PUF-IPA: A PUF-based Identity Preserving Protocol for Internet of Things Authentication

Mahmood Azhar Qureshi* and Arslan Munir†
Department of Computer Science, Kansas State University
Email: *mahmood102@ksu.edu, and †amunir@ksu.edu

***Abstract*—Physically unclonable functions (PUFs) can be used for Internet of things (IoT) based identification, authentication and authorization. However, PUF based authentication systems are vulnerable to various attacks including, but not limited to, replay and modeling attacks. In this paper, we propose PUF-IPA, a PUF-based identity-preserving protocol for IoT device authentication. The PUF-IPA provides stronger resilience against security attacks as compared to previous approaches assuming a threat model where adversary can conduct not only passive or active attacks during authentication phase but can also breach the server storing PUF credentials. The proposed PUF-IPA is robust against brute force, replay, and modeling attacks. In PUF-IPA, no partial/full challenge-response pairs (CRPs) or soft models associated to a PUF within a device are stored, generated, transmitted, or received by the server during authentication events. The PUF-IPA improves the PUF response accuracy by enabling self-checking. Results reveal that the PUF-IPA improves the PUF response accuracy from 89% to 98% without the use of hardware-expensive error correction codes.**

***Index Terms*—Lightweight authentication, PUFs, identity-preserving**

I. Introduction

Internet of things (IoT) paradigm connects a wide variety of heterogeneous devices to help realize an increasing number of novel applications in different domains including medical, defense, transportation, agriculture, and automation. Many of the IoT applications require IoT devices to be authenticated before joining the network as well as during operation on a regular basis to verify the legitimacy of the devices. Since many of the IoT devices have stringent resource constraints, certificate-based authentication does not provide a feasible and scalable solution for all IoT applications. Consequently, there is a need to investigate other methods for IoT authentication.

Physically unclonable functions (PUFs) [1] exploit the physical randomness associated with manufacturing variations in a device. These variations can be utilized for enhancing the device's security as they are not reproducible. A PUF, because of its uniqueness, can be considered a private fingerprint associated to a particular device, and thus can be used for IoT device authentication. However, this fingerprint and the identity of a device needs to be kept private.

Even though PUF-based authentication methods provide a very efficient solution for authentication of resource constraint IoT devices, they are not completely foolproof. Rührmair et al. [2] have demonstrated that advance machine learning (ML)-based modeling techniques can be used to break the unpredictability and therefore, the security of strong PUFs (SPUFs), previously considered secure. Similarly, Zhang et al. [3] have shown that given a small number of challenge-response pairs (CRPs) of a 64x64 *Arbiter PUF* [4], an adversary can easily and efficiently build a software model for the device's PUF with a prediction accuracy of 99.9%. These modeling attacks can reproduce the behavior of the responses of the actual device and make it susceptible to attacks. Only tapping the various communication links between the device and the server, an adversary can still generate an accurate prediction model based on the CRP exposure.

To thwart model building attacks, Controlled PUFs (CPUFs) are introduced in [5]. These PUFs prevent model-building attacks by encapsulating the device's PUF within a control logic. [6], [7] build a control logic which limits the exposure of the CRPs for the adversary. [3] obfuscates the CRPs in such a way that even if the adversary collects a number of CRPs, no efficient model can be built since the original CRP relationship is only known to the device and the server.

Preserving the identity and privacy of the PUF, and thus device as PUF is fingerprint of the device, during authentication is challenging. In the prior works [8], [6], [7], [3], the authors have assumed that the server is secure and that the identity of the devices in the system are known to the server. This identity is the parameters of the PUFs in the devices including original, unaltered CRPs and/or their software models. For these approaches, if the server gets breached, the adversary can easily get hold of all the parameters of PUFs in the devices therefore breaching the security of the entire network. Hence, developing an identity-preserving authentication mechanism is challenging and this work aims to address this research problem.

In this paper, we propose PUF-IPA that uses PUFs to provide *identity-preserving authentication*. The PUF-IPA stores obscured, uncorrelated information about the device's PUF in the server. The server authenticates the devices by using this information without having the original CRP data. Thus, even if the server gets breached and the data is acquired, the adversary will not be able to model the devices since the information about the PUF CRPs is only know to the device. Also, in [8], [6], [7], [3], it is assumed that the communication channel between the server and it's database is secure and an adversary cannot monitor the traffic on it. The proposed scheme considers this channel to be insecure with the adversary having the capability to monitor it. Results indicate that even after opening the channel for attacks, the proposed protocol still remains secure. This provides huge cost benefits as an IoT network can have a huge number of devices and

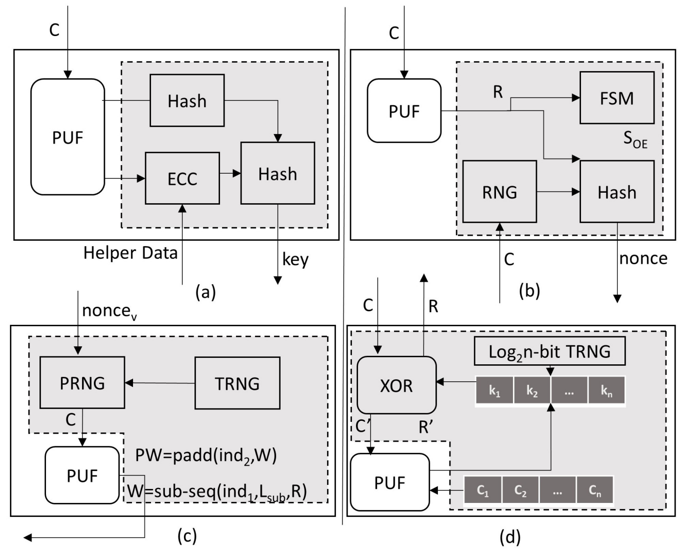


Fig. 1. Past approaches (a) Controlled PUF structure [5] (b) PUF-FSM [7] (c) Slender PUF [8] (d) DMOS-PUF structure [3]

storing the entire CRPs for all these devices' PUF in a *secure* server memory can have a huge cost. Our main contributions are as follows.

- We propose PUF-IPA that provides identity-preserving authentication for IoT devices employing bit shuffling. In PUF-IPA, the device is authenticated without storing CRPs or a model in the server's memory and the server authenticates the device based on obscured, uncorrelated information. The challenges are generated *on-the-fly* within the device by a shuffling algorithm. The security of the devices remains intact in the event of server breach.
- In PUF-IPA, the access to the PUF in an IoT device is *controlled*. A challenge is not issued and thus no response is generated without the application of a correct input stream to the device. This effectively locks out the device from unauthorized access. This protocol is the first that has the capability to lock out the device without even invoking the PUF within, and thus completely inhibits any model-building capability on the underlying PUF.
- The PUF-IPA provides an efficient and lightweight mechanism for IoT device authentication. Results verify that PUF-IPA has less area overhead as compared to other PUF-based authentication mechanisms while providing additional security advantages.

II. Related Work

Figure 1 shows the device side of the various PUF-based authentication schemes proposed in the past. Gassend *et al.*[5] hashes the input challenge as well as the output response but requires hardware-expensive error correction logic for stabilizing the noisy PUF responses. [5] also also transmits helper data to the device to be used by ECC. This exposure of helper data exposes the PUF to attacks focusing on noise side-channel information [9].

Yu *et al.* [6] upper bounds the limited number of CRPs to an adversary. In this case, only the server can validate the access to the new CRPs. [6] also introduces a device side nonce to prevent reliability-based attacks [9]. Gao *et al.* [7] have presented a finite state machine (FSM) based locking scheme 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. Application of a wrong challenge by an adversary will generate a response from the PUF which prevents the FSM from reaching the final state.

Slender PUF [8] uses neither an error-correction logic nor any cryptographic hashing but provides an open interface. An adversary can acquire information about CRPs as long as the device's interface access is maintained. [8] also does not provide mutual authentication. Zhang *et al.* [3] have proposed a dynamic multi key obfuscation structure (CMOS) which uses stable responses from a device's PUF as obfuscation keys. A particular number of stable challenges are stored in a non-volatile memory (NVM) inside the device. Responses are generated using these challenges and stored in register banks within the device as obfuscation keys. During authentication, two keys are selected randomly from the register bank by a true random number generator (TRNG). First key is XOR'ed with the input challenge and the second key is XOR'ed with the output response in the device. The obfuscated response is then sent back to the server for authentication.

All of the above approaches rely on the assumption that the server database cannot be breached. In all of the above approaches, the server stores full/partial CRPs or modeling parameters in a *secure* memory. Hence, a breach on the server-side can result in the compromise of entire network of devices.

III. Proposed Scheme

We treat the server as a breachable entity and thus attempt to protect the devices' privacy even in an event of breach. The only thing the server need to securely store is one encryption key. The PUF-IPA forces the verifying authority (i.e., a server/verifier) to go through a message/stream authentication block before the device's PUF can be evaluated. Figure 2 (a) and (b) depict the server-side and the device-side design of our protocol. The **stream authentication (SA)** block in the device is responsible for verifying the input to the PUF. The SA block serves two main purposes:

1) Preserves the privacy of the device by generating *on-the-fly* challenges within the device without explicitly storing any CRPs or a model in the server's database.
2) Renders the device inaccessible in case an adversary masquerades as a legitimate verifier and issues random challenges to the device in order to generate a model of the underlying PUF based on the the responses.

The server incorporates an advanced encryption standard (AES)-128 based encryption system, a comparator, and a querying mechanism for issuing queries and receiving responses from any global/local database in the system.

A. Threat Model

Like the previous authentication schemes [6], [7], [3], [5], [8], the proposed protocol consists of an *enrollment phase*

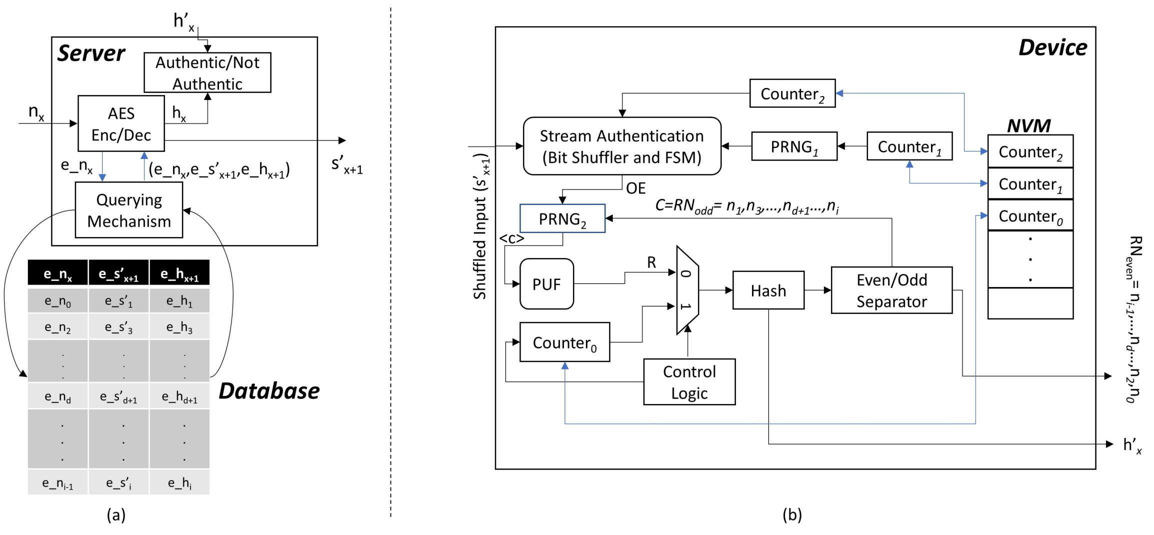


Fig. 2. Proposed scheme (a) Server-side along with the database, (b) Device-side

and an *authentication phase*. The enrollment phase occurs in a secure environment where the device's data (e.g. CRPs) is collected. All the devices in an IoT network are assigned unique identifiers by the server. The assigned IDs are represented as id_x where $x \in \mathbb{Z}^+$ denotes a device and $\mathbb{Z}^+$ denotes the set of positive integers (e.g., the ID of device 1 is denoted as id_1). The authentication of the devices occur in an insecure environment. Unlike previous approaches, the adversary, in our protocol, can eavesdrop, manipulate, and/or replay the data across all the communication links during various phases of authentication. These communication links include the channel between the server and the devices as well as between the server and its external/internal database storing the data. By using these links, the adversary can perform modelling attempts on the devices in the network. The adversary can also brute force query the device's open interface with any past or predicted future messages/challenges. We further assume that the adversary has complete read access to the server's database containing the data associated with various devices in the system.

B. Enrollment Phase

During the enrollment phase, different cryptographically secure random number streams are generated by hashing the output of $Counter_0$ as shown in Figure 2(b). The generated random numbers (RNs) are all 128 bits wide and represented as:

$$RNs = n_0, n_1, \ldots, n_d, \ldots, n_{i-1}, n_i, \ldots \quad (1)$$

where n_i is the i^{th} random number. Every even-indexed RN is encrypted and stored in the first column of the database (e_n_x) as shown in Figure 2(a). Here x represents the index number corresponding to an even index whereas $x+1$ corresponds to an odd index. Every odd-indexed RN is used as a seed for a pseudorandom number generator (PRNG), that is, $PRNG_2$ to derive sub-challenges ($<c>$) for the PUF. The corresponding responses are hashed then encrypted, and stored in third column of the database (e_h_{x+1}). We also store the encrypted, *shuffled* version of every odd indexed RN in the second column of the database (e_s'_{x+1}). The details of shuffling operation will be explained in Section III-D. After the completion of enrollment, the initial value of $Counter_0$, which generates the first RN, is stored in the device's NVM. It is worth mentioning here that since the enrollment occurs in a secure environment, the odd-indexed RNs $(n_1, n_3, ..., n_{d+1}, ...n_i, ...)$ are never exposed after enrollment rather their shuffled versions $(s'_1, s'_3, ..., s'_{d+1}, ..., s'_i, ...)$ are exposed. Also, the row indices for the database can be randomized such that the encrypted, first RN (e_n_0) and corresponding columns can be placed at an arbitrary row number within the database instead of the first row.

C. Authentication Phase

Figure 3 shows the series of operations during the first authentication event and can be extended to any authentication event *d*. The server initiates the authentication session and obtains the device identifier id' from the device. The server checks the id' obtained from the device against the id assigned to the device during enrollment for authenticity. The control logic in the device configures the MUX (MUX(1)), shown in Figure 2(b), to use $Counter_0$ value as input to the hash function. The device sends n_0, the first RN (even-indexed) generated after hashing the first $Counter_0$ value, to the server. The server encrypts n_0 and queries the first column of database to find the entry (e_n_0). The server retrieves the entire row (e_n_0,e_s'_1, e_h_1) and decrypts the row to get (n_0, s'_1, h_1). The server sends the second value s'_1 in the decrypted row to the device.

The device, after sending n_0, generates n_1 (odd-indexed), by hashing the next $Counter_0$ value. The RN n_1 is never sent out from the device and is only used by the SA block. The device then authenticates the server by passing s'_1, that device received from the server, to the SA block. If the SA block is evaluated correctly (i.e., $SA_{flags} = 1$), the device unlocks the PUF and sub-challenges are generated using n_1.

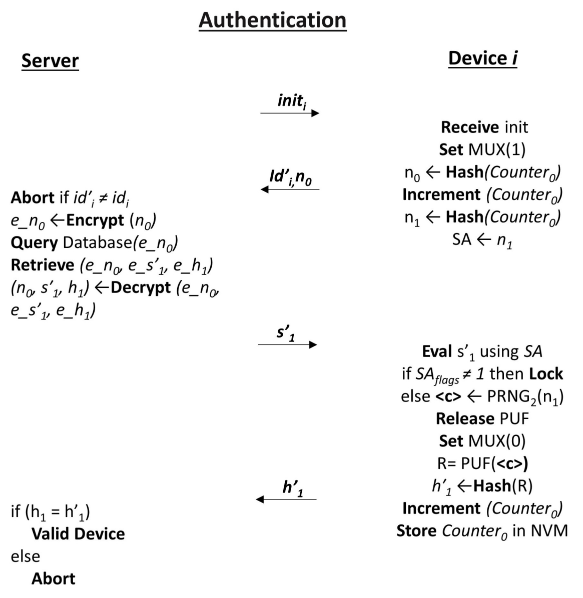


Fig. 3. Operational cycle of first authentication event

The control logic sets the MUX configuration (MUX(0)) to provide PUF response R as input to the hashing function. The hashed response (h'_1) is sent to the server for verification. The server verifies the device by comparing (h'_1) with the decrypted third entry (h_1) of the row that server retrieved from the database. If h'_1 from the device matches h_1, the device is successfully authenticated.

After the device sends the hashed response (h'_1) to the server, it increments $Counter_0$ and updates the NVM with the incremented value. The new value would be used to generate n_2, the next even-indexed RN, for the next authentication cycle. This is done in order to prevent the generation of n_0 again should the device be turned off and restarted which would reset $Counter_0$. This approach ensures that every authentication event is unique and has no correlation with past events.

D. Stream Verification

The Stream Authentication (SA) block separates the device's PUF from the input. It not only authenticates the server but also locks out the device in case an adversary tries brute forcing the device. This block uses a shuffling scheme followed by an FSM. The SA block also has a counter ($Counter_2$) that keeps track of the number of times a wrong input has been applied to the device.

1) Bit Shuffler: The PUF-IPA incorporates a modified Fisher-Yates shuffling algorithm [10]. The algorithm produces equally likely shuffles. Shuffling and deshuffling is performed during enrollment and authentication, respectively. Algorithm 1 shows the working of the shuffling operation.

Shuffling during Enrollment: During enrollment, for the first shuffled output corresponding to n_1 (i.e., s'_1), the $PRNG_1$ takes an initial counter value ($Counter_1$) as seed and generates a number sequence $[num_1, num_2, ..., num_{64}]$ used during the shuffling operation. The counter then increments and uses the incremented value as seed for $PRNG_1$ to generate a new number sequence for shuffling n_3, the second odd-indexed RN. In this way, the shuffled versions of all odd-indexed RNs are generated, encrypted (e_s'_{x+1}) and stored in the server's database during enrollment. The initial value of the counter used as first seed value of $PRNG_1$ is stored in the device's NVM.

Algorithm 1 Bit Shuffling

Input: n_{x+1}: 128-bit odd-indexed RN from the hash function
Output: s'_{x+1}: 128-bit shuffled version of the input

```
1: procedure BIT SHUFFLE
2:     Map $c_{64}, c_{63}, ..., c_1 \leftarrow b_{128}, ..., b_2, b_1$
3:     for $i \leftarrow 64$ to 1 do
4:         $j \leftarrow PRNG_1(\textbf{Range}(1, i))$
5:         Swap $(c_j, c_i)$
6:         $s'_{[65-i]} \leftarrow c_j$
7:     end for
8: end procedure
```

Deshuffling during Authentication: During authentication, s'_{x+1} is provided as input to the device and a deshuffling operation is performed. Taking the example of the first authentication event, s'_1 is provided as input to the device. The $PRNG_1$ gets the first counter value, and produces and temporarily saves the number stream $[num_1, num_2, ..., num_{64}]$. This stream is used in opposite manner (i.e., from num_1 to num_{64} in line 3 of algorithm 1) to reproduce the original number (i.e., n_1). The counter value is then incremented and the device's NVM is updated with this new value for the deshuffling of s'_3 in the next authentication cycle.

Illustrative Example of Shuffling and Deshuffling Operation: Figure 4 shows the process of shuffling and deshuffling on a small random test input.Note that this is just a test case and the actual input is 128-bits wide. An input bit string **100111001110** corresponding to random number *2510* is taken as an example. In the first step the input bits are grouped in a *tuple* of two bits. For the first run of Algorithm 1, an RN in the range *R (1,6)* is chosen as there are six two bit tuples corresponding to 12 bits in the test case. Assuming that the RN comes out to be 1, we swap the 1^{st} tuple of bits with the last tuple (*S (1,6)*). In the next run, we decrease the range by one (i.e., from *R (1,6)* to *R (1,5)*). Assuming that the RN comes out to be 2, we swap the 2^{nd} and the 5^{th} tuple (*S (2,5)*), and so on until we reach the last run (i.e., *R (1,1)*) where no swapping is done. The final shuffled output comes out to be **001011110110** corresponding to decimal *758*. Note that for any other random number stream, the output will be different then the one presented above. Deshuffling is performed using the same number stream but in opposite manner (i.e., from *R (1,1)* to *R (1,6)*) to regenerate the original input as shown in Figure 4(b). Note that the case of *R(1,1)* is not shown in Figure 4 (a) and (b) as no shuffling/deshuffling is performed.

2) FSM-based Locking: The second sub-block in *SA* is an **FSM**, which takes the deshuffled output as input and transitions through the FSM states. During the first authentication

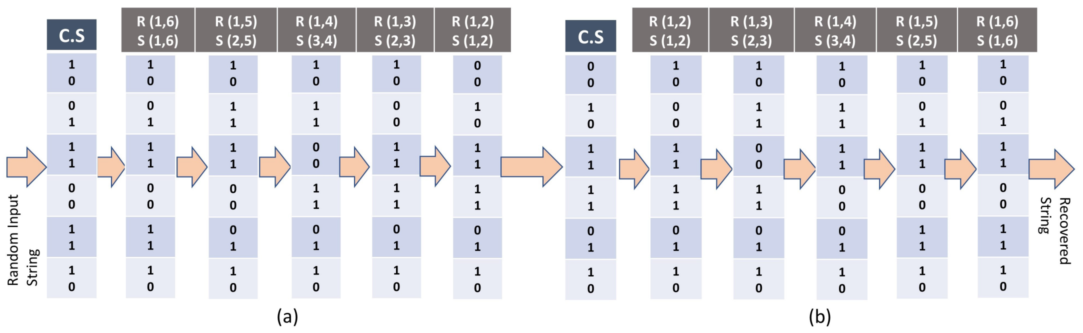


Fig. 4. (a) Shuffling of random input string, (b) Deshuffling to generate the original string

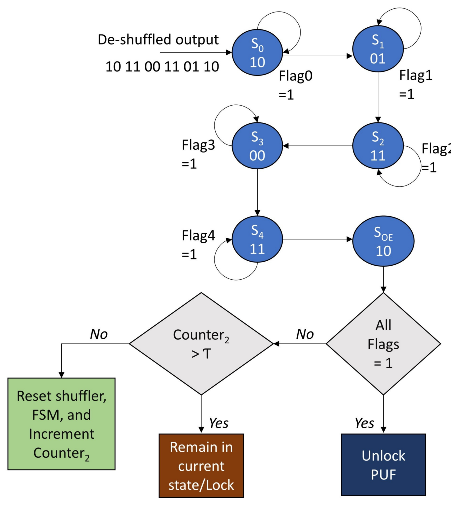


Fig. 5. FSM-based state traversions

event, n_1 generated within the device, is broken down into *k-bit* tuples and used as state transitions for the FSM. The deshuffled bit stream is also broken down into *k-bit* tuples and fed as input to the FSM. The FSM traverses through the m^{th} state if m^{th} *k-bit* tuple of the deshuffled output is equal to m^{th} *k-bit* tuple of n_1. This process continues for $128/k$ states from $S_0, S_1, ..., SOE_{128/k}$. The last state, SOE or the *Output Enable State*, is reached only if a correct stream of input is applied to the FSM which would be the case for a valid authentication event. If the deshuffled output comes out to be different than n_1, the FSM will be stuck in a state and SOE will not be reached.

A checking mechanism based on flag values of different states is also employed. Whenever the FSM transitions to a state, the flag of that state is incremented by one. The SOE raises the output enable (OE) signal if the flag values of all the states are equal to one which corresponds to a valid input stream and thus a valid authentication event. For any other flag value, OE is not triggered. This provides an additional layer of security against an adversary who aims to brute force the device with random inputs. We also employ a counter ($Counter_2$) which keeps track of the number of times a wrong input is applied. If the FSM doesn't reach SOE or the flag values are not equal to 1, the counter is incremented and the FSM resets all the states and waits for the new deshuffled output. Every time the counter increments, the incremented value is stored in the device's NVM. If $Counter_2$ reaches a threshold value τ (τ can be set by the designer or operator depending on the application), the device locks out and needs to be unlocked explicitly by the server. This FSM-based locking mechanism is possible in PUF-IPA as the device has complete control of its challenges and responses. The mechansism enables the device to circumvent any temporary hardware fault, not necessarily an adversarial attack, which might occur during device operation. Figure 5 elaborates the FSM locking/unlocking mechanism by an example of the same 12-bit random number that was used in shuffling/deshuffling operation in Figure 4.

IV. Protocol Validation

A. Reliability

Arbiter PUFs (APUFs) are prone to environmental variations and thus generate noisy responses. The average noise level for a basic 64-stage Arbiter PUF is 4% in the temperature range of -40C to 85C [6]. The PUF-IPA relies on the underlying APUF in the device to generate reliable responses since no model or CRP is being stored in the server's memory. The Hamming distance-based thresholding for authentication [6] is not possible in PUF-IPA because of response string hashing at the PUF output. Hardware overhead renders usage of ECC infeasible [5]. It has been shown that a k-stage APUF follows a *linear additive delay model* [2] of the form:

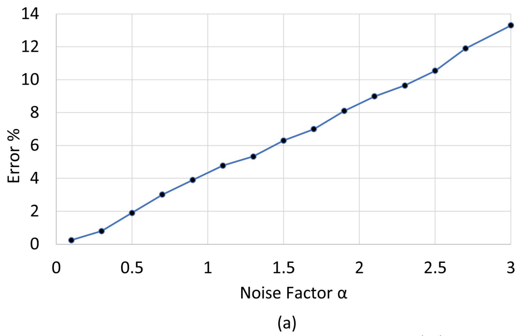
$$\Delta = \vec{w}^T \vec{\Phi}, \tag{2}$$

where $\vec{w}$ and $\vec{\Phi}$ are vectors of dimension $k+1$ in Eq. (2). The path delays of substages of an Arbiter PUF are encoded by the vector $\vec{w}$, whereas the *feature vector* $\vec{\Phi}$ is only dependent on the applied k-bit challenge vector $\vec{C}$. In order to find $\vec{w}$, runtime delay in stage q is computed and represented as $\delta_q^{0/1}$. In general, δ_q^0 represents the runtime delay of a stage q in crossed MUX configuration whereas δ_q^1 indicates the runtime delay of a stage q in straight MUX configuration. The vector $\vec{w}$ can represented as shown in [2] by:

$$\vec{w} = (\omega^1, \omega^2, \omega^3, ..., \omega^k, \omega^{k+1})^T, \tag{3}$$

where,

$$\omega^1 = \frac{\delta_1^0 - \delta_1^1}{2}, \quad \omega^q = \frac{\delta_{q-1}^0 + \delta_{q-1}^1 + \delta_q^0 - \delta_q^1}{2}, \tag{4}$$



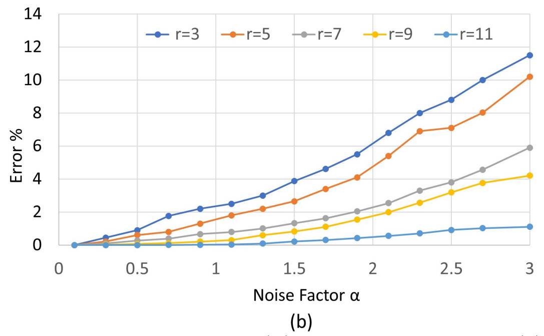


Fig. 6. (a) Response error (%) vs noise factor (α), (b) Response error (%) vs noise factor (α) against reliability factor (r).

$\forall\, q \in [2, ..., k]$, and $\omega^{k+1} = \frac{\delta_k^0 + \delta_k^1}{2}$. Since $\vec{\Phi}$ is dependent on the applied challenges, it is of the form,

$$\vec{\Phi}(\vec{C}) = (\vec{\Phi}^1(\vec{C}), \vec{\Phi}^2(\vec{C}), ..., \vec{\Phi}^k(\vec{C}), 1), \tag{5}$$

where $\vec{\Phi}^m(\vec{C}) = \prod_{q=m}^{k}(1-2b_q)$ for $m \in [1, ..., k]$. The output response bit i of the APUF is determined by the sign of the final delay difference Δ as,

$$i = \begin{cases} 1, & \Delta > 0 \\ 0, & \Delta < 0 \end{cases} \tag{6}$$

Instead of adding non-linearities in order to impede the adversaries' modeling capabilities, we implement a strong control logic for restricting access to the PUF circuit. To stabilize the response behavior, PUF-IPA introduces a *self-checking* mechanism in which a PUF's challenge is evaluated *r* times, where *r* is referred to as the *reliability factor*, and the output response bit is chosen based on the majority voting. During authentication phase, since the generation of challenge is controlled by the device itself, the device has the ability to perform *self-checking*. This checking is done by using the current odd-indexed RN. The PUF output response bit is chosen based on the majority voting between 0's and 1's. This is done for every response bit and the final response string is ensured to be stable.

We verify our proposed approach by testing the PUF model presented in Eq. (2) under noisy conditions. Since thermal noise can be approximated as Gaussian distribution, we can simulate the PUF's behavior by adding Gaussian noise in the path delay elements as,

$$\delta_x^{0'} = \delta_x^0 + N_x^0 \tag{7}$$

$$\delta_x^{1'} = \delta_x^1 + N_x^1, \tag{8}$$

where $N_x^0 = [n_1^0, n_2^0, ..., n_{k+1}^0]$ and $N_x^1 = [n_1^1, n_2^1, ..., n_{k+1}^1]$ are random variables selected from Gaussian Distribution $\mathcal{N}(\mu, \sigma^2_{noise})$ with mean μ and variance σ^2_{noise}. We set $\mu = 0$ and $\sigma^2_{noise} = \alpha \times 0.05$ where α is referred to as the *noise factor*, and is used to control the variance of noise (σ^2_{noise}).

Soft models of random PUF instances are generated and evaluated using three dimensional randomly generated challenge matrix of $50,000 \times 128 \times 128$ bits and corresponding $50,000 \times 128$ bits responses are stored for each PUF instance. These PUF instances are evaluated again by injecting noise (increasing α) into the path delays as shown in Eq. (7) and Eq. (8). We determine the response accuracy by calculating the hamming distance between the stored responses and new responses. Figure 6(a) shows the response accuracy against different values of α of a single PUF instance. Figure 6(b) shows the response accuracy for the same PUF instance but with *self-checking* enabled. It can be seen that the PUF response accuracy increases from 89% to $\approx$ 98% for the same value of α as we go from $r = 0$ to $r = 11$ thus ensuring reliability even in the absence of error correcting logic.

One of the greatest advantage that the PUF-IPA has over all the previously proposed protocols is that the device has the capability to perform *dynamic reliability checking* of the challenges. Since the device has explicit control of its own challenges, it can self-check the response accuracy without permission from the server at random time instances. Assuming that at time t_n, the odd-indexed RN is n_{d+1}, and the device evaluates the PUF with $r = 11$. If any of the response bits has an error percentage close to 40% (*minority* to *majority* response bit ratio of $\approx$ 4:7), then this RN is discarded and a new RN is generated which would be used for authentication. This ensures noise-free authentication events even during environmental variations. Thus, even though *self-checking* can improve response accuracy drastically, *dynamic checking* can prevent the usage of challenges that produce noisy responses because of environmental affects during authentication events.

B. Security Analysis and Area Overhead

1) Modeling Attacks: The PUF-IPA has no explicit storage of CRPs or model in the server's database, hence, no model can be generated even by acquiring the database. Furthermore, no PUF model is used by the server to estimate the responses during authentication. As a consequence, any adversary monitoring the server computations or channel will not get any CRP information for generating software models. Moreover, since the responses are hashed, no correlation between past and current messages (Figure 3) exists and every authentication event is unique. This impedes the adversaries' capability of generating an efficient software model of the underlying PUF.

2) Brute Force and Replay Attack: We consider the case where the adversary has complete access to the device's interface and can brute force query to obtain responses. Since the protocol restricts PUF's access, the adversary cannot apply random challenges and get responses. The shuffler and the FSM will halt the operation in case of a wrong input and thus no response will be generated by the device. The probability of guessing the correct input for the shuffler is $\frac{1}{64!} \approx 7 \times 10^{-90}$,

TABLE I
SECURITY AND AREA COMPARISON

Property	C-PUF[5]	Slender PUF[8]	Lockdown[6]	PUF-FSM[7]	Proposed PUF-IPA
ECC and helper data	✓	✗	✗	✗	✗
Scalable	✓	✓	✓	✓	✓
ML-based modeling attacks	✓	✗	✗	✗	✗
Privacy preserving	✗	✗	✗	✗	✓
Protection against server breach	✗	✗	✗	✗	✓
Dynamic self-checking	✗	✗	✗	✗	✓
Area (GE)	2.1k	NA	1.1k	1.1k	1.0k

thus making it extremely unlikely to be predicted correctly. The adversary can attempt replay attacks by using any previously sent input. Since $Counter_0$ generates a new RN after every valid authentication event and the shuffler as well as the FSM adapts to the new RN, any previously used *valid* RN will be treated as a wrong input which will subsequently lock out the device after specific number of attempts determined by $Counter_2$ and τ (Section III-D2).

Table I depicts that the PUF-IPA not only provides all the security features offered by previous approaches but also preserves the device privacy by not storing any model or CRP in the server. Unlike previous approaches, PUF-IPA also does not require any secure memory in the server or communication link implying that even in the case of a server breach, the security of the devices remains intact.

3) Implementation and Area Overhead: Implementation of PUF in FPGAs is a somewhat challenging task because of the symmetric routing requirements. A slight non symmetry can cause bias in delay circuit of the APUF. Authers in [11] introduced the concept of implementing the APUF using programmable delay lines for removal of delay bias from PUF circuit. We use the same concept and implement the PUF in Xilinx ZC-706 development board having a Zynq-7000 SoC. We also employ reliable response selection mechanism provided by [7]. By employing reliable responses, the error rate is significantly reduced. We compare the area overhead of PUF-IPA against various approaches in Table I. The gate equivalent (GE) of PUF-IPA is slightly less compared to the recently proposed PUF-FSM [7] but with many additional advantages as shown in Table I. In PUF-IPA, the server only needs to send 128-bit message to the device whereas in PUF-FSM [7], the server needs to send $160 \times 64 = 10240$ bits, which is a huge communication overhead. The hashing scheme employed by PUF-IPA is based on SPONGENT block cipher (as in [7]), which produces a 128-bit output at a cost of 737 GE. The FSM is realized using $\approx$30 GE. The total NVM required by PUF-IPA is 140 bits (i.e., 128+7+5 bits). Here 128 bits represent $Counter_0$ value, and 7 and 5 bits represent $Counter_1$ and $Counter_2$, respectively. Another advantage of PUF-IPA is that the server does not need to have a dedicated secure storage for storing CRPs or the modeling parameters of the devices. This reduces the cost associated with maintaining a *secure* storage as well as the communication links while not compromising the security.

V. CONCLUSION

In this paper, we have proposed PUF-IPA, a strictly controlled PUF-based IoT device authentication protocol, which (i) closes the open interface between the input and the PUF by implementing a strong controlled logic that denies the PUF access to adversaries, (ii) preserves the identity of the devices by removing any and all models or explicit CRPs from the server's database and the communication links, and (iii) is extremely lightweight and scalable for IoT based deployments. We have demonstrated the superiority of PUF-IPA in terms of security, reliability, and preservation of device privacy over previously proposed PUF-based authentication protocols.

REFERENCES

[1] 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*, pages 148–160, New York, NY, USA, 2002. ACM.

[2] 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. ACM, 2010.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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. IEEE, 2012.

[9] 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.

[10] 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. IEEE, 2015.

[11] 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'11, pages 17–32, Berlin, Heidelberg, 2011. Springer-Verlag.