Lightweight Unified SHA-3/SHAKE Architecture with a Fault-Resilient State

CHRISTIAN EWERT[1], AMRIT SHARMA POUDEL[1], MOUADH AYACHE[12], ANDRIJA NESKOVIC[1], RAINER BUCHTY[1], MLADEN BEREKOVIC[1], SEBASTIAN BERNDT[3], AND SALEH MULHEM[1],

[1]*Institute of Computer Engineering, Universität zu Lübeck, Lübeck, , Germany*

[2]*Synopsys GmbH, Munich, Germany*

[3]*Department of Electrical Engineering and Computer Science, Technische Hochschule Lübeck, Lübeck, Germany*

Hash functions have become a key part of standard Post-quantum cryptography (PQC) schemes, especially SHA-3 and SHAKE, calling for lightweight implementation. A fault-resilient design is always desirable to make the whole PQC system reliable. We, therefore, propose a) a unified hash engine supporting SHA-3 and SHAKE that follows a byte-wise in-place partitioning mechanism of the so-called KECCAK state, and b) an according fault detection for KECCAK state protection exploiting its cube structure by deploying two-dimensional parity checks. It outperforms the state-of-the-art (SoA) regarding area requirements at competitive register-level fault detection by achieving 100% detection of three and still near 100% of higher numbers of KECCAK state faults. Unlike SoA solutions, the proposed unified hash engine covers all standard hash configurations. Moreover, the introduced multidimensional cross-parity check mechanism achieves a 3.7× improvement in area overhead, with an overall 4.5× smaller fault-resilient engine design as demonstrated in ASIC and FPGA implementations. Integrated into a RISC-V environment, the unified hash engine with the integrated fault-resilient mechanism introduced less than 8 % area overhead. Our approach thus provides a robust and lightweight fault-detection solution for protecting hash functions deployed in resource-constrained PQC applications.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: KECCAK, SHA-3, SHAKE, PQC, Fault-Detection, Fault-Resilience

ACM Reference Format:
Christian Ewert[1], Amrit Sharma Poudel[1], Mouadh Ayache[12], Andrija Neskovic[1], Rainer Buchty[1], Mladen Berekovic[1], Sebastian Berndt[3], and Saleh Mulhem[1]. 2018. Lightweight Unified SHA-3/SHAKE Architecture with a Fault-Resilient State. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Over the years, KECCAK (cryptographic hash family) has gained popularity, especially its deployment in the design of SHA-3 and SHAKE standard hash functions. The US National Institute of Standards and Technology (NIST) has recently

Author's Contact Information: Christian Ewert[1], Amrit Sharma Poudel[1], Mouadh Ayache[12], Andrija Neskovic[1], Rainer Buchty[1], Mladen Berekovic[1], Sebastian Berndt[3], and Saleh Mulhem[1]
[1]*Institute of Computer Engineering, Universität zu Lübeck, Lübeck,, Germany*
[2]*Synopsys GmbH, Munich, Germany*
[3]*Department of Electrical Engineering and Computer Science, Technische Hochschule Lübeck, Lübeck, Germany.*

Manuscript submitted to ACM

announced post-quantum cryptography (PQC) standard algorithms [30–32]; SHAKE and SHA-3 play a crucial role in this standard [31, 32]. For instance, SHAKE is employed in the PQC standard called Stateless Hash-Based Digital Signature [32], while SHA-3 is a part of Module-Lattice-Based Key-Encapsulation Mechanism Standard [31]. Also other PQC algorithms deploy a KECCAK-based hash function in their schemes, such as NTRUEncrypt [17], Rainbow [7], or SPHINCS$^+$ [6]. Note that in the following sections, we generally use KECCAK as illustrated in Fig. 1. The bit sizes of the so-called rate r and capacity c with the message (input) padding mechanism determine if KECCAK configuration serves as SHA-3 or SHAKE. Therefore, we only explicitly use SHA-3 and SHAKE when needed.

1.1 Related Work

1.1.1 Unified Cryptographic Engine. It has always been desirable to design a unified unit or engine of cryptographic primitives [27] [22]. Such a unified engine can serve several modes and has become very important for PQC [2]. For instance, HMAC-Hash unit was proposed in [22] covering six cryptographic hashes, namely: MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160 implemented in parallel. The HMAC-Hash unit design deploys some bit controls to select whether the HMAC algorithm or hash function is to be the only one executed. In[3], a compact 8-bit unified coprocessor for AES and Grøstl hash function was designed for resource-constrained embedded systems. This coprocessor considers three AES modes with 128-, 192-, and 256-bit encryption keys and two Grøstl hash modes with 256- and 512-bit message digests. Similarly, the three AES modes were unified with the ECHO hash function as one coprocessor [9]. A unified AES and SHA-3 engine was proposed in [21]. This engine relies on a unified XOR section that executes key whitening in AES and SHA-3 transformations. In the content of PQC, several unified hash architectures for SHA and SHAKE have been proposed to accelerate different PQC algorithms. The key idea of the implementation relies on in-place partitioning of KECCAK state to serve several hash modes. The in-place partitioning mechanism is performed based on 64-bit data word size. In [37], a unified hash architecture for SHA-3-256, SHA-3-512, and SHAKE-128 was proposed to be a part of the SABER PQC algorithm. Similarly, a unified hash architecture for SHA-3-512, SHAKE-128, and SHAKE-256 was proposed in [44] to accelerate Dilithium and Kyber PQC algorithms on a field-programmable gate array (FPGA) board. This unified implementation also uses a 64-bit data word in-place partitioning of KECCAK state. Therefore, such a wide in-place partitioning mechanism makes SA unified architectures very costly in hardware and unsuitable for resource-constrained devices.

Furthermore, a reliable and secure implementation of a unified engine for cryptographic primitives has emerged as a new design prescriptive. In the case of fault injection, a fault-resilient unified engine for cryptographic primitives is

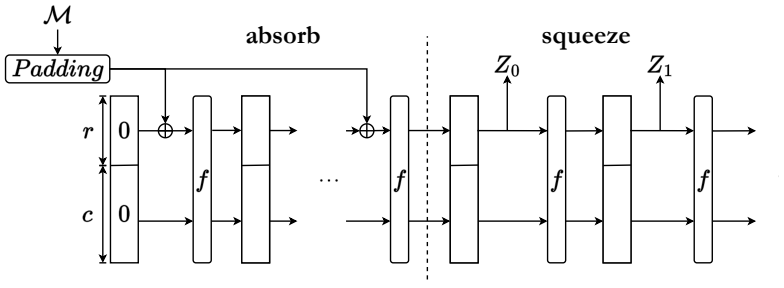

Fig. 1. Representation of KECCAK Sponge construction [28]. f denotes KECCAK-f[1600]

Table 1. State of the Art Overview

Paper	Protection Level	Protection Mechanism	Hardware Overhead
[19]	Logic	Fault Detection	High
[5]	Logic	Error Detection	Mid
[26]	Logic	Error Detection	High
[25]	Logic	Error Detection	High
[35]	Logic	Fault Detection	High
[24]	Logic	Error Detection	Mid - High
[41]	Register & Logic	Error Correction	High
[13]	Logic	Error Detection	High
This work	Register	Fault Detection	Low

highly desirable. However, few unified engines consider this design prescriptive (e.g., [46]). Thus, there is still a huge room for innovation and improvement.

1.1.2 Fault Resilient SHA-3 and SHAKE Engine. In the following, we quickly review the state-of-the-art (SoA) fault-resilient mechanisms of SHA-3 and SHAKE engines. In [36], faults are divided into two classes based on their causes: (i) intentional malicious faults covered in the security domain and (ii) random faults covered in the reliability domain. A fault-resilient design increases the overall dependability of the unified engine. However, fault-resilient mechanisms can protect the desired engine entirely or partially based on the targeted application [36]. Here, we focus on the designs of fault resilient SHA-3 and SHAKE, that can handle the faults and their propagation independently from the causes. For example, Differential Fault Analysis (DFA) [4, 10, 23] was performed on complete 24 KECCAK rounds. Such an analysis shows that DFA can potentially reveal the whole KECCAK state. DFA is known to be a powerful method to compromise many cryptographic primitives like DES, 3-DES, IDEA [10], AES [14] and RSA [8]. In [4], DFA was introduced as a random single-bit fault injected into the internal state of KECCAK at the beginning of the penultimate (22^{nd}) round. In [23], the analysis of DFA on SHA-3 described in [4] was extended to DFA on a byte-granular level in a similar fashion to [4]. The injected faults randomly alter a state byte, leading to faulty and non-faulty hashes of the same message. In contrast to the single-bit fault model, it applies to all four modes of SHA-3. This method can effectively identify the injected single-byte fault and recover the whole internal state of KECCAK. Therefore, several fault-resilient approaches have been proposed for SHA-3 and SHAKE engines.

In the SoA, several works have been carried out and proposed a fault resilient KECCAK. Table 1 shows an overview of SoA fault resilient KECCAK countermeasures. We mainly categorize in *fault detection*, *error detection*, and *error correction*. All of the mentioned detection approaches focus on the protection of KECCAK layers at the logic level [5, 13, 19, 24, 26, 35]. Only one error-correction approach also considers register-level protection [41], providing a holistic protection mechanism for KECCAK. The protection mechanisms mentioned at the logic level may give strong security guarantees but exhibit mid- to high hardware overhead, making them unsuitable for resource-constrained devices.

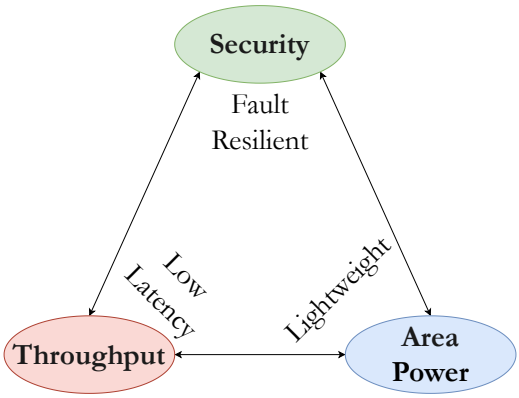

Fig. 2. Trade-offs in Cryptographic Hardware Design

1.2 Motivation & Contributions

Besides the classical digital signatures, data integrity, and several forms of authentication, SHA-3 and SHAKE are deployed in several parts of PQC algorithms [31, 32]. Therefore, a unified architecture of SHA-3 and SHAKE can simultaneously serve in modern and post-cryptographic systems. This fact first motivates us to design such a unified engine. The SoA requirements for the design vary based on the application domain. Therefore, our second motivation is to follow the design principle of lightweight cryptography [29] and choose the design objectives that focus on the hardware cost, low latency, and fault resiliency as shown in Fig. 2. The paper's motivations can be summarized as follows:

- A unified architecture of SHA-3 and SHAKE covering all their standard modes [28]: SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, SHAKE128, and SHAKE-256.
- The proposed unified engine is fault-resilient at minimum hardware overhead. In particular, the proposed fault-detection mechanism meets the requirements of lightweight cryptography: We focus on register-level protection that provides a low-overhead fault countermeasure of KECCAK.

SHA-3 and SHAKE have a huge KECCAK state of 1600 bits. This state has a dedicated configuration of its parameters (rate r and capacity c) to serve as SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, SHAKE-128, and SHAKE-256. This is exactly the main design challenge of a unified SHA-3/SHAKE engine. To cover all standard hash modes, partitioning into two parts with variable sizes according to the hash modes is required. Moreover, it makes the design of the fault-resilient state even more complex. Addressing this challenge, our contributions to this work are listed as follows:

(1) We introduce a new in-place state-partitioning mechanism. The proposed mechanism is a byte-wise state partitioning. Thus, it is flexible, lightweight, and can serve in all required hash standard modes, namely: SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, SHAKE-128, and SHAKE-256. This allows building a unified SHA-3/SHAKE engine that meets the requirements of lightweight cryptography, specifically regarding area and power.
(2) For reliable and secure implementation, we propose a new low-overhead fault-resilient mechanism for KECCAK state. It is a multi-dimensional cross-parity check, exploiting the cube structure of the KECCAK state. First, we present a one-dimensional (column) parity check. Second, the proposed mechanism is extended by two-dimensional parity: one-dimensional parity for the lane and one more for the sheet to realize a three-dimensional cross-parity, detecting up to three faulty bits in the KECCAK state.

(3) Since RISC-V is a key player for future embedded system-on-chip and microcontroller design, we integrate the resulting fault-resilient unified hash engine into a 32-bit RISC-V-based microcontroller and implement it using a 45 nm standard library and in a Virtex-7 FPGA. The hardware implementation results show that the integration of the proposed unified SHA-3/SHAKE engine into a 32-bit RISC-V-based microcontroller results in a suitable approach for resource-constrained embedded systems applications, especially for PQC at the edge level.

1.3 Outline

First, the description of SHA-3 and SHAKE in general and KECCAK in particular is provided in Section 2. The new design of a unified SHA-3/SHAKE engine architecture is proposed in Section 3. Subsequently, the new fault-resilient mechanism of KECCAK state is illustrated in Section 4. The corresponding implementation results of the unified SHA-3/SHAKE engine and the fault–resilient mechanism are presented in Section 5 followed by the findings of the paper summarized in Section 6.

2 BACKGROUND

This section highlights the definitions and notation used in this work. It then describes the hash functions SHA-3 and SHAKE in general and, in particular, KECCAK as the core component of the overlying hash functions.

2.1 Notation

We use the following notation: We denote a message consisting of n bytes of data as $\mathcal{M} = [m_1, m_2, \ldots, m_n]$, where m_i specifies the i^{th} message byte. Similarly, the generated hash value comprising m bytes of data is described as $\mathcal{H} = [h_1, h_2, \ldots, h_t]$, where h_k specifies the k^{th} hash byte. Furthermore, we denote S as an array representing the bit-wise KECCAK state, and $S[x, y, z]$ as a corresponding bit, where every other array throughout the paper is described similarly. To prohibit the disclosure of the faulted hash, we denote signalizing a detected fault via the signal *error*.

2.2 SHA-3 and SHAKE

In 2012, NIST declared KECCAK [15, 16] the winner of the SHA-3 competition, which was initiated in 2007. This competition aimed to create a new cryptographic hash function to complement the existing SHA-2 family. Consequently, NIST published FIPS202 [28] in 2015, which standardized the hash-function family SHA-3. In contrast to its predecessors, KECCAK is based on the novel sponge construction concept as described in [15]. KECCAK has since then found applications in many other cryptographic primitives like SPONGEWRAP [15] and KMAC [20], SPONGENT [11].

SHA-3 comprises four distinct hash functions, each distinguished by the length of the resulting digest they produce. These variants are known as SHA3-224, SHA3-256, SHA3-384, and SHA3-512, where the suffix following the hyphen signifies the size of the output digest. In addition, this family also consists of two extendable-output functions (XOFs), named SHAKE-128 and SHAKE-256 [28]. For these functions, the output length can be chosen arbitrarily to meet the requirements of individual applications [28]. The proposed term SHAKE combines the term *Secure Hash Algorithm* with KECCAK [28]. They all share the common underlying Sponge construction, thus making them examples of sponge functions.

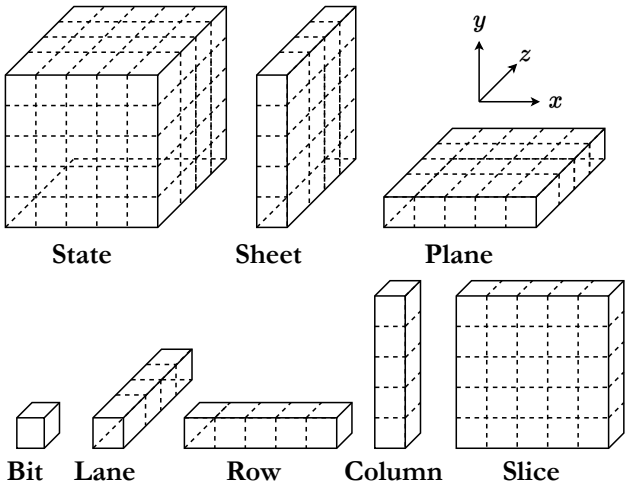

Fig. 3. Naming convention for the subelements of a Keccak state [28].

2.3 Keccak

As the foundation of all Sha-3 and Shake hashes, Keccak relies on the sponge construction as shown in Fig. 1. The sponge construction works on input data with variable length and arbitrary output length, as proposed by Bertoni et al. [15, 28].

The sponge construction consists of a three-dimensional state $b = x \times y \times 2^l = 5 \times 5 \times 64 = 1600$ for the standardized Sha-3 and Shake, where l is defined as $l = \log_2(b/25)$. The state (block) consists of sub-elements noted as *column*, *lane*, *row*, *sheet*, *slice* and *plane* as shown in Fig. 3.

A *sheet* consists of 64 *columns* in z- or 5 *lanes* in y-direction, respectively. A *slice* is built out of 5 *columns* in x- or 5 *rows* in the y-direction. A plane is described as an x/z-direction layer consisting of 5×64 *bits*, 64 *rows*, or 5 *lanes*, respectively. To build the described elements, column and row contain 5 bits each, and a lane comprises 64 bits. The state b is split by the Sha-3 and Shake mode-dependent rate r and capacity c in two consecutive parts, such that $b = r + c$. For a hash generation, the input message is padded based on Sha-3 or Shake to multiple blocks with r-size as shown in Algorithm 1, resulting in a string P such that $m + len(P)$ is a positive multiple of r.

Algorithm 1 Pad Sha-3 / Shake [28]

Input: Rate size r, Message size m, Hash Mode *mode*
Output: String P such that $m + len(P)$ is a positive multiple of r.

if mode == SHA-3 **then**
  $j \leftarrow (-m - 4) \bmod r$
  $P \leftarrow 01||1||0^j||1$
else if mode == SHAKE **then**
  $j \leftarrow (-m - 6) \bmod r$
  $P \leftarrow 1111||1||0^j||1$
end if
return P

Then, it is XOR'ed with the rate r as illustrated in Fig. 1 before. For every r-sized block and the current capacity c, the KECCAK-$f[b]$ permutation for $n_r = 12 + 2l = 24$ rounds is performed as shown in Algorithm 2:

Algorithm 2 KECCAK-f[1600] [28]

Input: KECCAK State Array S, Round Constant rc
Output: KECCAK State Array S'

for all $0 \leq i < 24$ **do**
  for all triples (x, y, z) such that $0 \leq x < 5, 0 \leq y < 5$ and $0 \leq z < 64$ **do**
    $\theta[x, z] \leftarrow S[x, y, z] \oplus (\oplus_{y=0}^{4} S[x-1, y, z]) \oplus (\oplus_{y=0}^{4} S[x+1, y, z-1])$
    $\rho[x, y, z] \leftarrow \theta[(x + 3y) \bmod 5, x, z]$
    $\pi[x, y, z] \leftarrow \rho[(x + 3y) \bmod 5, x, z]$
    $\chi[x, y, z] \leftarrow \pi[x, y, z] \oplus ((\pi[(x + 1) \bmod 5, y, z] \oplus 1) \cdot \pi[(x + 2) \bmod 5, y, z])$
    $\iota[0, 0, z] \leftarrow \chi[0, 0, z] \oplus rc[i, z]$
    $S'[x, y, z] \leftarrow \chi[x, y, z], S'[0, 0, z] = \iota[0, 0, z]$
  end for
end for
Return S'

In the θ layer, the sum of every column is generated and interlocked with a corresponding column of the state. In the layers ρ and π, bit-shuffling in every *sheet* and *slice* is performed, followed by the non-linear layer χ. In ι, the round-constant $rc(i)$ of the corresponding round i is added to the mid-lane. After the padded massage is *absorbed*, the digest is *squeezed* by taking the data from the *rate* and refresh the state by KECCAK-$f[1600]$ if the digest size is greater than the size of the rate as shown in Fig. 1.

3 Unified SHA-3/SHAKE Architecture

In this section, we introduce a unified low-cost hardware architecture for both SHA-3 and SHAKE standard hash functions. In particular, we present the design of a byte-wise in-place state-partitioning mechanism. This lightweight partitioning mechanism allows the building of a unified SHA-3/SHAKE engine for all required hash modes that meet the requirements of lightweight cryptography, specifically regarding area and power.

Since both hash functions have KECCAK core as a key building block, the proposed unified engine of SHA-3 and SHAKE can share and jointly use a KECCAK core. The design decisions are made to achieve low power, low area consumption, and high throughput, targeting embedded and IoT applications. For this purpose, the input message m_i and the hash output h_k work on a byte-granular data resolution.

Fig. 4 depicts the proposed unified hash engine. It comprises a KECCAK core, a padding and update unit, a control unit, and a register holding the KECCAK state. Considering the SHA-3 and SHAKE standards, the size of the KECCAK state S is 1600 bits. This huge state contributes a large area to the overall engine design. The proposed engine utilizes in-place data processing (byte-wise) to minimize the impact of the corresponding state register i.e., the state is byte-wise logically partitioned and has a special padding and update unit to cover all SHA-3 and SHAKE standard hash modes, namely [28]: SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, SHAKE-128, and SHAKE-256 as shown in Fig. 5.

We consider two different rate sizes to support all the hash modes. The first rate size is denoted as $r_{sr} = 1344$, which is pre-determined by the SHAKE-128 rate size and the largest of all SHA-3 and SHAKE hash modes. Only the part $S[1343:0]$ of the KECCAK state specified by r_{sr} is updated and configured as a shift register to enable in-place data processing and keeps the state update delay as low as possible. The state's capacity part $S[1599:1344]$ consists of $c_{sr} = 256$ bits and is

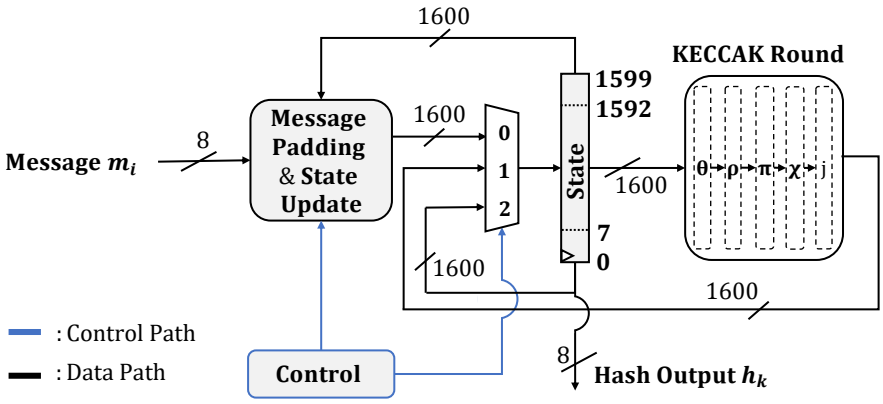

Fig. 4. SHA-3/SHAKE Unified Engine Design

configured only to be updated by the output of KECCAK. The second rate size r_{mode} is determined using the hash mode. If a hash mode other than SHAKE-128 is selected, the mode-specific rate r_{mode} is smaller than r_{sr}, otherwise r_{mode} is equal to r_{sr}. The padding and update unit performs the state update in place by XOR'ing the message byte m_i with the state's least significant byte (LSB) and appending it to the most significant byte (MSB) of the rate as

$$S'[1343:0] \leftarrow (m_i \oplus S'[7:0]) \parallel (S[1343:0] \gg 8)\,. \tag{1}$$

A *ratecount* increases with every consumed message byte. When $ratecount = r_{mode}$, the mode-specific rate r update is finished. To keep the state consistent, the remaining $r_{sr} - r_{mode}$ bytes are XOR'ed with zero bytes and appended to the MSB side of the shift register as described before. This mechanism makes the remaining bytes extend the capacity size c_{sr} for the corresponding hash mode by $r_{sr} - r_{mode}$ bytes. To perform the padding according to SHA-3 and SHAKE as described in Algorithm 1, a 5-to-1 multiplexer selects the corresponding padding byte, so the state update can be performed as described before. Further, when the shift register part $S[1344:0]$ of the KECCAK state is updated by the (padded) message and the mode-dependent capacity if needed, the KECCAK-f permutation is performed. To achieve a

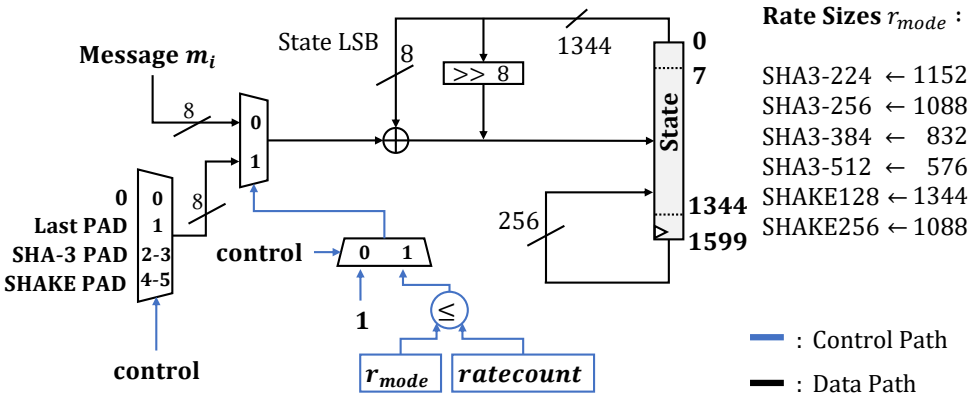

Fig. 5. KECCAK State Update by (Padded) Message

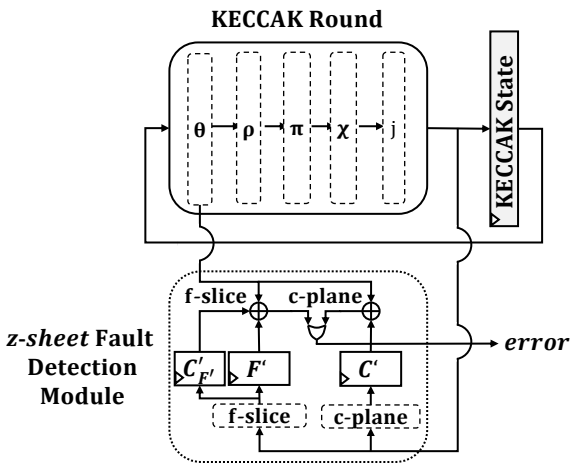

Fig. 6. Combined FD Schemes for KECCAK

low-area and low-power design while ensuring sufficiently high performance, the permutation works as a round-based implementation, updating the state without introducing an additional register. To reduce the computing latency, the KECCAK-f permutation rounds can be unrolled for the cost of additional hardware overhead. The output of a hash byte h_k is performed by selecting the LSB of the KECCAK state $S[7:0]$. To keep the state consistent, the rate update is performed as described before by XOR'ing the hash byte h_k with a zero byte and appending it to the MSB side of the shift register. The KECCAK state is preserved to perform a state update by KECCAK in the case more hash bytes are requested than the rate provides. This procedure is repeated until all requested hash bytes are transmitted.

4 Lightweight Fault Detection Mechanism for KECCAK State

This section presents the lightweight fault detection (FD) mechanism for KECCAK. The main goal of the proposed FD approach is to protect the KECCAK state against faults at minimal hardware cost. The proposed FD mechanism combines spatial and information redundancy as FD schemes. Then, we apply the proposed FD mechanism in the unified hash engine. We show the FD mechanism's flexibility and integration in two realizations of the unified hash engine, e.g., round-based and unrolled implementations. To demonstrate, we integrate the resulting fault-resilient unified hash engine into a RISC-V-based microcontroller.

4.1 Spatial & Information Redundancy Scheme for Fault Detection

Fig. 6 shows the proposed FD mechanism as a hardware module placed alongside KECCAK realizing spatial redundancy. The proposed mechanism consists of two main building blocks: *c-plane* and *f-slice* as one-dimensional parity check schemes. Both one-dimensional parity checks are combined to provide a two-dimensional parity check along a *sheet*, denoted by *z-sheet*. Therefore, the proposed multi-dimensional parity check mechanism can detect up to three-bit flips.

4.1.1 c-plane-based Parity Check. Algorithm 3 illustrates the theta layer of KECCAK. Let $\boldsymbol{S}$ be the three-dimensional KECCAK state that holds the 1600 state bits. In the first step, the sum of every column is generated by XOR-ing the corresponding bits. This generates the plane $\boldsymbol{C}$, accounting for 320 bits.

Algorithm 3 Theta Layer [28]

Input: KECCAK State Array S
Output: KECCAK State Array θ

for $x \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
  $C[x,z] \leftarrow S[x,0,z] \oplus S[x,1,z] \oplus S[x,2,z] \oplus S[x,3,z] \oplus S[x,4,z]$
end for
for $x \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
  $D[x,z] \leftarrow C[(x-1) \bmod 5, z] \oplus C[(x+1) \bmod 5, (z-1) \bmod 63]$
end for
for $x \leftarrow 0$ to 4, $y \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
  $\theta[x,y,z] \leftarrow S[x,y,z] \oplus D[x,z]$
end for
Return θ

After calculating the plane C, the plane D is generated by adding two diagonal columns nearby. The output of the theta layer θ is calculated by adding a bit of the D plane to all bits in the corresponding column of the KECCAK state S. With our approach, we take advantage of established column sum generation performed in the theta layer and realize a FD based on parity checks. This solution exhibits low hardware and latency overhead. In particular, we exploit the result of the C plane computation, which is denoted by the *c-plane* and shown in Fig. 7. In detail, the *c-plane*-based FD mechanism is described in Algorithm 4:

Algorithm 4 *c-plane*-based fault detection

Input: *c-plane* C from the KECCAK theta layer, KECCAK state update S'
Output: Fault detection signal *error*

$error \leftarrow 0$
▷ Calculate the column sum bits C' for comparison in the next KECCAK round inside the protection unit
for $x \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
  $C'[x,z] \leftarrow S'[x,0,z] \oplus S'[x,1,z] \oplus S'[x,2,z] \oplus S'[x,3,z] \oplus S'[x,4,z]$
end for
wait for next clock cycle
▷ Compare the *c-plane* C from the KECCAK theta layer and the stored *c-plane* C' for parity-checking
for $x \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
  $error \leftarrow error \vee (C[x,z] \oplus C'[x,z])$
end for
Return *error*

When a KECCAK permutation round is performed, the output signal of the KECCAK round function S' updates the KECCAK state register. The state-update signal S' is routed into the FD unit to calculate the sum of all 320 columns and storing them in the register of C' for comparison in the next KECCAK permutation round. While performing the next permutation round, the column sums C are generated in the KECCAK theta layer as described in Algorithm 3. Every bit of C is compared with the corresponding pre-computed bit stored in the register of C'. This approach provides a parity check for each column of the stored KECCAK state S. If at least one of the bits in C differs from the stored bits of C', the *error* signal is raised to determine a detected fault.

4.1.2 f-slice Lane Sum Computation. We extend the existing column sum calculation *c-plane* present in the theta layer with a lane sum generation called *f-slice* as shown in Fig. 8.

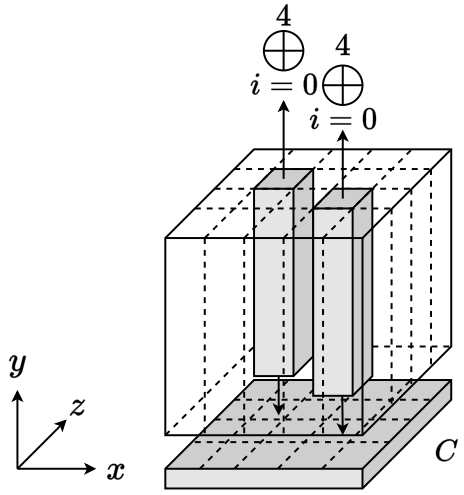

Fig. 7. *c-plane* Column Sum Generation

Algorithm 5 describes *f-slice* in details. Here, every lane's sum of the KECCAK state is calculated, and the results generate a slice for comparison, accounting for an additional 25 bits.

By adding the *f-slice* to the theta layer and FD module, a parity-check similar to the *c-plane*-based parity check is performed. Combining the *c-plane*- and *f-slice*-based parity check enables a two-dimensional cross-parity check. The so-called *z-sheet*-based parity-check provides a *sheet*-based protection mechanism described as follows.

4.1.3 z-sheet-based Parity Check. The *z-sheet*-based cross-parity check is described in Algorithm 6. Similar to the *c-plane*-only approach, the state update signal S' serves as an input for the FD module. In addition to the generation of the plane C', the slice F' is calculated and the resulting bits are stored inside the FD module to be compared in the next

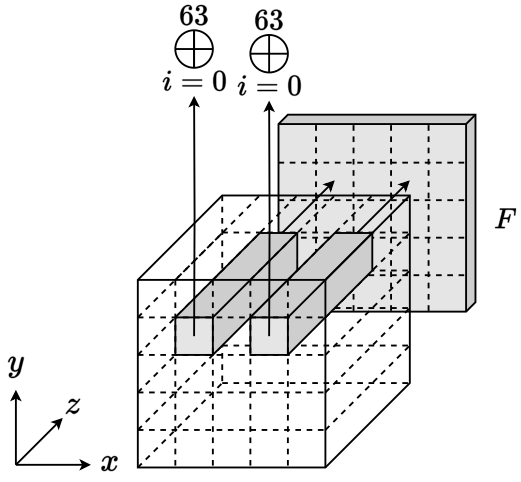

Fig. 8. *f-slice* Lane Sum Generation

Algorithm 5 *f-slice* Lane Sum Computation

Input: Keccak State Array $\boldsymbol{S}$
Output: Lane Sum Slice $\boldsymbol{F}$

$\boldsymbol{F} \leftarrow 0$
for $x \leftarrow 0$ to 4, $y \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
    $F[x,y] \leftarrow F[x,y] \oplus S[x,y,z]$
end for
Return $\boldsymbol{F}$

Keccak round with the extended slice $\boldsymbol{F}$ of the theta layer. To protect the register of $\boldsymbol{F'}$, a parity check is added and performed as shown in Fig. 9. During the calculation of $\boldsymbol{F'}$, the sum $\boldsymbol{C'_{F'}}$ of every $\boldsymbol{F'}$s column is generated, accounting for another 5 sum bits. Similar to the parity check of the Keccak state $\boldsymbol{S}$, the column-wise sum $\boldsymbol{C_{F'}}$ of the $\boldsymbol{F'}$ registers are calculated and compared to the bits previously stored in the registers of $\boldsymbol{C'_{F'}}$.

Algorithm 6 *z-sheet*-based Fault Detection

Input: *c-plane* $\boldsymbol{C}$ & *f-slice* $\boldsymbol{F}$ from the Keccak theta layer, Keccak state update $\boldsymbol{S'}$
Output: Fault detection signal *error*

$\boldsymbol{F'} \leftarrow 0$, $\boldsymbol{C'_{F'}} \leftarrow 0$, $\boldsymbol{C_{F'}} \leftarrow 0$, $error \leftarrow 0$
▷ Calculate the column sum bits $\boldsymbol{C'}$ for comparison in the next Keccak round
for $x \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
    $C'[x,z] \leftarrow S'[x,0,z] \oplus S'[x,1,z] \oplus S'[x,2,z] \oplus S'[x,3,z] \oplus S'[x,4,z]$
end for
▷ Calculate the lane sum bits $\boldsymbol{F'}$ for comparison in the next Keccak round
for $x \leftarrow 0$ to 4, $y \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
    $F'[x,y] \leftarrow F'[x,y] \oplus S'[x,y,z]$
end for
▷ Calculate the sum of a $\boldsymbol{F'}$ column for comparison in the next Keccak round
for $x \leftarrow 0$ to 4, $y \leftarrow 0$ to 4 **do**
    $C'_{F'}[x] \leftarrow C'_{F'}[x] \oplus F'[x,y]$
end for
wait for next clock cycle
▷ Calculate the sum of a $\boldsymbol{F'}$ column for comparison
for $x \leftarrow 0$ to 4, $y \leftarrow 0$ to 4 **do**
    $C_{F'}[x] \leftarrow C_{F'}[x] \oplus F'[x,y]$
end for
▷ Compare the *c-planes* $\boldsymbol{C}/\boldsymbol{C'}$, *f-slices* $\boldsymbol{F}/\boldsymbol{F'}$ and $\boldsymbol{F'}$ column sums $\boldsymbol{C_{F'}}/\boldsymbol{C'_{F'}}$ for parity-checking
for $x \leftarrow 0$ to 4, $y \leftarrow 0$ to 4, $z \leftarrow 0$ to 63 **do**
    $error \leftarrow error \vee (C[x,z] \oplus C'[x,z]) \vee (F[x,y] \oplus F'[x,y]) \vee (C_{F'}[x] \oplus C'_{F'}[x])$
end for
Return *error*

The *z-sheet*-based parity check realizes a multidimensional parity check based on the *sheets* of the Keccak state. It should be noted that multidimensional cross-parity checks are an established technique for designing forward error correction (FEC) codes [12, 18, 38, 43] for data transmission over unreliable data channels. Here, the transmitted data is represented in a multidimensional space to perform FEC over the state space. However, a multidimensional cross-parity check for FEC is a costly technique from the hardware perspective. The proposed two-dimensional cross-parity check approach provides a lightweight FD approach detecting up to three faults (bit flips) in the register of the Keccak state $\boldsymbol{S}$.

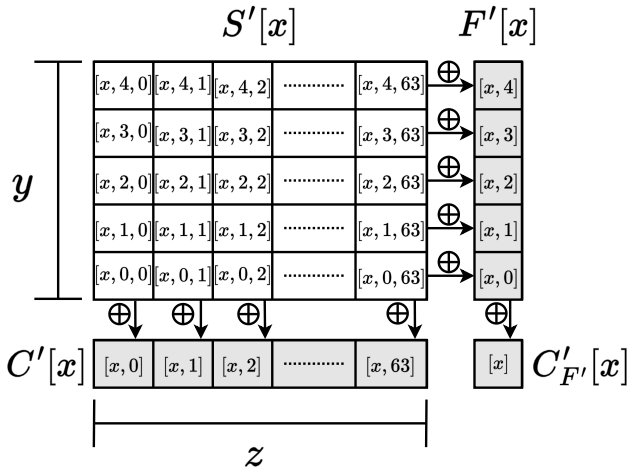

Fig. 9. *z-sheet* column and lane sum calculations carried out in the FD module

4.2 Integration of the FD Mechanism into the Hash-Engine

In the following, we show the integration of the proposed FD mechanism into the unified hash engine covering two implementation scenarios: round-based, and unrolled KECCAK implementations.

4.2.1 Round-based KECCAK Implementation. We integrate the proposed FD mechanism stated in Section 4 into the unified hash engine. The FD mechanism is implemented as a redundant module as depicted in Fig. 10.

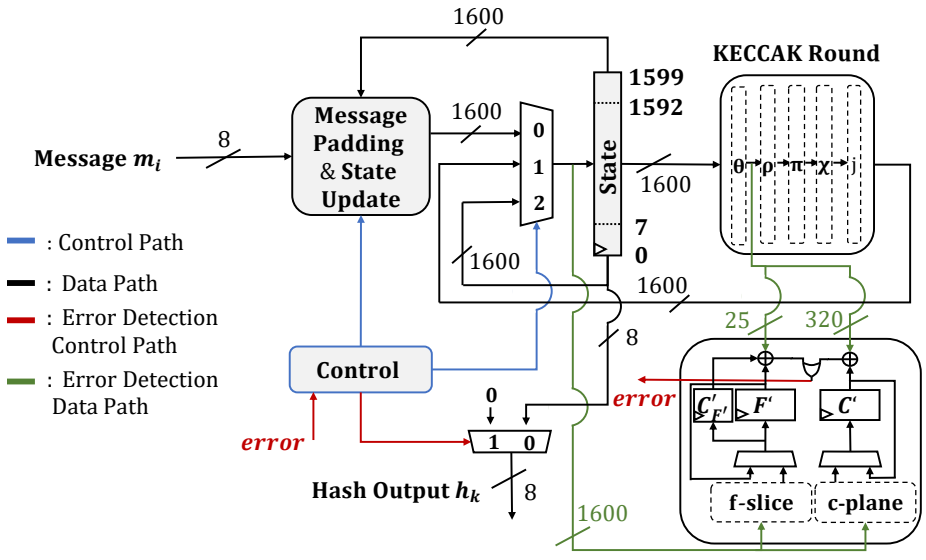

Fig. 10. SHA-3/SHAKE FD Mechanism Design

The state-update signal S' is forwarded as an input to the FD module. Here, the *c-plane* column and *f-slice* lane sum calculation is performed and stored in the registers of C' and F', respectively. During the next computation cycle, the column sum C and lane sum F is calculated in the theta layer of the Keccak and forwarded to the FD module for comparison with the stored values in C' and F'. In the case the *z-sheet*-based cross-parity check is performed the column sum of F' *f-slice* is generated likewise. It is stored in the register of $C'_{F'}$ and compared by the generated column sum of F' during the next computation cycle. If any comparison fails, the column-wise *c-plane*- or two-dimensional *z-sheet*-based parity check mechanism detects a fault and the signal *error* is raised. As a result, the hash engine output is masked, preventing the leakage of the faulty digest.

It should be noted that both the *c-plane*- and *z-sheet*-based parity check focus on protecting the Keccak state registers. This work does not include any further protection of the unified hash engine's control unit and data interface.

4.2.2 Unrolled Keccak Implementation. As our FD mechanism relies on the Keccak state-update signal S' to check the integrity of the state S's register, it applies to other realizations of the Keccak-f permutation as long as the state is updated accordingly. We show the flexibility of our approach not only by a round-based but also by different unrolled implementations. For this, we modified our unified hash engine and realized unrolled implementations of the Keccak-f permutation function as shown in Fig. 11.

Similar to the integration of the FD module into the round-based implementation of our unified hash engine, the state-update signal S' serves as an input for the FD module, and the plane C' is calculated. In case of the *z-sheet*-based cross-parity check, the slice F', and F's column sum $C'_{F'}$ is computed. Accordingly, after the state register of S is updated, the plane C and slice F from the theta layer of the first unrolled round is routed into the FD module for comparison. Similar to the round-based implementation, the comparison of $C_{F'}$ and $C'_{F'}$ for the *z-sheet*-based parity check is performed.

Based on the use case, our approach can be adapted from small round-based implementations to more sophisticated realizations with an unrolled design. Furthermore, our FD mechanism applies to unrolled and pipelined implementations

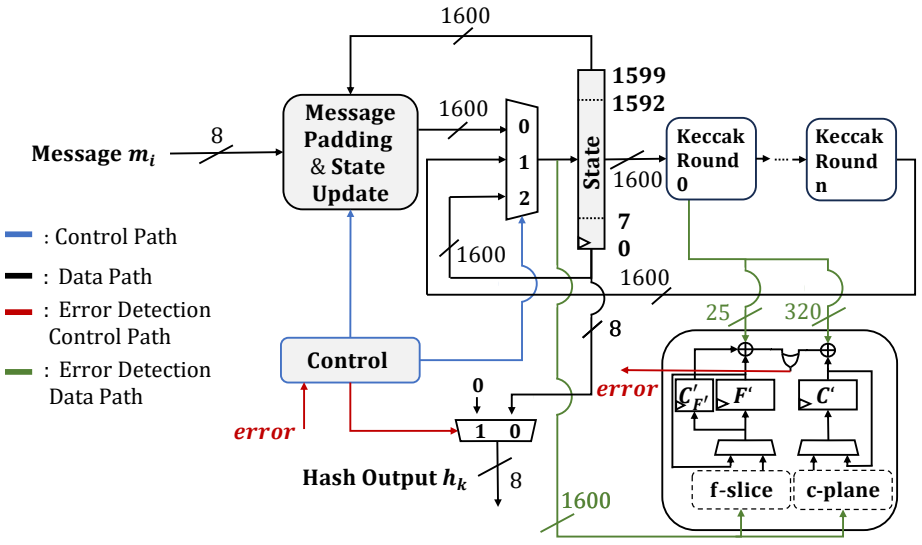

Fig. 11. Unrolled Sha-3/Shake Protected Hash Engine.

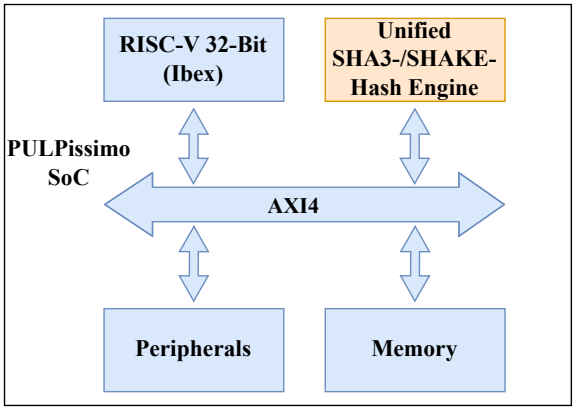

Fig. 12. Block Diagram of 32-bit RISC-V Microcontroller with the Proposed Unified Hash Engine

deployed in high-performance applications. In the latter case, the FD module is applied to every register holding the KECCAK state S and reports detected faults. The proposed unified hash engine is intended to be deployed in resource-constrained devices. Pipelined fault-resilient KECCAK implementations would have a huge impact on the required hardware, thus we do not consider such an implementation in this work.

4.3 Integrating the Unified Hash Engine into a RISC-V SoC

The unified hash engine's design provides a byte-wise input/output interface that can be connected to a bus interface, like AXI4 or Tilelink. Therefore, the engine can be integrated into any System-on-Chip (SoC) as a memory-mapped IP. The hash engine is to be integrated into a low-power 32-bit RISC-V microcontroller, as shown in Fig. 12, to be comparable with the SoA hardware used in PQC.

5 Implementation Results

This section shows the implementation results of the SHA3/SHAKE unified hash engine and the integration of the proposed FD mechanism and compares its security level to SoA fault injection countermeasures for KECCAK. Then, we compare our implementation results for ASIC and FPGA technology to SoA. We also showcase the performance and the integration results into the PULPissimo SoC of our design.

5.1 The Unified Hash Engine Implementation

The proposed SHA3/SHAKE engine is written in VHDL and synthesized for 45 nm FreePDK45 [42] standard library using Synopsys Design Compiler U-2022.12 [40]. We attached the proposed engine via AXI4 to the PULPissimo SoC [39] as a memory-mapped device to demonstrate integration into a lightweight SoC. We also implemented the design on the Xilinx ZCU102 evaluation board containing a Virtex-7 FPGA using Xilinx Vivado 2022.1 [45].

5.1.1 Round-based Implementation. In the following, we discuss the implementation results of the proposed SHA3/SHAKE engine in ASIC and FPGA technologies. It covers three cases: the unprotected, the *c-plane*, and the *z-sheet* protected engine.

Round-based FPGA Implementation Results. Table 2 shows the FPGA implementation results compared to the SoA unified SHA3/SHAKE engines of [37] and [44] based on FPGA primitives, namely Look-up-Tables (LUT), Flip-Flops (FF), and DSP; the latter denotes a cluster including LUTs and FFs.

Table 2. FPGA Implementation Comparison of Unified SHA-3/SHAKE Engines

Paper	Area			FPGA
	LUT	FF	DSP	
[37]	5113	3068	-	Ultrascale+
[44]	7376	3059	4	Kintex-7
[This work]	3566	1645	-	Virtex-7

Although our design of a unified SHA3/SHAKE engine covers all standard hash modes: SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, SHAKE-128, and SHAKE-256 while the design of [37] and [44] only realizes three hash modes, our design requires less area compared to [37], due to our proposal for byte-wise in-place state partitioning. This shows the flexibility and lightweightness of our unified SHA3/SHAKE engine.

For general FPGA comparison with the SoA KECCAK implementations, Table 3 summarizes the FPGA implementation results. Our *c-plane*-based resilient hash engine requires 2.07× fewer slices than the SoA solutions detecting up to one bit flip [19, 25]. Compared to SoA approaches, detecting up to three-bit flips [13], our *z-sheet*-based FD mechanism shows an improvement factor of 2.4 regarding LUT overhead (89% vs. 214%) and a 5× FF overhead improvement (20% vs. 100%). For the protected design, this results in overall 1.8× fewer LUTs and 1.6× fewer FFs. This underlines our FD mechanism as a lightweight, resilient solution that is competitive with the state of the art.

Round-based ASIC Implementation Results. We compare different SoA implementations by using the concept of *gate equivalent* (GE) as shown in Table 4. The results show that our protected design requires less area than the SoA fault detection approaches for one bit-flip. The *c-plane*-protected hash engine exhibits 1.48× smaller area requirements than the SoA solutions detecting up to one bit flip [5, 24]. Compared to the SoA solution detecting up to three-bit flips

Table 3. Round-based FPGA Implementation Results Comparison

Paper	without protection				with protection				FPGA	Fault Detectability (bit flips)
	LUT	FF	Slices	Freq. (MHz)	LUT	FF	Slices	Freq. (MHz)		
[13]	3953	1621	1036	222.2	12408	3260	3269	111.10	Artix-7	≤ 3
[41] ArchHC	2339	2361	-	228.25	28703	18192	-	45.89	Virtex-7	–
[41] ArchTMR_HC	2339	2361	-	228.25	27226	26197	-	63.58	Virtex-7	–
[25]	-	-	1370	258.60	-	-	1680	387.00	Virtex-5	≤ 1
[19]	-	-	1356	296.50	-	-	2260	291.30	Virtex-5	≤ 1
[26]	-	-	1350	252.40	-	-	1601	365.2	Virtex-5	< 1
[This work] c-plane	3566	1645	563	20.00	4868	1966	809	20.00	Virtex-7	≤ 1
[This work] z-sheet	3566	1645	563	20.00	6738	1996	1067	20.00	Virtex-7	≤ 3

Table 4. Round-based ASIC Implementation Results Comparison

Paper	without protection		with protection		PDK	Fault Detectability (bit flips)
	Area (kGE)	Freq. (MHz)	Area (kGE)	Freq. (MHz)		
[13]	57.10	1.316,00	177.70	719.40	FreePDK45	≤ 3
[24]	52.14	256.94	83.48	228.26	FreePDK45	≤ 1
[5]	45.40	676.00	49.10	1192.00	65nm TSMC	≤ 1
[35] KIT	-	-	89.60	-	FreePDK45	< 1
[35] FUR	-	-	177.20	-	FreePDK45	< 1
[35] FAST	-	-	1317.80	-	FreePDK45	< 1
[35] BLAZE	-	-	2494.60	-	FreePDK45	< 1
[This work] c-plane	25.65	714.29	33.17	666.67	FreePDK45	≤ 1
[This work] z-sheet	25.65	714.29	39.96	588.24	FreePDK45	≤ 3

[13], our *z-sheet*-based FD mechanism itself shows an improvement in area overhead by 3.7× (56 % vs. 211 %). This improvement results in an overall 4.5× smaller hash-engine design while achieving the same fault detectability.

5.1.2 Unrolled Implementation. The following discusses the unrolled implementation results of the proposed SHA3/SHAKE engine in ASIC and FPGA technologies. It covers several unrolled implementation cases, ranging from two to 24 rounds.

Unrolled FPGA Implementation Results. Table 5 shows the FPGA results of the unrolled implementation of the unified engine with/without the proposed FD mechanism. *Z-sheet* protection causes area overhead from 50.7% for two unrolled rounds to just 6.56% for a fully unrolled engine.

Table 5. Unrolled FPGA Implementation Results

# Unrolled Rounds	w/o protection			w/ *c-plane* protection			w/ *z-sheet* protection			
	LUT	FF	Slices	LUT	FF	Slices	LUT	FF	Slices	Slices Overhead
2	5694	1645	869	8358	1966	1273	8862	1996	1310	50.7 %
4	9930	1645	1460	12140	1966	1869	14637	1996	2318	58.7 %
6	14153	1645	2074	21120	1966	3162	19477	1996	2946	42.02 %
8	18384	1645	2784	21853	1966	3223	20479	1996	3199	14.9 %
12	26836	1645	4452	28759	1966	4677	31194	1996	5083	14.1 %
24	52131	1645	8625	56764	1966	9333	56503	1996	9191	6.56 %

The FPGA implementation demonstrates that increasing the amount of unrolling leads to a significant decrease in *z-sheet* protection area overhead.

Unrolled ASIC Implementation Results. Table 6 shows the impact of the proposed FD mechanism on unrolled implementations of the unified hash engine.

Table 6. ASIC results for unrolled implementations

# Unrolled Rounds	w/o protection		w/ *c-plane* protection		w/ *z-sheet* protection		
	Area (kGE)	Freq, (MHz)	Area (kGE)	Freq. (MHz)	Area (kGE)	(% Overhead)	Freq. (MHz)
2	45.57	617.28	53.81	540.54	60,04	31.7 %	423,73
4	64.06	403.23	73.57	367.65	75,90	18.4 %	280,11
6	89.17	276.24	93.01	202.43	109,97	23.3 %	223,21
8	121,24	201,21	130.40	193.05	135,92	12.1 %	176,37
12	175.75	135.69	182.25	131.75	190,57	8.4 %	124,69
24	314.76	72.25	333.15	71.28	339,06	7.7 %	68,45

The proposed FD mechanism (*z-sheet*) itself leads to an area overhead depending on the amount of unrolling, ranging from 31.7% (two rounds) to 7.7% (24 rounds). With increasing numbers of unrolled rounds, the *z-sheet* protection overhead significantly decreases. As a result, a fully unrolled engine (24 rounds) only shows 7.7% area overhead for applied *z-sheet* protection.

5.2 Power & Throughput Analysis

We evaluate the FD mechanism's impact on the round-based unified hash engine from power and throughput perspectives. For this, we synthesized the unified hash engine and FD module for the 45 nm FreePDK45 standard library [42]. Table 7 shows our design's power consumption for three configurations: unprotected, protected with *c-plane*, and protected with *z-sheet*.

Table 7. Power for unprotected and protected designs based on 45 nm FreePDK45 results

Design	Power / mW
Basic	9.50
w/ c-plane	8.63
w/ z-sheet	9.05

The *c-plane* protected design requires lower power than the unprotected design due to the decreasing clock frequency. In the case of *z-sheet* protection, the combinational logic and register increases, and more power is consumed compared to the *c-plane*-based protection. The throughput results presented in Fig. 13 are calculated based on the hash modes.

As expected from the lowered maximum clock frequency, the throughput for *c-plane* and *z-sheet* protected designs are lower than for the basic design. However, as outlined in Section 5.2, the throughput increases with hash modes using a higher rate size for operating on byte granularity and in-place updating of the (padded) messages.

5.3 Integration into PULPissimo System-on-Chip

For showcasing our design's applicability to a lightweight SoC running PQC applications , we integrated our unified hash engine as a round-based implementation into the PULPissimo SoC [39]. The engine is attached to the system bus as a memory-mapped device using AXI4 as illustrated in Fig. 12. For analysis, we deployed PULPissimo's FPGA toolflow [34], which implements the design on a the Virtex-7 FPGA for a 20 MHz clock frequency. The results are presented in

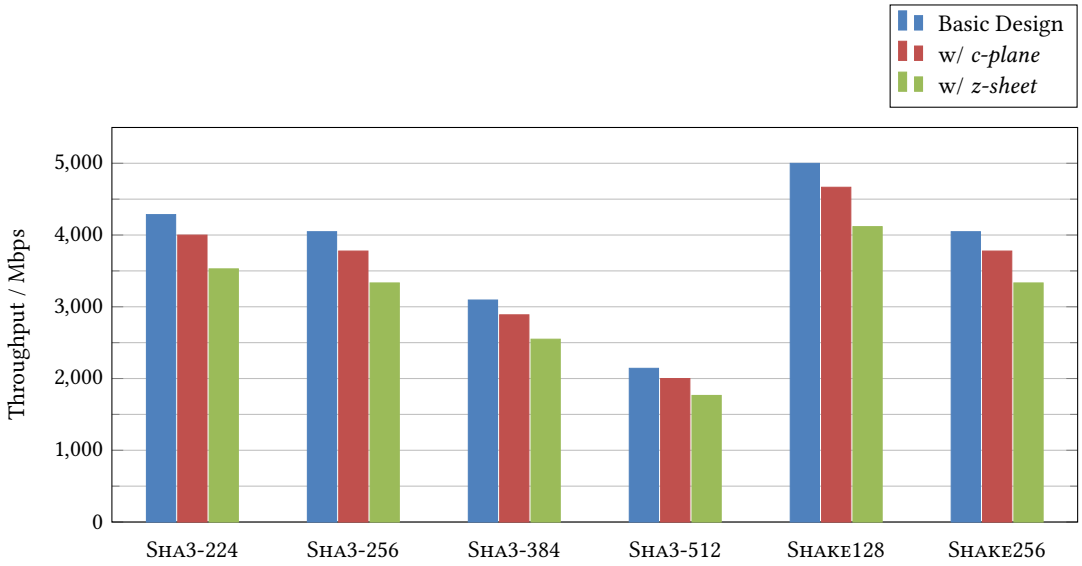

Fig. 13. Throughput of the hash-modes based on 45nm FreePDK45 results

Table 8. Hash Engine Area Overhead for PULPissimo SoC on Virtex-7 FPGA

Virtex-7 @ 20MHz	Protection Mechanism none	c-plane	z-sheet
LUTs (SoC)	52976	54046	55538
LUTs (Hash Engine)	3566	4868	6738
% (Hash Engine)	6.73 %	9.00 %	12.13 %
Flip-flops (SoC)	42317	42638	42668
Flip-flops (Hash Engine)	1645	1966	1996
% (Hash Engine)	3.89 %	4.61 %	4.68 %
Slices (SoC)	13247	13662	13849
Slices (Hash Engine)	563	809	1067
% (Hash Engine)	4.25 %	5.92 %	7.70 %

Table 8. Registers consume high area overhead compared to logic elements, hence, the amount of registers in our design is kept as low as possible, leading to a minimal register overhead of 3.9 % for the unprotected hash engine integrated in the PULPissimo SoC. With applied protection mechanisms, the register overhead increases to 4.7% for the *z-sheet*-based protection. Furthermore, an overhead of <13% for LUTs and <8% for slices is observed. These results emphasize the low area overhead of the resulting fault-resilient unified hash engine, thus making it suitable for microcontroller-based and embedded IoT applications.

5.4 Fault Detectability Analysis and Comparison

Multiple fault countermeasures and protection schemes have been proposed in the literature. Depending on whether the fault is detected or corrected, these countermeasures are mainly classified into fault detection and correction schemes [33].

These fault-tolerance techniques can be grouped into three main categories: Spatial redundancy, temporal redundancy, and information redundancy [33]. Therefore, we compare the proposed FD mechanism to SoA implementations presented in Fig. 14.

The comparison is performed as follows:

- **Spatial Redundancy**: This type is based on replicating logic/state and performing redundant computation at the hardware level. In [19], an FD mechanism for Keccak based on a double-redundant scrambling technique was proposed. This mechanism is effective against single bit-flips with a fault coverage of 99.996 %. According to Table 3, the proposed mechanism in [19] exhibits high overhead with even fault detectability compared to our work.
- **Temporal Redundancy**: This type is based on repeating the operation twice on the same logic with the output being compared between both runs. Although less cost-intensive in terms of area, this type of redundancy decreases the performance by 50 % on average for performing each operation twice. In [5], a recalculation with rotated operands was performed to provide a time-redundant FD scheme. This mechanism is efficient only against a single bit-flip. Another work provides a time-redundancy-based scheme by splitting the Keccak round into two halves rounds through the pipeline register [26]. While the result of the first half-round is fed to the second half of the Keccak round, the input of the first half is re-executed and compared with the result stored in a pipeline register. In case of a mismatch, an error is raised. This approach is deployed to the second half of Keccak round to protect the complete design. The performed fault analysis shows a fault-detection probability of 99.458 % for a single bit-flip, thus not providing 100 % detection capability. Both temporal redundant mechanisms of [5] and [26] show a huge impact on the throughput compared to our work.
- **Information Redundancy**: This type is based on incorporating redundant information in the form of parity bits for fault detection. Although this technique is considered cost-effective (e.g., when compared to spatial Redundancy), additional logic is required for parity-bit generation (parity generator) and decoding (parity checker). Further information redundancy-based schemes exist, such as error-correcting codes (ECC), which can detect and correct errors. In [13], the concept of *impeccable circuits* [1] was applied to Keccak, providing an FD mechanism. According to the principles of impeccable circuits [1], coding-scheme-based checkpoints verify the correctness of the underlying computation results and raise an error in case of a mismatch. Here, it detects faults up to three bit-flips. In [24], a predictor/compressor approach with parity checks was presented to protect Keccak against faults. For the theta layer, parity checks in the two-dimensional slices are calculated. As the parity checks do not involve the three-dimensional characteristics of the Keccak-state, only odd numbers of

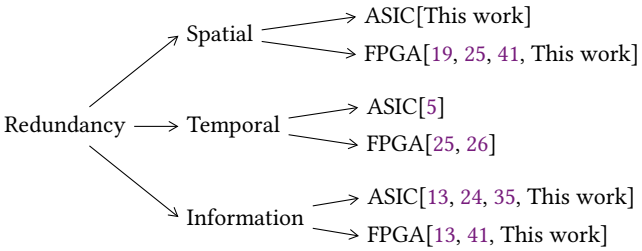

Fig. 14. Overview of State of the Art implementations

faulty bits can be detected. While fault detection based on impeccable circuits achieves the same level of fault detection, it comes at a high hardware overhead.

- **Combined Redundancy Approaches**: Based on the main categories, different redundancy techniques can be combined to form a hybrid FD approach. In [25], the approach presented in [19] and [26] to provide a spatial- and time-redundancy-based FD mechanism was adapted for KECCAK. The fault analysis shows a full fault-detection capability for a single bit-flip. In [41], an error-correction mechanism for KECCAK based on Hamming codes and triple modular redundancy (TMR) was proposed. It provides two basic implementations with a pipelined separation and round-based implementation of the mapping functions. Here, the registers are protected with a (12,8) Hamming code, thus capable of correcting a single-bit error per register byte. In contrast, the mapping functions are protected with TMR by triplicating each function and applying a voter at the outputs to correct a single fault.

Based on the detectability analysis, our FD mechanism can be classified as a low-overhead hybrid spatial- and information-redundancy approach performing (cross-)parity bit checks. It is able to detect up to three-bit flips.

6 Conclusion and Future Work

This paper presented a byte-wise in-place partitioning mechanism of the so-called KECCAK state. The proposed mechanism is lightweight and serves in all required hash modes. This allows us to build a unified SHA-3/SHAKE engine, covering all standard hash configurations, namely: SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, SHAKE-128, and SHAKE-256. Thus, the proposed engine suits modern cryptography and standard post-quantum cryptography (PQC) schemes. Such a unified engine design also meets the requirements of lightweight cryptography, specifically in terms of area and power.

Then, we proposed a lightweight fault-detection mechanism protecting the KECCAK state from bit-flip-induced faults. Here, we introduced a dedicated KECCAK register-state protection to achieve sufficient protection. We exploited the KECCAK state cube and functional characteristics to apply a lightweight cross-parity check mechanism relying on a combined spatial- and information redundancy scheme. Based on an existing column-sum calculation performed on the KECCAK state called *c-plane*, we added a lane-based sum called *f-slice*. In conjunction with a redundancy module, the combined sums provide a two-dimensional cross-parity check along a sheet, denoted as *z-sheet*. The evaluation of fault detectability shows that our proposed solution is efficient against up to three-bit flips. Further, we integrated the lightweight fault detection into an accordingly lightweight unified hash engine for both SHA-3 and SHAKE.

For demonstration and evaluation, we synthesized our design for the 45 nm FreePDK45 standard library and a Xilinx Virtex-7 FPGA. The added *z-sheet*-based protection module shows a vast reduction in area overhead compared to state-of-the-art solutions. With only 56% area overhead compared to 200%, it shows a 3.7× improvement. Our resulting *z-sheet*-protected hash engine demonstrates an even higher 4.5× area-overhead improvement compared to the state of the art. We furthermore integrated the fault-resilient hash engine into the PULPissimo SoC using their FPGA-centric design environment to showcase the usage of our design in resource-constrained microcontroller-based applications: On a Xilinx Virtex-7 FPGA, the engine increases the overall PULPissimo SoC by less than 8 %. These results emphasize the lightweightness of the proposed protection mechanism, proving it suitable for microcontroller-based and embedded IoT applications.

In our future work, we will address lightweight protection of Keccak's logic layer beyond the state of the art. This is required to increase the unified engine's reliability and security against faulting an arbitrary number of bits within a larger chip area.

Acknowledgments

This work was partially funded by the German Ministry of Education and Research (BMBF) via project RILKOSAN (16KISR010K).

References

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

[2] Aikata Aikata, Ahmet Can Mert, David Jacquemin, Amitabh Das, Donald Matthews, Santosh Ghosh, and Sujoy Sinha Roy. 2022. A unified cryptoprocessor for lattice-based signature and key-exchange. *IEEE Trans. Comput.* 72, 6 (2022), 1568–1580.

[3] Nuray At, Jean-Luc Beuchat, Eiji Okamoto, Ismail San, and Teppei Yamazaki. 2017. A low-area unified hardware architecture for the AES and the cryptographic hash function Grøstl. *J. Parallel and Distrib. Comput.* 106 (2017), 106–120.

[4] Nasour Bagheri, Navid Ghaedi, and Somitra Kumar Sanadhya. 2015. Differential Fault Analysis of SHA-3. In *Proceedings of the 16th International Conference on Progress in Cryptology – INDOCRYPT 2015 - Volume 9462*. Springer-Verlag, Berlin, Heidelberg, 253–269. doi:10.1007/978-3-319-26617-6_14

[5] Siavash Bayat-Sarmadi, Mehran Mozaffari-Kermani, and Arash Reyhani-Masoleh. 2014. Efficient and Concurrent Reliable Realization of the Secure Cryptographic SHA-3 Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 7 (2014), 1105–1109. doi:10.1109/TCAD.2014.2307002

[6] Daniel J Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. 2019. The SPHINCS+ signature framework. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2129–2146.

[7] Daniel J Bernstein and Tanja Lange. 2017. Post-quantum cryptography. *Nature* 549, 7671 (2017), 188–194.

[8] Alexandre Berzati, Cécile Canovas-Dumas, and Louis Goubin. 2012. A Survey of Differential Fault Analysis Against Classical RSA Implementations. In *Fault Analysis in Cryptography*, Marc Joye and Michael Tunstall (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–124. doi:10.1007/978-3-642-29656-7_7

[9] Jean-Luc Beuchat, Eiji Okamoto, and Teppei Yamazaki. 2011. A low-area unified hardware architecture for the AES and the cryptographic hash function ECHO. *Journal of Cryptographic Engineering* 1 (2011), 101–121.

[10] Adi Biham, Eliand Shamir. 1997. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology — CRYPTO '97*, Burton S. Kaliski (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 513–525.

[11] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. 2011. SPONGENT: A lightweight hash function. In *Cryptographic Hardware and Embedded Systems–CHES 2011: 13th International Workshop, Nara, Japan, September 28–October 1, 2011. Proceedings 13*. Springer, 312–325.

[12] Luděk Dudáček and Ivo Veřtát. 2016. Multidimensional Parity Check codes with short block lengths. In *2016 24th Telecommunications Forum (TELFOR)*. 1–4. doi:10.1109/TELFOR.2016.7818772

[13] Ivan Gavrilan, Felix Oberhansl, Alexander Wagner, Emanuele Strieder, and Andreas Zankl. 2024. Impeccable Keccak: Towards Fault Resilient SPHINCS+ Implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 2 (Mar. 2024), 154–189. doi:10.46586/tches.v2024.i2.154-189

[14] Christophe Giraud. 2012. Differential Fault Analysis of the Advanced Encryption Standard. In *Fault Analysis in Cryptography*, Marc Joye and Michael Tunstall (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72. doi:10.1007/978-3-642-29656-7_4

[15] Bertoni Guido, Daemen Joan, P Michaël, and VA Gilles. 2011. Cryptographic sponge functions.

[16] Bertoni Guido, Daemen Joan, P Michaël, and VA Gilles. 2011. The Keccak SHA-3 Submission - Round 3 submission to NIST SHA-3.

[17] Nick Howgrave-Graham, Joseph H Silverman, and William Whyte. 2005. Choosing parameter sets for NTRUEncrypt with NAEP and SVES-3. In *Cryptographers' Track at the RSA Conference*. Springer, 118–135.

[18] Ibrahim Jasim, Oguz Bayat, and Osman Ucan. 2019. Concatenation of polar codes with three-dimensional parity check (3D-PC) codes to improve error performance over fading channels. *International Journal of Communication Systems* 32 (05 2019), e3970. doi:10.1002/dac.3970

[19] Fatma Kahri, Hassen Mestiri, Belgacem Bouallegue, and Mohsen Machhout. 2017. Fault Attacks Resistant Architecture for KECCAK Hash Function. *International Journal of Advanced Computer Science and Applications* 8 (2017). https://api.semanticscholar.org/CorpusID:31594434

[20] John Kelsey, Shu-jen Chang, and Ray Perlner. 2016. SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. *NIST special publication* 800 (2016), 185.

[21] Ayesha Khalid, Arshad Aziz, Chenghua Wang, Máire O'Neill, Weiqiang Liu, et al. 2020. Resource-shared crypto-coprocessor of AES Enc/Dec with SHA-3. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67, 12 (2020), 4869–4882.

[22] Esam Khan, M Watheq El-Kharashi, Fayez Gebali, and Mostafa Abd-El-Barr. 2007. Design and performance analysis of a unified, reconfigurable HMAC-Hash unit. *IEEE Transactions on Circuits and Systems I: Regular Papers* 54, 12 (2007), 2683–2695.

[23] Pei Luo, Yunsi Fei, Liwei Zhang, and A. Adam Ding. 2016. Differential Fault Analysis of SHA3-224 and SHA3-256. Cryptology ePrint Archive, Paper 2016/709. https://eprint.iacr.org/2016/709 https://eprint.iacr.org/2016/709.

[24] Pei Luo, Cheng Li, and Yunsi Fei. 2016. Concurrent Error Detection for Reliable SHA-3 Design. In *Proceedings of the 26th Edition on Great Lakes Symposium on VLSI* (Boston, Massachusetts, USA) *(GLSVLSI '16)*. Association for Computing Machinery, New York, NY, USA, 39–44. doi:10.1145/2902961.2902985

[25] Hassen Mestiri and Imen Barraj. 2023. High-Speed Hardware Architecture Based on Error Detection for KECCAK. *Micromachines* 14, 6 (2023). doi:10.3390/mi14061129

[26] Hassen Mestiri, Imen Barraj, and Mohsen Machhout. 2021. Analysis and Detection of Errors in KECCAK Hardware Implementation. In *2021 IEEE International Conference on Design & Test of Integrated Micro & Nano-Systems (DTS)*. 1–6. doi:10.1109/DTS52014.2021.9497889

[27] Zeesha Mishra, Pallab Kumar Nath, and Bibhudendra Acharya. 2020. High throughput unified architecture of LEA algorithm for image encryption. *Microprocessors and Microsystems* 78 (2020), 103214.

[28] National Institute of Standards and Technology. 2015. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Federal Information Processing Standards Publication FIPS PUB 202. National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[29] NIST. 2018. National Institute of Standards and Technology: Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process,. Federal Information Processing Standards Publication. https://csrc.nist.gov/projects/lightweight-cryptography https://csrc.nist.gov/projects/lightweight-cryptography.

[30] NIST. 2024. Module-Lattice-Based Digital Signature Standard. https://doi.org/10.6028/NIST.FIPS.204

[31] NIST. 2024. Module-Lattice-Based Key-Encapsulation Mechanism Standard. https://doi.org/10.6028/NIST.FIPS.203

[32] NIST. 2024. Stateless Hash-Based Digital Signature Standard. https://doi.org/10.6028/NIST.FIPS.205

[33] Francisco Eugenio Potestad-Ordóñez, Erica Tena-Sánchez, Antonio José Acosta-Jiménez, Carlos Jesús Jiménez-Fernández, and Ricardo Chaves. 2022. Hardware countermeasures benchmarking against fault attacks. *Applied Sciences* 12, 5 (2022), 2443.

[34] PULP Platform [n. d.]. PULPissimo Github Page. https://github.com/pulp-platform/pulpissimo. Accessed: 06.04.2024.

[35] Antoon Purnal, Victor Arribas, and Lauren De Meyer. 2019. Trade-offs in Protecting Keccak Against Combined Side-Channel and Fault Attacks. In *Constructive Side-Channel Analysis and Secure Design*, Ilia Polian and Marc Stöttinger (Eds.). Springer International Publishing, Cham, 285–302.

[36] Enkele Rama, Mouadh Ayache, Rainer Buchty, Bernhard Bauer, Matthias Korb, Mladen Berekovic, and Saleh Mulhem. 2024. Trustworthy Integrated Circuits: From Safety to Security and Beyond. *IEEE Access* (2024).

[37] Sujoy Sinha Roy and Andrea Basso. 2020. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 443–466.

[38] Morris Rubinoff. 1961. N-Dimensional Codes for Detecting and Correcting Multiple Errors. *Commun. ACM* 4, 12 (dec 1961), 545–551. doi:10.1145/366853.366878

[39] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. 2018. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 1–3. doi:10.1109/S3S.2018.8640145

[40] Synopsys [n. d.]. Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html. Accessed: 06.04.2024.

[41] Alan Torres-Alvarado, Luis Alberto Morales-Rosales, Ignacio Algredo-Badillo, Francisco López-Huerta, Mariana Lobato-Báez, and Juan Carlos López-Pimentel. 2022. An SHA-3 Hardware Architecture against Failures Based on Hamming Codes and Triple Modular Redundancy. *Sensors* 22, 8 (2022). doi:10.3390/s22082985

[42] NC State University. 2024. FreePDK45™. https://eda.ncsu.edu/freepdk/freepdk45/ Accessed April 06, 2024.

[43] Ivo Vertat and Ludek Dudacek. 2019. Multidimensional Cross Parity Check Codes as a Promising Solution to CubeSat Low Data Rate Downlinks. In *2019 International Conference on Applied Electronics (AE)*. 1–5. doi:10.23919/AE.2019.8867037

[44] Wentao Xi, Xingjun Wu, and Kai Zhang. 2023. A low-cost configurable hash computing circuit for PQC algorithm. In *Proceedings of the 2023 13th International Conference on Communication and Network Security*. 112–116.

[45] Xilinx. [n. d.]. Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html. Accessed: 06.04.2024.

[46] Wei Zhao, Yi Wang, and Renfa Li. 2012. A unified architecture for DPA-resistant PRESENT. In *2012 International Conference on Innovations in Information Technology (IIT)*. IEEE, 244–248.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009