Reliable and Lightweight PUF-based Key Generation using Various Index Voting Architecture

Jeong-Hyeon Kim[1], Ho-Jun Jo[1], Kyung-Kuk Jo[2], Sung-Hee Cho[1], Jae-Yong Chung[3], Joon-Sung Yang[4]
[1]*Department of Semiconductor and Display Engineering*, *Sungkyunkwan University*, Suwon, Korea, hyeon0521@skku.edu
[2]*Department of Electrical and Computer Engineering*, *Sungkyunkwan University*, Suwon, Korea
[3]*Department of Electronics Engineering*, *Incheon National University*, Incheon, Korea
[4]*Department of Systems Semiconductor Engineering*, *Yonsei University*, Seoul, Korea, js.yang@yonsei.ac.kr

***Abstract*—Physical Unclonable Functions (PUFs) can be utilized for secret key generation in security applications. Since the inherent randomness of PUF can degrade its reliability, most of the existing PUF architectures have designed post-processing logic to enhance the reliability such as an error correction function for guaranteeing reliability. However, the structures incur high cost in terms of implementation area and power consumption. This paper introduces a Various Index Voting Architecture (VIVA) that can enhance the reliability with a low overhead compared to the conventional schemes. The proposed architecture is based on an index-based scheme with simple computation logic units and iterative operations to generate multiple indices for the accuracy of key generation. Our evaluation results show that the proposed architecture reduces the hardware implementation overhead by 2 to more than 5 times, without losing a key generation failure probability compared to conventional approaches.**

***Index Terms*—Physically Unclonable Function (PUF), Secret Key, Helper Data, Fuzzy Extractor (FE), Index-Based Syndrome (IBS) Coding, FPGA, ASIC**

I. Introduction

Physically Unclonable Function (PUF) is a promising cryptographic technology for Integrated Circuits (ICs). One of the most important attributes of the PUF is to use hardware-specific values generated by process variations for cryptographic algorithms. It makes PUF generate keys for various cryptographic primitives without secure storage and complex algorithms [1]. The distinct and inherent characteristics appear in a circuit output (response) to an input (challenge), which indicates the internal disorder leading to randomness or unpredictability of the device [2]. In other words, each hardware chip has unique characteristics even if it is made with the same design and technology process. Based on this property, the new security primitive does not need to store unique key information in a secure area. In addition, they consume less power and area as compared to the traditional approaches because PUFs do not need to perform complex cryptographic operations. For example, a Static Random Access Memory (SRAM) PUF generates responses by utilizing the initial state of SRAM cells during power-on [2] and a delay based on Arbiter-PUF exploits delay differences of two identical paths for response generation [1]. As in the example, the unique and unexpected values are directly generated via simple operation which could possibly remove the need for a secure storage requirement and complex cryptographic operations.

However, since PUF takes several advantages through the hardware characteristics, there would be a reliability risk of response change due to the uncertainty derived from manufacturing variations and environmental noise. Because it is impossible to control the process variations in silicon manufacturing, it is tremendously hard to manufacture PUFs which produce completely reliable and uniform responses. This is why an error correction scheme comes into play to enhance the PUF reliability. The Fuzzy Extractor (FE) is often used to derive a reliable key for a PUF-based key generation scheme when the PUF responses may be unreliable [3]. Conventional FEs have been introduced to enhance the reliability which use the error correction functions, such as Bose-Chaudhuri-Hocquenghem (BCH) Error Correction Codes (ECC) and a hash function [3]–[7]. However, they entail high computational complexity and a large area overhead.

FEs require helper data which is additional information used to derive a correct key. Because the helper data is stored in public storage, it should not include any information to deduce the key or PUF responses. Index-Based Syndrome (IBS) coding scheme is one of FE architectures and it uses an index (i.e. pointer referring to PUF outputs) as helper data [7]. Indices don't have any information about the responses and key, hence IBS scheme is still secure when helper data (i.e. index information) is exposed by unauthenticated attacker. However, IBS also uses a strong error correction method, so it requires a high computation complexity and area overhead.

In this paper, we propose a novel architecture, which is a simple, lightweight, and attack-resistant structure based on the classical IBS paradigm. This paper avoids the use of additional complex compensation logic like ECC to handle unstable responses. Instead, the proposed method manipulates PUF responses via multiplications and it compares multiple sub-key candidates generated from the manipulated PUF responses using a majority voter. The proposed method offers several advantages. 1) The key is safe even if the helper data is exposed by an attacker. The helper data only contains the location pointer which is nothing more than a number. 2) The proposed architecture can be implemented using simple logic with adder and shift register. 3) Despite the above advantages, the proposed architecture has a similar level of reliability of the conventional key generation models.

The rest of the paper is organized as follows. Section II

978-3-9819263-4-7/DATE20/©2020 EDAA

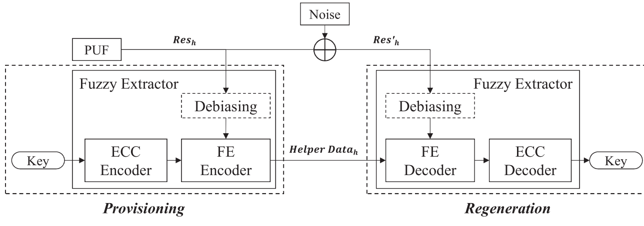


Fig. 1. PUF-based Key Generation with Fuzzy Extractor

explains the related works about PUF-based key generation methods. Section III introduces the proposed method and its probability model is given in Section IV. Section V provides an evaluation and analysis and Section VI concludes the paper.

II. Background and Related Works

In the PUF based key generation, Fuzzy Extractor (FE) is used to extract a stable value from a noisy PUF response [3]. The key generation consists of two stages 1) *Provisioning Step* – Generate helper data through initial key encoding and 2) *Regeneration Step* – Retrieve correct keys using helper data. Fig. 1 illustrates a key generation process for each step. In *Provisioning Step*, an initial key is fed to the ECC encoder to generate a codeword. The FE Encoder generates helper data using the codeword and PUF response, and the helper data is stored in public storage. In *Regeneration Step*, the codeword is recovered from FE Decoder using the helper data generated in Provisioning Step and the PUF response. Then, the codeword is decoded and the key is finally found.

Code-offset, Pattern Matching Key Generation (PMKG), Index-Based Syndrome (IBS) and Differential Sequence Coding (DSC) are examples of FE encoding/decoding methods. For Code-offset, in Provisioning Step, FE performs a bit-wise parallel XOR operation to generate a helper data with a PUF response and a codeword. In Regeneration Step, the bit-wise parallel XOR operation is conducted with a PUF response and the helper data to retrieve the codeword [3]. PMKG and IBS have different encoding/decoding methods than Code-offset. In Provisioning Step, PMKG uses a codeword to choose a sub-pattern of PUF response and uses the sub-pattern to a helper data [8], [9]. IBS coding uses an index to helper data which is obtained by calculating the predefined rule of the codeword and PUF response [5], [7]. DSC uses distance pointer and inversion bit to helper data which is generated by the codeword and PUF response [10].

To resolve the PUF reliability issues, conventional FEs use ECC logic such as Reed-Muller (RM) code [11], BCH code and Golay code [12]. In addition, some FEs use debiasing methods such as Von-Neumann Corrector (VNC) [13], Coset Code (CC) [14], Biased Masking (BM) [15] and IBS, to remove an entropy leakage of helper data derived from biased PUF responses. Note that IBS is not only encoding/decoding methods, but debiasing method.

III. Proposed Architecture

This section introduces VIVA (Various Index Voting Architecture), a new index-based key generation architecture, which has a low area overhead and low risk of information leak. Fig. 2 shows a basic architecture of the proposed method for two stages (Provisioning and Regeneration) of key generation. VIVA operates multiple iterations for key generation, and multiple helper data is generated for each round to achieve a reliable operation by enhancing an error correction capability. Fig. 3 depicts a full structure of VIVA.

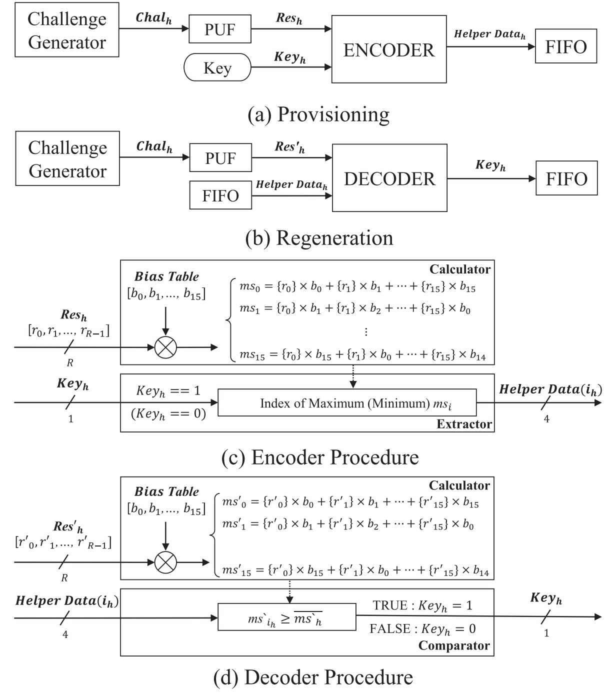


Fig. 2. Provisioning and Regeneration Architecture

A. Provisioning

Fig. 2(a) shows a provisioning process at h^{th} round where $1 \leq h \leq H$. The encoder receives a single-bit Key_h and an *R*-bit PUF response (Res_h) generated by a challenge $(Chal_h)$. Res_h can be also represented as $\{r_0, r_1, \ldots, r_{R-1}\}$, where r_α is a single-bit PUF response for $0 \leq \alpha \leq R-1$. As shown in Fig. 2(c), the encoder is composed of a *Calculator* receiving Res_h and an *Extractor* receiving Key_h.

The proposed method manipulates Res_h by multiplying it with a bias table. The bias table consists of predefined parameters to transform a response into various responses. There are 16 elements in the bias table, b_k $(0 \leq k \leq 15)$, which are powers of 2 values satisfying $b_0 \leq b_1 \leq \cdots \leq b_{15}$. One bias table is used for all rounds, *H*. For b_k, any integer value can be used, however, powers of 2 value make the implementation of multiplication simple with shift registers which requires a relatively low area overhead.

To reduce the overhead even further, the proposed method limits the range of b_k to $1 \leq b_k \leq 16$. Because there are 16 elements ($b_0 \sim b_{15}$), *R*-bit Res_h is partitioned into 16 groups, and each group, $\{r_k\}$, can be found using Equation 1.

$$\{r_k\} = \sum_{\alpha=uk}^{u(k+1)-1} 2^{\alpha \ mod \ u} \times r_\alpha, (0 \leq k \leq 15, u = \frac{R}{16}) \quad (1)$$

For example, if Res_h is 32-bit long, each $\{r_k\}$ group has two bits. If $\{r_0\}$ = 10, this corresponds to the value, 2 (= $1 \times 2 + 0 \times 1$). *Calculator* multiplies $\{r_k\}$ and b_k by shifting b_k for 16 times. Equation 2 expresses the multiplication sum.

$$ms_i = \sum_{k=0}^{15} \{r_k\} b_{(k+i) mod 16} \ (0 \leq i \leq 15) \tag{2}$$

ms_0 is first calculated as $\{r_0\} \times b_0 + \{r_1\} \times b_1 + \cdots + \{r_{14}\} \times b_{14} + \{r_{15}\} \times b_{15}$ and the result is stored. Then, *Calculator* calculates ms_1 by shifting b_k by one, and ms_1 can be expressed as $\{r_0\} \times b_1 + \{r_1\} \times b_2 + \cdots + \{r_{14}\} \times b_{15} + \{r_{15}\} \times b_0$. The multiplication continues until all $ms_0 \sim ms_{15}$ are found and stored, shown in Fig. 2(c).

Once the multiplication is done, *Extractor* encodes the data using Key_h. The proposed method follows a maximum or minimum policy according to Key_h. That is, if *Key* is 1, the maximum ms_i is selected; otherwise, the minimum ms_i is selected. At the same time, the index corresponding to the chosen ms_i is selected. Equation 3 explains how the index is determined. The extracted index is referred to as helper data, i_h, and i_hs extracted from all *H* rounds are stored in *FIFO*, as shown in Fig. 2(a).

$$i_h = \begin{cases} index(MAX(ms_i)) & \text{if } Key_h = 1 \\ index(MIN(ms_i)) & \text{if } Key_h = 0 \end{cases} \tag{3}$$

B. Regeneration

As described in Fig. 2(b), regeneration step restores the original key, Key_h, using two inputs: Res'_h which is generated by $Chal_h$, and i_h which is stored as helper data in provisioning. The decoder consists of two sub-modules, *Calculator* and *Comparator* like Fig. 2(d).

It should be noted that *Calculator* is shared for provisioning and regeneration. Hence, in the same manner as provisioning, ms's are generated using Res'. However, there might be a chance that ms' would be different from ms if a noisy Res'_h is generated from PUF with the same $Chal_h$ used for generating Res_h in provisioning. Instead of extracting Key_h using ms', the proposed method finds the average of ms', $\overline{ms'}$, once the multiplication is done.

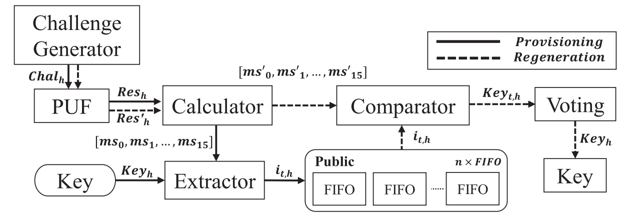
$$\overline{ms'} = \frac{1}{16} \sum_{i=0}^{15} ms'_i = \frac{1}{16} \sum_{i=0}^{15} \sum_{k=0}^{15} \{r'_k\} b_{(k+i) mod 16} \tag{4}$$

Then, using the stored i_h, $\overline{ms'}$ is compared with ms' to determine Key_h. The proposed method uses Equation 5 is how to conclude Key_h.

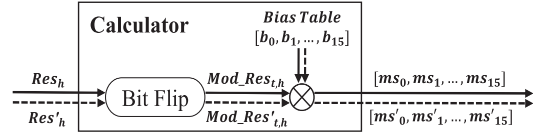
$$Key_h = \begin{cases} 1 & \text{if } ms'_{i_h} > \overline{ms'_h} \\ 0 & \text{if } ms'_{i_h} \leq \overline{ms'_h} \end{cases} \tag{5}$$

C. Voting scheme

To compensate the reliability issue, a majority voting scheme is used in the proposed design. Fig 3(a) shows an example of the key generation process with t^{th} helper data at h^{th} key generation round, where $1 \leq t \leq n$ and $1 \leq h \leq H$ (i.e., each key generation round includes *n* helper data generation).



(a) The proposed key generation architecture with multiple helper data



(b) Advanced calculator procedure

Res : $r_0\ r_1\ r_2\ r_3 \ldots r_{R-2}\ r_{R-1}$

Mod_Res₁ : $r_0\ r_1\ r_2\ r_3 \ldots r_{R-2}\ r_{R-1} \oplus 1\ 0\ 1\ 0 \ldots 1\ 0$ (*Bit_Flip info*)
$= \overline{r_0}\ r_1\ \overline{r_2}\ r_3 \ldots \overline{r_{R-2}}\ r_{R-1}$

Mod_Res₂ : $r_0\ r_1\ r_2\ r_3 \ldots r_{R-2}\ r_{R-1} \oplus 0\ 1\ 0\ 1 \ldots 0\ 1$ (*Bit_Flip info*)
$= r_0\ \overline{r_1}\ r_2\ \overline{r_3} \ldots r_{R-2}\ \overline{r_{R-1}}$

Mod_Res₃ : $r_0\ r_1\ r_2\ r_3 \ldots r_{R-2}\ r_{R-1} \oplus 1\ 1\ 0\ 0 \ldots 0\ 0$ (*Bit_Flip info*)
$= \overline{r_0}\ \overline{r_1}\ r_2\ r_3 \ldots r_{R-2}\ r_{R-1}$

Mod_Res₄ : $r_0\ r_1\ r_2\ r_3 \ldots r_{R-2}\ r_{R-1} \oplus 0\ 0\ 1\ 1 \ldots 1\ 1$ (*Bit_Flip info*)
$= r_0\ r_1\ \overline{r_2}\ \overline{r_3} \ldots \overline{r_{R-2}}\ \overline{r_{R-1}}$

(c) An example of bit flip operation

Fig. 3. Various Index Voting Architecture (VIVA)

To generate various helper data with a given response (Res_h), the response is transformed into different responses via bit-flipping in *Calculator*, illustrated in Fig 3(b). *Bit_Flip* logic provides (*n*-1) unique *R*-bit binary sequences which are composed of the same number of *1*'s and *0*'s. The response is XORed with the first bit-flipping sequence and this would make the first modified response (*Mod_Res*). This process iterates (*n*-1) times and (*n*-1) different *Mod_Res* are found. In *Provisioning Step*, for each round, the scheme generates *n* helper data using one *Res* and (*n*-1) *Mod_Res* and stores them in a public storage. An example with $n = 5$ is given in Fig. 3(c). In addition to the one ***Res***, this shows how four different *Mod_Res* are found. In *Regeneration Step*, for each round, the scheme generates key candidates $Key_{t,h}$ using *n* helper data. Then, the voter chooses the Key_h which is a major value among key candidates. For (*n*-1) bit-flipping sequences, the same logic can be shared.

IV. Failure Probability

The proposed method finds a correct key with noisy PUF responses by using the margin which is the difference between the mean and the maximum (or minimum) value of ms_i. A failure of key generation occurs when Res'(response in Regeneration Step) is different from Res (response in Provisioning Step). The difference between Res and Res' can be found by comparing them and the number of erroneous

bits (e) is expressed with a hamming distance between them, $HD(Res, Res')$. e can be rephrased as $\sum_{k=0}^{R-1} HD(r_k, r'_k)$. Note that each r_k is independently generated from identical PUF with an average bit error probability p. Therefore, the e distribution of Res' can be represented as

$$P(X = e) = f(e, R, p) = \binom{R}{e} p^e (1-p)^{R-e} \tag{6}$$

Through H round iterations, VIVA generates a full H-bit key by concatenating one-bit Key_h. Because each single-bit Key_h for each round is used to make a final full H-bit key, the failure probability of the full H-bit key, $P_{fail_{total}}$, can be calculated as follows.

$$P_{fail_{total}} = 1 - (1 - P_{fail_{round}})^H \tag{7}$$

$P_{fail_{round}}$ is found by the discrepancy between the original key used for provisioning step and the generated key from regeneration step. Hence, $P_{fail_{round}}$ can be expressed with a conditional probability as follows

$$\begin{aligned} P_{fail_{round}} =& P_{key_{pro}=1} \times P(key_{rgn} = 0 | key_{pro} = 1) \\ &+ P_{key_{pro}=0} \times P(key_{rgn} = 1 | key_{pro} = 0) \\ =& P_{key_{pro}=1} \times P_{f_1} + P_{key_{pro}=0} \times P_{f_0} \end{aligned} \tag{8}$$

, where key_{pro} is the key from provisioning stage and key_{rgn} is the key from regeneration stage. Because key_{pro} is either '1' or '0', the sum of $P_{key_{pro}=1}$ and $P_{key_{pro}=0}$ is 1 ($P_{key_{pro}=1} + P_{key_{pro}=0} = 1$). The upper (or lower) margin is defined as the difference between the mean and the maximum (or minimum) each round. Using a Bias Table which is a set of bias having a Gaussian distribution, the upper and lower margins have similar distribution. Then, P_{f_1} and P_{f_0} can be considered to be approximately the same, and Equation 8 can be replaced by

$$\begin{aligned} P_{fail_{round}} &= (1 - P_{key_{pro}=0}) \times P_{f_1} + P_{key_{pro}=0} \times P_{f_0} \\ &= P_{f_1} \ (\because P_{f_1} \simeq P_{f_0}) \end{aligned} \tag{9}$$

Based on the definition of P_{f_1}, it is the probability that an opposite key is generated when bit error occurs in responses from PUF. This means that the key generation failure in the proposed method would occur when provisioning determines the index i (giving the largest ms value) and ms'_i is smaller than or equal to $\overline{ms'}$. This can be expressed as follows

$$\begin{aligned} P_{f_1} =& P(key_{rgn} = 0 | key_{pro} = 1) \\ =& \sum_{i=0}^{15} \sum_{e=0}^{R} P(ms'_i \le \overline{ms'} | X = e, Index = i, key_{pro} = 1) \\ & \times P(X = e) \times P(Index = i) \end{aligned} \tag{10}$$

where $Index = i$ is the case that ms_i is the maximum in provisioning.

The summation of 16 indices in Equation 10 can be replaced by one index term because they all have an equal probability. To prove this, the National Institute of Standards and Technology (NIST) statistical random tests are performed using 4-bit indices derived from a Xilinx Zedboard FPGA implementation. The input sequence of 1.3 million bits is used for the random tests. It is generated by serially concatenating 4-bit indices stored as helper data. The frequency of all indices is the same. From the results of the random tests presented in Table I, it can be concluded that each index is independent of each other and has the same probability. Based on this analysis, Equation 10 can be substituted for Equation 11. The margin, which is dependent on bias table and *HD*(Res, Res'), has a significant impact on $P_{fail_{round}}$ (=P_{f_1}). The optimal b_k in the bias table, which can be predefined through experiments, will be discussed in the next section. Therefore, the margin is determined by the bit error rate in responses.

TABLE I
NIST STATISTICAL TEST RESULTS FOR HELPER DATA

Test ($1.3 \times 10^6 bits$)	*P value*	*Authenticity*
Frequency	0.492371	Pass
Block Frequency	0.444949	Pass
Cumulative Sum (F)	0.685796	Pass
Cumulative Sum (R)	0.737460	Pass
Runs	0.353819	Pass
Longest Runs	0.400656	Pass
Rank	0.756401	Pass
FFT	0.027507	Pass
NonOverlapping Template	0.531349	Pass
Overlapping Template	0.132493	Pass
Universal	0.538524	Pass
Approximate Entropy	0.251506	Pass
RandomExcursions	0.294756	Pass
RandomExcursionsVariant	0.436115	Pass
Serial	0.083268	Pass
Linear Complexity	0.793228	Pass

$$\begin{aligned} P_{f_1} &\simeq \sum_{e=0}^{R} (P(X = e) \times e^*) \\ e^* &= P(ms'_0 \le \overline{ms'} | X = e, Index = 0, key_{pro} = 1) \end{aligned} \tag{11}$$

From Equation 11, it is expected that the reliability drops sharply as the error bit rate of responses increases. Various indices help to reduce the failure probability of key generation than having a single index. The bias table achieving higher reliability can be selected, however, it would be still difficult to satisfy 100% reliability for all responses. The proposed method with multiple helper data increases the success rate of regeneration by alleviating irregular and non-linear characteristics of Res' with multiple Mod_Res'.

The proposed design uses multiple helper data to restore the key through voting each round. In this case, the failure occurs when more than half of the key candidates fail, so it can be expressed via binomial distribution. Hence, the failure probability of a single round with *n* helper data, ${P^n}_{fail_{round}}$, can be expressed as Equation 12.

$${P^n}_{fail_{round}} = \sum_{t=\lceil n/2 \rceil}^{n} \binom{n}{t} {P_{fail_{round}}}^t (1 - P_{fail_{round}})^{n-t} \tag{12}$$

Using Equation 12, the total failure probability after *H* rounds can be calculated as Equation 13.

$$P_{fail_{total}} = 1 - (1 - {P^n}_{fail_{round}})^H \tag{13}$$

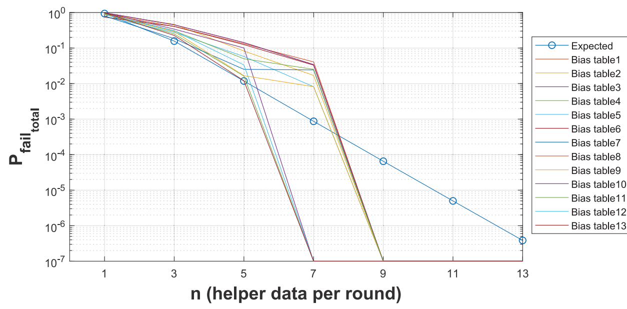


Fig. 4. The failure probability via 13 bias tables for different helper data size and PUF responses with p = 0.15 on R = 16

V. Evaluation And Analysis

In this section, experimental results and analyses are given for the proposed architecture. To evaluate VIVA, reliability, attack resistance, and area overhead are analyzed.

A. Reliability Evaluation

128-bit key generation simulations for 10 million iterations are performed with an average bit error probability $p = 0.15$, which is considered to be a conservative estimate [12] and is commonly used in the PUF-based key generation performance analysis. To confirm the failure probability $P_{fail_{total}} \leq 10^{-6}$ (referred by [12]) by the proposed method, simulations are performed incrementally increasing the number of helper data per each round, with respect to different response length (R) and various bias tables.

The failure probability results with growing the number of helper data per round when R is 16-bit ans 32-bit. We assume 2-bit and 5-bit errors occurring on average from 16-bit and 32-bit responses respectively because they are the mean values following the binomial distribution based on Equation 6. In addition, the same bias table $\{1, 1, 2, 2, 4, 4, 4, 4, 4, 4, 8, 8, 8, 16, 16, 16\}$ is used. For the expected $P_{fail_{total}}$, $P_{fail_{round}}$ is calculated by Equation 11. For calculation, e^* needs to be determined. It can be estimated for all responses through simulations based on the failure case of subkey in a round, according to the number of errors ($0 \leq e \leq R$). With e^* and the probability based on the Equation 6, ${P^{n=1}}_{fail_{round}}$ is determined as 2.1260×10^{-2} at $R = 16$ and 3.6095×10^{-2} at $R = 32$. Hence, using Equation 12 and 13, the expected $P_{fail_{total}}$ for each n can be calculated. As the number of helper data (n) changes to odd numbers (1,3,5,...,15), the probability for response length of 16 was shown as 9.36E-1, 1.57E-1, 1.18E-2, 8.69E-4 which is in accordance with the theoretical value.

Fig. 4 shows $P_{fail_{total}}$ with different bias tables when $R = 16$. For the same n, $P_{fail_{total}}$ varies with different bias tables. This means that a margin would vary depending on which bias table is used. $P_{fail_{total}}$ becomes smaller as n increases and $P_{fail_{total}}$ becomes lower than 10^{-7} when $n = 9$ regardless of the bias table used. When more than 7 helper data are used, the failure probability sharply saturates to 0 while the theoretical value continues to decrease. This possibly happens because the theoretical analysis assumes each helper data in helper data set as independent one while, in the experiments, helper data in helper data set are manipulated from one initial response. Experimental results in Fig. 4 show that, as more bit flip values are used, the relevance between the bit flip values appears to be becoming stronger and this helps the proposed method regenerate the correct key information.To find a bias table giving the smallest $P_{fail_{total}}$ (i.e., to find an optimal bias table), one-time simulation at the manufacturing stage is needed with PUF responses from the manufactured chip and the bias elements need to be programmed.

B. Attack Resistance Evaluation

Helper data are stored in public storage and attackers can easily access the helper data set (a group of helper data generated in one round). In addition, the predefined system information such as Bias Table and Bit Flip data could be leaked through side-channel attacks [16]. Thus, to prevent attackers from deducing the key information, the keys should not be speculated using the information and helper data set. Using the system information, via simulations, an attacker can obtain all cases of helper data from all cases of PUF responses. However, if the same number of helper data sets appear with the same number of occurrence for the Key is 1 and 0 states, the system is still safe even if the helper data set is exposed.

TABLE II
THE COMPARISON TO THE NUMBER OF OCCURENCE IN EACH CASE OF MIN AND MAX FOR EACH HELPER DATA SET WITH R=16

n	3	5	7	9	11	13	15
Total	2343	26639	50453	59524	64457	65224	65396
Same	2343	26639	50453	59524	64457	65224	65396
Diff	0	0	0	0	0	0	0

To evaluate the attack resistance, using the given bias table, we calculate all helper data sets for all cases that can be made with a response length of 16 bits. For the odd number of helper data (n from 3 to 15), we calculate the helper data set and compare the number of helper data occurrence in each case of Min and Max for each helper data set. As Table II shows the same number appears for all the helper data set. This means that predefined system values (such as Bias Table and Bit Flip data) do not provide any information about the key. The security entropy only relies on the length of the key (H). Each bit of the key can be 1 or 0 and it always guarantees to be H entropy. This makes VIVA resistant to modeling and machine learning attacks as well.

C. Area Overhead Analysis

VIVA is synthesized and implemented using Xilinx ISE on Xilinx Spartan-3E FPGA with 50MHz clock frequency with $R = 16$ and $n = 7$. Table III compares VIVA implementation results with previous approaches for generating 128-bit key with $P_{fail_{total}} \leq 10^{-6}$. VIVA needs ($H \times R$) PUF response bits and ($H \times n \times \log_2 R$) helper data bits because each helper data is 4-bit long.

TABLE III
FPGA IMPLEMENTATION OF REGENERATION PROCEDURES OF VIVA AND PREVIOUS WORKS

	Code-Offset + Golay [12]	Code-Offset + RM [6]	C-IBS + RM [5]	Compressed DSC + Conv. code [10]	VIVA
PUF Output Bits	3,696	1,536	2,304	1,224	2,048
Helper Data Bits	3,824	13,952	9,216	1,224	3,584
Slices	$\geq$ 907	237	250	272	166
Block RAM Bits	0	32,768	0	11,264	0

TABLE IV
FPGA IMPLEMENTATION OF VIVA

	Encoder	Decoder
Slices	165	166
Registers	157	151
Logic LUTs	272	276
Block RAMs	0	0
Critical Path	10.953 ns	10.886 ns

TABLE V
ASIC IMPLEMENTATION OF VIVA

Res = 16	Extractor	Comparator	Calculator	VIVA
Area (μm^2)	249.18	453.23	1649.83	3342.71
Cell	62	86	316	208
Power (μW)	84.44	144.46	560.53	1021.30

When compared with [10], VIVA requires 60% more response bits and 2X more helper data bits. While [10] significantly reduces helper data using compression techniques, the helper data randomness in the proposed method, as demonstrated with NIST test in Sec. IV, would limit the level of compression. However, it should be noted that the higher helper data randomness means the lesser correlation among helper data and this makes helper data-based attacks harder.

Table IV shows an FPGA implementation of VIVA. Because *Extractor*, *Comparator* and *Calculator* are implemented with simple logic, VIVA is implemented with a smaller number of slices as shown in Table III. VIVA can be implemented with 60$\sim$70% of slices used for [5], [6], [10], only with 20% of slices required than [12]. In addition, no Block RAM bits are needed for VIVA implementation which significantly saves an area.

VIVA is also synthesized with 32nm standard cell library. The results are provided in Table V with area, cell number and power consumption. Implementation details and results show the lightweight architecture of the proposed method.

VI. CONCLUSION

We have presented a reliable and lightweight index-based key generation architecture using PUF. The proposed architecture, VIVA, delivers the following advantages. VIVA does not rely on additional and complex error correction logic for ensuring the high reliability of key generation. This makes the implementation simpler and less costly. As results show VIVA retains the high security entropy with public data for randomness and unpredictability. This makes the proposed architecture robust against helper data based attacks.

ACKNOWLEDGMENT

This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-TB1803-02.

REFERENCES

[1] G. E. Suh and S. Devadas, “Physical Unclonable Functions for Device Authentication and Secret Key Generation,” in *2007 44th ACM/IEEE Design Automation Conference*, June 2007, pp. 9–14.

[2] C. Herder, M. Yu, F. Koushanfar, and S. Devadas, “Physical Unclonable Functions and Applications: A Tutorial,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, Aug 2014.

[3] Y. Dodis, L. Reyzin, and A. Smith, “Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data,” in *Advances in Cryptology - EUROCRYPT 2004*, C. Cachin and J. L. Camenisch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 523–540.

[4] W. Yan, F. Tehranipoor, and J. A. Chandy, “Puf-based fuzzy authentication without error correcting codes,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1445–1457, 2016.

[5] M. Hiller, D. Merli, F. Stumpf, and G. Sigl, “Complementary IBS: Application specific error correction for PUFs,” in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 2012, pp. 1–6.

[6] R. Maes, P. Tuyls, and I. Verbauwhede, “Low-overhead implementation of a soft decision helper data algorithm for SRAM PUFs,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2009, pp. 332–347.

[7] M. Yu and S. Devadas, “Secure and Robust Error Correction for Physical Unclonable Functions,” *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 48–65, Jan 2010.

[8] Z. Paral and S. Devadas, “Reliable and efficient PUF-based key generation using pattern matching,” in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, June 2011, pp. 128–133.

[9] J. Delvaux and I. Verbauwhede, “Attacking PUF-Based Pattern Matching Key Generators via Helper Data Manipulation,” in *Topics in Cryptology – CT-RSA 2014*, J. Benaloh, Ed. Cham: Springer International Publishing, 2014, pp. 106–131.

[10] M. Hiller and G. Sigl, “Increasing the efficiency of syndrome coding for PUFs with helper data compression,” in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[11] G. Schnabl and M. Bossert, “Soft-decision decoding of Reed-Muller codes as generalized multiple concatenated codes,” *IEEE transactions on information theory*, vol. 41, no. 1, pp. 304–308, 1995.

[12] C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls, “Efficient helper data key extractor on FPGAs,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2008, pp. 181–197.

[13] R. Maes, V. van der Leest, E. van der Sluis, and F. Willems, “Secure key generation from biased PUFs: extended version,” *Journal of Cryptographic Engineering*, vol. 6, no. 2, pp. 121–137, 2016.

[14] M. Hiller and A. G. Önalan, “Hiding secrecy leakage in leaky helper data,” in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 601–619.

[15] R. Ueno, M. Suzuki, and N. Homma, “Tackling Biased PUFs Through Biased Masking: A Debiasing Method for Efficient Fuzzy Extractor,” *IEEE Transactions on Computers*, vol. 68, no. 7, pp. 1091–1104, July 2019.

[16] D. Merli, D. Schuster, F. Stumpf, and G. Sigl, “Side-channel analysis of PUFs and fuzzy extractors,” in *International Conference on Trust and Trustworthy Computing*. Springer, 2011, pp. 33–47.