Authentica: A Secure Authentication Mechanism using a Software-defined Unclonable Function

1st Ripon Patgiri
Dept. of Computer Science & Engineering
National Institute of Technology Silchar
Assam, India
ripon@cse.nits.ac.in

2nd Laiphrakpam Dolendro Singh
Dept. of Computer Science & Engineering
National Institute of Technology Silchar
Assam, India
ldsingh@cse.nits.ac.in

***Abstract*—Password-based authentication is an extensively used method to authenticate users. It uses cryptography to communicate the authentication process. On the contrary, the physically unclonable function (PUF)-based authentication mechanism is also gaining popularity rapidly due to its usability in IoT devices. It is a lightweight authentication mechanism that does not use cryptography protocol. PUF-based authentication mechanisms cannot authenticate users. To overcome the drawback of PUF, we introduce a software-defined unclonable function (SUF, for short). Contrary to the PUF, the SUF is used to authenticate users, not devices. We use SUF to implement a lightweight password-based authentication mechanism termed Authentica. Authentica bridges the gap between the password-based and the PUF-based authentication mechanism. Authentica does not use cryptography for authentication. However, we establish challenge-response using cryptography during the registration phase, which is a one-time cost. Authentica addresses a) impersonation attacks, b) collision attacks, c) dictionary and rainbow table attacks, d) replay attacks, e) DDoS attacks, f) the domino effect issues, and g) the challenge-response database leakage issues.**

***Index Terms*—Physically unclonable function, Software-defined unclonable function, Password Hashing, Identity manager, Authentication, Security, Attacks.**

1. Introduction

The physically unclonable function (PUF) is emerging due to security viability in IoT devices [1]–[3]. The PUFs are designed based on a nanoscale disordered physical structure where copying the same structure is physically impossible even for its manufacturers [4]. It is a circuit embedded with a device, producing a unique output where no other devices can produce the same output for the same input. This advantage is exploited in the authentication mechanism of a device [5]–[10]. It is suitable for tiny device authentication. PUF-based authentication features authentication without cryptography; thus, it features a truly lightweight authentication protocol.

1.1. Motivation

PUF is useful in authenticating devices without encryption but cannot be applied to authenticate the users. The key drawbacks of PUF are: a) the PUF-based authentication mechanism cannot authenticate users, b) PUF is integrated with a physical device that costs money, c) the challenge-response (CRP) database is vulnerable to leakage, d) it cannot prevent the domino effect, and e) it cannot withstand DDoS attack. Similarly, the key drawback of the password-based authentication mechanism is the requirement of cryptography to authenticate users.

We introduce a software-defined unclonable function, SUF, to produce a unique output for a given challenge where no one can produce the same output using the same function. Similar to the PUF, a legitimate user can reproduce the response for the given challenge exclusively. For instance, the users $\mathbb{A}$ and $\mathbb{B}$ produce different responses for the same challenge as defined in Definition 1. The $\mathscr{R}_{\mathbb{A}} \neq \mathscr{R}_{\mathbb{B}}$ for the same challenge $\mathscr{C}$. The SUF is used to implement lightweight authentication, Authentica, and to overcome the drawback of PUF and password-based authentication mechanisms. Authentica uses two secret words: a password and a secret context. These secrets are used to thwart diverse attacks. However, we can use a single secret, but the adversary can break it using a guessing attack because guessing attacks are becoming more powerful nowadays [11]–[15]. Therefore, we rely on two secrets because a user always sets an easy-to-remember password. Moreover, Authentica requires symmetric and asymmetric key cryptography for establishing the challenge-response pair, which is a one-time cost. The authentication process does not require any cryptography.

Definition 1. *Given two users $\mathbb{A}$ and $\mathbb{B}$ and a common challenge $\mathscr{C}$. The user $\mathbb{A}$ produces $\mathscr{R}_{\mathbb{A}} \leftarrow \mathrm{SUF}(\mathscr{C})$ and the user $\mathbb{B}$ produces $\mathscr{R}_{\mathbb{B}} \leftarrow \mathrm{SUF}(\mathscr{C})$ where $\mathscr{R}_{\mathbb{A}} \neq \mathscr{R}_{\mathbb{B}}$ if $\mathbb{A} \neq \mathbb{B}$, then the function is called a software-defined unclonable function (SUF).*

Definition 2. *Let ω_1 and ω_2 be the two strings to shuffle pseudo-randomly. Let ζ be the context. The shuffling process is defined over $(\omega_1, \omega_2, \zeta)$ where the two strings ω_1 and ω_2*

are shuffled using the context ζ. The shuffling process is significantly influenced by the context ζ, but the context does not present in the shuffled strings. We denote this process as $\mathcal{SW} \xleftarrow{\zeta} \text{SHUFFLE}(\omega_1, \omega_2)$.

1.2. Our proposed method

Our key objective is to build an authentication protocol without encryption, i.e., a lightweight authentication protocol without loss of security. Therefore, we utilize SUF to build a lightweight authentication protocol named Authentica. Authentica comprises a challenge-response method for authenticating the users using SUF. We first demonstrate a software-defined unclonable function. The SUF is constructed based on shuffling two words using context as defined in Definition 2. For illustration, let the user password be $\mathscr{P}$, and the domain name be $\mathscr{D}$. We convert the password and domain name into hash values as

$$\begin{aligned} \mathcal{H}_{\mathscr{P}} &\leftarrow \text{HASH512}(\mathscr{P}) \\ \mathcal{H}_{\mathscr{D}} &\leftarrow \text{HASH512}(\mathscr{D}) \end{aligned} \tag{1}$$

We shuffle the hash value using a secret context ζ as

$$\mathcal{SW} \xleftarrow{\zeta} \text{SHUFFLE}(\mathcal{H}_{\mathscr{P}}, \mathcal{H}_{\mathscr{D}}) \tag{2}$$

The $\mathcal{SW}$ is converted into a hash value as

$$\mathscr{R} \leftarrow \text{HASH512}(\mathcal{SW}) \tag{3}$$

The response, $\mathscr{R}$, can be exclusively produced/reproduced by the legitimate user. We term the above equations as SUF. The Authentica adapts the above-described SUF for authentication. Authentica has two phases: registration and authentication phase. The registration phase requires cryptography to establish the challenge-response of a user. We securely establish the challenge-response pair of the client. The server encrypts the user's response to the given challenge. We use the server's PUF function to generate the encryption key as

$$\begin{aligned} \mathscr{K} &\leftarrow \text{PUF}(ServerKey) \\ \mathscr{K} &\leftarrow \text{HASH512}(\mathscr{K}) \\ \mathscr{K} &\leftarrow \text{ARGON2I}(\mathscr{K}) \end{aligned} \tag{4}$$

where the $ServerKey$ is the server's secret key and the ARGON2I( ) is a memory-hard hash function to derive an encryption/decryption key. The secret key $\mathscr{K}$ is used to encrypt the responses. The $\mathscr{K}$ is a single key to encrypt the entire response database; therefore, it is fatal for the challenge-response (CRP) database. We develop a multi-key encryption technique to encrypt the responses. Each response is encrypted with a different secret key so the adversary cannot accidentally get authentication. Moreover, it features a leaked-proof CRP database.

The authentication phase relies on a captcha, and the rest of the communications are unencrypted. Thus, it achieves lightweight authentication without loss of security. The captcha is used to prove whether the prover is a human or not. It prevents malicious requests from bots because DDoS attacks produce a few million such requests that can down the service. Therefore, it is a vital part of Authentica. Also, we use lightweight data structures to prevent diverse attacks. For authentication, the user extracts two challenges which are embedded with the login page. It is fixed for a domain and common for all users of that domain. Also, the login page contains a timestamp of the server. A client (prover) downloads the login page containing the challenges and the server's timestamp. The client retrieves the local timestamp. The client produces the two responses by inputting the challenges to the SUF. The two responses are hashed using timestamps by the SHA512 algorithm. The hashed responses are sent to the server (verifier) without encryption. The server receives these hashed responses. The server also hashes the stored responses using the timestamp to produce the hash values. The server compares the produced and received hashed responses for authentication. If the comparison is successful, the prover wins. Otherwise, it fails.

1.3. Our results

The SUF can uniquely produce a response against a challenge where the same response cannot be produced by other users using the same function and the same challenge. Authentica adapts SUF to implement a lightweight authentication protocol without loss of security. Authentica does not use a cryptography protocol to authenticate the users, and it can prevent diverse attacks, which are highlighted in the next subsections.

1.3.1. Impersonation attacks. In Authentica, the prover (user) proves its authenticity to the verifier (the server or identity manager). The adversary wants to impersonate the legitimate prover to the verifier. If the adversary proves its authenticity to the verifier, the adversary wins; otherwise, the adversary loses. The adversary must correctly reproduce the two responses to win the game. It requires two secrets to reproduce the responses correctly. Therefore, the adversary either guesses the two secrets within three trials or performs a collision by which it correctly produces the response. Since we use two secrets, thus, guessing becomes harder. Moreover, the collision is also harder due to the two responses using SHA512.

1.3.2. Collision attacks. There is a possibility of collision in the hash algorithm due to limited spaces of the hash value. Using a collision attack, the adversary can impersonate. However, SHA512 is a collision-resistant hash algorithm. Therefore, it is hard to perform a collision attack on a single response, whereas Authentica uses two responses to avoid accidental collision. Moreover, the adversary can impersonate if the adversary's hash value for the response can be constructed by the server using the timestamps and response of the given client. Thus, performing a collision attack is computationally hard.

1.3.3. Dictionary and rainbow table attacks. The dictionary and rainbow table attacks are applied to the leaked hash

values. Usually, the dictionary and rainbow table attacks are used against salted passwords and/or hashed passwords. In Authentica, the challenge-response (CRP) database is encrypted using a secret key generated by the server's PUF. Therefore, dictionary or rainbow table attacks do not apply. However, the adversary can perform a dictionary or rainbow table attack on the hashed value of the responses, which are unencrypted. The adversary needs to produce two hash values for the responses using the given timestamp by the server. For each communication, the hash value for the response changes because the server's timestamp changes. Therefore, the dictionary attack and rainbow table attack do not apply.

1.3.4. Replay attacks. The most common attack in challenge-response-based authentication is a replay attack since the communication is carried out without encryption. The adversary performs a replay attack using the previous hash values for the response. Notably, the server looks for the hash values from the client with the server's given timestamp, where the adversary measurably fails to produce the hash values for the response using the server's generated timestamp. Thus, an adversary cannot replay the previous communication for the authentication.

1.3.5. Domino effect. The domino effect occurs due to password reuse in state-of-the-art authentication systems. Therefore, password reuse is strongly discouraged in the state-of-the-art authentication mechanism [16]. On the contrary, our proposed system encourages users to reuse the password without any risks. Using the same password, the user is always able to create different hash values for the different domains. We use the domain name to facilitate password reuse, which is similar to Ross *et al.* [17]. Authentica considers two secrets: a password and a secret context. We use a secret context in the shuffling process, whereas the shuffled word does not contain the characters of the context. It influences the shuffling process, and it is computationally hard to reproduce the shuffled word without knowing the adversary's password and context. The challenge, domain word, and password are shuffled using a secret context to produce a response. The shuffled word is converted into a hash value using SHA512. Therefore, the response of a domain cannot be the same as the response of another domain with the same password and secret context. Therefore, there is no domino effect in Authentica.

1.3.6. DDoS attack. DDoS attackers do not aim to impersonate but slow down the services. To tackle such kinds of attacks, Authentica uses a captcha challenge. Moreover, Authentica uses Bloom Filter to filter out unwanted authentication requests. Due to the false positives of Bloom Filter, the malicious requests can reach the authentication table and failure table. The authentication table and failure table filter out the remaining malicious requests. We know that the Bloom Filter is a lightweight and fast data structure that can filter millions of requests per second. Moreover, the authentication table and failure table are hashtable which are also fast data structures. Therefore, Authentica can handle DDoS efficiently and effectively.

1.3.7. Database stealing issue. We assume that the adversary can easily steal the CRP database. Moreover, the adversary can also steal the secret $ServerKey$. To prevent such kind of accident, we use PUF to generate a secret key as $\mathscr{K} = PUF(ServerKey)$. The $\mathscr{K}$ is updated using SHA512 and Argon2i hash function. The $\mathscr{K}$ is used to encrypt/decrypt the responses. We use secret salt to generate a secret key where an adversary cannot generate the secret key without accessing the server physically. Moreover, each response is encrypted with a different key, and it is computationally hard to decrypt the response for the adversary. Therefore, the adversary cannot decrypt the response even if the CRP database is leaked.

2. Software-defined Unclonable Function

The PUF produces a unique response for a given challenge; no one can produce the same response [1]–[3]. The same response can be reproduced consistently for a given challenge by the legitimate PUF. The PUF-based authentication mechanism authenticates the devices but not the users. Therefore, we adopt a shuffling function to implement the software-defined unclonable function to authenticate users. The SUF also guarantees that no one can produce the same response for a given challenge. Moreover, the same response can be reproduced consistently for a given challenge by the legitimate user.

The shuffling process shuffles two strings using a secret context. Let ζ be a secret context, $\mathscr{P}$ is a password, and $\mathscr{C}$ be the challenge. The challenge is known to all, including adversaries. The secret context and password are known to the client, $\mathbb{C}$, exclusively. The client converts the password into a hash value by hashing using SHA512 as given in Equation (5).

$$\mathcal{H}_{\mathcal{P}} \leftarrow \textsc{Hash512}(\mathscr{P}) \tag{5}$$

Also, the client converts the given challenge into a hash value as

$$\mathcal{H}_{\mathscr{C}} \leftarrow \textsc{Hash512}(\mathscr{C}) \tag{6}$$

The $\mathcal{H}_{\mathcal{P}}$ is shuffled with another hash value $\mathcal{H}_{\mathscr{C}}$ with a context of ζ and produces a shuffled word $\mathcal{SW}$, as shown in Equation (7).

$$\mathcal{SW} \xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathscr{P}}, \mathcal{H}_{\mathscr{C}}) \tag{7}$$

Notably, the $\mathcal{SW}$ does not contain any character of the ζ. The shuffled word is hashed to produce a response $\mathscr{R}$ as shown in Equation (8).

$$\mathscr{R} \leftarrow \textsc{Hash512}(\mathcal{SW}) \tag{8}$$

The $\mathscr{R}$ is the unique value that the client, $\mathbb{C}$, can produce exclusively. Also, the $\mathbb{C}$ can consistently reproduce the same response, $\mathscr{R}$, for the same challenge $\mathscr{C}$. Theoretically, no one can reproduce the same response using the same challenge

$\mathscr{C}$ but $\mathcal{H}_{\mathscr{C}} = \mathcal{H}'_{\mathscr{C}}$ where $\mathscr{C} \neq \mathscr{C}'$. It occurs due to the collision in hashing.

Let ω_1 and ω_2 be the two words known to all, and the context is a secret word. The client converts the words into hash values as

$$\begin{aligned} \mathcal{H}_{\omega_1} &\leftarrow \textsc{Hash512}(\omega_1) \\ \mathcal{H}_{\omega_2} &\leftarrow \textsc{Hash512}(\omega_2) \end{aligned} \tag{9}$$

In this case, the adversary can produce the correct hash value since we assumed the words were public. These two hash values are shuffled using a secret context as

$$\mathcal{SW} \xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\omega_1}, \mathcal{H}_{\omega_2}) \tag{10}$$

The adversary cannot reproduce the $\mathcal{SW}$ since the adversary does not know about the context. We use SHA512; the hash values are 512-bit long and 128 characters in hexadecimal representation. Therefore, the length of the two strings is 128 in the shuffling process. The probability of selecting an index from 128 indexes at ω_1 to place a character from ω_2 into the ω_1 is $\frac{1}{128}$. Now, the length of ω_1 is increased by one, and ω_2 is decreased by one. Therefore, selecting an index from 129 indexes of ω_1 is $\frac{1}{129}$. This selection probability is an independent event. Therefore, the probability of placing two characters from ω_2 in ω_1 is $\frac{1}{128\times129}$. Thus, the total probability of correctly reproducing the shuffled word without knowing the secret context is

$$\frac{1}{128 \times 129 \times 130 \times \ldots \times 256} \approx 0 \tag{11}$$

The reproduction of a shuffled word without knowing the context is computationally hard.

The adversary can attack the secret context or password instead of reproducing the shuffled word. The probability of reconstruction of the context using the brute force method is $\frac{1}{\binom{94}{10}} = 1.1060409160874147e-13$ where ten characters to choose from 94 available characters for the context (excluding white space), which is easier than reproducing the shuffled word. But it is still computationally hard to reproduce the secret context if the secret context is a high entropy word. Moreover, the password is also a secret word. Therefore, it is difficult to perform a brute-force attack. However, a guessing attack is more powerful than a brute-force attack. Therefore, we assume the adversary can reproduce the secret context using a guessing attack. The adversary guesses the password and secret context using a guessing attack. Then, the adversary can evade the security of SUF. However, performing a guessing attack for the two secrets is difficult. Moreover, the guessing attack's success depends on the entropy of the secrets.

2.1. Implementation of SUF

SUF is implemented based on a shuffling process of two strings using a context. The shuffling process uses a non-cryptography string hash function, and we use Murmur2 for shuffling the strings with respect to the context. The shuffling process uses a context to randomize (pseudo-randomized) the shuffling process. Algorithm 1 alters the integer value τ times because the Murmur2 is an insecure string hash function. Algorithm 3 implements the shuffling process by selecting one character to insert into another word based on an integer value (θ). Initially, the θ is set to a 32-bit integer value. Algorithm 3 selects a character one by one from ω_2 to insert into ω_1. The selected character from ω_2 is inserted into ω_1 in a given index using Algorithm 2 where the index is generated by computing integer value

Algorithm 1 Computing integer value (θ) for utilization in the hash function.

```
procedure GETSEEDVALUE(ω, 𝒮, θ, τ)
    for i : 1 to τ do
        θ = PRIMARYHASH(ω, 𝒮, θ)
    end for
    return θ
end procedure
```

Algorithm 2 Inserting a character ch into a particular position of ω.

```
procedure INSERTCHARAT(ω, 𝒮, ch, pos)
    for i : 𝒮 to pos do
        ω[i] = ω[i − 1]
    end for
    ω[i] = ch
end procedure
```

Algorithm 3 Shuffling algorithm.

```
procedure SHUFFLE(ω1, ω2, ctx, θ, buff)
    𝒮1 = LENGTH(ω1)
    𝒮2 = LENGTH(ω2)
    𝒮3 = LENGTH(ctx)
    COPY(buff, ω1)
    τ = 16, δ = 1783, μ = 16
    θ = GETSEEDVALUE(ctx, 𝒮3, θ, τ)
    τ = θ%δ + μ
    θ = GETSEEDVALUE(ω2, 𝒮2, θ, τ)
    τ = θ%δ + μ
    θ = GETSEEDVALUE(ω1, 𝒮1, θ, τ)
    τ = θ%δ + μ
    while ω1[i] ≠ Null do
        θ = GETSEEDVALUE(ω2, 𝒮2, θ, τ)
        τ = θ%δ + μ
        θ = GETSEEDVALUE(ω1, 𝒮1, θ, τ)
        τ = θ%δ + μ
        θ = GETSEEDVALUE(ctx, 𝒮3, θ, τ)
        τ = θ%δ + μ
        ρ = θ%𝒮1
        INSERTCHARAT(buff, 𝒮1, ω2[i], ρ)
        𝒮1 = 𝒮1 + 1
        i = i + 1
    end while
end procedure
```

using Algorithm 1. Therefore, the selection process of an index of ω_1 depends on the two strings (ω_1 and ω_2) and the context. The context influences the selection process of the index. We can further improve the index selection process by pseudo-randomly selecting a character from ω_2 too. We can randomly (pseudo) select a character from ω_2 and insert the selected character into the ω_1. The probability of selecting the first character from ω_2 is $\frac{1}{128}$ and probability of selecting an index to insert at ω_1 is $\frac{1}{128}$. The probability of selecting the second character from ω_1 is $\frac{1}{127}$ and selecting an index to insert at ω_1 is $\frac{1}{129}$ and so on. Thus, it improves the probability as

$$\frac{1}{128 \times 128 \times 129 \times 127 \times 130 \times 126 \times \ldots \times 256 \times 1} \approx 0 \tag{12}$$

Equation (12) is smaller than Equation (11); hence, Equation (12) is stronger than Equation (11).

3. Authentica

Authentica has two phases: the registration phase and the authentication phase. We first discuss the registration phase, and then we discuss the authentication phase.

3.1. Registration

Unlike PUF, the SUF function does not require a physical device to be integrated with. Moreover, SUF cannot be applied to authenticate a physical device. The SUF implements a challenge-response method similar to the PUF. The challenge-response of a user is stored in a server's database. The registration phase establishes the challenge-response of the user, which cannot malfunction. Otherwise, Authentica fails. Therefore, Authentica uses existing cryptography (assumed SSL for encryption) to establish the challenge-response of the users in the server's database, which is a one-time cost. Let the challenges be $(\mathscr{C}_1, \mathscr{C}_2)$, which is public and common for all other users of a given domain. The two challenges are embedded with the registration and login page. The challenges are different for different identity managers, and the challenges cannot be the same for two different domain names. Let the responses be $(\mathscr{R}_1, \mathscr{R}_2)$ for the challenges $(\mathscr{C}_1, \mathscr{C}_2)$. The pair ($\mathscr{R}_1$, $\mathscr{R}_2$) need to be stored in the server's database against the user. The challenges are the same for all the users for a given domain and remain fixed throughout the lifetime of the identity manager. Still, the responses are different for each user because of the SUF. To store the pair ($\mathscr{R}_1$, $\mathscr{R}_2$) against the user, we need encryption and decryption for communication in the registration process, which is one time cost. We rely on existing encryption technology for registration to securely transmit the responses from the client to the server and securely establish the response into the server's CRP database.

Table 1 demonstrates the registration process. In the registration phase, the client downloads the registration pages where the challenges are embedded with the registration pages since the challenges are public. The client converts the challenge and domain word into hash values as

$$\begin{aligned} \mathcal{H}_{\mathscr{C}_1} &\leftarrow \textsc{Hash512}(\mathscr{C}_1) \\ \mathcal{H}_{\mathscr{C}_2} &\leftarrow \textsc{Hash512}(\mathscr{C}_2) \\ \mathcal{H}_{\mathscr{D}} &\leftarrow \textsc{Hash512}(\mathscr{D}) \end{aligned} \tag{13}$$

The domain word and challenges are public words that are known to all, including the adversary. Anyone can reconstruct the hash values of these three words. The client shuffles these two hash values using a secret context ζ as given below-

$$\begin{aligned} \mathcal{SD}_1 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathscr{C}_1}, \mathcal{H}_{\mathscr{D}}) \\ \mathcal{SD}_2 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathscr{C}_2}, \mathcal{H}_{\mathscr{D}}) \end{aligned} \tag{14}$$

These hash values are shuffled using a secret context where reproducing the shuffled word is computationally hard without knowing the secret context. The $\mathcal{SD}_1$ and $\mathcal{SD}_2$ can be reconstructed using the secret context consistently for the given challenge by the same user. The shuffled words are converted into hash values as

$$\begin{aligned} \mathcal{H}_{\mathcal{SD}_1} &\leftarrow \textsc{Hash512}(\mathcal{SD}_1) \\ \mathcal{H}_{\mathcal{SD}_2} &\leftarrow \textsc{Hash512}(\mathcal{SD}_2) \end{aligned} \tag{15}$$

The client converts the password into a hash value as

$$\mathcal{H}_{\mathscr{P}} \leftarrow \textsc{Hash512}(\mathscr{P}) \tag{16}$$

This process conceals the password from revealing it to the world. Again, the client shuffles two hash values as

$$\begin{aligned} \mathcal{SW}_1 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathcal{SD}_1}, \mathcal{H}_{\mathscr{P}}) \\ \mathcal{SW}_2 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathcal{SD}_2}, \mathcal{H}_{\mathscr{P}}) \end{aligned} \tag{17}$$

It requires the correct hash value of the password and a correct context to reconstruct $\mathcal{SW}_1$ and $\mathcal{SW}_2$. The new shuffled word $\mathcal{SW}_1$ and $\mathcal{SW}_2$ are converted into a hash value as given below-

$$\begin{aligned} \mathscr{R}_1 &\leftarrow \textsc{Hash512}(\mathcal{SW}_1) \\ \mathscr{R}_2 &\leftarrow \textsc{Hash512}(\mathcal{SW}_2) \end{aligned} \tag{18}$$

The client sends the response $(\mathscr{R}_1, \mathscr{R}_2)$ using the POST method to the server along with all the necessary information of the user; for instance, email ID, phone numbers, etc. to be inserted into the identity manager's database. We assumed SSL for the POST method.

The identity manager (the server) receives the user's data and responses. The responses are the crucial data to be secured. We utilize the concept of the PUF to generate the encryption/decryption key. The identity manager generates the key using its secret word, $ServerKey$ as

$$\begin{aligned} \mathscr{K} &\leftarrow PUF(ServerKey) \\ \mathscr{K} &\leftarrow \textsc{Hash512}(\mathscr{K}) \end{aligned} \tag{19}$$

The $ServerKey$ is not secure, and it can leak at any time. Therefore, we cannot rely on the identity manager's secret word. Hence, we use the PUF function to generate a key. The $\mathscr{K}$ cannot be generated without physically accessing

TABLE 1. REGISTRATION PROCESS OF A CLIENT IN A SERVER.

Client, $\mathbb{C}$	**Server,** $\mathbb{S}$
1. The client downloads the registration page and extracts the challenge embedded with the page. 2. The client produces the response pair $(\mathscr{R}_1, \mathscr{R}_2)$ using SUF. 3. The client sends the response pair $(\mathscr{R}_1, \mathscr{R}_2)$ along with user details to the server for registration using the POST method. Also, the client clicks on the captcha challenge.	
	1. The server receives the response pair $(\mathscr{R}_1, \mathscr{R}_2)$ along with the user details. 2. The server computes the secret key $\mathscr{K}$ to encrypt the response pair $(\mathscr{R}_1, \mathscr{R}_2)$ and stores the encrypted response pair into the database. 3. The server stores the user's details in its database. 4. The server creates a user ID $\mathcal{ID}$, and insert into the Bloom Filter. 5. Initiate the failure table $\mathcal{FT}$ and authentication table $(\mathcal{HT})$. 6. Registration is successful.
1. The client deletes all associated data upon successful registration.	

the server. Thus, it secures even if the $ServerKey$ is public. Moreover, we update the key using Argon2i memory-hard hash function as

$$\mathscr{K} \leftarrow \text{ARGON2I}(\mathscr{K}, \mathscr{S}) \tag{20}$$

where $\mathscr{S}$ is a salt. The salt can be generated as

$$\mathscr{S} \leftarrow \text{HASH512}(ServerKey) \tag{21}$$

The single-key encryption is fatal for the identity manager because the adversary can reveal the entire responses of all users if the $\mathscr{K}$ is comprised. To solve this issue, we use a set of $ServerKey$ for n users as defined below-

$$\begin{aligned} ServerKey =& \{(\mathcal{SK}_{11}, \mathcal{SK}_{12}), (\mathcal{SK}_{21}, \mathcal{SK}_{22}), \\ & (\mathcal{SK}_{31}, \mathcal{SK}_{32}) \ldots, (\mathcal{SK}_{n1}, \mathcal{SK}_{n2})\} \end{aligned} \tag{22}$$

The $ServerKey$ is a set of semi-private words. These words are stored in the server without encryption. The server can pick i^{th} pair of the server's key for the i^{th} user and computes the secret key for encryption as

$$\begin{aligned} \mathscr{K}_{i1} &\leftarrow \text{PUF}(ServerKey_{i1}) \\ \mathscr{K}_{i2} &\leftarrow \text{PUF}(ServerKey_{i2}) \end{aligned} \tag{23}$$

where $ServerKey_{i1} \neq ServerKey_{i2}$, and therefore, $\mathscr{K}_{i1} \neq \mathscr{K}_{i2}$. Moreover, the $\mathscr{K}_{i1} \neq \mathscr{K}_{j1}$ and $\mathscr{K}_{i2} \neq \mathscr{K}_{j2}$ which implies the $ServerKey$ is a set of unique words. The keys are updated as

$$\begin{aligned} \mathscr{K}_{i1} &\leftarrow \text{HASH512}(\mathscr{K}_{i1}) \\ \mathscr{K}_{i2} &\leftarrow \text{HASH512}(\mathscr{K}_{i2}) \end{aligned} \tag{24}$$

Again the keys are updated as

$$\begin{aligned} \mathscr{K}_{i1} &\leftarrow \text{ARGON2I}(\mathscr{K}_{i1}, \mathscr{S}_{i1}) \\ \mathscr{K}_{i2} &\leftarrow \text{ARGON2I}(\mathscr{K}_{i2}, \mathscr{S}_{i2}) \end{aligned} \tag{25}$$

where the salt can be generated as

$$\begin{aligned} \mathcal{S} =& \{(\mathcal{S}_{11}, \mathcal{S}_{12}), (\mathcal{S}_{21}, \mathcal{S}_{22}), (\mathcal{S}_{31}, \mathcal{S}_{32}), \\ & \ldots, (\mathcal{S}_{n1}, \mathcal{S}_{n2})\} \\ \mathscr{S}_{i1} &\leftarrow \text{PUF}(\mathcal{S}_{i1}) \\ \mathscr{S}_{i2} &\leftarrow \text{PUF}(\mathcal{S}_{i2}) \\ \mathscr{S}_{i1} &\leftarrow \text{HASH512}(\mathscr{S}_{i1}) \\ \mathscr{S}_{i2} &\leftarrow \text{HASH512}(\mathscr{S}_{i2}) \end{aligned} \tag{26}$$

The $\mathcal{S}$ is a set of pair of words to generate salt values where $\mathcal{S}_{i1} \neq \mathcal{S}_{i2}$, $\mathcal{S}_{i1} \neq \mathcal{S}_{j1}$, and $\mathcal{S}_{i2} \neq \mathcal{S}_{j2}$. The $\mathcal{S}$ is also semi-private or semi-secret, stored in the server without encryption. These are used to generate salt using PUF for the Argon2i hash function. Therefore, no one can generate the $\mathscr{K}$ and $\mathscr{S}$ without physical access to the server. The responses are encrypted as

$$\begin{aligned} \mathscr{E}_{\mathscr{R}_1} &\leftarrow \text{ENC}(\mathscr{R}_1, \mathscr{K}_{i1}) \\ \mathscr{E}_{\mathscr{R}_2} &\leftarrow \text{ENC}(\mathscr{R}_2, \mathscr{K}_{i2}) \end{aligned} \tag{27}$$

The ciphertext of responses $(\mathscr{E}_{\mathscr{R}_1}, \mathscr{E}_{\mathscr{R}_2})$ are inserted into the CRP database against the user. The ENC( ) is a symmetric AES or ECC encryption.

3.2. Authentication

A client, $\mathbb{C}$, wishes to acquire an authentication from the server, $\mathbb{S}$. The client needs to prove as human by using a captcha challenge. It is required to defend against malicious requests from bots. The client downloads the login page for authentication and clicks on the captcha. The challenges are embedded with the login page. The legitimate user can reproduce the responses from the challenges at any given

TABLE 2. AUTHENTICATION PROCESS USING CHALLENGE-RESPONSE USING SUF.

Client, $\mathbb{C}$	**Server,** $\mathbb{S}$
1. Client wishes to get authentication from server $\mathbb{S}$. 2. Client proves as human by captcha. 3. Client downloads the login page where the challenges ($\mathscr{C}_1$, $\mathscr{C}_2$) are embedded. Moreover, the client downloads the timestamp ($\mathcal{TN}$) from the page. 4. The client generates a timestamp $\mathcal{TR}$. 5. The client produces hash values of the responses, $\mathcal{H}_{\mathscr{R}_1}$ and $\mathcal{H}_{\mathscr{R}_2}$, using $\mathcal{TN}$ and $\mathcal{TR}$. 6. The client sends $\mathcal{H}_{\mathscr{R}_1}$ and $\mathcal{H}_{\mathscr{R}_2}$, $\mathcal{TN}$, and $\mathcal{TR}$ to the server without encryption.	
	1. The server receives $\mathcal{H}_{\mathscr{R}_1}$ and $\mathcal{H}_{\mathscr{R}_2}$, $\mathcal{TN}$, and $\mathcal{TR}$. 2. The server checks the user ID in Bloom Filter, authentication table, and failure table. 3. Server computes $\mathscr{K}$ to decrypt the stored responses of the client. 4. The server constructs the hash values of the decrypted response using the $\mathcal{TN}$ and $\mathcal{TR}$. 5. The constructed hash values and received hash values are compared for authentication. 6. If the comparison is successful, the user is authenticated. Otherwise, the authentication fails.
1. The client receives an authentication notification.	

time. The server sends the challenges with a random nonce for each login request. This random nonce is generated using a true random number generator and should not be repeated. Moreover, the random nonce is altered if the login page is refreshed. However, the repetition of generated random nonce depends on the true random number generator. Let the random nonce be $\mathcal{N}$, and the server will generate a different random nonce in each login page request. The random nonce is merged with the current timestamp of the server and sent to the client as

$$\begin{aligned} \mathcal{N} &\leftarrow \mathcal{G}() \\ \mathcal{T}_{\mathbb{S}} &\leftarrow \text{GETCURRENTTIMESTAMP}(\ ) \\ \mathcal{TN} &\leftarrow \eta \,||\, \mathcal{T}_{\mathbb{S}} \,||\, \mathcal{N} \end{aligned} \tag{28}$$

where $\mathcal{G}()$ is a generator for a random nonce and $\eta = \{1, 2, 3, \ldots\}$ in increasing order. The server maintains a counter for each user, termed η. For instance, if $\eta = 1$, it will be $\eta = 2$ in the next authentication. Notably, the $\mathcal{TN}$ will be different for each client. The client extracts the challenge ($\mathscr{C}_1, \mathscr{C}_2$) and the random nonce, $\mathcal{TN}$, from the login page. The client reproduce the responses ($\mathscr{R}_1, \mathscr{R}_2$) using Equations (13)-(18). The authentication process is communicated to the server without encryption. Therefore, the responses cannot be directly sent to the server. To conceal the response, the client generates a random nonce as

$$\begin{aligned} \mathcal{R} &\leftarrow \mathcal{G}() \\ \mathcal{T}_{\mathbb{C}} &\leftarrow \text{GETCURRENTTIMESTAMP}(\ ) \\ \mathcal{TR} &\leftarrow \mathcal{T}_{\mathbb{C}} \,||\, \mathcal{R} \end{aligned} \tag{29}$$

The client converts the random nonce into hash values as

$$\begin{aligned} \mathcal{H}_{\mathcal{TN}} &\leftarrow \text{HASH512}(\mathcal{TN}) \\ \mathcal{H}_{\mathcal{TR}} &\leftarrow \text{HASH512}(\mathcal{TR}) \end{aligned} \tag{30}$$

The client computes hashing and produces a hash value for $\mathscr{R}_1$ as

$$\begin{aligned} \mathcal{H}_{\mathscr{C}_1} &\leftarrow \text{HASH}(\mathscr{R}_1 \,||\, \mathscr{C}_1) \\ \mathcal{H}_{\mathcal{TN}_1} &\leftarrow \text{HASH}(\mathcal{H}_{\mathscr{C}_1} \,||\, \mathcal{H}_{\mathcal{TN}}) \\ \mathcal{H}_{\mathcal{TR}_1} &\leftarrow \text{HASH}(\mathcal{H}_{\mathcal{TN}_1} \,||\, \mathcal{H}_{\mathcal{TR}}) \\ \mathcal{H}_{\mathscr{R}_1} &\leftarrow \text{HASH}(\mathcal{H}_{\mathcal{TR}_1} \,||\, \mathscr{R}_1) \end{aligned} \tag{31}$$

Again, the client computes hashing and produces a hash value for $\mathscr{R}_2$ as

$$\begin{aligned} \mathcal{H}_{\mathscr{C}_2} &\leftarrow \text{HASH}(\mathscr{R}_2 \,||\, \mathscr{C}_2) \\ \mathcal{H}_{\mathcal{TN}_2} &\leftarrow \text{HASH}(\mathcal{H}_{\mathscr{C}_2} \,||\, \mathcal{H}_{\mathcal{TN}}) \\ \mathcal{H}_{\mathcal{TR}_2} &\leftarrow \text{HASH}(\mathcal{H}_{\mathcal{TN}_2} \,||\, \mathcal{H}_{\mathcal{TR}}) \\ \mathcal{H}_{\mathscr{R}_2} &\leftarrow \text{HASH}(\mathcal{H}_{\mathcal{TR}_2} \,||\, \mathscr{R}_2) \end{aligned} \tag{32}$$

The client sends the computed responses ($\mathcal{H}_{\mathscr{R}_1}, \mathcal{H}_{\mathscr{R}_2}$) to the server along with the random nonce $\mathcal{TR}$ and $\mathcal{TN}$ without encryption. The server receives the computed hash values and the random nonce. The server checks the validity of the user ID in Bloom Filter as

$$\begin{aligned} &\textbf{if } \mathcal{ID} \in \mathscr{BF} \\ &\quad \textit{Proceed to the next step} \\ &\textbf{else} \\ &\quad \textit{Invalid user ID} \end{aligned} \tag{33}$$

This process ensures unnecessary processing of requests from invalid user IDs. The server also examines whether the user is already authenticated or not as

$$\begin{aligned}
&\textbf{if } \mathcal{ID} \in \mathcal{HT} \textit{ and } \mathcal{HT}.\mathcal{ID}.logout = true \\
&\quad \textit{Proceed to the next step} \\
&\textbf{else if } \mid \mathcal{HT}.\mathcal{ID}.\mathcal{T}_{auth} - \mathcal{T}_{current} \mid \geq \delta \\
&\quad \mathcal{HT}.\mathcal{ID}.logout = true \\
&\quad \textit{Proceed to the next step} \\
&\textbf{else} \\
&\quad \textit{Block the request}
\end{aligned} \tag{34}$$

This process ensures that it should not process already authenticated users. Furthermore, it examines the blocking of the user ID in the failure table as

$$\begin{aligned}
&\textbf{if } \mid \mathcal{FT}.\mathcal{ID}.\mathcal{T}_{blocking} - \mathcal{T}_{current} \mid \leq \mathcal{FT}.\mathcal{ID}.level \\
&\quad \textit{Block the request} \\
&\textbf{else} \\
&\quad \textit{Proceed to the next step}
\end{aligned} \tag{35}$$

Initially, the $\mathcal{FT}.\mathcal{ID}.\mathcal{T}_{blocking}$ and $\mathcal{FT}.\mathcal{ID}.level$ are set to zero. The above equation examines whether the user's blocking is expired. Moreover, the server checks the received values for freshness as

$$\begin{aligned}
&\textbf{if } \mathcal{H}^{recent}_{\mathscr{R}_1} = \mathcal{ID}.\mathcal{H}^{recent}_{\mathscr{R}_1} \textit{ and } \mathcal{H}^{recent}_{\mathscr{R}_2} = \mathcal{ID}.\mathcal{H}^{recent}_{\mathscr{R}_2} \\
&\quad \textit{Already authenticated.} \\
&\quad \textit{It's a replay attack.} \\
&\quad \textit{Block the request} \\
&\textbf{else if } \mathcal{TN} \leq \mathcal{TN}_{recent} \textit{ and } \mathcal{TR} \leq \mathcal{TR}_{recent} \\
&\quad \textit{It's a replay attack.} \\
&\quad \textit{Block the request.} \\
&\textbf{else} \\
&\quad \textit{Proceed to the next step}
\end{aligned} \tag{36}$$

where $\mathcal{TN}_{recent}$ and $\mathcal{TR}_{recent}$ are the timestamps of the previous authentication. Notably, these values can be empty at the first time of authentication. If the request is fresh, the server requires a key to decrypt the stored responses. The server reproduces the decryption keys $\mathscr{K}_{i1}$ and $\mathscr{K}_{i2}$ using Equations (22)-(26). The server decrypts the responses of the user as

$$\begin{aligned}
\mathscr{R}^{dec}_1 &\leftarrow \textsc{Dec}(\mathscr{E}_{\mathscr{R}_1}, \mathscr{K}_{i1}) \\
\mathscr{R}^{dec}_2 &\leftarrow \textsc{Dec}(\mathscr{E}_{\mathscr{R}_2}, \mathscr{K}_{i2})
\end{aligned} \tag{37}$$

The server computes the hash values for the response $\mathscr{R}_1$ as

$$\begin{aligned}
\mathcal{H}^{\mathbb{S}}_{\mathscr{C}_1} &\leftarrow \textsc{Hash}(\mathscr{R}^{dec}_1 \mid\mid \mathscr{C}_1) \\
\mathcal{H}^{\mathbb{S}}_{\mathcal{TN}_1} &\leftarrow \textsc{Hash}(\mathcal{H}^{\mathbb{S}}_{\mathscr{C}_1} \mid\mid \mathcal{H}_{\mathcal{TN}}) \\
\mathcal{H}^{\mathbb{S}}_{\mathcal{TR}_1} &\leftarrow \textsc{Hash}(\mathcal{H}^{\mathbb{S}}_{\mathcal{TN}_1} \mid\mid \mathcal{H}_{\mathcal{TR}}) \\
\mathcal{H}^{\mathbb{S}}_{\mathscr{R}_1} &\leftarrow \textsc{Hash}(\mathcal{H}^{\mathbb{S}}_{\mathcal{TR}_1} \mid\mid \mathscr{R}^{dec}_1)
\end{aligned} \tag{38}$$

Again, the server computes the hash values for the response $\mathscr{R}_2$ as

$$\begin{aligned}
\mathcal{H}^{\mathbb{S}}_{\mathscr{C}_2} &\leftarrow \textsc{Hash}(\mathscr{R}^{dec}_2 \mid\mid \mathscr{C}_2) \\
\mathcal{H}^{\mathbb{S}}_{\mathcal{TN}_2} &\leftarrow \textsc{Hash}(\mathcal{H}^{\mathbb{S}}_{\mathscr{C}_2} \mid\mid \mathcal{H}_{\mathcal{TN}}) \\
\mathcal{H}^{\mathbb{S}}_{\mathcal{TR}_2} &\leftarrow \textsc{Hash}(\mathcal{H}^{\mathbb{S}}_{\mathcal{TN}_2} \mid\mid \mathcal{H}_{\mathcal{TR}}) \\
\mathcal{H}^{\mathbb{S}}_{\mathscr{R}_2} &\leftarrow \textsc{Hash}(\mathcal{H}^{\mathbb{S}}_{\mathcal{TR}_2} \mid\mid \mathscr{R}^{dec}_2)
\end{aligned} \tag{39}$$

The server compares the hash values of the server-side and client-side as

$$\begin{aligned}
&\textbf{if } \mathcal{H}_{\mathscr{R}_1} = \mathcal{H}^{\mathbb{S}}_{\mathscr{R}_1} \textit{ and } \mathcal{H}_{\mathscr{R}_2} = \mathcal{H}^{\mathbb{S}}_{\mathscr{R}_2} \\
&\quad User \textit{ is authenticated.} \\
&\textbf{else} \\
&\quad Authentication \textit{ fails.}
\end{aligned} \tag{40}$$

If the above-given equation holds, the user is authenticated. Otherwise, the authentication fails.

4. Bloom Filter

Bloom Filter is an approximate membership data structure with a tiny memory footprint for large input items. Bloom Filter returns either true or false on a given query. If Bloom Filter returns true, which means the item may be present, but it is not guaranteed. It has a false positive issue. However, if the Bloom Filter returns false, it guarantees the item is not present. We exploit this advantage (guarantee) of Bloom Filter. Authentica uses Bloom Filter to maintain the list of valid user IDs to prevent invalid users.

Let the adversary wants authentication from the server, and it cannot get authentication from the server due to an incorrect response for a given challenge. Let us assume that the adversary evades the security of the captcha because the authentication phase does not apply cryptography. The adversary wants to down the service by launching a DDoS attack on the server. The adversary launches many authentication requests. The adversary needs to generate a valid user ID to pass the barrier of the Bloom Filter because the Bloom Filter examines the validity of the user ID. Therefore, the adversary cannot cross the barrier of Bloom Filter with an invalid user ID. However, a few requests can cross the barrier of Bloom Filter due to false positive cases. Therefore, the server processes these requests but cannot gain authentication against the user IDs since the user IDs are invalid. Therefore, most of the requests cannot cross the barrier. Bloom Filter is a fast and lightweight data structure that handles millions of requests per second. Moreover, it uses a tiny memory footprint for many user IDs. Therefore, the adversary cannot launch a DDoS attack even if the captcha is not working.

There are diverse Bloom Filters available, which can be either multi-threaded Bloom Filter or non-threaded. The Authentica does not require a multi-threaded Bloom Filter or parallel Bloom Filter. Therefore, we can use standard Bloom Filter [18], or robustBF [19], [20]. robustBF consumes 10.40× and 44.01× less memory than SBF and CBF on average, respectively [19]. Also, robustBF is 2.038× and

2.48× faster in the insertion of 10M data than SBF [18] and CBF [21], respectively [19]. Thus, robustBF is an extremely efficient Bloom Filter that is suitable for Authentica.

5. Failure of authentication request

The failure table $\mathcal{FT}$ is a hashtable that maintains the user ID, the count of its failed attempt to get authentication, its recent timestamp for the failed attempt, and the backlist information. The blacklist has several levels of blocking: Level 0, Level 1, Level 2, Level 3, Level 4, Level 5, and Level 6. Each user starts from Level 0, which means no blocking. Level 1 blocks the user for five seconds on three consecutive failed attempts. After five seconds, Level 2 activates if the user performs another three failed attempts, i.e., the user cannot get authentication since the last blocking. The next level is activated upon three failed attempts. Level 2 blocks the user for 1 minute. Similarly, Level 3 blocks the user for 5 minutes, Level 4 blocks the user for half hour, Level 5 blocks the user for one hour, and Level 6 blocks the user for a day. After each blocking, three attempts are given to the user to get an authentication. Failure to get authentication causes inconvenience to the user. After expiring of Level 6, it reset to Level 1 again upon failure of getting authentication.

A user is authenticated upon successfully reproducing the responses for the challenges. Otherwise, the authentication fails. If authentication fails, the failed attempt counter is incremented as

$$\mathcal{FT}.\mathcal{ID}.counter = \mathcal{FT}.\mathcal{ID}.counter + 1 \tag{41}$$

If $\mathcal{FT}.\mathcal{ID}.counter \geq 3$, the blocking information of the user is updated as

$$\begin{aligned} &\textbf{if } \mathcal{FT}.\mathcal{ID}.counter \geq 3 \\ &\quad \mathcal{FT}.\mathcal{ID}.blocking = level\ l\%7 + 1 \\ &\quad \mathcal{FT}.\mathcal{ID}.\mathcal{T}_{auth} = \mathcal{T}_{current} \\ &\quad \mathcal{FT}.\mathcal{ID}.counter = 0 \end{aligned} \tag{42}$$

The blocking status is upgraded to the next level by $\mathcal{FT}.\mathcal{ID}.blocking = level\ l\%7 + 1$. If the blocking level is Level 6, then the blocking level is upgraded to Level 1. The timestamp of the blocking is recorded. Also, the counter is set to zero, allowing the user to perform three failed attempts upon expiration of the blocking.

6. Hashtable for user ID

The hashtable contains the user ID, timestamp of the authentication time, and authentication information. The user request to the server for authentication using its ID. The server checks the hashtable for the user ID. The server retrieves the current timestamp if the user ID is in the hashtable. Let the authentication timestamp be $\mathcal{T}_{auth}$ and the current timestamp be $\mathcal{T}_{current}$. The difference between the timestamp must be greater than a threshold δ; otherwise, assign the user ID as logged out as given below-

$$\begin{aligned} &\textbf{if } \mathcal{ID} \in \mathcal{HT} \text{ and } \mathcal{HT}.\mathcal{ID}.logout = true \\ &\quad Proceed\ to\ the\ next\ step \\ &\textbf{else if } \mid \mathcal{HT}.\mathcal{ID}.\mathcal{T}_{auth} - \mathcal{T}_{current} \mid \geq \delta \\ &\quad \mathcal{HT}.\mathcal{ID}.logout = true \\ &\quad Proceed\ to\ the\ next\ step \\ &\textbf{else} \\ &\quad Block\ the\ request \end{aligned} \tag{43}$$

If the user ID is logged out, i.e., $\mathcal{ID} \in \mathcal{HT}$ and $\mathcal{HT}.\mathcal{ID}.logout = true$, the server proceeds to the next step. Otherwise, the server checks the login session expiration time using the timestamps. This hashtable ensures the authenticated users are not authenticated again. Also, the user ID must belong to this hash table for verification. Therefore, the adversary cannot initiate the authentication process. However, the authentication process is performed without encryption so that the adversary can get all valid user IDs.

7. The challenge-response database

The challenge-response (CRP) database is prone to attack by adversaries. Attackers try to breach the server's security and leak the CRP database. To protect the CRP database, it requires encryption of the CRP database. A recent development suggests that the CRP database can be encrypted using a key generated by PUF [22]. The adversary's PUF cannot produce the same results as the server's PUF. To generate a key using the server's PUF, we assume that the server's PUF is noise free. However, the damage to the circuit causes an issue; therefore, it requires a backup server with a different PUF [22]. Let the function be the PUF( ). We generate a key $\mathscr{K}$ to encrypt the response of the client as

$$\mathscr{K} \leftarrow \text{PUF}(ServerKey) \tag{44}$$

There are two key drawbacks of the above-mentioned encryption: a) PUF damage caused no recovery, and b) if the $\mathscr{K}$ is compromised, the response can be revealed. Firstly, the PUF can damage or malfunction at any given time. Therefore, it requires an active backup of the CRP database where modification of the main CRP database triggers a modification in the backup server. There are two reasons to modify the CRP database: a) inserting a new user and b) forgotten responses. These operations are not frequent, and a backup server can easily be maintained. Secondly, the entire response database can be revealed if the adversary discovers the $\mathscr{K}$ either accidentally or using a collision attack (or any other method). To solve this issue, we use a set of $ServerKey$ as defined below-

$$\begin{aligned} ServerKey = \{&(\mathcal{SK}_{11}, \mathcal{SK}_{12}), (\mathcal{SK}_{21}, \mathcal{SK}_{22}), \\ &(\mathcal{SK}_{31}, \mathcal{SK}_{32}) \ldots, (\mathcal{SK}_{n1}, \mathcal{SK}_{n2})\} \end{aligned} \tag{45}$$

The server's keys are stored pairwise for each user. We term these pair as semi-secret or semi-private words because these words are stored unencrypted in the server. The server can pick i^{th} pair from the $Serverkey$ for i^{th} user and computes the secret key for encryption as

$$\begin{aligned} \mathscr{K}_{i1} &\leftarrow \text{PUF}(ServerKey_{i1}) \\ \mathscr{K}_{i2} &\leftarrow \text{PUF}(ServerKey_{i2}) \end{aligned} \tag{46}$$

The computed keys are updated using the SHA512 hash function as shown in Equation (24). Again, the keys are updated using Argon2i which requires salt to update the keys as

$$\begin{aligned} \mathscr{K}_{i1} &\leftarrow \textsc{Argon2i}(\mathscr{K}_{i1}, \mathscr{S}_{i1}) \\ \mathscr{K}_{i2} &\leftarrow \textsc{Argon2i}(\mathscr{K}_{i2}, \mathscr{S}_{i2}) \end{aligned} \tag{47}$$

where

$$\begin{aligned} \mathcal{S} =& \{(\mathcal{S}_{11}, \mathcal{S}_{12}), (\mathcal{S}_{21}, \mathcal{S}_{22}), (\mathcal{S}_{31}, \mathcal{S}_{32}), \\ & \ldots, (\mathcal{S}_{n1}, \mathcal{S}_{n2})\} \\ \mathscr{S}_{i1} &\leftarrow \text{PUF}(\mathcal{S}_{i1}) \\ \mathscr{S}_{i2} &\leftarrow \text{PUF}(\mathcal{S}_{i2}) \\ \mathscr{S}_{i1} &\leftarrow \textsc{Hash512}(\mathscr{S}_{i1}) \\ \mathscr{S}_{i2} &\leftarrow \textsc{Hash512}(\mathscr{S}_{i2}) \end{aligned} \tag{48}$$

Similar to the $ServerKey$, the $\mathcal{S}$ is also a semi-secret where the words are stored in the server unencrypted. Notably, we input two secrets into the Argon2i hash function: the secret key and the secret salt. Therefore, it is computationally hard to discover the secret key to decrypt the response because Argon2i is a memory-hard hash function suitable to prevent parallel computation of the adversary.

The ciphertexts ($\mathscr{E}_{\mathscr{R}_1}$ and $\mathscr{E}_{\mathscr{R}_2}$) are inserted into the response database. The server securely deletes the server's key $\mathscr{K}_{i1}$ and $\mathscr{K}_{i2}$. The $ServerKey$ can be public, but we termed it semi-private or semi-secret because these are stored in the server without encryption. Let us assume the adversary discovers $\mathscr{K}_{i1}$, and therefore, the adversary can leak $\mathscr{R}_1$. But the adversary cannot get $\mathscr{K}_{i2}$ to leak the $\mathscr{R}_2$. Let us assume the adversary discovers $\mathscr{K}_{i1}$ and $\mathscr{K}_{i2}$. So the adversary can reveal the $\mathscr{R}_1$ and $\mathscr{R}_2$, which does not mean that adversary can reveal other responses. Therefore, it is secure even if a key is leaked or compromised.

7.1. Securely deleting a variable

Securely deleting a variable is an issue for the developer due to the side-channel attack. Let us assume a developer uses a variable $SecretKey =$ "$Alan\ Turing$". If we directly release the variable, the character will not be deleted, and these characters will remain on the memory allocated for the variable. Therefore, before deleting the variable, we assign the variable with white spaces as $SecretKey =$ " " where all characters of the variable are replaced with white spaces. Now, we can free or delete the variable. This simple technique left no characters traceable for the adversary. Thus, an adversary can learn nothing from the allocated memory for the variables.

7.2. Client-side password strength

The password is not sent to the server, so the server cannot suggest the password strength. Therefore, the server cannot offer Ajax services. Hence, it requires client-side password monitoring. It is trivial to implement such kinds of systems. Such a system counts the symbols, capital letters, small letters, and digits. Based on the minimum counts of these characters, we can decide whether the password is strong or not. Let the symbol count, small letter count, upper letter count, and digits count are $Symb$, Ups, $Small$, and $Digit$, respectively. We measure the password and secret context's strength as

$$\begin{aligned} & min = \textsc{Minimum}(Symb, Ups, Small, Digit) \\ & \quad \textbf{if } min = 1 \\ & \qquad Weak\ secret \\ & \quad \textbf{else if } min = 2 \\ & \qquad Good\ secret \\ & \quad \textbf{else if } min = 3 \\ & \qquad Strong\ secret \\ & \quad \textbf{else} \\ & \qquad Very\ strong\ secret. \end{aligned} \tag{49}$$

Moreover, the password and context length must be greater than or equal to 10 characters.

8. Security Analysis

We analyze the security strength of Authentica, and it can prevent the domino effect, dictionary attacks, rainbow table attacks, collision attacks, impersonation attacks, replay attacks, and DDoS attacks described in this section.

8.1. Collision attack

The collision attack states that the hash algorithm produces the same hash values even if the two input strings are different. Precisely, the $\mathcal{H}_\omega = \mathcal{H}'_\omega$ where $\omega \neq \omega'$. There is always a possibility of collision in the hash algorithms because the bit size limits the hash value. On the contrary, the input space is infinitely large, and we can assume that the universe is the input space. Suppose the adversary can produce a response $\mathscr{R}_1$ of a given challenge $\mathscr{C}_1$ due to a collision. The adversary requires another response $\mathscr{R}_2$ for the challenge $\mathscr{C}_2$. The adversary needs to perform a collision attack on the challenge $\mathscr{C}_2$ to reproduce the response $\mathscr{R}_2$. Authentica uses the SHA512 hash algorithm to produce the hash values; therefore, it is computationally hard to produce the response $\mathscr{R}_2$ for the challenge $\mathscr{C}_2$. If it is computationally hard to reproduce $\mathscr{R}_2$, thus so $\mathscr{R}_1$ too. Let us assume two sets of words to perform a collision attack by the adversary: $\mathcal{X} = \{x_i, x_2, x_3, \ldots, x_m\}$ and $\mathcal{Y} = \{y_1, y_2, y_3, \ldots, y_n\}$ where m and n are sufficiently large enough to perform a collision attack. The probability of selecting a correct word that collides with the $\mathscr{R}_1$ is $\frac{1}{m}$

and the $\mathscr{R}_2$ is $\frac{1}{n}$. Alternatively, the x_i correctly produces $\mathscr{R}_1$ and the y_j correctly produces $\mathscr{R}_2$ where $i = 1, 2, 3, \ldots$ and $j = 1, 2, 3, \ldots$. The probability of selecting two words from $\mathcal{X}$ and $\mathcal{Y}$ that correctly produces $\mathscr{R}_1$ and $\mathscr{R}_2$, respectively, is $\frac{1}{\binom{mn}{2}}$ which is sufficiently small enough to thwarts collision attack. We assumed that the words from $\mathcal{X}$ and $\mathcal{Y}$ can correctly reproduce the $\mathscr{R}_1$ and $\mathscr{R}_2$, which is not necessary. Therefore, the collision attack is computationally hard in Authentica.

8.2. Impersonate attack

Authentication is a crucial process to protect the digital assets. It should ensure that only legitimate users can get authentication while illegitimate users are prevented from getting authentication. Let us assume the adversary generates responses $\mathcal{H}^{adv}_{\mathscr{R}_1}$ and $\mathcal{H}^{adv}_{\mathscr{R}_2}$. To perform an impersonation attack, the following equation must hold-

$$\begin{aligned}\mathcal{H}^{adv}_{\mathscr{R}_1} &= \textsc{ReconstructionHashValue}(\mathscr{R}^{dec}_1, \mathcal{TR}, \mathcal{TN})\\ \mathcal{H}^{adv}_{\mathscr{R}_2} &= \textsc{ReconstructionHashValue}(\mathscr{R}^{dec}_2, \mathcal{TR}, \mathcal{TN})\end{aligned} \tag{50}$$

where ReconstructionHashValue( ) is a reconstruction of the hash value using the given parameter by the server. To perform the above-given impersonation attack, the following equation must satisfy.

$$\begin{aligned}\mathcal{H}^{adv}_{\mathscr{R}_1} &\neq \mathcal{H}^{recent}_{\mathscr{R}_1}\\ \mathcal{H}^{adv}_{\mathscr{R}_2} &\neq \mathcal{H}^{recent}_{\mathscr{R}_2}\end{aligned} \tag{51}$$

Moreover, it must also satisfy the following equation.

$$\begin{aligned}\mathcal{TN}_{adv} &> \mathcal{TN}_{recent}\\ \mathcal{TR}_{adv} &> \mathcal{TR}_{recent}\end{aligned} \tag{52}$$

Suppose, the $\mathcal{TN}_{adv} > \mathcal{TN}_{recent}$ and the server generates a timestamp $\mathcal{TN}$ for authentication which is $\mathcal{TN}_{adv} = \mathcal{TN}$. If it holds, then Equation (50) cannot hold. The adversary is the legitimate user if Equation (50)-(52) hold. Thus, the adversary cannot perform an impersonation attack.

Theorem 1. *The two users, $\mathbb{A}$ and $\mathbb{B}$, cannot produce the same response for a common challenge, i.e., the $\mathscr{R}_{\mathbb{A}} \leftarrow \text{SUF}(\mathscr{C})$ and $\mathscr{R}_{\mathbb{B}} \leftarrow \text{SUF}(\mathscr{C})$ where $\mathscr{R}_{\mathbb{A}} \neq \mathscr{R}_{\mathbb{B}}$ if $\mathbb{A} \neq \mathbb{B}$.*

Proof. To impersonate, the adversary must satisfy the following equation

$$\mathscr{R}_{\mathbb{A}} = \mathscr{R}_{\mathbb{B}} \; and \; \mathbb{A} \neq \mathbb{B} \tag{53}$$

The above-given condition holds if the following equation holds

$$\mathscr{P}_{\mathbb{A}} = \mathscr{P}_{\mathbb{B}} \; and \; \zeta_{\mathbb{A}} = \zeta_{\mathbb{B}} \tag{54}$$

Coincidentally, the passwords and secret context are the same for the two different users. However, this case can easily be prevented in the registration phase. The duplicate value for $\mathscr{R}_1$ and $\mathscr{R}_2$ cannot be inserted into the response database. Thus, Equation (53) cannot satisfy two different users. If the adversary correctly guesses the two secrets of a user, then the adversary can gain authentication. The guessing should be precise for a given user. For instance, the adversary cannot impersonate if a user's password and secret context are correctly guessed within three attempts. Different users' passwords and secret contexts cannot impersonate another user. □

8.3. Replay attack

The server generates a random nonce which is generated by a true random number generator. Also, the server gets the current timestamp. Let the random nonce be $\mathcal{N}$, and the timestamp be $\mathcal{T}$. The concatenated string is $\mathcal{TN} \leftarrow \mathcal{T} \; || \; \mathcal{N}$. We know that the timestamp is comparable and unique. Therefore, the $\mathcal{TN}$ is also comparable and unique. Let the timestamps $\mathcal{T}_1, \; \mathcal{T}_2, \mathcal{T}_3, \ldots,$ be the timestamps. The $\mathcal{T}_i \neq \mathcal{T}_j$ where $i \neq j$. The $\mathcal{T}_i > \mathcal{T}_j$ if $i > j$. Similarly, the $\mathcal{T}_i = \mathcal{T}_j$ if $i = j$.

$$\begin{cases}\mathcal{T}_i = \mathcal{T}_j, & if \; i = j\\ \mathcal{T}_i > \mathcal{T}_j, & if \; i > j\\ \mathcal{T}_i < \mathcal{T}_j, & if \; i < j\end{cases} \tag{55}$$

The timestamps are comparable because these are constituted using time, increasing order of number, or both. For instance, $\mathcal{T}_1$ = 1_2023-04-20-10-30-15 and $\mathcal{T}_2$ = 2_2023-04-20-10-30-20. The true random number generator can generate a unique random nonce, say $\mathcal{N}_1, \; \mathcal{N}_2, \; \mathcal{N}_3, \ldots$. The $\mathcal{N}_i \neq \mathcal{N}_j$ where $i \neq j$.

$$\begin{cases}\mathcal{N}_i = \mathcal{N}_j, & if \; i = j\\ \mathcal{N}_i \neq \mathcal{N}_j, & if \; i \neq j\end{cases} \tag{56}$$

Similarly, the user computes the timestamp, denoted as $\mathcal{TR}$— the current time and date of the user's system and a random number. Unlike the server's timestamp, the user does not add a number of increasing orders. However, the $\mathcal{TR}$ is also comparable.

Theorem 2. *The $\mathcal{TN}_{adv} < \mathcal{TN}$ is a replay attack where $\mathcal{TN}$ is expected timestamp.*

Proof. The server and client exchange these timestamps without encryption. Therefore, the adversary can get a copy of these timestamps and hash values. The adversary can perform a replay attack using previous hash values to gain authentication. The server stores the recently authenticated users with their timestamps and hash values to prevent replay attacks. Let the recent hash values be the $\mathcal{H}^{recent}_{\mathscr{R}_1}$ and $\mathcal{H}^{recent}_{\mathscr{R}_2}$, and the recent timestamps be the $\mathcal{TN}_{recent}$ and $\mathcal{TR}_{recent}$. The user gains authentication from the server using these values, and the server stores these values against the user. Let the adversary captures these values and resend them to the server to gain authentication. The server examines these values in a hashtable in Equation (36). Let the hash values of the adversary be $\mathcal{ADV}_{\mathscr{R}_1}$ and $\mathcal{ADV}_{\mathscr{R}_2}$, and the timestamps of the adversary be $\mathcal{TN}_{adv}$ and $\mathcal{TR}_{adv}$. We have two cases: recent authenticated values and previous authentication values. The adversary can replay the

recently authenticated values. Therefore, the following equation holds for the first case.

$$\textbf{if } \mathcal{ADV}_{\mathscr{R}_1} = \mathcal{H}_{\mathscr{R}_1}^{recent} \; and \; \mathcal{ADV}_{\mathscr{R}_2} = \mathcal{H}_{\mathscr{R}_2}^{recent} \; It's \; a \; replay \; attack. \tag{57}$$

If the above equation holds, the user is already authenticated using the same values, and it is a case of a replay attack. For the second case, the adversary can store all values of a given user to perform a replay attack. Therefore, the above equation cannot hold. Thus, we check the timestamps as given below-

$$\textbf{if } \mathcal{TN}_{adv} \leq \mathcal{TN}_{recent} \; and \; \mathcal{TR}_{adv} \leq \mathcal{TN}_{recent} \; It's \; a \; replay \; attack. \tag{58}$$

The adversary must produce two hash values for a given $\mathcal{TN}$ to get authentication. Moreover, the adversary must produce the hash values within a few seconds; otherwise, it becomes a replay attack. □

This method causes inconvenience if the user cannot update its system time and date, i.e., the time and date of the user's system cannot malfunction. It ensures the adversary cannot perform a replay attack even if the server does not store the history of the authentication of a user. Therefore, the server does not require to store the history of the hash values and timestamps.

8.4. Dictionary and rainbow table attack

The dictionary attack is carried out by performing precomputed hash values. For instance, the dictionary is constructed as

$$Dict = \{h_1 : x_1, \; h_2 : x_2, \; h_3 : x_3, \ldots\} \tag{59}$$

The adversary performs a lookup operation on the dictionary for a given hash value to discover the corresponding input. For instance, if the adversary receives a hash value h_i, then the corresponding raw string is x_i. Assume that the adversary discovers the corresponding input for the response $\mathcal{H}_{\mathscr{R}_1}$ due to collision. The adversary must be able to create a collision on $\mathcal{H}_{\mathscr{R}_1}$ to evade the security, which is computationally hard due to the SHA512 hash algorithm. Moreover, the hash values of the responses alter in each communication. Therefore, it is difficult to discover the original password and context using the dictionary attack.

Similarly, the rainbow table attack is also a powerful method to crack passwords, similar to a dictionary attack. It creates a chain of precomputed hash values with a reduction function. For instance, the leaked hash value h_i is reduced to x_i for the hash value using a rainbow table attack. The rainbow table attack does not apply since the CRP database is encrypted using a key derived using Argon2i. However, the adversary can perform a rainbow table attack on $\mathcal{H}_{\mathscr{R}_1}$ and $\mathcal{H}_{\mathscr{R}_2}$. The rainbow table attack can be successful only when the communication is fixed hash values for $\mathcal{H}_{\mathscr{R}_1}$ and $\mathcal{H}_{\mathscr{R}_2}$ but Authentica alters the hash values in each communication. Thus, the rainbow table is computationally hard to perform.

8.5. Domino effect

The password-based authentication system is prone to a domino effect due to password reuse in multiple domains. We use domain words to defeat such kinds of attacks similar to Ross *et. al.* [17]. Also, we encourage the user to reuse the passwords in multiple domains.

Theorem 3. *Given two different domains ($\mathscr{D}_1$ and $\mathscr{D}_2$), the same challenges ($\mathscr{C}_1$ and $\mathscr{C}_2$), and the same secret words ($\mathscr{P}$ and ζ). The responses for domain $\mathscr{D}_1$ are $\mathscr{R}_1$ and $\mathscr{R}'_1$, and for domain $\mathscr{D}_2$ are $\mathscr{R}_2$ and $\mathscr{R}'_2$. If $\mathscr{D}_1 \neq \mathscr{D}_2$ holds then $\mathscr{R}_1 \neq \mathscr{R}_2$ and $\mathscr{R}'_1 \neq \mathscr{R}'_2$.*

Proof. We need to prove that a password and a secret context for two different domain words always translate into different responses for the same challenges, i.e., the adversary cannot perform the domino effect. Our method shuffles the challenge with a domain word in the context of a secret word. The domain words are $\mathscr{D}_1$ and $\mathscr{D}_2$ where $\mathscr{D}_1 \neq \mathscr{D}_2$ and non-empty. The challenges are $\mathscr{C}_1$ and $\mathscr{C}_2$, and password is $\mathscr{P}$. Let us construct the responses of the domain name $\mathscr{D}_1$. The client hashes the challenge and the domain word $\mathscr{D}_1$ as

$$\begin{aligned} \mathcal{H}_{\mathscr{C}_1} &\leftarrow \textsc{Hash512}(\mathscr{C}_1) \\ \mathcal{H}_{\mathscr{C}_2} &\leftarrow \textsc{Hash512}(\mathscr{C}_2) \\ \mathcal{H}_{\mathscr{D}_1} &\leftarrow \textsc{Hash512}(\mathscr{D}_1) \end{aligned} \tag{60}$$

We shuffle these two hash values using a secret context as

$$\begin{aligned} \mathcal{SD}_1 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathscr{C}_1}, \mathcal{H}_{\mathscr{D}_1}) \\ \mathcal{SD}'_1 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathscr{C}_2}, \mathcal{H}_{\mathscr{D}_1}) \end{aligned} \tag{61}$$

The client converts the shuffled words into a hash value as

$$\begin{aligned} \mathcal{H}_{\mathcal{SD}_1} &\leftarrow \textsc{Hash}(\mathcal{SD}_1) \\ \mathcal{H}'_{\mathcal{SD}_1} &\leftarrow \textsc{Hash}(\mathcal{SD}'_1) \end{aligned} \tag{62}$$

Again, the client hashes the password as

$$\mathcal{H}_{\mathscr{P}} \leftarrow \textsc{Hash}(\mathscr{P}) \tag{63}$$

The $\mathcal{H}_{\mathcal{SD}}$ and $\mathcal{H}_{\mathscr{P}}$ shuffled using a secret context as

$$\begin{aligned} \mathcal{SW}_1 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}'_{\mathcal{SD}_1}, \mathcal{H}_{\mathscr{P}}) \\ \mathcal{SW}'_1 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}'_{\mathcal{SD}_1}, \mathcal{H}_{\mathscr{P}}) \end{aligned} \tag{64}$$

The newly shuffled word is converted into a hash value as

$$\begin{aligned} \mathscr{R}_1 &\leftarrow \textsc{Hash}(\mathcal{SW}_1) \\ \mathscr{R}'_1 &\leftarrow \textsc{Hash}(\mathcal{SW}'_1) \end{aligned} \tag{65}$$

For the same secrets and challenge, the client computes the hash value for the different domain $\mathscr{D}_2$ as

$$\mathcal{H}_{\mathscr{D}_2} \leftarrow \textsc{Hash512}(\mathscr{D}_2) \tag{66}$$

The $\mathcal{H}_{\mathscr{D}_1} \neq \mathcal{H}_{\mathscr{D}_2}$ holds since $\mathscr{D}_1 \neq \mathscr{D}_2$. The hash values are shuffled with the hash value of the challenge as

$$\begin{aligned} \mathcal{SD}_2 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathscr{C}_1}, \mathcal{H}_{\mathscr{D}_2}) \\ \mathcal{SD}'_2 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}_{\mathscr{C}_2}, \mathcal{H}_{\mathscr{D}_2}) \end{aligned} \tag{67}$$

where

$$\begin{aligned} \mathcal{SD}_1 &\neq \mathcal{SD}_2 \\ \mathcal{SD}'_1 &\neq \mathcal{SD}'_2 \end{aligned} \tag{68}$$

The client converts the shuffled word into a hash value as

$$\begin{aligned} \mathcal{H}_{\mathcal{SD}_2} &\leftarrow \textsc{Hash}(\mathcal{SD}_2) \\ \mathcal{H}'_{\mathcal{SD}_2} &\leftarrow \textsc{Hash}(\mathcal{SD}'_2) \end{aligned} \tag{69}$$

Therefore, the following equation holds.

$$\begin{aligned} \mathcal{H}_{\mathcal{SD}_1} &\neq \mathcal{H}_{\mathcal{SD}_2} \\ \mathcal{H}'_{\mathcal{SD}_1} &\neq \mathcal{H}'_{\mathcal{SD}_2} \end{aligned} \tag{70}$$

The shuffled hash value is shuffled with the hash value of the password as

$$\begin{aligned} \mathcal{SW}_2 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}'_{\mathcal{SD}_2}, \mathcal{H}_{\mathscr{P}}) \\ \mathcal{SW}'_2 &\xleftarrow{\zeta} \textsc{Shuffle}(\mathcal{H}'_{\mathcal{SD}_2}, \mathcal{H}_{\mathscr{P}}) \end{aligned} \tag{71}$$

Also, the following equation holds.

$$\begin{aligned} \mathcal{SW}_1 &\neq \mathcal{SW}_2 \\ \mathcal{SW}'_1 &\neq \mathcal{SW}'_2 \end{aligned} \tag{72}$$

The shuffled word is converted into a hash value as

$$\begin{aligned} \mathscr{R}_2 &\leftarrow \textsc{Hash}(\mathcal{SW}_2) \\ \mathscr{R}'_2 &\leftarrow \textsc{Hash}(\mathcal{SW}'_2) \end{aligned} \tag{73}$$

We can conclude that two different domain words produce different responses for the same challenge and secret words, which is shown below

$$\begin{aligned} \mathscr{R}_1 &\neq \mathscr{R}_2 \\ \mathscr{R}'_1 &\neq \mathscr{R}'_2 \end{aligned} \tag{74}$$

Therefore, an adversary cannot gain authentication in other domains even if the adversary steals the challenge-response database of a server. The hash values for the same password and secret word differ for different domains. Thus, our proposed mechanism encourages the reuse of secret words in multiple domains without risking the authentication process. Thus, it eliminates the domino effect. □

8.6. Guessing attacks

The guessing attack is one of the most powerful attacks where the adversary can evade security using a few trials. The user always sets easy secrets such that they can easily remember the secrets. Therefore, the guessing attack becomes more powerful. Authentica uses two secrets: password and secret context. Let us assume that the client's name is Alan Turing, the password is Alan@1234, and the secret context is Turing@1234. It becomes easy to be guessed by the adversary. Therefore, we use the following rules to prevent a guessing attack-

- The minimum length of the secret context and password is ten characters long: composed of at least a capital letter, a small letter, a digit, and a symbol.
- The secret and password cannot be the email address.
- The password and the secret context cannot be the same.
- The given name and surname cannot be a substring of secret context and password.

The above-given rule for the password and the secret context is much harder to perform a guessing attack. However, there is still room for weak secrets but a combination of two secrets can make it stronger.

8.7. DDoS attack

It is easy to perform a distributed denial of service attack on a PUF-based authentication mechanism because the communications are unencrypted. Therefore, we rely on a captcha to prevent requests from bots. Moreover, Authentica uses a lightweight data structure, Bloom Filter, to filter out unwanted user IDs. Furthermore, the user ID is examined in the authentication and failure tables, which are implemented based on the hashtable. These data structures can handle millions of requests per second. Thus, the DDoS attack cannot down the service of Authentica.

8.8. Revealing password

Recent developments suggest that the adversary has already revealed a few billion passwords [23]–[25]. COMB publishes 3.2 billion unique pairs of cleartext emails and passwords. Similarly, RockYou2021 publishes 8.4 billion passwords. However, the CrackStation can perform lookup operations on a computed hash value to inverse. Authentica shuffles the hash value of the domain name, password, and challenge to produce a response using a secret context to prevent secrets leakage. The hash value is stored as a response in the server by encrypting it with a separate secret key. The server does not deal with the plaintext of the secrets. Therefore, it is computationally infeasible to leak the raw secret from the hash value.

9. Conclusion

In this paper, we presented SUF to authenticate users instead of authenticating the devices. The SUF is adapted to implement an authentication mechanism without encryption called Authentica. Authentica uses two secrets to prevent diverse attacks: impersonation attacks and collision attacks. Moreover, Authentica uses timestamps to defeat replay attacks. The rainbow table and dictionary attacks are addressed by encrypting each response using a different secret key. The DDoS attack is addressed by applying Captcha, Bloom Filter, and Hashtable. Also, Authentica uses two responses to avoid collision attacks. Moreover, the domino effect issue is addressed by shuffling the domain name with the challenge while producing a response so the user can reuse their secrets without any risks. Furthermore, it uses two secrets to defeat the guessing attacks. Thus, we have demonstrated how Authentica authenticates the users without using cryptography protocol yet better than the state-of-the-art authentication protocols.

References

[1] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, no. 5589, pp. 2026–2030, 2002. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.1074376

[2] Y. Gao, S. F. Al-Sarawi, and D. Abbott, "Physical unclonable functions," *Nat. Electron.*, vol. 3, pp. 81–91, Feb. 2020.

[3] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, 2014.

[4] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber, "Modeling attacks on physical unclonable functions," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 237–249. [Online]. Available: https://doi.org/10.1145/1866307.1866335

[5] S. Yu, A. K. Das, Y. Park, and P. Lorenz, "Slap-iod: Secure and lightweight authentication protocol using physical unclonable functions for internet of drones in smart city environments," *IEEE Transactions on Vehicular Technology*, vol. 71, no. 10, pp. 10 374–10 388, 2022.

[6] G. Bansal, N. Naren, V. Chamola, B. Sikdar, N. Kumar, and M. Guizani, "Lightweight mutual authentication protocol for v2g using physical unclonable function," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7234–7246, 2020.

[7] Q. Zhang, J. Wu, H. Zhong, D. He, and J. Cui, "Efficient anonymous authentication based on physically unclonable function in industrial internet of things," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 233–247, 2023.

[8] P. Gope, O. Millwood, and B. Sikdar, "A scalable protocol level approach to prevent machine learning attacks on physically unclonable function based authentication mechanisms for internet of medical things," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 3, pp. 1971–1980, 2022.

[9] J. Cui, J. Yu, H. Zhong, L. Wei, and L. Liu, "Chaotic map-based authentication scheme using physical unclonable function for internet of autonomous vehicle," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 3, pp. 3167–3181, 2023.

[10] H. M. S. Badar, S. Qadri, S. Shamshad, M. F. Ayub, K. Mahmood, and N. Kumar, "An identity based authentication protocol for smart grid environment using physical uncloneable function," *IEEE Transactions on Smart Grid*, vol. 12, no. 5, pp. 4426–4434, 2021.

[11] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, "Targeted online password guessing: An underestimated threat," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1242–1254. [Online]. Available: https://doi.org/10.1145/2976749.2978339

[12] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor, "Fast, lean, and accurate: Modeling password guessability using neural networks," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 175–191. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/melicher

[13] M. Xu, C. Wang, J. Yu, J. Zhang, K. Zhang, and W. Han, "Chunk-level password guessing: Towards modeling refined password composition representations," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 5–20. [Online]. Available: https://doi.org/10.1145/3460120.3484743

[14] F. Yu and M. V. Martin, "Gnpassgan: Improved generative adversarial networks for trawling offline password guessing," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2022, pp. 10–18.

[15] D. Pasquini, A. Gangwal, G. Ateniese, M. Bernaschi, and M. Conti, "Improving password guessing via representation learning," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1382–1399.

[16] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The tangled web of password reuse." in *NDSS*, vol. 14, 2014, pp. 23–26.

[17] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger password authentication using browser extensions," in *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association, Jul. 2005. [Online]. Available: https://www.usenix.org/conference/14th-usenix-security-symposium/stronger-password-authentication-using-browser-extensions

[18] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building a Better Bloom Filter," in *Algorithms – ESA 2006*. Berlin, Germany: Springer, 2006, pp. 456–467.

[19] S. Nayak and R. Patgiri, "Robustbf: A high accuracy and memory efficient 2d bloom filter," 2021.

[20] "robustBF," Jul. 2023, [Online; accessed 20 June 2023]. [Online]. Available: https://github.com/patgiri/robustBF

[21] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," in *Algorithms – ESA 2006*. Berlin, Germany: Springer, 2006, pp. 684–695.

[22] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford, "Ersatzpasswords: Ending password cracking and detecting password leakage," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 311–320. [Online]. Available: https://doi.org/10.1145/2818000.2818015

[23] "COMB: over 3.2 Billion Email/Password Combinations Leaked | Cybernews," Jul. 2022, [Online; accessed 15 June 2023]. [Online]. Available: https://cybernews.com/news/largest-compilation-of-emails-and-passwords-leaked-free

[24] "CrackStation - Online Password Hash Cracking - MD5, SHA1, Linux, Rainbow Tables, etc." Jul. 2023, [Online; accessed 15 June 2023]. [Online]. Available: https://crackstation.net

[25] "RockYou2021: Largest Ever Password Compilation Leaked | Cybernews," Feb. 2023, [Online; accessed 20 June 2023]. [Online]. Available: https://cybernews.com/security/rockyou2021-alltime-largest-password-compilation-leaked

Appendix

TABLE 3. THE USED SYMBOLS THROUGHOUT THE PAPER AND THEIR DESCRIPTIONS.

Notation	**Description**
$\mathbb{C}$	The client or user
$\mathbb{S}$	The server or identity manager
$\mathscr{P}$	User's password
ζ	User's secret context
$\mathscr{C}$	An example challenge
$\mathscr{R}$	An example response generated from $\mathscr{C}$
$\mathscr{C}_1$ and $\mathscr{C}_2$	Two challenges of Authentica for a single domain
$\mathscr{R}_1$ and $\mathscr{R}_2$	Two response generated from $\mathscr{C}_1$ and $\mathscr{C}_2$ for a single user
ω_1 and ω_2	Two example word
$\mathscr{D}$	Domain word or domain name
$\mathcal{SW}$ and $\mathcal{SD}$	Two shuffled word
$\mathcal{S}$	A positive integer value
τ	Number of iteration
$\mathscr{S}$	A salt to input into Argon2i
$\mathscr{L}$	Length of a given string
HASH512( )	SHA512 hash function
PUF( )	Physically unclonable hash function
SUF( )	Software-defined unclonable hash function
ARGON2I( )	Agron2i, a memory-hard hash function for key derivation
$\mathscr{K}$	A secret key for encryption or decryption
$\mathcal{H}_{\mathscr{P}}$	The hash value of password
$\mathcal{H}_{\mathscr{D}}$	The hash value fo the domain name
$\mathcal{H}_{\mathscr{C}}$	The hash value of the challenge
$\mathcal{H}_{\mathscr{R}}$	The hash value of the response
$\mathcal{TN}$	A timestamp generated by the server (identity manager)
$\mathcal{TR}$	A timestamp generated by the client (user)
$\mathcal{G}()$	A true random number generator
$\mathscr{BF}$	A Bloom Filter
$\mathcal{FT}$	A failure table implemented using hashtable
$\mathcal{HT}$	A hashtable for the list of users.