Concurrent Error Detection for Reliable SHA-3 Design

Pei Luo*, Cheng Li†, and Yunsi Fei*
* Electrical & Computer Engineering Department, Northeastern University, Boston, MA 02115 USA
† Electrical & Computer Engineering Department, University of Rochester, Rochester, NY 14627 USA
silenceluo@coe.neu.edu, lich.univ@gmail.com, yfei@ece.neu.edu

ABSTRACT

Cryptographic systems are vulnerable to random errors and injected faults. Soft errors can inadvertently happen in critical cryptographic modules and attackers can inject faults into systems to retrieve the embedded secret. Different schemes have been developed to improve the security and reliability of cryptographic systems. As the new SHA-3 standard, Keccak algorithm will be widely used in various cryptographic applications, and its implementation should be protected against random errors and injected faults. In this paper, we devise different parity checking methods to protect the operations of Keccak. Results show that our schemes can be easily implemented and can effectively protect Keccak system against random errors and fault attacks.

CCS Concepts

•**Networks** → **Error detection and error correction;**
•**Hardware** → **Robustness;**

Keywords

SHA-3, Keccak, Random errors, Fault injection, Reliability

1. INTRODUCTION

Keccak is a new algorithm that can be used for hashing, stream encryption, pseudo-random sequence generation, message authentication codes (MAC), etc., and has been recently selected as SHA-3 standard [1]. As SHA-3 will be widely used in cryptographic applications, reliability and security of its implementations will be of vital importance to the SHA-3 based security engine.

Cryptographic systems are sensitive to random errors caused by aging, ambient environment such as temperature and X-ray radiation [9, 19]. For cryptographic systems used for encryption, authentication and integrity checking, etc., the random errors will cause incorrect results and make the cryptographic systems unreliable. Attackers can also inject faults temporarily to the system to retrieve the secret key or state. Analyzing the correct output and faulty output is called differential fault analysis (DFA), which has been shown to be very effective in cracking block ciphers including Data Encryption Standard (DES) and Advanced Encryption Standard (AES) [6, 17]. A recent work [3] shows that DFA can also be used to attack SHA-3 implementations to recover the internal states.

To protect cryptographic systems against random errors and injected faults, different error detection methods have been adopted in cryptographic systems [8, 9, 10, 11, 13, 14, 19]. Many schemes are based on redundancy - with another copy of the original implementation for outputs comparison so as to detect error occurring in either copy. To further improve the reliability, the second copy can have different implementation from the first copy [8]. To reduce the resource overhead, error detection coding is also widely used. Parity checking code has been used in the protection of AES [19] because of its efficient design and high error coverage. Besides parity checking, non-linear codes are also introduced to further improve system reliability with higher fault coverage [10].

Previous works on Keccak mainly focus on side-channel analysis and collisions attacks on Keccak [7, 12, 16, 18], while leaving its reliable design against injected faults and random errors almost blank. To the best of our knowledge, only one work about reliable Keccak design has been published [5]. In [5], the authors find a property of Keccak algorithm: each lane of the Keccak state can be rotated by a random number before each round operation, and then shifted back after Keccak operations without changing the results. Based on this property, they implement another copy of Keccak with this kind of rotation for comparison. Any error can cause output mismatch of the two copies and thus can be detected. With different implementations in the two copies, it is unlikely that same errors will appear in the two copies. However, this method has high resource overhead due to the extra copy.

In this paper, we propose a simple and efficient parity checking based error detection method for Keccak. Optimized designs are also proposed to further improve the efficiency of the proposed scheme. We implement the proposed scheme in VHDL and simulate fault injection at gate level to get the fault coverage of the proposed scheme. Results show that our scheme has a high fault coverage for hardware implementations with very small resource overhead.

The rest of this paper is organized as follows. In Section 2, some basic knowledge of Keccak and preliminaries of error detection are introduced. In Section 3, the mathemat-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GLSVLSI '16, May 18-20, 2016, Boston, MA, USA
© 2016 ACM. ISBN 978-1-4503-4274-2/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2902961.2902985

ical property of Keccak is analyzed and our parity checking scheme for Keccak operations is proposed. In Section 4, implementation and fault injection simulation results of the proposed schemes are given. Finally, the paper is concluded in Section 5.

2. PRELIMINARIES

2.1 Preliminaries of Keccak Hash Function

Keccak can work in different modes and have variable length [1]. In this paper, we use Keccak-1600 as an example to demonstrate our scheme. All of the 1600-bit states are organized in a 3-D array, as shown in Figure 1. Each bit is addressed by three coordinates, denoted as $S(x, y, z)$, $x, y \in \{0, 1, ..., 4\}$, $z \in \{0, 1, ..., 63\}$. 2-D entities, plane, sheet and slice, and 1-D entities, lane, column and row, are also defined in Keccak and shown in Figure 1.

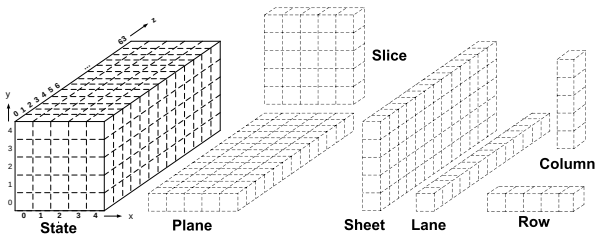

Figure 1: Terminology used in Keccak

Notations: We use 3-D coordinates x, y and z to locate each bit. We also define vectors $X = [0 : 4]$, $Y = [0 : 4]$ and $Z = [0 : 63]$ to stand for multiple bits in one row, column and lane, respectively. Note that coordinates x and y are modular 5 while z is modular 64.

Keccak relies on a Sponge architecture to iteratively absorb message inputs and squeeze out outputs by a f permutation function. The f function consists of 24 rounds, where each round has five sequential steps:

$$S_{i+1} = \iota \circ \chi \circ \pi \circ \rho \circ \theta(S_i),\ i \in \{0, 1, \cdots, 23\} \quad (1)$$

in which S_0 is the initial input. Details of each step are described below:

$-\ \theta$ is a linear operation which involves 11 input bits and outputs a single bit. Each output state bit is the XOR of the input state bit and two intermediate bits produced by its two neighbor columns. We denote the input to θ operation as θ_i while the output as θ_o, and the operation is given as follows:

$$\theta_o(x, y, z) = \theta_i(x, y, z) \oplus (\oplus_{y=0}^{4} \theta_i(x-1, y, z)) \oplus (\oplus_{y=0}^{4} \theta_i(x+1, y, z-1)). \quad (2)$$

$-\ \rho$ is a permutation over the bits of the state along z-axis (in lanes), and the amount of shifted bits depend on the (x, y) coordinates.

$-\ \pi$ is a permutation over the bits of the state within slices. Only the center bit ($x = 0,\ y = 0$) of the slice does not move. All other bits are permuted to other positions depending on their original coordinates.

$-\ \chi$ is a non-linear step that contains mixed binary operations. Every bit of the output state is the result of an XOR between the corresponding input state bit and its two neighboring bits along the x-axis (in a row):

$$\chi_o(x, y, z) = \chi_i(x, y, z) \oplus (\overline{\chi_i(x+1, y, z)} \cdot \chi_i(x+2, y, z)).$$

$-\ \iota$ is a binary XOR with a round constant which is publicly known.

Further details of Keccak can be found in [1].

2.2 Reliable Design with Error Detection

Reliable cryptographic modules rely on error detection to detect random errors and injected faults in systems. The basic structure of error detection is shown in Figure 2, in which the cryptographic module is protected by introducing another module, Protector. Protector is composed of three modules, which are Predictor, Compressor and Comparator.

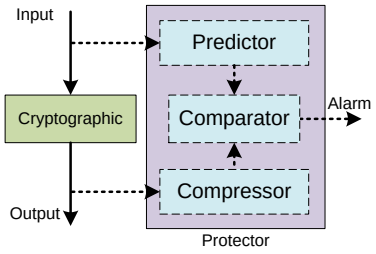

Figure 2: Conception of error detection

Predictor and Compressor read the input and output of cryptographic module respectively, and generate some intermediate computation results. Comparator compares the results of Predictor and Compressor for error detection. If errors happen in cryptographic module, results of Predictor and Compressor may not match, and errors will be detected. Design of predictor and compressor will significantly determine the resource overhead and fault coverage. For example, in redundancy-based error detection, predictor is another copy of the original implementation and there is no compressor.

3. PARITY CHECKING OF KECCAK

In this section, we will propose our parity checking based concurrent error detection scheme for Keccak. We take each operation of Keccak as a cryptographic module shown in Figure 2, and the goal is to design efficient and effective predictor and compressor suitable for each operation.

3.1 Analysis of the Protection of Keccak

Sponge function involves a lot of simple bitwise operations such as XOR, AND, NOT etc., rather than complex nonlinear operations such as S-box in block ciphers. Such bit-wise operations can be combined according to Boolean algebra for efficient predictor and compressor design. For each 3-D input and output state of a cryptographic operation, parity checking can be implemented at different granularities. For example, it can be implemented in row, column, or lane, or in 2-D entities, in unit of slice, sheet, or plane. This will result in different compression ratio, yielding different error coverage and resource overhead.

For concurrent error detection, usually some steps can be combined to achieve higher efficiency. For example, the error detection of ShiftRows, MixColumns and AddRoundKey of AES can be combined together to achieve lower overhead

and higher efficiency [10]. Instead of protecting each operation of Keccak separately, we propose to combine the protections of some operations to improve the efficiency and save resource. For example, ι is a binary XOR operation with a constant number, and thus the protection of ι can be efficiently combined with the protection of χ. In this section, we show how to make use of the property of Keccak and combine parity checking of some steps to achieve higher efficiency.

The overall protection scheme is shown in Figure 3. The protections of χ and ι, ρ and π, are combined respectively for higher efficiency, with the combined operations denoted as χ' and ρ'.

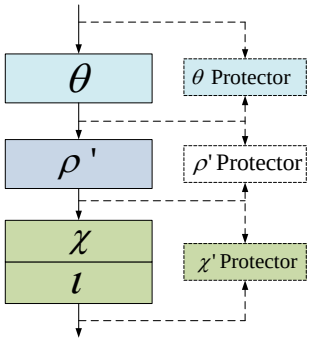

Figure 3: Error detection structure of Keccak

Note that depending on the implementation of Keccak, the ρ' protector may be optional. This is because both ρ and π are permutation operations and only change the position of bits without changing the values. If no storage (state register) is used in these operations, they will be synthesized as wires in FPGA or ASIC. Any errors (stuck-at-zero or stuck-at-one) will be detected by the previous θ protector or the following χ' protector. Nevertheless, we present protection on ρ and π in this section at the algorithm level, and leave the implementation to the next section.

3.2 Parity Checking of θ

3.2.1 Parity Checking of θ in 1-D Entities

As shown in formula (2), each θ operation works on one input bit θ_i and two nearby columns (each five bits). Operations on five bits in the same column of θ_i involve the same two nearby columns. Thus we propose to implement parity checking of θ along the y axis (in column). For each column of θ_o, the parity checking is as follows:

$$\bigoplus_{y=0}^{4}\theta_o(x,y,z)=\bigoplus_{y=0}^{4}\theta_i(x,y,z)\oplus(\bigoplus_{y=0}^{4}\bigoplus_{y=0}^{4}\theta_i(x+1,y,z-1))\oplus(\bigoplus_{y=0}^{4}\bigoplus_{y=0}^{4}\theta_i(x-1,y,z)).$$

Due to the property of XOR operation ($a\oplus 0=a$, $a\oplus a\oplus a=a$), we can simplify the above equation as:

$$\bigoplus_{y=0}^{4}\theta_o(x,y,z)=\bigoplus_{y=0}^{4}\theta_i(x,y,z)\oplus(\bigoplus_{y=0}^{4}\theta_i(x+1,y,z-1))\oplus(\bigoplus_{y=0}^{4}\theta_i(x-1,y,z)). \quad (3)$$

Where the parity checking of each column of θ_o ($\theta_o(x,Y,z)$ on the left hand) is the parity checking of three θ_i columns ($\theta_i(x,Y,z)$, $\theta_i(x-1,Y,z)$ and $\theta_i(x+1,Y,z-1)$).

We denote the parity checking result of θ_i in each column as $P[\theta_i](x,z)$, which means to compress θ_i along the y direction. The parity checking of the input state is a plane ($X=[0:4]$, $Z=[0:63]$). Similarly, we denote the parity checking result of θ_o in each column as $P[\theta_o](x,z)$. We can represent (3) as follows:

$$P[\theta_o](x,z)=P[\theta_i](x,z)\oplus P[\theta_i](x+1,z-1)\oplus P[\theta_i](x-1,z). \quad (4)$$

The parity checking can also be done along the row. Similarly, we have:

$$P[\theta_o](y,z)=P[\theta_i](y,z)\oplus P[\theta_i](z)\oplus P[\theta_i](z-1), \quad (5)$$

in which $P[\theta_i](z)$ stands for the compression of each slice of θ_i and therefore the input state is compressed to a parity lane, and $P[\theta_i](y,z)$ is the parity of a row. Compared to (4), it involves multiple parity generations (both in row and in slice).

3.2.2 Parity Checking of θ in Slice

Equation (4) shows that parity checking in each column involves nearby columns (with one on the same slice). Equation (5) shows that parity checking in each row already involves two slices. We examine parity checking in 2-D entities too. Parity of each slice is denoted as:

$$\bigoplus_{x=0}^{4}\bigoplus_{y=0}^{4}\theta_o(x,y,z)=(\bigoplus_{x=0}^{4}\bigoplus_{y=0}^{4}\theta_i(x,y,z))\oplus(\bigoplus_{x=0}^{4}\bigoplus_{y=0}^{4}\bigoplus_{y=0}^{4}\theta_i(x-1,y,z))\oplus(\bigoplus_{x=0}^{4}\bigoplus_{y=0}^{4}\bigoplus_{y=0}^{4}\theta_i(x+1,y,z-1)). \quad (6)$$

Due to the property of XOR operation, we find a unique property for slice based parity checking of θ operation:

$$\bigoplus_{x=0}^{4}\bigoplus_{y=0}^{4}\theta_o(x,y,z)=\bigoplus_{x=0}^{4}\bigoplus_{y=0}^{4}\theta_i(x+1,y,z-1). \quad (7)$$

It means that the parity of each slice of θ_o is the parity of a nearby slice of θ_i. For the $1,600$-bit state of Keccak, there are 64 slice-based parity checking bits for θ_i and θ_o respectively. Define $P[\theta_o](Z)$ and $P[\theta_i](Z)$ as following:

$$\begin{cases} P[\theta_i](z)=\oplus_{x=0}^{4}\oplus_{y=0}^{4}\theta_i(x,y,z) \\ P[\theta_o](z)=\oplus_{x=0}^{4}\oplus_{y=0}^{4}\theta_o(x,y,z) \end{cases},$$

then (7) can be represented as:

$$P[\theta_i](z)=P[\theta_o](z+1),\ Z=[0:63], \quad (8)$$

which means that the slice-based parity lane of θ_o is a round shift of the parity lane of θ_i. This makes the slice-based parity checking for θ very efficient.

For the three parity generation and checking schemes introduced in Section 3.2.1 and Section 3.2.2, we compare their resource overhead in terms of XOR2 gates. Take parity checking in slice as an example, the predictor compresses every slice into one bit, requiring 24 XOR gates, and the predictor needs 1536 (24 * 64) XOR gates in total. The XOR gates consumption of different protection schemes of θ are listed in Table 1.

Table 1: XOR gates overhead of θ protection

	Predictor	Compressor	Comparator	Total
Column	1920	1280	639	3839
Row	2176	1280	639	4095
Slice	1536	1536	127	3199
Duplicate	3520	0	3199	6719

For schemes with parity generation in row and column, they protect θ for every five bits, while the slice based scheme protect θ operation for every 25 bits. Thus the row and column based schemes have higher error coverage than the slice based scheme. Table 1 shows that the slice based parity checking scheme has lower area overhead than the other two schemes. A balance should be made between error coverage and resource overhead during design. Meanwhile, comparing with the proposed parity checking schemes, duplication based error detection requires much higher resource overhead. In this paper, we use slice-based parity generation to implement the parity checking of θ operation.

3.3 Parity Checking of ρ and π

As discussed in Section 3.1, ρ and π can be left unprotected if they are implemented using wires in circuit. For implementations in which ρ and π use registers (pipelined design, for example), the protections of ρ and π can be implemented either separately or combined together.

For ρ, the permutation is along z-axis, thus we propose to compress ρ operation along each lane, and the protection function is as following:

$$P[\rho_i](x,y) = P[\rho_o](x,y). \tag{9}$$

In (9), the left side is the predictor while the right side is the compressor design. Similarly, while π permutes the state bits inside each slice, we propose to compress the bits inside each slice:

$$P[\pi_i](z) = P[\pi_o](z). \tag{10}$$

The protection of ρ and π can both be implemented efficiently because of their simple operations. To further improve the implementation efficiency, we propose to combine ρ and π as a new operation ρ' and protect it instead. For the protection of ρ', parity checking can be efficiently implemented in z-axis. The protector of ρ' can be designed as following:

$$\pi(P[\rho'_i](x,y)) = P[\rho'_o](x,y), \tag{11}$$

in which the left-hand side is predictor and the right-hand side is for compressor design. For predictor design, the 1,600-bit state is firstly compressed to a slice $P[\rho'_i](x,y)$, and this slice is then permuted according to π operation. For this protection scheme, the predictor and compressor both require $1,575$ XOR gates, and the comparator needs 49 XOR gates. Although θ can be also combined with ρ', the efficiency is not improved significantly, thus we protect them separately in this paper.

3.4 Parity Checking of χ and ι

3.4.1 What Should We Avoid in Protection of χ?

χ is the only non-linear step in Keccak and it involves both NOT and AND operations. This makes the parity checking of χ step different from previous operations. In this section, we first show a pitfall for the protection of χ which should be avoided in practical design.

We take one row of χ_i as an example here, we denote the five bits of this row as $\{a,b,c,d,e\}$. Then five bits of corresponding χ_o output row can be denoted as $a \oplus (\bar{b} \cdot c)$, $b \oplus (\bar{c} \cdot d)$, $c \oplus (\bar{d} \cdot e)$, $d \oplus (\bar{e} \cdot a)$ and $e \oplus (\bar{a} \cdot b)$. The parity bit of this single row is:

$$\begin{aligned} &a \oplus b \oplus c \oplus d \oplus e \oplus (\bar{b} \cdot c) \oplus (\bar{c} \cdot d) \oplus (\bar{d} \cdot e) \oplus (\bar{e} \cdot a) \oplus (\bar{a} \cdot b) \\ &= (a \cdot b) \oplus (b \cdot c) \oplus (c \cdot d) \oplus (d \cdot e) \oplus (e \cdot a). \end{aligned} \tag{12}$$

For equation (12), if one bit out of five bits in this row is flipped, the final result of equation (12) may not change. Take bit a as an example, the change of a will affect both $(e \cdot a)$ and $(a \cdot b)$. According to De Morgan's laws:

$$(e \cdot a) \oplus (a \cdot b) = a \cdot (b \oplus e). \tag{13}$$

When a flips, if $b \oplus e$ is already 0, the above result does not change, then the result of (12) will not change either. Assume all bits are independent, the probability of $b \oplus e = 0$ is 50% and therefore the fault coverage of this scheme is 50% for single bit errors. So parity checking in each row of χ should be avoided because of the non-linearity of χ operation, and thus parity checking in each slice will not be applicable for χ operation either.

3.4.2 Parity Checking of χ in Each Lane

In this section, we show how to build practical error detection module for χ operation. As x-axis compression of χ is not a good choice, we considering compressing χ results along either z-axis or y-axis. For parity generation along z-axis, 64 bits in each lane are compressed to one bit and thus the $1,600$ bits are compressed to one slice. The parity generation can be denoted as following:

$$P[\chi_o](x,y) = \bigoplus_{z=0}^{63} \chi_o(x,y,z). \tag{14}$$

According to the definition of χ operation, we have:

$$\begin{aligned} P[\chi_o](x,y) = &\bigoplus_{z=0}^{63} (\chi_i(x,y,z) \oplus \chi_i(x+2,y,z) \\ &\oplus \chi_i(x+1,y,z) \cdot \chi_i(x+2,y,z)) \\ =&P[\chi_i](x,y) \oplus P[\chi_i](x+2,y) \\ &\oplus P[\chi_{and}](x,y), \end{aligned}$$

in which $\chi_{and}(x,y,z) = \chi_i(x+1,y,z) \cdot \chi_i(x+2,y,z)$. Thus the predictor design of parity checking for χ at z-axis is also easy to implement. It involves AND operations first, then it compresses the data in z-axis to generate the parity for checking.

Meanwhile, the parity of χ can also be generated in each column, along y-axis direction. For this scheme, five bits in each column are compressed to one bit, and the $1,600$-bits state is compressed to one plane. The compressor works as follows:

$$P[\chi_o](x,z) = P[\chi_i](x,z) \oplus P[\chi_i](x,z) \oplus P[\chi_{and}](x,z).$$

In this scheme, the overhead is higher than the z-axis compression scheme due to its lower compression ratio, while it has higher fault coverage. Thus designers can choose the best scheme according to the system requirement. In this paper, we implement the protection of χ operation along

Table 2: Resource overhead and error coverage results

	Area		Timing		Power		Error
	(um^2)	Normalized	(ns)	Normalized	(mW)	Normalized	coverage
Original	41611.7	100.00%	3.892	100.00%	17.95	100.00%	0%
Proposed	52867.2	127.05%	4.500	115.62%	26.69	148.69%	83.60%
Design 2	62429.4	150.01%	5.476	140.70%	41.78	232.76%	83.34%
Design 3	66621.0	160.10%	4.381	112.56%	44.07	245.52%	89.89%

the z-axis direction (lane) for fault injection simulation and resource overhead evaluations.

3.4.3 Combination of χ and ι

While ι only adds a constant number to χ result, it can be easily combined with χ as discussed in previous section. If χ is checked at z-axis, the combined parity checking is

$$P[\iota_o](x,y) = P[\chi_i](x,y) \oplus P[\chi_i](x+2,y) \oplus P[\chi_{and}](x,y) \oplus P[\iota_c](x,y), \quad (15)$$

and parity checking of χ combining with ι at other directions are similar. Note here that $P[\iota_c](x,y)$ is computed at design stage to avoid computations for each run. This protection scheme requires $3,200$ XOR gates and $1,600$ AND gates for the predictor, 1,575 XOR gates for the compressor, and 49 XOR gates for the comparator.

4. IMPLEMENTATION AND FAULT INJECTION RESULTS

4.1 Implementation Results

To evaluate the proposed scheme, we implement the unprotected Keccak implementation (referring to the official implementation provided online [2]) and the proposed scheme in Figure 3 . We implement three variants of the proposed scheme:

- Proposed combines the protection of χ and ι as described in Section 3.4.3, and leaves ρ and π unprotected;
- Design 2 protects χ and ι separately, and leaves ρ and π unprotected;
- Design 3 combines the protection of χ and ι as described in Section 3.4.3, and protects ρ and π together referring to (11).

For the above three designs, we implement the protection of θ using the scheme in Section 3.2.2, which is to implement parity checking in each slice. Each protector is composed of three parts, as described in Figure 2. All the designs have five steps in each round within one clock cycle using combinational circuits. For the proposed schemes, we implement the original circuit and the protection circuits in parallel and they work simultaneously.

For integrated circuit resource evaluation, the implementations (with and without protections) are modeled in VHDL and synthesized in Cadence Encounter RTL Compiler with a 45nm Opencell library (NanGate FreePDK45 v1_3_v2009_07). The designs were placed and routed using Cadence Encounter. The power and area overhead of the protection schemes were estimated using Concurrent Current Source (CCS) model under typical operation conditions assuming a supply voltage of 1.1V and a temperature of 25 Celsius degree. The results including area, timing delay and power consumption are shown in Table 2.

Table 2 shows that Proposed has much higher performance while lower area and power consumption than Design 2 and Design 3.

Comparing with original implementation, Proposed has about 27.05% area resource overhead. Meanwhile, our proposed scheme maintains high performance because it has the protection modules works concurrently with Keccak module. What's more, the proposed scheme combines ρ and π, χ and ι together respectively, thus it has small timing delay overhead.

Comparing with Proposed, Design 2 does not optimize the protections of χ and ι and protect these two modules separately. It shows that Design 2 has much higher area resource overhead than the proposed design because of this separate protection design. Meanwhile, it has larger timing dealy and power consumption than the proposed design as well. Design 3 protects ρ and π together as described in Section 3.4.3, results show that it has much higher area resource overhead than the proposed scheme, as well as larger timing delay and power consumption. Thus the proposed optimization methods can help to save resources in the protection of Keccak.

4.2 Fault Coverage Analysis

Theoretically, for parity checking schemes, if odd number bits are flipped, the errors will be detected with 100% probability; while if even number bits are flipped, the errors will be always undetected. For real hardware systems, it will be very difficult for the attackers to precisely control the numbers and positions of faulty gates in the circuit [4]. What's more, the errors in the circuit will randomly propagate and cause different numbers of faulty bits in the output [15]. Thus, fault injection simulation results at gate level are required for error coverage evaluation.

In this paper, we randomly inject one to ten stuck-at-0 and stuck-at-1 faults into Keccak circuit for error coverage simulation. To get the fault coverage result, we give random plaintext input for one round of Keccak, then randomly inject one to ten stuck-at-0 or stuck-at-1 faults into the system. We check the results of the protected implementation and the alarm signals to see if we miss any errors. For each design, we run about 10^8 fault injection trials and the error coverage results are shown in Table 2.

Table 2 shows that the proposed scheme has error coverage about 83.60%. Meanwhile, results show that separate protections of χ and ι in Design 2 will not increase the error coverage. This is because ι module is very small and only occupies very small ratio of gates in the design. The probability of errors happening in ι is very small, and the error coverage of Design 2 is almost the same as the proposed design.

For Design 3, it has a large part of gates used for the protection of ρ and π, and errors happen in ρ' module with high probability. In such case, we can assume that a large part of errors are injected into the protection module of ρ' and this part of errors will be detected with high probability. Results show that the error coverage will increase to 89.89% for Design 3. Thus for pipelined designs which use registers to store the results of ρ and π, the protection method proposed in Section 3.3 should be implemented for higher error coverage.

In conclusion, according to the synthesis results in Section 4.1 and fault injection simulation results in Section 4.2, the proposed scheme has a small resource overhead and high performance, and it can detect the injected faults with a high probability. Thus our proposed scheme strikes a good balance between resource overhead and error coverage, and can be efficiently implemented for the protection of Keccak implementations.

5. CONCLUSION

In this paper, we look into the parity checking of Keccak to protect it against random errors and injected faults. We make use of the mathematical properties of Keccak to implement parity checking based error detection. We find that combining protections of some steps reduces the area overhead, timing delay, and power consumption significantly without sacrificing the error coverage. Results show that our scheme has small resource overhead, while the timing delay and power consumption are also very small. Under multiple bit random errors model, fault injection simulation results show that our method can detect 83.60% injected faults. The future work will be more efficient protection methods of Keccak against random errors and injected faults, and protections of Keccak against other kinds of attacks.

Acknowledgment: The authors would like to thank Dr. Rex Xiaofei Guo at Intel for his help. This work was supported in part by National Science Foundation under grants SaTC-1314655 and MRI-1337854.

6. REFERENCES

[1] DRAFT FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2014. Federal Information Processing Standards Publication.

[2] *Keccak Hardware implementation in VHDL Version 3.1*, 2015 (accessed July 14, 2015).

[3] N. Bagheri, N. Ghaedi, and S. Sanadhya. Differential fault analysis of SHA-3. In *Progress in Cryptology – INDOCRYPT 2015*, volume 9462, pages 253–269. 2015.

[4] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, Nov 2012.

[5] S. Bayat-sarmadi, M. Mozaffari-Kermani, and A. Reyhani-Masoleh. Efficient and concurrent reliable realization of the secure cryptographic SHA-3 algorithm. *IEEE Trans. on Computer-Aided Design of Integrated Circuits & Systems*, 33(7):1105–1109, July 2014.

[6] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology – CRYPTO '97*, volume 1294, pages 513–525. 1997.

[7] I. Dinur, O. Dunkelman, and A. Shamir. Collision attacks on up to 5 rounds of SHA-3 using generalized internal differentials. In *Fast Software Encryption*, volume 8424, pages 219–240. 2014.

[8] X. Guo and R. Karri. Invariance-based concurrent error detection for Advanced Encryption Standard. In *Proc. IEEE Design Automation Conf.*, pages 573–578, 2012.

[9] X. Guo and R. Karri. Recomputing with permuted operands: A concurrent error detection approach. *IEEE Trans. on Computer-Aided Design of Integrated Circuits & Systems*, 32(10):1595–1608, Oct 2013.

[10] M. Karpovsky, K. Kulikowski, and A. Taubin. Differential fault analysis attack resistant architectures for the Advanced Encryption Standard. In *Smart Card Research and Advanced Applications VI*, volume 153, pages 177–192. 2004.

[11] R. Karri, K. Wu, P. Mishra, and Y. Kim. Concurrent error detection of fault-based side-channel cryptanalysis of 128-bit symmetric block ciphers. In *Proc. IEEE Design Automation Conf.*, pages 579–584, 2001.

[12] P. Luo, Y. Fei, X. Fang, A. Ding, D. Kaeli, and M. Leeser. Side-channel analysis of MAC-Keccak hardware implementations. In *Hardware & Architectural Support for Security & Privacy*, 2015.

[13] P. Luo, Y. Fei, L. Zhang, and A. Ding. Side-channel power analysis of different protection schemes against fault attacks on AES. In *Int. Conf. ReConFigurable Computing & FPGAs (ReConFig)*, pages 1–6, Dec 2014.

[14] P. Maistri and R. Leveugle. Double-data-rate computation as a countermeasure against fault analysis. *IEEE Trans. on Computers*, 57(11):1528–1539, Nov 2008.

[15] M. Mozaffari-Kermani and A. Reyhani-Masoleh. A lightweight high-performance fault detection scheme for the Advanced Encryption Standard using composite fields. *IEEE Trans. on Very Large Scale Integration Systems*, 19(1):85–91, Jan 2011.

[16] M. Naya-Plasencia, A. RÃűck, and W. Meier. Practical analysis of reduced-round keccak. In *Progress in Cryptology - INDOCRYPT 2011*, volume 7107, pages 236–254. 2011.

[17] G. Piret and J.-J. Quisquater. A differential fault attack technique against SPN structures, with application to the AES and Khazad. In *Cryptographic Hardware and Embedded Systems*, volume 2779, pages 77–88. 2003.

[18] M. Taha and P. Schaumont. Side-channel analysis of MAC-Keccak. In *IEEE Int. Symp. on Hardware-Oriented Security & Trust*, pages 125–130, June 2013.

[19] C.-H. Yen and B.-F. Wu. Simple error detection methods for hardware implementation of Advanced Encryption Standard. *IEEE Trans. on Computer*, 55(6):720–731, June 2006.