Recomputing with Permuted Operands: A Concurrent Error Detection Approach

Xiaofei Guo and Ramesh Karri

Polytechnic Institute of New York University
xg243@nyu.edu, rkarri@poly.edu

Abstract. Naturally occurring and maliciously injected faults reduce the reliability of cryptographic hardware and may leak confidential information. We develop a concurrent error detection (CED) technique called Recomputing with Permuted Operands (REPO). We show that it is cost effective in Advanced Encryption Standard (AES) and a secure hash function Grøstl. We provide experimental results and formal proofs to show that REPO detects all single-bit and single-byte faults. Experimental results show that REPO achieves close to 100% fault coverage for multiple byte faults. The hardware and throughput overheads are compared with those of previously reported CED techinques on two Xilinx Virtex FPGAs. The hardware overhead is 12.4-27.3%, and the throughput is 1.2-23Gbps, depending on the AES architecture, FPGA family, and detection latency. The performance overhead ranges from 10% to 100% depending on the security level. Moreover, the proposed technique can be integrated into various block cipher modes of operation. We also discuss the limitation of REPO and its potential vulnerabilities.

1 Introduction

Cryptographic primitives such as secret key encryptions and cryptographic hash functions are important in the security of data transmission in modern information systems. Secret key encryptions such as block ciphers and stream ciphers are widely used to provide data confidentiality [26]. A block cipher is a deterministic algorithm operating on groups of fixed-size bit strings, called blocks, with an unvarying transformation that is specified by a key. Block ciphers are important cryptographic primitives in the design of many cryptographic protocols and are widely used to encrypt data. A stream cipher exclusive-ors a plaintext bit with the corresponding keystream bit and generates the ciphertext bit. The National Institution of Standards and Technology (NIST) specifies Advanced Encryption Standard (AES) as the standard secret key encryption [33]. AES can also be used as stream ciphers such as LEX [**?**].

Cryptographic hash functions have many information security applications, notably in digital signatures, message authentication codes, and other forms of authentication [26]. They are used to index data in hash tables, for fingerprinting, to detect duplicate data or uniquely identify files, and as checksums to detect accidental data corruption. A cryptographic hash function takes an arbitrary bit string and returns a fixed-size hash value, so that an accidental or intentional change to the data will change the hash value with a very high probability. NIST started a secure hash function (SHA-3) competition in 2007. Several candidates in the competition utilize the AES-style structure as a building block such as ECHO, Grindahl, Grøstl, LANE, and SHAvite-3 [32]. Grøstl was selected in the final round competition after extensive assessment. Because Grøstl employs the AES-style structure, it makes all known, generic attacks on the hash function much more difficult.

As transistor size keeps scaling, increasing rates of faults [30], device variations [6], and aging [2] are posing serious challenges for VLSI designers. Faults that occur in VLSI chips are classified into two categories: transient faults, that eventually die away, and permanent faults. The origin of these faults could be internal phenomena in the system such as threshold changes, shorts, opens, etc., or external influences, such as electromagnetic radiation. These faults affect the memory as well as the combinational parts of a circuit and are detected using concurrent error detection (CED) [41].

Recently, differential fault analysis (DFA) has been proposed to extract secret keys in AES [3, 4, 13, 18, 27, 31, 35, 36, 40, 42] and Grøstl [11]. In DFA, the assumption is that the attacker is able to control the timing and the location of the fault, but has little influence on the actual fault value. This model is typical for laser injection, which is one of the most powerful tools for fault attacks. On the other hand, other means are more affordable and accessible, though less precise; clock glitches and drops in power supply are easier to set up and can inject exploitable faults as well. Using a completely general fault model, it was proven that the last round key can still be recovered.

To thwart fault injection attacks, two categories of countermeasures are proposed, i.e., *detection* and *infection*. Detection countermeasures detects faults in the execution of the hardware. If faults are detected, the hardware does not output the faulty ciphertexts. Therefore it prevents potential exploitation. Infection countermeasures prevent the exploitation of faulty ciphertexts by changing the logical effect of a fault in such a way that it significantly modifies the results. Therefore attackers cannot exploit the faulty ciphertexts [21].

NIST formulates security requirements for cryptographic modules in FIPS 140 [34]. It defines four levels of security. At security level four, the highest, it emphasizes that the physical security mechanisms must provide a complete envelope of protection around the cryptographic module with the intent of detecting and responding to all unauthorized attempts at physical access. Therefore, we focus on detection in this work.

1.1 Related Work

Previous CEDs can be classified into four types of redundancy: hardware, time, information, and hybrid redundancy.

Hardware redundancy duplicates the function and detects faults by comparing the outputs of two copies.

Time redundancy: The function is computed twice on the same input and the results are compared. A simple time redundancy technique for AES is proposed in [24]. The authors simply recompute the encryption after a normal computation, and results are compared. A variation of time redundancy called double-data-rate (DDR) is described in [23]. In this technique, the function is computed on both clock edges. This speeds up the computation. Under some conditions, this technique allows the encryption to be computed twice without affecting the global throughput. This technique becomes more difficult to implement as technology scales.

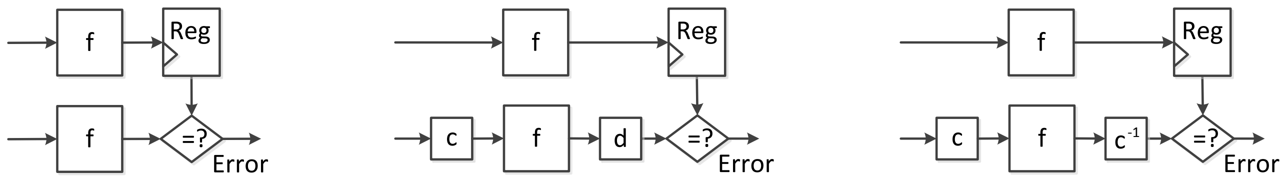
In [8], a variation of recomputing with shifted operands is proposed for AES. During the recomputation, each row is cyclically shifted before entering substitution-boxes (S-boxes), and the order is restored after S-boxes. This technique is quite effective because it uses different hardware to compute the data in the computation and recomputation. It detects both transient and permanent faults in S-boxes. However, it cannot detect permanent faults in other round operations.

Information redundancy: Many error detection techniques are based on error detecting codes. A few check bits are generated from the input message; then they propagate along with the input message and are finally validated when the output message is generated. In the basic parity technique [5], each predicted parity bit is generated from an input byte. Then, the predicted parity bits, and actual parity bits of output are compared to detect the faults. This technique incurs large hardware overhead. In another technique, only one bit parity is used for the entire 128-bit output, and the parity bit is checked once for the round [43]. However, these techniques only apply to lookup table-based (LUTs) S-box implementation. In [28], parity is obtained for S-box implementation using logic gates. In [22], a general parity-based technique is proposed to protect the S-box regardless of its implementation. This parity technique is later extended to Grøstl [29]. All these parity techniques share the same limitation. If an even number of faults occurs in the same byte, none of these techniques detect them. To address this limitation, systematic robust code is proposed [16, 19]. It provides uniform fault coverage. The key idea is to construct a prediction circuit at the round input to predict the nonlinear property of the round output. While this technique provides high fault coverage compared to parity code, the hardware overhead of this technique is 77%. Another approach uses cyclic redundancy check codes over $GF(2^8)$ [9]. It detects all odd number of faults.

Hybrid redundancy: In [17], the authors consider CED at the operation, round, and algorithm levels. In these techniques, an operation, a round, or the entire encryption and decryption are followed by their inverses, and the results are compared with the original input. Although these techniques detect most faults, they require both encryption and decryption to be on chip and can suffer from more than 100% throughput overhead. An improvement of this idea is proposed to merge the encryption and decryption datapath to achieve small performance and area overhead [39]. In this technique, both encryption and decryption are deeply pipelined to increase clock frequency. In encryption, each stage will perform a function in one clock cycle, and the inverse function in the next clock cycle. The authors optimize the area by sharing hardware between functions and their inverse. In [37], the authors propose a technique that computes data on different pipeline stages. However, this technique is only applicable to pipeline architecture.

1.2 Contributions

We propose a low-overhead, implementation independent time redundancy CED technique called Recomputing with Permuted Operands (REPO), which provides secure and reliable implementation for AES with 128-bit datapath and Grøstl with 512-bit datapath. The technique achieves a fault detection capability that is close to [17] and hardware redundancy, the state-of-the-art countermeasures, but with much lower cost. Our contributions are as follows:



(a) (b) (c)

Fig. 1: (a) The concept of time redundancy (b) Variation of time redundancy. (c) Time redundancy with inverse functions.

- We present REPO for AES-style functions and we generalize REPO for ALU and other logic operations.
- We reduce the performance overhead of REPO with adaptive checking and Randomized CED Round Insertion from 100% to 50%.
- We prove that REPO detects all single-bit and single-byte faults.
- REPO achieves an order of 10^5 lower fault miss rate than the best parity techniques for multiple burst and multiple random faults.
- We apply REPO to AES-inspired hash function Grøstl, and it achieves even higher fault coverage.

This paper is organized as follows: In Section 3.1, we introduce the AES algorithm. In Section 3.2, we explain the key idea of the proposed REPO technique, and we show the fault coverage, hardware overhead, and detailed analyses of the technique. We also compare the fault coverage of different invariances. In Section 4, we extend the idea to the AES-inspired Grøstl hash function and analyze the scalability of our technique. Section 6 concludes the paper.

2 Recomputing with Permuted Operands

In this section, we introduce REPO and how it is a generalization of recomputing with shifted operands (RESO) and recomputing with rotated operands (RERO).

As shown in Fig. 1(a), let x be the input to a computation unit f and $f(x)$ be the output. In time redundancy, $f(x)$ is computed twice. The result of the first computation is stored in a register and then compared with the result of the second computation. The drawback of this technique is that faults are detected only if they do not affect both of the computations. Therefore, permanent faults and transient faults that affect both of the computations will not be detected.

To solve this problem, one can build two functions c and d, such that $d(f(c(x))) = f(x)$ as shown in Fig. 1(b). We first compute $f(x)$ and then $d(f(c(x)))$. If c and d are properly chosen, then a fault in unit f will affect $f(x)$ and $d(f(c(x)))$ in a different way. Therefore, the outputs of the two computations will not match. When the functions c and d are inverses of each other, i.e., $d(c(x)) = x$ for all x, we have $c^{-1}(f(c(x))) = f(x)$ as illustrated in Fig. 1(c). Fig. 1 is the basis of time redundancy techniques. Function c can be any bijection function including shifting, rotation, permutation, etc.

Definition: A *permutation* of a set S is defined as a bijection from S to itself. This is related to the rearrangement of S in which each element s takes the place of the corresponding $f(s)$. The collection of such permutations forms a symmetric group.

The key to its structure is the possibility of composing permutations: performing two given rearrangements in succession defines a third rearrangement, the composition. Permutations may act on composite objects by rearranging their components, or by certain replacements (substitutions) of symbols. If S is a finite set of n elements, then there are $n!$ permutations. Therefore, permutation includes cyclical shift, i.e., rotation. Because shifting and rotation are specials form of permutation, REPO is a generalization of RESO and RERO.

The REPO architecture is shown in Fig. 2. Let X and Y be the inputs of function f. First, X and Y are computed by f without any permutation. The result is stored in a register. Second, X and Y are both permuted and then computed by f. The output of f is inverse permuted and compared with the previous result stored in the register. If the results are not equal, faults are detected. In the next two sections, we apply REPO to AES and Grøstl, respectively.

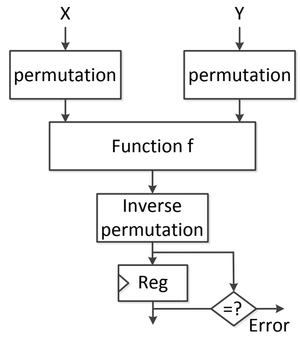


Fig. 2: REPO Architecture

3 REPO for AES

In this section, we will introduce the AES algorithm and how REPO is used for AES.

3.1 Advanced Encryption Standard

In this paper, we consider 128-bit AES as specified by NIST [33]. AES encrypts a 128-bit plaintext into a 128-bit ciphertext with a user key using 10 nearly identical rounds plus an initial special round (round 0). One AES encryption round consists of SubBytes, ShiftRows, MixColumns, and AddRoundKey, denoted by B, S, M, and A, respectively, as shown in Fig. 3. In round 0, only AddRoundKey is performed and in round 10, MixColumns is not performed. Each operation in every round acts on a 128-bit input *state*, where each state element is a byte in $GF(2^8)$. In this paper, each byte is denoted by $\boldsymbol{s_{r,c}}$ $(0 \leq r, c \leq 3)$, and it indicates that this byte is in row $\boldsymbol{r}$ and column $\boldsymbol{c}$ in state matrix.

$$S = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = [s_{r,c}]_{r,c=0..3} \tag{1}$$

In SubBytes, all the bytes are processed separately by 16 S-boxes (SBs). Each S-box performs a nonlinear transformation of the input byte. Let X be the input to the SBs. The resulting output is:

$$Y = B(X) = [y_{r,c}]_{r,c=0..3} \tag{2}$$

In ShiftRows, the rows of the state are shifted cyclically byte-wise using a different offset for each row. Row 0 is not shifted, while rows 1, 2, and 3 are cyclically shifted to the left by 1 byte, 2 bytes, and 3 bytes, respectively. The resulting output is:

$$Z = S(Y) = \begin{bmatrix} y_{0,0} & y_{0,1} & y_{0,2} & y_{0,3} \\ y_{1,1} & y_{1,2} & y_{1,3} & y_{1,0} \\ y_{2,2} & y_{2,3} & y_{2,0} & y_{2,1} \\ y_{3,3} & y_{3,0} & y_{3,1} & y_{3,2} \end{bmatrix}$$

$$= [y_{r,(r+c)\ mod\ 4}]_{r,c=0..3} = [z_{r,c}]_{r,c=0..3} \tag{3}$$

In MixColumns, the output state is obtained by multiplying a constant matrix with the output of ShiftRows. The resulting output is:

$$U = M(Z) = [u_{r,c}]_{r,c=0..3}$$

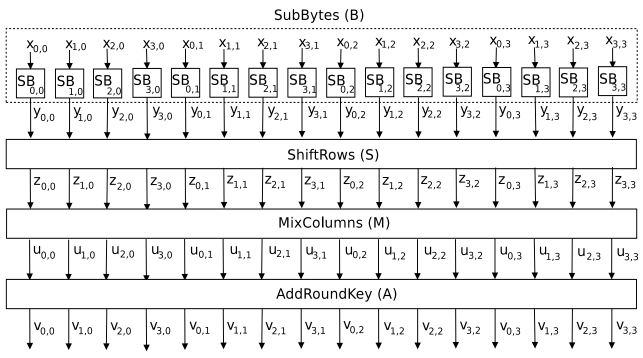


Fig. 3: One AES encryption round (The last round does not have MixColumns)

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} z_{0,0} & z_{0,1} & z_{0,2} & z_{0,3} \\ z_{1,0} & z_{1,1} & z_{1,2} & z_{1,3} \\ z_{2,0} & z_{2,1} & z_{2,2} & z_{2,3} \\ z_{3,0} & z_{3,1} & z_{3,2} & z_{3,3} \end{bmatrix} \tag{4}$$

In AddRoundKey, the input state is added (modulo-2) to the round key, i.e., the 128-bit state U with matrix $K = [k_{r,c}]_{r,c=0..3}$. The resulting output is:

$$V = A(K, U) = [u_{r,c}]_{r,c=0..3} + [k_{r,c}]_{r,c=0..3} = [v_{r,c}]_{r,c=0..3} \tag{5}$$

The S-box of AES is composed as a multiplicative inversion in $GF(2^8)$ modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, followed by an affine transformation.

3.2 An invariance of AES

In [20], the authors have shown that AES exhibits various round and mapping invariances[1]. Up till now, invariances in cryptographic algorithms have been identified by cryptanalysts. Because regularity of algorithm components can lead to new cryptanalytic insights and approaches, the authors were investigating these as possible source of weakness in AES. In contrast, we use these invariances for CED to protect the AES against random faults and malicious attacks by checking these properties [15]. We analyze three round level invariances of AES. We give a formal proof of the invariance property α_1, which is the most effective invariance according to our experimental results. We analyze the effectiveness of the remaining invariances in Section 3.7.

Theorem 1. *An AES round can be represented as*

$$A(K, M(S(B(X))))$$

where X is the 128-bit input to the round. Byte permutation α exists such that the following holds true:

$$A(K, M(S(B(X)))) = \alpha^{-1}(A(\alpha(K), M(S(B(\alpha(X)))))) \tag{6}$$

where α^{-1} denotes the inverse function of α.

One of the byte permutation is:

$$\alpha_1(X) = \alpha_1([x_{r,c}]_{r,c=0..3}) = \begin{bmatrix} x_{0,3} & x_{0,0} & x_{0,1} & x_{0,2} \\ x_{1,3} & x_{1,0} & x_{1,1} & x_{1,2} \\ x_{2,3} & x_{2,0} & x_{2,1} & x_{2,2} \\ x_{3,3} & x_{3,0} & x_{3,1} & x_{3,2} \end{bmatrix}$$

[1] Let $f : \{0,1\}^{128} \rightarrow \{0,1\}^{128}$ denotes an operation on the state space of AES which operates completely on the Galois field $GF(2^8)$. A property $P \subseteq \{0,1\}^{128}$, $P \neq 0$, is called an invariance of f, if P is preserved by f, i.e., for every $x \in P$ it follows that $f(x) \in P$. [20]

$$= [x_{r,(c+3)\ mod\ 4}]_{r,c=0..3} \tag{7}$$

$$\alpha_1^{-1}([x_{r,(c+3)\ mod\ 4}]_{r,c=0..3}) = [x_{r,c}]_{r,c=0..3} \tag{8}$$

Proof. Let us start from the right-hand side of the equation. First, we apply permuted input $X^{'} = \alpha_1(X)$ to SubBytes, and from (2) and (7), we get:

$$Y^{'} = [y^{'}_{r,c}]_{r,c=0..3} = [y_{r,(c+3)\ mod\ 4}]_{r,c=0..3} = \alpha_1(Y) \tag{9}$$

Given $Y^{'}$ as the input to ShiftRows, we get:

$$Z^{'} = S(Y^{'}) = \begin{bmatrix} y^{'}_{0,0} & y^{'}_{0,1} & y^{'}_{0,2} & y^{'}_{0,3} \\ y^{'}_{1,1} & y^{'}_{1,2} & y^{'}_{1,3} & y^{'}_{1,0} \\ y^{'}_{2,2} & y^{'}_{2,3} & y^{'}_{2,0} & y^{'}_{2,1} \\ y^{'}_{3,3} & y^{'}_{3,0} & y^{'}_{3,1} & y^{'}_{3,2} \end{bmatrix}$$

$$= \begin{bmatrix} y_{0,3} & y_{0,0} & y_{0,1} & y_{0,2} \\ y_{1,0} & y_{1,1} & y_{1,2} & y_{1,3} \\ y_{2,1} & y_{2,2} & y_{2,3} & y_{2,0} \\ y_{3,2} & y_{3,3} & y_{3,0} & y_{3,1} \end{bmatrix} = [z^{'}_{r,c}]_{r,c=0..3} \tag{10}$$

From (3) and (10), we find that:

$$Z^{'} = [z_{r,(c+3)\ mod\ 4}]_{r,c=0..3} = \alpha_1(Z) \tag{11}$$

Applying $Z^{'}$ to MixColumns, we get:

$$U^{'} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} z_{0,3} & z_{0,0} & z_{0,1} & z_{0,2} \\ z_{1,3} & z_{1,0} & z_{1,1} & z_{1,2} \\ z_{2,3} & z_{2,0} & z_{2,1} & z_{2,2} \\ z_{3,3} & z_{3,0} & z_{3,1} & z_{3,2} \end{bmatrix}$$

$$= [u_{r,(c+3)\ mod\ 4}]_{r,c=0..3} = \alpha_1(U) \tag{12}$$

Then we apply the permuted round key matrix $K^{'} = \alpha_1(K)$ resulting in:

$$V^{'} = [u^{'}_{r,c}]_{r,c=0..3} + [k^{'}_{r,c}]_{r,c=0..3}$$

$$= [u_{r,(c+3)\ mod\ 4}]_{r,c=0..3} + [k_{r,(c+3)\ mod\ 4}]_{r,c=0..3} = \alpha_1(V) \tag{13}$$

We apply inverse permutation α_1^{-1} to the output. We get:

$$\alpha_1^{-1}(V') = \alpha_1^{-1}(\alpha_1(V)) = V \tag{14}$$

3.3 REPO Architectures

We design two REPO architectures for AES: Fully pipelined and iterative.

Fully pipelined: There are 11 stages in our pipelined architecture. For each pipeline stage in Fig. 4(a), we add two muxes (mux_x and mux_k) and a comparator (cmp). α is a permutation of wires based on the invariance property. α^{-1} is the inverse permutation. We need two encryption cycles to detect the faults; let us call them C1 and C2. In C1, let the input and the key of round 1 be $X1$ and $K1$. We run the encryption with mux_x and mux_k, selecting $X1$ and $K1$ as the inputs. The round result $V1$ is stored in the data register. Then we run C2; we run the encryption with mux_x and mux_k, selecting permuted inputs $X1^{'}$ and $K1^{'}$, respectively. At the end of C2, we inverse permute output $V1^{'}$ and compare it with the value $V1$ stored in the data register. If the results are equal, no fault is detected. Otherwise, the comparator will assert the fault indication flag. The comparator does not add delay to the critical path because the comparison can be performed when the next round input is executed. However, to prevent the attacker from obtaining the faulty output, the comparison should be done in the same round. This may increase the delay of the critical path. We see that C1 can be any normal encryption

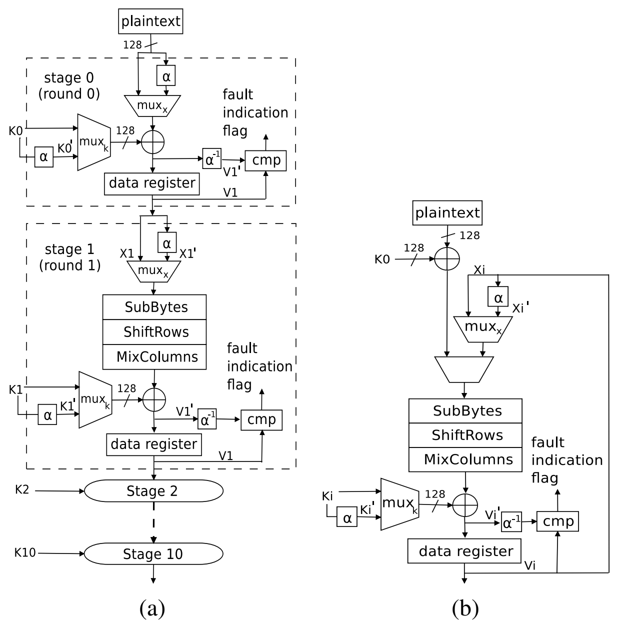


Fig. 4: AES hardware architectures (a) Fully pipelined. (b) Iterative. cmp stands for comparator. We assume that the comparator is fault tolerant.

cycle, and C2 is the corresponding extra cycle, which selects the permuted inputs to be performed after every C1. One can add several redundant C2 every encryption; we call R the checking ratio which is the total number of rounds (10) divided by the number of redundant rounds. R can be changed based on the tradeoff between performance, reliability, and security specified by the designer. For a detailed analysis, please see Section 3.5.

Iterative: As shown in Fig. 4(b), we add mux_x and mux_k and a comparator. There is one security benefit of iterative implementation. In an iterative architecture, each ciphertext takes 10 cycles to generate. The designer only needs to check specific rounds to enhance security and the details are discussed in Section 3.5. As long as the faults are detected before the ciphertext is generated, the faulty ciphertext will not be sent to the output. This will prevent an attacker from stealing the secret key. In the fully pipelined architecture, a ciphertext is generated every cycle. Therefore, if the faults are generated before the comparison, faulty outputs will be obtained by the attacker.

3.4 Fault Analysis

REPO detects all single-bit and single-byte faults. It is noted that single-byte fault is the most commonly used and practical fault model in fault attacks [35,36]. Our simulations show that this CED technique detects 99.99999997% of multiple burst faults and 100% random faults. Fault coverage (FC) is calculated as:

$$FC = 1 - FMR$$

where FMR is the fault miss rate calculated as:

$$FMR = \frac{T_{undetected}}{T_{total} - T_{correct}}$$

where $T_{undetected}$ is the number of tests in which faults are excited but not detected. T_{total} is the total number of tests we applied. $T_{correct}$ represents the tests in which the faults are not excited. Because single-bit fault is a special case of single-byte fault, we are going to prove the 100% fault coverage for single-byte fault.

Theorem 2. *If a single-byte fault in any of the steps in a round affects the outputs of the final result of that round, REPO will detect it.*

Proof. **Case 1: A single-byte fault in S-box (SB)**. In Fig. 3, let the $SB_{i,j}$ $(0 \leq i, j \leq 3)$ have a single-byte fault. If the SBs are implemented using ROMs, the considered fault corresponds to an address fault of the ROM, a fault in memory location, or a fault in the output data lines. If the SBs are implemented using combinational logic, the considered fault can appear in any gate of the implementation.

In C1, the $SB_{i,j}$ generates faulty output $y_{i,j}$. After ShiftRows, the outputs are $[z_{r,c}]_{r,c=0..3}$ = $[y_{r,(r+c)\ mod\ 4}]_{r,c=0..3}$, and the faulty state element is $z_{i,(j-i)\ mod\ 4} = y_{i,j}$. In MixColumns, a single faulty input causes four bytes within the same column to be faulty. The faulty state elements are represented as $[u_{r,(j-i)\ mod\ 4}]_{r=0..3}$. After AddRoundKey, $[v_{r,(j-i)\ mod\ 4}]_{r=0..3}$ are the faulty state elements. In C2, we apply $X^{'}$ and $K^{'}$ as the permuted inputs. Using the same steps shown above, faulty state elements are represented as $[v^{'}_{r,(j-i)\ mod\ 4}]_{r=0..3}$. From (8), we know that

$$[v^{'}_{r,(j-i)\ mod\ 4}]_{r=0..3} = [v_{r,((j-i)+3\ mod\ 4)\ mod\ 4}]_{r=0..3}$$

$$[v_{r,c}]^{3}_{r,c=0} = P^{-1}([v^{'}_{r,c}]^{3}_{r,c=0}) = P^{-1}([v_{r,(c+3)\ mod\ 4}]^{3}_{r,c=0}) \tag{15}$$

Therefore, the faulty column in C1 is $(j-i)\ mod\ 4$, but the faulty column in C2 corresponds to column $((j-i)\ mod\ 4) + 3)\ mod\ 4$ in C1; note that $0 \leq i, j \leq 3$ and $(j-i)\ mod\ 4 \neq (((j-i)\ mod\ 4) + 3)\ mod\ 4$.

Because faulty $SB_{i,j}$ affects different columns in C1 and C2, we always compare a faulty column with a fault-free column, and REPO detects the fault as long as it affects the output. Even if the fault does not affect the output of the check round, the outputs are still different. For a concrete example, let $SB_{1,2}$ be faulty, thus, $y_{1,2}$ is the faulty output byte. After ShiftRows, $z_{1,1} = y_{1,2}$. After MixColumns, the faulty state elements are shown as $[u_{r,1}]_{r=0..3}$. Then we apply this as the input of AddRoundKey, so faulty state elements are shown as $[v_{r,1}]_{r=0..3}$. Then we run C2, and faulty state elements are represented as $[v^{'}_{r,1}]_{r=0..3}$. Because $[v^{'}_{r,1}]_{r=0..3} = [v_{r,0}]_{r=0..3}$, the faulty columns in C1 and C2 are different, and we detect the fault by comparing the outputs.

Case 2: A single-byte fault in ShiftRows. A fault in ShiftRows is equivalent to a fault at the input of MixColumns, because the ShiftRows on the FPGA and ASIC implementations are only wiring. Thus, we prove this in case 3.

Case 3: A single-byte fault in MixColumns. Because MixColumns is mainly implemented with XOR and a few other basic gates, we consider a fault in MixColumns in three scenarios: the input, the internal logic gates, and the output. If there is a fault in the input, the fault will propagate to all four bytes in the same column. Assuming that column $[u_{r,j}]_{r=0..3}$ is faulty in C1, we know that the column $[v_{r,j}]_{r=0..3}$ is faulty in the final output of the current round. In C2, we know that the column $[u^{'}_{r,j}]_{r=0..3}$ is faulty, and the column $[v^{'}_{r,j}]_{r=0..3}$ of the output of the round is also faulty. From (12) and (14), we know that $j^{'} = (j+3)\ mod\ 4$. Because the faulty columns of the two outputs are different, we detect the fault. If there is a fault in the output, we detect the fault because the fault will affect a different column in C1 and C2. If there is a fault in the internal gates, we also detect the fault. Because the circuits for the four columns are separate, the fault again will affect only one column in C1 and a different column in C2.

Case 4: A single-byte fault in AddRoundKey. AddRoundKey is mainly implemented as bit-wise XOR gates. We consider a fault in AddRoundKey as fault in the input or output. The fault at the input is equivalent to the fault in the output of MixColumns. Let us prove the theorem true for a single-byte fault at the output of AddRoundKey. Let the faulty byte be $v_{i,j}$ in the C1 and $v^{'}_{i,j}$ in C2. From (14), we know that $v^{'}_{i,j} = v_{i,(j+3)\ mod\ 4}$. Again, the faulty columns in C1 and C2 are different, and thus we detect the fault.

To achieve a 100% single-byte fault coverage, one needs hardware redundancy or hybrid redundancy [17], both of which have more than 100% hardware overhead. Our fault simulation confirms that REPO detects all single-bit and single-byte faults.

Fault Coverage for Multiple Byte Faults We simulated multiple byte faults for REPO and compared the fault coverage with the one proposed in [22]. These models cover both natural faults and fault attacks [7]. Due to technology constraints, an attacker may not be able to inject a single-bit fault [7]. Multiple byte faults are injected in the process. We use burst and random fault models [7].

Burst faults occur at the 128-bit input or output of only one operation at a time. These are transient faults. We use Fibonacci Linear Feedback Shift Register (LFSR) with 128-bit output taps to inject faults. The maximum sequence length polynomial for the LFSR is selected as $L(X) = X^{128} + X^{29} + X^{27} + X^{2} + 1$. The number of faults in each fault injection

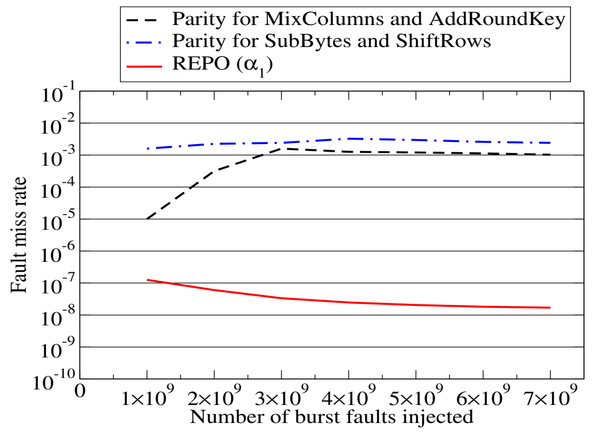


Fig. 5: Simulation results show that the fault miss rate of REPO is superior to that of the parity technique [22] with multiple byte burst fault model.

is determined by the LFSR. For each injection, the LFSR will generate 128 bit value, corresponding to 128 bit datapath in AES. If an LFSR bit value is 1, the simulator flips the corresponding AES bit. The location are determined by the random number generated from Boost C++ library [1].

The simulation results for burst faults are shown in Fig. 5. We compare our miss rate with [22]. This technique has two stages for their parity comparison. The first stage is parity for SubBytes and ShiftRows, and faults are injected only in the two operations. The second stage is parity for MixColumns and AddRoundKey, and similarly, faults are injected only in these two operations. REPO is implemented for the entire round. The dot-dash line respresents the fault miss rate of SubBytes and ShiftRows of [22]. The dash line represents the fault miss rate of MixColumns and AddRoundKey of [22]. The solid line represents the fault miss rate of REPO. We have injected up to 7×10^9 burst faults at the operation outputs and monitored detected faults. We count the number of faults being injected as shown in the x-axis in Fig. 5. For each fault injection, we run one AES round with 10^4 different test vectors. There are around 10^8 fault injections. All together, there are $10^4 \times 10^8 = 10^{12}$ tests. In around 3×10^4 tests, REPO did not detect the faults. The fault coverage is $1 - \frac{3\times10^4}{10^{12}} = 99.99999997\%$. The fault miss rate for [22] is between 10^{-2} and 10^{-3}. The miss rate of our scheme is between 10^{-7} and 10^{-8}; a reduction of 10^5. The fault coverage of REPO will still be very high for each stage because each data are computed by different hardware units during the computation and recomputation.

Random faults are injected at random locations, i.e., the 128-bit inputs or outputs of the operations. The size, location, and type of faults are determined by the random number generated from Boost C++ library [1]. In another simulation, we observe 100% fault coverage after injecting up to 7×10^9 random faults. The reason why the fault coverage is 100% is because the undetectable fault space is too small compared to the entire fault space. It is difficult to encounter such cases in the simulation. We discuss the undetectable fault in Section 5.2. Our fault model is similar to the one used in [7,22]. In this model, faults occur in the input or output wires of each operation. Because the AES is implemented on the FPGA, in the physical fault model, the S-boxes are implemented as slice LUTs and registers are implemented as slice registers. Faults in slice LUTs and slice registers are equivalent to the ones in the output wires. ShiftRows only uses wires. Faults on the wire or the input/output of a gate are equivalent to faults in the input/output wire. AddRoundKey uses one level of gates. For MixColumns, we consider single fault and multiple byte faults in the input or output wires. Thus, the physical fault model matches the simulation fault model.

Comparison of Fault Coverage As shown in Table 1, for single-bit and single-byte faults, REPO provides 100% fault coverage, the same as [17] and hardware redundancy. It is noted that [17] requires both encryption/decryption to be on chip to achieve such fault coverage. While most parity schemes achieve 100% fault coverage for single-bit fault, they can only provide 50% fault coverage for single-byte fault [28]. For multiple byte faults, [17] and hardware redundancy provide 100% fault coverage. REPO provides 99.99999997% fault coverage, and much higher than parity-based schemes. The

Table 1: Comparison of fault coverage. a. burst faults b. random faults

FDS	Fault coverage		
	Single bit	Single byte	Multiple bit
HW red.	100%	100%	100%
Parity 1 [5]	100%	50%	99.997%
Parity 2 [43]	98.7%	50%	48-53%
Parity 3 [22]	100%	50%	99.996%
Hybrid Red. [17]	100%	100%	100%
REPO	**100%**	**100%**	**99.99999997%** [a] **100%** [b]

tradeoff between performance and detection latency can be explored by varying the checking ratio R, which is the ratio of the number of results computed without invariance to the number of results computed with REPO.

3.5 Security Analysis

We propose four CED checking policies that the designers can employ for their design depending on the implementation details as well as security and reliability requirements. They are *Security without Decryption, Security with Decryption, Security without Decryption + Additional Reliability, and Security + Reliability.* A technique called Randomized CED Round Insertion (RCRI) is also proposed to reduce the performance overhead while maintaining reliability.

Security without Decryption (REPO-S) Our investigation into all previous DFA shows that the attack is only able to utilize faults that are injected into the last four rounds, i.e, from the 7^{th} to 10^{th} rounds. By injecting faults into the last four rounds, the attacker is able to reduce the time to brute force the key dramatically, i.e., the brute force complexity will be in the range of 2^8 to 2^{32}. However, if the attacker inject faults into other rounds, the brute force complexity will be 2^{128} which is computationally infeasible to get the key with the current computer. Therefore, it is more secure to only check the last four rounds against a computationally bounded attacker. This will reduce the 100% performance overhead to 40%. The checking ratios in this case is 10/4 = 2.5.

Security with Decryption (REPO-SD) In the DFA community, the general assumption is that the attacker only needs to know the ciphertexts (faulty and fault-free). DFA is powerful because attacker does not need to know the plaintext. However, for some implementations, an attacker can access the decryption module, ask for decryption of arbitrary message, and obtain the decryption output. With this extra capability, attacker can inject faults in the first four rounds of the encryption, then ask the decryption unit to decrypt the faulty ciphertext and obtain a faulty plaintext. The effect is similar to injecting a fault in the last 4 rounds of decryption. Although the attacker inject faults in the encryption unit, the attacker can launch DFA for the decryption unit with the fault-free and faulty plaintexts. In this case, the first and last 4 rounds of the encryption and decryption all needs to be protected. Therefore, eight rounds needs to be protected. The checking ratio will be 10/8 = 1.25.

Security without Decryption + Additional Reliability (REPO-SAR) For reliability purposes, one still needs to check the first 6 rounds. Therefore, we propose RCRI.

In RCRI, the positions of the CED rounds C1 and C2 are randomized during the 10 round AES encryption process for iterative architecture. The check round C2 is only inserted into the first 6 rounds. This can be implemented as shown in the state diagram of Fig. 6. A random number can be obtained using the randomness property of the AES algorithm.

For example, a Rand register can be incorporated into the circuit with some random number stored in it at manufacture time and for every subsequent encryption performed, the resulting ciphertext is exclusive-ored with Rand to get a Temp number. When an encryption is performed, the algorithm enters the normal execution state. Normal encryption rounds are performed until the value of the Temp modulo 6 equals the round number. Once this condition is satisfied, the CED round C1 is performed. Depending on whether 6 normal rounds have been performed, either C2 or the remaining normal rounds are performed. The encryption process is complete when 6 normal rounds and the randomly inserted C1 and C2

Table 2: Comparisons of implementation of CED schemes on two Xilinx FPGAs. We use the metrics, FPGA platform, and results from [22]. Our pipeline implementation are shown in bold, and we implement the iterative architectures. a. the latency is 2x the original AES encryption/decryption b. using two (256 $\times$ 9) ditributed memories for CED of each S-box or inverse S-box c. using (256 $\times$ 9) distributed memories for CED of each S-box or inverse S-box d. checking ratio is 1 e. checking ratio is 2.5 f. checking ratio is 1.25 g. checking ratio is 2 h. This is implemented on xc2v1000 using an architecture with 4 S-boxes i. This is implemented in composite field with encryption and decryption sharing the logic. So the implementation numbers for the encryption and decryption are the same. j. The unit is K gates k. The unit is Kbps/gate

FPGA (Model)	Arch.	Scheme	Encryption				Decryption			
			Slice (overhead)	Freq. (MHz)	Thro. (Gbps)	Eff(Mbps /slice)	Slice (overhead)	Freq. (MHz)	Thro. (Gbps)	Eff.(Mbps /slice)
$Virtex$-4 (xc4vlx160-12)	Pipe.	Original	18335(-)	240.5	30.8	1.7	19322(-)	203.5	26.0	1.3
		HW Red.	36684(100.1%)	240.5	30.8	1.1	38658(100.1%)	203.5	26.0	0.7
		Parity 1 [5][b]	39104(113.3%)	163.5	20.9	0.5	40244(108.3%)	145.4	18.6	0.5
		Parity 2 [43][c]	21211(15.7%)	240.5	30.8	1.4	22280(15.3%)	203.5	26.0	1.1
		Parity 3 [22]	20127(9.8%)	240.5	30.8	1.5	20909(8.2%)	203.5	26.0	1.2
		Hybrid Red. [17]	38273(108.7% [22]) (1.6%)	194.9	24.9[a]	0.6	38273(98.1% [22]) (1.6%)	194.9	24.9[a]	0.6
		REPO-SR	**21253(15.9%)**	**232.3**	**14.9[d]**	**0.7[d]**	**22240(15.1%)**	**197.6**	**12.6[d]**	**0.6[d]**
	Iter.	Original	1905(-)	224.6	2.9	1.5	2002(-)	192.0	2.5	1.2
		REPO-S	**2156(13.2%)**	**217.4**	**2[e]**	**0.9[e]**	**2253(12.5%)**	**186.7**	**1.7[e]**	**0.8[e]**
		REPO-SD	**2161(13.4%)**	**217.4**	**1.6[f]**	**0.7[f]**	**2258(12.9%)**	**186.7**	**1.3[f]**	**0.6[f]**
		REPO-SAR	**2170(13.9%)**	**217.4**	**1.9[g]**	**0.9[g]**	**2267(13.2%)**	**186.7**	**1.6[g]**	**0.8[g]**
		REPO-SR	**2154(13.1%)**	**217.4**	**1.4[d]**	**0.7[d]**	**2251(12.4%)**	**186.7**	**1.2[d]**	**0.5[d]**
$Virtex$-II	Pipe.[h]	RESO [8]	(2.7%/-2.4%)	-	-	-	-	-	-	-
90nm ASIC	Iter.[i]	Hybird [39]	16.1K (24.8%)[j]	362.32	2.21	137.18[k]	16.1K (24.8%)[j]	362.32	2.21	137.18[k]
$Virtex$-5 (xc5vlx110-3)	Pipe.	Original	2960(-)	371.7	47.6	16.1	3906(-)	296.3	37.9	9.7
		HW Red.	5934(100.5%)	371.7	47.6	10.2	7826(100.4%)	296.3	37.9	5.5
		RESO [8]	(2.7%/-2.4%)	-	-	-	-	-	-	-
		Parity 1 [5][b]	5590(88.9%)	282.8	36.2	6.5	6680(71.2%)	260.2	33.3	4.9
		Parity 2 [43][c]	3619(22.3%)	304.0	38.9	10.7	4426(13.3%)	277.0	35.5	8.0
		Parity 3 [22]	3757(26.9%)	371.7	47.6	12.7	4286(9.7%)	296.3	37.9	8.8
		Hybrid Red. [17]	5849(97.6% [22]) (-14.8%)	284.4	36.4[a]	6.2	5849(49.7% [22]) (-14.8%)	284.4	36.4	6.2
		REPO-SR	**3664(23.9%)**	**358.9**	**23.0[d]**	**6.6[d]**	**4434(13.5%)**	**288.9**	**18.5[d]**	**4.2[d]**
	Iter.	Original	344(-)	347.0	4.4	12.8	462(-)	286.4	3.7	7.9
		REPO-S	**433(25.9%)**	**335.8**	**3.1[e]**	**7.2[e]**	**534(15.6%)**	**273.1**	**2.6[e]**	**4.9[e]**
		REPO-SD	**434(26.2%)**	**335.8**	**2.4[f]**	**5.6[f]**	**535(15.8%)**	**273.1**	**2.0[f]**	**3.7[f]**
		REPO-SAR	**438(27.3%)**	**335.8**	**2.9[g]**	**5.9[g]**	**539(16.7%)**	**273.1**	**2.4[g]**	**4.9[g]**
		REPO-SR	**432(25.6%)**	**335.8**	**2.2[d]**	**5.1[d]**	**533(15.4%)**	**273.1**	**1.8[d]**	**3.4[d]**
$Virtex$-II	Pipe.[h]	RESO [8]	(2.7%/-2.4%)	-	-	-	-	-	-	-
90nm ASIC	Iter.[i]	Hybird [39]	16.1K (24.8%)[j]	362.32	2.21	137.18[k]	16.1K (24.8%)[j]	362.32	2.21	137.18[k]

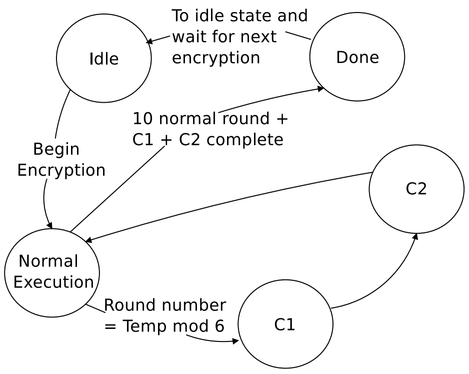

To idle state and wait for next encryption
Idle
Done
10 normal round + C1 + C2 complete
Begin Encryption
C2
Normal Execution
Round number = Temp mod 6
C1

Fig. 6: State machine for RCRI

CED round are complete. For the mod operation, we can take the last 4 bits of Temp and apply it to a lookup table which contains modulo 6 results from input 0 to 15. For the pipeline architecture, this method creates unbalance load between pipeline stages. Therefore, we insert the check round after every normal round.

If $R = 1$, all rounds are checked. The fault miss rate will remain the same for permanent and transient faults. If $R > 1$ ($R \leq 5$), every R^{th} result will be checked. Let us assume the transient faults appear for N cycles. When $R \leq N$, the fault coverage remains the same, because the results of C1 and C2 are checked before the faults disappear. When $R > N$, the probability of detecting a single-bit and single-byte fault is $\frac{N}{R} \times 100\%$ and that of multiple burst faults is $\frac{N}{R} \times 99.99999997\%$. For normal reliability requirement, adding one check round yields a checking ratio of 10/5 = 2.

Security + Reliability (REPO-SR) If the reliability requirement is very high, the designer can check all the rounds. In this case the checking ratio is 10/10 = 1.

3.6 Implementation and Comparison

The implementation results shown in Table 2 are all post place-and-route. We implement fully pipelined and iterative architectures. We use pipelined distributed memories for S-boxes and inverse S-boxes similar to [22]. Hardware redundancy, information redundancy [5,22,43], hybrid redundancy [17], and REPO are compared. The metrics include (1) slice utilization (the number of occupied slices), (2) slice overhead (ratio of number of slices for CED schemes over the number of slices for AES), (3) maximum clock frequency, (4) throughput, and (5) efficiency (ratio of (4) over (1)).

For pipeline architecture, because the load of pipeline stages need to be balanced to maximize performance, REPO-SR is used and throughput overhead is 100%. The hardware overhead of REPO-SR is much lower than that of hardware redundancy for both encryption and decryption. Parity 1 has 16 parity bits for the 128-bit datapath. It expands S-boxes and inverse S-boxes for parity predictions, i.e., two blocks of 256×9 memory cells. Therefore it has more than 100% hardware overhead. Parity 2 has 1 parity bit for the 128-bit datapath. Therefore, it also has very low hardware overhead. Parity 3 has 16 parity bits for the 128-bit datapath. Because it uses a novel parity formation technique, it is able to reduce the hardware overhead. Because the scheme in [43] uses 1-bit signatures for the 128-bit block of data, it has lower hardware overhead and higher efficiency compared to REPO-SR. However, from Table 1, the fault coverage of this scheme is the lowest. [5] and [22] use 16 bits for each 128-bit block, and this leads to much higher fault coverage. REPO-SR has much smaller hardware overhead and higher efficiency than [5], but provides higher fault coverage. Another limitation of [5] and [43] is that they are only applicable to S-box implementation using LUT. On Virtex-5, REPO-SR has higher efficiency than most CEDs except [22]. Although REPO-SR has approximately the same hardware overhead compared to [22], it detects all single-byte faults and lowers the fault miss rate of multiple burst faults by an order of 10^5. The schemes in [17] are only applicable when encryption and decryption are on the same chip. Therefore, if only encryption or decryption is on chip, the hardware overhead of [17] is in the 49.7–108.7% range [22], e.g., 108.7% for AES encryption on Virtex-4 FPGA. If both encryption and decryption are on the same chip, the hardware overhead of [17], which is from the comparator, is very low. For Virtex-5, the overhead of this scheme is -14.8%, because the slice utilization of this scheme is smaller than the total slice

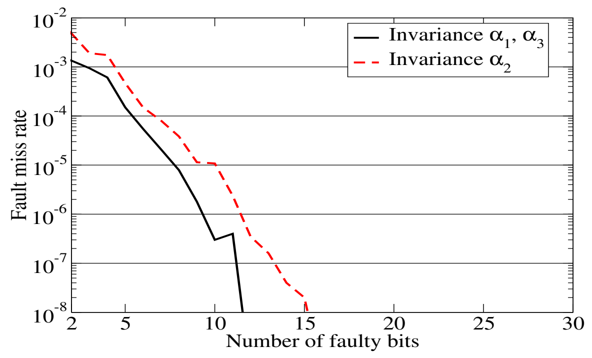


Fig. 7: Fault miss rate for invariance α_1, α_2, and α_3

utilization of encryption and decryption. However, the efficiency of REPO-SR is higher than that of [17]. Most importantly, REPO-SR and all other CEDs do not require both encryption and decryption to be on chip. The RESO technique in [8] is implemented on Virtex-II FPGA. It has 2.7% additional flips flops (2023 against 1969), but 2.4% less slices (1699 against 1740) compared to no CED. Although the hardware overhead is quite small, this technique only focuses on fault detection in S-boxes. For iterative architecture, because round 0 is performed in the same clock cycle as round 1, an extra delay is added in the critical path. The designer can specify four different checking policies, i.e., REPO-S, REPO-SD, REPO-SAR, and REPO-SR. REPO-SR has the least hardware overhead because it does not need extra state registers and decoders. The hardware overhead of REPO as a 15.4-27.3% is slightly higher than that of the pipeline architecture on Virtex-5. If REPO is implemented in ASIC, it needs an extra metal layer for routing, so that α (or α^{-1}) will not overlap with the original datapath. This observation also applies to RESO. Another hybrid redundancy technique in [39] is implemented using 90nm CMOS ASIC library. The compact implementations achieved performances of 2.21 Gbps with 16.1 Kgates. In contrast, the performances without CED are 1.66 Gbps with 12.9 Kgates. The performance increase are due to sub-pipelining. The hardware savings of this technique come from the resource sharing between the encryption and decryption. Therefore, although over 100% hardware overhead is expected for both encryption and decryption unit, the sharing reduce the hardware overhead to only 24.8%.

3.7 Evaluation of Round Level Invariances

There are other invariances that can be used for CED [20]. Most of them restrict the pattern of the inputs and thus are not effective when realistic random inputs are provided. However, there are two other invariances that allow us to perform CED on any inputs:

$$\alpha_2(X) = \begin{bmatrix} x_{0,2} & x_{0,3} & x_{0,0} & x_{0,1} \\ x_{1,2} & x_{1,3} & x_{1,0} & x_{1,1} \\ x_{2,2} & x_{2,3} & x_{2,0} & x_{2,1} \\ x_{3,2} & x_{3,3} & x_{3,0} & x_{3,1} \end{bmatrix} \tag{16}$$

$$\alpha_3 = \alpha_1(\alpha_{pre}(X)) = \alpha_1(\begin{bmatrix} x_{0,0} & x_{0,3} & x_{0,2} & x_{0,1} \\ x_{1,0} & x_{1,3} & x_{1,2} & x_{1,1} \\ x_{2,0} & x_{2,3} & x_{2,2} & x_{2,1} \\ x_{3,0} & x_{3,3} & x_{3,2} & x_{3,1} \end{bmatrix}) \tag{17}$$

α_2 swaps the first and third columns and also the second and fourth columns in the state matrix. α_3 is the same as α_1 except that the initial input X is permuted by α_{pre} before being applied to the input. Fault miss rate for CED using invariances α_1, α_2, and α_3 are compared in Fig. 7. In our experiment, we gradually increase the number of faulty bits from two to 30. We have done 10^4 fault injections for each fixed number of faulty bits. In each fault injection, a fix number of faults is injected, e.g., 11 faults. For each injection, we applied 10^9 test vectors. The fault miss rate dropped sharply to below 10^{-7} after we injected 11 faults using α_1 and α_3, and 15 faults for α_2. After we injected 15 faults, the fault miss rate

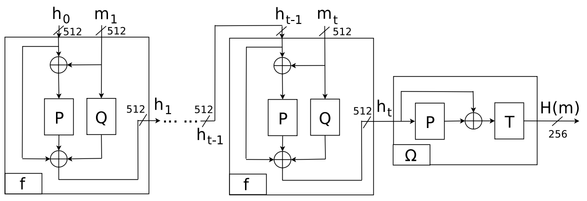


Fig. 8: Grøstl-256 algorithm [12].

was very low and thus not shown in Fig. 7. The fault miss rates of invariance α_1 and α_3 are lower than the fault miss rate of invariance α_2 with any number of faults injected. Compare to α_2, the area and performance overheads of α_1 are the same. For α_3, one first need to apply $\alpha_{pre}(X)$ as input to run C1, and store the result V_{pre}. Then use $\alpha_1(\alpha_{pre}(X))$ as input to run C2, and apply α_1^{-1} to the result $V_{pre}^{'}$ to compare with V_{pre}. Both C1 and C2 are extra overhead for performance. Compare with α_3, C2 is the only performance overhead for α_1. Therefore, α_1 is the most effective invariance.

There are several other invariances that can also be used for CED in AES. However, these invariances are not applicable to all the round operations. Rather, they are applicable to a subset of the round operations. We discuss this in Section 5.1.

4 REPO for AES-style Primitives

AES is the standardized symmetric key algorithm and it is the most widely used symmetric key algorithm. REPO is also applicable to many AES-like structures such as Khazad, Anubis, Whirlpool, Grindahl, Grøstl, LANE, and SHAvite-3. By giving a discussion on AES and Grøstl, it will benefit many other algorithms.

4.1 Grøstl Algorithm

Grøstl hash function is capable of returning digests of any number of bytes from 1 to 64; i.e., from 8 to 512 bits in 8-bit steps. The variant returning n bits is called Grøstl-n. In this paper, we consider Grøstl-256 for brevity, but the same technique can also be used for other digest sizes.

Grøstl-256 takes an arbitrary length message and generates a 256-bit digest as shown in Fig. 8. Let us denote the message to be hashed as M. This message is first padded and then split into 512-bit message blocks. We denote these blocks by m_i ($1 \leq i \leq t$). Grøstl-256 has one or more compression function f and a truncate function Ω. Function f compresses two 512-bit inputs (h_{t-1} and m_t) to obtain one 512-bit output (h_t). One first chooses an initial value for the chaining input h_0. Each intermediate function generates output according to $h_i = f(h_{i-1}, m_i)$. When the last message block is processed, h_t is generated. Then, an output transformation Ω takes the 512-bit output from function f and truncates it into 256 bits to generate the final digest.

4.2 Compression Function

As shown in Fig. 8, f is constructed from two permutations, P, and Q, such that $f(h_{i-1}, m_i) = P(h_{i-1} + m_i) + Q(m_i) + h_{i-1}$, where h_{i-1} and m_i are the chaining input, and message block, respectively. P and Q are both inspired by AES round operations; thus, their structures are similar to AES. P and Q both contain multiple rounds, and 10 rounds are used for Grøstl-256. One permutation round consists of AddRoundConstant, SubBytes, ShiftBytes, and MixBytes, denoted by A_g, B_g, S_g, and M_g, respectively. It is noted that the only two differences between P and Q are the constant values in AddRoundConstant and the position shifts in ShiftBytes. The input of each operation acts on a 512-bit input state where each state element is a byte in $GF(2^8)$. In this paper, each byte is denoted by $s_{r,c}$ ($0 \leq r, c \leq 7$), and indicates that this byte is in row r and column c in state matrix.

$$S = [s_{r,c}]_{r,c=0..7} \tag{18}$$

In AddRoundConstant, the input state is added (modulo-2) to the round-dependent constant matrix $D[j]$. Let X be the input and the resulting output is:

$$Y = A_g(D[j])X = [s_{r,c}]_{r,c=0..7} \oplus [d[j]_{r,c}]_{r,c=0..7} \tag{19}$$

where $D[j]$ is the round constant used in round j. P and Q have different round constant matrices. It is noted that in permutation P, all the bytes of the round constants are zero, except for the first row, which contains byte constants that depend on round j. The elements of the first row are:

$$[d[j]_{1,c}]_{c=0..7} = c0 \oplus j \tag{20}$$

where c is the column number. Similarly, in Q, all the bytes in the round constant matrices are 0xff, except for the last row which has round-dependent constants. The elements of the last row are:

$$[d[j]_{7,c}]_{7,c=0..7} = (f-c)f \oplus j \tag{21}$$

In SubBytes, all the bytes are processed separately by 64 SBs. Each SB is the same as the ones in AES. The resulting output is:

$$Z = B_g(Y) = [y_{r,c}]_{r,c=0..7} \tag{22}$$

In ShiftBytes of P, the rows of the state are shifted cyclically byte-wise using a different offset for each row. The eight rows from row 0 to 7 are cyclically shifted to the left by k bytes, where k is the number of the row. The resulting output is:

$$U = S_g(Z) = [z_{r,(r+c)\ mod\ 8}]_{r,c=0..7} = [u_{r,c}]_{r,c=0..7} \tag{23}$$

In ShiftBytes of Q, the number of bytes shifted are:

$$f(r) = \begin{cases} 2 \times r + 1 & 0 \leq r \leq 3 \\ (2 \times r)\ mod\ 8 & 4 \leq r \leq 7 \end{cases} \tag{24}$$

The resulting output is:

$$U = S_g(Z) = [z_{r,(c+f(r))\ mod\ 8}]_{r,c=0..7} = [u_{r,c}]_{r,c=0..7} \tag{25}$$

In MixBytes, the output state is obtained by multiplying a constant matrix N with the output of ShiftBytes. N is shown below:

$$\begin{bmatrix} 02\ 02\ 03\ 04\ 05\ 03\ 05\ 07 \\ 07\ 02\ 02\ 03\ 04\ 05\ 03\ 05 \\ 05\ 07\ 02\ 02\ 03\ 04\ 05\ 03 \\ 03\ 05\ 07\ 02\ 02\ 03\ 04\ 05 \\ 05\ 03\ 05\ 07\ 02\ 02\ 03\ 05 \\ 04\ 05\ 03\ 05\ 07\ 02\ 02\ 03 \\ 03\ 04\ 05\ 03\ 05\ 07\ 02\ 02 \\ 02\ 03\ 04\ 05\ 03\ 05\ 07\ 02 \end{bmatrix}$$

The resulting output of P is:

$$V = M_g(U) = [v_{r,c}]_{r,c=0..7} \tag{26}$$

The resulting output of Q is:

$$V = M_g(U) = [v_{r,c}]_{r,c=0..7} \tag{27}$$

Finally, the function Ω discards all but the trailing 256 bits of $P(h_t) + h_t$, where h_t is the 512-bit output of the last function f. The details of permutation P have been presented earlier in this subsection.

4.3 Invariance of Grøstl

We discover that a mapping invariance similar to AES also holds true for P and Q in Grøstl. Because the proof for this invariance is similar to the one for AES, we state the theorem without giving a formal proof.

Theorem 3. *A round in permutation P is represented as*

$$M_g(S_g(B_g(A_g(D[j])(X))))$$

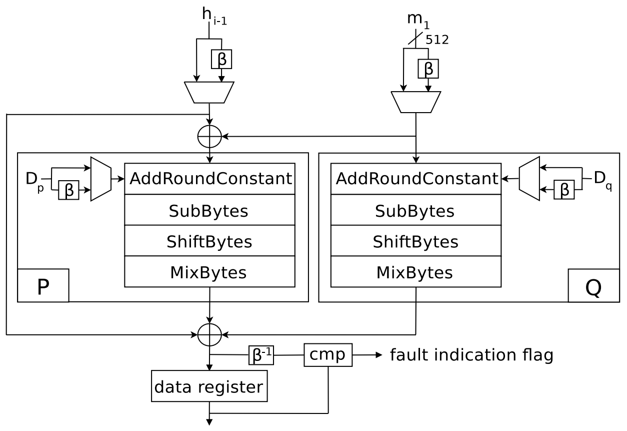


Fig. 9: REPO architecture 1 for Grøstl.

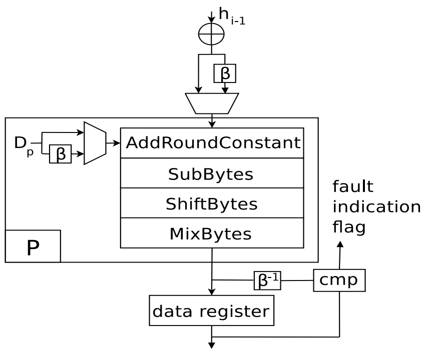


Fig. 10: REPO architecture 2 for Grøstl.

where X is the 512-bit input to the round. Byte permutation β exists such that the following holds true:

$$M_g(S_g(B_g(A_g(D[j])(X)))) = \beta^{-1}(M_g(S_g(B_g(A_g(\beta(D[j]))(\beta(X)))))) \tag{28}$$

where β^{-1} denotes the inverse function of β.

One of the byte permutations is:

$$\begin{aligned}\beta(X) = \beta([x_{r,c}]_{r,c=0..7}) = [x_{r,(c+7)\ mod\ 8}]_{r,c=0..7} \\ \beta^{-1}([x_{r,(c+7)\ mod\ 8}]_{r,c=0..7}) = [x_{r,c}]_{r,c=0..7}\end{aligned} \tag{29}$$

4.4 REPO for Grøstl

We propose two REPO architectures for Grøstl as shown in Figs. 9 and 10.

REPO architecture 1: The permutation is done at the input of the compression function. The output is inverse permuted and compared with the previous output from the compression function. The CED round can either be the first round in the ten round compression function, or it can be an extra ten round compression in the entire hash execution. This REPO architecture uses four muxes and one comparator.

REPO architecture 2: The permutation is done at the input of both P and Q. In this architecture, an arbitrary number of CED rounds can be inserted before or after any of ten rounds in P and Q. Therefore, this architecture offers more flexibility. But it duplicates the xor gates and add comparators to compare with the originals. This REPO architecture uses four muxes, 1024 xor gates, and four comparators.

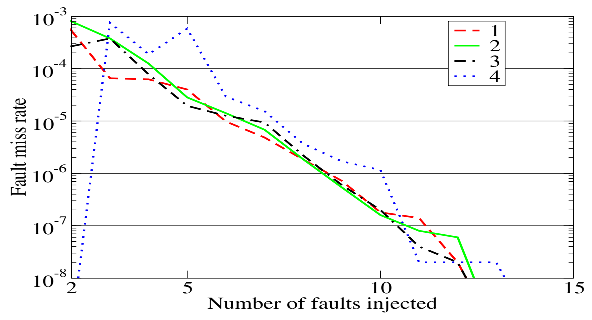


Fig. 11: Fault miss rate for P with different amount of permutation.

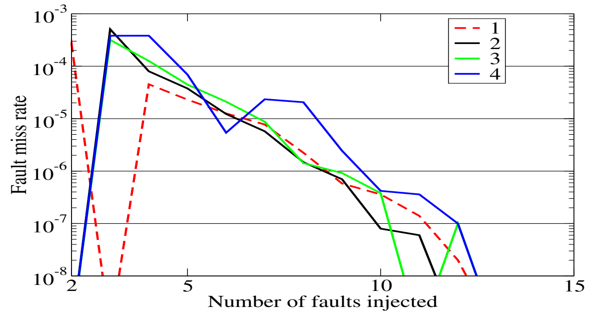


Fig. 12: Fault miss rate for Q with different amount of permutation.

4.5 Fault Analysis

Single-Bit and Single-Byte faults REPO also detects all single-bit and single-byte faults for Grøstl. The proof is similar to that of theorem 2.

Fault Coverage for Multiple Faults We have simulated different numbers of faults in P and Q from 2 to 30. For each number of faults, we apply 5×10^8 test vectors in the simulations. The fault miss rates in the presence of different numbers of faults for P and Q are shown in Figs. 11 and 12, respectively. The number of permuted bytes is from one to four. The simulation results show that in both cases, the fault miss rate decreases to below 10^{-8} after we inject 13 faults. The results also shows the fault coverage with respect to the number of different amount of byte permuted. The fault coverage does not diverge significantly. The variation in both figures is because the random number generator we use is not perfectly random.

4.6 Implementation Results

As shown in Table 3, we have implemented Grøstl and two CED architectures. The Grøstl merges function Ω within function f to achieve compact implementation. Both REPO architectures decrease the frequency of Grøstl because of the extra gates added in the critical paths. The throughputs of REPO decreases because of the extra CED rounds. In REPO architecture 2, when the checking ratio is 10 and 1, the throughput is 2.08Gbps and 1.14Gbps, respectively.

4.7 Grøstl Invariances

Fix points invariance [12]: One chooses m arbitrarily, then we have the following relationship:

$$h = P^{-1}(Q(m)) \oplus m \Rightarrow f(h, m) = h$$

Thus, this invariance requires a certain relation between h and m. one needs to compute the special h at runtime and feed it back with m to the compression function. Because P^{-1} is not provided in the original Grøstl algorithm, this method

Table 3: Comparisons of implementation of REPO for Grøstl on Xilinx Virtex-5 FPGA (xc5vlx110-3). a. checking ratio is 10 b. checking ratio is 1

Scheme	Slice(overhead)	Freq. (MHz)	Thro. (Gbps)	Eff.(Mbps /slice)
Grøstl	3091(-)	191.3	2.45	0.79
REPO 1	3354(8.5%)	175.7	2.23	0.67
REPO 2	3720(20.3%)	178.4	2.08[a]– 1.14[b]	0.56

Table 4: Comparison of time redundancy techniques for AES. B is SubBytes. S is ShiftRows. M is MixColumns. A is AddRoundKey. X means any input pattern. Δ means restricted pattern.

Time	Transient Fault				Permanent Fault				
Redundancy	B	S	M	A	B	S	M	A	DFA
Simple [24](X)	√	√	√	√					
DDR [23](X)	√	√	√	√					
General RESO [8](X)	√	√	√	√	√				
$(B(X))^{277182}$					√				
$M(S(X))^8$						√	√		
$A(M(X))^8$							√	√	
$M(S(B(\Delta)))$					√	√	√		
$(M(S(B(\Delta))))^2$					√	√	√		
$(M(S(B(\Delta))))^4$					√	√	√		
REPO (X)	√	√	√	√	√	√	√	√	√

requires extra hardware. Alternatively, one can precompute many pairs of h and stores them along with their associated m on the chip. This can be use as a built-in-self-test (BIST), however, BIST requires extra memories and other logics.

Preimage invariance [12]: For a given target T, M and X exist such that:

$$T = H + P(H + M) + Q(M) = X + P(X) + M + Q(M)$$

Note that $H = X + M$.

To compute M and X, one needs to use cycle finding algorithm, so it is not applicable at runtime. One can also precompute many pairs of M and X and use BIST technique for fault detection. As mentioned previous, this is costly compared with the invariance-based scheme.

5 Discussion

In this section, we compare REPO with other time redundancy techniques. We also point out its limitations.

5.1 Compared with Time Redundancy for AES

In this section, we analyze the advantage and disadvantage of each of different time redundancy technique in Table 4. At a first glance, REPO looks very similar to simple time redundancy but requires some more hardware. However, simple time redundancy cannot detect permanent faults and long transient faults that last for the normal computation and recomputation. REPO can detect both transient and permanent faults because the hardware computing the same data byte are different in the computation and recomputation. The DDR technique [23] is a form of time redundancy. Attackers have successfully injected long transient faults to break this countermeasure [8]. Moreover, our technique is a complementary technique with the DDR technique. A person who employs our technique can also add DDR to it and vice versa.

In [8], they cyclically shift the input of S-Boxes and restore it after S-boxes. Therefore, the technique can detect permanent faults in S-boxes, but they do not detect permanent faults in ShiftRows, MixColumns, or AddRoundKey. REPO indicates that applying the general RESO to other three round operations can protect the whole round.

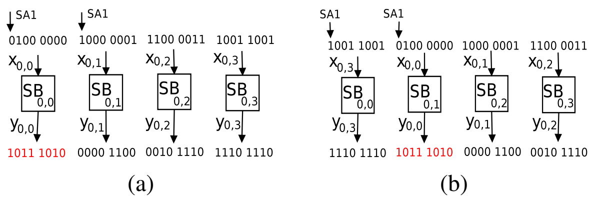


Fig. 13: A case when two stuck-at faults are not detected

Invariance $(B(X))^{277182}$: If you apply an initial input and feed the output back to the input of S-box for 277182 times, the S-box output will be the same as the initial input. If one uses this invariance for CED, the chip needs to stop the normal computation and switch to CED mode. In the CED mode, it will run 277182 cycles to compare the output with the original input, which incurs huge performance overhead. It will not detect transient fault when the chip is not operating in the CED mode. Moreover, this invariance only protects the SubBytes operation.

Invariance $M(S(X))^8$ and $A(M(X))^8$: These invariances has similar drawbacks as the previous invariance. As shown in Table IV, they only protect some round operations and they cannot detect transient faults if AES is not in CED mode.

Invariance $M(S(B(\Delta)))$, $(M(S(B(\Delta))))^2$, and $(M(S(B(\Delta))))^4$: These invariances have restrictions on input patterns. Let $w, x, y, z, s, t, u, v \in GF(2^8)$. $\alpha_4 = (x, x, ..., x)$. From this invariance, we know that if all input bytes are the same, the output bytes are also the same. Because it is unlikely that all the input bytes are the same, to use this invariance, we can also use the same hardware but we extend the first input bytes to the rest of the bytes. In the check round, we select this input. The hardware is the same as invariance α_1. If S-boxes are implemented using memory, there is a significant drawback of this technique. For one test vector, it tests only one memory location. Therefore, the fault coverage for S-box is only around 1/256. There are several other invariances similar to α_4. Interested readers can find more information in [20]. Another problem is that it cannot detect transient fault if the AES is not in test mode.

Because DFA can utilize faults injected in the targeted round and it is not limited to faults injected in a specific round operation, all the round operations need to be protected. Even if all the rounds are protected, it is still not enough to defend against DFA. As demonstrated in [8], the attacker can inject faults that last for both the computation and recomputation to defeat the DDR technique. The general RESO technique detects permanent faults in ShiftRows, but not permanent faults in the other three round operations. REPO detects permanent faults in all four round operations.

5.2 Undetectable Faults

In Fig. 13, let two stuck-at-one faults be injected; one fault is in $SB_{0,0}$, and it affects the left-most bit of the S-box input. The other is in $SB_{0,1}$, and it also affects the left-most bit of the S-box input . The inputs are $x_{0,0}$ = 0100 0000, $x_{0,1}$ = 1000 0001, $x_{0,2}$ = 1100 0011, $x_{0,3}$ = 1001 1001. In C1, the inputs and outputs of the S-boxes are shown in Fig. 13(a). Although there are two faults, only the output of $SB_{0,0}$ is affected, because the faulty bit position of $SB_{0,1}$ has the same value as the input vector. Similarly, in C2, Fig. 13(b) shows that only the output of $SB_{0,1}$ is affected because the faulty bit position of $SB_{0,0}$ has the same value as the input vector. REPO cannot detect the faults in this special case. If two faults are different stuck-at fault types, then REPO will detect them. Not only do faults need to appear in specific positions, they also need to be excited by the input vectors in a specific way to be undetectable. If we have more than two faults, they will only be undetectable if they manifest themselves in a way that is similar to the two fault cases. However, α_2 can detect this kind of faults. One can implement both α_1 and α_2 for REPO. In this case, for each normal round, two check rounds with permutation α_1 and α_2 are followed. The performance overhead will range from 100% to 200%. If we check for five rounds, then we will need five extra rounds for each of the two invariances. So we need a total of 10 extra rounds. Therefore, the performance overhead is 100%. However, if one needs high reliability, one can also add 20 extra rounds. Then the performance overhead is 200%. Although the performance overhead are much higher, this kind of faults is detected. Another case requires the attacker to inject faults that are a multiple of four. As shown in Fig. 14(a), the attacker flips four bits which are the first bits of four bytes in the same row. In C1, all outputs of the four S-boxes are faulty. In C2, if the attacker is able to flip all the first bits in the four bytes in the same row shown in Fig.14(b), the S-boxes will produce the permuted version of the wrong output. After inverse permutation, the two faulty outputs will be the same. With this

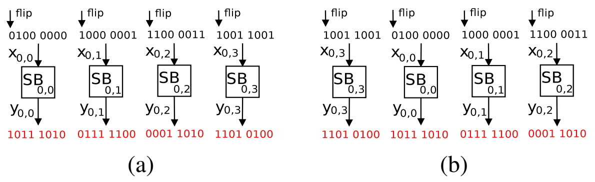


Fig. 14: A case when four bit flip faults are not detected

special case, REPO cannot detect the faults. However, the assumption of this attacker is really strong, it is not clear how the attacker can achieve this by the current fault injection technology.

5.3 Limitations

We would like to point out that REPO does rely on the redundant hardware structure in the circuit. Let's take a look at the S-boxes in AES. To detect faults in S-boxes, the same data bytes need to be computed on different S-boxes in the normal round and the check round.

Therefore, in AES implementations with 32-bit datapath [38] [25] [10], REPO will not be as effective. Because the data will be computed by the same hardware in the normal and the check round, REPO will have the same fault detection capability as time redundancy. Similarly, REPO is the same as time redundancy in AES implementations with 8-bit datapath [14]. In those implementations, we recommend the designers to use full hardware redundancy or hybrid redundancy.

6 Conclusion

As fabrication technologies advance, reliability has become a significant challenge for hardware designers. Moreover, fault attacks show the occurrence of errors must be considered seriously in secure implementations. Error detections schemes can improve the security of cryptographic implementations.

In this work, we propose REPO for AES and AES-style hash function Grøstl. The fault coverage for both AES and Grøstl are 100% for single-bit and single-byte faults, and close to 100% for multiple-bit burst faults and random faults. The hardware overheads is around 12.4-27.3% for AES and 8.5-20.3% for Grøstl. REPO relies on the algorithmic property of AES round invariances, thus, it is applicable to many AES-style cryptographic primitives including stream ciphers and other cryptographic hash functions. Although we implement REPO based on AES-128, it can also be applied to AES with other key lengths such as 192 and 256.

Moreover, AES-style structure is made of very simple and fast operations that can reach very fast hardware implementation. For cryptographic circuits embedded in a larger system as a coprocessor, the cryptographic hardware is likely to work a lower frequency than its maximum capacity. Thus, a large portion of the clock cycle would be wasted. In this case, one can combine REPO with DDR approach to improve performance. Therefore, even when the checking ratio is one, the error detection capabilities incurs a very low cost. The control unit needs to be protected as well. We suggest protecting the control unit with state validation, transition verification, and duplication of selected components.

Acknowledgement

The authors would like to thank the reviewers for their valuable comments. This material is based upon work supported by the NSF CNS program under grant 0831349.

References

1. Boost C++ Libraries. http://www.boost.org/.
2. M. Agarwal, M. Zhang B. C. Paul, and S. Mitra. Circuit Failure Prediction and Its Application to Transistor Aging. In *VTS*, pages 277–286, 2007.

3. Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When Clocks Fail: On Critical Paths and Clock Faults. In *CARDIS*, pages 182–193, 2010.
4. Alessandro Barenghi, Cédric Hocquet, David Bol, François-Xaiver Standaert, Francesco Regazzoni, and Israel Koren. Exploring the Feasibility of Low Cost Fault Injection Attacks on Sub-Threshold Devices through An Example of A 65nm AES Implementation. pages 48–60. in Proc. Workshop RFID Security Privacy, 2011.
5. Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, and Vincenzo Piuri. Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard. *IEEE Trans. Computers*, 52(4):492–505, 2003.
6. S. Borkar. Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation. *MICRO*, 25(6):10–16, 2005.
7. Luca Breveglieri, Israel Koren, and Paolo Maistri. An Operation-Centered Approach to Fault Detection in Symmetric Cryptography Ciphers. *IEEE Trans. Computers*, 56:635–649, May 2007.
8. G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin. Glitch and Laser Fault Attacks onto a Secure AES Implementation on a SRAM-Based FPGA. *Journal of Cryptology*, 24, 2011.
9. Y. Chih-Hsu and W. Bing-Fei. Simple Error Detection Methods for Hardware Implementation of Advanced Encryption Standard. *IEEE Trans. Computers*, 55(6):730–731, 2006.
10. Pawel Chodowiec and Kris Gaj. Very Compact FPGA Implementation of the AES Algorithm. In *CHES*, pages 319–333, 2003.
11. W. Fischer and C.A. Reuter. Differential Fault Analysis on Grøstl. In *FDTC*, pages 44–54, Sept. 2012.
12. P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl-A SHA-3 Candidate. http://www.groestl.info/Groestl.pdf, 2011.
13. Christophe Giraud. DFA on AES. In *AES*, pages 27–41, 2005.
14. Tim Good and Mohammed Benaissa. AES on FPGA from the Fastest to the Smallest. In *CHES*, pages 427–440, 2005.
15. Xiaofei Guo and R. Karri. Invariance-based Concurrent Error Detection for Advanced Encryption Standard. In *DAC*, pages 573–578, Jun 2012.
16. Mark Karpovsky, Konrad J. Kulikowski, and Alexander Taubin. Robust Protection Against Fault-Injection Attacks of Smart Cards Implementing the Advanced Encryption Standard. In *DNS*, pages 93–101, 2004.
17. Ramesh Karri, Kaijie Wu, P. Mishra, and Y. Kim. Concurrent Error Detection Schemes of Fault Based Side-Channel Cryptanalysis of Symmetric Block Ciphers. *IEEE Trans. Computer-Aided Design*, 21(12):1509–1517, Dec 2002.
18. Farouk Khelil, Mohamed Hamdi, Sylvain Guilley, Jean Luc Danger, and Nidhal Selmane. Fault Analysis Attack on an AES FPGA Implementation. In *NTMS*, pages 1–5, 2008.
19. Konrad J. Kulikowski, Mark G. Karpovsky, and Alexander Taubin. Robust codes and robust, fault-tolerant architectures of the advanced encryption standard. *Journal of System Architecture*, 53(2-3):139–149, feb 2007.
20. Tri Van Le, Rudiger Sparr, Ralph Wernsdorf, and Yvo Desmedt. Complementation-Like and Cyclic Properties of AES Round Functions. In *AES*, pages 128–141, 2005.
21. V. Lomne, T. Roche, and A. Thillard. On the Need of Randomness in Fault Attack Countermeasures - Application to AES. In *FDTC*, pages 85–94, Sept. 2012.
22. M. Mozaffari-Kermani and A. Reyhani-Masoleh. Concurrent Structure-Independent Fault Detection Schemes for the Advanced Encryption Standard. *IEEE Trans. Computers*, 59(5):608–622, 2010.
23. P. Maistri and R. Leveugle. Double-Data-Rate Computation as a Countermeasure against Fault Analysis. *IEEE Trans. Computers*, 57(11):1528–1539, Nov 2008.
24. T.G. Malkin, F.-X. Standaert, and Moti Yung. A Comparative Cost/Security Analysis of Fault Attack Countermeasures. In *FDTC*, pages 109–123, Sept 2005.
25. Scott McMillan and Cameron Patterson. JBitsTM Implementations of the Advanced Encryption Standard (Rijndael). In *FPL*, pages 162–171, 2001.
26. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. Handbook of Applied Cryptography. *CRC Press*, 1996.
27. Amir Moradi, Mohammad T. Manzuri Shalmani, and Mahmoud Salmasizadeh. A Generalized Method of Differential Fault Attack against AES Cryptosystem. In *CHES*, pages 91–100, 2006.
28. Mehran Mozaffari-Kermani and Arash Reyhani-Masoleh. A Lightweight High-Performance Fault Detection Scheme for the Advanced Encryption Standard Using Composite Field. *IEEE Trans. VLSI Systems*, 19(1):85–91, 2011.
29. Mehran Mozaffari-Kermani and Arash Reyhani-Masoleh. Reliable Hardware Architectures for the Third-Round SHA-3 Finalist Grøstl Benchmarked on FPGA Platform. In *DFT*, pages 325–331, Oct 2011.
30. S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The Soft Error Problem: An Architectural Perspective. In *HPCA*, pages 243–247, 2005.
31. Debdeep Mukhopadhyay. An Improved Fault Based Attack of the Advanced Encryption Standard. In *AFRICACRYPT*, pages 421–434, 2009.
32. National Institute of Standards and Technology (NIST). SHA-3 First Round Candidates. http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html.

33. National Institute of Stardards and Technology (NIST). Advanced Encryption Standard (AES). http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, Nov 2001.
34. National Institute of Stardards and Technology (NIST) Federal Information Processing Standards (FIPS) publication 140-2. Security requirements for cryptographic modules. http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf, Mar 2001.
35. G. Letourneux P. Dusart and O. Vivolo. Differential Fault Analysis on AES. In *Cryptology ePrint Archive*, 2003.
36. G. Piret and J.J. Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In *CHES*, pages 77–88, Sept 2003.
37. J. Rajendran, H. Borad, S. Mantravadi, and R. Karri. SLICED: Slide-based Concurrent Error Detection Technique for Symmetric Block Cipher. In *HOST*, pages 70–75, Jul 2010.
38. Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A Compact Rijndael Hardware Architecture with S-Box Optimization. In *ASIACRYPT*, pages 239–254, 2001.
39. Akashi Satoh, Takeshi Sugawara, Naofumi Homma, and Takafumi Aoki. High-Performance Concurrent Error Detection Scheme for AES Hardware. In *CHES*, pages 100–112, Aug 2008.
40. Nidhal Selmane, Sylvain Guilley, and Jean-Luc Danger. Practical Setup Time Violation Attacks on AES. pages 91–96. EDCC, 2008.
41. Daniel P. Siewiorek and Robert S. Swarz. Reliable Computer Systems: Design and Evaluation. *A K Peters/CRC Press; 3 edition*, 1998.
42. Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault. In *WISTP*, pages 224–233, 2011.
43. Kaijie Wu, Ramesh Karri, G. Kuznetsov, and M. Goessel. Low Cost Concurrent Error Detection for the Advanced Encryption Standard. In *ITC*, pages 1242–1248, Oct 2004.