CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Efficient error detection for NTT using algebraic invariants (AIC) checking for PQC and FHE on FPGA
Authors	Paul, Rourab;Baidya, Paresh;Mandal, Swagata;Guha, Krishnendu
Publication date	2026-07-08
Original Citation	Paul, R, Baidya, P, Mandal, S & Guha, K 2026, 'Efficient error detection for NTT using algebraic invariants (AIC) checking for PQC and FHE on FPGA', ACM Transactions on Embedded Computing Systems, vol. 25, no. 4, 70, pp. 1-20. https://doi.org/10.1145/3822512
Type of publication	Article (peer-reviewed)
Link to publisher's version	https://www.scopus.com/pages/publications/105045955393 - 10.1145/3822512
Rights	© 2026, Copyright held by the owner/author(s). This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. - https://creativecommons.org/licenses/by-nc-nd/4.0/
Download date	2026-11-07 01:36:16
Item downloaded from	https://hdl.handle.net/10468/19132

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Efficient Error Detection for NTT Using Algebraic Invariants (AIC) Checking for PQC and FHE on FPGA

ROURAB PAUL, Computer Science & Engineering, Shiv Nadar University Chennai, Chengalpattu, India
PARESH BAIDYA, Computer Science and Emgineering, Siksha O Anusandhan University, Bhubaneswar, India
SWAGATA MANDAL, Electronics and Communication, Jalpaiguri Government Engineering College, Jalpaiguri, India
KRISHNENDU GUHA, School of Computer Science and Information Technology, University College Cork, Cork, Ireland

Polynomial multiplication is the most computationally demanding arithmetic operation used in many Post-Quantum Cryptographic (PQC) and Fully Homomorphic Encryption (FHE) algorithms. The Number Theoretic Transform (NTT) is the most efficient technique for performing polynomial multiplication in these schemes. However, at the implementation level, NTT designs used in PQC and FHE are vulnerable to information leakage due to intentional fault injection attacks. Preventing both intentional and unintentional faults has become a major concern for next-generation secure processors. In this regard, we introduce Full and Partial Recomputation–based Algebraic Invariant Checking (FR-AIC and PR-AIC) schemes to robustly safeguard the arithmetic operations of the NTT processing element (PE). The FR-AIC achieves a high fault detection rate, and the lightweight PR-AIC offers reduced hardware overhead at the cost of a slightly lower detection capability. The proposed architecture is scalable across different NTT variants, supporting arbitrary polynomial sizes (n), the number of polynomial coefficients) and data widths ($\log q$), thereby making it suitable for a wide range of PQC and FHE applications. Furthermore, the design is fully compatible with multi-PE parallel architectures, enabling efficient acceleration to meet high-throughput NTT performance requirements. Extensive simulations and fault-injection emulation of multiple NTT variants implemented on an Artix-7 FPGA for PQC and FHE applications show that the proposed fault detection mechanism efficiently detects nearly 100% of faults without introducing any latency overhead. The area and energy overheads of our schemes are tolerable for practical implementations of NTT.

CCS Concepts: • **Computer systems organization → Reliability**; • **Security and privacy → Hardware security implementation**; **Cryptography**; • **Hardware → Reconfigurable logic and FPGAs;**

Additional Key Words and Phrases: Fault detection, NTT, FPGA, PQC, FHE

This research is supported by Taighde Éireann – Research Ireland under Grant number 13/RC/2077_P2 at CONNECT: the Research Ireland Center for Future Networks.

Authors' Contact Information: Rourab Paul, Computer Science & Engineering, Shiv Nadar University Chennai, Chengalpattu, Tamil Nadu, India; e-mail: rourabpaul@gmail.com; Paresh Baidya, Computer Science and Emgineering, Siksha O Anusandhan University, Bhubaneswar, OD, India; e-mail: pareshbaidya@soa.ac.in; Swagata Mandal, Electronics and Communication, Jalpaiguri Government Engineering College, Jalpaiguri, WB, India; e-mail: swaga89@gmail.com; Krishnendu Guha (corresponding author), School of Computer Science and Information Technology, University College Cork, Cork, Ireland; e-mail: kguha@ucc.ie.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

© 2026 Copyright held by the owner/author(s).
ACM 1539-9087/2026/07-ART70
https://doi.org/10.1145/3822512

ACM Reference Format:
Rourab Paul, Paresh Baidya, Swagata Mandal, and Krishnendu Guha. 2026. Efficient Error Detection for NTT Using Algebraic Invariants (AIC) Checking for PQC and FHE on FPGA. *ACM Trans. Embedd. Comput. Syst.* 25, 4, Article 70 (July 2026), 20 pages. https://doi.org/10.1145/3822512

1 Introduction

The **National Institute of Standards and Technology** (**NIST**) standardized ML-KEM (based on CRYSTALS-Kyber) as a key encapsulation mechanism in FIPS 203 [24] and ML-DSA (based on CRYSTALS-Dilithium) as a digital signature algorithm in FIPS 204 [25] in August 2023. These standards were subsequently formally issued after August 2024 as **Federal Information Processing Standards** (**FIPS**): FIPS 203 [24] for module-lattice-based key encapsulation mechanisms, FIPS 204 [25] for module-lattice-based digital signatures, and FIPS 205 [1] for stateless hash-based digital signatures. Recently, Racoon [11] is proposed side-channel-resistant digital signature scheme aimed at improving resilience against implementation-level attacks. However, it is relatively less mature and less extensively studied as compared with standardized schemes like Dilithium. On the other hand, NIST is actively evaluating FHE schemes such as **Brakerski/Fan–Vercauteren** ((**BFV**) [13]) and **Brakerski–Gentry–Vaikuntanathan** ((**BGV**) [8]) as part of its future homomorphic cryptographic frameworks. All these PQC schemes and FHEs (BGV [8], BFV [13], etc.) require efficient large-scale polynomial multiplications. NTT is also adopted in signature schemes and identification schemes. NTT is the most resource and timing-critical component, significantly reducing the computational complexity of polynomial multiplication in various PQC and FHE schemes from $\mathcal{O}(n^2)$ to $\mathcal{O}(N\log N)$. Therefore, over the last decade, a significant amount of research [7, 15, 16, 19–21] effort has been devoted to the design of hardware accelerators for the NTT. Florian et al. [19] implement an open-source Python framework for generating RTL designs of different NTT variants, including normal-input/reverse-output and reverse-input/normal-output configurations. The framework supports various NTT types used in PQC and FHE algorithms. Javeed et al. implemented two NTT architectures in [16] and [15]. In [16], the authors propose an area–time efficient, fully lookup-table-based reduction scheme for polynomial multiplication in the NTT used in Kyber. In Ref. [15], a unified butterfly unit is introduced that integrates modular multiplication, addition, and subtraction operations. This design employs a laddering approach based on interleaved multiplication. However, the PQC and FHE algorithms are mathematically secure but they are vulnerable at the implementation level to **side-channel attacks** (**SCAs**). In SCA, sensitive information may leak through physical observables such as execution time, power consumption, or electromagnetic emissions. By analyzing these side-channel signals, attackers can correlate the observed patterns with secret-dependent operations and potentially guess secret keys. The threat becomes more severe in the presence of malicious modifications, such as hardware Trojans inserted into the target circuit. Additionally, attackers may induce faults using techniques such as voltage or clock glitching to flip or perturb specific bits during computation. By observing the resulting changes in timing behavior and power consumption, attackers can further refine their analysis and increase the likelihood of successful secret recovery. Therefore, ensuring the protection of critical components in PQC and FHE systems is essential. All NTT implementations [7, 15, 16, 19–21] are vulnerable to side-channel attacks, unintentional faults caused by device aging, thermal effects, and other environmental factors, as well as intentional fault attacks launched by adversaries.

1.1 Threat Model

Espitau et al. [12] demonstrated a loop-abort fault attack on several lattice-based cryptosystems, including BLISS, NewHope, Frodo, GPV signatures, and CRYSTALS-Kyber. In this attack, a fault was

injected to prematurely terminate the loop responsible for sampling Gaussian secret coefficients. During normal execution, the sampler outputs a full polynomial $e = [e_0, e_1, e_2, \dots, e_{n-1}]$. However, Espitau et al. [12] modify the value of the loop counter register using a clock glitch injection technique, causing the loop to terminate prematurely. Therefore, the resulting polynomial becomes $e = [e_0, e_1, \dots, e_{m-1}, 0, 0, \dots]$. This produces a low-degree or low-rank noise vector, breaking the intended masking and leaking linear equations that reveal information about the secret. Pessl et al. [26] performed fault-injection attacks via instruction skipping or glitching against the decryption routines of Kyber to recover secret key information. The authors here wanted to correlate the decryption noise and the secret by injecting faults. They disturb the noise term in such a way that the relation becomes exploitable.

Ravi et al. [28] identified a vulnerability in the PQC on ARM Cortex-M4 (PQM4) library [17] that enables a practical fault attack on the NTT implementation. They demonstrated successful attacks on both Kyber and Dilithium by modifying the program counter address using electromagnetic fault injection, thereby redirecting the access to the twiddle factor array. As a result, the ARM Cortex-M4 is forced to load incorrect twiddle factors, which can effectively become zero. Since secret vectors and error polynomials in PQC and FHE schemes (e.g., SEAL, OpenFHE) are processed through the NTT, zeroizing the twiddle factors drastically reduces the randomness of these values and significantly lowers their entropy. Similar to the attack strategy of Espitau et al. [12], Ravi et al. [28] applied the same approach to the secret vector s and error polynomial e in Kyber, which are processed through the NTT as the input α. As shown in Line 9 of Algorithm 1, the secret vector $s = (s_0, s_1, \dots, s_{n-1})$ and the error polynomial $e = (e_0, e_1, \dots, e_{n-1})$ enter the NTT as the input α and are multiplied with the twiddle factor ω by a multiplier. If an attacker injects a fault into the multiplier inside the PE and forces ω to become zero for all NTT stages, the resulting transformed polynomials become $\bar{s} = (s_0, 0, \dots, 0)$ and $\bar{e} = (e_0, 0, \dots, 0)$. This collapses the entropy of s and e to only their first coefficients, s_0 and e_0, respectively. Similarly, key generation in the BGV FHE [8] scheme, the public key is $pk = (a, b = -a \cdot s + e)$, where a is a public, uniformly random polynomial. If an attacker is able to force the twiddle factor $\omega = 0$, the product $a \cdot s$ becomes partially or fully zero. Consequently, the public key component b degenerates to $b \approx e$, causing the secret key contribution to disappear. As a result, the public key loses its intended randomness and leaks structural information, enabling an attacker to detect the faulty key and potentially mount distinguishing or secret key recovery attacks. In BGV encryption, the ciphertext components are given by $c_0 = b \cdot r + e_1 + m$ and $c_1 = a \cdot r + e_2$, where a and b are public, uniformly random polynomials and r is a secret ephemeral polynomial. Faults in the NTT twiddle factors can cause the polynomial multiplications $a \cdot r$ and $b \cdot r$ to partially collapse, leading to abnormal noise behavior. This deviation introduces statistical bias in the ciphertext, which can be exploited by an attacker. It is to be noted that PE is the basic block of NTT which executes Line 9, Line 11 and Line 12 of Algorithm 1.

Performing the aforementioned attacks on NTT implementations becomes considerably easier in the context of **third-party intellectual property** (**3PIP**) cores. Adversaries within third-party design houses may intentionally insert hardware Trojans into the FPGA fabric during design or fabrication stages, or manipulate the bitstream at the post-design or deployment stage. Hardware Trojans can be inserted into the system through compromised CAD tools [33]. A malicious CAD tool could bypass testing and post-silicon validation [5]. Trojans can also be introduced during deployment phase by tampering with the bitstream [32]. The observational attacks like **Soft Analytical SCAs** (**SASCA**) [27, 29] are potentially powerful attacks if the security processor are unprotected.

1.2 Related Works

The existing literature on fault detection in NTT implementations can be broadly categorized into two groups: (i) Embedded-processor-based NTT implementations [6, 28], and (ii) Hardware-

accelerator-based NTT implementations [2, 3, 9, 14, 18, 30, 31]. The fundamental building block of the NTT is the PE inside the butterfly unit. The PE primarily consists of a multiplier, an adder, and a subtractor. These three components are critical because an attacker who targets any of them can leak information about the input polynomial α. As stated in the NTT algorithm (Algorithm 1), the input polynomial is denoted by α, and the output of the NTT is $\overline{\alpha}$. For example, if an attacker faults the multiplier (Line 9 of Algorithm 1) and forces the twiddle factor ω to become zero, they can reduce the entropy and directly recover $\alpha[k]$ from the outputs $\alpha[k] + \alpha[k+m] \cdot \omega$ (Line 11 of Algorithm 1) from the adder and $\alpha[k] - \alpha[k+m] \cdot \omega$ (Line 12 of Algorithm 1) from subtractor of the PE. Similarly, manipulating ω inside the adder or subtractor can reduce the entropy of α, making the secret polynomial more guessable for Kyber. Such vulnerabilities are unacceptable in schemes like Kyber and in FHE schemes such as BGV and BFV. In Kyber, the NTT processes secret key and error polynomials that must remain unpredictable and secure. In FHE schemes, the inputs to the NTT include secret and error polynomials used in key generation and ciphertext operations, and revealing or corrupting them can compromise the security assumptions of the underlying RLWE problem. Table 1 reports a comparison of existing fault detection solutions for NTT.

1.2.1 Ahmadi et al. [3]. Ahmadi et al. [3] implemented a correlated checksum on the polynomial coefficient of NTT on Artix-7 FPGA. During the fault injection method, Ahmadi et al. [3] targeted only the multiplier block that multiplies the twiddle factor ω with the polynomial coefficient $\alpha[k]$ in Kyber's NTT. In support of focusing solely on the multiplier for fault detection, they argued that an attacker could inject a fault that forces ω to become zero. In such a scenario, the input polynomial $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ would collapse to $\overline{\alpha} = (\alpha_0, 0, \ldots, 0)$, thereby reducing the entropy of α down to a single coefficient α_0. However, their analysis considers only the multiplier as a fault location. In practice, the adder and subtractor blocks within the PE of butterfly unit can also be faulted to remove the influence of ω in the NTT operations. Such manipulations similarly reduce the entropy of the input polynomial, making it more easily guessable from the resulting NTT outputs.

If $n-1$ is the degree of polynomial, the additional operations for fault detection in NTT by Ahmadi et al. [3] costs $\frac{13n}{2}$ Mults, $\frac{5n}{2} + n - 1$ Additions, n Modulo, $\frac{n}{2}$ Exponentiation, $\frac{3n}{2}$ Modular inverse. However, Ahmadi et al. [3] reuse expensive hardware blocks, such as multipliers, to reduce area overhead. This reuse, however, results in a 41.3% timing overhead, which consequently increases the energy overhead by 51.5%. Ahmadi et al. [3] detect faults only after the NTT operation is completed. Since fault detection is performed only at the end of the NTT computation, it cannot identify which NTT operation was corrupted or when the fault occurred, resulting in limited diagnosability. Moreover, late detection leads to wasted computation and energy, which is not preferable for high-performance systems, especially for large NTT sizes. Furthermore, as the proposed methodology is based on a checksum computed over the processed polynomial coefficients, identical increments in one coefficient and decrements in another may cancel each other out and thus remain undetected.

1.2.2 Ravi et al. [28]. Ravi et al. [28] proposed a lightweight Shannon entropy checking mechanism to detect faults at the NTT output coefficients with reduced-bit representations. They compared it against a fault-free reference entropy. Since correct NTT outputs are uniformly distributed modulo q, they exhibit high entropy. Faults introduce bias and correlation in the output distribution of NTT, resulting in a measurable reduction in entropy. An error is detected when the entropy deviation exceeds a predefined threshold. The entropy calculation used by Ravi et al. [28] requires $n-1$ floating-point additions $n-1$ floating-point multiplications, frequency counting, and a division, which are expensive operations in software and hardware. Ravi et al. [28] implement this in Arm Cortex-M4, however, they did not report area, time and energy overhead. On the other hand, here, the entropy calculation is performed only after the entire NTT operation is completed. Therefore, similar to Ahmadi et al. [3], this approach allows compromised or low-entropy secret polynomials

Table 1. Overhead of the Proposed Fault Detection Module Compared with Existing Literature for NTT

Work	Fault Detection Strategy	Overhead (%)			Remarks	Overhead operations
		Area	Delay	Energy		
Ahmadi et al. [3]	**Correlated Checksum on Polynomial Coefficients**, able to detect fault on mult, adder and sub of PE	25.1	41.3	51.5	**Fault injection carried out only in the multiplier**, achieving 53–99.9% detection for 1–16 & above faults occurrences. Faults are detected **only after all coefficients are computed** (512 cycles with two PEs), **No fault localization, less diagnosability, waste of computation power**, **identical increment and decrement** on coefficients **cannot be detected** by proposed checksum	$\frac{13n}{2}$ Mults, $\frac{5n}{2}$ + $n-1$ Additions, n Modulo, $\frac{n}{2}$ Exponentiation, $\frac{3n}{2}$ Modular inverse, $\frac{n}{2}$ integer Reverse
Ausmita et al. [31]	**Recomputation with Negate Operand (RENO)**	18.3	11.4	10.67	**Faults can be detected only on the multiplier** of the PE, achieving 99.9% detection for up to 6-bit faults in the multiplier.	$\frac{n\log n}{2}$ 2's complement, $\frac{n\log n}{2}$ Mults
Ausmita et al. [30]	on multiplier of PE, not able to detect faults on adder and sub	20.2/ 15.3/ 21.5	8.46/ 15.88/ 13.71	15.6/ 7.6/ 11.2	Faults in the multiplier are detected instantly in each clock cycle, whereas **faults in the adder or sub remain undetectable**.	
Ravi et al. [28]	**Checking Entropy** at NTT output using **Shannon Entropy** Method	-	NR	NR	NTT is **software-based on ARM Cortex-M4**; faults are detected by checking NTT-output distribution. **Fault coverage is not reported**. **Entropy checks are simple in software** but **expensive in hardware**. Processor-based NTT is slower than hardware NTT, **No fault localization, less diagnosability, waste of computation power,**	$n-1$ floating point addition $n-1$ floating point mult, a Frequency counting, a division
Bauer et al. [6]	Polynomial **Evaluation and Interpolation**	-	72	NR	Cortex-M4–based NTT based **Polynomial evaluation and interpolation are expensive in hardware** in terms of area, latency, and energy. Although they offer high fault-detection capability, the reported **72% latency overhead is unsuitable** for **high-speed practical applications**, **No fault localization, less diagnosability, waste of computation power,**.	2n–1 mult and 2n–2 additions.
Jati et al. [14]	**Randomized Memory Addr** to access polynomial coefficients α and twiddles from memory.	NR	NR	NR	Memory addresses for polynomial coefficients α and twiddles ω are randomized instead of sequential, breaking the correlation between power consumption and NTT iterations and significantly improving SCA resistance; only address-control signals are protected; **faults in data paths and logic are not covered**.	$n \log n$ random number generator
Khan et al. [18]	Area Optimize and Run-Time optimize protection on NTT Memory using **Hamming and parity** Code α and twiddles from memory.	29.1/ 44.8	1.6/ 1.44	NR	Fault-tolerant RAM and ROM modules were implemented using Hamming codes and parity bits to protect polynomial coefficients and twiddle factors in the NTT, **Fault during arithmetic ops. not detectable**. **Area Overhead is quite high**	$2 \cdot 5 \cdot \log q + \frac{n\log n}{2} \cdot (\log q - 1)$ XOR operations, a decoder logic
Canto et al. [10]	**Recomputation with Shifted Operand (RESO)** for FrodoKEM, NTRU & Saber	28.4/ 39.6/ 36.6	32.7/ 16.7/ 28.3	0.4/ 3.2/ 1.2	**Faults can be detected only on the multiplier** & and a adder of the MAC, achieving 99.9% detection, **latency Overhead is quite high**	$\frac{n\log_2 n}{2}$ mult
Aghapour et al. [2]	**Modular offset based Recomputation** & word wise summation for **BMM** & **A counter** to count no. of loop executed in the *while* loop of **BMR**.	17.1 /19.4	16.16/ 2.02	21.87/ 31.73	Fault detection is **limited to the multiplier**; despite ~29% area and ~22% energy overhead, faults in the **adder and subtractor remain undetectable.** Probabilistic behaviour based BMR requires many samples, word summation-based BMM, making **No fault localization, less diagnosability, waste of computation power,**.	BMR: $\frac{n\log_2 n}{2}$ additions, $\frac{n\log_2 n}{2}$ checkers of $(r \geq N)$; BMM: $\frac{n\log_2 n}{2}$ mults, $\frac{3n\log_2 n}{2}$ additions
Our FR-AIC &PR-AIC	Algebraic Invariant Check with Kyber parameter	29.8/ 20.01	0/0	4/ 2.8	Supports **per-cycle fault detection** in **multiplier**, **adder**, and **subtractor** operations, therefore **more diagnosability**, early fault detection avoids redundant computation, unlike [2, 3, 6, 28], achieving 96.16%~~100% detection efficiency (FR-AIC) with no timing overhead and limited area (29%) and energy (4%) overheads. **Fault injection carried out in** *mult, sub* **and** *adder.*	$\frac{n\log_2 n}{2}$ mult & $\frac{n\log_2 n}{2}$ additions

NR: Not Reported; $n-1$: polynomial degree; q: modulus.

to appear at the NTT output. Like the method of Ahmadi et al. [3], it cannot identify the time at which the fault occurred or which NTT operation was corrupted, resulting in limited diagnosability. Moreover, late detection wastes computation power.

1.2.3 Ausmita et al. [30, 31] and Canto et al. [10]. Both Ausmita et al. [30, 31] and Canto et al. [10] implemented a recomputation-based fault detection method primarily targeting the multiplication operation, which can also be applied to other hardware components, including the PEs of the FPGA based NTT. Ausmita et al. [30, 31] used recomputation with the negate-number method, in which the two's complement of α and ω is taken and multiplied. Since $(-\alpha) \cdot (-\omega)$ must be equal to $\alpha \cdot \omega$, the result is used to detect faults. On the other hand, Canto et al. [10] used **Recomputation with Shifted Operand** (**RESO**) where they right-shift α and ω before performing the multiplication. The final left-shifted result is then compared with the original multiplication result to detect fault. The work by Ausmita et al. [31] incurs an overhead of 18.3% in area, 11.4% in delay, and 10.67% in energy. In contrast, other work by Ausmita et al. [30] reports higher overheads of 21.5% in area, 13.71% in delay, and 11.2% in energy. For NTRU implementation, Canto et al. [10] report 39.6% in area, 16.7% in delay, and 3.2% in energy. Both RENO and RESO require $\frac{n \log n}{2}$ additional multiplications. Moreover, RENO requires $\frac{n \log n}{2}$ two's-complement operations, whereas RESO requires $\frac{n \log n}{2}$ shift operations. The main disadvantage of the RENO and RESO models is that fault detection is based only on the multiplier of the PE. As discussed in Section 1.2.2, the entropy of secret polynomials in PQC and FHE can be reduced by attacking the multiplier and forcing the twiddle factor ω to zero. However, entropy reduction is also possible if an attacker forces the intermediate value V ($\alpha[k+m] \cdot \omega$) to zero in the adder or subtractor of the PE during the NTT operation. Although RENO and RESO can detect both intentional and unintentional faults occurring in the multiplier immediately, it cannot detect faults occurring in the adder or the subtractor.

1.2.4 Bauer et al. [6]. Bauer et al. [6] implemented a fault detection module for NTT based on evaluation and interpolation mechanism. They have taken a polynomial $\alpha(X)$, the forward NTT of $\alpha(X)$ is represented as $\alpha(w^j)$. Here, interpolation (I) and evaluation (E) refer to algebraic techniques where the NTT output is treated as polynomial evaluations used by Bauer et al. [6] at roots of unity. Fault detection is achieved by reconstructing polynomial $\alpha(X)$ through interpolation and verifying correctness via redundant evaluation. The fault checking equation can be written as: $E(NTT^{-1}(\alpha(w^j))) = I(NTT(\alpha(X)))$. The ARM Cortex M4-based fault detection module for NTT proposed by Bauer *et al.* [6] requires an additional $2n-1$ multiplications and $2n-2$ additions to perform fault detection, resulting in approximately 72% timing overhead. Such a significant increase in execution time may render NTT unsuitable for modern high-speed security processors that support 5G, 6G, and future high-throughput communication systems. Here, fault detection is performed only after the completion of the NTT computation. Therefore, similar to Ahmadi *et al.* [3], the method proposed by Bauer *et al.* [6] exhibits limited diagnosability and wastes computation energy.

1.2.5 Aghapour et al. [2]. Aghapour et al. [2] do not directly target the NTT itself; instead, they focus on the **Barrett Modular Multiplication** (**BMM**) and **Barrett Modular Reduction** (**BMR**) operations implemented using an FPGA-based approach, which constitute core components of the NTT computation. In BMR, two remainders (r_1 and r_2) need to be computed. As shown below, the final remainder r is the difference of between r_1 and r_2. $r_1 = u \bmod b^{p+1}, r_2 = (\hat{q}N) \bmod b^{p+1}, r = r_1 - r_2$. Here, u is a $2p$-bit integer, N is a p-bit integer that serves as the base of the modulus, $b = 2^p$, and $\hat{q}$ is a precomputed constant. To compute the final result from the r, BMR needs another *while* loop as stated : **while** $(r \geq N)$ **do** $r \leftarrow r - N$. According to the theoretical properties of Barrett reduction [23], with a probability of approximately 99%, only a single subtraction is required in

the **while** loop, whereas with a probability of about 1%, two corrective subtractions are required. Aghapour *et al.* [2] exploit this probabilistic behavior to detect faults in BMR. Specifically, any deviation from the expected subtraction probability is interpreted as an indication of a fault in the BMR process. For BMM, they employ the **Recomputation with Modular Offset** (**REMO**) method. In this approach, during the computation of $XY \bmod N$, a recomputed result $(X + k_1 N)(Y + k_2 N) \bmod N$ is evaluated in parallel, where k_1 and k_2 are randomly chosen integers. Under fault-free conditions, both computations must yield identical results; any mismatch indicates a fault. If you accommodate both BMR and BMM in the NTT, there will be $\frac{n \log_2 n}{2}$ checks of $(r \geq N)$ and $\frac{n \log_2 n}{2}$ addition operations to compute the probability ratio of the **while** loop in BMR. The BMR incurs an overhead of 19.4% in area, 2.02% in time, and 31.73% in energy.

The BMM in NTT incurs $n \log_2 n$ additions operations for proposed modular offset method, $\frac{n \log_2 n}{2}$ additions for wordwise summation, and $\frac{n \log_2 n}{2}$ multiplications operations for backup of barrett multiplications, leading to area, time, and energy overheads of 17.1%, 16.16%, and 21.87%, respectively. Similar to Ausmita et al. [30, 31] and Canto et al. [10], this fault model is not designed to detect faults in the adder and subtractor of the NTT, despite the possibility of entropy-reduction attacks using the adder and the subtractor. The probabilistic behavior–based fault detection method in BMR requires a large number of samples, which limits its diagnosability and leads to wasted computation power, similar to Refs. [3, 6], and [28].

1.2.6 Khan et al. [18]. Khan et al. [18] implemented parity and Hamming code based fault checking for the polynomial coefficient memory and twiddle factor memory of the NTT on an FPGA. In this architecture, parity-based fault detection for NTT coefficients requires $\frac{n \log n}{2} \cdot (\log q - 1)$ additional XOR operations. The encoder and syndrome computation in the Hamming-code-based protection of twiddle factors require $2 \cdot 5 \cdot \log q$ XOR operations, and a decoder logic. All detection and correction operations are fully pipelined with the NTT butterfly computation, resulting in negligible timing overhead but an $\sim$ 45% increase in hardware area.

ALGORITHM 1: Iterative NTT (Decimation-in-Time)

Input: $\alpha[0..n-1]$, twiddle table $\omega[\,]$, modulus q
Output: $\overline{\alpha}[0..n-1]$
$m \leftarrow 1$
while $m < n$ **do**
  $m_2 \leftarrow 2m$ ▷ Butterfly size
  for $j = 0$ to $m - 1$ **do**
    $w \leftarrow \omega[j \cdot (n/m_2)]$ ▷ Twiddle for this j
    for $k = j$ to $n - 1$ step m_2 **do**
      $V \leftarrow w \cdot \alpha[k + m] \bmod q$
      $U \leftarrow \alpha[k]$
      $\alpha[k] \leftarrow (U + V) \bmod q$
      $\alpha[k + m] \leftarrow (U - V) \bmod q$
    end for
  end for
  $m \leftarrow m_2$
end while
return $\overline{\alpha}$

1.3 Our Contribution

- **Efficient Fault Detection for NTT:** We introduce a new algebraic invariant-based fault checking (AIC) methodology that supports polynomial coefficients of arbitrary degree $n-1$ and modulus q, making it broadly applicable to PQC and FHE implementations. This design also supports any number of PEs to accelerate the NTT based on the requirements of the application. The proposed AIC framework offers two variants: Full Recomputaion based AIC ($FR-AIC$) and Partial Recomputaion based AIC ($PR-AIC$) for fault detection. While $FR-AIC$ maximizes fault-detection coverage with higher area, latency, and energy costs, $PR-AIC$ enables a lightweight design by trading detection coverage for substantially reduced overhead.
- **Comprehensive Simulation and Emulation of Fault Attack:** The proposed AIC-based fault-tolerant NTT architecture is evaluated under a wide range of simulation and emulation fault scenarios. Fault injection in the multiplication, addition, and subtraction units of the PE is emulated using a dedicated fault-injection framework. During operation, the NTT executes natively on the FPGA, while the locations of the faulty bits are supplied by a host CPU via the PCI bus, enabling controlled and repeatable fault-emulation experiments. In addition, the architecture is evaluated through simulation in a Python environment. Two fault models are considered in both emulation and simulation: random bit flips and burst bit flips.
- **Hardware Benchmarking:** Different NTT variants, derived from FHE and PQC parameter sets, are evaluated using the proposed AIC framework on Artix 7 FPGA. Both $FR-AIC$ and $PR-AIC$ operate fully in parallel with the NTT datapath and therefore introduce no timing overhead. $FR-AIC$ incurs area and energy overheads of 29.88% and 4%, respectively, while $PR-AIC$ reduces these overheads to 20.06% and 3%. The error-detection efficiency of $FR-AIC$ ranges from 96.16% to 99.99%, depending on the fault-injection mode, number of corrupted bits, and fault location. $PR-AIC$ achieves 96.15%–99.99% detection efficiency for faults in the adder and subtractor units.

The remainder of this article is organized as follows. Section 2 presents the preliminaries of the NTT. Section 3 describes the proposed fault-detection strategy. Section 4 details the hardware architecture. Section 5 discusses the implementation and experimental results. Section 6 concludes the article.

2 Preliminaries

The **Number Theoretic Transform** (**NTT**) is a special case of the **Discrete Fourier Transform** (**DFT**) to finite-field arithmetic, defined over $\mathbb{Z}_q$, where q is the prime modulus. An n-point NTT contains a primitive Nth root of unity ω_n in the field $\mathbb{Z}_q$, which is referred as the twiddle factor. It takes an input vector $a = (\alpha_0, \alpha_1, \alpha_3, \ldots, \alpha_{n-1})$ and produce the output vector $\overline{\alpha} = (\overline{\alpha}_0, \overline{\alpha}_1, \overline{\alpha}_2, \ldots\ldots, \overline{\alpha}_{n-1})$ where $\alpha_i, \overline{\alpha}_i \in \mathbb{Z}_q$. The output vector $\overline{\alpha}$ is computed as $\overline{\alpha}_j = NTT(\alpha)_j = \sum_{i=0}^{n-1} \alpha_i \omega_n^{ji} \ mod\ q$, where $j = 0, 1, \ldots, n-1$. The n-point inverse NTT (INTT) follows a similar computation as NTT. The twiddle factor ω_n is replaced by its inverse ω_n^{-1} and the resulting output normalized by the scaling factor n^{-1}, whereas $\omega_n^{-1}, n^{-1} \in \mathbb{Z}_q$. The INTT is defined by $\alpha_j = INTT(\overline{\alpha})_j = n^{-1} \sum_{i=0}^{n-1} \overline{\alpha}_i \omega_n^{-ji} \ mod\ q$. The NTT algorithm, adapted from the Fast Fourier Transform, such as **Decimation-in-Time** (**DIT**) can be computed in $O(n \log n)$. It uses the modular integer operations instead of complex arithmetic, which makes NTT suitable for high performance and hardware friendly polynomial multiplication in cryptographic schemes. Algorithm 1 implements an iterative DIT NTT to compute the transformation of n-point vector $\alpha[0, 1, 2, \ldots, n-1]$ over the finite field $\mathbb{Z}_q$. It requires a precomputed table of twiddle factors containing $\omega^0, \omega^1, \omega^2, \ldots, \omega^{n-1}$. Line-5 partitions the input vector into blocks of $2m$ size. In each block pairwise butterfly operations U and V are performed using

the precomputed twiddle factor, as shown in lines 9–10. Then, Lines 11–12 perform the modular addition and subtraction to update the corresponding coefficients in place. The algorithm completes the NTT operations in $\log_2 n$ stages.

3 Proposed Fault Detection Strategy

This section proposes our recomputation based **Algebraic Invariant Checking** (**AIC**) scheme for the NTT architecture. Since the PE is the fundamental building block of the NTT architecture, our proposed AIC scheme is specifically designed to safeguard all three arithmetic operations: addition performed by the *adder*, subtraction performed by the *sub*, and multiplication performed by the *mult*, within the PE. We discuss two types of AIC countermeasures for NTT: (A) Full Recomputation-based AIC (*FR-AIC*) and (B) Partial Recomputation-based AIC (*PR-AIC*).

3.1 Full Recomputation Based AIC (FR-AIC)

The PE used for the butterfly operation inside NTT consists of three fundamental computational blocks: a $\log q$ bits multiplier (*mult*), a $\log q$-bit adder (*add*) and a $\log q$-bit subtractor (*sub*). As shown in line 11 and line 12 of NTT Algorithm 1:

$$\alpha[k] = (\alpha[k] + \alpha[k] + \alpha[k+m].\omega) \; mod \; q \tag{1}$$

$$\alpha[k+m] = (\alpha[k] + \alpha[k] - \alpha[k+m].\omega) \; mod \; q \tag{2}$$

Equations (1) and (2) can equivalently be expressed as Equations (3) and (4), respectively at the output of NTT butterfly operation.

$$\alpha[k] = (U + V) \; mod \; q \tag{3}$$

$$\alpha[k+m] = (U - V') \; mod \; q \tag{4}$$

Our *FR-AIC* requires an extra $\log q$ bits multiplier (*mult AIC*) and a $\log q$ bits adder/subtractor (*add*/*sub*). The *adder* inside the *PE* of the NTT is used to add U and V, as shown in Equation (3). The *sub* block inside the *PE* of the NTT is used to subtract V' from U, as shown in Equation (4). Similarly, *mult* block is used to multiply $\alpha[k+m]$ and ω to compute V. The *FR-AIC* uses the *mult AIC* to compute V' (line 10 of Algorithm 2) to multiply $\alpha[k+m]$ and ω again. This V' is a backup of V. The multiplication blocks *mult* and *mult AIC* are used to compute V and V' in our NTT are not a standard schoolbook multiplier; it performs Barrett multiplication, where the two $\log q$-bit operands $\alpha[k+m]$ and ω are multiplied and subsequently reduced modulo q. Then, the V and V' are sent to the *adder* and *sub* blocks. This extra multiplier (*mult AIC*) is required to detect faults that occur during the multiplication of $\alpha[k+m]$ and ω. If the same multiplier is used, faults occurring in the *adder* and *sub* units can be detected by our *AIC*. However, faults in the *mult* unit cannot be detected because the effect of a fault occurring in *mult* may cancel out in the final algebraic invariant calculations of E_1 and E_2, as shown in Equations (5) and (6), respectively. As told earlier, Our *AIC* method also adds an extra block named as adder/ subtractor (*adder*/*sub*) to compute Equations (5) and (6) for E_1 and E_2 (lines 14 and 15 of Algorithm 2), respectively.

$$E_1 = (U + V) + (U - V') \; (mod \; q) \tag{5}$$

$$E_2 = (U + V) - (U - V') \; (mod \; q) \tag{6}$$

Here $U, V \in \mathbb{Z}_q$, and define the NTT butterfly outputs. If no fault occurs in the multiplier, then $V = V'$, and if no fault occurs in the *adder* and *sub*, using the additive inverse property of modular arithmetic in Equation (5):

$$V + (-V') \equiv 0 \pmod{q} \tag{7}$$

Thus,

$$E_1 \equiv U + U \pmod{q} \equiv 2 \cdot U \pmod{q} \equiv 2 \cdot \alpha[k] \pmod{q} \tag{8}$$

Similarly, by distributing the subtraction (multiplying the inner terms by -1), and assuming no fault occurs in the multiplier so that $V = V'$, and no fault occurs in the *adder* or *sub*, we obtain:

$$\begin{aligned} E_2 \equiv (U + V) - (U - V') \pmod{q} &\equiv U + V - U + V' \pmod{q} \\ \equiv 2 \cdot V \pmod{q} &\equiv 2 \cdot \alpha[k+m] \cdot \omega \pmod{q} \end{aligned} \tag{9}$$

Here, the sizes of E_1 and E_2 before modulus reduction are $\log q + 1$ bits because $\alpha[k]$ and $\alpha[k+m] \cdot \omega$ (the outputs of Barrett reduction) are $\log q$ bits each. To store twice the value of $\alpha[k]$ and $\alpha[k+m] \cdot \omega$, only one extra bit is needed. If there is no fault occurs in *mult*, *adder* and the *sub* of the *PE* of NTT, the 1^{st} bit to $\log q^{th}$ bits of E_1 should be equal to $\alpha[k]$. Similarly, if there is no any fault occurred in *mult*, *adder* and the *sub*, the 1^{st} bit to $\log q^{th}$ bit of E_2 should be equal to $\alpha[k+m] \cdot \omega$. As shown in the proposed NTT in Algorithm 2, the 0^{th} bit of the memory address k of the polynomial coefficient memory is used in the computation of E_1 and E_2. For any odd memory address k in the polynomial coefficient memory, E_1 is used to detect faults in the *PE*, whereas for any even memory address k, E_2 is used to detect faults in the PE.

ALGORITHM 2: Proposed $FR-AIC$ for NTT

Input: $\alpha[0..n-1]$, twiddle table $\omega[\,]$, modulus q
Output: $\overline{\alpha}[0..n-1]$
$m \leftarrow 1$
while $m < n$ **do**
  $m_2 \leftarrow 2m$ ▷ Butterfly size
  for $j = 0$ to $m-1$ **do**
    $w \leftarrow \omega[j \cdot (n/m_2)]$ ▷ Twiddle for this j
    for $k = j$ to $n-1$ step m_2 **do**
      $V \leftarrow w \cdot \alpha[k+m] \bmod q$
      $V' \leftarrow w \cdot \alpha[k+m] \bmod q$
      $U \leftarrow \alpha[k]$
      $\alpha[k] \leftarrow (U+V) \bmod q$
      $\alpha[k+m] \leftarrow (U-V') \bmod q$
      $E_1 = (U+V) + (U-V')\ (mod\ q)$
      $E_2 = (U+V) - (U-V')\ (mod\ q)$
      if k[0]=1 **then**
        if $E_1[\log q\ :\ 1]! = \alpha[k]$ **then**
          Fault=1
        end if
      else
        if $E_2[\log q\ :\ 1]! = \alpha[k+m].\omega$ **then**
          Fault=1
        end if
      end if
    end for
  end for
  $m \leftarrow m_2$
end while
return $\overline{\alpha}$

The algebraic invariant checks on E_1 and E_2 are performed in the PE for each NTT butterfly operation to detect unintended or deliberate faults in the arithmetic operations of the adder, subtractor, and multiplier within the PE.

3.2 Partial Recomputation Based AIC (PR-AIC)

The proposed *FR-AIC* costs one extra Barrett multiplier (*mult AIC*), an adder cum subtractor (*add/sub*) block. In our design, we use DSP blocks available in FPGA for both these *mult* and *mult AIC*. The DSP blocks in Xilinx FPGAs are significantly more latency-efficient than LUT-based multipliers and reduction operations, especially when the input sizes are large. However, DSP blocks have more area overhead compared with LUT based design. To reduce the area overhead of the Full AIC scheme, we propose a partial recomputation-based AIC approach. Instead of performing a full width multiplication followed by modular reduction to compute V' from α and ω, we use the $\frac{\log q}{2}$ least significant bits of α, denoted by α_l, and the $\frac{\log q}{2}$ least significant bits of ω, denoted by ω_l. This reduced precision multiplication is implemented using LUTs rather than DSP blocks. The smaller bit-width multiplication implemented using LUTs does not significantly affect latency. Now, α and ω can be represented by Equations (10) and (11), respectively.

$$\alpha[k+m] = \alpha_h[k+m] \cdot 2^{\frac{\log q}{2}} + \alpha_l[k+m] \tag{10}$$

$$\omega = \omega_h \cdot 2^{\frac{\log q}{2}} + \omega_l \tag{11}$$

If $\alpha[k+m]$ and ω are $\log q$ bits in size, then $\alpha_h[k+m]$ and ω_h correspond to the bits ranging from $\frac{\log q}{2}$ to $\log q - 1$ of $\alpha[k+m]$ and ω, respectively. Similarly, $\alpha_l[k+m]$ and ω_l correspond to the bits ranging from 0 to $\frac{\log q}{2} - 1$ of $\alpha[k+m]$ and ω, respectively. Therefore, the product of $\alpha[k+m]$ and ω can be expressed as shown in Equation (12).

$$\begin{aligned}
&\alpha[k+m] \times \omega \\
&= (\alpha_h[k+m] \cdot 2^{\frac{\log q}{2}} + \alpha_l[k+m])(\omega_h \cdot 2^{\frac{\log q}{2}} + \omega_l) \\
&= \alpha_l[k+m] \cdot \omega_l + 2^{\frac{\log q}{2}}(\alpha_h[k+m] \cdot \omega_l + \alpha_l[k+m] \cdot \omega_h) + 2^{\log q} \cdot \alpha_h[k+m] \cdot \omega_h \\
&= \alpha_l[k+m] \cdot \omega_l + 2^{\frac{\log q}{2}} \cdot S + 2^{\log q} \cdot \alpha_h[k+m] \cdot \omega_h
\end{aligned} \tag{12}$$

If we take a modulus by $2^{\frac{\log q}{2}}$ in the both side of Equation (12), we get:

$$\alpha[k+m] \times \omega \,(mod\, 2^{\frac{\log q}{2}}) = \alpha_l[k+m] \cdot \omega_l \tag{13}$$

Equation (13) shows that the least $\frac{\log q}{2}$ bits of the final product of $\alpha[k+m]$ and ω (the left-hand side of Equation (13)) are equal to the product of $\alpha_l[k+m]$ and ω_l. In our proposed *PR-AIC* scheme, the *mult AIC* component of *FR-AIC* is replaced with a partial multiplication operation, denoted as *mult PAIC*, which computes $\alpha_l[k+m] \cdot \omega_l$ (left hand side of Equation (13)) instead of the full-width product $\alpha \cdot \omega$. In Partial AIC, the 0 to $\log q - 1$ of V' is taken from the output of *mult PAIC* and the $\log q$ to $2\log q - 1$ bits of V' are taken from V available at the output of *mult*. Therefore, V' can be represented in Equation (14).

$$V' = \underbrace{V(2 * \log q - 1 \,:\, \frac{\log q}{2})}_{from\ mult} \,||\, \underbrace{\alpha_l[k+m] \cdot \omega_l}_{form\ mult\ PAIC} \tag{14}$$

To compute V', the bit range from $\frac{\log q}{2}$ to $2\log q - 1$ is taken from the output of the *mult* unit, while the bit range from 0 to $\frac{\log q}{2} - 1$ is taken from the *mult PAIC* unit. As stated in Equation (14),

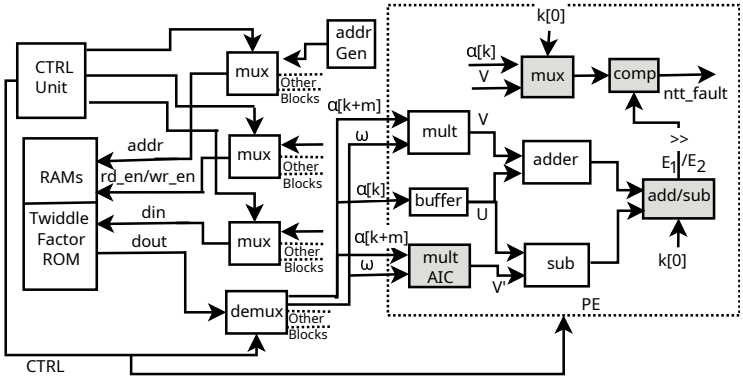


Fig. 1. NTT architecture (1-PE) with proposed FR-AIC fault-detection module.

the result of the backup multiplication process in Partial AIC depends on V and $\alpha_l[k+m] \cdot \omega_l$. If a fault occurs in the *mult* unit and affects the bit range from 0 to $\log q - 1$ of V (the output of *mult*), the result of $\alpha_l[k+m] \cdot \omega_l$ will be altered. Consequently, V and V' will no longer match, allowing the fault to be detected by the proposed model. However, if a fault occurs in the *mult* unit and affects the bit range from $\frac{\log q}{2}$ to $2 \log q - 1$, the values of V and V' remain identical. Consequently, such faults cannot be detected by the proposed model.

4 Hardware Architecture of AIC Based NTT

The NTT based on the proposed *FR-AIC* and *PR-AIC* has the following subcomponents.

4.1 PEs of FR-AIC

An unprotected PE [19] of the NTT performs three fundamental arithmetic operations: addition, subtraction, and multiplication, which are executed by the *adder*, *sub*, and *mult* units, respectively. All operations are performed on modulo q. The *mult* unit multiplies $\alpha[k+m]$ with the twiddle factor ω to produce the output V. The *adder* computes the sum of U and V, as stated in Equation (1), while the *sub* computes the difference by subtracting V from U, as stated in Equation (2). The primary arithmetic components of the proposed *FR-AIC* scheme include an additional multiplier, denoted as *mult AIC*, which is used to compute V', and an adder–cum–subtractor unit, *add/sub*, which computes E_1 and E_2 as defined in Equations (5) and (6), respectively. In each PE iteration, a new pair of coefficients $\alpha[k]$ and $\alpha[k+m]$ is fetched using address k and used to compute the intermediate values U, V, and V'. Rather than employing separate adder and subtractor units, we adopt a unified *add/sub* block to calculate E_1 and E_2. The operation of the *add/sub* block is controlled by the parity of the memory address k. When k is even, the block performs addition by directly forwarding the output of the PE's subtractor, *sub*, which computes $U - V'$. When k is odd, subtraction is computed by feeding the two's complement of the subtractor output, $U - V'$, to the *add/sub* block. In each clock cycle of the NTT, the index k is updated. This parity-based control is implemented using the least significant bit of k and incurs only minimal hardware overhead. By alternating between addition and subtraction in successive cycles, the proposed *add/sub* block efficiently replaces dedicated adder and subtractor units, thereby reducing hardware cost while maintaining full computational functionality. As shown in Figure 1, E_1 and E_2 are right-shifted and then compared with the original values of U and V using the comparator (*comp*). If the values match, no fault is detected; otherwise, the *comp* module signals the presence of a fault in the NTT computation.

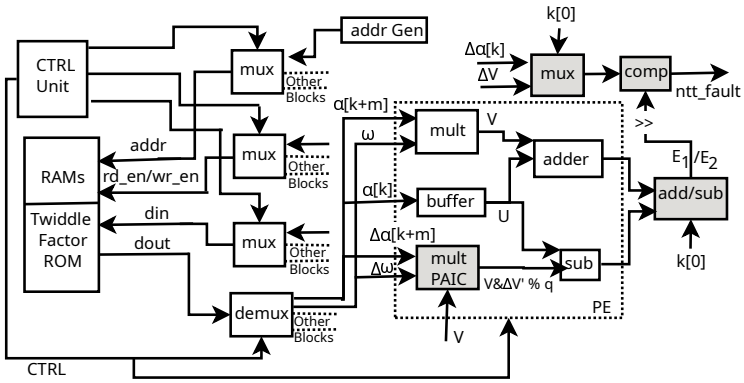


Fig. 2. NTT architecture (1-PE) with proposed PR-AIC fault-detection module.

4.2 PEs of PR-AIC

The primary difference between *PR-AIC* and *FR-AIC* lies in the computation of V' using the backup multiplier. In contrast to the $\log q$ multiplier (*mult AIC*) employed in *FR-AIC*, the *PR-AIC* scheme utilizes a reduced $\frac{\log q}{2} \times \frac{\log q}{2}$ multiplier, referred to as *mult PAIC*. Unlike *FR*-AIC, which is implemented using DSP blocks, the *PR-AIC* is designed using **lookup tables** (**LUTs**). The LUT-based multiplier in *PR-AIC* reduces the effective slice utilization of the NTT and, consequently, lowers power consumption without introducing any additional timing overhead. As defined in Equation (14), the computation of V' in *mult PAIC* is carried out by combining two terms. The first term on the right-hand side is obtained from the main multiplier *mult*, while the second term is generated by the *mult PAIC* unit itself. These two terms are then concatenated to produce the final value of V'. Once V' is available, the remaining operations involving the *add*/*sub* and *comp* blocks, controlled by the index k, are identical to those in the *FR*-AIC scheme described in Section 4.1. The architectural details of *PR-AIC*, along with the other components of the NTT, are shown in Figure 2.

4.3 Memory Units (RAM and ROM)

The proposed NTT architecture requires a RAM to access the polynomial coefficients α and a ROM to store the twiddle factors ω. For the Kyber specification, $n = 256$, $q = 3329$, and $\log q = 12$. Therefore, the RAM and ROM require a depth of 256 and a data width of 12 bits for Kyber-based PQC. In contrast, for FHE applications such as BGV, BFV, the RAM and ROM typically require a depth ranging from 1024 to 16384 and a data width between 60 and 218 bits.

4.4 Control (CTRL) Unit

The control unit is responsible for generating, coordinating, and sequencing the control signals that control the operation of all subcomponents, including RAM, ROM, PEs, and the **address generator** (**AddrGen**)

4.5 Address Generator (AddrGen)

The number of iterations of NTT for n polynomial coefficients is $n \times \frac{\log n}{2}$, where the address of polynomial coefficients RAM is controlled by k and the twiddle factor ROM is controlled by the address j. These j and k are generated by this *AddrGen* block.

4.6 MUX and DEMUX

The inputs and outputs of the NTT can be accessed from different memory units, depending on the specific PQC or FHE algorithm being implemented. For example, in Kyber, a public polynomial vector A is multiplied with the secret polynomial s and the ephemeral random vector r using the NTT. Therefore, the NTT architecture must support flexible selection of input and output memory locations. It is important to note that the input polynomial vectors are often stored in separate memory units rather than in a single shared memory. In particular, the secret vector s is a highly sensitive input and may require a dedicated, isolated memory to satisfy security requirements. Therefore, to provide flexibility in the NTT input and output interfaces, **multiplexers** (**MUX**) and **demultiplexers** (**DEMUX**) are employed to control the datapath between multiple memory units and the NTT input/output ports. The selection signals for the MUX and DEMUX are generated by the $CTRL$ unit.

5 Implementation and Result

The implementation and verification workflow for the baseline NTT and its $FR-AIC$ and $PR-AIC$ variants consists of three phases: implementation of the NTT variants, error injection during simulation and emulation, and final result analysis.

5.1 Implementation

The different PQC and FHE variants of the NTT listed in Table 2 are implemented on an Artix-7 FPGA using the Vivado and Vitis design suites from Xilinx–AMD. The Artix-7 FPGA is interfaced with a host CPU, a 12th Gen Intel Core i5-1245U (12 cores), running Ubuntu 24.04.2 LTS through PCI bus. The NTT variants are characterized by the polynomial degree $(n-1)$, the logarithm of the modulus $\log q$, and the number of PEs, and are denoted as $\{n, \log q, \text{PE}\}$. Table 2 summarizes the implementation details of four NTT variants employed in various PQC and FHE algorithms: $\{256, 12, 1\}$, $\{256, 12, 2\}$, $\{256, 24, 1\}$, and $\{4096, 24, 1\}$.

5.2 Error Injection in Simulation and Emulation

The error injection process to verify the efficiency of our proposed AIC model is made in two subphases such that simulation and emulation. Both these sub-phases have two modes of error injection: *Random Bit Flip* and *Burst Bit Flip*. In the *random bit flip* mode, bits at random positions within the output of *mult*, *adder*, and *sub* units are flipped. In the *burst fault* mode, bits at consecutive positions at the output of the *mult*, *adder*, and *sub* units are flipped. Using these random and burst fault injection modes, we evaluate the error detection efficiency of the proposed *FR-AIC* and *PR-AIC* schemes by varying the number of faulty bits (η) and the location of the faults within the arithmetic units.

5.2.1 Error Simulation. The above-mentioned error scenarios are simulated with Python and executed on an Ubuntu 24.04 system with an *i*5 processor and 8 GB of RAM, utilizing 1.5 million samples.

5.2.2 Error Emulation. For error emulation, an error injector is designed to inject faults inside the *mult*, *adder* and *sub* of the PE. As discussed in the threat model in Section 1.1, we assume the presence of a fault injector (may be in form of Hardware Trojan) is introduced into the NTT design through a compromised CAD tool, possibly at various stages of the design or fabrication process. Although the fault injector is developed using HDL code and *FIFO_Gen* IP [4] at the design stage, it represents an adversarial modification rather than part of the original design.

The fault injector can be triggered by a specific input generated by the host CPU. The host CPU also mentions the number of faulty bits and the mode of fault injection. As shown in Figure 3,

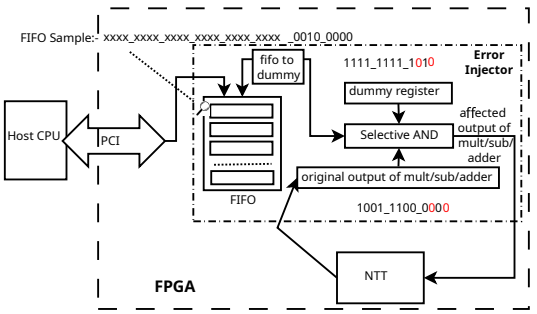


Fig. 3. Fault injector for our NTT.

a dummy register is used to perform a logical AND operation with the original outputs of the arithmetic blocks of the NTT PE: *mult*, *sub*, and *adder*. If the fault is not triggered, the dummy register stores all logic '1's, which are ANDed with the original outputs of the *mult*, *sub*, and *adder*, thereby leaving their outputs unchanged. The */dev/urandom* available on the Ubuntu host CPU is used to generate 1024 random samples, each of size 32 bytes, which are then written into the `FIFO_Gen` through PCI interface. Each sample stored in the FIFO is used to inject faults during each iteration of the NTT. For example, to inject 500 faults over the 1024 iterations of the NTT corresponding to 256 polynomial coefficients in Kyber, a total of 500 random samples must be stored in the FIFO. The 32-bit FIFO data are divided into eight segments, each of 4 bits. That means our fault injector can inject a maximum of 8 faults at the outputs of the *mult*, *sub*, and *adder* blocks. Each 4-bit segment encodes the position of a faulty bit. Consequently, in the *random bit flip* mode, the fault injector can introduce faults in up to eight bit positions simultaneously. As shown in Figure 3, when the fault mode is *random bit flip* and the number of faulty bits is two, only the least significant $4 + 4 = 8$ bits of the FIFO data are utilized. Say the FIFO data is: *xxxx_xxxx_xxxx_xxxx_xxxx_xxxx*_0010_0000. This indicates the fault will be injected at 0^{th} and 2^{nd} positions of the original output of *mult*, *sub*, and *adder*. If we want to insert a fault in *sub*, the dummy register changes its value from 1111_1111_1111 to 1111_1111_1010. The value stored in the dummy register is then ANDed with the original output of the *sub* block. The resulting corrupted output from the *Selective_AND* block is subsequently fed back into the NTT datapath. In the *burst bit flip* mode, multiple 4-bit segments from a FIFO sample are not required; instead, a single 4-bit segment is sufficient to indicate the starting position of the burst fault.

5.3 Results

For our proposed AIC-based fault detection scheme for NTT variants, we report two types of results: overhead and fault detection efficiency.

5.3.1 Overheads. Table 2 reports the area, latency, and energy overheads of *FR-AIC* and *PR-AIC* fault models under different configurations of polynomial degree $n-1$, modulus q, and number of PEs on FPGA platform. For the proposed *FR-AIC* scheme, the area and energy overheads of the NTT implementation under the Kyber configuration $\{n, \log q, \text{PE}\} = \{256, 12, 1\}$ are 29.88% and 4%, respectively. For the same configuration, employing two PEs results in only a negligible increase in area and energy overheads. However, the two-PE NTT completes the NTT operation in 512 clock cycles, whereas the single-PE NTT requires 1024 clock cycles. For a lightweight FHE application based on a tiny BFV configuration, the *FR-AIC* scheme results in area and energy overheads of 18.29% and 4%, respectively. For the proposed *PR-AIC* scheme, the area and energy overheads of the NTT implementation under the Kyber configuration are 20.01% and 2.8%, respectively. For the

Table 2. Comparison of Overhead of *FR-AIC* and *PR-AIC* Methods in Our NTT Variants with Existing Fault Detection Schemes of NTT on FPGA

Work	n, $\log q$, PE	LUT / FF / BRAM / DSP	SEC*	Freq. (MHz)	CC	Total Latency (μs)	Energy (nJ)	ADP̂	T/S
Our (Baseline) in Artix-7 For PQC	256, 12, 1	324 / 504 / 1.5 / 6	1044	186	1024	5.5	825	5638	0.507
Our FR-AIC (Protected) in Artix-7 for PQC	256, 12, 1	359 / 534 / 1.5 / 9	1356 ↑**29.88%**	186	1024	5.5 ↑**0%**	858 ↑**4%**	7322	0.39
Our PR-AIC (Protected) in Artix-7 for PQC	256, 12, 1	351 / 523 / 1.5 / 8	1254 ↑**20.01%**	186	1024	5.5 ↑**0%**	848 ↑**2.8%**	6772	0.422
Ahmadi et al. [3] (Baseline) in Artix-7 for PQC	256, 12, 2	971 / - / 6 / 4	1642	140	648	4.626	693	11740	0.385
Ahmadi et al. [3] (Protected) in Artix-7 for PQC	256, 12, 2	1020 / - / 6 / 6	2055 ↑**25.1%**	140	920	6.568 ↑**41.9%**	1050 ↑**51.51%**	14693	0.217
Ausmita et al. [31] (Baseline) in Zynq Ultrascale+ for PQC	256, 13, 2	17352 / 5171 / NR / NR	4985	NR	NR	2.959	5101	NR	0.215
Ausmita et al. [31] (Protected) in Zynq Ultrascale+ for PQC	256, 13, 2	20420 / 6346 / NR / NR	5899 ↑**18.3%**	NR	NR	3.297 ↑**11.4%**	5645 ↑**10.67%**	NR	0.163
Canto et al. [10] (FrodoKEM Protected) in Kintex Ultrascale+ for PQC	1344, 16, -	NR / NR / NR / NR	1644^{+} ↑**28.4%**	50	NR	NR ↑**32.7%**	NR ↑**0.4%**	10342	NR
Canto et al. [10] (NTRU Protected) in Kintex Ultrascale+ for PQC	701, 13, -	NR / NR / NR / NR	33006^{+} ↑**39.6%**	50	NR	NR ↑**16.7%**	NR ↑**3.2%**	196023	NR
Canto et al. [10] (Saber Protected) in Kintex Ultrascale+ for PQC	256, 13, -	NR / NR / NR / NR	10902^{+} ↑**36.6%**	50	NR	NR ↑**28.3%**	NR ↑**1.2%**	63591	NR
Khan et al. [18] A-0 (Protected) in Virtex-7 for PQC	256, 12, 1	10200 / NR / NR / NR	2550 ↑**29.1%**	59.37	NR	NR ↑**1.6%**	NR	42942	NR
Khan et al. [18] R-0 (Protected) in Virtex-7 for PQC	256, 12, 1	11300 / NR / NR / NR	2825 ↑**44.8%**	71.12	NR	NR ↑**1.44%**	NR	39720	NR
Aghapour et al. [2] BMM (Protected) in Artix UltraScale for PQC	-, 1024, -	13720 / 10522 / - / 4	5145 * ↑**17.1%**	40	7365	184.12 ↑**16.16%**	7.91 ↑**21.87%**	170868	0.001
Aghapour et al. [2] BMR (Protected) in Artix UltraScale for PQC	-, 2048, -	18801 / 11646 / - / 4	6556 * ↑**19.4%**	50	2264	45.28 ↑**2.02%**	3.03 ↑**31.73%**	117352	0.006
Our (Baseline) in Artix-7 for PQC	256, 12, 2	589 / 832 / 3 / 12	2051	186	512	2.75	481	11075	1.034
Our FR-AIC (Protected) in Artix-7 for PQC	256, 12, 2	637 / 860 / 3 / 18	2666 ↑**29.9%**	186	512	2.75 ↑**0%**	501 ↑**4.1%**	14396	0.796
Our PR-AIC (Protected) in Artix-7 for PQC	256, 12, 2	612 / 839 / 3 / 16	2458 ↑**19.8%**	186	512	2.75 ↑**0%**	496 ↑**3.1%**	13273	0.862
Our (Baseline) in Artix-7 for PQC	256, 24, 1	614 / 1077 / 2 / 10	1688	185	1024	5.53	1114	9124	0.313
Our FR-AIC (Protected) in Artix-7 for PQC	256, 24, 1	657 / 1217 / 2/ 15	2216 ↑**31.2%**	185	1024	5.53 ↑**0%**	1173 ↑**5.2%**	11977	0.238
Our PR-AIC (Protected) in Artix-7 for PQC	256, 24, 1	643 / 1211 / 2/ 14	2066 ↑**22.3%**	185	1024	5.53 ↑**0%**	1150 ↑**4.1%**	11167	0.256
Our (Baseline) in Artix-7 for FHE	4096, 24, 1	742 / 1136 / 6.5 / 8	2427	192	24576	127.7	27455	12640	0.151
Our FR-AIC (Protected) in Artix-7 for FHE	4096, 24, 1	825 / 1322 / 6.5/ 12	2871 ↑**18.29%**	192	24576	127.7 ↑**0%**	28554 ↑**4%**	14952	0.127
Our PR-AIC (Protected) in Artix-7 for FHE	4096, 24, 1	802 / 1305 / 6.5/ 11	2764 ↑**13.8%**	192	24576	127.7 ↑**0%**	28241 ↑**2.9%**	14395	0.132

(∗) SEC= (LUT× 0.25) + (FF × 0.125) + × (BRAMs × 200) + (DSPs × 100); This method is taken from [22].
(NR) = Not Reported, (+) : 1 CLB=2 SEC, (ˆ) refer to ADP=SEC × Critical Path(ns), (**)T/S= $\frac{throughput\ in\ mb/s}{SEC}$

Table 3. Error Detecting Efficient for η Bit Random and Burst Flipping Using FR-AIC

# fault bits(η)	Fault Detection Efficiency(%)					
	fault in *adder*		fault in *mult*		fault in *sub*	
	random	burst	random	burst	random	burst
1	96.16	-	97.2	-	97.91	-
2	99.67	99.28	99.93	99.28	99.81	99.28
3	99.95	99.62	99.95	99.62	99.97	99.62
4	99.99	99.71	99.99	99.87	99.99	99.71
5	99.99	99.78	99.99	99.88	99.99	99.79
6	99.99	99.89	99.99	99.89	99.99	99.88
7	99.99	99.99	99.99	99.99	99.99	99.93
9	99.99	99.99	99.99	99.99	99.99	99.99
$\log q$=12, sample size=1.5 million						

same configuration in *PR-AIC*, employing two PEs results in similar in area and energy overheads. For a lightweight FHE application based on a tiny BFV configuration, the *PR-AIC* scheme results in area and energy overheads of 13.8% and 2.9%, respectively. There is no latency overhead for either *FR-AIC* or *PR-AIC*, because both of these hardware modules run in parallel with the minimal NTT variants. To the best of our knowledge, this work presents the first comprehensive comparison of existing FPGA-based NTT fault detection schemes, as summarized in Table 2. The proposed method achieves the lowest latency and energy overhead, with approximately 99.99% fault detection efficiency for *FR-AIC* under multiplier, subtractor, and adder faults, and approximately 99.99% fault detection efficiency for *PR-AIC* under subtractor and adder faults. While for multiplier faults the detection efficiency ranges from 23.18% to 97.23% for *PR-AIC* scheme.

5.3.2 Error Correction Efficiency. As stated in Section 5.2, the error detection efficiency of the *FR-AIC* and *PR-AIC* methods is studied under random bit-flip and burst bit-flip modes by varying the number of faulty bits η at different positions in the *mult*, *sub*, and *adder* units, using the simulation methods described in Section 5.2. For *FR-AIC*, the error detection efficiency is consistently high across all units: *adder*, *mult*, *sub*, and for both random and burst bit flip modes, achieving over 99% efficiency for $\eta \geq 2$ faulty bits. This indicates that *FR-AIC* provides robust error detection irrespective of the type or location of faults. In contrast, *PR-AIC* shows similarly high efficiency for faults in the *adder* and *sub* units but demonstrates lower efficiency for faults in the *mult* unit, particularly under random bit flips for small η. The detection efficiency gradually improves as the number of faulty bits η increases. However, burst errors in the *mult* unit are detected less reliably compared with *FR-AIC*. Overall, *FR-AIC* offers more uniform and reliable fault detection across all operations, while *PR-AIC* is slightly less effective for multiplication faults, highlighting the advantage of *FR-AIC* in scenarios where multiplication errors are critical. Overall, *FR-AIC* offers more uniform and reliable fault detection across all operations, while *PR-AIC* is slightly less effective for multiplication faults; however, *PR-AIC* provides the advantage of significantly lower hardware cost, reducing area overhead by approximately one-third and energy consumption by about one-fourth. It is to be noted that any fault in the adder, subtractor, or multiplier that causes a one-value difference in the computed values $2 \cdot V$ or $2 \cdot U$ cannot be detected, since applying a one-bit right shift removes the LSB of $2 \cdot V$ or $2 \cdot U$. It should be noted that we report fault occurrences in only a single arithmetic block at a time in Tables 3 and 4, achieving more than 96% detection efficiency in most cases. Fault occurrences in multiple arithmetic blocks simultaneously are not reported, as they exhibit detection efficiency exceeding 97%. In this model, $\log q$ is consistently

Table 4. Error Detecting Efficient for η Bit Random and Burst Flipping Using PR-AIC

# fault bits(η)	Fault Detection Efficiency(%)					
	fault in *adder*		fault in *mult*		fault in *sub*	
	random	burst	random	burst	random	burst
1	96.15	-	23.18	-	97.91	-
2	99.68	97.91	41.23	26.34	99.80	99.86
3	99.95	98.86	54.98	26.89	99.97	99.37
4	99.99	99.37	65.81	28.36	99.99	99.65
5	99.99	99.65	74.67	29.92	99.99	99.80
6	99.99	99.88	86.74	31.53	99.99	99.93
7	99.99	99.93	90.62	33.27	99.99	99.96
8	99.99	99.99	93.48	35.22	99.99	99.99
9	99.99	99.99	95.56	37.45	99.99	99.99
11	99.99	99.99	97.23	42.23	99.99	99.99
log q=12, sample size=1.5 million						

rounded up to its ceiling value. It is to be noted that Tables 3 and 4 show fault locations in only a single arithmetic block. If faults occur in multiple arithmetic blocks, the fault detection efficiency increases.

6 Conclusion and Future Scope

Since the multiplication block (*mult*) in the NTT PE exhibits significantly greater logic depth and width than the adder (*adder*) and subtractor (*sub*), faults in *mult* tend to have higher propagation impact and more severe security implications. Consequently, most existing fault detection schemes for NTT primarily focus on protecting the *mult* unit. However, despite their relatively lighter logic, both *adder* and *sub* remain vulnerable to intentional and unintentional faults. In particular, the zeroization fault attack on twiddle factors discussed in Section 1.2.1 is not limited to the *mult* unit and can also affect the *adder* and *sub* blocks. To address this, we propose robust Algebraic Invariant Checking schemes, namely *FR-AIC* and *PR-AIC*, that provide fault protection across all three arithmetic units: *mult*, *adder*, and *sub*. Extensive simulation and emulated fault-injection results show that *FR-AIC* offers better fault detection capability across all arithmetic units, especially in the multiplier, making it suitable for security-critical applications. *PR-AIC*, while lighter in terms of area and energy overhead, exhibits reduced detection efficiency for multiplication faults, although its performance for adder and subtractor faults remains equivalent to *FR-AIC*. These results emphasize a design tradeoff between hardware cost and fault detection robustness. The proposed AIC scheme achieves near-perfect fault coverage ($\approx$ 99.99%) while maintaining modest hardware overheads of $\sim$ 28% in area and $\sim$ 4% in energy. Unlike many existing countermeasures, the proposed approach introduces no timing degradation, thus achieving a well-balanced trade-off between performance and implementation cost. The present AIC model focuses on fault detection and does not perform fault correction. In future work, we plan to incorporate fault correction mechanisms, such as error correction algorithms and dynamic FPGA reconfiguration. Specifically, the system can be reconfigured with a fresh NTT bitstream, either in the same region or a different FPGA floor, depending on the nature of the fault. Additionally, the current AIC approach exclusively targets PE operations. In future work, we will aim at covering other critical components, including memory and interconnect buses, to provide more comprehensive protection. We will also explore techniques to reduce the area overhead associated with both fault detection and correction mechanisms.

References

[1] 2024. *Stateless Hash-Based Digital Signature Standard*. Technical Report FIPS 205. National Institute of Standards and Technology (NIST). Federal Information Processing Standards Publication.

[2] Saeed Aghapour, Kiarash Sedghighadikolaei, Attila A. Yavuz, Bechir Hamdaoui, and Mehran Mozaffari-Kermani. 2025. Efficient fault-detection architectures for barrett reduction and multiplication in classical and post-quantum cryptographic systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2025), 1–13. DOI: https://doi.org/10.1109/TVLSI.2025.3609666

[3] Kasra Ahmadi, Saeed Aghapour, Mehran Mozaffari Kermani, and Reza Azarderakhsh. 2025. Efficient algorithm-level error detection for number-theoretic transform used for kyber assessed on FPGAs and ARM. *ACM Trans. Embed. Comput. Syst.* 24, 5, Article 79 (Sept. 2025), 23 pages. DOI: https://doi.org/10.1145/3762186

[4] AMD Adaptive SoCs and FPGAs Intellectual Property. 2025. FIFO Generator. Retrieved from https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/fifo_generator.html

[5] Kanad Basu, Samah Mohamed Saeed, Christian Pilato, Mohammed Ashraf, Mohammed Thari Nabeel, Krishnendu Chakrabarty, and Ramesh Karri. 2019. CAD-base: An attack vector into the electronics supply chain. *ACM Trans. Des. Autom. Electron. Syst.* 24, 4, Article 38 (April 2019), 30 pages. DOI: https://doi.org/10.1145/3315574

[6] Sven Bauer, Fabrizio De Santis, Kristjane Koleci, and Anita Aghaie. 2024. A Fault-Resistant NTT by Polynomial Evaluation and Interpolation. Cryptology ePrint Archive, Paper 2024/788. Retrieved from https://eprint.iacr.org/2024/788

[7] Jonas Bertels and Ingrid Verbauwhede. 2025. A High Throughput Kyber NTT. Cryptology ePrint Archive, Paper 2025/1763. Retrieved from https://eprint.iacr.org/2025/1763

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory* 6, 3, Article 13 (July 2014), 36 pages. DOI: https://doi.org/10.1145/2633600

[9] Alvaro Cintas Canto, Mehran Mozaffari Kermani, and Reza Azarderakhsh. 2022. Reliable constructions for the key generator of code-based post-quantum cryptosystems on FPGA. *J. Emerg. Technol. Comput. Syst.* 19, 1, Article 5 (Dec. 2022), 20 pages. DOI: https://doi.org/10.1145/3544921

[10] Alvaro Cintas Canto, Ausmita Sarker, Jasmin Kaur, Mehran Mozaffari Kermani, and Reza Azarderakhsh. 2023. Error detection schemes assessed on FPGA for multipliers in lattice-based key encapsulation mechanisms in post-quantum cryptography. *IEEE Transactions on Emerging Topics in Computing* 11, 3 (2023), 791–797. DOI: https://doi.org/10.1109/TETC.2022.3217006

[11] Rafael del Pino, Thomas Espitau, Shuichi Katsumata, Mary Maller, Fabrice Mouhartem, Thomas Prest, Mélissa Rossi, and Markku-Juhani Saarinen. 2023. *Raccoon: A Side-Channel Secure Signature Scheme*. Technical Report. National Institute of Standards and Technology (NIST). Retrieved from https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-1/spec-files/raccoon-spec-web.pdf

[12] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2018. Loop-abort faults on lattice-based signature schemes and key exchange protocols. *IEEE Trans. Comput.* 67, 11 (2018), 1535–1549. DOI: https://doi.org/10.1109/TC.2018.2833119

[13] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144. Retrieved from https://eprint.iacr.org/2012/144

[14] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. 2024. A configurable CRYSTALS-Kyber hardware implementation with side-channel protection. *ACM Trans. Embed. Comput. Syst.* 23, 2, Article 33 (March 2024), 25 pages. DOI: https://doi.org/10.1145/3587037

[15] Khalid Javeed and David Gregg. 2025. Efficient number theoretic transform architecture for CRYSTALS-Kyber. *IEEE Transactions on Circuits and Systems II: Express Briefs* 72, 1 (2025), 263–267. DOI: https://doi.org/10.1109/TCSII.2024.3465273

[16] Khalid Javeed and David Gregg. 2026. AT-NTT: Area-time efficient polynomial multiplication architecture for CRYSTALS-Kyber. *Computers and Electrical Engineering* 133 (2026), 111069. DOI: https://doi.org/10.1016/j.compeleceng.2026.111069

[17] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. 2019. pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844. Available at https://eprint.iacr.org/2019/844

[18] Safiullah Khan, Ayesha Khalid, Ciara Rafferty, Yasir Ali Shah, Maire O'Neill, Wai-Kong Lee, and Seong Oun Hwang. 2023. Efficient, error-resistant NTT architectures for CRYSTALS-Kyber FPGA accelerators. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*. 1–6. DOI: https://doi.org/10.1109/VLSI-SoC57769.2023.10321885

[19] Florian Krieger, Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. 2025. A flexible hardware design tool for fast fourier and number-theoretic transformation architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 1–1. DOI: https://doi.org/10.1109/TCAD.2025.3595834

[20] Stefanus Kurniawan, Phap Duong-Ngoc, and Hanho Lee. 2023. Configurable memory-based NTT architecture for homomorphic encryption. *IEEE Transactions on Circuits and Systems II: Express Briefs* 70, 10 (2023), 3942–3946. DOI: https://doi.org/10.1109/TCSII.2023.3289489

[21] Bin Li, Yunfei Yan, Yuanxin Wei, and Heru Han. 2024. Scalable and parallel optimization of the number theoretic transform based on FPGA. *IEEE Trans. Very Large Scale Integr. Syst.* 32, 2 (Feb. 2024), 291–304. DOI: https://doi.org/10.1109/TVLSI.2023.3312423

[22] Weiqiang Liu, Sailong Fan, Ayesha Khalid, Ciara Rafferty, and Máire O'Neill. 2019. Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 10 (2019), 2459–2463. DOI: https://doi.org/10.1109/TVLSI.2019.2922999

[23] Johannes Mittmann and Werner Schindler. 2020. Timing Attacks and Local Timing Attacks Against Barrett's Modular Multiplication Algorithm. Cryptology ePrint Archive, Paper 2020/946. Retrieved from https://eprint.iacr.org/2020/946

[24] National Institute of Standards and Technology. 2024. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication 203. Retrieved from https://csrc.nist.gov/pubs/fips/203/final

[25] National Institute of Standards and Technology (NIST). 2024. Module Lattice-Based Digital Signature Standard. Federal Information Processing Standards Publication 204. Retrieved from https://csrc.nist.gov/pubs/fips/204/final

[26] Peter Pessl and Lukas Prokop. 2021. Fault attacks on CCA-secure lattice KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 2 (Feb. 2021), 37–60. DOI: https://doi.org/10.46586/tches.v2021.i2.37-60. Artifact available at https://artifacts.iacr.org/tches/2021/a9

[27] Prasanna Ravi, Romain Poussier, Shivam Bhasin, and Anupam Chattopadhyay. 2020. On configurable SCA countermeasures against single trace attacks for the NTT. In *Security, Privacy, and Applied Cryptography Engineering*, Lejla Batina, Stjepan Picek, and Mainack Mondal (Eds.). Springer International Publishing, Cham, 123–146.

[28] Prasanna Ravi, Bolin Yang, Shivam Bhasin, Fan Zhang, and Anupam Chattopadhyay. 2022. Fiddling the Twiddle Constants - Fault Injection Analysis of the Number Theoretic Transform. IACR Transactions on Cryptographic Hardware and Embedded Systems 2023, 2 (2022), 447–481.

[29] Rafael Carrera Rodriguez, Emanuele Valea, Florent Bruguier, and Pascal Benoit. 2024. Hardware Implementation and Security Analysis of Local-Masked NTT for CRYSTALS-Kyber. Cryptology ePrint Archive, Paper 2024/1194. Retrieved from https://eprint.iacr.org/2024/1194

[30] Ausmita Sarker, Alvaro Cintas Canto, Mehran Mozaffari Kermani, and Reza Azarderakhsh. 2023. Error detection architectures for hardware/software co-design approaches of number-theoretic transform. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 7 (2023), 2418–2422. DOI: https://doi.org/10.1109/TCAD.2022.3218614

[31] Ausmita Sarker, Mehran Mozaffari Kermani, and Reza Azarderakhsh. 2022. Efficient error detection architectures for postquantum signature falcon's sampler and KEM SABER. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, 6 (2022), 794–802. DOI: https://doi.org/10.1109/TVLSI.2022.3156479

[32] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, and Christof Paar. 2015. FPGA trojans through detecting and weakening of cryptographic primitives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 8 (2015), 1236–1249. DOI: https://doi.org/10.1109/TCAD.2015.2399455

[33] Zhiming Zhang, Laurent Njilla, Charles A. Kamhoua, and Qiaoyan Yu. 2019. Thwarting security threats from malicious FPGA tools with novel FPGA-oriented moving target defense. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 3 (2019), 665–678. DOI: https://doi.org/10.1109/TVLSI.2018.2879878

Received 10 February 2026; revised 11 June 2026; accepted 13 June 2026