PassPro: A Secure Password-based Authentication Mechanism using SHF

Ripon Patgiri and L. Dolendro Singh
National Institute of Technology Silchar

***Abstract*—The password-based authentication system is a widely used authentication mechanism. However, it has several issues, including guessing attacks, dictionary attacks, rainbow table attacks, collision attacks, domino effects, phishing attacks, and database leakage issues. To avoid these attacks, FIDO authentication avoids passwords and uses a public key to implement a challenge-response-based and digital signature-based authentication mechanism. It implements a passwordless authentication mechanism to avoid various attacks that are present in password-based authentication mechanisms. The FIDO authentication mechanism stores the private key in the client's devices. Therefore, FIDO authentication is susceptible to device-specific issues, so cross-platform management becomes a concern. This issue is overcome by passkeys synchronizing with the cloud; however, hardware dependency is still an issue for such a system. The security of such a system is entirely dependent on the security of the hardware.**

To address these issues, we present a client-side password hashing method called PassPro. PassPro uses a shuffle and hash function (SHF) to implement PassPro. The SHF is used to create a unique hash value for two publicly known words using a secret context. The SHF is used to create a unique hash value on the client side, preventing the transmission of the raw password to the server. Moreover, PassPro can protect the password database using two different methods: PassPro with encryption (PassProE) and PassPro with SHF (PassProS). PassProE encrypts the password database using mutually reproducible secret keys instead of using hashing, such as Argon2i. PassProS hashes the password database with SHF. In this paper, we exemplify how PassPro can prevent various attacks, including guessing attacks, dictionary attacks, rainbow table attacks, collision attacks, domino effects, phishing attacks, and database leakage issues. Moreover, PassPro users can reuse their password in different or the same domains. Also, PassPro guarantees that adversaries cannot retrieve the user's original password from the leaked password database.

1. Introduction

The password-based authentication mechanism is one of the most popular authentication mechanisms. Conventionally, passwords are stored on the server by hashing with a secret salt to prevent several attacks. Interestingly, conventional computers are equipped with more powerful processors and GPUs at a relatively low cost, making it easier for adversaries to attack the security system due to the capabilities of launching parallel attacks. Therefore, memory-hard functions are introduced to address such kinds of parallel attacks [1], [2], [3], [4], [5], [6], [7], [8]. The memory-hard hash function restricts parallelism by allocating a significant amount of memory to hash an input, such as Argon2 [2], which makes it difficult for an adversary to launch a parallel attack. However, the memory-hard hash function consumes a significant amount of memory, so it affects the performance of the server for the following reasons: a) it can affect the performance of a memory-intensive task of a server, and b) it is slower than conventional SHA function and symmetric encryption algorithms. Moreover, these password hashing algorithms cannot prevent guessing attacks, domino effects, and phishing attacks. To this end, Fast IDentity Online (FIDO) introduces a passwordless authentication that relies on fingerprint (biometrics) and public/private key encryption [9], [10]. FIDO's webauth uses a digital signature-based authentication mechanism where a user signs the challenge using their private key, and the verifier verifies the authenticity with the public key. The FIDO's webauth securely stores the private key in the user's device, and the public key is stored in the server. Therefore, webauth is highly sensitive to the hardware because it strictly depends on the user's device, where cross-platform authentication becomes an issue [10]. Therefore, FIDO introduces passkeys to overcome the cross-platform authentication issue by synchronizing the private key with the cloud. The private key is stored in the cloud by encrypting it with a secret key generated by the user's hardware. Thus, the private key is secure even if it is stored in the cloud. Interestingly, FIDO does not have issues related to guessing attacks, dictionary attacks, rainbow table attacks, collision attacks, domino effects, phishing attacks, or database leakage. However, the webauth or passkeys still depend on the users' devices. Intuitively, the webauth or passkeys rely on the security of the user's device, and the user's device security relies on biometrics. Therefore, the security of such an authentication system depends on biometrics. Alternatively, the weakness of FIDO lies in the security of the user's device. Moreover, the fingerprint or biometric reader is a mandatory component in FIDO authentication, as its primary objective is to implement a passwordless authentication mechanism. Such systems are useful in diverse applications, such as WhatsApp or mobile applications. However, it is not suitable for financial institutions or defense organizations due to the absence of passwords. Once a biometric is compromised, there is no replacement. Moreover, the biometrics are prone to spoofing attacks. Also, the secret credentials can be breached using the passkey management service (PMS) [11].

1.1. Our proposed work

To address the drawbacks of FIDO and password-based authentication mechanisms, we propose a new password-based authentication mechanism called PassPro. We propose a shuffle and hash function (SHF) to implement PassPro. The PassPro hashes the user's secret credentials at the client side using SHF and produces four unique hash values. These four hash values are sent to the server instead of sending the raw password to the server to hash. Our proposed work hashes the secret credentials at the client side using SHF to prevent the transmission of the raw passwords. The strength of PassPro lies in SHF. The SHF implements a shuffle function to shuffle two different strings using a secret context to produce the shuffled word as defined in Definition 1. The secret context is not mixed with the shuffled word, and it is used to influence the shuffling process, producing a unique shuffled word. The shuffling algorithm is based on a pseudo-random algorithm but is deterministic, where the legitimate user can reproduce the shuffled word consistently for correct input. The PassPro utilizes this shuffling algorithm to shuffle the user credentials. The resultant shuffled strings are hashed to conceal the original raw data. The hashed values are sent to the verifier for authentication.

We propose two methods for storing secret credentials on the server: PassPro with symmetric key encryption (PassProE) and PassPro with SHF (PassProS). The PassProE encrypts the secret credentials using symmetric keys on the server side, rather than hashing with a salt. Therefore, the server cannot decrypt the client's secret credentials without the client's secret key. Similarly, the client cannot decrypt the hash value without the server keys. So, both the client and server control the symmetric key; therefore, this symmetric key is termed a mutually reproducible secret key. On the contrary, PassProS hashes the secret credentials using SHF on the server. Both client and server controls the SHF on the server side. Thus, the client cannot reproduce the hash value without the server's secret credentials. Similarly, the server cannot reproduce the hash value without using the client's secret credentials. We avoid using resource-intensive computations, such as a memory-hard hash function, to secure the password database. Thus, our method prevents overconsumption of the server's resources for security. Moreover, memory-hard hash functions are slower than symmetric key cryptography. Our proposed solution can secure the password database under various assumptions without relying on a memory-hard hash function. Moreover, our proposed work encourages users to reuse their password and user ID across multiple domains, which contrasts with Wang and Reiter [12].

Conventionally, the server is trusted not to leak the hashed password and salt simultaneously, and there is no guarantee of the leakage of secret salts. We assume that the hash values are leaked along with the salts, and the adversary can derive the equivalent or the exact password using a guessing attack or other attacks. In that case, the attacker can reuse the derived password to impersonate different domains if the user reuses the password across multiple domains. The current state-of-the-art system does not guarantee that the adversary cannot gain authentication using the derived password from the leaked hash values in a different domain if the user reuses the password. If the user reuses the passwords in other domains, there is a possibility that the adversary can gain authentication using the derived password. Moreover, conventional authentication mechanisms are unable to prevent phishing attacks. However, there is no such attack in FIDO authentication. The passwords are retrieved using phishing attacks in the traditional password-based authentication to gain access to other domains. To address the issues mentioned above, Ross *et al.* [13] utilize a domain word to hash the password, which incorporates client-side password hashing. Our proposed method, PassPro, is similar to Ross *et al.* [13] to prevent phishing attacks. However, PassPro differs from Ross *et al.* [13] in the password storage on the server side.

1.2. Key contributions

Our key contributions are outlined below-

- PassPro prevents the domino effect from the leaked password database. It shuffles the user ID, password, and domain word using a secret context to produce four unique hash values. Therefore, a user can use the same user ID and password in multiple domains.
- PassPro is designed based on the client-side password hashing to conceal the original password. It ensures that the adversary never gets the original passwords in raw form from the leaked password database.
  - PassProE: The secret credentials of each user are encrypted with a different key and stored in the encrypted ciphertext in the password database of the server. The server computes a mutually reproducible secret key using the hash value provided by the user and the server key.
  - PassProS: The secret credentials are hashed using SHF on the server side. Both the client and the server control the secret keys to hash the password with SHF.
- PassPro protects users' identities from adversaries. The secret credentials are protected with different keys; therefore, the adversary cannot decrypt the password even if the entire password database is leaked because the server does not store the secret keys.
- PassPro shows strong resistance against the collision attacks. However, there are no guessing attacks, dictionary attacks, rainbow table attacks, or collision attacks. Moreover, it can prevent domino effects, phishing attacks, and database leakage issues.

1.3. Our results

PassPro differs from conventional password-based authentication systems, where the raw passwords are never transmitted to the server for hashing. We present our results in the next sub-subsections. The PassPro solely depends on SHF to produce a unique hash value. The SHF is performed on the client side to send the secret credentials to the server for authentication.

1.3.1. Impersonation attacks. In PassPro, the prover (user/client) proves its authenticity to the verifier (the server or authentication 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 four hash values using SHF to win the game. It requires two secrets (a password and a secret context) to reproduce the hash values correctly. Therefore, the adversary either guesses the two secrets or performs a collision that correctly produces the response. Since we use two secrets, guessing becomes harder.

1.3.2. Collision attacks and rainbow table attacks. The collision is inevitable in a hash algorithm due to space limitations. The hash algorithm can resist collisions, such as SHA. Therefore, PassPro uses four hash values to authenticate the users, and these hash values are protected on the server to prevent unwanted database leakage. Thus, the adversary rainbow table and pre-computation attacks on the leaked databases are due to the encryption of the database. However, the adversary can perform a collision attack on the leaked password database, but it is computationally infeasible.

1.3.3. Dictionary attacks and guessing attacks. The dictionary attacks are addressed by introducing high-entropy passwords. However, users often set passwords that are easy to remember. Thus, guessing attacks and dictionary attacks become easy. To address this, we use a secret context to create unique hash values using SHF. We know that the guessing attack is among the most potent attacks in password-based authentication mechanisms. It is challenging to prevent offline password-guessing attacks; we protect the password database with different techniques. Therefore, offline dictionary and guessing attacks do not apply to the leaked password database of PassPro due to the secret key generated by PUF. For online attacks, let us also assume that the server does not protect against three incorrect attempts. In this setting, the adversary can perform an infinite number of guesses to gain authentication. The adversary cannot produce the correct hash values without the correct secret context. The adversary must reproduce the correct password and secret context to perform an online guessing attack. However, it can be protected using three or five incorrect login attempts. Note that PassPro's security remains intact even if the password is compromised due to the secret context.

1.3.4. Domino effect and password reuse. The domino effect occurs due to password reuse in different domains. Therefore, password reuse is strongly discouraged in the state-of-the-art authentication mechanism [14]. On the contrary, our proposed system encourages users to reuse the password. Using the same password, the user can create different hash values for each domain. We use the domain name to facilitate password reuse, which is derived from Ross *et al.* [13]. We use a secret context in the shuffling process, whereas the shuffled word does not contain any of the characters from the context. It is computationally hard to reproduce the shuffled word for the adversary without knowing the password and the secret context. The user ID, domain word, and password are shuffled using a secret context to produce a response. The shuffled word is converted into a hash value using the SHA algorithm. Therefore, the hash value of a domain cannot be the same as the hash value of another domain with the same password and secret context, which exhibits resistance to a domino effect in Authentica. Users can reuse their password even if it has been compromised. Moreover, the same user can set the same password for the same domain or different domains. The adversary cannot use the derived password from the leaked password database.

1.3.5. Phishing attacks. Our proposed method exhibits strong resistance to phishing attacks, similar to Ross *et al.* [13]. The client shuffles the domain with a password to produce a unique hash value using SHF. The client transmits four hash values to the server instead of a raw password. The different domains create different hash values for the same password and secret context. The domain name of the adversary can generate a different response for the same password and secret context. Therefore, the adversary can never use the received hash values in other domains to impersonate the legitimate user.

1.3.6. Database leakage issue. We assume that the adversary can easily leak the password database from the server. Therefore, we have two approaches to secure the password database: PassProE and PassProS. Also, we assume that the adversary can also leak the server keys. Therefore, we use a physically unclonable function (PUF) to generate the secret keys [15], [16]. The server does not store the server keys because the secret keys can be reproduced by the PUF using the same challenge. Moreover, we also reduce the reliance on the server for password security. The PassProE utilizes a mutually reproducible secret key to encrypt the user's secret credentials, thereby securing the password database. The user of PassProE supplies the secret key to the server, and the server merges the client's secret key with the server's secret key to generate an encryption or decryption key. This encryption key is termed as the mutually reproducible secret key. Thus, the server needs the mutually reproducible secret key to encrypt or decrypt the password database. So, an adversary cannot generate the mutually reproducible secret key without knowing the secret key supplied by the user and the server key for that particular user. Moreover, each

secret credential of the user is encrypted with a different key, and it is computationally hard for the adversary to decrypt the hash values without knowing the mutually reproducible secret key. Therefore, the adversary cannot decrypt the hash values even if the password database is leaked.

2. Shuffle and hash function

We propose a shuffle and hash function (SHF) to produce unique hash values for implementing PassPro. The key idea behind shuffling two strings with a secret context is to ensure that the secret context remains concealed and is not exposed, published, or transmitted to the server either directly or indirectly. The SHF is a function that shuffles two strings based on a secret context. The SHF is used to create a unique hash value. The SHF has three key applications, which are highlighted below-

(a) Unique hash value: The SHF can be used to generate a unique hash value by shuffling two publicly known words using a secret context.
(b) Key generation: The SHF can be used to generate a key using a password and a secret context.
(c) Password stretching: The low-entropy password can be converted into a high-entropy password using SHF.
(d) Password hashing: The SHF can be used to hash the password on the server to secure the password database.
(e) Keyed-hash function: The SHF can also be used to implement a keyed hash function.
(f) Slow hash function: The SHF can be tuned to make it a slower hash function.

The SHF relies on SHA and shuffle functions. The SHF works as follows:

(i) The SHF hashes the input strings using SHA2 to make the running time independent of the input string size.
(ii) The SHF shuffles the hash values using secret context to produce a unique shuffled word as defined in Definition 1. The shuffling process does not mix the context with the shuffled word.
(iii) The shuffled word is hashed using SHA2.

The above simple steps are used to compute a unique hash value by shuffling two words using a secret context. The SHF is simple yet powerful, and it can be applied to diverse applications, such as key generation and authentication, as mentioned above.

Definition 1. *We define the shuffle function as* $\textsc{Shuffle}(\omega_1, \omega_2, ctx, seed)$ *where* ω_1 *and* ω_2 *are two strings to be shuffled,* ctx *is a secret context,* $seed$ *is a constant. A shuffle function mixes the two strings,* ω_1 *and* ω_2*, by the influence of a secret context* ctx *and produces a shuffled word* $\mathcal{SW}$*. The* $\mathcal{SW}$ *does not contain characters from the context* ctx*.*

Let ω_1 and ω_2 be the two words to be shuffled using a secret context ctx. We define the SHF function as $SHF(\omega_1, \omega_2, ctx, seed)$, which returns a hash value. We convert the words ω_1, ω_2, and ctx into hash values as given in Equation (1).

$$\begin{aligned} \mathcal{H}_{\omega_1} &\leftarrow \textsc{Hash}(\omega_1) \\ \mathcal{H}_{\omega_2} &\leftarrow \textsc{Hash}(\omega_2) \\ \mathcal{H}_{ctx} &\leftarrow \textsc{Hash}(ctx) \end{aligned} \tag{1}$$

Equation (1) makes the SHF algorithm a constant-time algorithm irrespective of the input size. The $\mathcal{H}_{\omega_1}$ is shuffled with another hash value $\mathcal{H}_{\omega_2}$ by using the secret context ctx and produces a shuffled word $\mathcal{SW}$, as shown in Equation (2).

$$\mathcal{SW} \leftarrow \textsc{Shuffle}(\mathcal{H}_{\omega_1}, \mathcal{H}_{\omega_2}, \mathcal{H}_{ctx}, seed) \tag{2}$$

Notably, the $\mathcal{SW}$ does not contain any character of the ctx. Let us consider ω_1 =‘abcdefg’, ω_2=‘tuvwxyz’, ctx =‘KLMNOP’ and seed=‘0x9e3779b9’. Equation (2) produces a shuffled word $\mathcal{SW}$ =‘ztvwgfaebxycud’ for the above-assumed input. In the shuffled word, the $\mathcal{SW}$ does not contain any character of ctx=‘KLMNOP’. The shuffled word is hashed to produce a hash value $\mathscr{R}$, as shown in Equation (3).

$$\mathscr{R} \leftarrow \textsc{Hash}(\mathcal{SW}) \tag{3}$$

The $\mathscr{R}$ is the unique hash value that the client can produce exclusively. It is a deterministic algorithm that the client can consistently reproduce the same output, $\mathscr{R}$, for the correct inputs. Alternatively, the SHF$(\omega_1, \omega_2, ctx, seed)$ can be defined as follows-

$$\begin{aligned} \mathcal{H}_{\omega_1} &\leftarrow \textsc{Hash}(\omega_1), \quad \mathcal{H}_{\omega_2} \leftarrow \textsc{Hash}(\omega_2) \\ \mathcal{H}_{ctx} &\leftarrow \textsc{Hash}(ctx) \\ \mathcal{SW} &\leftarrow \textsc{Shuffle}(\mathcal{H}_{\omega_1}, \mathcal{H}_{\omega_2}, \mathcal{H}_{ctx}, seed) \\ \mathscr{R} &\leftarrow \textsc{Hash}(\mathcal{SW}), \quad Return\ \mathscr{R} \end{aligned} \tag{4}$$

Let us describe the shuffling process of two publicly known words (ω_1 and ω_2) using a secret context, ctx. Let ω_1 = “$Hello$”, ω_2 = “$World$”, ctx = “$MySecretContext$” and $seed = 276337$. Therefore, the hash value of ω_1 using SHA256 is

$$\begin{aligned} \mathcal{H}_{\omega_1} =& 185f8db32271fe25f561a6fc938b2e264306ec304e \\ & da518007d1764826381969 \end{aligned} \tag{5}$$

The hash value of the ω_2 using SHA256 is

$$\begin{aligned} \mathcal{H}_{\omega_2} =& 78ae647dc5544d227130a0682a51e30bc7777fbb6d \\ & 8a8f17007463a3ecd1d524 \end{aligned} \tag{6}$$

We shuffle $\mathcal{H}_{\omega_1}$ and $\mathcal{H}_{\omega_2}$ using the secret context and produce the following by the shuffling process:

$$\begin{aligned} \mathcal{SW} = & e6af768856541412ead813f0d9454a2714cd2fcdb \\ & 835b74023722b2ecf07084d7063231137bc09af80d57680 \\ & 6f57856e84a62c6737a71e0e1d39ba8616213ed5 \end{aligned} \tag{7}$$

The SHF function returns the following hash values of $\mathcal{SW}$ using SHA512 is-

$$\begin{aligned}\mathscr{R} =& f0dd9a1551fc5f6241db729e5aa4e8b8fd10737d52e\\ & ed970b22bb587933e67a20554ebea1fa90389cba6494\\ & 298a8dfe2dcac7823d9e699135e54f2f1505721d6\end{aligned} \tag{8}$$

Now, we change the context to $ctx =$ "$SecretContext$" from $ctx =$ "$myContext@123$", and the values of the ω_1 and ω_2 are unchanged as in the previous example. We shuffle $\mathcal{H}_{\omega_1}$ and $\mathcal{H}_{\omega_2}$ using different ctx and produces significantly different $\mathcal{SW}'$-

$$\begin{aligned}\mathcal{SW}' =& 851d946a4a36103d2288b7b500fa7e7bd3a12f69a\\ & 40fce6ee61282a3507d2ec14f0b230f58d26832947c31a0\\ & 18e1787fd56180c5776676d75c85e24db3641743\end{aligned}$$

There is a huge difference between $\mathcal{SW}$ and $\mathcal{SW}'$ for the same $\mathcal{H}_{\omega_1}$ and $\mathcal{H}_{\omega_2}$ because we change the value of ctx from "$MySecretContext$" to "$SecretContext$". A tiny change in the input changes the shuffled word dramatically because of the avalanche effect of the shuffle function, as shown in Table 1. Thus, the SHF function returns the hash value as given below-

$$\begin{aligned}\mathscr{R}' =& 3094ee1aff8c2a99f4fea9dbc4b1c8191070608d84d9\\ & 092569f707f0a69709181ab1df5bd7b1cfb58c53797c\\ & 7a70ff0b5917b6b0f1017108d5541652c1dd2578\end{aligned}$$

The $\mathscr{R}$ and $\mathscr{R}'$ are not equal because a minor change can show an avalanche effect due to SHF. Therefore, the SHF function exhibits the following properties-

$$Output = \begin{cases} \mathscr{R} \neq \mathscr{R}' & if\ \mathscr{R} = \text{SHF}(\omega_1, \omega_2, ctx, seed) \\ & \quad and\ \mathscr{R}' = \text{SHF}(\omega_1', \omega_2, ctx, seed) \\ \mathscr{R} \neq \mathscr{R}' & if\ \mathscr{R} = \text{SHF}(\omega_1, \omega_2, ctx, seed) \\ & \quad and\ \mathcal{R}' = \text{SHF}(\omega_1, \omega_2', ctx, seed) \\ \mathscr{R} \neq \mathscr{R}' & if\ \mathscr{R} = \text{SHF}(\omega_1, \omega_2, ctx, seed) \\ & \quad and\ \mathscr{R}' = \text{SHF}(\omega_1, \omega_2, ctx', seed) \end{cases} \tag{9}$$

Equation (9) shows that any change in the input changes the output dramatically because of the SHF functions. Table 2 shows the SHF outputs after changing their values. We change the value of ω_1, ω_2, and ctx to demonstrate the change. In Test Case 1, we input $\omega_1 =$ "$Hello''$, $\omega_2 =$ "$World'$, $ctx =$ "$MySecretContext'$ and $seed = 276337$ to SHF, and the SHF produces an output which is given in Table 2. We change the value of the context as $ctx = SecretContext$ in Test Case 2 in Table 2. Thus, Test Case 1 and Test Case 2 produce different outputs. Similarly, Test Case 1 and Test Case 3 produce different outputs because of the different values of ω_2. Moreover, Test Case 1 and Test Case 4 produce different hash values because of the different values of ω_1.

Let ω_1 and ω_2 be two publicly known words. We know that the context is a secret word. The client converts the words into hash values using the SHF function as $\mathscr{R} = SHF(\omega_1, \omega_2, ctx, seed)$. The SHF function shuffles the hash values of the ω_1 and ω_2 using ctx and produces $\mathcal{SW}$ as shown in Table 1. The adversary cannot reproduce the $\mathcal{SW}$ since the adversary does not know about the secret context. For the same ω_1 and ω_2, the shuffled word $\mathcal{SW}$ is different for the different ctx. The $\mathcal{SW}$ is hard for the adversary to reconstruct because of the secret context, even if we use two publicly known words (ω_1 and ω_2). The secret context influences the shuffling process, but the shuffled word $\mathcal{SW}$ does not shuffle the secret context ctx. The adversary does not know how to reproduce the shuffled word because of the secret context. So, it is computationally infeasible to reconstruct the shuffled word of the two publicly known strings ω_1 and ω_2 due to the secret context.

Let us assume the hash algorithm is SHA512, so the hash values are 512-bit long and 128 characters in hexadecimal representation. Therefore, the length of the two strings is 256 in the shuffling process. The probability of selecting a character correctly from either $\mathcal{H}_{\omega_1}$ or $\mathcal{H}_{\omega_2}$ to place a character at $\mathcal{SW}$ is $\frac{1}{256}$. Similarly, the probability of selecting another character correctly from either $\mathcal{H}_{\omega_1}$ or $\mathcal{H}_{\omega_2}$ to place a character at $\mathcal{SW}$ is $\frac{1}{255}$ and so on. The total probability is

$$\frac{1}{256 \times 255 \times 254 \times \ldots \times 1} = \frac{1}{256!} \approx 0 \tag{10}$$

Alternatively, we can form 256! different words using a string length of 256. The probability of selecting a correct word from 256! is $\frac{1}{256!}$. The reproduction of a shuffled word without knowing the context is computationally hard.

2.1. Implementation of SHF

The SHF is dependent on the shuffle function and SHA as shown in Algorithm 1. Initially, the SHF function converts the input string into a hash value using SHA2. This process ensures constant-time execution of the SHF, which is independent of the input string. Thus, the SHF computes the hash values of the input string ω_1, ω_2, ctx using SHA256. The shuffling process produces a shuffled word $\mathcal{SW}$, and the shuffled word $\mathcal{SW}$ is converted into a hash value using SHA512.

Algorithm 1 Shuffle and hash function.

procedure SHF($\omega_1, \omega_2, ctx, seed$)
  $\mathcal{H}_{\omega_1} = \text{HASH}(\omega_1)$
  $\mathcal{H}_{\omega_2} = \text{HASH}(\omega_2)$
  $\mathcal{H}_{ctx} = \text{HASH}(ctx)$
  $\mathcal{SW} = \text{SHUFFLE}(\mathcal{H}_{\omega_1}, \mathcal{H}_{\omega_2}, \mathcal{H}_{ctx}, seed)$
  $\mathscr{R} = \text{HASH}(\mathcal{SW})$, **return** $\mathscr{R}$
end procedure

The SHF shuffles two strings to produce a unique word using a secret context. The shuffling process produces a unique shuffled word $\mathcal{SW}$ as shown in Algorithm 2. The shuffling process computes an integer value using a function $\text{GETVALUE}(\omega_1, \omega_2, ctx, seed)$. The integer value is used to compute the index of a character to be copied from the string. It copies a character from either ω_1 or ω_2 based

TABLE 1. THE PRODUCED SHUFFLED WORD $\mathcal{SW}$ BY THE SHUFFLE FUNCTION FOR VARIOUS TEST CASES.

Test case	ω_1	ω_2	Secret context ctx	Output of the shufle function ($\mathcal{SW}$)
1	Hello	World	MySecretContext	e6af768856541412ead813f0d9454a2714cd2fcdb835b74023722b2ecf07084d7063231137b c09af80d576806f57856e84a62c6737a71e0e1d39ba8616213ed5
2	Hello	World	SecretContext	851d946a4a36103d2288b7b500fa7e7bd3a12f69a40fce6ee61282a3507d2ec14f0b230f58d 26832947c31a018e1787fd56180c5776676d75c85e24db3641743
3	Hello	Universe	MySecretContext	7e20af37b2c3f686277796b121eebac58f9d54ffaf151826186f647fc6187c34707851e2648 0bc128663d3f2ddf3863180840757d585101c8532e5a6c4ed8029
4	Hi	World	MySecretContext	3fe094b501de73af1291128c7358a236a2bccb3ab14d661aa70749f8d866c5e2cf874bc093d 5057c8d73bee0c8219727ab272081fb97a5a383610ede7dd04632

TABLE 2. THE OUTPUT OF THE SHF FUNCTION FOR VARIOUS TEST CASES.

Test case	ω_1	ω_2	Secret context cts	Output of the SHF function ($\mathcal{R}$)
1	Hello	World	MySecretContext	f0dd9a1551fc5f6241db729e5aa4e8b8fd10737d52eed970b22bb587933e67a20554ebea1fa 90389cba6494298a8dfe2dcac7823d9e699135e54f2f1505721d6
2	Hello	World	SecretContext	3094ee1aff8c2a99f4fea9dbc4b1c8191070608d84d9092569f707f0a69709181ab1df5bd7b 1cfb58c53797c7a70ff0b5917b6b0f1017108d5541652c1dd2578
3	Hello	Universe	MySecretContext	c4cb9b4bb7fa61bf40bceefb6ec8528e708c2d83c100dca9e62cb6518af79e5604ea366cf3e 48678f8e55212cbbbb0d85025178fe2f8275cd5a4cf53ec021330
4	Hi	World	MySecretContext	953a4c0bbd04d81ac905a0f833eafd9d3a5ed0931acf3cdb7f2a9d7624227c47ec64450e9c7 79c824790c50d3f575edbe4b01de2bd164235b3224a492cd0af08

Algorithm 2 Shuffling two strings, ω_1 and ω_2, using a secret context ctx.

procedure SHUFFLE($\omega_1, \omega_2, ctx, seed$)
  $l_1 = Length(\omega_1), \quad l_2 = Length(\omega_2)$
  $s_1 = \omega_1, \quad s_2 = \omega_2$
  $seed =$ GETVALUE($ctx, \omega_2, \omega_1, seed$)
  while $l1 > 0 \ \ and \ \ l2 > 0$ **do**
    $seed =$ GETVALUE($ctx, \omega_2, \omega_1, seed$)
    if $seed \ \& \ 1$ **then**
      $seed =$ GETVALUE($\omega_1, \omega_2, ctx, seed$)
      $\mathcal{SW} += s_1[seed \ \% \ l_1]$
      Remove the copied character from s_1
      $l_1 = l_1 - 1$
    else
      $seed =$ GETVALUE($\omega_2, \omega_1, ctx, seed$)
      $\mathcal{SW} += s_2[seed \ \% \ l_2]$
      Remove the copied character from s_2
      $l_2 = l_2 - 1$
    end if
  end while
  $\mathcal{SW} += s1$
  $\mathcal{SW} += s2$
  return $\mathcal{SW}$
end procedure

on the computed integer value. Alternatively, the characters are copied from ω_1 and ω_2 to form the $\mathcal{SW}$ using the computed integer value by GETVALUE($\omega_1, \omega_2, ctx, seed$). The GETVALUE($\omega_1, \omega_2, ctx, seed$) depends on Murmur hash function [17] as shown in Algorithm 3. The GETVALUE($\omega_1, \omega_2, ctx, seed$) computes an integer value by iterating η times where $\eta = 3 \ or \ 5$. It alters the seed value in each function call, and finally, it returns the computed integer value. We can tune the value of η in Algorithm 3 to slow down the SHF.

Algorithm 3 Computing an integer value using ω_1, ω_2, and ctx.

procedure GETVALUE($\omega_1, \omega_2, ctx, seed$)
  for $i : \ 1 \ to \ \eta$ **do** ▷ Default value of η is 3.
    $seed =$ MURMUR($\omega_1, length(\omega_1), seed$)
    $seed =$ MURMUR($\omega_2, length(\omega_2), seed$)
    $seed =$ MURMUR($ctx, length(ctx), seed$)
  end for
  return $seed$
end procedure

3. PassPro: The proposed system

The PassPro has two phases: registration and authentication. Additionally, the PassPro is classified into two key categories based on its implementation: PassPro with Encryption (PassProE) and PassPro with SHF (PassProS). These proposed systems rely on existing CAPTCHA to differentiate between robots and humans. Moreover, it depends on three correct password attempt mechanisms. Furthermore, it relies on existing cryptography techniques (SSL), such as the POST method, to send the four hash values to the server.

3.1. Registration

In a conventional password-based authentication system, the user enters a user ID and password to prove their authenticity. The user's raw password is transmitted to the server, and the server hashes the password with salt. Modern password hashing algorithms employ a memory-hard hashing technique to thwart various attacks, such as Argon2i.

3.1.1. Registration of PassProE. Figure 1 demonstrates the registration process of PassPro using symmetric key encryption. The user registers their secret credentials at the server by entering the user ID, a password, and a secret key

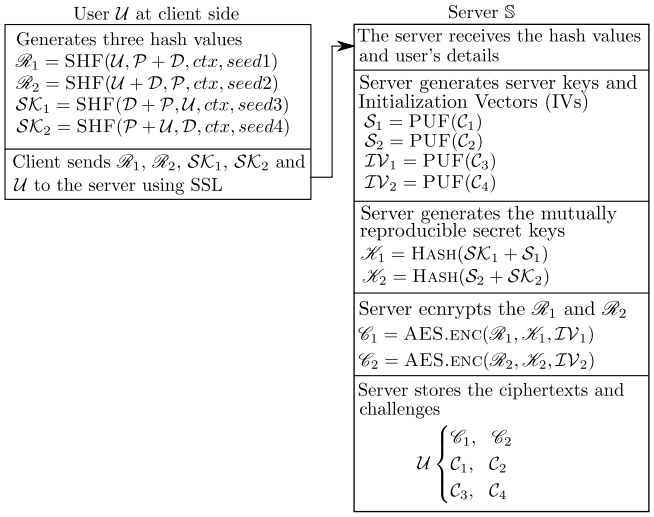


Figure 1. PassProE: Registration process of PassPro using symmetric key cryptography.

as shown in Table 3. Let the user ID be $\mathcal{U}$, the password be $\mathcal{P}$, the secret context be ctx, and the domain name be $\mathcal{D}$. The client device performs SHF to produce four hash values, namely, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, and $\mathcal{SK}_2$ using Equation (11).

$$\begin{aligned} \mathscr{R}_1 &= \text{SHF}(\mathcal{U}, \mathcal{P}+\mathcal{D}, ctx, seed1) \\ \mathscr{R}_2 &= \text{SHF}(\mathcal{U}+\mathcal{D}, \mathcal{P}, ctx, seed2) \\ \mathcal{SK}_1 &= \text{SHF}(\mathcal{D}+\mathcal{P}, \mathcal{U}, ctx, seed3) \\ \mathcal{SK}_2 &= \text{SHF}(\mathcal{P}+\mathcal{U}, \mathcal{D}, ctx, seed4) \end{aligned} \tag{11}$$

The $\mathscr{R}_1$, and $\mathscr{R}_2$ are transmitted to the server to store in the database by encrypting with mutually reproducible secret key. The $\mathcal{SK}_1$ and $\mathcal{SK}_2$ are transmitted to the server to produce the mutually reproducible secret keys. The client sends $\mathcal{U}$, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, and $\mathcal{SK}_2$ to the server using existing cryptography techniques such as SSL. Also, the client sends the user's details for registration.

The server receives the $\mathcal{U}$, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$ and $\mathcal{SK}_2$ as shown in Table 4 and 5. The server generates two server keys using PUF; let the server keys be $\mathcal{S}_1$ and $\mathcal{S}_2$, as shown in Equation 12. Additionally, the server requires two random Initialization Vectors (IVs) for encryption using AES; let these be denoted as $\mathcal{IV}_1$ and $\mathcal{IV}_2$. Each user has a unique set of server keys and initialization vectors (IVs). The server generates the server keys and IVs using the PUF according to Equation (12).

$$\begin{aligned} \mathcal{S}_1 &= \text{PUF}(\mathcal{C}_1), & \mathcal{S}_2 &= \text{PUF}(\mathcal{C}_2) \\ \mathcal{IV}_1 &= \text{PUF}(\mathcal{C}_3), & \mathcal{IV}_2 &= \text{PUF}(\mathcal{C}_4) \end{aligned} \tag{12}$$

where the challenges are stored in the password database on the server against the user ID $\mathcal{U}$. The server generates two mutually reproducible secret keys and lets the keys be $\mathcal{K}_1$ and $\mathcal{K}_2$. The server generates the two mutually reproducible secret keys using Equation (13).

$$\begin{aligned} \mathscr{K}_1 &= \text{HASH}(\mathcal{SK}_1 + \mathcal{S}_1) \\ \mathscr{K}_2 &= \text{HASH}(\mathcal{S}_2 + \mathcal{SK}_2) \end{aligned} \tag{13}$$

The server encrypts the received secret credentials, $\mathscr{R}_1$ and $\mathscr{R}_2$, using Equation (14).

$$\begin{aligned} \mathscr{C}_1 &= \text{AES.ENC}(\mathscr{R}_1, \mathscr{K}_1, \mathcal{IV}_1) \\ \mathscr{C}_2 &= \text{AES.ENC}(\mathscr{R}_2, \mathscr{K}_2, \mathcal{IV}_2) \end{aligned} \tag{14}$$

The server stores the ciphertexts $\mathscr{C}_1$ and $\mathscr{C}_2$ in the password database as shown in Equation (15). Each user has different and unique $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{IV}_1$ and $\mathcal{IV}_2$. The server securely deletes $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{IV}_1$, $\mathcal{IV}_2$, $\mathcal{K}_1$, and $\mathcal{K}_2$ after the insertion of the ciphertexts into the password database.

$$\mathcal{U}\begin{cases} \mathscr{C}_1, & \mathscr{C}_2 \\ \mathcal{C}_1, & \mathcal{C}_2, \quad \mathcal{C}_3, \quad \mathcal{C}_4 \end{cases} \tag{15}$$

The ciphertexts are indexed against user ID $\mathcal{U}$. Each user has different secret keys, server keys, and IVs.

Conventionally, passwords are hashed on the server using a traditional password hashing algorithm or a memory-hard hash function with a salt. On the contrary, PassProE encrypts the secret credentials of the user on the server side. Let us consider four test cases for a single user ID and four different scenarios of the user as shown in Table 3. The first rows of the Table 3, 4, and 5 are demonstrated for a single test case, which is test case 1. Note that the user ID is unique in the password database, but we have demonstrated different credentials for a single user ID for a better understanding of PassPro. The user enters the secret credentials at the client devices as shown in Table 3. These secret credentials are converted into four hash values using SHF as shown in Equation (11). The four hash values, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$ and $\mathcal{SK}_2$ (shown in Table 4 and 5) are sent to the server. The server generates random and unique server keys $\mathcal{S}_1$ and $\mathcal{S}_2$ using PUF. The server merges $\mathcal{SK}_1$ with $\mathcal{S}_1$ and $\mathcal{SK}_2$ with $\mathcal{S}_2$ to produce the mutually reproducible secret keys $\mathcal{K}_1$ and $\mathcal{K}_2$. The $\mathcal{K}_1$ and $\mathcal{K}_2$ are used to encrypt $\mathscr{R}_1$, $\mathscr{R}_2$. The resultant ciphertexts are stored in the password database. The $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{K}_1$, and $\mathcal{K}_2$ are securely deleted after the insertion of the completion of the operations.

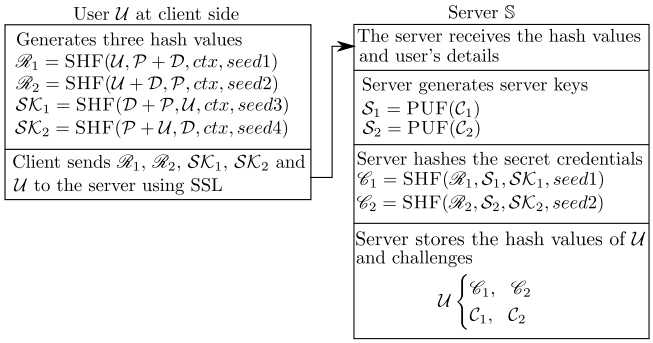


Figure 2. PassProS: Registration process of PassPro using SHF.

3.1.2. Registration of PassProS. The second method utilizes SHF to store the user's secret credentials in the password database, as illustrated in Figure 2. The user sends $\mathcal{U}$, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, and $\mathcal{SK}_2$ to the server and the server receives

TABLE 3. PASSPROE: SECRET CREDENTIALS OF USER AT CLIENT DEVICE

Test case	User ID $\mathcal{U}$	Password $\mathcal{P}$	Context ctx	Domain $\mathcal{D}$
1	mymail@example.com	MyPassWord#123$	MySecretCtx#123$	mydomain.com
2	mymail@example.com	PassWord#123$	MySecretCtx#123$	mydomain.com
3	mymail@example.com	MyPassWord#123$	SecretCtx#123$	mydomain.com
4	mymail@example.com	MyPassWord#123$	MySecretCtx#123$	mydomain.org

TABLE 4. PASSPROE: THE CLIENT COMPUTES $\mathscr{R}_1$ AND $\mathscr{R}_2$ TO SEND THE SERVER.

Test case	$\mathscr{R}_1$	$\mathscr{R}_2$
1	7f276d5f69b77b0abbebf05f7946fa8c880f7a5dac875d64bed38155f ad98ece808f79b56444ed42d6ad6514ec0635d1348e66ad1690ab6c3 19839362a96004d	d0a35e5345e882dcfcac9790faa759c3cbeb92acef546f93a3cfbbcef4 a42f4c91f99430b6976fa815facdc573e5b135d4b04a56c8ba23e386 a19e059b52b632
2	8e4b2e60897567ed1b27b5542748e78d90cbbbc2ebfb5988fe06ec60 656023acacf3c6ce55b48b38f52075e3b8f93b91f472e042ddfcf8f596 5e72313ce462dc	e11161ffe3875a36fc140dac7caf3fdd805242c91a2d476e83f74972 126f34f9af8ef8091827bf91422141278c677c51a01a833c91981700 9b6bbf27c8bc3e74
3	74b5fe98bc51f1abc157ae58f46d3f12598d72f956ec762f870457851 3b32cea44a53b88c598ca5b4349a3a9f15bd59bef9d2733c7d135293 6082d4ee072b4a4	9017a21c9d34127f69fbfedf520027516c35c47447b43cbf6baf1f9b e09116d43f6de5b2875d5074b2fe831501ae36418c2c3e4e189c5f3 6d2d43b89b11c1aa0
4	3cb49048753678beea327b7e3afb583b8bce3fde990c21841c3ddafc 5c3891992408b7bd11c7df15314191505d4c313e148bad790ad6d61 e2f20138c1496fe6f	0f4c6c4cad0ea9d7493cc9b220ff368ccc5d06b0cf33788ef0a2d5b 706aa5736cece765f3ad0c38c03d28caccd6fe3b83a65a55b7ebf2 b9a16f3704ddf3196bb

TABLE 5. PASSPROE: THE CLIENT COMPUTES $\mathcal{SK}_1$ AND $\mathcal{SK}_2$ TO SEND THE SERVER.

Test case	$\mathcal{SK}_1$	$\mathcal{SK}_2$
1	2ae724f169a508815be33afe335b7a9e92e510292ef8767e71bdacd bc864aae0bd2e878ccbe4fe9603470184fa0ac42189730235e63b34 bdb0342061a33b8091	163b217c8c41d8c35c5c2ae094bf1dffbbf50a95024b05ed029f5dc95 7262571a4599d952c52d55ea02872a2ea657263ca2a620d03945d5f b7f9868c0e926ba8
2	eb50044ecf2387aa619108962598103159af5f6bde317c6588421bc bfc1426f088b41cebc221564fbc62622fdcc3e4fa23204784f5735e1 f9877ed0ebe1965c8	eb50044ecf2387aa619108962598103159af5f6bde317c6588421bcb fc1426f088b41cebc221564fbc62622fdcc3e4fa23204784f5735e1f9 877ed0ebe1965c8
3	edb4d23c243e6443958eef09b79289c41fb8e22563678ff951a41f0 a6f8822fe766de64adce65317008e04f40581ace5dc8efa2420bc5f bd2f18fbe788010c2f	686aea855aa34cb2447be4bb0b503a6aa3e4a42ae7901d5711197c7 bc390ec501a4b7d014dbb0321484e7323eb67c3269c37ce6602282 5c734828a062701768c
4	176eaccf36981a6a353aa8a720af65f064bb3a82af0d4cc3ee089a8 f55300da5c5617d87439bf6d817b93114bfdd5147270791df1360 6742cbd531074aa70ad9	775ccf23e00b916a4b942eed919eb76fc8b817e06bd774a4a1d5e11 2d66b5700e53e7d49295375ad0652f52f57a6f0791b87bd301818e bc5d3ebb54a3cdd158a

these data. The server computes the server keys $\mathcal{S}_1$ and $\mathcal{S}_2$ using PUF, as shown in Equation (12), but it does not require $\mathcal{IV}_1$ and $\mathcal{IV}_2$. The server performs hashing using SHF as follows-

$$\begin{aligned} \mathscr{C}_1 &= \text{SHF}(\mathscr{R}_1, \mathcal{S}_1, \mathcal{SK}_1, seed_1) \\ \mathscr{C}_2 &= \text{SHF}(\mathscr{R}_2, \mathcal{S}_2, \mathcal{SK}_2, seed_2) \end{aligned} \tag{16}$$

The server stores the $\mathscr{C}_1$ and $\mathscr{C}_2$ in the password database indexed against user ID $\mathcal{U}$, as shown in Equation (17). The server securely deletes $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$, and $\mathcal{S}_2$ after successfully inserting the secret credentials into the password database.

$$\mathcal{U}\begin{cases} \mathscr{C}_1, & \mathscr{C}_2 \\ \mathcal{C}_1, & \mathcal{C}_2 \end{cases} \tag{17}$$

Table 3, 4, and 5 will not change, and these are the same for the PassProE and PassProS if the inputs are the same. These values are reproducible for the same input. The server uses the $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$ to hash the credentials using SHF, and produces $\mathscr{C}_1$ and $\mathscr{C}_2$. The $\mathscr{C}_1$ and $\mathscr{C}_2$ are inserted into the password database, as shown in (17).

3.2. Authentication process

The authentication process also has two methods similar to the registration phase, namely, PassProE and PassProS. We discuss the authentication process of PassProE and then PassProS in the next sub-subsections.

3.2.1. Authentication of PassProE. Figure 3 demonstrates the authentication process using symmetric key cryptography. The user inputs the user ID (email ID), password, and secret context. The user device computes $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, and $\mathcal{SK}_2$ using the SHF with the user ID, password, secret context, and domain word. The client sends the $\mathcal{U}$, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, and $\mathcal{SK}_2$ to the server to prove its authenticity. The transmission process must use a conventional encryption process such as SSL. The server receives $\mathcal{U}$, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, and $\mathcal{SK}_2$. The server retrieve the challenges, $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ and $\mathcal{C}_4$ from the password database. The server reproduces the server keys and IVs for the user ID $\mathcal{U}$ using Equation (12). The server merges the server keys with user shared secret keys ($\mathcal{S}_1$ with $\mathcal{SK}_1$ and $\mathcal{S}_2$ with $\mathcal{SK}_2$) to produce mutually reproducible secret keys $\mathscr{K}_1$ and $\mathscr{K}_2$ using Equation (13). The server encrypts the $\mathscr{R}_1$ and $\mathscr{R}_2$ using the $\mathscr{K}_1$, $\mathscr{K}_2$, $\mathcal{IV}_1$

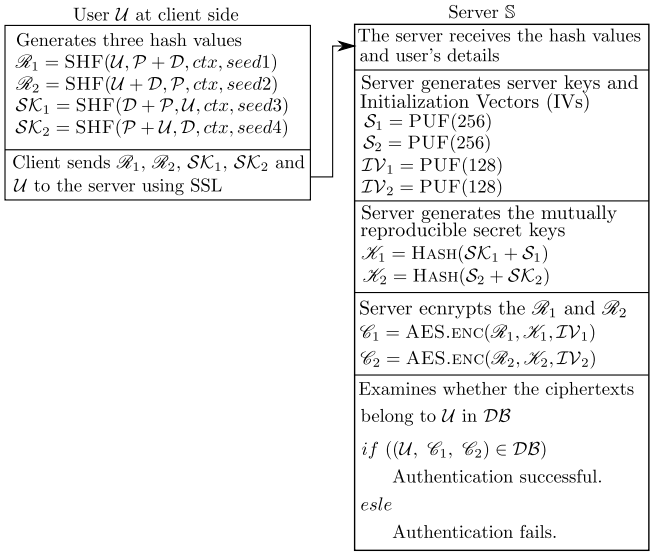


Figure 3. Authentication process of PassProE using symmetric key cryptography.

and $\mathcal{IV}_2$ using Equation (14). The server also retrieves the ciphertexts of the user ID $\mathcal{U}$ from the server as follows-

$$\begin{aligned} \mathscr{C}_1 &= \text{RETRIEVE}(\mathscr{C}_1) \\ \mathscr{C}_2 &= \text{RETRIEVE}(\mathscr{C}_2) \end{aligned} \tag{18}$$

The server decrypts the hash values from the stored ciphertexts as follows-

$$\begin{aligned} \mathscr{R}_1^{dec} &= \text{AES.DEC}(\mathscr{C}_1, \mathscr{K}_1, \mathcal{IV}_1) \\ \mathscr{R}_2^{dec} &= \text{AES.DEC}(\mathscr{C}_2, \mathscr{K}_2, \mathcal{IV}_2) \end{aligned} \tag{19}$$

The server compares the decrypted hash values and the received hash values for authentication as follows-

$$\begin{aligned} &if\ \mathcal{U} \in \mathcal{DB}\ and\ \mathscr{R}_1^{dec} = \mathscr{R}_1\ and\ \mathscr{R}_2^{dec} = \mathscr{R}_2^{dec} \\ &\quad \text{Authentication is successful.} \\ &else \\ &\quad \text{Authentication is failure.} \end{aligned} \tag{20}$$

We can also verify the authenticity of the user using the encrypted ciphertexts without decrypting them. In this alternative method, the server produces the ciphertexts $\mathscr{C}_1$ and $\mathscr{C}_2$. The server examines whether the ciphertexts belong to the password database against the user ID $\mathcal{U}$ using the following equations-

$$\begin{aligned} &if\ ((\mathcal{U},\ \mathscr{C}_1,\ \mathscr{C}_2) \in \mathcal{DB}) \\ &\quad \text{Authentication successful.} \\ &esle \\ &\quad \text{Authentication fails.} \end{aligned} \tag{21}$$

If the ciphertexts are indexed in the password database against user ID $\mathcal{U}$, the user is authenticated; otherwise, the authentication fails. Thus, PassProE uses symmetric key encryption to secure the password database instead of hashing the password.

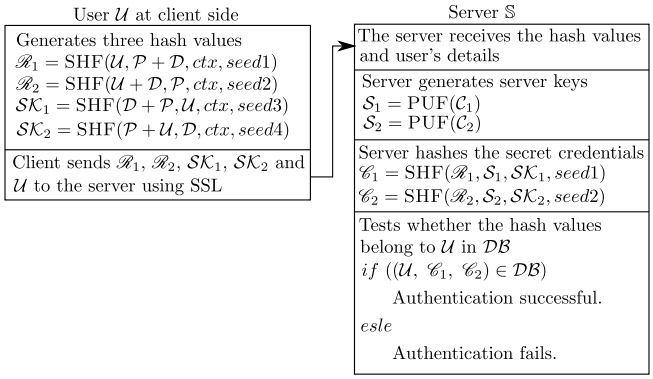


Figure 4. Authentication process of PassPro using SHF.

3.2.2. Authentication of PassProS. Figure 4 demonstrates the authentication mechanism of PassProS. It uses SHF instead of symmetric key cryptography. The client sends the $\mathcal{U}$, $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, and $\mathcal{SK}_2$ to server to prove its authenticity. The server reproduces the server keys using Equation (12), but it does not require the IVs. The server generates the hash values $\mathscr{C}_1$ and $\mathscr{C}_2$ using the $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$ using SHF as shown in Equation (16). The server examines the authenticity of the user using $\mathcal{U}$, $\mathscr{C}_1$, and $\mathscr{C}_2$ as shown in Equation (21).

4. Prevention of the domino effect

Conventionally, the adversary tries to derive passwords from the leaked hash values of the passwords to impersonate other domains only if the user reuses the password in other domains. The adversary can impersonate other domains if the adversary can derive a password from the leaked hash values. A study estimates that about 43-51% of users reuse passwords in multiple domains [14]. PassPro deals with hash values with a shuffled user ID, password, and domain word, which encourages password reuse. Our proposed approach is similar to Ross *et al.* [13]. PassPro deals with hash values with a shuffled user ID, password, and domain word. Therefore, it is computationally infeasible to reconstruct the original string from the hash values $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$ and $\mathcal{SK}_2$ if the adversary does not know the input strings except the domain word. Moreover, the $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$ and $\mathcal{SK}_2$ are transmitted to the server with SSL. Table 3, 4, and 5 show that the $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$ and $\mathcal{SK}_2$ are different for different password, secret context and domain name. Let the two domains be the $\mathcal{D}$ and $\mathcal{D}'$. The SHF produces the $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$ and $\mathcal{SK}_2$ for the domain name $\mathcal{D}$ as shown in Equation (11). The SHF produces the $\mathscr{R}'_1$, $\mathscr{R}'_2$, $\mathcal{SK}'_1$ and $\mathcal{SK}'_2$ for the domain name $\mathcal{D}'$ as shown in Equation (22).

$$\begin{aligned} \mathscr{R}'_1 &= \text{SHF}(\mathcal{U}, \mathcal{P}+\mathcal{D}', ctx, seed1) \\ \mathscr{R}'_2 &= \text{SHF}(\mathcal{U}+\mathcal{D}', \mathcal{P}, ctx, seed2) \\ \mathcal{SK}'_1 &= \text{SHF}(\mathcal{D}+\mathcal{P}, \mathcal{U}, ctx, seed3) \\ \mathcal{SK}'_2 &= \text{SHF}(\mathcal{P}+\mathcal{U}, \mathcal{D}', ctx, seed4) \end{aligned} \tag{22}$$

From Equations (11) and (22), we can draw the following equation-

$$\begin{aligned} \mathscr{R}_1 \neq \mathscr{R}'_1, \qquad & \mathscr{R}_2 \neq \mathscr{R}'_2 \\ \mathcal{SK}_1 \neq \mathcal{SK}'_1, \qquad & \mathcal{SK}_2 \neq \mathcal{SK}'_2 \end{aligned} \tag{23}$$

Equation (23) shows that the SHF always creates different hash values for the different domains for the same user ID, password, and secret context. Thus, a user can reuse the password and secret context in a different domain.

5. Phishing Attacks

Conventionally, the passwords are transmitted to the server for hashing. Therefore, the phishing attacker can clone a legitimate website to deceive users, and the user enters their secret credentials. These secret credentials are transmitted to the adversary's server, enabling the adversary to obtain the user's secret credentials easily. Let us assume the adversary clones a website of x and let the legitimate domain name be the $\mathcal{D}$ and the adversary's domain name be $\mathcal{D}'$. Conventionally, if the user enters the secret credentials, the adversary can get them in raw form. So, PassPro creates hash values with shuffling with a domain name similar to the Ross *et al.* [13]. Similar to the Ross *et al.* [13], PassPro always creates different hash values as shown in Equation (11), (22), and (23). Thus, the adversary cannot use the $\mathscr{R}'_1$, $\mathscr{R}'_2$, $\mathcal{SK}'_1$ and $\mathcal{SK}'_2$ to gain authentication in other domains. Moreover, the adversary cannot derive the original password and secret context from the $\mathscr{R}'_1$, $\mathscr{R}'_2$, $\mathcal{SK}'_1$, and $\mathcal{SK}'_2$.

6. Password database leakage

Conventionally, the users' credentials are stored on the server by hashing. The server heavily depends on the salt and slow hash algorithm to secure the password. However, the password database can also be leaked along with the salt. So, we assume that everything can be leaked from the server. This assumption leads to a client-controlled password database. The client generates a key using SHF, and the server generates a key using PUF. A recent development suggests that the password database can be encrypted using a key generated by PUF [18]. 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. Moreover, the noise can be removed using the fuzzy extractor [19]. However, recent developments suggest that the PUF can be used without the need for a fuzzy extractor [20]. Notably, the damage to the PUF circuit causes an issue; therefore, it requires a backup server with a different PUF [18]. The PUF can malfunction or fail at any time. Therefore, an active backup of the password database is required, where modification of the main password database triggers a modification in the backup server. There are two reasons to modify the password backup database: a) inserting a new user, and b) forgotten passwords. The registration of the user triggers the registration of both the main site and the backup site. Similarly, the forgotten passwords must be reset on both the main site and the backup site. These operations are not frequent, and a backup server can easily be maintained.

We have two approaches: PassProE and PassProS. Let us consider the PassProE. The PassProE encrypts the secret credentials using mutually reproducible secret keys ($\mathscr{K}_1$ and $\mathscr{K}_2$). The $\mathscr{K}_1$ and $\mathscr{K}_2$ can be produced by the secret of the user ($\mathcal{SK}_1$ and $\mathcal{SK}_2$) and server keys ($\mathcal{S}_1$ and $\mathcal{S}_2$). The server does not store the $\mathscr{K}_1$, $\mathscr{K}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{IV}_1$ and $\mathcal{IV}_2$ but server stores the ciphertexts $\mathscr{C}_1$, $\mathscr{C}_2$, and challenges in the password database. The adversary cannot decrypt the $\mathscr{R}_1$ and $\mathscr{R}_2$ from the $\mathscr{C}_1$ and $\mathscr{C}_2$ without the $\mathscr{K}_1$, $\mathscr{K}_2$, $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{IV}_1$ and $\mathcal{IV}_2$. Therefore, it is safe even if the password database is leaked.

Let us consider the PassProS. It uses SHF to hash the password database. To hash the secret credentials, the server requires $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$. After the successful hashing, the server securely deletes the $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$ and server stores the hash value $\mathscr{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_1$ and $\mathcal{C}_2$. So, the adversary needs to $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$ gain authentication from the leaked password database.

TABLE 6. ADVERSARY'S REQUIREMENT FROM THE LEAKED DATABASE.

Features	**PassProE**	**PassProS**
Secrets cannot be leaked	$\mathscr{K}_1$, $\mathscr{K}_2$, $\mathcal{IV}_1$ and $\mathcal{IV}_2$ (To derive $\mathscr{K}_1$, $\mathscr{K}_2$, the adversary needs $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$, $\mathcal{S}_2$)	$\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$
Secret can be leaked	$\mathscr{C}_1$, $\mathscr{C}_2$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ and $\mathcal{C}_4$	$\mathscr{C}_1$, $\mathscr{C}_2$, $\mathcal{C}_1$ and $\mathcal{C}_2$
Reversibility	Yes	No
Collision attack	On secrets credentials but infeasible	On secrets credentials but infeasible
Dictionary attack	No	No
Guessing attack	No	No
Precompuation attack	No	No
Rainbow table attack	No	No
Phishing attack	No	No
Domino effect	No	No

Table 6 compares the PassProE and PassProS. In PassProE, the adversary does not need $\mathscr{R}_1$ and $\mathscr{R}_2$ from the leaked database. The adversary can decrypt the hash values from the ciphertext if the adversary can derive $\mathscr{K}_1$, $\mathscr{K}_2$, $\mathcal{IV}_1$, and $\mathcal{IV}_2$. Alternatively, the security strength of PassProE depends on $\mathscr{K}_1$, $\mathscr{K}_2$, $\mathcal{IV}_1$ and $\mathcal{IV}_2$. To derive $\mathscr{K}_1$ and $\mathscr{K}_2$, the adversary need $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$. Let us assume that $\mathscr{C}_1$ and $\mathscr{C}_2$ are leaked along with the challenges. The adversary needs the $\mathscr{K}_1$, $\mathscr{K}_2$, $\mathcal{IV}_1$, and $\mathcal{IV}_2$ to decrypt $\mathscr{R}_1$ and $\mathscr{R}_2$ from the ciphertexts $\mathscr{C}_1$ and $\mathscr{C}_2$ even if the adversary has the correct password and secret context. The server cannot decrypt $\mathscr{R}_1$ and $\mathscr{R}_2$ from the ciphertexts $\mathscr{C}_1$ and $\mathscr{C}_2$ without the user's secret keys. Similarly, the client cannot decrypt the $\mathscr{R}_1$ and $\mathscr{R}_2$ from the ciphertexts $\mathscr{C}_1$ and $\mathscr{C}_2$ without the $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{IV}_1$, and $\mathcal{IV}_2$ even if the user has the correct password and correct secret context.

The PassProS hashes the user's secret credentials using SHF. Therefore, there is no reversal of the $\mathscr{R}_1$ and $\mathscr{R}_2$.

Moreover, the server never stores the $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$ and $\mathcal{SK}_2$. The security strength of the PassProS depends on $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$, $\mathcal{SK}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$ if the password database is leaked. Furthermore, the $\mathcal{SK}_1$ and $\mathcal{SK}_2$ are used as secret contexts, and these are not shuffled with other strings to form $\mathscr{C}_1$ and $\mathscr{C}_2$. The $\mathcal{SK}_1$ and $\mathcal{SK}_2$ are used to influence the shuffling process. Therefore, the password database is safe even if the adversary leaks it. Moreover, the PassProS is stronger than PassProE for the following reasons:

- Security of the PassProS depends on six secret credentials, even if the password database is leaked, whereas the security of the PassProE depends on four secret credentials.
- Adversary cannot revert back the $\mathscr{R}_1$ and $\mathscr{R}_2$ from the $\mathscr{C}_1$ and $\mathscr{C}_2$ whereas it is possible in PassProE.

6.1. Attacks on leaked database

Diverse attacks are performed on the leaked password database to derive the equivalent or original password, including precomputation attacks, rainbow table attacks, dictionary attacks, guessing attacks, and collision attacks. Conventionally, password hashing security relies on salts, which can also be leaked. Therefore, the adversary can perform a dictionary attack or a guessing attack on the leaked password database. However, the parallelism of the adversary can be restricted using Argon2i, where the adversary requires a substantial memory footprint to carry out the attack. On the contrary, PassPro uses a client-controlled password database where the client supplies the secret keys. Moreover, PassPro uses either encryption or SHF to secure the password database. Also, the server uses PUF to generate the server keys and IVs. Therefore, it is infeasible to perform the collision, precomputation, and rainbow table attack on the leaked password database of PassPro.

6.1.1. Collision attack on PassProE. Let us assume that adversary leaked the $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ and $\mathcal{C}_4$ in PassProE. The adversary needs $\mathscr{K}_1$, $\mathscr{K}_2$, $\mathcal{IV}_1$, and $\mathcal{IV}_2$. The adversary decrypts the hash values from the leaked password database and let the hash values be $\mathscr{R}_1^{dec}$ and $\mathscr{R}_2^{dec}$. The adversary decrypts the hash values from the leaked password database as follows-

$$\begin{aligned} \mathscr{R}_1^{dec} &= \text{AES.}\textsc{dec}(\mathscr{C}_1, \mathscr{K}_1', \mathcal{IV}_1') \\ \mathscr{R}_2^{dec} &= \text{AES.}\textsc{dec}(\mathscr{C}_2, \mathscr{K}_2', \mathcal{IV}_2') \end{aligned} \tag{24}$$

Let us assume the following equation-

$$\mathscr{R}_1 = \mathscr{R}_1^{dec}, \qquad \mathscr{R}_2 = \mathscr{R}_2^{dec} \tag{25}$$

where the following equation holds-

$$\begin{aligned} \mathscr{K}_1 \neq \mathscr{K}_1', \qquad & \mathscr{K}_2 \neq \mathscr{K}_2' \\ \mathcal{IV}_1 \neq \mathcal{IV}_1', \qquad & \mathcal{IV}_2 \neq \mathcal{IV}_2' \end{aligned} \tag{26}$$

If Equations 24, 25 and (26) hold, then Equation (25) holds due to collision. Moreover, the encryption process produces the same ciphertexts as the legal ones for different keys due to a collision. However, the collision on such encryption and decryption is computationally infeasible.

6.1.2. Collision attack on PassProS. Let us assume that adversary leaked the $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathcal{C}_1$ and $\mathcal{C}_2$ in PassProS. The adversary needs legitimate secrets ($\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{SK}_1$, and $\mathcal{SK}_2$) to gain authentication. Let us assume that the adversary uses $\mathscr{R}_1'$, $\mathscr{R}_2'$, $\mathcal{S}_1'$, $\mathcal{S}_2'$, $\mathcal{SK}_1'$, and $\mathcal{SK}_2'$ to produce the has values as follows-

$$\begin{aligned} \mathscr{C}_1' &= \text{SHF}(\mathscr{R}_1', \mathcal{S}_1', \mathcal{SK}_1', seed1) \\ \mathscr{C}_2' &= \text{SHF}(\mathscr{R}_2', \mathcal{S}_2', \mathcal{SK}_2', seed2) \end{aligned} \tag{27}$$

Let us assume the following equation-

$$\begin{aligned} \mathscr{R}_1 \neq \mathscr{R}_1', \qquad & \mathscr{R}_2 \neq \mathscr{R}_2' \\ \mathcal{S}_1 \neq \mathcal{S}_1', \qquad & \mathcal{S}_2 \neq \mathcal{S}_2' \\ \mathcal{SK}_1 \neq \mathcal{SK}_1', \qquad & \mathcal{SK}_2 \neq \mathcal{SK}_2' \end{aligned} \tag{28}$$

For the above equation, let us assume the following equation-

$$\mathscr{C}_1 = \mathscr{C}_1', \qquad \mathscr{C}_2 = \mathscr{C}_2' \tag{29}$$

If Equations (28) and (29) hold, then the adversary can gain authentication. The collision attack on the secret key is the only way to perform an attack on the leaked database in PassProS; however, it is infeasible for the adversary to execute such an attack on the leaked database.

6.1.3. Guessing attacks and dictionary attacks. The adversary cannot perform a guessing attack or a dictionary attack on the leaked password database due to the server's secret credentials. Let the entire password database of the PassPro be leaked. In PassProE, the adversary has the ciphertexts and the challenges. The adversary cannot decrypt the $\mathscr{R}_1$ and $\mathscr{R}_2$ from $\mathcal{C}_1$ and $\mathcal{C}_2$ even if the adversary has the correct password and secret context. Even the legitimate user cannot decrypt the $\mathscr{R}_1$ and $\mathscr{R}_2$ from $\mathcal{C}_1$ and $\mathcal{C}_2$ from the leaked password database. Let the adversary have produced the correct password and secret context to generate the correct secret keys, but is unable to decrypt the $\mathscr{R}_1$ and $\mathscr{R}_2$ from the $\mathscr{C}_1$ and $\mathscr{C}_2$. The adversary has the correct the $\mathcal{SK}_1$ and $\mathcal{SK}_2$ to derive the $\mathscr{R}_1$ and $\mathscr{R}_2$ from the $\mathscr{C}_1$ and $\mathscr{C}_2$. However, the adversary needs $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{IV}_1$ and $\mathcal{IV}_2$ to correctly decrypt the $\mathscr{R}_1$ and $\mathscr{R}_2$ from the $\mathscr{C}_1$ and $\mathscr{C}_2$. Therefore, the adversary cannot perform a guessing attack.

In PassProS, the $\mathscr{C}_1$ and $\mathscr{C}_2$ are hash values using SHF. Let the adversary have the correct password and secret context. Therefore, the adversary tries to produce $\mathscr{C}_1$ and $\mathscr{C}_2$ using the correct password and secret context. So, the adversary supplies the $\mathscr{R}_1$, $\mathscr{R}_2$, $\mathcal{SK}_1$ and $\mathcal{SK}_2$. The adversary also requires $\mathcal{S}_1$ and $\mathcal{S}_2$, which can be reproduced by the server exclusively. Therefore, the adversary cannot produce $\mathscr{C}_1$ and $\mathscr{C}_2$ to match with the leaked password database even if the adversary has the correct password. Even the legitimate user cannot produce the $\mathscr{C}_1$ and $\mathscr{C}_2$ in the client devices using the correct password and secret context. Therefore, the guessing attacks and dictionary attacks do not apply to the leaked password database.

7. Performance

We have conducted experiments on Ubuntu 18.04.6 LTS, Intel® Core™ i7-7700 CPU @ 3.60GHz × 8 in Node.js version 16.20.2. The experimental results are shown in Table 7.

TABLE 7. EXECUTION TIME OF SHA, AES, ARGON2, SHUFFLE AND SHF IN SECONDS.

Methods	Execution time in seconds
SHA256	0.0003975430
SHA512	0.0003990520
AES Encryption	0.0007142880
AES Decryption	0.0000945650
Argon2 (t=2, p=1, m=64MB, l=256)	0.2291294360
Argon2 (t=3, p=1, m=64MB, l=256)	0.3252478013
Argon2 (t=4, p=1, m=64MB, l=256)	0.4375857007
Shuffle	0.0059097150
SHF	0.0064556570

Table 7 shows the execution time of SHA256, SHA512, AES encryption, AES decryption, Argon2, shuffle function, and SHF. SHA256 and SHA512 exhibit similar performance. The performance of SHA and AES is also almost similar. The AES encryption is slower than the decryption. The Argon2 is slower than all other functions, and it requires more memory than other methods compared in Table 7. The security strength of Argon2 depends on the time, memory, and parallelism. The higher the time and memory, the better the security of Argon2. For instance, the security of Argon2 at $m = 128$ MB is superior to that at $m = 64$ MB. Moreover, we have experimented with Argon2 for $p = 1$ (parallelism). The higher value of parallelism is faster than the lower value; however, Argon2 is used to restrict parallelism. Moreover, a single function should not consume many server resources. Thus, it requires higher resources than non-memory-hard functions. AES encryption and decryption, as well as SHA, are the fastest operations. The SHF is slower than SHA and AES. Notably, we can tune the η of Algorithm 3 to slow down the SHF.

8. Discussion

Table 8 compares FIDO's passkey and our proposed work. Our proposed work is similar to Ross *et al.* [13]; however, the password storage mechanism differs. Ross *et al.* [13] depends on a conventional password hashing mechanism. Therefore, it can have different types of attacks based on the implementation. The recent password hashing competition winner is Argon2i, which can prevent precomputation attacks, rainbow table attacks, and parallel attacks by utilizing a memory-hard hash function and salt. It consumes a large memory footprint to prevent the adversary's parallelism. However, it is not designed to prevent offline dictionary, guessing attacks, phishing attacks, and the domino effect. On the contrary, FIDO implements a passwordless authentication mechanism to prevent various attacks on password-based authentication mechanisms. FIDO's passkey relies on biometrics and device security. The key drawback of a passkey is the absence of a password. The biometrics are not spoof-proof, and once compromised, they remain compromised. Additionally, it can yield false positives, and physical damage to the biometric, such as a fingerprint, can also cause an issue. Moreover, the secret credentials can also be breached through the passkey management service (PMS) [11]. So, Islam *et al.* [11] design a decoy to detect such a breach. However, PassPro can prevent diverse attacks using a password and a secret context.

8.1. Secret Context

The SHF is used to conceal the original string and prevent it from being transmitted to the server in its raw form. The SHF uses a secret context to produce a unique hash value. The secret context (ctx) can be either secret numbers or a secret word. The secret context is used to influence the shuffling process. The SHF produces the shuffled word by shuffling two strings using a secret context, but the secret context is not mixed with the shuffled word. The secret context is never published, exposed, or transmitted outside of the user's devices, either directly or indirectly. Thus, it is guaranteed that the adversary cannot derive the secret context from the leaked password database.

Moreover, the secret context can also be biometric. However, we have not implemented biometrics as a secret context in PassPro, and we keep it for future work. The biometric will never be published, exposed, or transmitted to any other entity if we make the biometric a secret context for SHF. The biometric is used to influence the shuffling process between the hash values of two words. The fusion between the password and biometric can provide stronger security guarantees.

8.2. Shuffling process

The user often uses the same user ID and password across multiple domains; consequently, an adversary can access all other domains once the password is compromised. Therefore, we shuffle the user ID and password with a domain word. We use a secret context to shuffle the strings so that the adversary cannot reproduce the shuffled word even if the password is compromised. The legitimate user can correctly reproduce the shuffled word, but it is hard for adversaries. Alternatively, we can term it as a password stretching algorithm that converts a low-entropy password into a high-entropy password using SHF. Let us assume that the password of the given user is compromised. Still, the adversary cannot prove its authenticity because the produced hash values will be different due to the secret context. We have demonstrated the SHF using ω_1 and ω_2 in Equation (4). The probability of reproducing the $\mathscr{R}$ using SHF for the inputs ω_1 and ω_2 without knowing the secret context is almost zero, even if we assume the ω_1 and ω_2 as publicly known words. Additionally, PassPro utilizes the password in SHF, along with a secret context, to generate the hash values. The adversary requires two secrets to produce the correct

TABLE 8. Comparative analysis of diverse authentication mechanisms.

Name	Authenticates	Security protocol	Remarks
Ross *et al.* [13]	Users	Public key, Captcha and 2FA	1. Uses salts and password hash functions. 2. Provide strong resistance to phishing attacks. 3. Weak resistance to a domino effect. 4. Password database protection depends on secret salts.
FIDO's passkey [10]	Registered devices	1. Passwordless, Biometrics 2. Public-private key	1. It does not have precomputation attacks, rainbow table attacks, guessing attacks, dictionary attacks. 2. It is also prone to phishing attacks, which can lead to a domino effect [11]. 3. Sensitive to device security. 4. Prone to spoofing attacks.
PassPro	Users	1. Password and Secret context 2. SHF and PUF	1. Strong resistance to collision attacks. 2. The precomputation attacks, rainbow table attacks, guessing attacks, dictionary attacks, phishing attacks, and the domino effect do not apply.

hash values. From Equations (9), we can conclude that the SHF always translates different hash values as follows:

1) Different user ID with the same password, secret context, and domain.
2) Different domain name with the same user ID, password, and secret context.
3) Different passwords with the same user ID, domain, and secret context.
4) Different secret context with the same user ID, password, and domain.

8.3. Client-side hashing

Conventionally, password hashing is performed on the server side. So, the client device must transmit the password to the server. Hence, a memory-hard function is required to hash the passwords. Moreover, it creates more dependence on the server's security; thus, we cannot assume that the password database and salts will not be leaked simultaneously. On the contrary, PassPro does not transmit the raw password and user ID to store them in the password database for authentication purposes. Instead, PassPro uses client-side password hashing for transmission. Firstly, the client shuffles the user ID and password with a domain word using a secret context. Secondly, the PassPro client converts the shuffled word into hash values for transmission to the server. The hash value of the password is shuffled with other hash values to produce a shuffled word. The shuffled word is converted into a hash value using SHA. Thus, the password is not directly exposed to the adversary.

8.4. Mutually reproducible secret key

Interestingly, password encryption on the server side is often discouraged due to the complexity of managing secret keys and their potential for leakage. If the secret key is compromised, the original passwords can be retrieved. Therefore, a conventional authentication mechanism avoids password encryption on the server side and hashes the password with a salt on the server side. Thus, passwords are hashed with salts to prevent diverse attacks, but managing the secret salt is also a challenge. It can also be leaked. Additionally, memory-hard hash functions are employed to replace conventional password hashing algorithms. However, we found that encryption of password is better than hashing the password using memory-hard hash function because of the following reasons: a) symmetric key encryption is faster than memory-hard hash function, b) symmetric key encryption does not require large memory footprint, and c) provides stronger security guarantees against diverse attacks, such as guessing attacks. Notably, the server does not store the user's secret key. Instead, the user supplies the secret key and produces the mutually reproducible secret key to encrypt or decrypt. Thus, the security of the password database is controlled by the user and server.

9. Conclusion

In this paper, we have presented an SHF and shown that the SHF can be applied to a) produce unique hash values, b) generate a key, c) stretch a password, d) hash a password on the server, and e) hash a string using a keyed (keyed-hash function). We have demonstrated the hashing process of the two publicly known words using a secret context. The SHF is faster than Argon2 and slower than SHA and AES. Moreover, we use the SHF to implement PassPro. We have also presented two methods to protect the password database: a) PassProE and b) PassProS. Both are capable of preventing precomputation attacks and rainbow table attacks. Also, we have shown that a collision attack is computationally infeasible in both PassProE and PassPros. Moreover, the guessing attacks and dictionary attacks do not apply to both PassProE and PassPros because the legitimate user also cannot decrypt the hash values from the leaked password database or reproduce the leaked ciphertext. Furthermore, the adversary cannot use the derived password from the leaked password database in other domains to impersonate the legitimate users. So, the phishing attack and the domino effect do not apply in PassPro. Our proposed work is the first of its kind to prevent diverse attacks, such as collision attacks, precomputation attacks, rainbow table attacks, dictionary attacks, guessing attacks, phishing attacks, and the domino effect, without using a significant memory footprint or slowing down the algorithm.

References

[1] C. Percival, "Stronger key derivation via sequential memory-hard functions," in *BSDCan - The BSD Conference*. University of Ottawa, Canada: BSDCan, 2009, pp. 1–16. [Online]. Available: http://www.tarsnap.com/scrypt/scrypt.pdf

[2] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. Saarbruecken, Germany: IEEE, 2016, pp. 292–302.

[3] J. Alwen and V. Serbinenko, "High parallel complexity graphs and memory-hard functions," in *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 595–603. [Online]. Available: https://doi.org/10.1145/2746539.2746622

[4] B. Chen and S. Tessaro, "Memory-hard functions from cryptographic primitives," in *Advances in Cryptology – CRYPTO 2019*, A. Boldyreva and D. Micciancio, Eds. Cham: Springer International Publishing, 2019, pp. 543–572.

[5] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro, "Scrypt is maximally memory-hard," in *Advances in Cryptology – EUROCRYPT 2017*, J.-S. Coron and J. B. Nielsen, Eds. Cham: Springer International Publishing, 2017, pp. 33–62.

[6] J. Blocki, B. Harsha, S. Kang, S. Lee, L. Xing, and S. Zhou, "Data-independent memory hard functions: New attacks and stronger constructions," in *Advances in Cryptology – CRYPTO 2019*, A. Boldyreva and D. Micciancio, Eds. Cham: Springer International Publishing, 2019, pp. 573–607.

[7] J. Blocki and B. Holman, "Sustained space and cumulative complexity trade-offs for data-dependent memory-hard functions," in *Advances in Cryptology – CRYPTO 2022*, Y. Dodis and T. Shrimpton, Eds. Cham: Springer Nature Switzerland, 2022, pp. 222–251.

[8] D. Boneh, H. Corrigan-Gibbs, and S. Schechter, "Balloon hashing: A memory-hard function providing provable protection against sequential attacks," in *Advances in Cryptology – ASIACRYPT 2016*, J. H. Cheon and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 220–248.

[9] "Passkeys - FIDO Alliance," Nov. 2024, [Online; accessed 14. Dec. 2024]. [Online]. Available: https://fidoalliance.org/passkeys

[10] L. Lassak, E. Pan, B. Ur, and M. Golla, "Why aren't we using passkeys? obstacles companies face deploying FIDO2 passwordless authentication," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 7231–7248. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/lassak

[11] M. Islam, S. S. Arora, R. Chatterjee, and K. C. Wang, "Detecting compromise of passkey storage on the cloud," in *Proceedings of the USENIX Security Symposium*. USENIX Association, 2025, pp. 1–18. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity25/sec25cycle1-prepub-215-islam.pdf

[12] K. C. Wang and M. K. Reiter, "How to end password reuse on the web," in *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA, USA: NDSS, 2019, pp. 1–15.

[13] 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, pp. 17–31. [Online]. Available: https://www.usenix.org/conference/14th-usenix-security-symposium/stronger-password-authentication-using-browser-extensions

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

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

[16] 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.

[17] A. Appleby, "murmurhash," [Online], Retrieved on December 2023 from https://github.com/aappleby/smhasher/tree/master, 2023.

[18] 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

[19] J. Delvaux, D. Gu, I. Verbauwhede, M. Hiller, and M.-D. M. Yu, "Efficient Fuzzy Extraction of PUF-Induced Secrets: Theory and Applications," in *Cryptographic Hardware and Embedded Systems – CHES 2016*. Berlin, Germany: Springer, Aug. 2016, pp. 412–431.

[20] F. Cirillo and C. Esposito, "Efficient puf-based iot authentication framework without fuzzy extractor," in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 695–704. [Online]. Available: https://doi.org/10.1145/3672608.3707853