by, including c_3 and c_4.

TABLE IV
IMPLEMENTATION RESULTS COMPARED TO PREVIEWS WORK

Work	Resistance			Comm./Stor. Cost (bits)			Device		Clock Cycles (×1000)			Time (ms)	
	CCA2	Fault	Timing	S. Key	P. Key	Cipher	Name	Arch.	Key Gen.	Enc.	Dec.	Enc.	Dec.
Buch'16. [10] (n=256, q=256)	×	×	×	256 b	2 Kb	4 Kb	ATxmega	8-bit	-	1,507	700	44.9	20.9
							Cortex-M0	32-bit	-	944	403	28.1	12.0
This work* (n=256, q=256)	✓	✓	✓	256 b	2 Kb	4.5 Kb	ATxmega	8-bit	1,335	2,691	4,037	80.2	120.3
							Cortex-M0	32-bit	1,481	1,755	3,404	52.3	101.4

Moreover, the proposed implementations are 50% and 78% slower compared to previous work [10] in *encryption* and *decryption* operations, respectively. This is caused by the timing complexity overhead discussed in Section IV-E.

To conclude, the proposed fault-resilient encryption and decryption operations require 80 and 120 ms to be executed on AVR ATxmega128A1, respectively. Moreover, performing the same operations takes 52 and 101 ms on ARM Cortex-M0, respectively. These results indicate that the proposed implementations are still practical for daily operations of IoT resource-constrained devices while providing fault resilience along with timing attack resistance in such nodes.

VI. CONCLUSION

In this article, we proposed (to the best of our knowledge) the first fault-resilient Ring-BinLWE software implementation, which is practical on lightweight and resource-constrained IoT nodes. Moreover, we formally prove the fault resilience of our implementations that require only 80 and 120 ms for encryption and decryption, respectively, on 8-b AVR microcontrollers. Although we exploited the Ring-BinLWE scheme, the proposed countermeasures can also be applied to other Ring-LWE schemes such as homomorphic ones. However, managing the growing overhead on such schemes will be more challenging due to larger key/ciphertext sizes.

REFERENCES

[1] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu, "Security vulnerabilities of Internet of Things: A case study of the smart plug system," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1899–1909, Dec. 2017.

[2] N. Zhang, R. Wu, S. Yuan, C. Yuan, and D. Chen, "RAV: Relay aided vectorized secure transmission in physical layer security for Internet of Things under active attacks," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8496–8506, Oct. 2019.

[3] Q. Li and L. Yang, "Beamforming for cooperative secure transmission in cognitive two-way relay networks," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 130–143, May 2019, doi: 10.1109/TIFS.2019.2918431.

[4] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[5] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, vol. 48, no. 177, pp. 203–209, 1987.

[6] R. Chaudhary, G. S. Aujla, N. Kumar, and S. Zeadally, "Lattice based public key cryptosystem for Internet of Things environment: Challenges and solutions," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4897–4909, Jun. 2019.

[7] C. Patrick and P. Schaumont, "The role of energy in the lightweight cryptographic profile," in *Proc. NIST Lightweight Cryptography Workshop*, 2016, pp. 1–16.

[8] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. IEEE 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.

[9] S. Ebrahimi, S. Bayat-Sarmadi, and H. Mosanaei-Boorani, "Post-quantum cryptoprocessors optimized for edge and resource-constrained devices in IoT," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 5500–5507, Jun. 2019.

[10] J. Buchmann, F. Göpfert, T. Güneysu, T. Oder, and T. Pöppelmann, "High-performance and lightweight lattice-based public-key encryption," in *Proc. ACM 2nd Int. Workshop IoT Privacy Trust Security*, 2016, pp. 2–9.

[11] L. Chen *et al.*, *Post-Quantum Cryptography*, U.S. Dept. Commerce, Nat. Inst. Stand. Technol., Gaithersburg, MA, USA, 2016.

[12] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2010, pp. 1–23.

[13] *National Institute of Standards and Technology*. Accessed: Sep. 10, 2019. [Online]. Available: https://www.nist.gov/

[14] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, p. 34, 2009.

[15] C. Peikert, "Public-key cryptosystems from the worst-case shortest vector problem," in *Proc. ACM 41st Annu. Symp. Theory Comput.*, 2009, pp. 333–342.

[16] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-LWE cryptoprocessor," in *Proc. Workshop Cryptograph. Hardw. Embedded Syst.*, 2014, pp. 371–391.

[17] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[18] N. Bindel, J. Buchmann, and J. Krämer, "Lattice-based signature schemes and their sensitivity to fault attacks," in *Proc. IEEE Workshop Fault Diagn. Tolerance Cryptography (FDTC)*, 2016, pp. 63–77.

[19] F. Valencia, T. Oder, T. Güneysu, and F. Regazzoni, "Exploring the vulnerability of R-LWE encryption to fault attacks," in *Proc. ACM 5th Workshop Cryptography Security Comput. Syst.*, 2018, pp. 7–12.

[20] L. G. Bruinderink and P. Pessl, "Differential fault attacks on deterministic lattice signatures," in *Proc. IACR Trans. Cryptograph. Hardw. Embedded Syst.*, 2018, pp. 21–43.

[21] T. Oder, T. Schneider, T. Pöppelmann, and T. Güneysu, "Practical CCA2-secure and masked ring-LWE implementation," in *Proc. IACR Trans. Cryptograph. Hardw. Embedded Syst.*, 2018, pp. 142–174.

[22] *Adafruit Company*. Accessed: Feb. 10, 2020. [Online]. Available: https://www.adafruit.com/

[23] *Arduino Computing Platforms*. Accessed: Feb. 10, 2020. [Online]. Available: https://www.arduino.cc/

[24] *Texas Instruments Semiconductor Company*. Accessed: Feb. 10, 2020. [Online]. Available: http://m.ti.com/

[25] F. Göpfert, C. van Vredendaal, and T. Wunderer, "A hybrid lattice basis reduction and quantum search attack on LWE," in *Proc. Int. Workshop Post Quant. Cryptography*, 2017, pp. 184–202.

[26] D. Vigilant, "RSA with CRT: A new cost-effective solution to thwart fault attacks," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2008, pp. 130–145.

[27] P. Rauzy and S. Guilley, "A formal proof of countermeasures against fault injection attacks on CRT-RSA," *J. Cryptograph. Eng.*, vol. 4, no. 3, pp. 173–185, 2014.

[28] T. Schneider, A. Moradi, and T. Güneysu, "ParTI: Towards combined hardware countermeasures against side-channel and fault-injection attacks," in *Proc. Annu. Cryptol. Conf.*, 2016, pp. 302–332.

[29] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede, "Efficient ring-LWE encryption on 8-bit AVR processors," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, 2015, pp. 663–682.

[30] T. Prescott, *Random Number Generation Using AES*. Accessed: Sep. 10, 2019. [Online]. Available: http://www.atmel.com/Images/article_random_number.pdf

[31] E. E. Targhi and D. Unruh, "Post-quantum security of the Fujisaki–Okamoto and OAEP transforms," in *Proc. Theory Cryptography Conf.*, 2016, pp. 192–216.

[32] D. Labor, *Crypto-AVR-LIB*. Accessed: Sep. 10, 2019. [Online]. Available: https://git.cryptolib.org/avr-crypto-lib.git