Analysis of EM Fault Injection on Bit-sliced Number Theoretic Transform Software in Dilithium

RICHA SINGH, Worcester Polytechnic Institute, USA
SAAD ISLAM, Worcester Polytechnic Institute, USA
BERK SUNAR, Worcester Polytechnic Institute, USA
PATRICK SCHAUMONT, Worcester Polytechnic Institute, USA

Bitslicing is a software implementation technique that treats an N-bit processor datapath as N parallel single-bit datapaths. Bitslicing is particularly useful to implement data-parallel algorithms, algorithms that apply the same operation sequence to every element of a vector. Indeed, a bit-wise processor instruction applies the same logical operation to every single-bit slice. A second benefit of bitsliced execution is that the natural spatial redundancy of bitsliced software can support countermeasures against fault attacks. A k-redundant program on an N-bit processor then runs as N/k parallel redundant slices. In this contribution, we combine these two benefits of bitslicing to implement a fault countermeasure for the number-theoretic transform (NTT). The NTT efficiently implements a polynomial multiplication. The internal symmetry of the NTT algorithm lends itself to a data-parallel implementation, and hence it is a good candidate for the redundantly bitsliced implementation. We implement a redundantly bitsliced NTT on an advanced 667MHz ARM Cortex-A9 processor, and study the fault coverage for the protected NTT under optimized electromagnetic fault injection (EMFI). Our work brings two major contributions. First, we show for the first time how to develop a redundantly bitsliced version of the NTT. We integrate the protected NTT into a full Dilithium signature sequence. Second, we demonstrate an EMFI analysis on a prototype implementation of the Dilithium signature sequence on ARM Cortex-M9. We perform a detailed EM fault-injection parameter search to optimize the location, intensity and timing of injected EM pulses. We demonstrate that, under optimized fault injection parameters, about 10% of the injected faults become potentially exploitable. However, the redundantly bitsliced NTT design is able to catch the majority of these potentially exploitable faults, even when the remainder of the Dilithium algorithm as well as the control flow is left unprotected. To our knowledge, this is the first demonstration of a bitslice-redundant design of the NTT that offers distributed fault detection throughout the execution of the algorithm.

CCS Concepts: • **Security and Privacy → Post-Quantum Lattice-Based Cryptography**; *Digital Signatures*; Fault Attacks and Countermeasures.

Additional Key Words and Phrases: Dilithium, Bit-slicing, Intra-Instruction Redundancy, Fault Attack Countermeasure, ARM Cortex-A9, Electromagnetic Fault Injection

1 Introduction

Due to advancements in Quantum Computing and the discovery of Shor's algorithm [44], there is a looming threat to our existing public key infrastructure (PKI). With the possibility of powerful and large-scale quantum computers in not too distant future, common RSA- and ECC-based protocols become vulnerable.

Authors' addresses: Richa Singh, rsingh7@wpi.edu, Worcester Polytechnic Institute, 100, Institute Road, Worcester, Massachusetts, USA, 01609; Saad Islam, sislam@wpi.edu, Worcester Polytechnic Institute, 100, Institute Road, Worcester, Massachusetts, USA, 01609; Berk Sunar, sunar@wpi.edu, Worcester Polytechnic Institute, 100, Institute Road, Worcester, Massachusetts, USA, 01609; Patrick Schaumont, pschaumont@wpi.edu, Worcester Polytechnic Institute, 100, Institute Road, Worcester, Massachusetts, USA, 01609.

1539-9087/2023/3-ART $15.00
https://doi.org/10.1145/3583757

NIST recently initiated a Post-Quantum Cryptography (PQC) Standardization process to select and standardize quantum-resistant algorithms for Public Key Encryption (PKE), Key Establishment Mechanisms (KEM) and Digital Signatures (DS) in order to replace RSA and ECC schemes. This process is currently in its fourth and final round which has three finalist candidates for digital signing: Dilithium, Falcon and Sphincs+ [4]. The criterion that determines the selection process includes classical as well as post-quantum (PQ) security guarantees. Besides theoretical security guarantees, the selection criteria also consider implementation cost and performance, as well as resistance against active and passive implementation attacks.

In this contribution, we specifically consider active attacks against post-quantum schemes. Several authors have reported the use of fault injection on structured lattice-based schemes as a basis for attack [14, 15, 32, 36, 37]. The common fault injection vectors in such cryptographic implementations are controlled EMFI pulses [36, 37] and clock glitches [15, 47]. These faults appear as bit-flips in the cryptographic state of the selected PQ scheme which are then used for cryptanalysis. The dominant design concept in countermeasures against fault injection is to apply redundancy in the implementation. The idea is that a fault injection on a redundant design can be detected by analyzing the consistency between redundant executions. For example, time-redundant implementations execute each cryptographic operation multiple times, and compare the consistency between each computed result. The time-redundancy can be implemented at multiple levels of abstraction. At assembly-level, instructions can be duplicated or triplicated [9]. At algorithm-level, critical portions can be computed multiple times [8, 16, 43]. At system-level, a digital signature can be verified after it is computed [20]. Time-redundant implementations may still fail in the following two cases. First, if an attacker has precise control over the fault injection process, it may be possible to inject identical faults in each redundant copy. Multiple-fault injection may fool the consistency check of the countermeasure. Second, the comparison of time-redundant copies must be fault-resistant by itself. A verify-after-sign countermeasure, for example, may still be subject to fault injection on the non-redundant verify operation. An alternate strategy relies on information redundancy provided through error coding techniques [2, 5]. Coding-based techniques are challenging and require not only the generation and error-checking of code-words, but also the modification of computations that operate on encoded information bits.

In this contribution, we pursue a third route for fault-countermeasure development based on spatial redundancy, in which redundant computations are executed simultaneously. We use the intra-instruction redundancy (IIR) technique [35]. The application is re-written as a Boolean program (a single-bit algorithm with simple logic operations) which is then redundantly mapped on the bits of an N-bit processor word. Such a Boolean program can be generated from the application using logic synthesis. As a fault countermeasure, the advantage of spatial redundancy over temporal redundancy is that every single instruction simultaneously operates on redundant bits. Hence, an adversary who wants to thwart the countermeasure not only has to be able to inject identical faults in each redundant copy, but he also has to inject these faults *concurrently* over multiple bits of a single N-bit processor word.

Technically, a Boolean program can be created by bitslicing the application. The bitslicing technique, as proposed by Biham, creates N redundant copies of an application on an N-bit processor [12]. In a spatially-redundant fault countermeasure, groups of k slices compute on the same data to implement k-fold redundancy. Hence, assuming k is a divisor of N, there will be N/k fault-protected redundant copies of the application. We will focus on a spatially-redundant implementation of the Number Theoretic Transform (NTT) as the application. The NTT is a central computation step in many post-quantum schemes including Dilithium, and it is used to efficiently perform polynomial multiplication. The FFT-like structure of the NTT is well suited for protection by bitslicing. The k-fold protected implementation will compute N/k parallel butterfly operations in each stage of the NTT.

Even though the NTT by itself is not a complete signing algorithm, it serves as a critical component in several lattice-based post-quantum algorithms, and it contributes a significant portion of computation cycles to post-quantum Signing and Verification operations. For example, Kim *et al.* [28] benchmark the NTT in a

reference implementation of Dilithium on an ARMv8 processor as using 65.7% and 50.4% of the total cycles needed for Dilithium Signature Generation and Verification, respectively. However, while our protected NTT implementation protects a significant portion of a post-quantum Digital Signature algorithm, we do not claim to fully protect the Dilithium Signature algorithm. A fully protected post-quantum digital signature operation is not within the scope of this paper.

1.1 Our Contributions

The core of our work is a spatially redundant fault countermeasure for the NTT, and its assessment on a realistic target architecture and a realistic application scenario.

(1) We propose a software countermeasure for fault attacks based on data-redundant bitslicing of the NTT. To our knowledge, this is the first work to present a bitsliced NTT implementation, and the first to demonstrate its application as a spatially-redundant fault countermeasure.
(2) We implement the bitsliced NTT design on a 667MHz ARM Cortex-A9 processor integrated in a Zynq FPGA. We make use of the Neon instructions ($N = 64$) and create a dual data-redundant design ($k = 2$).
(3) We perform an exhaustive search process to identify the EM fault-injection parameter space, determining the spatial location of the EM probe, and the moment and power level of the EMFI pulse such that we optimize the fault impact. We are able to increase the percentage of potentially exploitable faults to 10% of the fault injection attempts. To our knowledge, this is the first time that EMFI results are presented for NTT on an advanced CPU (667MHz, Cortex-A9).
(4) We evaluate the fault detection capability of the redundantly implemented NTT. For this dual data-redundant design, we determine that about 62% of all injected faults directly affect the data flow.

1.2 Outline

In Section 2, we introduce the related literature and relevant background knowledge. Section 3 presents our bitsliced design for NTT/Inverse NTT, our fault attack countermeasure design based on dual data-redundant bitslicing and the mapping of the design on the ARM Cortex-A9 processor. Section 4 describes the experimental setup, including the hardware platform, the integration of the protected NTT into Dilithium, and the associated performance analysis of the design. Section 5 presents the EMFI evaluation of the protected NTT, covering the top-down EMFI parameter search for fault injection tuning, and the evaluation of fault protection coverage for the protected NTT. Section 6 concludes the paper.

2 Background

This section summarizes important background knowledge concepts, including Electromagnetic Fault Injection (EMFI), implementation of bitsliced code, and fault countermeasures based on redundant bitslicing.

2.1 EMFI Fault Injection

Fault injection is a well known threat to cryptographic chips. An attacker injects faults in a cryptographic computation by pressuring the chip out of its nominal operating conditions. These faults may result in faulty ciphertext or faulty signatures, which are the starting point for Differential Fault Analysis (DFA) [13] and other cryptanalysis. However, fault injection may also be used to cause data corruption [30], change the control flow of software [40], bypass security mechanisms such as secure boot [17] or cause privileged escalation [22]. Recently, remote (or software-induced) fault injection techniques such as Rowhammer [33], and CLKSCREW [46], have increased the attention to fault effects in cryptographic implementations considerably.

Physics of EMFI There are a multitude of attack vectors to induce faults in digital circuits such as power glitching [49], clock glitching [3], heating [23], EM pulse injection [18], or laser pulse injection [41]. The injected

faults can have transient or permanent effects, and they are injected with varying degrees of temporal and spatial resolution. A higher precision generally implies a more expensive and time consuming fault injection method.

In this contribution we focus on fault injection through electromagnetic pulses. An EM pulse is injected using a tightly wound coil positioned perpendicularly over the chip surface. A current surge through the coil emits a rapidly changing magnetic flux into the chip's metal wires. When these wires form a loop, such as with the on-chip power distribution mesh, the change of magnetic flux results in an electromagnetic force $\mathcal{E}$, a temporary increase or decrease of the voltage across the loop terminals. The electromagnetic force can cause a variety of effects depending on the nature of the wire (power, ground, signal, clock), including temporary signal bit-flips ($0 \rightarrow 1$ or $1 \rightarrow 0$), clock glitches, and flip-flop flips. The ensemble of possible effects stemming from EMFI is captured in the *sampling fault model* introduced by Orbas *et. al* [34]. A more general treatment of fault models is given by Richter-Brockmann *et. al* [38].

The spatial resolution of EMFI is in the millimeter range, because of the need to mechanically position the EM injection coil over the chip. The area affected by the EMFI pulse is directly proportional to the radius of the coil windings, and is typically in millimeter-range as well. The temporal resolution of EMFI is in the order of tens of nanoseconds depending on the power control of the EM injection coil. For these reasons, EMFI is a fault injection tool with mid-range precision. It achieves better spatial locality than classic clock/voltage glitching techniques, but it is not as precise as optical fault injection. However, EMFI is an easy fault injection technique that requires very limited sample preparation. In our experiments, we inject EMFI directly into the chip package of an IC on a standard FPGA development board.

EMFI on ARM processors The target of our EMFI experiments is an ARM processor that executes post-quantum cryptographic software. Thus, the target of EMFI is secure software, while the actual fault occurs in the hardware. Before a hardware fault affects the control- and data-flow of secure software, the fault must propagate across several abstraction layers, including the micro-architecture and instruction-set architecture of the processor. Several authors have studied EMFI for ARM. Early work on EMFI is presented in [19, 42]. Moro *et al.* introduced an EMFI fault model for a 32-bit ARM micro-controller [31], and Elmohr *et al.* later refined the fault model by showing that multi-instruction skips are possible with a single EM pulse [21]. All parts of a typical ARM core are vulnerable to EMFI. For example, Rivière *et al.* performed EM fault injection on the cache of an ARM Cortex-M4 micro-controller [40]; Menu *et al.* presents an EMFI attack on the data-prefetch mechanism of an ARM micro-controller to achieve AES key recovery and AES key resetting with a single fault injection [30]. These examples illustrate the wide variety of fault effects achievable through EMFI.

We observe that the majority of these EMFI are done on relatively simple and low-speed processor architectures, such as AVR ATMEGA [19], ARM Cortex-M0 [21], ARM Cortex-M3 [30, 31], and ARM Cortex-M4 [40]. The selection of EMFI parameters, including the spatial and temporal location of the EM pulse, as well as the power of the EM pulse, becomes a major challenge in complex (pipelined and high-speed) architectures such as the ARM Cortex-A9 considered in our experiments. A significant contribution of our paper is a detailed explanation of the EMFI parameter tuning process for the ARM Cortex-A9 running at 667 MHz.

2.2 Bitsliced Implementation

Bitslicing is a software optimization technique that is popular for cryptographic software implementations because it decouples the wordlength required to execute the algorithm from the wordlength provided by the processor. By reformulating the cryptographic algorithm as a Boolean (1-bit) program, and by allocating sufficient parallel copies of the Boolean program, we can ensure full processor utilization. The downside of bitsliced execution is that the instruction-set of a 1-bit program is very limited and restricted to Boolean operations. Indeed, bitsliced programs are written using only bitwise logical operations (AND, OR, NOT, XOR). That also means that operations that normally benefit from dedicated hardware support, such as hardware multiply, become less

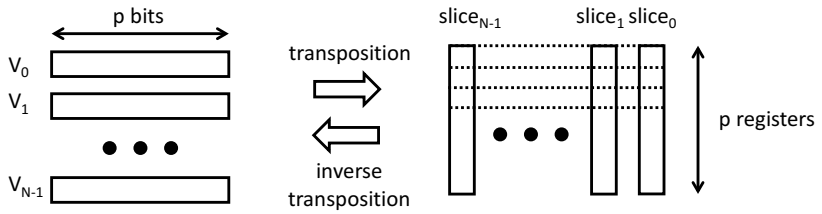


Fig. 1. Interfacing Direct and Bitsliced Code requires Transposition

efficient in bitsliced software [45]. A second drawback is that the parallelization of a boolean program to the full wordlength of the processor significantly increases the register pressure on the processor. Indeed, instead of executing as a single instance of a cryptographic design, the bitsliced program executes as N instances on an N-bit processor, and thus requires N times as many registers to achieve the same storage efficiency.

Bitslicing in cryptography Despite this drawback, cryptographic engineers still apply bitslicing as a manual optimization technique, motivated by primarily two advantages. First, the performance argument is an important consideration for cryptographic engineering. Bitslicing has been used for traditional symmetric-key performance records [1, 24] and lightweight cryptography [7]. Second, because of the lack of control flow operations and the very simple instruction set, bitsliced implementations are naturally constant-time and free from micro-architecture effects. That property has been used to develop constant-time public-key [11] and symmetric-key implementations [29].

Generation of Bitsliced Code The main challenge in writing bitsliced programs is their low abstraction level. Boolean programs are single-bit implementations, at the same abstraction level as gate-level netlists in hardware. Most bitsliced implementations therefore require careful manual logic synthesis and optimization to achieve an efficient design. Kiaei *et al.* developed a programming model called *Parallel Synchronous Programming* (PSP) to support the development of bitsliced programs [26]. The input of PSP is a high-level synchronous description in the form of a Register-transfer Level description in Verilog. Through logic synthesis, the RTL description is converted into a gate-level netlist. The gate-level netlist is then converted into a Directed Acyclic Graph (DAG), where nodes represent boolean operations and edges represent data dependencies. The DAG is leveled to a single node per level. Mapping each node into an equivalent bitwise processor instruction then results in a bitsliced program. The main advantage of the PSP model over direct bitsliced design is that PSP can also capture logic state (as flip-flops) and control flow (as finite state machines). One clock cycle of a PSP program corresponds to one execution of the DAG. Kiaei *et al.* present an automatic C code generation tool based on Yosys, which converts Verilog RTL descriptions into bitsliced PSP functions. We make use of this environment to develop bitsliced versions of the NTT.

Mixing Direct and Bitsliced Code In a practical implementation, bitslicing can be applied to a single function or to a complete program. Therefore we need an interface to transfer data between direct code and bitsliced code. This interface transposes the argument bits as illustrated in Fig. 1. If a given function accepts a variable V_0 of p bits as input, then the bitsliced version of this function will store each bit of V_0 in a different register, and thus use p registers to represent V_0. On an N bit processor, the bitsliced function will compute on N copies of V_0 in parallel, and therefore operate on a block of variables V_0 to V_{N-1}, corresponding to slice 0 to slice $N-1$ of the bitsliced execution. When direct C code calls a bitsliced C function, this bit-level rearrangement is implemented by transposing a bit matrix of $N \times p$ bits into a bit matrix of $p \times N$ bits. On a standard processor, the transposition

is implemented using bit-level manipulation, and contributes an additional overhead to the use of bitsliced implementation. Kiaei *et al.* have proposed custom instructions to accelerate the transposition operation [25].

2.3 Bitsliced Fault Countermeasures

Patrick *et al.* proposed intra-instruction redundancy as a technique to obtain spatial redundancy in software [35]. Given a program P, then the bitsliced execution of P will execute as parallel redundant copies over multiple slices. Depending on the allocation of the redundant copies to slices, different forms of fault countermeasures are obtained. Data redundancy protects the data flow (computations and storage), while control redundancy protects the control flow of a redundant program.

Data redundancy By straightforward duplication of slices, the program becomes data-redundant and can detect faults that affect the computations or storage of data bits. In data redundancy, slice i and slice $i + N/2$ are subjected to the same computations and thus should hold identical results. The rationale is that, because the attacker has limited control over the fault injection process, it is hard to obtain identical faults in two different slices [35].

Fault checking of data-redundant bitslices does not require an `if-then` test which is potential fault injection target of the verification. Rather, fault checking can be done using only data computations. Assume that V is a 32-bit integer containing redundant slices in the upper half of the processor word, then the expression

```
MASK = -((V ^ (V >> 16)) & 0xFFFF) >> 16
```

will compute a `MASK` with value 0 or `0xFFFFFFFF` depending on the occurence of a fault. This mask can be used to mask off (faulty) ciphertext when a fault has occured.

Control redundancy Control-flow faults such as instruction skip cannot be detected using data redundancy. For those cases, Patrick *et al.* propose a hybrid form of redundant bitslicing that introduces time redundancy during bitsliced computation [35]. For example, during the execution of a loop, different slices can compute different iterations of the loop. The design of control redundancy depends on the control flow of the application. In this contribution, our focus is on data-redundant execution of the NTT. In the following section we explain our bitsliced NTT design.

3 Proposed Bit-sliced design of NTT-based Polynomial Multiplication

We propose to use the bitslicing technique on NTT and then utilize this bitsliced NTT design to construct a data-redundant countermeasure for NTT to provide fault-attack resistance.

3.1 Definition of NTT and Inverse NTT

NTT is a generalized version of the well-known Discrete Fourier Transform (DFT) where all the arithmetic is performed in the prime finite field $\mathbb{F}_q$. In ideal lattice-based cryptography, the main operation is polynomial multiplication in the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, where, ring $\mathbb{Z}_q[x] = (\mathbb{Z}/q\mathbb{Z})[x]$ denotes the set of polynomials with integer coefficients modulo q and R_q denotes the ring of polynomials with integer coefficients from the ring $\mathbb{Z}_q$ reduced by a n^{th} cyclotomic polynomial $x^n + 1$. NTT-based polynomial multiplication requires that n is a power of two and that the modulus q is chosen to be a prime such that $q \equiv 1 \bmod 2n$. This way $\mathbb{Z}_q$ contains a primitive n-th root of unity ω and its square root ψ, which means that $\omega^n \equiv 1 \bmod q$.

Definition Let two polynomials $a(x), b(x) \in R_q$ have coefficients $a(x) = (a[0], a[1], \ldots, a[n-1])$ and $b(x) = (b[0], b[1], \ldots, b[n-1])$ respectively. Then, polynomial multiplication $c = a \cdot b \in R_q$ is defined as

$$c = \text{INTT}(\text{NTT}(a) \odot \text{NTT}(b)) \tag{1}$$

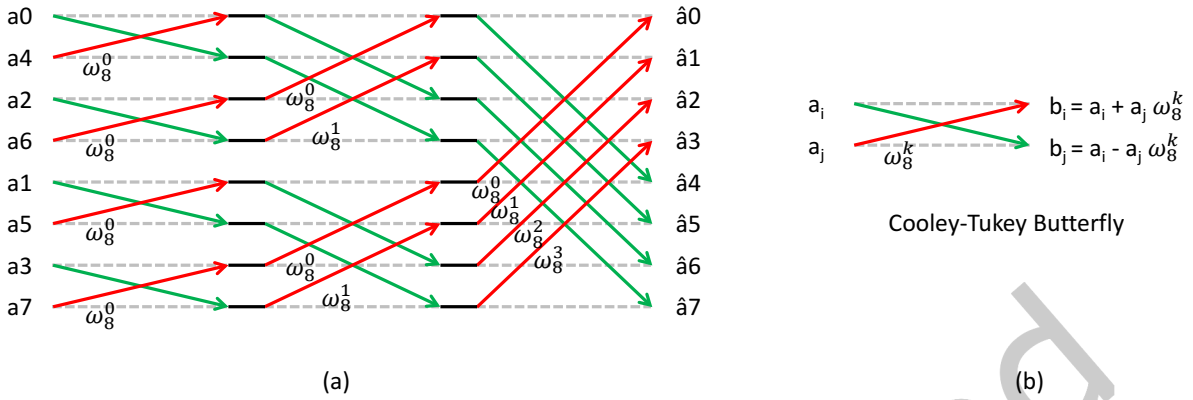


Fig. 2. (a) 8-point Decimate-in-time NTT (b) Cooley-Tukey Butterfly Operation

where, $\odot$ denotes the point-wise multiplication of polynomials. To eliminate the overhead of zero-padding a and b to length $2n$, negative-wrapped convolution property of NTT is used. So, the transformation NTT(a) generates a polynomial $\hat{a}$ whose coefficients can be defined as

$$\hat{a}[i] = \sum_{j=0}^{n-1} \omega_n^{ij} \psi^j a[j] \bmod q \quad \forall i \in [0, n-1] \tag{2}$$

The inverse NTT is computed in exactly same way except that ω_n^{ij} is replaced with ω_n^{-ij} and the final results are scaled by $\psi^{-i} n^{-1}$ factor. The inverse NTT transforms the polynomial $\hat{a}$ back to a whose coefficients are obtained as

$$a[i] = \psi^{-i} n^{-1} \sum_{j=0}^{n-1} \omega_n^{-ij} \hat{a}[j] \bmod q \quad \forall i \in [0, n-1] \tag{3}$$

3.2 Bitsliced NTT/INTT Design

In this section, we describe the decomposition of the NTT/INTT operation into stages, and into bitsliced computation.

Decomposition of NTT/INTT operations The sums of Equation 2 and Equation 3 are computed efficiently using an FFT-like structure [48] which recursively decomposes an N-point NTT into two $N/2$ point NTTs. Figure 2a illustrates the decomposition of an 8-coefficient polynomial NTT into Cooley-Tukey butterfly operations. This NTT completes in $log_2(8) = 3$ stages. Each stage contains $8/2 = 4$ butterfly operations. Figure 2b illustrates the equivalent operations inside each Cooley-Tukey butterfly. Each butterfly operation is completed in $\mathbb{F}_q$ modular arithmetic with a modular addition, a modular subtraction and a modular constant multiplication. The overall dataflow of the NTT requires the input coefficients to be organized in bit-reversed order. For example, the first butterfly uses coefficients $a_0 = a_{000b} = a_{bitrev(000b)}$ and $a_4 = a_{100b} = a_{bitrev(001b)}$.

Data parallelism in NTT/INTT To make bitsliced operation efficient, a high degree of data parallelism is required, so that each slice in an N bit processor word can contribute useful work. The NTT offers data parallelism through the large number of butterfly operations that must be computed in each stage of the Cooley-Tukey NTT. With 32-bit signed coefficients a_i, each butterfly operation (Figure 2b) includes a 32-bit modular addition,

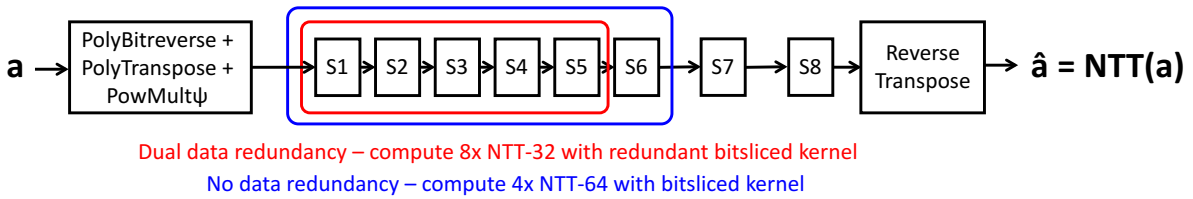


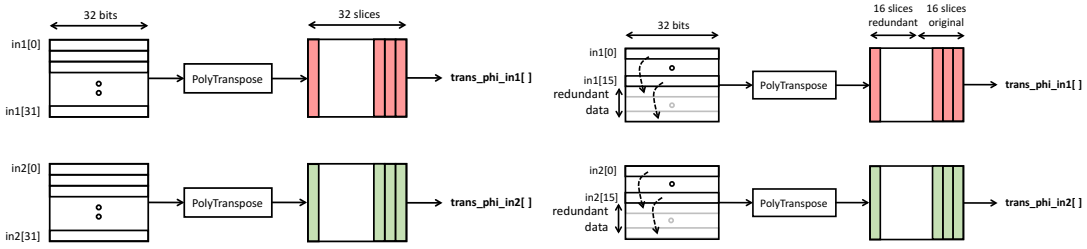
Fig. 3. Block diagram for Bitsliced 256-point NTT.

a 32-bit modular subtraction and a 32-bit modular multiplication. We built a bitsliced butterfly function to compute one stage of the NTT for a 64-coefficient polynomial. The function implements 32 parallel butterfly operations. We mapped Figure 2b in Verilog. Modular reduction for addition and subtraction is performed using conditional subtraction and addition respectively. Modular reduction for multiplication is based on Barrett Reduction algorithm [10] as described in Algorithm 6 of Bannerjee *et al.* [6]. Using the PSP methodology [27], we then generated bitsliced C code from the Verilog expressions. With 32-bit coefficients, our bitsliced butterfly kernel uses 9,294 bitwise operations. This function computes one stage of a 64-point NTT, and it is the core function in our design. Next, we use this function as a building block to create a complete NTT/INTT.

Mapping NTT/INTT computations into bit-sliced format We demonstrate mapping of NTT into bit-sliced computation for 32-bit coefficient precision and 256-element polynomials ($N = 32$, $n = 256$). Fig. 3 shows the step-wise transformation of an input polynomial **a** into a 256-point NTT output polynomial **â**. The 256-point NTT is broken down into four 64-point NTTs, each of which is computed using six calls (S1 to S6 in Figure 3) to the bitsliced kernel. The results of four 64-point NTTs are then combined into two 128-point NTTs (stage 7) and finally into one 256-point NTT (stage 8).

We next briefly explain each step in Fig. 3.

- **PolyBitReversal** - Because of the Cooley-Tukey NTT design (see Figure 2a), the coefficients of the input polynomial **a** must be permuted into bit-reversed order so that the coefficients of the NTT polynomial **â** appear in sequential order.
  The bit-reversed input polynomial is then partitioned into two arrays *in*1 and *in*2, holding 128 even-indexed and 128 odd-indexed coefficients from the bit-reversed polynonial, respectively. The arrays are split into 4 subsets of 32 coefficients each, and these subsets of *in*1 and *in*2 are then pairwise combined to form the 64 inputs to a 64-point NTT block. In this manner, the first 64-point NTT is computed on inputs *in*1[0], *in*1[1], . . . , *in*1[31] and *in*2[0], *in*2[1], . . . , *in*2[31]; the second 64-point NTT is computed on inputs *in*1[32], *in*1[33], . . . , *in*1[63] and *in*2[32], *in*2[33], . . . , *in*2[63]; and so on.
- **PolyTranspose** - Before 32-bit input polynomial coefficients can be processed by a bitsliced function, the inputs have to be transposed. This step as shown in Fig. 4a performs transpose on the two butterfly inputs arrays and converts 32 inputs of *in*1 and *in*2 into 32 bitsliced inputs containing red and green colored slices. This produces the subsets *trans_psi_in*1 and *trans_psi_in*2.
- **PowMul$_\psi$** - The transposed butterfly subsets *trans_psi_in*1 and *trans_psi_in*2 are then passed as input arguments to pointwisemultiplier() bitsliced function at PowMul$_\psi$ step to realize multiplication of polynomial coefficients with respective powers of negative convolution (ψ) factors.
- **Stages 1 to 6** - Output subsets *trans_in*1 and *trans_in*2 from PowMul$_\psi$ step are then processed by the bitsliced butterfly kernel butterflyCompute(), resulting in two butterfly outputs subsets, *trans_out*1 and *trans_out*2, which together from the outputs of a 64-point NTT block. However, an additional reshuffle is needed in between each stage of the NTT. The reason for this can be seen in Figure 2. After any NTT stage,



(a) Bitslicing format (b) Dual data-redundancy bitslicing format

Fig. 4. Representation of NTT polynomial coefficients in bitslicing format without and with dual data-redundancy.

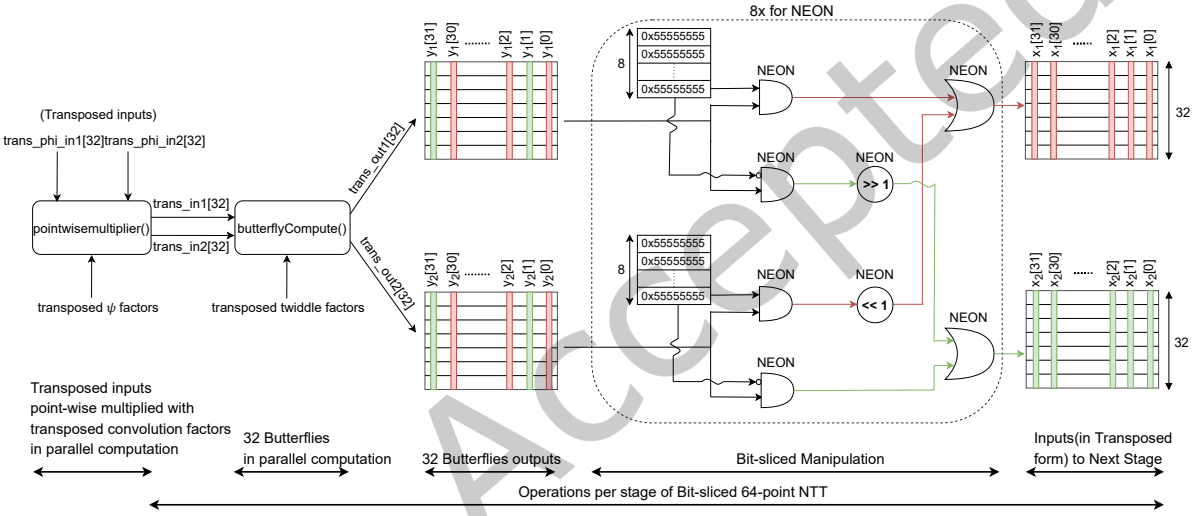


Fig. 5. Data flow diagram of Bit-sliced 256-point NTT composed of four 64-point NTTs. The figure shows operations that occur in each stage of a Bit-sliced 64-point NTT.

some of the red slices of stage i end up at a green slice for stage $i+1$, and vice versa. In order to process 2^{nd} stage inputs by the bitsliced kernel BUTTERFLYCOMPUTE(), all the first butterfly inputs represented by red slices should be placed together in one subset and all the second butterfly inputs represented by green slices should to be placed together in the other subset. We achieve this reshuffling using a bitmasking process as illustrated in Fig. 5. The implementation of this reshuffling can be optimized by using vectorized bitwise instructions, which reduces 32 bitwise operations to 8 SIMD bitwise operations. The last stage of a 64-point NTT does not require reshuffling because all the 32 butterflies are interleaved with each other which causes both the butterfly output subsets to already have all the first butterfly outputs in one subset and all the second butterfly outputs in the other subset. A 64-point NTT output in bit-sliced domain is represented in Fig. 6 using a rectangular block with top-half/red portion occupied by 32 first outputs (in 32 adjacent red slices spread across 32 words) and bottom-half/green portion occupied by 32 second outputs (in 32 adjacent green slices spread across 32 words) of 32 butterflies in the 6^{th} stage.

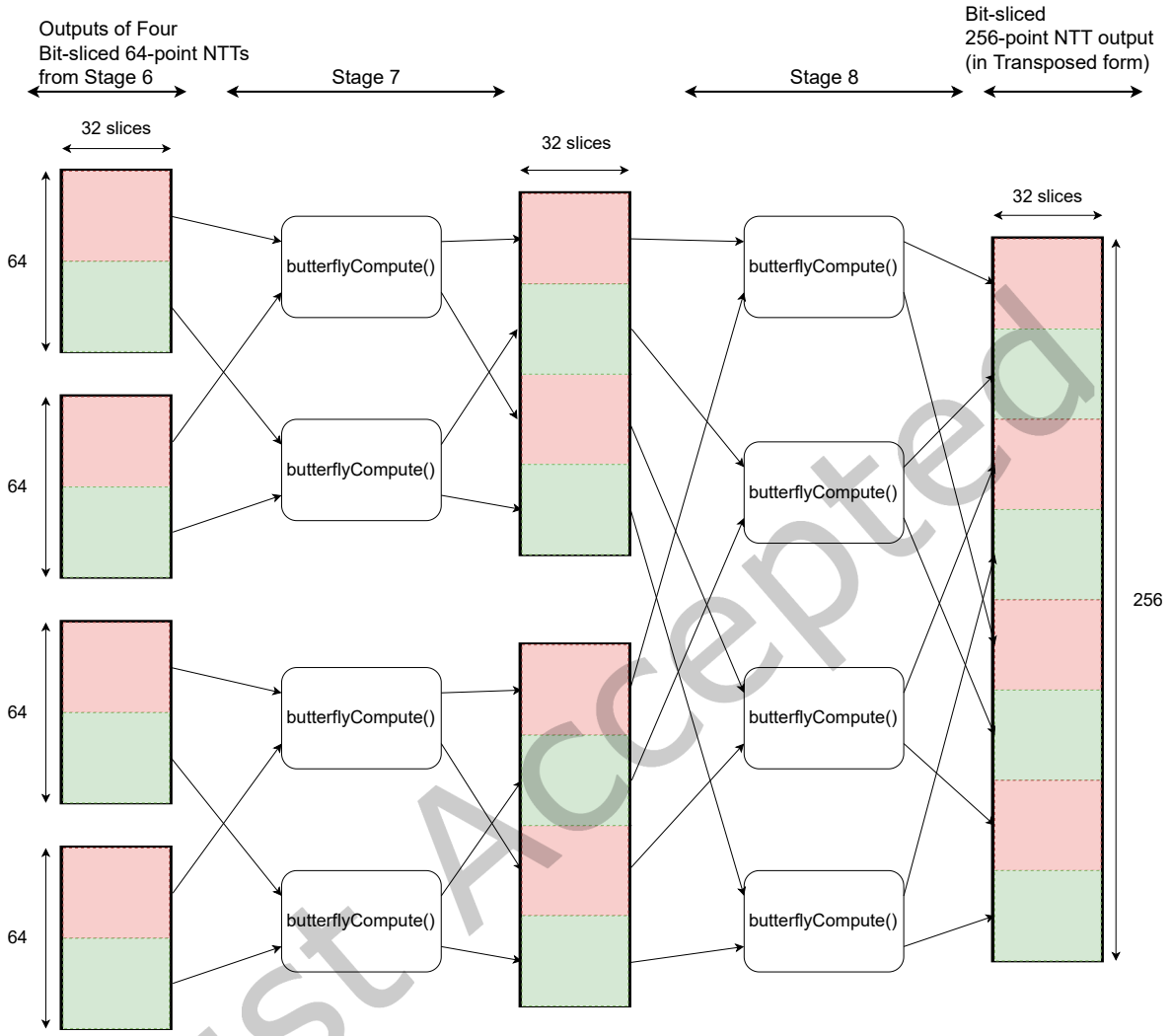


Fig. 6. Data flow diagram in Stages 7 and 8 of Bit-sliced 256-point NTT. A bit-sliced 64-point NTT output from the 6^{th} stage of 256-point NTT is represented using a rectangular block with top-half/red portion occupied with 32 first outputs (in 32 adjacent red slices spread across 32 words) and bottom-half/green portion occupied with 32 second outputs (in 32 adjacent green slices spread across 32 words) of 32 butterflies.

- **Stages 7 to 8 -** To compute the 7^{th} stage of 256-point NTT, four 64-point NTT blocks are combined to form 2 128-point NTT blocks by 4 invocations to the bitsliced butterfly kernel butterflyCompute(). Fig. 6 shows that red portions and green portions of top two 64-point NTT blocks are inputs for the first 32 butterflies and next 32 butterflies, respectively, in the 7th stage of 256-point NTT which combine together to produce first 128-point NTT block. Butterfly outputs are stored back into same memory locations where

the inputs are in. Going to the 8^{th} stage, composition of 2 128-point NTT blocks leads to a single 256-point NTT block which is computed by 4 invocations to the bitsliced butterfly kernel butterflyCompute().

- **ReverseTranspose** - We obtain the final 256-point NTT output by performing last step in Fig. 3, ReverseTranspose to convert the 32-bit 256-point NTT outputs from bit-sliced domain to normal domain.

Algorithm 1 outlines the pseudocode for bitsliced 256-point NTT. We use pre-computed transposed twiddle factors: ω, $\omega^{-1} \bmod q$ and ψ, $n^{-1}\psi^{-1} \bmod q$ in our bit-sliced implementation of 256-point NTT and INTT. The bit-sliced implementation for 256-point INTT is same as 256-point NTT except that there is PowMul$_\psi$ step at the end before the ReverseTranspose step instead of the beginning to perform point-wise multiplication of INTT output polynomial coefficients with respective powers of $n^{-1}\psi^{-1}$ using pointwisemultiplier() bit-sliced function.

3.3 Data-Redundant Countermeasure NTT/INTT Design

In this section, we describe the implementation of Fault-Attack Countermeasure based on data redundancy for NTT/INTT. We utilize the bit-sliced construction defined for NTT/INTT in Section 3.2 as the foundation to add dual data redundancy to it. Redundancy is introduced during the transposition step (Fig. 4b). Throughout the computation we maintain a single redundant slice for each real slice, so that faults can be detected at the end of the 256-point NTT. Because of the additional redundant data, the bitsliced NTT kernel can only be performed on 32 coefficients at a time. The data-redundant bitsliced kernel now covers only 5 stages (Fig. 3) and a 32-point NTT.

The conversion of a direct butterfly to a data-redundant butterfly is easy, since each slice is controlled separately. Except for the use of redundant twiddle factors ω_n^{ij}, a data-redundant bitsliced butterfly kernel is identical to a non-data-redundant bitsliced butterfly kernel. Furthermore, the bitsliced execution ensures that spatial redundancy is maintained at every step of the NTT computation.

However, to achieve a 256-point NTT, stage 6, 7 and 8 have to be modified from the non-redundant version. The data-redundant bitsliced buttefly kernel only supports 32-point NTTs, so it must be applied 8 times at each stage to cover a 256-point NTT. Fig. 7 shows the overall data flow. Direct and redundant data slices are marked with a filled and hashed pattern, respectively. Even and odd sets are marked with red and green colors, as before. A redundant stage-6 NTT computes four 64-point NTTs redundantly using eight calls to the redundant bitsliced butterfly kernel. Similarly, stage 7 and stage 8 can be computed using eight calls to the redundant bitsliced butterfly kernel.

This countermeasure only provides data redundancy and does not cover control faults. We treat the extension of redundant bitslicing to cover control faults on an NTT as future work, and do not further consider it in this paper.

4 Experimental Setup

In this section, we describe the experimental setup, perform performance & footprint evaluation and overhead analysis of the proposed countermeasure.

4.1 Experimental Fault Injection Setup

This section describes our measurement setup for EM Pulse Injection. The setup consists of a hardware part, device under attack, a software part and experimental process.

4.1.1 Hardware Setup: The EM fault injection bench is composed of a control PC, the targ device, an automated XYZ stage, a pulse generator, an oscilloscope and an EMFI transient probe. We use commercially available FI hardware and software tools [39] to build this setup. The target is placed on the XYZ stage as shown in the Fig. 8. We use a classic EMFI probe tip made of a copper winding around a ferrite core. It has a flat-head tip

Algorithm 1 Bitsliced 256-point NTT

Input: $a(x) \in R_q$
Output: $\hat{a}(x) \in R_q$ such that $\hat{a} = \text{NTT}(a)$
Setup: Pre-computed transposed twiddle factors $trans\_w$ and transposed ψ factors $trans\_psi1$ and $trans\_psi2$
$state1 \leftarrow 0, state2 \leftarrow 0$ ▷ Initialization
$bitshufflemask[5] \leftarrow$ {0x55555555, 0x33333333, 0x0F0F0F0F, 0x00FF00FF, 0x0000FFFF}
$bitshufflemask[5] \leftarrow$ {0xAAAAAAAA, 0xCCCCCCCC, 0xF0F0F0F0, 0xFF00FF00, 0xFFFF0000}
$b \leftarrow$ PolyBitReversal(a) ▷ Polynomial Bit-Reversal
for $i \leftarrow 0$ **to** $n/2$ **do** ▷ Store polynomial in butterfly inputs form
    $in1[i] \leftarrow b[i * 2]$
    $in2[i] \leftarrow b[i * 2 + 1]$
end for
for $i \leftarrow 0$ **to** $n/64$ **do** ▷ Transpose butterfly inputs polynomial to bitsliced format
    $trans\_psi\_in1[i * 32] \leftarrow$ Transpose($in1[i * 32]$)
    $trans\_psi\_in2[i * 32] \leftarrow$ Transpose($in2[i * 32]$)
end for
for $i \leftarrow 0$ **to** $n/64$ **do** ▷ Point-wise multiply butterfly inputs polynomial with respective negative convolution factors
    POINTWISEMULTIPLIER($\&trans\_psi\_in1[i * 32]$, $\&trans\_psi1[i * 32]$, $\&trans\_in1[i * 32]$, $state2$)
    POINTWISEMULTIPLIER($\&trans\_psi\_in2[i * 32]$, $\&trans\_psi2[i * 32]$, $\&trans\_in2[i * 32]$, $state2$)
end for
for $j \leftarrow 0$ **to** $n/64$ **do** ▷ Loop over 4 64-point NTTs
    $num \leftarrow 0$
    while $num < 6$ **do** ▷ Loop over 6 stages of 64-point NTT
        BUTTERFLYCOMPUTE($\&trans\_in1[j * 32]$, $\&trans\_in2[j * 32]$, $\&trans\_w[j * 32]$, $\&trans\_out1[j * 32]$, $\&trans\_out2[j * 32]$, $state1$) ▷ Bitsliced computation for 32 butterflies
        if $num != 5$ **then** ▷ Bitsliced manipulation to configure butterfly inputs for next stage
            $mask \leftarrow$ vdupq_n_u32($bitshufflemask[num]$)
            $invmask \leftarrow$ vdupq_n_u32($bitshufflemask[num]$)
            $shiftval \leftarrow 2^{num}$
            for $i \leftarrow 0$ **to** 8 **do** ▷ Iterate over 32 butterfly outputs in bitsliced format
                $out1 \leftarrow$ vld1q_u32($\&trans\_out1[i * 4 + j * 32]$)
                $out2 \leftarrow$ vld1q_u32($\&trans\_out2[i * 4 + j * 32]$)
                $andout1 \leftarrow$ vandq_u32($out1, mask$)
                $andout2 \leftarrow$ vandq_u32($out2, mask$)
                $andnout1 \leftarrow$ vandq_u32($out1, invmask$)
                $andnout2 \leftarrow$ vandq_u32($out2, invmask$)
                $lsout \leftarrow$ vshlq_n_u32($andout2, shiftval$)
                $orout1 \leftarrow$ vorrq_u32($andout1, lsout$)
                $rsout \leftarrow$ vshrq_n_u32($andnout1, shiftval$)
                $orout2 \leftarrow$ vorrq_u32($andnout2, rsout$)
                vst1q_u32($\&trans\_in1[i * 4 + j * 32], orout1$)
                vst1q_u32($\&trans\_in2[i * 4 + j * 32], orout2$)

```
            end for
        end if
    end while
end for
for i ← 0 to n/2 by 32 do                          ▷ Configure butterfly inputs for 7^th NTT stage
    for j ← 0 to 32 do
        trans_in1[j + i] ← trans_out1[j + i]
        trans_in2[j + i] ← trans_out2[j + i]
    end for
end for
                                  ▷ Bitsliced computation for 4 blocks with 32 butterflies/block at 7^th NTT stage
ButterflyCompute(&trans_in1[0], &trans_in1[32], &trans_w[num * 32],
&trans_out1[0], &trans_out1[32], state1)
ButterflyCompute(&trans_in2[0], &trans_in2[32], &trans_w[(num + 1) * 32],
&trans_out2[0], &trans_out2[32], state1)
ButterflyCompute(&trans_in1[64], &trans_in1[96], &trans_w[num * 32],
&trans_out1[64], &trans_out1[96], state1)
ButterflyCompute(&trans_in2[64], &trans_in2[96], &trans_w[(num + 1) * 32],
&trans_out2[64], &trans_out2[96], state1)
for i ← 0 to n/2 by 32 do                          ▷ Configure butterfly inputs for 8^th NTT stage
    for j ← 0 to 32 do
        trans_in1[j + i] ← trans_out1[j + i]
        trans_in2[j + i] ← trans_out2[j + i]
    end for
end for
                                  ▷ Bitsliced computation for 4 blocks with 32 butterflies/block at 8^th NTT stage
ButterflyCompute(&trans_in1[0], &trans_in1[64], &trans_w[(num + 1) * 32],
&trans_out1[0], &trans_out1[64], state1)
ButterflyCompute(&trans_in2[0], &trans_in2[64], &trans_w[(num + 2) * 32],
&trans_out2[0], &trans_out2[64], state1)
ButterflyCompute(&trans_in1[32], &trans_in1[96], &trans_w[(num + 3) * 32],
&trans_out1[32], &trans_out1[96], state1)
ButterflyCompute(&trans_in2[32], &trans_in2[96], &trans_w[(num + 4) * 32],
&trans_out2[32], &trans_out2[96], state1)
for i ← 0 to n/64 do                               ▷ Reverse Transpose NTT outputs polynomial
    out1[i * 32] ← Transpose(trans_out1[i * 32])
    out2[i * 32] ← Transpose(trans_out2[i * 32])
end for
for i ← 0 to n/64 do                    ▷ Store outputs in final 256-point NTT output vector
    for j ← 0 to 32 do
        â[j + 2 * i * 32] ← out1[j + i * 32]
        â[j + (2 * i + 1) * 32] ← out2[j + i * 32]
    end for
end for
```

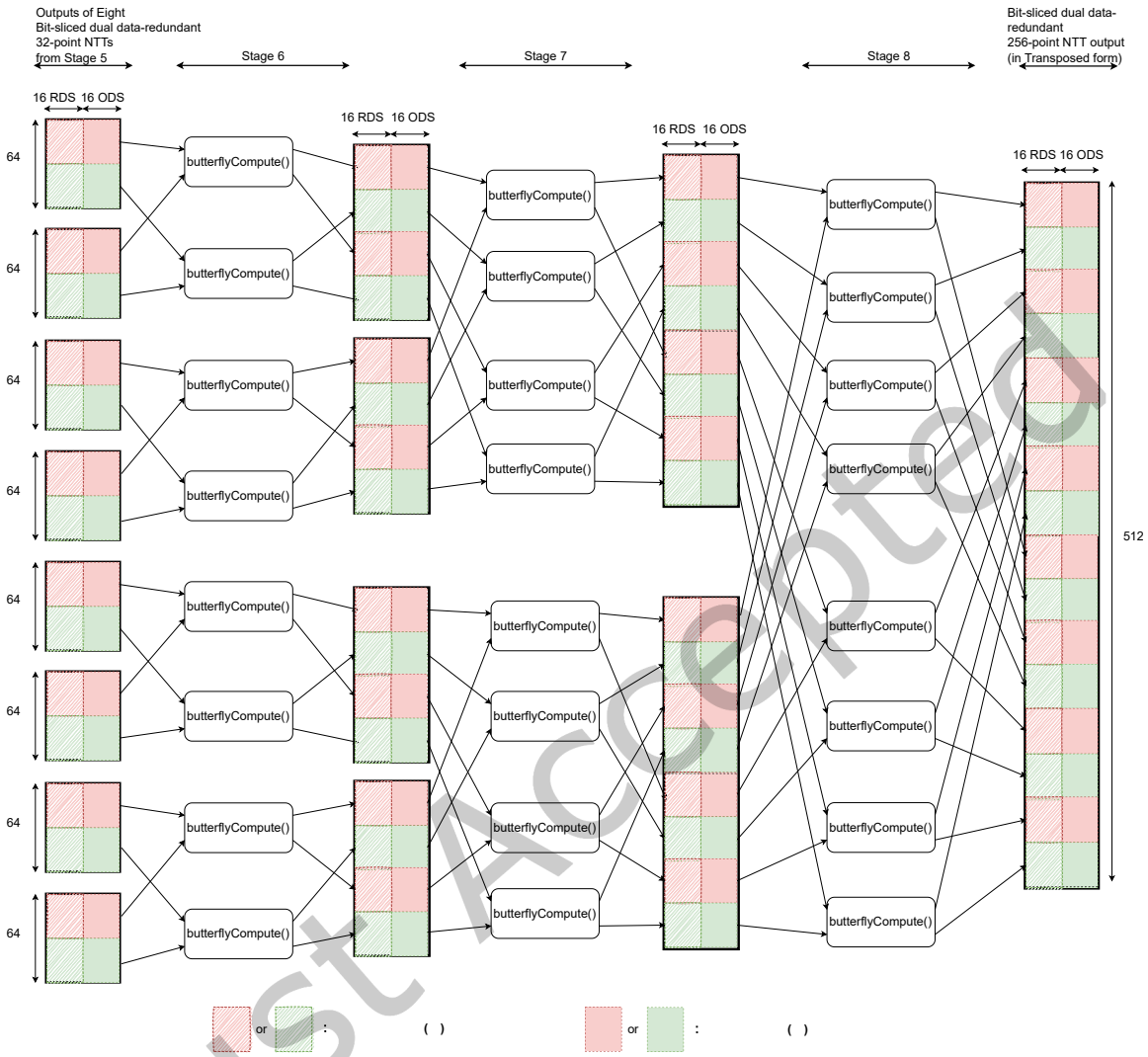


Fig. 7. Data flow diagram in stages 6, 7 and 8 of Bit-sliced 256-point NTT with dual data-redundancy

of positive polarity and 1.5 mm diameter in order to cause disturbance in small part of the device. The probe is connected to the pulse generator and oscilloscope. Oscilloscope is used to measure the trigger and probe coil current signals. As soon as the probe receives a pulse from the pulse generator at its digital glitch input, it discharges the capacitor bank into the coil at the probe tip thereby creating a single EM pulse. The PC controls every part of the setup including configuring the injection parameters and capturing the results for analysis. The control software running on the PC synchronizes its operations with the other components of the setup through serial communication.

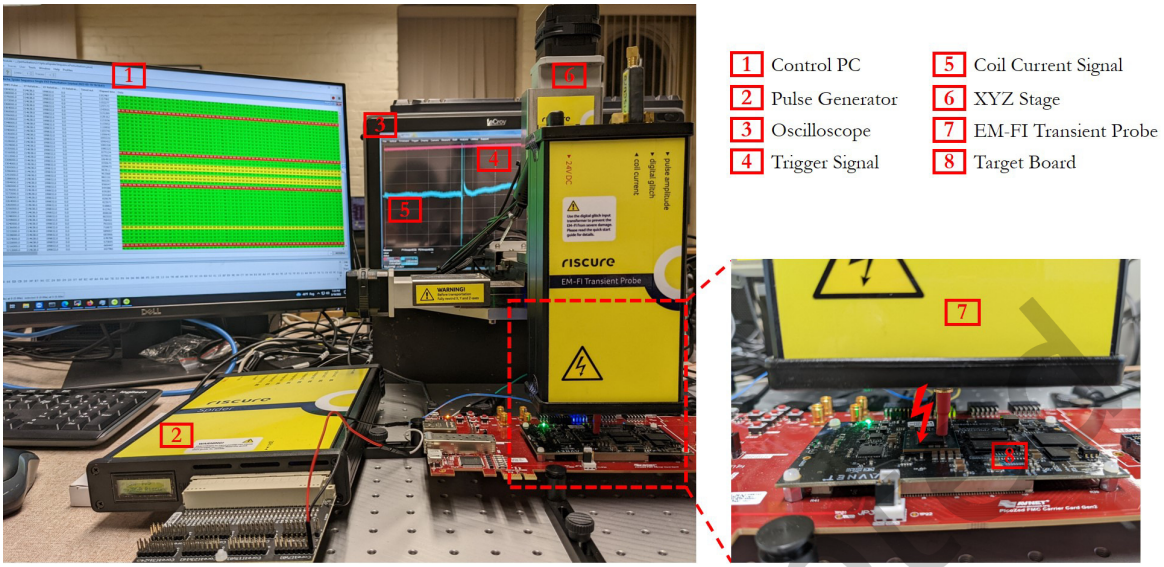


Fig. 8. Overview photo of EMFI setup and close-up photo of the injection coil. The probe tip is positioned at approximately 0.4mm from the top of the processor package.

4.1.2 Device Under Attack: The attack is realized on a Zynq-7000 SoC, XC7Z030SBG485 Flip-Chip Lidless BGA device having 19x19 mm package size. This SoC is embedded on Avnet PicoZed, our target board and it consists of a dual-core 32-bit ARM Cortex-A9 MPCore based processing system (PS) and Xilinx programmable logic (PL) on CMOS 28nm technology implementing the ARMv7-a ISA and runs by default at 667 MHz. The ARM processor has two separate 32 KB L1 caches for instruction and data, 256 KB on-chip memory and shares 512 KB L2 cache with NEON co-processor. The target board has 1 GB external DDR3 Memory and 128 MB QuadSPI Flash.

4.1.3 Software Setup: The test program is booted from external flash in PS Master Boot Mode and is executed from DDR3. We used Xilinx Vitis platform to program flash and for on-chip hardware debugging. We power the target during all the experiments at nominal voltage. A trigger is implemented using a general purpose IO pin of the target. The test program running on the target sets this trigger signal high just before the beginning of attack window and then sets the trigger signal low after the window ends. The test program used for all the experiments is open-source reference C implementation of Dilithium Round 3 [1] version integrated with our countermeasure. This implementation was compiled with the arm-none-eabi-gcc-9.2.0 using compiler flags `-mcpu=cortex-a9 -mfpu=neon-vfpv3 -mfloat-abi=hard -O0`. Protected NTT of s_1 operation in Line 13 of Algorithm 2 in the Appendix A is our point of interest for fault injection in the test program and this target operation execution forms the attack window for all our experiments. For every fault injection, the test program sends a fault response and complete signature at the end of execution to the PC over UART for faults classification.

4.1.4 Experimental Process: We configure four injection parameters in our EMFI setup for getting and increasing the probability of a successful fault injection. Since the full parameter search space is huge to be exhaustively covered, we rather focus on searching the optimal values for the most important parameters and keeping remaining parameters fixed:

[1] https://github.com/pq-crystals/dilithium/tree/master/ref (Commit f1f8085 on Sep 13, 2021)

- **Spatial location -** The x-y position is defined as the 2D-position of injection probe relative to the reference points set on the top surface area of the chip. We keep the Z-position fixed at approximately 0.4mm from the top of the chip.
- **Temporal location -** The amount of time between the trigger is set high and the actual EM pulse injection. It is also referred to as **EMFI Pulse Delay**.
- **Injection voltage -** The Intensity of EM pulse. It configures the maximum voltage over the coil which effects the current induced into the chip. It is also referred to as **EMFI Pulse Power** and this value is a percentage of the highest injection voltage of the probe, which is 475V.
- **Pulse duration -** The amount of time the probe continuously supplies variable coil current. We fixed it to 500 kns.

In this setup, the probe tip coil emits a single EM pulse for a fault injection. Initially, we gather golden signature response from the target running a fault-free execution for a fixed message and seed. Next, for every EM fault injection measurement, the output signature response from the target is compared against the golden signature response. In addition to the output signature, we collect the fault response of the target device. The fault response indicates if the original data slices and their redundant data slices produce the same or a different output at the end of the signature computation. The outcome of the experiment can be classified and grouped into different categories:

- **No Effect -** When the target output matches the expected response and no fault response is detected.
- **Crash -** When the target halts in an exception condition such as Data Abort, Pre-fetch Abort or an Undefined Instruction. For each of these exceptions, there is an exception handler set up with an infinite loop.
- **Faults Not Detected -** When the output signature response is different from the expected response but fault response is not detected.
- **Faults Detected -** When the output signature response is different from the expected response but fault response is detected.

We do a software reset of the target after every measurement so that the result state of the previous measurement does not effect next measurement. After a reset, the target boots from flash and runs the test program automatically. In the crash cases, we do manual external reset of the target. Faulty signatures are those which pass the rejection checks (Lines 16 and 20) when a fault is injected during the attack window in the signing procedure (Algorithm 2 in the Appendix A). When a signature generated in the presence of a fault injection attempt is internally detected by the rejection sampling steps, signature re-generation takes place until the rejection conditions are satisfied. For such cases, we set an upper bound on the number of times signature re-generation can take place in order to complete the measurement of a single fault injection attempt in an experiment. In a real-world setting of the proposed countermeasure, when a fault is detected by our countermeasure, it would not output any faulty signature for secret key analysis by the attacker.

4.2 Performance Analysis

We performed performance hotspots analysis of the proposed countermeasure integrated in Dilithium Algorithm. To perform this measurement on ARM Cortex-A9, we use Xilinx Vitis TCF Profiler. Results of profiling the protected Dilithium algorithm are shown in Table 1. We observe that butterflyCompute() and pointwisemultiplier() bit-sliced functions are the largest contributors to CPU cycles with a CPU usage of 58.6% and 20.4% on ARM Cortex-A9 target platform. These two functions form the core of bit-sliced polynomial multiplications and protects the most time-consuming operations from data faults.

Table 2 shows the performance impact of the proposed bit-slicing based IIR countermeasure on the key operations of Dilithium on ARM Cortex-A9 target platform. We report results for the NIST security level 2 of

Table 1. Percentage of CPU cycles consumed across active functions and number of active functions calls during the combined execution of key generation and signature generation of Protected Dilithium on ARM Cortex-A9 CPU.

Active Functions	CPU Cycles (%)	No. of Calls
butterflyCompute	58.6	16064
pointwisemultiplier	20.4	5360
pointwiseaccumulator	0.26	1024

Table 2. Performance evaluation of bitslicing-based IIR countermeasure for key operations of Dilithium Algorithm on ARM Cortex-A9 @ 667MHz. The cycle counts are median for 10,000 executions. Unprotected Dilithium is the reference C implementation with no added countermeasure.

Dilithium Main Operations	Cycles Count			Slowdown (×)	
	Unprotected	Bitsliced	Protected	Bitsliced	Protected
Forward NTT	28,928	925,824	1,827,584	32.0	63.2
Inverse NTT	30,848	928,128	1,833,152	30.1	59.4
Point-wise Multiplication	4,864	302,144	547,904	62.1	112.6
Full Multiplication	93,888	3,082,496	6,037,888	32.8	64.3
Key Generation	2,231,872	32,030,400	57,659,968	14.4	25.8
Signature Generation	7,150,272	191,078,144	351,530,304	26.7	49.2

Dilithium (Dilithium2). These numbers are median cycle counts for 10,000 executions of Dilithium, including key generation and signing procedures only, as verification operates on public information. Note that hardware performance counters are used to measure cycles count. All the implementations for performance analysis are compiled with `-O3` optimization flag. Both the slowdowns are calculated by dividing by the corresponding unprotected Dilithium metric. For NTT/INTT, slowdown directly comes from the butterflyCompute() and pointwisemultiplier() bit-sliced functions since they involve a large number of data transfers as compared to compute cycles. Their contribution in runtime of Dilithium in terms of number of functions calls is shown in Table 1. It is observed that Dilithium key generation and signing incurs high performance overhead since a significant portion of their runtime is dominated by NTT/INTTs, thereby, leading to large call counts of these bit-sliced functions. From the Table 2, it should be noticed that protected version have almost twice the slowdown compared to the bit-sliced version since the protection is based on dual spatial redundancy.

4.3 Footprint Analysis

Footprint is calculated by measuring the compiled program size. Table 3 shows the footprint results for unprotected, bit-sliced and protected Dilithium implementations. All the implementations for footprint analysis are compiled with `-Os` optimization flag. Increased memory requirements in `.text` memory section for bit-sliced implementation comes from the butterflyCompute(), pointwisemultiplier() and pointwiseaccumulator() bit-sliced functions which occupy 77020 bytes, 82708 bytes and 3672 bytes, respectively, of code segment on ARM Cortex-A9 platform. The code size of a bit-sliced function is directly proportional to the cycles it takes to

Table 3. Footprint evaluation of bitslicing-based IIR countermeasure on ARM Cortex-A9 @ 667MHz target platform. Overhead is calculated by dividing the total code size by the corresponding unprotected implementation.

Dilithium Implementation	Code Size (bytes)				Overhead (×)
	.text	.data	.bss	Total	
Unprotected	48,572	1,144	145,544	195,260	-
Bitsliced	222,488	1,144	159,056	382,688	1.95
Protected	232,064	1,144	176,464	409,672	2.09

Table 4. Evaluation of different bit-sliced functions used in our Bit-sliced Dilithium Implementation on ARM Cortex-A9 @ 667MHz. Overhead is calculated as the number of ldr and str instructions as a percentage of the total

Bit-sliced function	Instruction Mix							Overhead (%)
	ORR	EOR	AND	ADD	SUB	LDR	STR	
butterflyCompute	1830	2957	3994	2	2	5210	3130	48.70
pointwisemultiplier	1931	3270	4331	2	2	5618	3295	48.31
pointwiseaccumulator	51	97	120	13	1	277	194	62.54

execute due to its linear code structure. This is evident by the protection cost of Dilithium implementation when protected with our bit-slicing based countermeasure mentioned in Table 2. While pre-computed twiddle and convolution factors used in bit-sliced polynomial multiplication computations are also part of `.text` section. Bit-sliced implementation also shows an overhead in `.bss` memory section due to uninitialized global intermediate array variables of polynomial size used in bit-sliced NTT/INTT operations. Table 3 shows that the footprint overhead of protected Dilithium implementation on target platform is 2.09 times.

4.4 Overhead Analysis

Table 4 shows our analysis of different bit-sliced functions used in the Bit-sliced Dilithium implementation in terms of instruction breakdown and overhead of data movements. The overhead values reported are calculated as the number of LDR and STR instructions divided by the total number of instructions. We observe the number of ORR, EOR, AND, ADD and SUB instructions involved during the execution of butterflyCompute(), pointwisemultiplier() and pointwiseaccumulator() bit-sliced functions. Furthermore, we observe that moving data from memory to processor is expensive for these functions. The number and composition of logical bit-wise instructions originates directly from the Boolean gate-level netlist generated from the Verilog description. The overhead of LDR and STR instructions, however, is introduced by the compiler due to increased register pressure. There is register pressure because net width in the netlist, i.e., the number of active variables in bit-sliced code exceeds the number of registers available in the underlying hardware. The overhead related to register spilling into memory is about 48-62% in terms of instruction count. This overhead explains the slower performance of the bit-sliced NTT, INTT, Full Multiplication and Matrix-Vector Multiplication operations of Dilithium.

Table 5. Experiment 1: Injection parameters settings

Injection Parameter	Value
x-y Probe Position	Right Upper Quadrant Scan
EMFI Pulse Delay	Random between 0 ns and 10M ns
EMFI Pulse Power	Random between 80% and 100%

5 Fault Countermeasure Evaluation

In this section, we present the experimental fault attack resistance evaluation of the proposed countermeasure. To inject faults, we use Electromagnetic Fault Injection (EMFI) setup as described in Section 4.1. Our goal is to maximize the chances of obtaining a successful fault which gets detected by the proposed countermeasure. For this, determination of injection parameters is an essential step which consists of finding the spatial location, temporal location and injection voltage. We demonstrate the effectiveness of the proposed countermeasure by performing fault injection on protected NTT of s_1, our point of interest in the test program and it is timed using a GPIO-based trigger as explained in Section 4.1. Protected NTT is divided into three phases: transpose, compute and reverse-transpose. As we know that redundant computations take place in the compute phase, therefore, we set compute phase as our attack window and it takes roughly 10M ns.

We first performed full chip scan with a 20 x 20 grid resolution and EMFI Pulse Delay and EMFI Pulse Power were randomly set to values between 0 ns & 10M ns and 70% & 100%, respectively, for each measurement. We observed through this initial trial that successful faults are obtained in the right upper quadrant of the chip which also correlates to an area containing target Processing System (PS). Also, there were no successful faults when EMFI Pulse Power is less than 80%. With this partial identification of successful faults distribution, we perform three experiments to fine-tune our injection parameters in a top-down approach:

5.1 Experiment 1: Randomly-chosen EMFI Pulse Power and EMFI Pulse Delay from an estimated range with a step-wise right upper quadrant chip scan

To determine the best probe x-y position within right upper quadrant of the chip, we performed scan in a 10 x 10 measurement grid and injected 30 faults per probe position, this led to 3000 measurements in total. The configuration of injection parameters for this experiment are summarized in the Table 5. Table 6 lists the outcome for this surface area exploration while injecting into the test program in terms of percentage faults in each category. Areas where no effect of fault injection is seen are shown in the Fig. 9 while positions which are sensitive to fault injection where faults are either resulting in a crash, faults being detected and faults not detected are shown in the Fig. 10a. We can also observe the spatial occurrence of individual fault categories where an effect of fault injection is visible in the following figures Fig. 10b, 10c and Fig. 10d.

We can identify the best probe x-position as 214638 since 5 out 7 faults detected occur at the same x-position while best y-position is 199832 by taking the average of y co-ordinates of faults detected that fall in the top-half area. This means the probe x-y position can be fixed to decrease the spatial location parameter search space, significantly increasing the faults detection rate. We can conclude from Fig. 11 that there are more faults detected when injection is performed between 0 to 5M ns duration of attack window and EMFI Pulse Power is less than 90%. At higher pulse intensities, more crash behavior is observed while lower pulse intensities does not cause enough voltage variations in the device to induce computational faults.

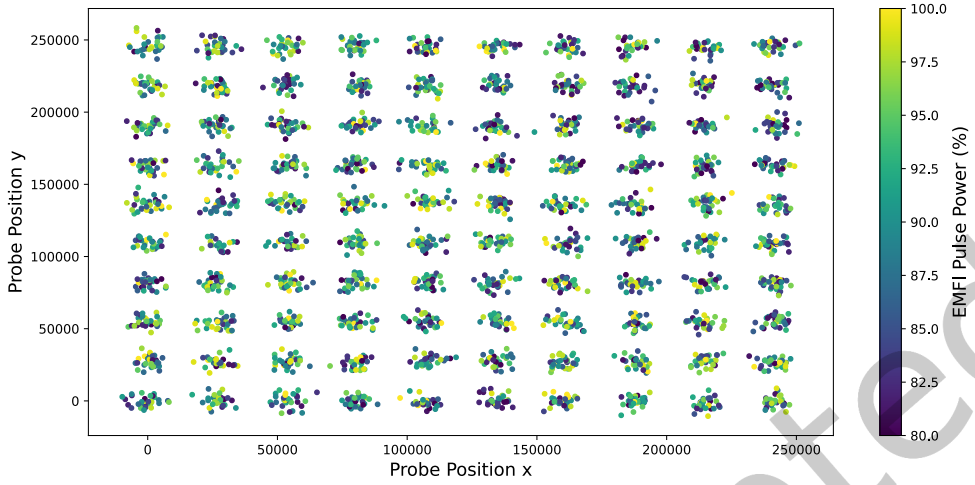


Fig. 9. Experiment 1: Probe positions over the chip leading to "No Effect" fault cases. Note that it is a jittered scatter plot to prevent overlapping dots at the same position.

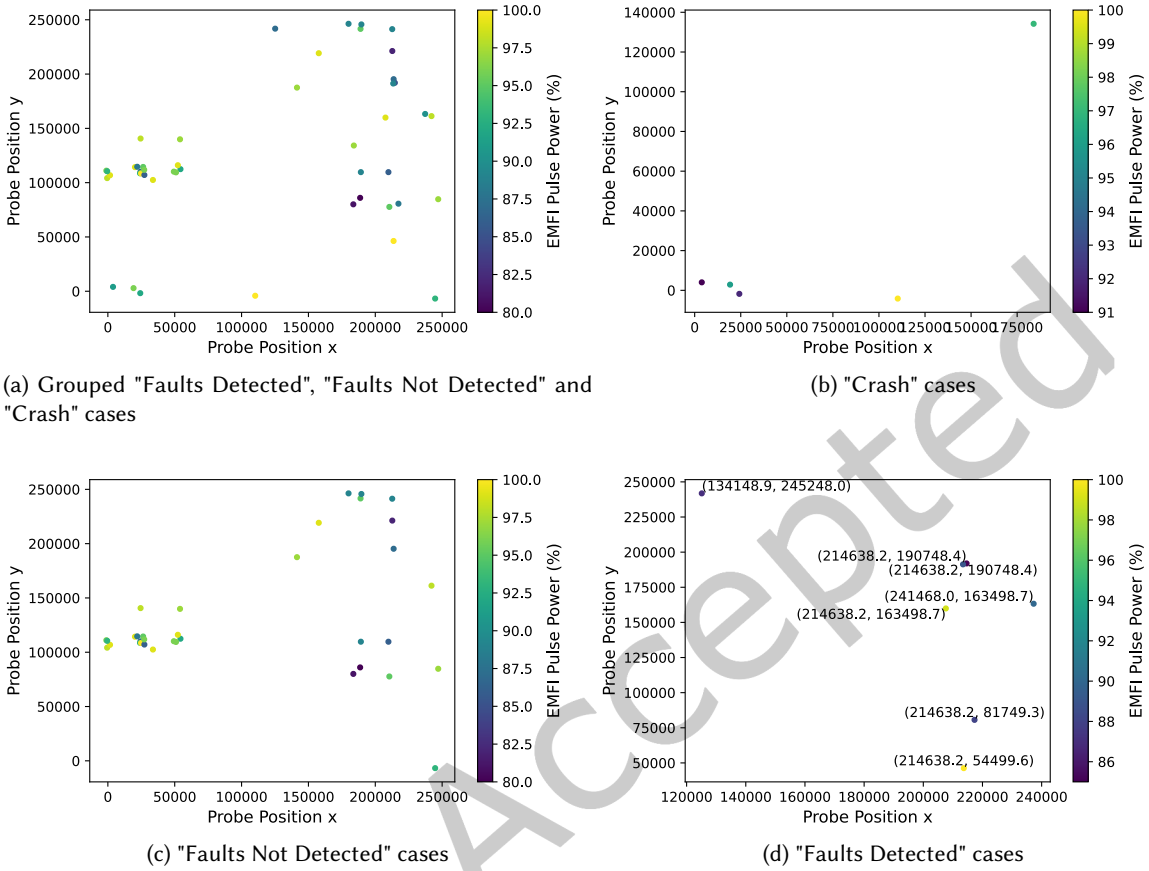
Table 6. Experiment 1: Percentage occurrence of different fault categories

Classification	Amount	Percentage (%)
No Effect	2953	98.4333
Crash	35	1.16667
Faults Not Detected	5	0.16666
Faults Detected	7	0.23333

5.2 Experiment 2: Randomly-chosen EMFI Pulse Power and EMFI Pulse Delay within an identified range with a fixed x-y probe position

This experiment is performed at a fixed spatial location determined through Experiment 5.1. Also, broad EMFI delay and intensity spectrum has been reduced from previous experiment and those settings are summarized in the Table 7. Using these parameters, we investigate the relationship between the EMFI Pulse Delay and EMFI Pulse Power to find an optimal parameter range in order to enhance the probability of detected successful faults. We perform total 6000 measurements in 21 hours and the results of this experiment are shown in Fig. 12 and Table 8.

The plot shows that a lot of faults detected are spread out in a time window of 3M ns to 3.8M ns after the trigger. Also, we factorize the faults into two categories: potentially exploitable and non-exploitable. Potentially exploitable faults are usable for an attacker if the test program executes completely but computes a non-zero faulty signature whereas zero faulty signatures and correct signatures are non-exploitable. Crashes are faults aborting the normal execution of test program hence are not usable for an attack. Note that we get few cases



(a) Grouped "Faults Detected", "Faults Not Detected" and "Crash" cases

(b) "Crash" cases

(c) "Faults Not Detected" cases

(d) "Faults Detected" cases

Fig. 10. Experiment 1: Probe positions over the chip leading to fault cases which change the test program's intended behavior. Note that it is a jittered scatter plot to prevent overlapping dots at the same position.

where faults get detected but the signature response is as expected. These are also counted as non-exploitable faults being detected. We are able to find an easily glitchable area in the plot when EMFI Pulse Power is between 83% and 90% where we can spot clustering of fault detected red crosses. We can observe from the plot that there is a high correlation between the sensitive area for crashes and the sensitive area for potentially exploitable faults. To conclude, this experiment shows that the percentage of potentially exploitable faults that are data faults is 63.77%.

5.3 Experiment 3: Parametric-sweep of EMFI Pulse Power and EMFI Pulse Delay with a fixed x-y probe position

In this experiment, we perform parameter sweep between the narrowed-down ranges identified through Experiment 5.2. Using the parameters given in Table 9, we perform 3 measurements per step that leads to total 4824

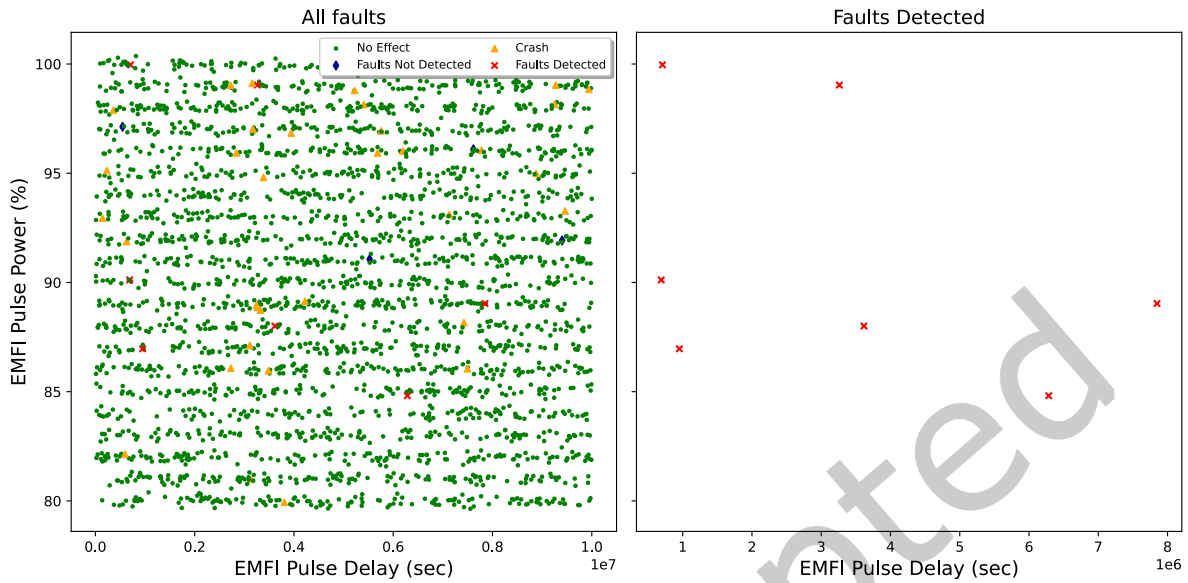


Fig. 11. Experiment 1: Relation between EMFI Pulse Delay and EMFI Pulse Power for faults of different categories and only "Faults Detected" category is shown in the left and right plot, respectively.

Table 7. Experiment 2: Injection parameters settings

Injection Parameter	Value
x-y Probe Position	Fixed at (214638, 199832)
EMFI Pulse Delay	Random between 0 ns and 5M ns
EMFI Pulse Power	Random between 80% and 90%

Table 8. Experiment 2: Percentage occurrence of different fault categories and their breakup into potentially exploitable faults and non-exploitable faults

Classification	Amount	Percentage (%)	Potentially Exploitable	Non-Exploitable
No Effect	5635	93.9157	0	5635
Crash	229	3.81667	0	229
Faults Not Detected	49	0.816667	46	3
Faults Detected	87	1.45	81	6

measurements completed in 16 hours. The results of this experiment are shown in the Fig. 13 and Table 10 further shows the distribution of fault response.

It is observed that the number of potentially exploitable faults has increased from 127 in Experiment 5.2 to 382 in this experiment. This result further confirms that the spatial parameter search process led to a precise

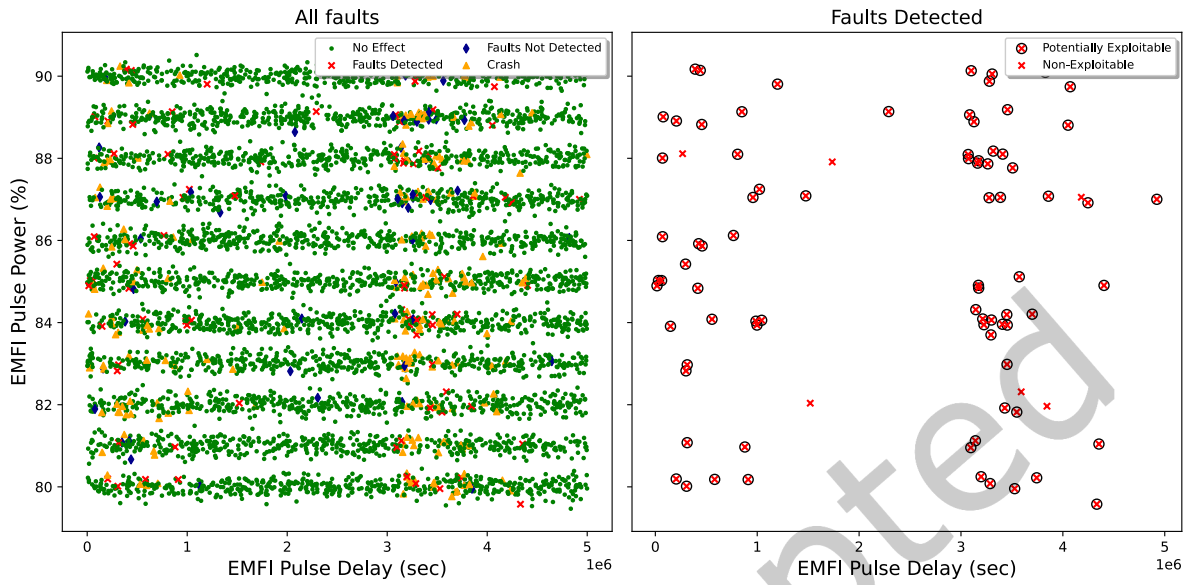


Fig. 12. Experiment 2: Relation between EMFI Pulse Delay and EMFI Pulse Power for faults of different categories and only "Faults Detected" category is shown in the left and right plot, respectively.

Table 9. Experiment 3: Injection parameters settings

Injection Parameter	Value
x-y Probe Position	Fixed at (214638, 199832)
EMFI Pulse Delay	Sweep between 3M ns and 3.8M ns by steps of 4000 ns
EMFI Pulse Power	Sweep between 83% and 90% by steps of 1%

location on the chip where we are able to achieve large number of attacker usable faults. The plot shows the same observation as in previous experiments that the attacker usable faults occur at similar locations or locations closer to crashes. In conclusion, our approach to injection parameters exploration is able to find reliable and highly repeatable potentially exploitable faults which affect the datapath are 61.7% while the remaining are control flow faults or memory faults.

6 Conclusions

We conclude that hardening the polynomial multiplication operations in Dilithium with the proposed countermeasure significantly reduced the probability of a successful fault injection attack and increased the time required to gain enough faulty signatures to mount a complete attack at the cost of increased computational complexity. FI attack surface for Dilithium is so large that any part of code may become a potential FI attack vector. Our countermeasure reduces the attack surface by protecting polynomial multiplication operations, second largest contributor to the run-time of the algorithm after hash operations. The proposed countermeasure estimates the percentage of potentially exploitable faults which affect the datapath as 62%. However, some exploitable faults are not detected because time redundancy is not implemented in the proposed countermeasure to detect control

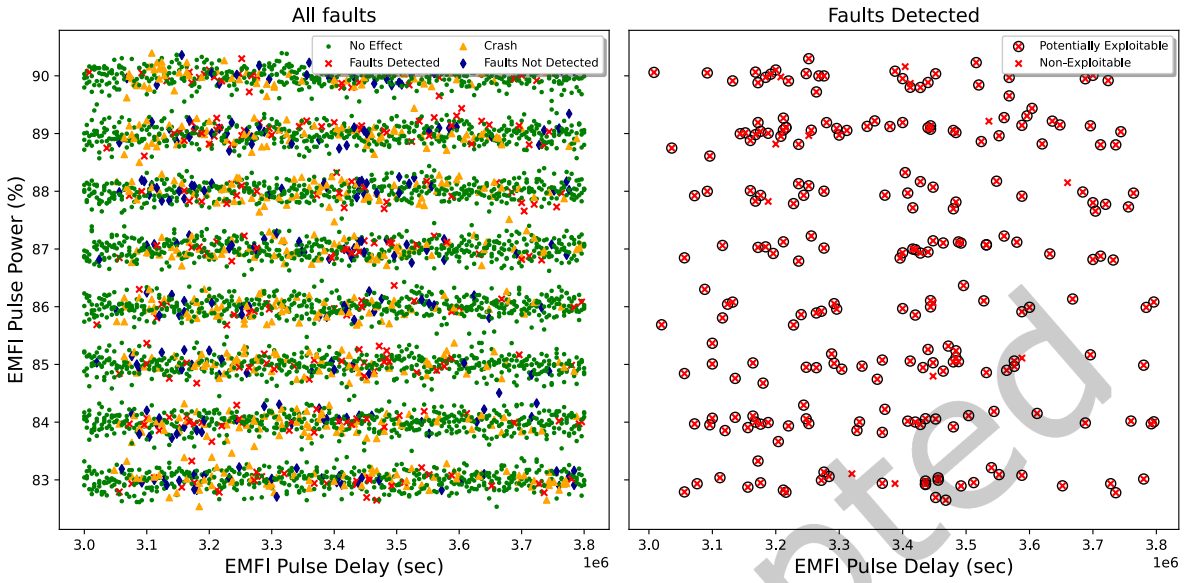


Fig. 13. Experiment 3: Relation between EMFI Pulse Delay and EMFI Pulse Power for faults of different categories and only "Faults Detected" category is shown in the left and right plot, respectively.

Table 10. Experiment 3: Percentage occurrence of different fault categories and their breakup into potentially exploitable faults and non-exploitable faults

Classification	Amount	Percentage (%)	Potentially Exploitable	Non-Exploitable
No Effect	3762	78	0	3762
Crash	583	12	0	583
Faults Not Detected	230	4.77	146	84
Faults Detected	249	5.23	236	13

flow faults. Further, characterization of fault effects is required to better understand the behavior of undetected potentially exploitable faults at software level. We plan to extend our bit-sliced NTT design with temporal IIR to detect faults in the control flow. Also, we intend to investigate the fault attack resistance of other finalist lattice-based schemes with our generic countermeasure.

References

[1] Alexandre Adomnicai and Thomas Peyrin. 2021. Fixslicing AES-like Ciphers New bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 1 (2021), 402–425. https://doi.org/10.46586/tches.v2021.i1.402-425

[2] Anita Aghaie, Amir Moradi, Shahram Rasoolzadeh, Aein Rezaei Shahmirzadi, Falk Schellenberg, and Tobias Schneider. 2020. Impeccable Circuits. *IEEE Trans. Computers* 69, 3 (2020), 361–376. https://doi.org/10.1109/TC.2019.2948617

[3] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. 2010. When Clocks Fail: On Critical Paths and Clock Faults. In *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau,*

Germany, April 14-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6035), Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny (Eds.). Springer, 182–193. https://doi.org/10.1007/978-3-642-12510-2_13

[4] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Angela Robinson Ray Perlner and, Daniel Smith-Tone, and Yi-Kai Liu. 2022. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process.

[5] Sabine Azzi, Bruno Barras, Maria Christofi, and David Vigilant. 2017. Using linear codes as a fault countermeasure for nonlinear operations: application to AES and formal verification. *J. Cryptogr. Eng.* 7, 1 (2017), 75–85. https://doi.org/10.1007/s13389-016-0138-1

[6] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. 2019. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019, 4 (Aug. 2019), 17–61. https://doi.org/10.13154/tches.v2019.i4.17-61

[7] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. 2017. GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10529)*, Wieland Fischer and Naofumi Homma (Eds.). Springer, 321–345. https://doi.org/10.1007/978-3-319-66787-4_16

[8] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE* 94, 2 (2006), 370–382. https://doi.org/10.1109/JPROC.2005.862424

[9] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. 2010. Countermeasures against Fault Attacks on Software Implemented AES: Effectiveness and Cost. In *Proceedings of the 5th Workshop on Embedded Systems Security* (Scottsdale, Arizona) *(WESS '10)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/1873548.1873555

[10] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings (Lecture Notes in Computer Science, Vol. 263)*. Springer, 311–323. https://doi.org/10.1007/3-540-47721-7_24

[11] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. 2015. McBits: fast constant-time code-based cryptography. *IACR Cryptol. ePrint Arch.* (2015), 610. http://eprint.iacr.org/2015/610

[12] Eli Biham. 1997. A fast new DES implementation in software. In *Fast Software Encryption (FSE)*. https://doi.org/10.1007/BFb0052352

[13] Eli Biham and Adi Shamir. 1997. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1294)*, Burton S. Kaliski Jr. (Ed.). Springer, 513–525. https://doi.org/10.1007/BFb0052259

[14] Nina Bindel, Johannes Buchmann, and Juliane Krämer. 2016. Lattice-based signature schemes and their sensitivity to fault attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 63–77.

[15] Leon Groot Bruinderink and Peter Pessl. 2018. Differential fault attacks on deterministic lattice signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 21–43.

[16] Mathieu Ciet and Marc Joye. 2005. Practical Fault Countermeasures for Chinese Remaindering Based RSA (Extended Abstract).

[17] Ang Cui and Rick Housley. 2017. BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/woot17/workshop-program/presentation/cui

[18] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. 2012. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, September 9, 2012*, Guido Bertoni and Benedikt Gierlichs (Eds.). IEEE Computer Society, 7–15. https://doi.org/10.1109/FDTC.2012.15

[19] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. 2012. Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 7–15. https://doi.org/10.1109/FDTC.2012.15

[20] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 238–268.

[21] Mahmoud A. Elmohr, Haohao Liao, and Catherine H. Gebotys. 2020. EM Fault Injection on ARM and RISC-V. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*. 206–212. https://doi.org/10.1109/ISQED48828.2020.9137051

[22] Clément Gaine, Driss Aboulkassimi, Simon Pontié, Jean-Pierre Nikolovski, and Jean-Max Dutertre. 2020. Electromagnetic Fault Injection as a New Forensic Approach for SoCs. In *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*. 1–6. https://doi.org/10.1109/WIFS49906.2020.9360902

[23] Michael Hutter and Jörn-Marc Schmidt. 2014. The Temperature Side Channel and Heating Fault Attacks. *IACR Cryptol. ePrint Arch.* (2014), 190. http://eprint.iacr.org/2014/190

[24] Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. *IACR Cryptol. ePrint Arch.* (2009), 129. http://eprint.iacr.org/2009/129

[25] Pantea Kiaei, Tom Conroy, and Patrick Schaumont. 2021. Architecture Support for Bitslicing. *IACR Cryptol. ePrint Arch.* (2021), 1236. https://eprint.iacr.org/2021/1236

[26] Pantea Kiaei and Patrick Schaumont. 2021. Synthesis of Parallel Synchronous Software. *IEEE Embed. Syst. Lett.* 13, 1 (2021), 17–20. https://doi.org/10.1109/LES.2020.2992051

[27] Pantea Kiaei and Patrick Schaumont. 2021. Synthesis of Parallel Synchronous Software. *IEEE Embedded Systems Letters* 13, 1 (2021), 17–20. https://doi.org/10.1109/LES.2020.2992051

[28] Youngbeom Kim, Taek-Young Youn, Jingyo Song, and Seog Chung Seo. 2022. Crystals-Dilithium on ARMv8. *Security and Communication Networks* 2022 (2022). https://doi.org/10.1155/2022/5226390

[29] Robert Könighofer. 2008. A Fast and Cache-Timing Resistant Implementation of the AES. In *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4964)*, Tal Malkin (Ed.). Springer, 187–202. https://doi.org/10.1007/978-3-540-79263-5_12

[30] Alexandre Menu, Shivam Bhasin, Jean-Max Dutertre, Jean-Baptiste Rigaud, and Jean-Luc Danger. 2019. Precise Spatio-Temporal Electromagnetic Fault Injections on Data Transfers. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 1–8. https://doi.org/10.1109/FDTC.2019.00009

[31] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. 2013. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 77–88. https://doi.org/10.1109/FDTC.2013.9

[32] Koksal Mus, Saad Islam, and Berk Sunar. 2020. QuantumHammer: A Practical Hybrid Attack on the LUOV Signature Scheme. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1071–1084.

[33] Onur Mutlu and Jeremie S. Kim. 2020. RowHammer: A Retrospective. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39, 8 (2020), 1555–1571. https://doi.org/10.1109/TCAD.2019.2915318

[34] Sébastien Ordas, Ludovic Guillaume-Sage, and Philippe Maurine. 2017. Electromagnetic fault injection: the curse of flip-flops. *J. Cryptogr. Eng.* 7, 3 (2017), 183–197. https://doi.org/10.1007/s13389-016-0128-3

[35] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty, and Patrick Schaumont. 2016. Lightweight Fault Attack Resistance in Software Using Intra-instruction Redundancy. In *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10532)*, Roberto Avanzi and Howard M. Heys (Eds.). Springer, 231–244. https://doi.org/10.1007/978-3-319-69453-5_13

[36] Prasanna Ravi, Mahabir Prasad Jhanwar, James Howe, Anupam Chattopadhyay, and Shivam Bhasin. 2019. Exploiting determinism in lattice-based signatures: practical fault attacks on pqm4 implementations of NIST candidates. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 427–440.

[37] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. 2019. Number "not used" once-practical fault attack on pqm4 implementations of NIST candidates. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 232–250.

[38] Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. 2021. Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice. *IACR Cryptol. ePrint Arch.* (2021), 296. https://eprint.iacr.org/2021/296

[39] Riscure. 2001. *Inspector FI.* Retrieved March 25, 2022 from https://www.riscure.com/security-tools/inspector-fi

[40] Lionel Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. 2015. High precision fault injections on the instruction cache of ARMv7-M architectures. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 62–67. https://doi.org/10.1109/HST.2015.7140238

[41] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. 2013. Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, August 20, 2013*, Wieland Fischer and Jörn-Marc Schmidt (Eds.). IEEE Computer Society, 89–98. https://doi.org/10.1109/FDTC.2013.17

[42] Jörn-Marc Schmidt and Michael Hutter. 2007. Optical and EM Fault-Attacks on CRT-based RSA: Concrete Results. In *Austrochip 2007, 15th Austrian Workhop on Microelectronics, 11 October 2007, Graz, Austria, Proceedings*. Verlag der Technischen Universität Graz, 61–67. Austrochip 2007 ; Conference date: 11-10-2007 Through 11-10-2007.

[43] Jörn-Marc Schmidt and Marcel Medwed. 2012. *Countermeasures for Symmetric Key Ciphers* (1 ed.). Springer, 73–88.

[44] Peter W. Shor. 1994. Polynomial time algorithms for discrete logarithms and factoring on a quantum computer. In *Algorithmic Number Theory*, Leonard M. Adleman and Ming-Deh Huang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 289–289.

[45] Richa Singh, Thomas Conroy, and Patrick Schaumont. 2020. Variable Precision Multiplication for Software-Based Neural Networks. In *2020 IEEE High Performance Extreme Computing Conference, HPEC 2020, Waltham, MA, USA, September 22-24, 2020*. IEEE, 1–7. https://doi.org/10.1109/HPEC43674.2020.9286170

[46] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1057–1074. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[47] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. 2021. Fault-Injection Attacks against NIST's Post-Quantum Cryptography Round 3 KEM Candidates. Springer-Verlag. https://doi.org/10.1007/978-3-030-92075-3_2

[48] Neng Zhang, Bohan Yang, Chen Chen, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. Highly Efficient Architecture of NewHope-NIST on FPGA using Low-Complexity NTT/INTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 2 (Mar. 2020), 49–72. https://doi.org/10.13154/tches.v2020.i2.49-72

[49] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. 2013. Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS), Chania, Crete, Greece, July 8-10, 2013*. IEEE, 110–115. https://doi.org/10.1109/IOLTS.2013.6604060

A Proposed FA Countermeasure Protected Polynomial Multiplication Instances in Dilithium Signing and Key Generation Algorithms

Algorithm 2 Polynomial multiplications within the Dilithium Signature Generation [20] algorithm which are protected using the bit-slicing based FA countermeasures are highlighted in blue.

Input: sk - Secret Key, M - Message
Output: σ - Signature
$\mathbf{A} \in R_q^{k \times l} \leftarrow ExpandA(\rho)$
$\mu \in \{0,1\}^{384} \leftarrow CRH(tr \parallel M)$
$\kappa \leftarrow 0, (z,h) \leftarrow \perp$
$\rho' \in \{0,1\}^{384} \leftarrow CRH(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{384}$ randomized)
while $(z,h) = \perp$ **do**
  $y \in S_{\gamma 1}^{l} \leftarrow ExpandMask(\rho', \kappa)$
  $w \leftarrow \mathbf{A}y$ ▷ $\mathbf{A}y$ is protected
  $w_1 \leftarrow HighBits_q(w, 2\gamma_2)$
  $\tilde{c} \in \{0,1\}^{256} \leftarrow H(\mu \parallel w_1)$
  $c \in B_\tau \leftarrow SampleInBall(\tilde{c})$
  $z \leftarrow y + c.s_1$ ▷ $c.s_1$ is protected
  $r_0 \leftarrow LowBits_q(w - c.s_2, 2\gamma_2)$ ▷ $c.s_2$ is protected
  if $\|z\| \geq \gamma_1 - \beta$ or $\|r_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(z,h) \leftarrow \perp$
  else
    $h \leftarrow MakeHint_q(-c.t_0, w - c.s_2 + c.t_0, 2\gamma_2)$ ▷ $c.t_0$ and $c.s_2$ are protected
    if $\|c.t_0\|_\infty \geq \gamma_2$ or the # of 1's in $h > \omega$ **then** ▷ $c.t_0$ is protected
      $(z,h) \leftarrow \perp$
    end if
  end if
  $\kappa \leftarrow \kappa + l$
end while
return $\sigma = (z, h, \tilde{c})$

Algorithm 3 Polynomial multiplication within the Dilithium Key Generation [20] algorithm which are protected using the bit-slicing based FA countermeasures are highlighted in blue.

1: **Output:** pk - Public Key, sk - Secret Key
2: $\zeta \leftarrow \{0,1\}^{256}$
3: $(\rho, \varsigma, K) \in \{0,1\}^{256\times 3} \leftarrow H(\zeta)$
4: $(s_1, s_2) \in S_\eta^l \times S_\eta^k \leftarrow H(\varsigma)$
5: $A \in R_q^{k\times l} \leftarrow ExpandA(\rho)$
6: $t \leftarrow As_1 + s_2$ ▷ As_1 is protected
7: $(t_1, t_0) \leftarrow Power2Round_q(t, d)$
8: $tr \in \{0,1\}^{384} \leftarrow CRH(\rho \| t_1)$
9: **return** $(pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0))$