PUF-RAKE: A PUF-based Robust and Lightweight Authentication and Key Establishment Protocol

Mahmood Azhar Qureshi, *Student Member, IEEE,* and Arslan Munir, *Senior Member, IEEE*

Abstract—Physically unclonable functions (PUFs) bind a device's identity to its physical hardware and thus, can be employed for device identification, authentication and cryptographic key generation. However, PUFs are susceptible to modeling attacks if a number of PUFs' challenge-response pairs (CRPs) are exposed to the adversary. Furthermore, many of the embedded devices requiring authentication and inter-device communication in a real-time environment/system have stringent resource and low latency requirements, and thus require a lightweight authentication and key establishment mechanism to quickly realize an authenticated and secure connection. We propose PUF-RAKE, a PUF-based lightweight, highly reliable authentication and key establishment scheme. The proposed scheme enhances the reliability of PUF as well as alleviates the resource constraints by employing error correction in the server instead of the device as well as removing cryptographic hashing required by earlier PUF-based protocols. The proposed PUF-RAKE is robust against masquerade, brute force, replay, and modeling attacks. In PUF-RAKE, we introduce an inexpensive yet secure stream authentication scheme inside the device which authenticates the server before the underlying PUF can be invoked. This prevents an adversary from brute forcing the device's PUF to acquire CRPs essentially locking out the device from unauthorized model generation. Additionally, we also introduce a lightweight CRP obfuscation mechanism involving XOR and shuffle operations. The security of PUF-RAKE has been formally verified. A prototype of the protocol has been implemented on two Xilinx Zynq 7000 system-on-chips with one present on Xilinx zc706 evaluation board and the other present on the Avnet Zedboard. Observations, security analysis and results verify that the PUF-RAKE is secure against a probabilistic polynomial time adversary under both the unauthenticated link and authenticated link adversarial models while providing ∼99% reliable authentication. In addition, PUF-RAKE provides a reduction of 60% and 72% for look-up tables (LUTs) and register count, respectively, in the programmable logic (PL) part of the Zynq 7000 as compared to a recently proposed approach while providing additional advantages.

Index Terms—Authentication, key establishment, PUFs, security, reliability, lightweight, bit shuffling

1 Introduction

Data security between embedded communicating devices presents one of the major challenges in designing today's complex infrastructure and cyber-physical systems spanning different domains including medical, defense, transportation, agriculture, and automation. The applications requiring secure data transmission include Internet of things (IoT), in-vehicle network communication in self-driving cars, vehicle-to-vehicle (V2V), vehicle-to-infrastructure (V2I) communication, smart grid communication, and many more. Often devices in a network can generate massive amounts of data relevant to their *status*. This data needs to be secured against an unauthorized entity as the leakage of this data can have wide reaching consequences including identity theft and fraudulent verification.

Fig. 1 depicts an authentication scenario for various devices. In essence, communication between two devices in a network is a two-prong process, the first one being authentication, and the second being secret key establishment. Traditionally, the tasks of authentication and key exchange have been handled by public key encryption schemes. The two most widely used paradigms for public key encryption are public key infrastructure (PKI) and identity based encryption (IBE) [1]. Different protocols have been developed in [2] to address the intellectual property (IP) protection problem on field-programmable gate arrays (FPGAs) using physically unclonable functions (PUFs) [3] and PKI-based public key cryptography. However, traditional PKI approach has been afflicted by several shortcomings, the most important being the distribution and handling of certificates by a trusted third party to potentially billions of devices. This makes it highly infeasible for resource-constrained deployments. IBE seems to be a better alternative than PKI, however, IBE utilizes a public key generator (PKG) to generate and distribute private keys to nodes over secure channels. This makes key exchange cumbersome and difficult to manage when there are billions of devices. Moreover, the nodes need to store some secrets within their non-volatile secure memories (NVM) which is infeasible for cost and resource constraint devices. Furthermore, the secrets stored in NVM of devices can be extracted by an adversary. To overcome the deficiencies in previous works, we propose a protocol which uses the hardware secrets generated by PUFs in a lightweight authentication scheme and realizes a secure masking function as a message authentication mechanism using one-time session nonces. This effectively removes the requirement of PKG because the identity of the devices is tied to a hardware specific *unclonable* instance

- *M.A. Qureshi and A. Munir are members of Intelligent Systems, Computer Architecture, Analytics, and Security (ISCAAS) Lab, Department of Computer Science, Kansas State University, Manhattan, KS, 66506.*

E-mail: mahmood102@ksu.edu, amunir@ksu.edu

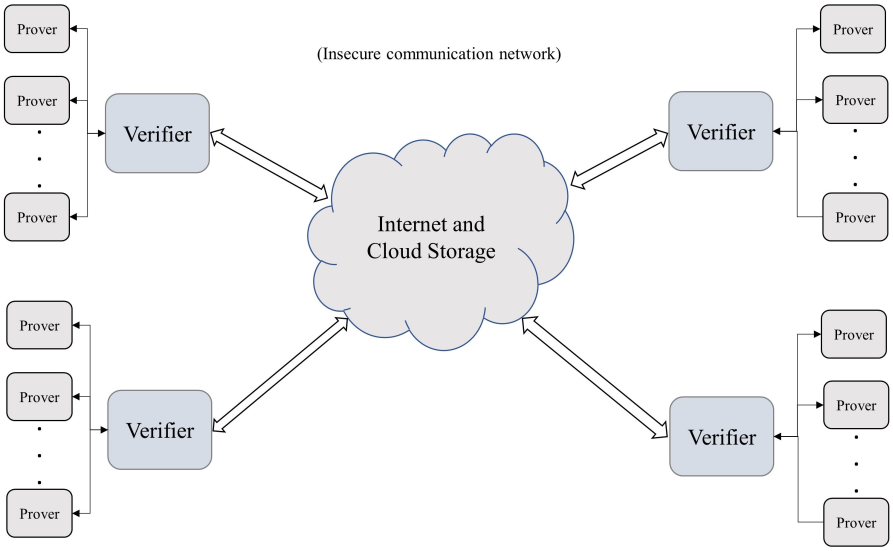

Fig. 1. Authentication scenario for various devices.

within the device and does not require any on-chip NVM.

Ever since the introduction of PUFs more than one and a half decade ago, extensive research has been done in using these uncontrollable manufacturing variations for enhancing the device's security. With the advent of advanced machine learning (ML)-based modeling techniques [4], strong PUFs (SPUFs), previously considered secure, now have their security in question. Given a number of challenge-response pairs (CRPs) of a 64x64 *Arbiter PUF* [5], an adversary can build a soft model for the device with a prediction accuracy of 99.9% [6]. This is due to the fact that a plain arbiter PUF follows a linear additive delay model [4] and given enough CRPs, an adversary can very accurately determine the parameters of the model governing the PUF circuit. Many approaches have been presented [7], [8] which add non-linearities into the PUF circuits to thwart model building attacks. However, as shown in [7], these PUFs are still susceptible to modelling attacks.

Controlled PUFs (CPUFs) [9] are another class of SPUFs which enhance the security and resistance against ML-based modeling. These PUFs thwart model-building attacks by wrapping the PUF inside a control logic. One approach is to build the control logic in such a way as to limit the exposure of CRPs for the adversary [10], [11]. Another approach is to obfuscate the CRPs in such a way that even if the adversary can collect a number of obfuscated CRPs, no effective model can be built since the original CRP relationship is only known to the device and the verifier [6].

Reliability is another factor which plagues the usage of PUFs in communicating devices. Because the embedded devices need to operate under varying environmental conditions, the PUFs within the devices should be reliable enough. A PUF, for a given n-bit challenge **C**, is a mapping α to a particular m-bit response **R**, that is, $\alpha : \{0,1\}^n \rightarrow \{0,1\}^m$. Ideally, this mapping for a particular challenge **C** to a fixed response **R** should always hold under varying environmental conditions. This type of ideality however, is not possible in hardware as the response of a PUF to a particular challenge is dependent on the physical characteristics of the device. Under varying conditions (e.g., temperature, voltage etc.), these physical characteristics differ, resulting in generation of responses with bit flips associated with errors. Thus, the output response under varying environmental conditions is $\mathbf{R'} \neq \mathbf{R}$. Majority voting can help to reduce the errors but it does not guarantee high reliability under highly variable conditions. Contemporary approaches like [9] use error correction codes (ECC) in the device as a fix for the PUF's reliability problem. These approaches do not consider the high hardware area overhead associated with computationally expensive error correction schemes in a low cost device. Moreover, ECC requires helper data to be communicated to the device by the server during an authentication round. This exposure of helper data provides another attack vector to the adversary who can use this information for modeling as well as side-channel analysis [12]. We employ a different approach where the error correction is not present in the device, rather, the server is responsible for correcting the noisy responses of the device's PUF. By employing this, we not only guarantee a reliability of $\sim$99% but also make the device extremely lightweight.

Our main contributions are as follows:

- We develop a novel, lightweight *masking* function which serves two main purposes: (i) protecting the device's PUF by obfuscating its CRPs, and (ii) providing a lightweight solution for verifying the data integrity and message authenticity. Also, no challenge or response in its original form is exposed on any communication link (i.e., between the device $\Longleftrightarrow$ server and server $\Longleftrightarrow$ database). Unlike some of the previous approaches, which use cryptographic hash functions with high hardware overhead, inside the device for response obfuscation, our scheme provides an extremely lightweight, low hardware cost dynamic obfuscation mechanism. It also acts as a lightweight message authentication code (HMAC) using dynamic, one time keys instead of a hard HMAC IP based on traditional, high overhead hashing schemes using a secured key. We also formally prove the security of our masking function.
- In the proposed scheme, the access to the underlying PUF in the device is strictly controlled. No challenge

is issued and thus no response is generated unless a correct input stream is applied to the device, essentially locking the device from unauthorized access. This scheme is the first that locks out the device without even invoking the PUF, and thus completely inhibits any model-building as well as side-channel analysis attack on the underlying PUF.

- The proposed scheme, unlike all previously introduced schemes, improves the PUF's response accuracy and therefore, the reliability, by employing error correction in the server instead of the device. The device sends the noisy, *masked* responses to the server which, after unmasking, corrects any underlying bit errors. This greatly reduces the hardware overhead of the devices in the system without compromising the reliability which makes this protocol easily deployable in low cost devices operating in challenging environmental conditions.
- The proposed protocol extends the authentication and also includes a secure key establishment phase which makes it deployable to IoT-based systems. Unlike traditional PKI, this scheme does not require any Certificate Authority (CA) for signing and issuing digital certificates nor does it require any PKG for issuing public/private key pairs. This is due to the fact that the server/verifier can validate the authenticity of the device by using its PUF instance. Thus, the PUF acts as a *root-of-trust* inside the device. The device uses the run-time variables in conjunction with the masking function to authenticate the public/private key pairs. This not only reduces the computational complexity involved with hash-based schemes, but also drastically reduces the space-time complexity, as no key pairs need explicit storage.
- We implement the entire scheme on hardware and show the area, latency and communication overheads associated with the protocol in the device hardware. We also show how the proposed protocol is better than the previously proposed schemes in terms of device overheads.

It should be noted that this work is an extension of our previously proposed scheme, presented the first time in [13]. In our previous work, the device had to keep a track of the masking counters for successive authentication cycles which introduced a security vulnerability. In this work, we eliminate that need and the device no longer needs to keep track of any variable. Assuming that for the j^{th} authentication round, the masking variables are $k_1, k_2, ..., k_n$, then for $j+1^{st}$ authentication round, the masking variables $l_1, l_2, ...l_n$ have no correlation to $k_1, k_2, ..., k_n$. Thus, all the randomly generated variables during one authentication cycle are destroyed immediately once the authentication is successfully completed and a different set of randomly generated variables are used for the next cycle.

2 Related Work

Various schemes have been proposed in the past that implement a controlled strong PUF for authentication of devices. Gassend *et al.* [9] have proposed hashing of the input challenge and the response. However, this configuration requires hardware-expensive hashing as well as error correction logic in the device which makes it highly infeasible for low cost platforms. Also, the server in [9] needs to send the raw helper data to the device for stabilizing the noisy PUF responses. This exposes the PUF to attacks focusing on side-channel information [14].

Yu *et al.* [10] have proposed an approach that upper bounds the available number of CRPs to an adversary. Only the trusted entity or the server can authorize the access of new CRPs. However, this approach supports only a limited number of authentication cycles (roughly 10,000) which makes it infeasible for applications where devices require long operating lifetimes. Gao *et al.* [11] have presented a finite state machine (FSM) locking mechanism at the output of the PUF circuit. A challenge is applied to the device and after evaluation, the responses from the PUF are fed to an FSM which traverses a given set of states till it reaches the final state. If a wrong input/challenge is applied to the PUF by an adversary, the response generated will prevent the FSM from reaching the final state thus not producing a valid response. The protocol presented in [11] seems to be sound but under strict ideal conditions (which is not a realistic assumption) where it is assumed that the device's PUF response will have a 0% variation. This is because [11] hashes the output of the PUF response without error correction. Even a one bit error in the generated response during authentication will result in an *avalanche effect* in the hashed output and thus, the protocol will fail under noisy conditions. Also, the inclusion of hash in the device drastically increases the hardware overhead.

Rostami *et al.* [15] have introduced Slender PUF, which uses neither an error-correction logic nor any cryptographic protocol but it provides an open interface to the adversary. An adversary can acquire information about CRPs as long as the device's interface access is maintained. Moreover, both [10] and [15] use a PRNG with a fixed feedback polynomial in all the devices and the server. If even one of the devices or the server gets attacked and the PRNG design leaks out, then the security of all the devices gets compromised. Herrewege *et al.* [16] have proposed a reverse fuzzy extractor which enables lightweight mutual authentication for PUF-based RFID tags. This scheme, however, does not support key establishment and uses hashing inside the device, which can increase the hardware overhead and latency. Hussain *et al.* [17] have proposed a secure hamming distance-based mutual authentication protocol employing weak intrinsic PUFs which supports an unlimited number of authentication cycles. This protocol, however, has a high latency (∼487ms) on an embedded processor which can be unsuitable for real time authentication scenarios. The protocol also does not support key establishment. Various other works [18], [19], [20], [21], [22], [23], [24] have proposed authentication schemes based on PUFs. However, most of these schemes face severe shortcomings in terms of scalability, reliability and/or security as shown in [25]. Furthermore, these schemes are limited to only authentication and do not include key-exchange mechanism which renders them undeployable for IoT-based networks where the devices need to actually communicate with one another.

Chatterjee *et al.* [26] have proposed a protocol which uses PUF-based authentication in an Internet of things (IoT) scenario and replaces the traditional certificate-based authentication. It is the first work in literature which considers the server's database to be breachable and secures it using keyed hash function. The server in [26] only stores a single key in its NVM. However, during the authentication phase, the server sends raw challenges as well as the helper data associated to the PUF to the device. This exposure of helper data and challenges can result in side-channel attacks targeted on the device's PUF as the device's interface is open to random queries. Other than that, [26] uses Bose-Chaudhuri-Hocquenghem (BCH) encoder/decoder based error correction logic inside the device to correct the noisy response which results in a high hardware area overhead. [26] also uses multiple hashing operations inside the device's software during authentication and key exchange phase. This increases the end-to-end execution time of the protocol to a great extent. Similarly, the protocols proposed in [24], [27] and [28] have a huge area overhead which makes them unsuitable for low cost authentication purposes.

In the proposed protocol, we sequentially tackle all the problems that render the previous approaches either unusable or expensive for deployment in an IoT-based system. We, first of all, develop a lightweight, multi-purpose, invertible *MASK* function. The main motivation behind developing this function was to remove the expensive hash-based obfuscation for PUF's CRPs. We then extend the usage of this function to also serve as a message authentication code. We also address the PUF's reliability problem by incorporating a BCH-based error correction scheme. This error correction however, is performed on the server instead of the device. This again, is only possible, by masking the noisy responses at the output of the PUF circuit and sending them to the server which, after unmasking and correcting, verifies the authenticity of the device. Finally, we use elliptic curve cryptography for generation of public/private key pairs and setting up a communication platform for two (or more) IoT nodes.

3 Preliminaries

3.1 Notations

Binary vectors (V), that is, $V \in \{0,1\}^n$, are represented by lower case, bold alphabets, for example, **r**, **x** etc. $\oplus$ is used for bit-wise XOR, whereas $||$ is used to concatenate two vectors. Function names are all italic, upper case alphabets and can accept n number of arguments, that is, *MAP*(k_1,k_2,k_3,...,k_n) represents a function *MAP* accepting n arguments. Functions can also accept functions as arguments. In this case, the argument function is evaluated first and the result is used in the main function, that is, *MAP*$_2$(*FUN*$_1$(l_1),*FUN*$_2$(l_2)) evaluates *FUN*$_1$(l_1) and *FUN*$_2$(l_2) first and uses the result to evaluate *MAP*$_2$. $< . >$ represents an indexed list. Assuming **c** is an n-bit binary vector, then $< c >$ represents a list of m n-bit binary vectors generated from **c**, where, $m, n \in Z^+$. $|\mathbf{r}|$ represents the length of the binary vector **r**. *HW(**r**)* computes the hamming weight, that is, the number of ones in the binary vector **r**. *HW* for an n-bit binary string **x** can be calculated by the following equation:

$$\mathcal{HW}(\mathbf{x}) = \sum_{i=0}^{n-1} (\mathbf{x}[i] \oplus 0) \tag{1}$$

3.2 Definitions

3.2.1 Public Key Operations

The proposed scheme uses Elliptic Curve Cryptography (ECC) for public key operations. ECC uses considerably smaller key lengths compared to the more common RSA, while providing the same level of security. For example, a 160-bit ECC provides the same level of security as a 1024-bit RSA [29]. This smaller key length in ECC makes it an attractive alternative over the RSA.

An elliptic curve $E_{p(a,b)}$ is defined over a finite field F and consists of all the points which satisfy the equation $y^2 = x^3 + ax + b$, where a and b are two constants that satisfy the condition $4a^3 + 27b^2 \neq 0$. The base point P of $E_{p(a,b)}$ has a prime order q. The security of ECC relies on two computationally hard problems.

- **Elliptic Curve Discrete Logarithm Problem (ECDLP)**: Suppose E is an elliptic curve defined over a finite field F and it contains a point $L : L \in E(F)$. Suppose a point M is a multiple of L, then by definition, $\exists\alpha|\alpha \in F$ such that $M = \alpha.L$. ECDLP is the computation of α given the points M and L. ECDLP is a computationally hard problem as no polynomial time algorithm exists that can compute α.
- **Elliptic Curve Diffie-Hellman Problem (ECDHP)**: Given an elliptic curve E over a finite field F, a generator point $G \in E(F)$, two points $P = \beta.G$ and $Q = \gamma.G$, such that $\beta, \gamma \in F^*$ are two unknowns, then ECDHP involves calculating the point $\beta.\gamma.G$, which is a computationally hard problem.

3.2.2 Masking Function

Given a binary l length input vector $\mathbf{r} = [r_1, r_2, r_3, ..., r_l]$, the *MASK* function applies a transformation $\Omega(\mathbf{r})$ using a set of l integers, to produce an l length output vector $\mathbf{r_m}$, comprising of a random combination of the elements in **r**. Similarly, the *MASK* function can also be used in a reversible transformation $\Omega^{-1}(\mathbf{r_m})$ using the same set of l integers to produce the original vector **r**. Both these transformations can be represented as:

$$MASK(\mathbf{r}, K) : \mathbf{r} \Longrightarrow \mathbf{r_m} \tag{2}$$

$$UNMASK(\mathbf{r_m}, K) : \mathbf{r_m} \Longrightarrow \mathbf{r} \tag{3}$$

where, $K = \{k_1, k_2, k_3, ..., k_l \mid k \in \mathbb{Z}^+\}$ is a set of l positive integers. We will use the above defined masking function as $MASK(\mathbf{r}, K)$, where the length of binary input vector **r** is equal to the number of elements in K. Similarly, the *UNMASK* function will be used in the same manner as *MASK* but with input $\mathbf{r_m}$ to regenerate **r**. Details of the *MASK* and *UNMASK* functions will be shown later. For now, it suffices to say that both the *MASK* and *UNMASK* functions generate random binary output vectors which have no correlation to the input vectors. Masking operation has two main purposes: 1) it effectively hides the relationship between the challenges and the responses of the device's PUF and 2) it provides a verification for the input to the device. Without validating the input stream, the PUF is not activated and thus no response is generated by the device which effectively prevents the device from any brute force attack.

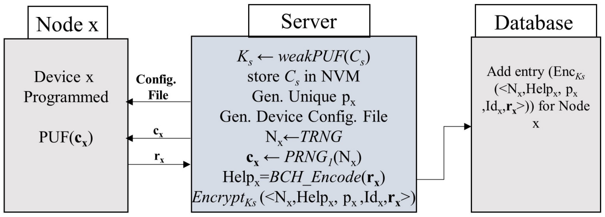
Node x
Device x
Programmed
PUF($\mathbf{c_x}$)
Config.
File
$\mathbf{c_x}$
$\mathbf{r_x}$
Server
$K_s \leftarrow weakPUF(C_s)$
store C_s in NVM
Gen. Unique p_x
Gen. Device Config. File
$N_x \leftarrow TRNG$
$\mathbf{c_x} \leftarrow PRNG_1(N_x)$
$Help_x = BCH\_Encode(\mathbf{r_x})$
$Encrypt_{Ks}$ (<N_x,$Help_x$, p_x ,Id_x,$\mathbf{r_x}$>)
Database
Add entry (Enc_{Ks} (<N_x,$Help_x$, p_x ,Id_x,$\mathbf{r_x}$>)) for Node x

Fig. 2. Enrollment Phase

3.3 Threat Model

We consider a threat model where the authentication and key exchange does not take place in a secure environment. Both the unauthenticated link adversarial model (UM) as well as the authenticated link adversarial model (AM) are considered in this scheme. The adversary, in our threat model, can eavesdrop, manipulate, or replay the traffic across all the communication links during the authentication events. These communication links include the channel between the server and the devices as well as between the server and the database storing the device's data. By using these communication links, the adversary can collect the exchanged messages and attempt to find a repetitive pattern. The adversary can also perform *man-in-the-middle* as well as spoofing attacks in order to gain unauthorized access. The device also has an open interface and the adversary can brute force query the device with any past or possibly adaptively chosen current messages/challenges.

4 Proposed Scheme

4.1 Enrollment Phase

Similar to other authentication schemes [6], [9], [10], [11], [15], [26], this scheme has an *enrollment phase* and an *authentication phase*. In addition to these two, it also provides a *key establishment phase* where the devices share a secret key which can be used for secure communication. The enrollment phase takes place in a secure environment during which the server assigns unique identifiers (IDs) to all the devices in the system. The assigned IDs are represented as id_x where $x \in \mathbb{Z}^+$. Other than this, the server generates an exclusive *primitive polynomial* p_x for the PRNG$_1$ circuit in the device x. The configuration file, with the PRNG$_1$ circuit, is then used to program the device. Afterwards, device data (e.g., CRPs) is collected. The steps involved during the enrollment phase, shown in Fig. 2, are as follows:

- The server generates an encryption key K_s by passing a random challenge C_s through a weak PUF implemented in the server. The challenge C_s is stored inside the NVM of the server.
- The server then assigns unique IDs to all the devices. It then generates a primitive polynomial (p_t) for one of the PRNGs (*PRNG$_1$*) in the design. This primitive polynomial is unique for all the devices. It should be noted that, other than the PRNG, the rest of the design is completely identical.
- The server, after modifying the PRNG design, generates a configuration file (.bin) and programs the device to set it up for the collection of CRPs.
- The server then uses its *TRNG* to generate a set of k random numbers (RN) ($N_k \leftarrow TRNG$). Each random number is m-bits long and the total numbers generated, depends on the total number of CRPs to be used for authentication purposes.

Algorithm 1 Enrollment of x devices

Input: x devices
Output: Setup of x devices

```
 1: procedure ENROLLMENT
 2:     Generate a Key $K_s$
 3:     for $t \leftarrow 0$ to $x$ do
 4:         Assign $\mathrm{ID_t}$ to the device $t$
 5:         $p_t \leftarrow$ Unique primitive polynomial
 6:         Generate $PRNG_1$ with $p_t$
 7:         Program device $t$
 8:         $N_0, N_1, \ldots, N_k \leftarrow TRNG$
 9:         for $i \leftarrow 0$ to $k$ do
10:             $\mathbf{c_i} \leftarrow PRNG_1(N_i)$
11:             Send $\mathbf{c_i}$ to the device
12:             Receive $\mathbf{r_i}$ from the device
13:             $Help_i \leftarrow BCH\_Encode(\mathbf{r_i})$
14:             $Enc_{i,x} \leftarrow Encrypt_{\mathrm{K_s}} < N_i, \mathbf{r_i}, p_t, ID_t, Help_i >$
15:             Store $Enc_{i,x}$ at $(i,x)^{th}$ location in the database
16:         end for
17:     end for
18: end procedure
```

- Afterwards, the server uses each RN as a seed to the *PRNG$_1$* to generate an m-bits long challenge $\mathbf{c_i}$ ($\mathbf{c_i} \leftarrow$ *PRNG$_1$*(N_i)). This raw challenge is sent to the device over a secure channel.
- The device instantiates its PUF, using the challenge $\mathbf{c_i}$ it receives, to generate an m-bits long response $\mathbf{r_i}$ ($\mathbf{r_i} \leftarrow$ *PUF*($\mathbf{c_i}$)). This raw response is sent to the server over a secured channel.
- The server upon receiving the response generates the helper data associated to that response using a BCH Encoder ($Help_i \leftarrow BCH\_Encoder(\mathbf{r_i})$).
- The server then encrypts ($< N_i, \mathbf{r_i}, p_t, ID_t, Help_i >$) using the encryption key K_s and stores the encrypted output into the portion of the database designated for the device with ID_t.

Steps 3 to 17 in Algorithm 1 are repeated for a total of x devices by the server. Considering that N_k, $\mathbf{r_k}$ and p_t are all m-bits long, ID_t is q-bits, and $Help_x$ is n bits, the total space complexity for all the enrolled devices would be $\mathcal{O}(((2m + n) \times k + m + q) \times x)$. Also, prior to the authentication, the server and all the devices agree upon the public domain perimeters $\{p, a, b, G, n, h\}$ of the elliptic curve. Where, p defines the finite field over which the elliptic curve is defined over, a, b are the curve parameters, G is the generator point, n and h are the order and cofactor of G, respectively.

4.2 Authentication Phase

After successful enrollment of the device in the system, the next phase is the authentication and key exchange phase. This occurs in an insecure environment, where the communication channels can be monitored by untrusted parties. We assume that two nodes with A and B want to communicate with each other. The various steps of the authentication and key exchange protocol, as shown in Fig. 3, are as follows:

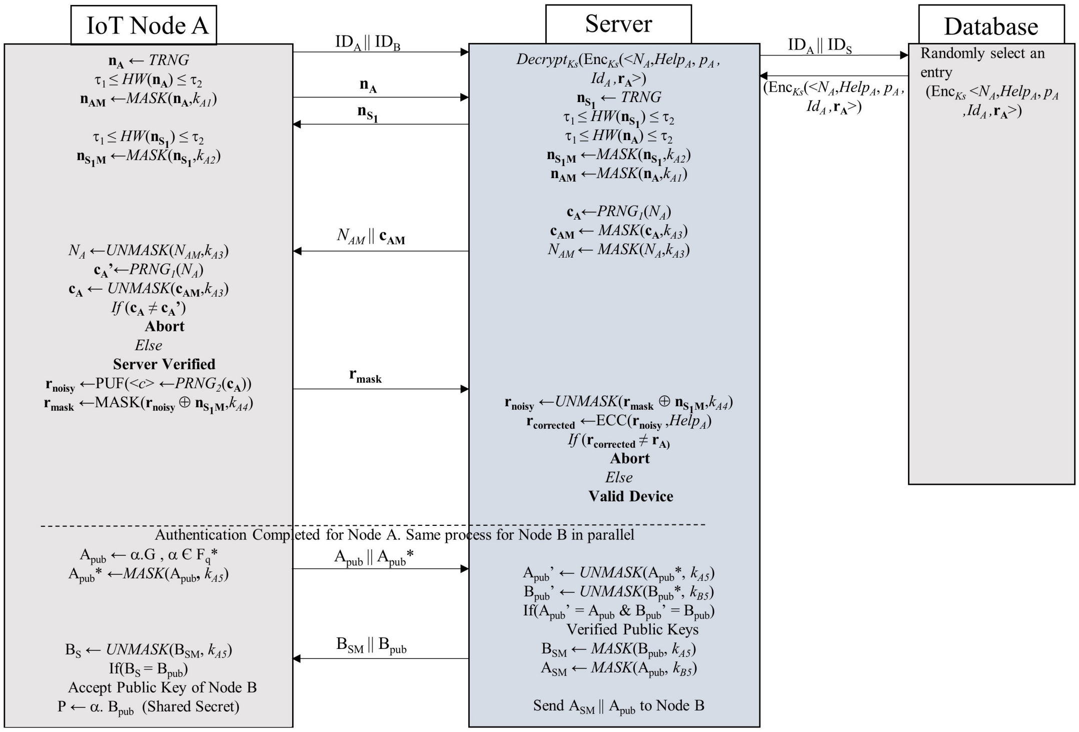
IoT Node A
Server
Database
$ID_A || ID_B$
$ID_A || ID_S$
$(Enc_{Ks}(<N_A, Help_A, p_A, Id_A, \mathbf{r_A}>)$
Randomly select an entry
$(Enc_{Ks} <N_A, Help_A, p_A, Id_A, \mathbf{r_A}>)$
$\mathbf{n_A} \leftarrow TRNG$
$\tau_1 \leq HW(\mathbf{n_A}) \leq \tau_2$
$\mathbf{n_{AM}} \leftarrow MASK(\mathbf{n_A}, k_{A1})$
$\mathbf{n_A}$
$\mathbf{n_{S_1}}$
$\tau_1 \leq HW(\mathbf{n_{S_1}}) \leq \tau_2$
$\mathbf{n_{S_1M}} \leftarrow MASK(\mathbf{n_{S_1}}, k_{A2})$
$Decrypt_{Ks}(Enc_{Ks}(<N_A, Help_A, p_A, Id_A, \mathbf{r_A}>)$
$\mathbf{n_{S_1}} \leftarrow TRNG$
$\tau_1 \leq HW(\mathbf{n_{S_1}}) \leq \tau_2$
$\tau_1 \leq HW(\mathbf{n_A}) \leq \tau_2$
$\mathbf{n_{S_1M}} \leftarrow MASK(\mathbf{n_{S_1}}, k_{A2})$
$\mathbf{n_{AM}} \leftarrow MASK(\mathbf{n_A}, k_{A1})$
$\mathbf{c_A} \leftarrow PRNG_1(N_A)$
$\mathbf{c_{AM}} \leftarrow MASK(\mathbf{c_A}, k_{A3})$
$N_{AM} \leftarrow MASK(N_A, k_{A3})$
$N_{AM} || \mathbf{c_{AM}}$
$N_A \leftarrow UNMASK(N_{AM}, k_{A3})$
$\mathbf{c_A}' \leftarrow PRNG_1(N_A)$
$\mathbf{c_A} \leftarrow UNMASK(\mathbf{c_{AM}}, k_{A3})$
$If\ (\mathbf{c_A} \neq \mathbf{c_A}')$
Abort
Else
Server Verified
$\mathbf{r_{noisy}} \leftarrow PUF(<c> \leftarrow PRNG_2(\mathbf{c_A}))$
$\mathbf{r_{mask}} \leftarrow MASK(\mathbf{r_{noisy}} \oplus \mathbf{n_{S_1M}}, k_{A4})$
$\mathbf{r_{mask}}$
$\mathbf{r_{noisy}} \leftarrow UNMASK(\mathbf{r_{mask}} \oplus \mathbf{n_{S_1M}}, k_{A4})$
$\mathbf{r_{corrected}} \leftarrow ECC(\mathbf{r_{noisy}}, Help_A)$
$If\ (\mathbf{r_{corrected}} \neq \mathbf{r_A})$
Abort
Else
Valid Device
Authentication Completed for Node A. Same process for Node B in parallel
$A_{pub} \leftarrow \alpha.G$, $\alpha \in F_q^*$
$A_{pub}^* \leftarrow MASK(A_{pub}, k_{A5})$
$A_{pub} || A_{pub}^*$
$A_{pub}' \leftarrow UNMASK(A_{pub}^*, k_{A5})$
$B_{pub}' \leftarrow UNMASK(B_{pub}^*, k_{B5})$
$If(A_{pub}' = A_{pub}\ \&\ B_{pub}' = B_{pub})$
Verified Public Keys
$B_{SM} \leftarrow MASK(B_{pub}, k_{A5})$
$A_{SM} \leftarrow MASK(A_{pub}, k_{B5})$
$B_{SM} || B_{pub}$
$B_S \leftarrow UNMASK(B_{SM}, k_{A5})$
$If(B_S = B_{pub})$
Accept Public Key of Node B
$P \leftarrow \alpha.\ B_{pub}$ (Shared Secret)
Send $A_{SM} || A_{pub}$ to Node B

Fig. 3. Authentication and Key Exchange

- The node A initiates a communication request with the server by sending its own ID (ID_A) and the ID of node B (ID_B) to the server. The server validates the IDs and sends a request to the database for a random entry pertaining to the node A.
- While the server is busy in the acquisition of data from the database, the device generates an m-bit random number n_A using its own *TRNG* circuit. It then calculates the hamming weight of n_A. For an ideal *TRNG* circuit with high bit entropy, $P(X) = 0.5$, where $X = 0$ or $X = 1$. We define τ_1 and τ_2 as the range of values that the function $HW(n_A)$ can take. The choice of τ_1 and τ_2 will be discussed in the security analysis.
- The device then performs random masking of n_A as:
$$\mathbf{n_{AM}} = MASK(\mathbf{n_A}, k_{A1}) \tag{4}$$
where, k_{A1} is a set of m integers generated by the *PRNG*$_1$ as:
$$k_{A1} = PRNG_1(\mathbf{n_A}) \tag{5}$$
The device then sends $\mathbf{n_A}$ to the server over the channel while retaining $\mathbf{n_{AM}}$.
- The server upon acquiring the random encrypted entry Enc_A from the database, decrypts it using the key K_s, acquired after passing the secured challenge C_s from the weak PUF in the server, to produce a tuple of five values as follows:
$$< N_A, \mathbf{r_A}, p_A, ID_A, Help_A > = Decrypt_{K_s}(Enc_A) \tag{6}$$
- The server then validates the entry by checking the decrypted ID_A. After verification, it configures its software-based *PRNG*$_1$ using the feedback polynomial (p_A), unique to the node A, it retrieves from the database.
- The server then repeats the same steps as the device, that is, it generates its own $\mathbf{n_{S_1}}$ from a *TRNG*, checks the HW and the proceeds to mask $\mathbf{n_{S_1}}$ to generate $\mathbf{n_{S_1M}}$. k_{A2}, in this case, is a set of m integers generated by the *PRNG*$_1$ as:
$$k_{A2} = PRNG_1(\mathbf{n_{S_1}}) \tag{7}$$
- Both the server and the device exchange their respective *TRNG* outputs, verify that the HW threshold is met and then perform the *MASK* operation. In this way, at the end of this phase, both the device and the server has a set of 4 m-bit random numbers which are $\mathbf{n_A}$, $\mathbf{n_{AM}}$, $\mathbf{n_{S_1}}$, and $\mathbf{n_{S_1M}}$. It should be noted that only $\mathbf{n_A}$ and $\mathbf{n_{S_1}}$ have been exchanged on the insecure channel and no information about $\mathbf{n_{AM}}$ and $\mathbf{n_{S_1M}}$ is visible outside the device and the server because of the random *MASK* function.
- The server now uses the first element N_A of the decrypted tuple as a seed to *PRNG*$_1$ to generate the challenge $\mathbf{c_A}$. It then masks N_A and $\mathbf{c_A}$ to generate N_{AM} and $\mathbf{c_{AM}}$ by using k_{A3} where,
$$k_{A3} = PRNG_1(\mathbf{n_{AM}}) \tag{8}$$
It then sends the masked, concatenated $N_{AM}||\mathbf{c_{AM}}$ to the device.
- The device upon receiving $N_{AM}||\mathbf{c_{AM}}$ unmasks both of them using k_{A3} to generate N_A and $\mathbf{c_A}$. Since this k_{A3} is the same as that of the server, both of the values generated in the device and the server are the same. The device then uses it's own copy of *PRNG*$_1$ to generate $\mathbf{c'}$ and verifies the identity of the server.
- If the server is verified, the device unlocks the PUF circuit and generates sub-challenges $< c >$ using $\mathbf{c_A}$. A response $\mathbf{r_{noisy}}$ is generated, XORed with $\mathbf{n_{S_1M}}$ and masked using k_{A4} where,

$$k_{A4} = \mathit{PRNG_1}(\mathbf{n_{S_1M}}) \tag{9}$$

to produce $\mathbf{r_{mask}}$. This is sent to the server for verification.

- The server upon receiving $\mathbf{r_{mask}}$, XORs and unmasks it with $\mathbf{n_{S_1M}}$ and k_{A4}, respectively, to regenerate $\mathbf{r_{noisy}}$. This $\mathbf{r_{noisy}}$ is then corrected using $Help_A$ to generate $\mathbf{r_{corrected}}$. This is compared with $\mathbf{r_A}$ to validate the device. Both the server and the device at this point have mutually authenticated each other. This is also true for Node B who's authentication is done in parallel by the server in a separate instance following the same steps.

4.3 Key-Establishment Phase

The next phase of operation is the key establishment phase which is done using ECDHP.

- The device generates a $private$ key α. This can be done using the TRNG circuit designed in hardware or through software based random number generator. It then uses elliptic curve point operations to generate a public key A_{pub} and then masks the public key by k_{A5} to generate $A_{pub}*$ where,

$$k_{A5} = \mathit{PRNG_1}(\mathbf{r_{noisy}}) \tag{10}$$

It should be noted that the $*$ in $A_{pub}*$ is just a notation and does not represent the kleene star operation. Thus, by eq. (10), the public key of node A is tied to the noisy response generated by its PUF. It should be noted that $\mathbf{r_{noisy}}$ is never exposed outside the device and only a valid server can retrieve it from $\mathbf{r_{mask}}$. The server then verifies A_{pub} by unmasking $A_{pub}*$. The same process is repeated for B_{pub} for node B. It should be noted that only the server can retrieve the unique r_{noisy} from both nodes A and B and these two nodes do not know each other's r_{noisy} because of the PUF's uniqueness property.

- In the final step, the server masks the public key of node B using the r_{noisy} of node A and vice versa. It then sends the masked public key (B_{SM}) and the original public key (B_{pub}) of Node B to Node A. The Node A then unmasks the masked public key and compares it against the original public key. If the comparison is successful, Node A accepts the public key of Node B and generates the shared secret ($P = \alpha.B_{pub}$). Similarly, Node B validates the public key of Node A using its own r_{noisy} and generates the shared secret ($P = \beta.A_{pub}$).

This concludes the key establishment phase between the two nodes A and B. The server and the nodes may discard all the private variables associated to this authentication and key establishment cycle. The nodes can use any encryption scheme to encrypt the messages using the shared secret P. The exact details of the encryption are not covered in this work, but for the sake of completeness, we can present one most commonly used method. The two nodes can use the x-coordinate of the shared secret P as a key and discard the y-coordinate. To prevent usage of a $biased$ key, the nodes may $hash$ the x-coordinate and use the hashed output as a key to an AES-based symmetric encryption scheme.

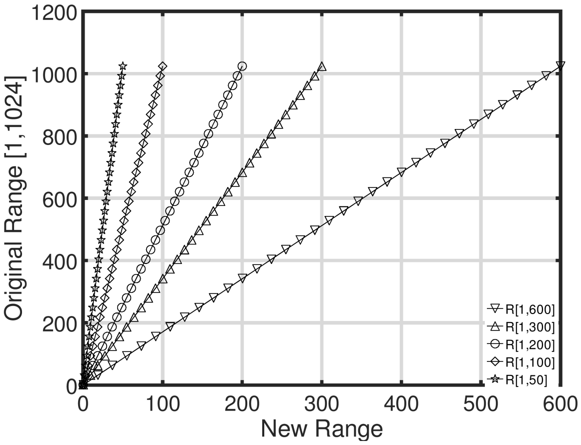

Fig. 4. Range Transformations

4.4 Input Stream Verification via Masking

We defined the usage of $MASK$ function in the earlier sections. In this section, we will elaborate $MASK$ more thoroughly and present the operations which are undertaken when a call to this function is made. Since $MASK$ is frequently used and plays a pivotal role in the proposed scheme, we will also do a formal security verification of masking.

The $MASK$ function takes two input arguments; an m-bit binary vector $\mathbf{x}$ to be masked and an integer set K. As shown before, K is generated using $\mathit{PRNG_1}$ as ($K \leftarrow \mathit{PRNG_1}(\mathbf{y})$), where $\mathbf{y}$ is an n-bit binary vector. Thus, by definition, we can also write masking function as $MASK(\mathbf{x},\mathbf{y})$ which takes two binary vectors; an m-bit $\mathbf{x}$ and an n-bit $\mathbf{y}$, as inputs and outputs an m-bit binary vector $\mathbf{x_m}$ which is a random combination of the elements(bits) in $\mathbf{x}$. We will use combination and permutation interchangeably in this text. The three main operations carried out during masking are as follows:

4.4.1 Integer Set Generation

: The binary vector $\mathbf{y}$ is used as a seed to a PRNG circuit ($\mathit{PRNG_1}$) to generate a set of positive integers $\{k \mid k \in \mathbb{Z}^+\}$. The number of elements in the set are equal to m. All the integers in the integer set K are treated as n-bit positive numbers, where the maximum value that any integer can have is $2^n - 1$.

4.4.2 Range Transformation

We define a function $RANGE$ as a linear mapping transformation which, given an m-bit integer, $k \in K$, with the value in the range $[0, 2^m - 1]$, generates a new set Q $\{q \mid q \in \mathbb{Z}^+\}$ of l-bit integers in the range $[0, 2^l - 1]$, where $l \leq m$. The linear range mapping is governed by the equation:

$$N_{new} = \left\lfloor \frac{(N_{old} - N_{oldMin}) \times (N_{newMax} - N_{newMin})}{(N_{oldMax} - N_{oldMin})} \right\rfloor \tag{11}$$

where, $N_{old} \in K$ is an input to the $RANGE$ function. N_{oldMin} is the minimum and N_{oldMax} is the maximum value in the old range $[0, 2^m - 1]$, which in this case are 0 and $2^m - 1$, respectively. N_{newMax} and N_{newMin} are the maximum and minimum values in the new range $[0, 2^l - 1]$. In our case, the N_{newMin} remains a constant 0, whereas N_{newMax} can vary and is provided as the second input to the $RANGE$ function. Thus, by incorporating these changes, the updated range equation becomes:

$$N_{new} = \frac{(N_{old} \times N_{newMax})}{N_{oldMax}} \tag{12}$$

Because N_{oldMax} is in powers of 2, the equation can be further reduced by performing a simple right shift operation instead of division to conserve the hardware resources. The final equation becomes:

$$N_{new} = (N_{old} \times N_{newMax}) >> m \tag{13}$$

Fig. 4 shows some of the linear range transformations governed by (13).

4.4.3 Bit Shuffling

The final step of the *MASK* function is *bit shuffling*. The shuffling algorithm used in the proposed scheme is based on Durstenfeld version of Fisher-Yates Shuffler [30]. For a list of n distinct elements, the Durstenfeld shuffler produces $n!$ permutations, all of whom are equally likely. We will formally prove the security of the shuffler in the next section.

We are now in the position to summarize the *MASK* function in terms of an algorithm. Algorithm 2 shows the entire masking process involving integer set generation, range adjustment and bit shuffling. It can be seen from the Algorithm 2 that the Durstenfeld shuffler shuffles the data *in-place*, this means that the m-bit binary vector **x** can also be used as output at the end of the algorithm execution. For the sake of simplicity, we use the m-bit binary vector $\mathbf{x_m}$ as output. It is also evident from the algorithm that the output $\mathbf{x_m}$ can be unmasked and transformed back to **x** by performing *UNMASK*($\mathbf{x_m}$,**y**). That is,

$$\mathbf{x} = UNMASK(MASK(\mathbf{x}, \mathbf{y}), \mathbf{y}) \tag{14}$$

For the case of *UNMASK*, the same integer set $\{k_1, k_2, ..., k_m\}$ is used, however, the loop is iterated in reverse, that is, from m to 1.

Algorithm 2 Masking Process

Input: (**x**,**y**): m-bit binary vector **x**, n-bit binary vector **y**
Output: $\mathbf{x_m}$: m-bit binary vector

```
1: procedure MASK(x,y)
2:     Integer Set K: $\{k_1, k_2, ..., k_m\} \leftarrow PRNG_1(\mathbf{y})$
3:     for $i \leftarrow 1$ to $m$ do
4:         $N_{new} \leftarrow RANGE(k_i, m+1-i)$
5:         $\mathbf{x} \leftarrow SWAP(\mathbf{x}_{N_{new}}, \mathbf{x}_{\text{m-i+1}})$
6:         $\mathbf{x_m} \leftarrow \mathbf{x}$
7:     end for
8: end procedure
```

Illustrative Example of Shuffling and Deshuffling

Fig. 5 shows the process of masking and unmasking on a 6-bit binary test vector, $\mathbf{x} = [1, 0, 0, 1, 0, 1]$, by applying Algorithm 2. Thus, in this case, $m = 6$. The binary vector **y** is the same as **x**, that is, we are applying masking as *MASK*(**x**,**x**). Note that this is just a test case and the actual input should be much wider ($\geq$ 64 bits) for providing any reasonable security. We assume that by inputting **x** to a *PRNG₁* circuit, we get the integer set $K = \{43, 32, 60, 54, 12, 28\}$. The algorithm masks the input **x** as follows:

- During the first iteration, that is, $i = 1$, $N_{old} = 43$, $N_{newMax} = 6$, the value of N_{new}, as calculated by (13), comes out to be 4. We update **x** by *SWAP*-ing the 4^{th} and 6^{th} element. The new value of **x** is copied to $\mathbf{x_m}$. The arrows in Fig. 5 indicate the elements which are swapped.

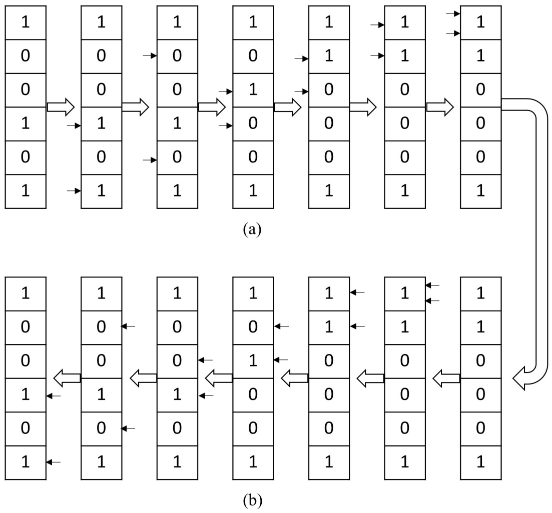

Fig. 5. Masking and Unmasking. (a) *MASK* operation (b) *UNMASK* Operation

- During the second iteration, that is, $i = 2$, the value of N_{new} comes out to be 2. We swap the 2^{nd} and 5^{th} element and copy updated **x** to $\mathbf{x_m}$.
- We use the same procedure for all future iterations, that is, from $i = 3$ to $i = m = 6$. The final output $\mathbf{x_m}$, after the completion of mask operation, comes to be $\mathbf{x_m} = [1, 1, 0, 0, 0, 1]$. This is shown in Fig. 5. It should also be noted that since the bit indexing is done from 1 to m, if at any stage, N_{new} comes out to be 0, we update it to 1 for the proper indexing of the swap operation.

We follow the same process as above for unmasking the masked output, $\mathbf{x_m}$, as *UNMASK*($\mathbf{x_m}$,**x**). The same integer set $K = \{43, 32, 60, 54, 12, 28\}$ is used during unmasking operation.

- During the first iteration, that is, $i = m = 6$, $N_{old} = 28$, $N_{newMax} = 1$, the value of N_{new} comes out to be 0, which is adjusted to 1. We update $\mathbf{x_m}$ by *SWAP*-ing the 1^{st} bit by itself which produces no change.
- During the second iteration, that is, $i = 5$, the value of N_{new} comes out to be 1. We swap the 1^{st} and 2^{nd} bit and copy updated $\mathbf{x_m}$ to **x**.
- We repeat this process for the rest of the iterations, that is, from $i = 4$ to $i = 1$ and subsequently update $\mathbf{x_m}$ and **x**. For the first iteration, that is, $i = 1$, the value of N_{new} comes out to be 4. We update $\mathbf{x_m}$ by swapping the 4^{th} and 6^{th} element. It can be seen that the final output of unmasking operation is equal to the input of the masking operation.

4.4.4 Security of the Shuffler

In this section, we will formally prove the security of the bit shuffler. The shuffling process is described in Algorithm 2 from lines 2 to 7. Our main assumption while proving the security of the shuffler is that the shuffler receives highly random values from the random number generator (line 2 of Algorithm 2) and uses those values for shuffling the m-bit

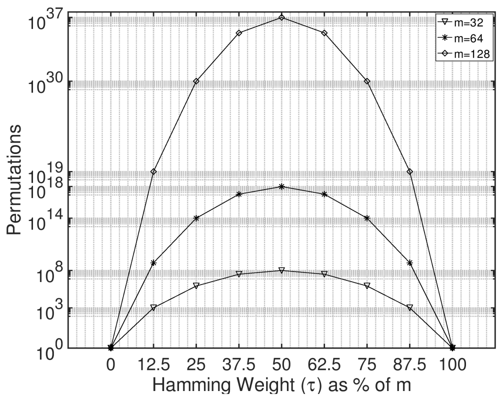

Fig. 6. Permutations vs. HW as a function of input size

input. Under this assumption, we will show that it is highly improbable to determine the m-bit output vector produced by the shuffler given the input. It can also be seen from Algorithm 2 that the security of the masking operation is contingent upon the security of the shuffler. This is because the shuffler adds randomness into the input binary vector by generating an output vector which is a random permutation of the input vector, where, every permutation is equally likely.

Since the m-bit input vector $\mathbf{x}$ is binary, that is, $\mathbf{x} \in \{0,1\}^m$, the total $unique$ permutations of $\mathbf{x}$ are given by:

$$\mathit{Permutations} = \frac{{}^{m}P_{m}}{\tau_1! \times (m - \tau_1)!} \tag{15}$$

where,

$${}^{m}P_{m} = \frac{m!}{(m\text{-}m)!} = m! \tag{16}$$

In equation (15), τ_1 is the hamming weight, that is, the number of 1s and $(m - \tau_1)$ represents the number of 0s in $\mathbf{x}$. Fig. 5 shows the total number of permutations against τ_1, for different values of m. It can be observed that for different values of m, the maximum number of unique permutations occur when $\tau_1 = \frac{m}{2}$. Similarly, the minimum occur when $\tau_1 = m$ or $\tau_1 = 1$. This provides the reasoning as to why, in Fig. 3, after the generation of $\mathbf{n_A}$ and $\mathbf{n_{S_1}}$, both the server and the device check the HW of their respective nonces. If the nonce has a HW which does not satisfy the lower bound, τ_1, that nonce is discarded and a new one is generated. It should be noted that τ_2 is just the number of zeros and is equal to $(1 - \tau_1)$. The number of permutations given by equation (15) will still remain the same regardless of whether τ_1 or τ_2 is used but for convenience we use HW, that is, τ_1. Also, after the server and the device transmit their respective nonces, HW is again checked. This is *critical* because, as shown in Fig. 6, as the HW is decreased to 12.5% of $m = 128$, the total number of random permutations the shuffler could produce, are only 10^{19}. Decreasing the HW further would even lower the total number of permutations produce-able by the shuffler. An adversary can modify the nonce during transit and decrease the HW substantially (e.g., 5 1′s and 123 0′s for m = 128), the consequence of which is the decrease in the number of unique permutations (only $\approx 10^8$) that the shuffler can produce. The adversary can then just permute through all the possible number of permutations to find the correct *shuffled* versions of $\mathbf{n_A}$ or $\mathbf{n_{S_1}}$, produced by the shuffler. Thus, as long as the HW lies within the specified boundary of τ_1 and τ_2 (as defined by the protocol), the nonce will be accepted regardless of whether it was modified by the adversary during transit. This is because, as shown in Figure 6, the total number of unique permutations producible by the shuffler, is a function of input size m and hamming weight τ and not of the specific shape of the input.

All of the above discussion assumes that given an m-bit binary vector $\mathbf{x}$, the shuffler produces an m-bit output $\mathbf{y}$ which can be any of the $m!\,/(\tau_1! \times (m - \tau_1)!\,)$ permutations, where every permutation has equal probability of occurrence. For the case, when the vector $\mathbf{x}$ has distinct elements, the total number of permutations, by definition, will be $m!$. For simplicity, we will prove the case that when $\mathbf{x}$ has unique elements, the shuffler can produces any of the $m!$ permutations, with each permutation having equal probability of occurrence, that is, $\frac{1}{m!}$. This can then be extended for the case when $\mathbf{x}$ has binary elements with repetitions.

Theorem 1. The m-length output vector $\mathbf{Y}$ $\{y[1], y[2], ..., y[m]\}$, produced at the end of the shuffling algorithm, by an m-length input vector $\mathbf{X}$ with *distinct* elements, can be any of the $m!$ permutations with the same probability.

Proof 1. We will proof the above theorem by induction on m.

Base Case: For the case of $m = 1$, the proof is trivial.

Induction Hypothesis: We assume that the theorem holds for $m = j$, that is, $\{y[1], y[2], ..., y[j]\}$ produced at the output can be any of the $j!$ permutations with the same probability.

Induction Step: We will show now that the theorem holds for $m = j + 1$. We assume that Y is the intermediate list $(\{y[1], y[2], ..., y[j]\})$ produced before processing $y[j + 1]$ and Y' is the list $(\{y[1], y[2], ..., y[j], y[j + 1]\})$ produced after processing $y[j + 1]$. Let q be the random number used to process Y', then it is obvious that Y' is a function of (Y, q). Here, q has $(j+1)$ possible choices. Since Y had $j!$ possible choices before processing $y[j+1]$, so Y', using q, will have $(j+1)!$ possible choices. In other words, there are $(j+1)!$ distinct pairs of (Y, q). Now we will prove two lemmas that will inadvertently prove the correctness of the theorem.

Lemma 1. The probability of occurrence of every pair of (Y, q) in the algorithm is the same.

Proof. By our induction hypothesis for $m = j$, we can see that there will be a total of $j!$ unique permutations with each permutation having the same probability of occurrence, that is, $1/j!$. For the case of $m = j + 1$, the value of q can be anywhere between 1 and $j+1$, inclusive and hence, for q generated from a reasonably random source, every value of q in this interval will have the probability of occurrence $\approx 1/(j + 1)$. Thus every pair (Y, q) occurs with the probability $1/(j + 1)!$.

Lemma 2. Every pair (Y, q) produces a distinct Y'.

Proof. Let (Y_1, q_1) and (Y_2, q_2) be any two distinct pairs. Then, at least one of the following conditions hold, that is, $Y_1 \neq Y_2$, $q_1 \neq q_2$. We assume that Y_1' and Y_2' are produced from the first and second pair, respectively. We will show that $Y_1' \neq Y_2'$. There are two cases:

- Case 1: For the case when $Y_1 \neq Y_2$. Let i be the smallest number such that the i^{th} element of Y_1 differs from Y_2. If $q_1 \neq q_2$ or $q_1 = q_2 \neq i$, then Y_1' differs from Y_2' at the i^{th} element. If $q_1 = q_2 = i$, then Y_1' is still different from Y_2' on the $(i+1)^{st}$ element.
- Case 2: For the case when $Y_1 = Y_2$ and $q_1 \neq q_2$, Y_1' is still different than Y_2' on the $(j+1)^{st}$ element because of the the algorithm works. Hence, $Y_1' \neq Y_2'$.

This proves the correctness of Theorem 1. We can extend this proof for any m-bit binary vector $\mathbf{x}$ and show that the shuffler produces $m!\,/(\tau_1!\times(m-\tau_1)!\,)$ unique permutations, where each permutation is equally likely and has the probability of occurrence equal to $(\tau_1!\times(m-\tau_1)!\,)/m!$.

Input Re-Appearance: Another important consideration while evaluating the security of the shuffler is finding out the probability that the output produced by the shuffler is equal to the input. It is evident that this probability needs to be extremely low. Intuitively, from Theorem 1, we can see that this probability $(\mathcal{P})$, is the same as the probability of occurrence of any other random permutation, that is:

$$\mathcal{P} = \frac{\tau_1!\times(m-\tau_1)!}{m!} \tag{17}$$

However, for quantitative analysis and better understanding, we implement an algorithm which can calculate the exact probability of input re-appearance.

Algorithm 3 Probability of Input Re-appearance

Input: $\mathbf{x}$: m-bit binary input vector $\mathbf{x}$ = $\{x[1], x[2], ..., x[m]\}$
Output: *Pr(Shuffle(**x**)=**x**)*: Probability that the shuffler outputs the same vector as the input

```
procedure Probable(x)
    for $i \leftarrow m$ to 1 do
        if $(x[i] == 1)$ then
            for $j \leftarrow i$ to 1 do
                if $(x[j] == 1)$ then
                    $count1$++
                end if
            end for
            $Pr = Pr \times \frac{count1}{i}$
        else
            for $j \leftarrow i$ to 1 do
                if $(x[j] == 0)$ then
                    $count0$++
                end if
            end for
            $Pr = Pr \times \frac{count0}{i}$
        end if
    end for
end procedure
```

Algorithm 3 follows the shuffling process and iterates through the entire string. It calculates the probability that the value (0 or 1), at a particular index, will be replaced by the same value by counting the number of similar values in the remaining string. Fig. 7 shows the result of running Algorithm 3 on random m-bit binary strings with different HWs. It follows that as the size m of the input string is increased and $HW \approx \frac{m}{2}$, the probability of input re-appearance decreases drastically.

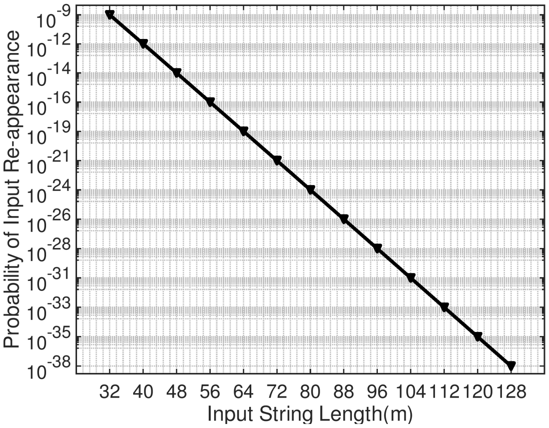

Fig. 7. $\mathcal{P}$ variation with input size

Input Independence: Another attractive property of this shuffling scheme is its input independence. This is a direct consequence of Theorem 1. The shuffling scheme treats any m-bit input as a random permutation of an m-bit string and generates another random permutation. Thus, even if the adversary modifies the input string, while preserving the HW to prevent discardment of the string by the protocol, the total unique permutations that the shuffler can produce will still be given by (15) and all the permutations will be equally likely as indicated by Theorem 1.

We will now present a theorem that quantifies the security level provided by the shuffler and thus compares the security of the shuffler against the security of commonly used block ciphers.

Theorem 2. A shuffler of length m bits provides a security level of $m - \log_2 r_s$ bits, where r_s is the search space ratio of the search space of a symmetric cipher with a key length of m bits to the search space furnished by the shuffler.

Proof 2. The search space imparted by a symmetric cipher of key length m bits is equal to 2^m whereas the search space (i.e., the number of possible permutations) provided by the shuffler of length m and hamming weight τ is equal to $m!\,/(\tau!\times(m-\tau)!\,)$ (follows from Eq. (15)). The search space ratio r_s, that is, the ratio of the search space of symmetric cipher with key length of m bits to the search space furnished by the shuffler can be given as

$$r_s = \frac{2^m \times \tau!\times(m-\tau)!}{m!} \tag{18}$$

The r_s quantifies how much more search space explorations are required to break a symmetric cipher of key length m as compared to a shuffler of length m. The number of bits that accounts for these additional search space explorations imparted by a block cipher of key length m can be given as $\log_2 r_s$. Hence, a symmetric block cipher with a key length of $m - \log_2 r_s$ bits provides an equal search space as that of the shuffler with length m. Alternatively, the equivalent key length (EKL), that is, the key length of the symmetric cipher which will impart an equivalent level of security as the shuffler is equal to $m - \log_2 r_s$ bits. Conversely, the security level offered by a shuffler of length m is equal to $m - \log_2 r_s$ bits. ■

Lemma 3. The maximum security level provided by a shuffler of length m bits and hamming weight τ is for $\tau = m/2$.

Proof. The search space ratio r_s in Eq. (18) is minimized when $\tau = m/2$. The maximum values of r_s is obtained for the extreme cases, viz., $\tau = 0$ and $\tau = m$ both of which gives the value of $r_s = 2^m$. The value of r_s decreases as τ approaches $m/2$ and reaches the minimum value at $\tau = m/2$, that is, $\lim_{\tau \to m/2} r_s(\tau) = r_{s_{min}} = (2^{m+1} \times (m/2)!\,)/m!$, where $r_{s_{min}}$ denotes the minimum value of r_s. The maximum security level is offered by a shuffler of length m when r_s is minimized, that is, the maximum security level corresponds to $m - \log_2 r_{s_{min}}$. Since $r_{s_{min}}$ is attained when $\tau = m/w$, this implies that the maximum security level that can be provided by a shuffler of length m bits and hamming weight τ is when $\tau = m/2$.

<u>*Example:*</u> Consider a shuffler with m = 128 and τ = 50% of m. The total unique permutations that can be generated by the shuffler in this case is equal to $2 \times 10^{37} \approx 2^{124}$ (follows form Eq. (15) and Fig. 6). Also, for a 128-bit block cipher, the search space is 2^{128}. The r_s in this case comes out to be $\approx 2^4$, which gives an EKL of $128 - \log_2 2^4 = 128 - 4 = 124$. Thus, the shuffler will provide a security equivalent to a block cipher with key size $\approx$ 124 bits.

4.4.5 PRNG Design

We use linear feedback shift register (LFSR) based PRNGs because of their low hardware overhead and nearly uniform statistical outputs. Some important properties of PRNGs to look for during the design phase are linearity, circularity and predictability. We use two LFSR-based PRNGs in this protocol. *PRNG₁* is used in both the *MASK* operation as well as the $\mathbf{c_A}$ generation. *PRNG₂* is used for $< c >$ generation. Both of the LFSRs are maximum-length LFSRs, having state $\mathbf{s}$. The feedback polynomial, extended over the finite field $GF(2)$, is *primitive*. The LFSR is initialized with a given seed $\mathbf{s_0}$ and it cycles through $2^{|\mathbf{s_0}|} - 1$ states. *Lock-up* state, that is, *all-zeros* state is avoided by design for an *XOR*-based LFSR and *all-ones* state is avoided for an *XNOR*-based LFSR. We will specifically talk about *PRNG₁* as it is unique for every node and as is more critical because of it's usage in the *MASK* function.

For an m-bit minimum length LFSR and a *prime* power n, the total number of primitive polynomials, $t_q(m)$, over *GF(q)* are given by:

$$t_q(m) = \frac{\phi(q^m - 1)}{m} \tag{19}$$

where,

$$\phi(m) = \prod_{i=1}^{n} (P_i^{e_i} - P_i^{e_i - 1}) \tag{20}$$

is the Euler's totient function. We assume that m has a canonical factorization such that $m = p_1^{e_1}.p_2^{e_2}...p_n^{e_n}$ for a prime p and $n \in Z^+$. Fig. 8 shows the growth in the total number of primitive polynomials as the length of *PRNG₁* increases from $m = 1$ to $m = 64$. By referring to Fig. 8, we can conclude that by choosing a *PRNG₁* of length $m = 64$, there are $\approx 10^{18}$ unique polynomials, the direct consequence of which is that more than a *quadrillion* devices can be enrolled in the system with each having a unique polynomial hardwired inside of it. The server also does not need to store any polynomial information as it extracts the said information from the database and configures its *software*-based *PRNG₁* during an authentication run.

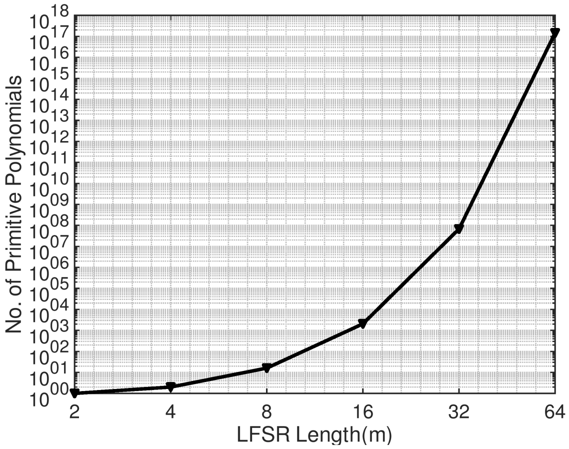

Fig. 8. Number of unique primitive polynomials against LFSR length (m)

The seed value or $\mathbf{iv}$ for *PRNG₁* during the process of masking is defined as $\mathbf{s_0} = \mathbf{iv}||\mathbf{n'_x}$. Thus, equations (5), (7), (8), (9) and (10) are expanded as *PRNG₁*($\mathbf{n_x}$) = *LFSR*($\mathbf{iv}||\mathbf{n'_x}$). Here, $|\mathbf{s_0}|= m$, where m is the length of the binary vector $\mathbf{n_x}$, inputted to the *MASK* function. Thus, both the $\mathbf{iv}$ and $\mathbf{n'_x}$ are $m/2$ in length. Choice of $\mathbf{n'_x}$ is made during the design phase. Any $m/2$ bits can be chosen from $\mathbf{n_x}$ during an authentication cycle and used as a part of the seed to the LFSR. It should be noted, however, that both the server and the device should agree upon a common choice of $\mathbf{n'_x}$, otherwise the masked outputs will be different.

We will now establish some design rules for *PRNG₁*, specifically for the generation of integer set K, used during the *MASK* operation. From Algorithm 2, we can see that the output of *PRNG₁* is used by the *RANGE* function which generates integers with a decreasing range. If m integers are generated by the PRNG, that is, $\{k_1, k_2, ..., k_m\}$, then, during the first iteration, the range is between $(1, m)$, inclusive. During the second iteration, the range is from $(1, m - 1)$, inclusive and so on. This implies that during the first iteration, every number in the range $(1, m)$ is equally probable and has the probability $1/m$. Similarly, for the second iteration, with range $(1, m-1)$, every number has the probability $1/(m - 1)$ and so on. Keeping this observation in mind, we can develop an $m \times m$ *Probability Matrix* ($\mathbf{P_M}$) which is a mapping of the range adjusted PRNG outputs to their exact probability of occurrences for all successive iterations of the *MASK* algorithm.

$$\begin{bmatrix} 1 & 2 & 3 & \dots & m \\ 1 & 2 & 3 & \dots & m \\ 1 & 2 & 3 & \dots & m \\ \vdots & \ddots & & & \\ 1 & 2 & 3 & \dots & m \end{bmatrix} \Longrightarrow \mathbf{P_M} \tag{21}$$

where,

$$\mathbf{P_M} = \begin{bmatrix} 1/m & 1/m & \dots & 1/m & 1/m \\ 1/(m-1) & 1/(m-1) & \dots & 1/(m-1) & 0 \\ 1/(m-2) & 1/(m-2) & \dots & 0 & 0 \\ \vdots & \ddots & & & \vdots \\ 1/2 & 1/2 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \end{bmatrix} \tag{22}$$

Here, (21) represents the mapping of the range-adjusted PRNG outputs $\{k_1, k_2, ..., k_m\}$. k_1 is any number in the range $(1, m)$ represented by the first row in N_R. All the numbers in the first row are equally probable by nature of the algorithm and thus, mapped to the probability $1/m$ in $\mathbf{P_M}$. Thus, first row of N_R and P_M represents the first iteration of Algorithm (2). During the second iteration, the range is $(1, m-1)$, and thus, all the elements, other than the m^{th} are equally probable with the probability $1/(m-1)$. In the last iteration, since the range is $(1, 1)$, thus, 1 has the probability of 1 and all other numbers in $\mathrm{N_R}$ have a 0 probability of occurrence.

Assuming that the Algorithm 2 runs k times, each time with m iterations, the *probability matrix* ($\mathbf{P_M}$) can be used to calculate the *expected outcome matrix* ($\mathbf{E_M}$) as:

$$\mathbf{E_M} = k \times \mathbf{P_M} \tag{23}$$

Here, $\mathbf{E_M}$ maps the numbers in N_R to their expected occurrence. For $m = 10$ and $k = 100$, every number from $(1, m)$ should ideally occur $k/m = 10$ times during the first run of the algorithm. Similarly, every number from $(1, m-1)$ should occur $k/(m-1) \approx 11$ times in the second iteration and so on. A PRNG design which produces an *Actual Outcome Matrix* ($\mathbf{A_M}$), *close* to $\mathbf{E_M}$, after running k iterations, will provide highly unbiased inputs to the shuffler in the *MASK* function. We say *close* because, if the PRNG produces $\mathbf{A_M}$ exactly like $\mathbf{E_M}$, that is, there is no deviation between the expected and the observed values then one might question the randomness of the PRNG.

As a design example, we generate a 64-bit PRNG with $m = 64$. We run the PRNG several times to come up with a choice of **iv** that gives the most random PRNG outputs. We then run the Algorithm 2 a total of $k = 10,000$ times and generate both the $\mathbf{E_M}$ as well as the $\mathbf{A_M}$ matrix. To test the goodness of fit between $\mathbf{A_M}$ and $\mathbf{E_M}$, we use two-sided *Chi-Square Goodness of Fit* test with 63 degrees of freedom and 95% confidence interval with two critical values, $\alpha = 0.025$ and $\alpha = 0.975$, for the left and the right side, respectively. Our test statistic χ^2 comes out to be ≈ 65. Comparing against a standard *Chi-Square Table*, our test software concluded that the null hypothesis, that is, H_0: The PRNG design is well suited for the shuffler, to be *TRUE*.

4.4.6 Correctness Proof of the Protocol

Now that we have presented all the necessary details of the various operations during the protocol execution, we can prove the correctness of the scheme. We assume a scenario where two communicating parties, that is, Node A and Server S are authenticating each other. Both the parties generate their respective outputs after every authentication stage. We assume that both S and A are running protocol γ shown in Fig. 3. We define the correctness of protocol γ as follows:

Definition 3. (Correctness of Protocol) The protocol γ will be *correct* if the output generated by node A, $Output_{A,\gamma}(n_{s_1},N_{AM}||c_{AM})$, and the one generated by the server S, $Output_{S,\gamma}(n_A,Enc_{Ks}(N_A,Help_A,r_A))$, can only differ ϵ times after d authentication cycles.

It should be noted that the quantity ϵ should be extremely small whereas d should be sufficiently large. The above definition can be written as:

$$Pr[Output_{A,\gamma}(n_{s_1},N_{AM}||c_{AM}) \neq \\ Output_{S,\gamma}(n_A,Enc_{Ks}(N_A,Help_A,r_A))] \leq \epsilon$$

Representing *MASK* and *UNMASK* functions as $\mathcal{M}$ and $\mathcal{U}$, respectively, it can be observed that:

$$Output_{A,\gamma} = (BCHDecoder(\mathcal{U}(\boldsymbol{r_{mask}} \oplus \boldsymbol{n_{s_1 m}}, k_{A4}), Help_A)) = \\ Decrypt_{K_S}(N_A,Help_A,\mathbf{r_A}) = Output_{S,\gamma}$$

where,

$$\mathbf{r_{mask}} = \mathcal{M}(PUF(PRNG(\mathcal{U}(\mathbf{c_{AM}}, k_{A3}))))$$

The above equations show that the outputs of the node A and the server S will be equal when the authentication criteria is met, that is, $\mathbf{r_{corrected}} = \mathbf{r_A}$. This is only possible when both the node A and the server S are *honest* and the communication channel is *unhindered* and *lossless*.

5 Resilience Against Attack Scenarios in the Threat Model

In this section, we analyze different attack scenarios by which the adversary can attempt to break the protocol. Similar to [26], the proposed scheme considers two major adversarial models.

- **The Unauthenticated Link Adversarial Model (UM):** Here a probabilistic polynomial time (PPT) adversary $\mathcal{A}dv$ can actively attack the communication channels between two parties. The adversary $\mathcal{A}dv$ may read, modify or delete the messages exchanged. In addition, $\mathcal{A}dv$ may also obtain some sort of secret information hidden in the communicating parties' internal memories.
- **The Authenticated link Adversarial Model (AM):** In the AM model, the adversary $\mathcal{A}dv$ has limited control over the communicating parties and the information exchanged between them. $\mathcal{A}dv$ may choose to not deliver the messages generated by the parties but if delivered, $\mathcal{A}dv$ can not alter the messages.

It is apparent that the UM model is more realistic in nature given the bulk of computing power available to the $\mathcal{A}dv$. Thus, if a protocol is protected against a UM model, it is, by definition, protected against the AM model. We show that the proposed protocol γ is secure against UM adversary $\mathcal{A}dv$ under the following assumptions:

- **PUF Uniqueness Problem (PUP):** Our first security assumption relies on the premise that a PPT $\mathcal{A}dv$ can not replicate the behaviour of the device's PUF even if $\mathcal{A}dv$ copies the entire design of the PUF. This assumption is based on the mathematical and physical unclonability of PUF structures. Two *structurally* similar PUF instances, PUF_A and PUF_B, on different silicon chips, provided the same n-bit challenge C: $\{0,1\}^n$ will produce two m-bit response strings, R_1: $\{0,1\}^m$ and R_2: $\{0,1\}^m$, where the probability of R_1 and R_2 being the same is negligible. That is:

$$\mathcal{P}[PUF_A(C:\{0,1\}^n) = PUF_B(C:\{0,1\}^n)] = \varepsilon \tag{24}$$

It has been shown in literature [31] that the basic Arbiter PUF and many of its variants, e.g., the XOR PUF have very low uniqueness properties. These PUFs make the protocols, employing them, vulnerable to *impersonation* attacks, where the adversary copies the publicly available PUF structure and impersonates as a legitimate entity. To circumvent these issues, an APUF variant called *m-nDAPUF* has been proposed in [32]. This PUF has high uniqueness properties and is a good candidate for authentication protocols. [26] also uses a variant of the DAPUF. Even though the DAPUF provides good resilience against impersonation attacks, which rely on low uniqueness, it is still vulnerable to software based modeling where the PUF's CRPs, over the channel, are recorded and used to generate a software model which can predict the response to an unseen challenge. Infact, [33] recently proposed a deep neural network-based attack which can accurately predict the response of a DAPUF to unseen challenges with probability of 88.4% which makes the protocol proposed in [26] vulnerable by software modelling. [26] also reveals the raw helper data for error correction on the insecure channel. This exposure of helper data provides another attack vector for the adversary. Our protocol is secure against such ML-based modelling attacks because we do not explicitly reveal any raw challenge or the generated response directly on the channel and instead only reveal *masked* challenges and responses. Other than that, our raw helper data is never sent on the insecure communication channel as the server is responsible for implementing the error corrector (BCH decoder).

- **Server Encryption Key (K_S):** Our second security assumption is that the challenge C_s, which generates the server encryption key K_s, is not revealed to the $\mathcal{A}dv$. This means that even if the $\mathcal{A}dv$ reads the encrypted messages exchanged between the server and the database, it cannot decrypt those messages. The $\mathcal{A}dv$ can, however, modify or replay the previously sent encrypted messages.
- **Masking PRNG Polynomial:** Our final security assumption is that the LFSR-based PRNG circuit, used in the *MASK* function, has a characteristic polynomial which is not publicly available. This is because every device in the system has a unique *PRNG$_1$* structure. Making the *PRNG$_1$* structure publicly known makes little sense as there could be millions of devices in the system with each one having it's own structure. This assumption, however, certainly does not limit the capabilities of $\mathcal{A}dv$, as $\mathcal{A}dv$ can attempt to extract this information from the device through brute force or device *de-encapsulation*.

We will now present different attack scenarios through which the adversary $\mathcal{A}dv$ can attempt to break the protocol γ.

Case-1: Impersonation of Node A:

- We assume that the PPT $\mathcal{A}dv$ monitors Node A ever since it's enrollment into the system and stores the data, associated to all the previous authentication cycles d, between the server S and A and between S and the database D.
- After d authentication cycles, A hands over it's entire design to $\mathcal{A}dv$ including the details of PUF, the TRNG etc. $\mathcal{A}dv$, however, does not have the information about *PRNG$_1$*. We will represent this *fake* node, controlled by $\mathcal{A}dv$, as A_F.
- A_F initiates the $d+1^{st}$ authentication cycle with S. It sends a previously used valid device nonce $\mathbf{n_A}$ to S.
- The server, upon receiving the nonce, performs its own set of initial operations which include, extracting the encrypted data from D, setting up its *PRNG$_1$* and generating the nonce $\mathbf{n_s}$. S and A_F, after the first few operations, will both have a set of four random numbers. A_F will have $\{\mathbf{n_A}, \mathbf{n_{AM}}', \mathbf{n_{S_1}}, \mathbf{n_{S_1M}}'\}$ and S will have $\{\mathbf{n_A}, \mathbf{n_{AM}}, \mathbf{n_{S_1}}, \mathbf{n_{S_1M}}\}$. Here, we can see that since A_F does not have the characteristic polynomial of the *MASK* PRNG, thus $\mathbf{n_{AM}} \neq \mathbf{n_{AM}}'$ and $\mathrm{n_{S_1M}} \neq \mathbf{n_{S_1M}}'$.
- The server S will send $N_{AM}||c_{AM}$ to A_F both of whom are masked using $\mathbf{n_{AM}}$. A_F will attempt to unmask them using $\mathbf{n_{AM}}'$ but will fail because of the random permutation property of the *MASK* function.
- A_F will attempt spoofing attack by re-sending the $\mathbf{r_{mask}}$ associated to the authentication in which the valid node A sent the nonce $\mathbf{n_A}$. This will fail however, because $\mathbf{r_{mask}}$ is generated by *XOR*-ing the noisy response, $\mathbf{r_{noisy}}$, with the current $\mathbf{n_{S_1M}}$ and then masked using $\mathbf{n_{S_1M}}$. Therefore, the server will fail to authenticate the *fake* node A_F.

Case-2: Impersonation of Server S:

- We assume that the PPT adversary $\mathcal{A}dv$ corrupts the server and takes control of it with the exception of C_s used to generate the encryption key K_S. We will call this *fake* server as S_F. The main goal of S_F is to validate itself to two IoT nodes A and B, establish a *fake* key-exchange between the two and then, listen to the communication and data exchanged between the two.
- Upon receiving a query from a valid node A, S_F generates $\mathbf{n_{S_1F}}$ and sends it to node A. It also receives encrypted data from D.
- Since S_F does not have the encryption key K_S, it can not decrypt the data obtained and thus, can not extract the *PRNG$_1$* design. It resorts to using some random PRNG design instead. The probability of this PRNG design being the same as a valid design, designated for this node, is the reciprocal of $t_q(m)$ given in (19). For $m = 64$, this probability comes out to be $\approx 10^{-18}$.
- The node A will send its own $\mathbf{n_A}$ to S_F. Both A and S_F will mask theirs and each others respective nonces to produce a set of four random numbers. Node A will have $\{\mathbf{n_A}, \mathbf{n_{AM}}, \mathbf{n_{S_1F}}, \mathbf{n_{S_1FM}}\}$ and S_F will have $\{\mathbf{n_A}, \mathbf{n_{AM}}', \mathbf{n_{S_1F}}, \mathbf{n_{S_1FM}}'\}$. Similar to the previous case, in this case, $\mathbf{n_{AM}}' \neq \mathbf{n_{AM}}$ and $\mathbf{n_{S_1FM}} \neq \mathbf{n_{S_1FM}}'$.

- Now, again, because the corrupted server S_F does not have access to C_s because of which it can not generate K_s, it can not decrypt the acquired encrypted data, it received from the database to produce $N_{AM}'||c_{AM}'$. Therefore, it will resort to sending some previously sent $N_{AM}||c_{AM}$ to the node A.
- Node A upon receiving $N_{AM}||c_{AM}$ will unmask it with the current set of private variables and will immediately recognize that $\mathbf{c_A} \neq \mathbf{c_A'}$. Thus, it will reject the corrupted server S_F and will not unlock its PUF to produce any response.

Case-3: Man-in-the-Middle Attack:

- We consider another protocol level attack known as *Man-in-the-Middle* (MITM) attack which is the extension of Cases 1 and 2. In this attack, the adversary secretly monitors the messages exchanged between legitimate communicating parties. The adversary can delay, alter, or eavesdrop the messages over an insecure network. The MITM is prevented by employing *mutual authentication*, where both parties mutually authenticate each other's messages. In the proposed protocol, masking/unmasking is used as a lightweight message authentication scheme to prevent the MITM attack.
- The first target of the MITM attack for the $\mathcal{A}dv$, in this case, is the message $N_{AM}||\mathbf{c_{AM}}$, sent from the server to the device; however, as seen from Fig. 3, $N_{AM}||\mathbf{c_{AM}}$ is masked by the server and subsequently unmasked by the device. If the unmasking process in the device does not generate a valid $\mathbf{c_A}$, the protocol aborts. Because of the nature of the masking and the unmasking function, as explained in the previous cases, only a valid $N_{AM}||\mathbf{c_{AM}}$ will produce a correct $\mathbf{c_A}$. Any manipulation of $N_{AM}||\mathbf{c_{AM}}$ will produce an incorrect $\mathbf{c_A}$ which will be rejected by the device. Thus, we can see that message authentication is performed at the device side which thwarts the MITM attack.
- The second attack point of an MITM attack is the $\mathbf{r_{mask}}$ sent from the device to the server, as shown in Fig. 3. Here too, the device masks the noisy response generated from the PUF. The server unmasks the response and generates the error-corrected response $\mathbf{r_{corrected}}$. Only a valid masking/unmasking pair will generate the correct response at the server side, and subsequently authenticate the device. Thus, message authentication is also performed at the server side and any manipulation in the exchanged messages will result in the abortion of the protocol.

Case-4: Leakage of PRNG Feedback Polynomial:

- We consider the final case where we assume that the entire internal design of the device including the masking PRNG feedback polynomial has been revealed to the adversary $\mathcal{A}dv$ by device de-encapsulation. The $\mathcal{A}dv$, using this design, reconstructs another Node A'.
- Upon initiating the protocol γ with the server S, both A' as well S generate the same set of the private variables $\{\mathbf{n_{AM}}, \mathbf{n_{S_1M}}\}$.

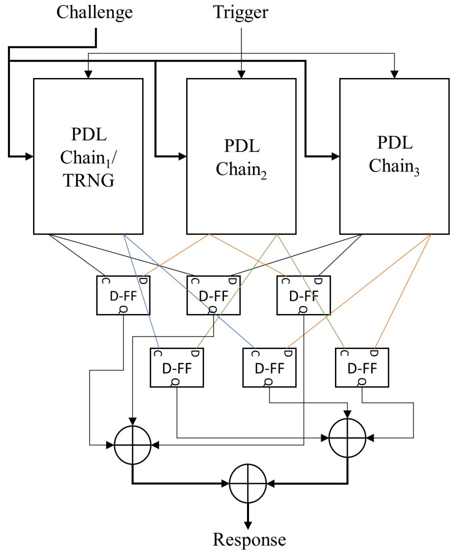

Fig. 9. 3-1 DAPUF

- A' upon receiving $N_{AM}||c_{AM}$ from S correctly unmasks the two parameters to generate $\mathbf{c_A}$ and $\mathbf{c_A'}$ such that $\mathbf{c_A} = \mathbf{c_A'}$.
- A' unlocks its PUF circuit and provides $\mathbf{c_A}$ as input to the circuit and generates $\mathbf{r_{noisy}}$. From this $\mathbf{r_{noisy}}$, A' generates $\mathbf{r_{mask}}$ and sends it to S.
- S upon receiving $\mathbf{r_{mask}}$, unmasks it to retrieve $\mathbf{r_{noisy}}$ and uses $HELP_A$ of the original node A to correct $\mathbf{r_{noisy}}$. However, because of the **PUP**, the corrected response $\mathbf{r_{corrected}}$ does not match with $\mathbf{r_A}$ and S fails to authenticate A'.

Thus, the security of protocol γ, for Case-4, relies on the inability of $\mathcal{A}dv$ to solve the PUF uniqueness problem (PUP).

The only way the $\mathcal{A}dv$ can successfully break the protocol is by using side-channel analysis attack to extract the PRNG design from the device without destroying the device's structure. Even then, the security of the network of devices will not be broken since only one device will be compromised and can easily be un-enrolled from the database without affecting the security of any other device. Moreover, side-channel attacks require expensive equipment and are generally not feasible. Also, many techniques have been proposed [34], [35] which can provide strong countermeasures against side-channel analysis and can be easily implemented without increasing the hardware overhead dramatically.

6 Experimental Setup and Implementation

We use Xilinx *Zynq-7000* zc706 evaluation board, as the device, for the implementation of the proposed scheme. The programmable logic (PL) part of the Zynq board implements a finite state machine (FSM) which goes through all the stages of the device presented in Fig. 3. The FSM calls all the major routines including the TRNG/PUF circuit, the *MASK* function and the PRNGs. The processing system (PS) part of the Zynq board is responsible for the elliptic curve point operations as well as the communication with a desktop/server running MATLAB R2018b via TCP/IP using lightweight IP stack (LwIP).

6.1 PUF

Figure 9 shows a standard 3-1 DAPUF consisting of 3 programmable delay lines (PDL) [36] and six D type flip

TABLE 1
3-1 DAPUF Evaluation Metrics

Property	Ideal	3-1 APUF	3-1 DAPUF
Uniqueness (%)	50	6.34	51.7
Reliability (%)	100	97.8	87.5
Randomness (%)	50	54.33	53.2

flops. The D, clock input (C) and the output (Q) connections are also shown in Figure 9. The working and structure of 3-1 DAPUF was first shown in [37]. We use the same structure in this work. Table 1 shows a comparison of the various evaluation metrics of a 3-1 DAPUF and a basic 3-1 XOR APUF. The table is taken by averaging out the values generated after the analysis performed by [32].

6.1.1 Uniqueness

Uniqueness ($\mathcal{U}$) tells the amount of variation in PUF responses among different chips/instances. Since one of the security assumptions of this protocol is high uniqueness of the PUF circuit, thus, during the design phase, care must be taken to choose a PUF which provides good uniqueness measures. For two chip instances, x and y $(x \neq y)$, having m-bit responses, R_x and R_y, respectively for a challenge C, the average inter-chip hamming distance (HD) or $\mathcal{U}$ among q chips is given by:

$$\mathcal{U} = \frac{2}{q(q-1)} \sum_{x=1}^{q-1} \sum_{y=x+1}^{q} \frac{HD\left(R_x, R_y\right)}{m} \times 100\%, \tag{25}$$

As shown in Table 1, the uniqueness of a basic 3-1 XOR APUF is very low which makes it unsuitable for this and many other authentication protocols which rely on the PUF uniqueness. In comparison, the 3-1 DAPUF has a uniqueness much closer to the ideal value which makes it much more suitable for the authentication purposes.

6.1.2 Reliability

For reliability testing, the PUF circuit is evaluated under varying conditions (e.g., voltage and temperature) and HD is calculated between the ideal and the obtained responses. Ideally, the same PUF instance should output the same response given a particular challenge. However, this is usually not the case and the generated responses have bit flips. To calculate the reliability ($\mathcal{R}$) of a chip x, an input challenge C is provided to the PUF at normal operating condition and an m-bit *reference* response R_x is recorded. The device x is subjected to different operating conditions (temperature and voltage), the same challenge C is applied and the m-bit response R'_x is recorded. A total of t samples of R'_x are recorded. The reliability ($\mathcal{R}$) of the chip x is given as:

$$\mathcal{R} = 100\% - \frac{1}{t} \sum_{i=1}^{t} \frac{HD\left(R_x, R'_{x,j}\right)}{m} \times 100\%, \tag{26}$$

where $R'_{x,j}$ is the j^{th} sample of the response string R'_x. In case of 3-1 DAPUF, the reliability is, on average, 13% less then the ideal value and can go further down by, as high as, 15% [32]. This is a very high error rate and can significantly impact the authentication reliability. Without error correction, the server and the device will have to restart the protocol in case of errors in the PUF response which can significantly increase the execution time. Implementing error correction in the device, as was the case for all previous approaches employing error correction, significantly increases the area overhead. Furthermore, since error correction requires a particular b number of *syndrome/helper* bits, to be communicated to the device, an adversary can get b bits about the PUF delay circuit. Since, the proposed scheme provides the flexibility to implement error correction at the server end, in addition to not exposing the raw helper bits on any communication link, a good error correction scheme can be implemented in software on the server without increasing the device's area overhead. As a test case, we implement BCH encoder/decoder in software using MATLAB R2018b using the *bchenc/bchdec* function. The implemented BCH decoder has the capability to correct up-to 20% error rate. This accounts to a practical reliability of ∼99% in addition to a ∼60% decrease in the area overhead of the device.

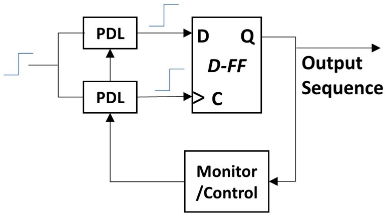

Fig. 10. TRNG Feedback Loop

6.1.3 Randomness

The final evaluation metric of a PUF is it's randomness. Randomness refers to the ratio of 0's and 1's in the PUF responses. Ideally, a PUF response should comprise of equal number of 0's and 1's. As shown in Table 1, the 3-1 DAPUF has, on average, a randomness of 53% which is close to the ideal value.

6.2 TRNG

As mentioned before, the nonce from the device, is generated using the left most chain of the 3-1 DAPUF circuit, as shown in Fig. 9. The main FSM controls whether the PUF needs to be used for CRP-based authentication or for generation of the random nonce. We use the approach presented in [38] for generation of nonces in the PL of the *Zynq-7000*. The operation of this TRNG is governed by enforcing a metastable state on a flip-flop through a closed loop feedback control as shown in Fig. 10. A metastable condition on a D-flip flop occurs when the setup/hold time of the flip flop is violated. The data captured by the flip-flop in this state is unpredictable and this unpredictability serves as a source of randomness for the nonce generation.

After implementing the basic controller design, we validate the findings in [38] that the nonces generated by a simple feedback loop were too biased and failed most of the tests of the NIST design suite. To circumvent this problem, we added a Von Neumann post processing filter, as indicated by [38], for unbiasing the TRNG output. In our design, the filter waits for the TRNG to produce two bits. If the two bit pattern is 01, the filter outputs a 0. If the bit pattern is 10, the filter outputs a bit 1. For the other two cases, when the input is 00 or 11, the filter discards them and waits for a new pattern. A counter is maintained which adjusts the feedback loop based on the number of 1's. If the number of 1's in the final m-bit nonce are greater than a threshold specified by the design variable τ_1, the nonce is discarded and the process is repeated again. Because of the unbiasing done by the Von Neumann corrector, the random nonces generated, satisfied all the NIST randomness tests as indicated in [38]. Other than providing good randomness properties, the hardware footprint of this TRNG is extremely low since the PUF chain is multiplexed.

6.3 MASK Function

The masking operation requires three main components: PRNG, range adjuster and a bit shuffler. The PRNG circuit uses an m-bit register and the total number of LUTs equal to the number of feedback taps representing the primitive polynomial. The range adjuster requires one multiplier. The bit shuffler is implemented using a *dual port BRAM* primitive in the FPGA. The BRAM has independent read-write address ports. The input data, to be shuffled, is first written into the BRAM in binary. Thus, the space complexity of BRAM for an m-bit input data is only $\mathcal{O}(m)$. The output from the range adjuster is used as the read address and a decrementing counter is used as the write address of the BRAM. This way, the *for* loop in Algorithm 2 is implemented in hardware. Bit swapping is done during each clock cycle till the decrementing counter goes from m to 0.

6.4 Hardware Footprint and Latency Overhead

For the implementation, our system parameters are given in Table 3. The hardware footprint and the latency of the modules in the PL of *Zynq*, based on the system parameters in Table 3, are shown in Table 2. The PL of *Zynq* implements all the modules given in Table 2 whereas, the PS of *Zynq* implements the ECC operations. We use the same ECC curve given in [26] in this work.

All the operations required by the server including error correction, PRNGs, masking function and elliptic curve operations are implemented on MATLAB running on *Intel Xeon E5-1620*@$3.50GHz$. The data is communicated to the device using MATLAB's TCP/IP API. The security of the scheme is evaluated by acquiring Ethernet packets exchanged between the device and the server using Wireshark. Based on these packets, impersonation attacks, explained in section 5, were carried out but none of the attacks were successful even after acquiring data from more than 50,000 valid authentication cycles.

6.5 Comparison With Other PUF-Based Protocols

Tables 4 and 5 show the comparison of various properties and hardware overhead, respectively, between PUF-RAKE and the past approaches.

6.5.1 Scalability

PUF-RAKE, similar to the past approaches, is scalable. Devices can easily be added into the PUF-RAKE network without any burden on the existing devices and the server. The memory requirement of the cloud storage will increase very slightly. We assume that the new device has a total of 10,000 CRPs with each challenge and response being 64 bits in length. The ID length of the device is also 64 bits as is the PRNG polynomial length. The helper data associated to one challenge is 256 bits. Thus, the total data bits associated with one authentication cycle of a particular device are $64 + 64 + 64 + 64 + 256 = 512$ bits. This data is encrypted using AES galois counter mode (GCM) of authenticated encryption with the encrypted output being 512 bits in length and the authentication tag being 128 bits long. Thus, the total memory requirement for one device will only be $10,000 \times (512 + 128) = 6.4$ Mb or 0.8 MB. Assuming that we have a total of 1 million devices in a particular network, this would only correspond to a memory requirement of approximately 0.8 TB which is very meager considering tons of terabytes of memory available in commercial cloud-based services.

6.5.2 Mutual Authentication

PUF-RAKE supports full mutual authentication which means that both the device and the server mutually authenticate each other during an authentication run. The protocols [9] and [15] only support authentication on the device end and thus, these protocols can be broken by replay attacks on the server.

6.5.3 Cryptographic Algorithm in the Device

PUF-RAKE does not contain any cryptographic hashing scheme in the device. This makes it suitable for low cost, resource constraint platforms. Many PUF-based protocols proposed in the past including [9], [11], [26] include hashing inside the device which significantly increases the latency and area overhead in the device as shown in Table 4.

6.5.4 Error Correction and Exposure of Helper Data

As already shown, PUF-RAKE is unique when compared against the previous protocols in a way that it does not include error correction in the device. This significantly decreases the hardware overhead of the device. It also ensures that no helper data is exposed on the channel which can potentially make the device vulnerable to side-channel attacks.

6.5.5 Server-Database Link

PUF-RAKE assumes that the link between the server and the database is open and the adversary has full access to this link. This makes the protocol realizable in practical scenarios where the device data can be offloaded to a cloud-based storage. Without opening the link between the server and the database, as is the case for [9], [10], [11], [15], the server will need a secure memory to store the data for all the devices in the network which can prove to be very costly.

6.5.6 Open Device Interface

PUF-RAKE, because of its construction, assures that the devices' interface is not open to random queries. If the adversary tries to perform brute force attack on the device by sending random messages, the device will not respond and thus, no data can be gathered by the adversary. This property is present because of the presence of the *MASK* function. Many protocols, including the recently proposed [26] has an open device interface which makes them vulnerable to attacks focusing on brute force.

6.5.7 Key Establishment

PUF-RAKE supports key establishment between two devices in a network. This makes it deployable in various real world scenarios including IoT device network, home automation, automotive communication and many more. Some recently proposed PUF-based protocols including [24], [26], [27], [28] support key establishment but all of them

TABLE 2
Hardware and Latency Overhead

Module	LUT count	FF count	BRAM	No. of Operations	Latency
3-1 DAPUF	387	6	0	1	$\approx 250\mu s$
TRNG	Shared with PUF	12	4K	1	$\approx 46\mu s$
MASK / UNMASK	150	128	4K	7	$\approx 98 \times 7 = 686\mu s$
PRNG_1	2	66	0	1	$\approx 2\mu s$
main FSM	97	380	0	1	$\approx 5\mu s$
Total	**636**	**592**	**8K**	**11**	**≈1ms**

TABLE 3
System Parameters

Parameter	Value
Clock Freq. (MHz)	50
nonce length (m)	64
HW(τ_1) range	$25 \leq \tau_1 \leq 39$
Length of PUF chains	64
Challenge Length	64
Response Length	64

TABLE 4
Hardware Overhead Comparison

Protocols	LUT count	FF count
[9]	not reported	not reported
[10]	not reported	not reported
[11]	960	1500
[26]	1591	1933
[24]	9207	2921
[27]	6034	1724
[28]	3543	1275
PUF-RAKE	636	592

TABLE 5
Comparison against Previous Protocols

Property	[15]	[10]	[11]	[16]	[21]	[26]	This
Scalable	✓	✓	✓	✓	✓	✓	✓
Mutual Auth.	✗	✓	✓	✓	✓	✓	✓
Crypto Algo.	✗	✗	✓	✓	✗	✓	✗
Error Correction.	✗	✗	✗	✗	✗	✓	✗
Help Data Exposed	✗	✗	✗	✓	✗	✓	✗
Auth. Rounds	∞	d	∞	∞	∞	∞	∞
S-D Link open	✗	✗	✗	✓	✗	✓	✓
Open Device Interface	✓	✗	✗	✓	✓	✓	✗
Key Establishment	✗	✗	✗	✗	✗	✓	✓

suffer from large overhead and high latency problems as shown in Table 4.

Overall, it can be seen that the PUF-RAKE not only retains the cumulative advantages of all the previous approaches while circumventing their limitations, it also reduces the hardware overhead to a great extent without compromising the security and reliability. [26] provides advantages somewhat similar to PUF-RAKE but PUF-RAKE has ∼60% and ∼72% reduction in LUT and FF count, respectively. Also, PUF-RAKE closes the open device interface which [26] does not. [24], [27], [28] have a huge area overhead, as shown in Table 4, which makes them unsuitable for low cost platforms.

7 Conclusions

In this paper, we have proposed PUF-RAKE, a controlled PUF-based, authentication and secret key establishment protocol, which (i) closes the open interface between the input and the PUF by implementing a strong control logic that denies the PUF's access to the adversaries, (ii) makes the scheme highly reliable by incorporating error correction in the server thereby not revealing any helper data on insecure channels, (iii) reduces the hardware overhead drastically by incorporating a lightweight CRP obfuscation mechanism employing bit shuffling and XOR operations, and (iv) performs key establishment between two or more nodes in a network, thereby enabling communication between the devices. The security of the shuffling scheme of PUF-RAKE has been formally verified. Many different adversarial test cases have been considered and it has been shown that PUF-RAKE is secure against all of the considered adversarial attacks. Results also reveal that PUF-RAKE is highly reliable and provides 99% reliable authentication in addition to being extremely lightweight. It provides a reduction of 60% and 72% for look-up tables (LUTs) and register count, respectively, in FPGA as compared to a recently proposed approach while furnishing many additional advantages. Our future goal is to incorporate PUF-RAKE into application-specific networking applications including IoT and automotive and evaluate its performance over traditional approaches used in these applications. We also plan to evaluate the security of PUF-RAKE against side-channel attacks.

Mahmood Azhar Qureshi is currently a Ph.D. Candidate in the Department of Computer Science (CS) at Kansas State University (K-State). He received his B.S. in Electrical Engineering from National University of Science and Technology (NUST), Pakistan in 2013 and M.S. in Electrical Engineering from the University of Engineering and Technology (UET), Taxila, Pakistan in 2018. He worked as a Senior Design Engineer at Center for Advanced Research in Engineering (CARE) Pvt. Ltd, Islamabad, Pakistan from 2014 to 2018. His research interests include hardware security, computer architecture, and design validation.

Arslan Munir (M'09, SM'17) is currently an Assistant Professor in the Department of Computer Science (CS) at Kansas State University (K-State). He holds a Michelle Munson-Serban Simu Keystone Research Faculty Scholarship from the College of Engineering. He was a postdoctoral research associate in the Electrical and Computer Engineering (ECE) department at Rice University, Houston, Texas, USA from May 2012 to June 2014. He received his M.A.Sc. in ECE from the University of British Columbia (UBC), Vancouver, Canada, in 2007 and his Ph.D. in ECE from the University of Florida (UF), Gainesville, Florida, USA, in 2012. From 2007 to 2008, he worked as a software development engineer at Mentor Graphics Corporation in the Embedded Systems Division.

Munir's current research interests include embedded and cyber-physical systems, secure and trustworthy systems, hardware-based security, computer architecture, parallel computing, reconfigurable computing, artificial intelligence (AI) safety and security, and fault tolerance. Munir received many academic awards including the doctoral fellowship from Natural Sciences and Engineering Research Council (NSERC) of Canada. He earned gold medals for best performance in electrical engineering, gold medals and academic roll of honor for securing rank one in pre-engineering provincial examinations (out of approximately 300,000 candidates). He is a Senior Member of IEEE.

References

[1] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Annual international cryptology conference*, pages 213–229. Springer, 2001.

[2] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. Physical unclonable functions and public-key crypto for FPGA IP protection. In *2007 International Conference on Field Programmable Logic and Applications*, pages 189–195. IEEE, 2007.

[3] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proc. of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 148–160, New York, NY, USA, 2002.

[4] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *Proc. of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 237–249, Chicago, IL, USA, 2010.

[5] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Techniques for design and implementation of secure reconfigurable PUFs. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 2(1):5, 2009.

[6] Jiliang Zhang, Lu Wan, Qiang Wu, and Gang Qu. DMOS-PUF: Dynamic multi-key-selection obfuscation for strong PUFs against machine learning attacks. *arXiv preprint arXiv:1806.02011*, 2018.

[7] M. S. Alkatheiri and Y. Zhuang. Towards fast and accurate machine learning attacks of feed-forward arbiter PUFs. In *IEEE Conference on Dependable and Secure Computing*, Aug 2017.

[8] C. Zhou, K. K. Parhi, and C. H. Kim. Secure and reliable xor arbiter puf design: An experimental study based on 1 trillion challenge response pair measurements. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.

[9] Blaise Gassend, Marten Van Dijk, Dwaine Clarke, Emina Torlak, Srinivas Devadas, and Pim Tuyls. Controlled physical random functions and applications. *ACM Trans. on Information and System Security (TISSEC)*, 10(4):3, 2008.

[10] Meng-Day Yu, Matthias Hiller, Jeroen Delvaux, Richard Sowell, Srinivas Devadas, and Ingrid Verbauwhede. A lockdown technique to prevent machine learning on PUFs for lightweight authentication. *IEEE Trans. on Multi-Scale Computing Systems (TMSCS)*, 2(3):146–159, 2016.

[11] Yansong Gao, Hua Ma, Said F Al-Sarawi, Derek Abbott, and Damith C Ranasinghe. PUF-FSM: A controlled strong PUF. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 37(5):1104–1108, 2018.

[12] Lars Tebelmann, Michael Pehl, and Georg Sigl. EM Side-Channel Analysis of BCH-based Error Correction for PUF-based Key Generation. In *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security*, ASHES '17, New York, NY, USA, 2017. ACM.

[13] Mahmood Azhar Qureshi and Arslan Munir. PUF-RLA: A PUF-based Reliable and Lightweight Authentication Protocol employing Binary String Shuffling. In *Proc. of IEEE International Conference on Computer Design (ICCD)*, Abu Dhabi, U.A.E., November 2019.

[14] Georg T Becker. On the pitfalls of using arbiter-PUFs as building blocks. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 34(8):1295–1307, 2015.

[15] M Rostami, M Majzoobi, Farinaz Koushanfar, Dan S Wallach, and Srinivas Devadas. Slender PUF protocol: A lightweight, robust, and secure authentication by substring matching. In *IEEE Symposium on Security and Privacy Workshops*, pages 33–44, San Francisco, CA, USA, 2012.

[16] Anthony Van Herrewege, Stefan Katzenbeisser, Roel Maes, Roel Peeters, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-Enabled RFIDs. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, pages 374–389, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[17] Siam Umar Hussain, M. Sadegh Riazi, and Farinaz Koushanfar. SHAIP: Secure Hamming Distance for Authentication of Intrinsic PUFs. *ACM Trans. Des. Autom. Electron. Syst.*, 23(6), December '18.

[18] Erdinç Öztürk, Ghaith Hammouri, and Berk Sunar. Towards robust low cost authentication for pervasive devices. In *2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 170–178. IEEE, 2008.

[19] M. A. Qureshi and A. Munir. PUF-IPA: A PUF-based Identity Preserving Protocol for Internet of Things Authentication. In *2020 IEEE 17th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–7, 2020.

[20] Stefan Katzenbeisser, Ünal Kocabaş, Vincent Van Der Leest, Ahmad-Reza Sadeghi, Geert-Jan Schrijen, and Christian Wachsmann. Recyclable PUFs: Logically reconfigurable PUFs. *Journal of Cryptographic Engineering*, 1(3):177, 2011.

[21] S. S. Zalivaka, A. A. Ivaniuk, and C. Chang. Reliable and Modeling Attack Resistant Authentication of Arbiter PUF in FPGA Implementation with Trinary Quadruple Response. *IEEE Transactions on Information Forensics and Security*, 14(4):1109–1123, 2019.

[22] Ünal Kocabaş, Andreas Peter, Stefan Katzenbeisser, and Ahmad-Reza Sadeghi. Converse PUF-based authentication. In *International Conference on Trust and Trustworthy Computing*. Springer, 2012.

[23] Marten van Dijk and Ulrich Rührmair. Physical unclonable functions in cryptographic protocols: Security proofs and impossibility results. 2012.

[24] J. Kong, F. Koushanfar, P. K. Pendyala, A. Sadeghi, and C. Wachsmann. PUFatt: Embedded platform attestation based on novel processor-based PUFs. In *51st ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, USA, June 2014.

[25] Jeroen Delvaux, Dawu Gu, Dries Schellekens, and Ingrid Verbauwhede. Secure lightweight entity authentication with strong PUFs: Mission impossible? In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014.

[26] U. Chatterjee, V. Govindan, R. Sadhukhan, D. Mukhopadhyay, R. S. Chakraborty, D. Mahata, and M. M. Prabhu. Building PUF based authentication and key exchange protocol for IoT without explicit CRPs in verifier database. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.

[27] Wenjie Che, Mitchell Martin, Goutham Pocklassery, Venkata K. Kajuluri, Fareena Saqib, and James F. Plusquellic. A privacy-preserving, mutual PUF-based authentication protocol. *Cryptography*, 1:3, 2016.

[28] Aydin Aysu, Ege Gulcan, Daisuke Moriyama, Patrick Schaumont, and Moti Yung. End-to-end design of a PUF-based privacy preserving authentication protocol. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, pages 556–576. Springer Berlin Heidelberg, 2015.

[29] Christof Paar and Jan Pelzl. *Understanding cryptography: for students and practitioners*. Springer Science & Business Media, 2009.

[30] Tapan Kumar Hazra, Rumela Ghosh, Sayam Kumar, Sagnik Dutta, and Ajoy Kumar Chakraborty. File encryption using Fisher-Yates shuffle. In *International Conference and Workshop on Computing and Communication (IEMCON)*, pages 1–7, Vancouver, British Columbia, Canada, 2015.

[31] Abhranil Maiti, Vikash Gunreddy, and Patrick Schaumont. A systematic method to evaluate and compare the performance of physical unclonable functions. In *Embedded systems design with FPGAs*, pages 245–267. Springer, 2013.

[32] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama. Implementation of double arbiter PUF and its performance evaluation on FPGA. In *The 20th Asia and South Pacific Design Automation Conference*, pages 6–7, Jan 2015.

[33] H. Awano, T. Iizuka, and M. Ikeda. PUFNet: A deep neural network based modeling attack for physically unclonable function. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2019.

[34] Eric Peeters, François-Xavier Standaert, Nicolas Donckers, and Jean-Jacques Quisquater. Improved higher-order side-channel attacks with FPGA experiments. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2005.

[35] M. Nassar, Y. Souissi, S. Guilley, and J. Danger. RSM: A small and fast countermeasure for AES, secure against 1st and 2nd-order zero-offset SCAs. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1173–1178, March 2012.

[36] M. Majzoobi, F. Koushanfar, and S. Devadas. FPGA PUF using programmable delay lines. In *IEEE Int. Workshop on Information Forensics and Security*, pages 1–6, Seattle, WA, USA, Dec 2010.

[37] T. Machida, D. Yamamoto, M. Iwamoto, and K. Sakiyama. A new mode of operation for arbiter puf to improve uniqueness on fpga. In *2014 Federated Conference on Computer Science and Information Systems*, pages 871–878, 2014.

[38] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. FPGA-based true random number generation using circuit metastability with adaptive feedback control. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 17–32, Berlin, Heidelberg, 2011. Springer-Verlag.