PassPro: A Secure Password-based Authentication Mechanism to Prevent Attacks

Ripon Patgiri National Institute of Technology Silchar ripon@cse.nits.ac.in

Abstract

The password-based authentication system is a widely used authentication mechanism. However, it has several issues, including the domino effect, guessing attacks, dictionary attacks, rainbow table attacks, and database leakage issues. To address these issues, we present a client-side password hashing method called PassPro. PassPro uses two secrets and a domain word to shuffle the strings. The shuffled strings are converted into hash values and sent to the identity manager for authentication or identity creation. The shuffling is based on a pseudo-random algorithm. The legitimate user can reproduce the shuffled string again. The hash values are encrypted in the password database using a password-based encryption method with a mutually reproducible secret word for each user. Therefore, PassPro features- a) client-side password metering, b) client-side password hashing, c) prevention of the domino effect from leaked password database, d) protection of the password database leakage, e) encryption of the hash values using a mutually reproducible secret word, and g) prevention of dictionary and guessing attacks. Also, PassPro guarantees that adversaries, including authentication managers, cannot retrieve the user's original password and user ID. Alternatively, the original user ID and password cannot be retrieved even if the password database is given to the adversary. Furthermore, a password database's user ID and password are invalid in other domains, even if the user uses the same user ID and password in multiple domains.

1 Introduction

The advent of the computing power of adversaries demands the revision of existing authentication mechanisms because authentication mechanisms protect precious digital assets. Moreover, conventional computers are equipped with more powerful processors and GPUs at a low cost, and attacking them becomes easier for adversaries. Interestingly, the password-based authentication mechanism is a widely used method to protect digital assets from being misused. The Laiphrakpam Dolendro Singh National Institute of Technology Silchar Idsingh@cse.nits.ac.in

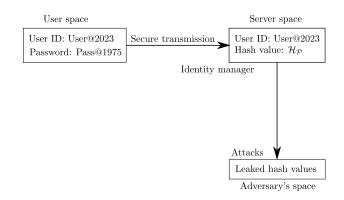


Figure 1: Block diagram of authentication mechanism of the state-of-the-art method.

password-based authentication mechanism has numerous challenges, including the domino effect, guessing attacks, dictionary attacks, rainbow table attacks, and database leakage. The state-of-the-art password-based authentication system uses salt to defeat dictionary attacks but has an issue with short salt and salt reuse [7]. However, we cannot assume that the hash values and the salts are leaked in the conventional authentication system.

Definition 1 A client sets the user ID as U and password as \mathcal{P} in multiple domains. Once the U and \mathcal{P} are compromised, then the adversary can use the identity to gain authentication in multiple domains. This phenomenon is called the domino effect.

Definition 2 A function \mathcal{F} transforms an input string, ω , into an output, \mathcal{O} , by the significant influence of another input, \mathscr{I} , and the \mathscr{I} is called a context. Alternatively, $\mathcal{F}(\omega) \xrightarrow{\mathscr{I}} \mathcal{O}$ or $\mathscr{O} \xleftarrow{\mathscr{I}} \mathcal{F}(\omega)$.

1.1 Motivation

Recent developments suggest that users' passwords are published widely, which is fatal for the password-based authentication system. COMB publishes 3279064312 passwords online, which is approximately 3.27 billion pairs of emails and passwords [9]. The report further highlights that 3.27 billion is 40% of the world's total population. Similarly, Rock-You2021 publishes a whooping 8.4 billion passwords (100GB text data) [10]. The number of passwords is almost equivalent to the world's population. The COMB and RockYou2021 can reveal the raw passwords and identities from the state-of-theart identity manager, which is highly risky. The adversary uses a rainbow table attack to reveal the passwords from the leaked hash values. The password stretching algorithm such as [4] does not affect the domino effect and rainbow table attack. However, it can defeat guessing attacks and dictionary attacks. Moreover, numerous models have already been proposed to perform guessing attacks, including deep-learning models [6, 8, 11, 12, 18, 20], targeted guessing attacks [18], pattern-based guessing attacks [17] and large-language models [15]. These models are efficient and effective in predicting users' passwords to evade. Moreover, the targeted online password guessing also poses serious threats to password-based authentication systems [18]. Diverse defensive mechanisms have already been proposed to address the issues such as rainbow table attacks.

The state-of-the-art password-based authentication mechanism assumes that a) hash values and salts cannot be leaked simultaneously and b) identity managers are honest by default. The attacker can derive equivalent passwords if the hash values and salts are leaked simultaneously. Moreover, a dishonest identity manager can store the original passwords in a different database because the raw passwords are transmitted to the server. Thus, we cannot assume a dishonest identity manager in the conventional password-based authentication mechanism. These two assumptions lead to our proposed solution. We assume that everything can be leaked from the server (identity manager), including the master secret password of the identity manager, salts, and the entire password database. Moreover, we assume that the identity manager is a dishonest authentication manager.

1.2 Our proposed work

The recent password-hashing competition winner was Argon2, where Argon2i is suitable for password hashing [3]. Argon2 features memory hardness for password hashing where the legitimate user hashes the password using some memory footprint, whereas the adversary requires a huge memory footprint, which restricts the parallelism [13]. It is designed to hash the password on the server side, which contrasts our proposed work. Moreover, our proposed work encourages the users to reuse their password and user ID in multiple domains, which contrasts Wang and Reiter [19]. Wand and Reiter [19] proposes password lookup in different domains to prevent password reuse which is not a viable option.

Figure 1 demonstrates the scenes of attacking a password.

We assume that the server stores the hash values of the passwords. Moreover, we assume that the hash values are leaked and the adversary derives the equivalent or the exact password using a rainbow table attack or other attacks. In that case, the attacker can reuse the derived password to gain authentication in a different domain. The current state-of-the-art system does not guarantee that the adversary cannot gain authentication using the derived password in a different domain if the user reuses the password. If the user password reuses the passwords in different domains, there is a possibility that the adversary can gain authentication using the derived password, which is derived by rainbow table attacks or other methods. Moreover, we assume that a dishonest authentication manager to prove the superiority of our proposed method over a state-of-the-art authentication manager. The state-of-the-art method stores the hash values of the password, but the raw password is transmitted from the user to the server. A dishonest authentication manager can easily store the raw password in a different database. Using these credentials, the dishonest authentication manager can easily gain authentication in different domains if the user reuses the password in different domains. To address the above-raised issues, we propose a new password-based authentication derived from Ross et al. [16], called PassPro, which features client-side password hashing. We present a stronger defense against diverse security threats. Therefore, we draw our key objectives as outlined below-

- To prevent getting the direct raw form of the user IDs and passwords by the adversary, including the authentication manager (we use the identity manager as analogous to the authentication manager throughout the paper).
- To ensure that the adversary can never retrieve the original user IDs and passwords, including the identity manager.
- To prevent the domino effect and the password leakage issue.

Our proposed work features a client-side hashing to achieve the above objectives. We implement a shuffle function to shuffle two different strings. The shuffling algorithm is based on a pseudo-random algorithm 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 identity manager for authentication.

1.3 Key contributions

Our key contributions are outlined below-

• PassPro completely prevents the domino effect from the leaked password database. It shuffles the user ID,

password, and domain word to produce three unique strings. Therefore, the same user ID and password can be used in multiple domains without worrying about the domino effect. In this case, we assume that the attacker cannot correctly guess all the secret credentials in three incorrect login attempts.

- PassPro is designed based on the client-side password hashing to conceal the original user ID and password. It ensures that the adversary never gets the original user ID and passwords in raw form from the leaked password database. Moreover, the authentication manager never receives the users' credentials in raw form.
- The Identity manager of PassPro never receives the original user ID and password in raw form; therefore, it features client-side password metering. Alternatively, the server (identity manager or authentication manager) can never suggest the strength of a password or user ID. Therefore, it invalidates the use of AJAX in the framework.
- The secret credentials of each user are encrypted with a different key in the password database of the identity manager. The identity manager computes a mutually reproducible secret word using the hash value provided by the user and the master secret word from the identity manager.
- PassPro protects users' identities from stealing by adversaries. The user ID and password are encrypted with different passwords; therefore, the adversary cannot decrypt the user ID and password even if the entire password database of the identity manager is leaked.
- PassPro adds more complexities for the guessing attacker using two secrets.

1.4 Our results

PassPro differs from conventional password-based authentication systems, where hashing is performed on the client side rather than hashing a password on the server side (identity manager). We present our results in the next sub-subsections.

1.4.1 Shuffling with a context

The user enters an email ID (\mathcal{E}) , a user ID (\mathcal{U}) , and a password (\mathcal{P}) for authentication where the \mathcal{E} is publicly visible identity. In contrast, the \mathcal{U} and \mathcal{P} are secret credentials. PassPro retrieves a domain word (\mathcal{D}) to shuffle the user ID and password. We shuffle the user's credentials as $\mathscr{U} \xleftarrow{\mathcal{U}} \mathcal{U} \cup \mathcal{D}$, $\mathscr{P} \xleftarrow{\mathcal{P}} \mathcal{P} \cup \mathcal{D}$ and $\mathscr{K} \xleftarrow{\mathcal{U}} \mathcal{U} \cup (\mathcal{P} \mid\mid \mathcal{D})$. Therefore, shuffling produces a shuffled user ID (\mathscr{U}) , a shuffled password (\mathscr{P}) , and part of a mutually reproducible secret word (\mathscr{K}) . The shuffling algorithm produces different shuffled strings-

- For the same user IDs and passwords for different domains.
- For the same user IDs and domains for different passwords.
- For the same passwords and domains for different user IDs.

We can easily retrieve the original user ID and password can be retrieved from the \mathcal{U} and \mathcal{P} by removing the characters of the domain word. We hash the shuffled strings to conceal the original raw string of the user's secret credentials before sending the user's credentials to the authentication manager. Thus, it is difficult to reproduce the original raw strings from the hashed values. Therefore, client-side hashing is required to conceal the original user ID and password.

1.4.2 Client-side hashing

A hash function can be invoked at the client's computers to convert shuffled user ID, password, and domain word into hash values. Our proposed system relies on secure hash functions, such as SHA2-256 or SHA512. Thus, our proposed algorithm produces three hash values: $\mathcal{H}_{\mathscr{U}} \leftarrow \mathcal{H}(\mathscr{U})$, $\mathcal{H}_{\mathscr{P}} \leftarrow \mathcal{H}(\mathscr{P})$, and $\mathcal{H}_{\mathscr{K}} \leftarrow \mathcal{H}(\mathscr{K})$. Therefore, it prevents the transmission of the original password and user ID to the authentication manager. Hence, the authentication manager cannot retrieve the actual user ID and password from the hash values, including the adversaries. The hash values of the shuffled strings cannot be the same for the different domains, even if for the same password and user ID.

1.4.3 Domino-effect

The domino effect is a crucial effect that needs to be prevented due to the adversary's presence in password-based authentication systems. The domino effect occurs due to password reuse in multiple domains. The user of PassPro sends the hash values to the identity manager; therefore, it is computationally infeasible to reconstruct the original string from the hash values $\mathcal{H}_{\mathscr{U}}$ and $\mathcal{H}_{\mathscr{P}}$. This method prevents the domino effect because the different domain words create different shuffled hash values for the same user ID and password. Thus, it permits the reuse of user IDs and passwords in multiple domains.

1.4.4 Encryption and decryption

The user's hash values are encrypted using AES; however, we can apply a password-based encryption method [2]. Identity manager computes a mutually reproducible secret word \mathfrak{W} using $\mathcal{H}_{\mathscr{K}}$ and a master secret word from the identity manager \mathcal{W} , i.e., $\mathfrak{W} \leftarrow \mathcal{H}(\mathcal{S}\mathcal{W})$ where $\mathcal{S}\mathcal{W} \xleftarrow{\mathcal{W}} (\mathcal{H}_{\mathscr{K}} \cup \mathcal{H}_{\mathcal{W}})$. Moreover, the \mathfrak{W} is updated using Argon2i [3]. The encryption is performed using $\mathcal{C}_{\mathscr{U}} \leftarrow AES.Enc(\mathcal{H}_{\mathscr{U}},\mathfrak{W})$ and

 $\mathcal{C}_{\mathscr{P}} \leftarrow AES.Enc(\mathcal{H}_{\mathscr{P}},\mathfrak{W})$ where AES is Advanced Standard Encryption. Alternatively, each user's hash values are encrypted with a different secret key. The algorithm requires a hash value from the user and a master secret word from the identity manager to encrypt or decrypt the hash values on the server side. The user supplies a hash value of the shuffled string to the identity manager, where shuffling is performed among the user ID, password, and domain word with a context. We encrypt the hash values using a mutually reproducible secret word. Therefore, the identity manager cannot reproduce the mutually reproducible secret key without the hash value from the user. Besides, the user cannot reproduce the mutually reproducible secret key without the master secret word from the identity manager. PassPro encrypts the individual shuffled-then-hashed user ID and password with an individual secret key. Alternatively, the identity manager is unable to decrypt the stored hash values without the user's consent. Similarly, the user cannot decrypt its own hash values from the password database without permission from the identity manager.

1.4.5 Leakage of the password database

The adversary always tries to steal the password database to gain authentication in multiple domains. Strict measures are required to prevent such attacks. Therefore, we introduce a mutually reproducible word to encrypt using the AES method to encrypt and decrypt. Consequently, it prevents the identity manager from decrypting the hash values without permission from the user. Similarly, the user cannot decrypt the encrypted hash values from the password database without having permission from the identity manager, even if the identity manager provides the user's own encrypted hash values to the user. With the condition mentioned above, the adversary is unable to decode the encrypted hash values even if the adversary evades the security of the identity manager to steal the database. Therefore, it is computationally infeasible to decrypt all the encrypted hash values from the password database for the adversary. Therefore, the user IDs and passwords are intact even after the adversary steals the password database.

1.4.6 Guessing and dictionary attacks

The passwords are created eight characters long- using capital letters, small letters, digits, and special symbols, which creates huge password spaces. However, a user always chooses easy-to-remember passwords; thus, the password space becomes small for the attackers. Thus, the guessing attacks become a reality. Also, the dictionary size of the adversary becomes smaller. For instance, almost all users' passwords are published at [9, 10]. Therefore, it becomes easy for adversaries to perform the attacks. Hence, a new measurement is needed to thwart such kinds of attacks. Our proposed system uses

two secret words: user ID and password. Consequently, the guessing and dictionary attacks become harder even if the passwords are published at [9, 10].

2 PassPro: The proposed system

In a password-based authentication system, the user enters a user ID and password to prove the genuineness. However, there are many issues with conventional password-based authentication systems. The prominent issue is the domino effect. People often reuse the same password and user ID for various identity managers so that they can remember them easily. Therefore, if the password is derived from the leaked database, the adversary can break the security of all other identity managers if the user reuses the password in different domains. Moreover, conventional identity managers store passwords by applying salt to them, but still, there is a possibility of leaking the salts. If the hash values and the slats are leaked, it becomes easy for attackers. Hence, our key objectives are as given below-

- To protect the users' identities from stealing and publishing by adversaries.
- To protect users' raw password from the identity manager.
- To prevent the domino effect where the adversary cannot use a derived password from the leaked password database to gain authentication in multiple domains.

Our proposed system relies on existing OTP mechanisms to identify the desired user. Also, it relies 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 three hash values to the identity manager: $\mathcal{H}_{\mathcal{U}}$, $\mathcal{H}_{\mathcal{P}}$, and $\mathcal{H}_{\mathcal{K}}$.

2.1 Identity creation

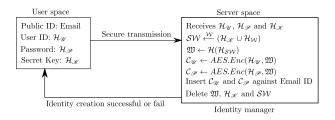


Figure 2: Block diagram of PassPro in the identity creation process.

Table 1 shows a user's identity creation at the identity manager and is also demonstrated in a block diagram in Figure 2. In our proposed system, the email ID and user ID are treated

Table 1: Step-by-step	overview of th	e identity	creation of	a user.

Client	Identity Manager (IDM)
• User retrieves a domain words \mathcal{D} .	
• User enters user ID \mathcal{U} and password \mathcal{P} .	
• The user performs shuffling-	
$\mathcal{U} \cup \mathcal{D} \xrightarrow{\mathcal{P}} \mathscr{U}, \mathcal{P} \cup \mathcal{D} \xrightarrow{\mathcal{U}} \mathscr{P},$	
$ \begin{array}{c c} \mathcal{U} \cup (\mathcal{P} \mid\mid \mathcal{D}) \xrightarrow{\mathcal{U} \mid\mid \mathcal{P}} \mathscr{K} \end{array} \end{array} $	
• The user performs hashing-	
$\mathcal{H}_{\mathscr{U}} \leftarrow \mathcal{H}(\mathscr{U}), \mathcal{H}_{\mathscr{P}} \leftarrow \mathcal{H}(\mathscr{P}), \mathcal{H}_{\mathscr{K}} \leftarrow \mathcal{H}(\mathscr{K})$	
• The user sends the $\mathcal{H}_{\mathcal{U}}, \mathcal{H}_{\mathcal{P}}$, and $\mathcal{H}_{\mathcal{K}}$ to the identity	
manager using POST method.	
	• IDM receives the hash values- $\mathcal{H}_{\mathcal{U}}, \mathcal{H}_{\mathcal{P}}, \text{ and } \mathcal{H}_{\mathcal{K}}.$
	• IDM computes a mutually reproducible secret
	word \mathfrak{W} using $\mathcal{H}_{\mathscr{K}}$ and a master secret word from the identity
	manager \mathcal{W} .
	• The IDM encrypts the hash values-
	$AES.Enc(\mathcal{H}_{\mathscr{U}},\mathfrak{W}) \text{ and } AES.Enc(\mathcal{H}_{\mathscr{P}},\mathfrak{W})$
	• The IDM inserts these encrypted hash values
	into the password database and deletes $\mathcal{H}_{\mathcal{K}}$, \mathcal{SW} and \mathfrak{W} .

differently. The email ID is a public ID, and the user ID is a secret word similar to the password. Initially, the user retrieves a domain word (\mathcal{D}) , and the user enters two secrets: a user ID (\mathcal{U}) and password (\mathcal{P}) . The user ID, password, and domain word are shuffled on the client side to create a unique identity (see Algorithm 2 for shuffling) as given below-

መ

$$\begin{aligned} \mathcal{U} \cup \mathcal{D} \xrightarrow{\mathcal{T}} \mathscr{U} \\ \mathcal{P} \cup \mathcal{D} \xrightarrow{\mathcal{U}} \mathscr{P} \\ \mathcal{U} \cup (\mathcal{P} \mid\mid \mathcal{D}) \xrightarrow{\mathcal{U} \mid\mid \mathcal{P}} \mathscr{K} \end{aligned} \tag{1}$$

The \cup represents shuffling, and $\xrightarrow{\mathcal{P}}$ represents \mathcal{P} as context in the shuffling process. For instance, \mathcal{U} and \mathcal{D} is shuffled using a context \mathcal{P} to produces a shuffled string \mathscr{U} and we represent the entire process as $\mathcal{U} \cup \mathcal{D} \xrightarrow{\mathcal{P}} \mathscr{U}$. Notably, the characters of \mathcal{P} are not included in the shuffled string \mathscr{U} . The shuffled strings are converted into hash values by the client as given below-

$$\begin{aligned} &\mathcal{H}_{\mathscr{U}} \leftarrow \mathcal{H}(\mathscr{U}) \\ &\mathcal{H}_{\mathscr{P}} \leftarrow \mathcal{H}(\mathscr{P}) \\ &\mathcal{H}_{\mathscr{K}} \leftarrow \mathcal{H}(\mathscr{K}) \end{aligned}$$
(2)

The correct hash values can be reproduced exclusively by the legitimate user. The user sends these hash values $(\mathcal{H}_{\mathscr{U}}, \mathcal{H}_{\mathscr{P}}, \mathcal{H}_{\mathscr{K}})$ to the identity manager using the POST method. The identity manager receives the hash values $(\mathcal{H}_{\mathscr{U}}, \mathcal{H}_{\mathscr{P}}, \mathcal{H}_{\mathscr{K}})$. The identity manager shuffles the $\mathcal{H}_{\mathscr{K}}$ and the hash value of the master secret word, \mathscr{W} , from the identity manager to form $\mathcal{S}\mathcal{W}$ as given below-

$$\mathcal{SW} \xleftarrow{\mathcal{W}} (\mathcal{H}_{\mathscr{K}} \cup \mathcal{H}_{\mathscr{W}})$$
(3)

The identity manager applies the hash function to produce a mutually reproducible secret word, as shown below-

$$\mathfrak{W} \leftarrow \mathcal{H}(\mathcal{SW}) \tag{4}$$

Moreover, we update the **W** using Argon2i as

$$\mathfrak{W} \leftarrow Argon2i(\mathfrak{W}, \mathcal{H}_{Salt}) \tag{5}$$

where *Salt* is generated from the \mathcal{W} as

$$Salt \leftarrow \mathcal{H}'_{\mathcal{K}} \mid\mid \mathcal{W} \tag{6}$$

The $\mathcal{H}'_{\mathcal{K}}$ is formed by selecting 32 characters from $\mathcal{H}_{\mathcal{K}}$ pseudo-randomly, where it can be reconstructed for the correct inputs. Thus, the adversary cannot guess the salt, and each user has a different salt value. We apply Argon2i because it restricts the adversary from launching parallel computation by limiting the memory size of the adversary. The two hash values $\mathcal{H}_{\mathcal{U}}$, and $\mathcal{H}_{\mathscr{P}}$ are encrypted using AES method as given below-

$$C_{\mathscr{U}} \leftarrow AES.Enc(\mathcal{H}_{\mathscr{U}},\mathfrak{W}) \\ C_{\mathscr{P}} \leftarrow AES.Enc(\mathcal{H}_{\mathscr{P}},\mathfrak{W})$$
(7)

The $C_{\mathscr{U}}$ and $C_{\mathscr{P}}$ are inserted into the password database against the user's email ID and delete the $\mathfrak{W}, S\mathcal{W}, \mathcal{H}_{\mathcal{W}}, \mathcal{H}_{\mathscr{K}}$ and $\mathcal{H}'_{\mathscr{K}}$.

2.2 Authentication process

Table 2 shows a similar procedure as the identity creation for the authentication process except for the decryption process.

Table 2: Step-by-step overview of the authentication of a user.			
Client	Identity Manager (IDM)		
• User retrieves a domain words \mathcal{D} .			
• User enters user ID \mathcal{U} and password \mathcal{P} .			
• The user performs shuffling-			
$\mathcal{U} \cup \mathcal{D} \xrightarrow{\mathcal{P}} \mathscr{U}, \mathcal{P} \cup \mathcal{D} \xrightarrow{\mathcal{U}} \mathscr{P},$			
$\mathcal{U} \cup (\mathcal{P} \mid\mid \mathcal{D}) \xrightarrow{\mathcal{U} \mid\mid \mathcal{P}} \mathscr{K}$			
 The user performs hashing- 			
$\mathcal{H}_{\mathscr{U}} \leftarrow \mathcal{H}(\mathscr{U}), \mathcal{H}_{\mathscr{P}} \leftarrow \mathcal{H}(\mathscr{P}), \mathcal{H}_{\mathscr{K}} \leftarrow \mathcal{H}(\mathscr{K})$			
• The user sends the $\mathcal{H}_{\mathcal{U}}$, $\mathcal{H}_{\mathcal{P}}$, and $\mathcal{H}_{\mathcal{K}}$ to the			
identity manager using POST method.			
	• IDM receives the received hash values- $\mathcal{H}_{\mathcal{U}}, \mathcal{H}_{\mathcal{P}}$, and $\mathcal{H}_{\mathcal{K}}$.		
	• IDM computes a mutually reproducible secret word \mathfrak{W} as		
	shown in Table 1.		
	• The IDM decrypts the stored hash values-		
	$\mathcal{H}^{decrypted}_{\mathscr{U}} \leftarrow AES.Dec(\mathcal{C}_{\mathscr{P}},\mathfrak{W}), \text{ and }$		
	$\mathcal{H}^{decrypted}_{\mathscr{P}} \leftarrow AES.Dec(\mathcal{C}_{\mathscr{P}},\mathfrak{W})$		
	• The IDM compares the hash values for authentication-		
	$\mathcal{H}_{\mathscr{U}} = \mathcal{H}_{\mathscr{U}}^{decrypted}$, and $\mathcal{H}_{\mathscr{U}} = \mathcal{H}_{\mathscr{U}}^{decrypted}$.		
	If this comparison is successful, the user is authenticated; otherwise,		
	the authentication fails.		
	• The identity manager discards $\mathcal{H}_{\mathcal{K}}, \mathcal{H}'_{\mathcal{K}}, \mathcal{SW}, \mathcal{H}_{\mathcal{W}}$ and \mathfrak{W} .		

Table 2: Step-by-step overview of the authentication of a user.

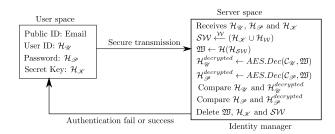


Figure 3: Block diagram of authentication process of PassPro.

A block diagram in Figure 3 also depicts the authentication process. The user shuffles the user ID, password, and domains word as $\mathcal{U} \cup \mathcal{D} \xrightarrow{\mathcal{P}} \mathcal{U}, \mathcal{P} \cup \mathcal{D} \xrightarrow{\mathcal{U}} \mathcal{P}$, and $\mathcal{U} \cup (\mathcal{P} \mid\mid \mathcal{H}_{\mathcal{D}}) \xrightarrow{\mathcal{U} \mid\mid \mathcal{P}} \mathcal{H}$. The user converts these shuffled strings into the hash values and produces $\mathcal{H}_{\mathcal{U}}, \mathcal{H}_{\mathcal{P}}$, and $\mathcal{H}_{\mathcal{K}}$ as shown in Equation (2). The user sends these hash values to the identity manager using the POST method. The identity manager receives these hash values and produces \mathfrak{W} using $\mathcal{H}_{\mathcal{H}}$ and $\mathcal{H}_{\mathcal{W}}$ as shown in Equations (3)-(6). The identity manager decrypts the stored hash values as given below-

$$\mathcal{H}_{\mathscr{U}}^{decrypted} \leftarrow AES.Dec(\mathcal{C}_{\mathscr{P}}, \mathfrak{W}) \\ \mathcal{H}_{\mathscr{P}}^{decrypted} \leftarrow AES.Dec(\mathcal{C}_{\mathscr{P}}, \mathfrak{W})$$
(8)

The \mathfrak{W} is a high entropy key to encrypt or decrypt. Therefore, it is strong enough to secure the encrypted data with a memory-hard hash function, Argon2i. The decrypted hash values and received hash values are compared for authentication as given below-

$$\begin{array}{l} if \ (\mathcal{H}_{\mathscr{U}} = \mathcal{H}_{\mathscr{U}}^{decrypted} \ and \ \mathcal{H}_{\mathscr{P}} = \mathcal{H}_{\mathscr{P}}^{decrypted}) \\ Authentication \ is \ success ful \\ else \\ Authentication \ fails. \end{array}$$

As shown above, the user is authenticated if the comparison is successful. Otherwise, the authentication fails. The identity manager securely deletes the $\mathcal{H}_{\mathcal{K}}, \mathcal{H}'_{\mathcal{K}}, \mathcal{SW}, \mathcal{H}_{W}$ and \mathfrak{W} .

2.3 User ID

The conventional system uses the user ID as a publicly visible identifier, for instance, mail@example.com. However, our proposed scheme treats the user ID as equally important as the password. Alternatively, the user ID and password are the secret words. If a user ID is secret, it is stronger and can protect against diverse attacks. A weak password can reveal the secret of shuffling. Therefore, we consider the user ID as a secret word similar to the conventional password other than the email ID. A user ID can be formed using the same rules as the password. For instance, a user ID can be User@2025. The user ID and password cannot be the same. The user ID consists of eight characters, including 26 small-case letters, 26 upper-case letters, 10 digits, and 32 symbols (excluding white space). Therefore, by choosing eight characters from 94 available characters, the maximum number of possible user IDs is

$$\binom{94}{8} = 111,315,063,717\tag{9}$$

We have eight character strings comprising of alphabets, digits, and special symbols. Therefore, we can form 111 billion user IDs. Theoretically, it is large enough to thwart brute-force attacks but cannot withstand guessing attacks and dictionary attacks.

2.4 Password

In the creation of a password, we can follow the same rules as current practices. It is similar to conventional passwords with a length of a minimum of ten characters. The password is constituted of ten characters from 26 small-case letters, 26 upper-case letters, 10 digits, and 32 symbols (excluding white space). Therefore, by choosing ten characters from 94 available characters, the maximum number of possible passwords is

$$\binom{94}{10} = 9,041,256,841,903 \tag{10}$$

The challenge is to defeat the guessing and dictionary attacks. Theoretically, a password with ten characters can withstand diverse attacks by the above-mentioned rules; however, it cannot withstand dictionary and guessing attacks even if the password space is large.

2.5 Dictionary attacks

The dictionary and/or guessing attacks can be effectively carried out on stolen password databases because of the three login attempt restrictions. We assume all user IDs and passwords are listed in COMB or RockYou2021. Therefore, performing a guessing or dictionary attack on either the user ID or the password is trivial. Our proposed system comprises two secrets, namely, user ID and passwords. Let the user ID and passwords be in the COMB dictionary (the size of COMB is less than the size of RockYou2021), i.e., $\mathcal{U}, \mathcal{P} \in COMB$. But both \mathcal{U} and \mathcal{P} must belong to the same user and be correct. Therefore, the probability of selecting the correct user ID and password from COMB that belongs to a single user is as given below-

$$\frac{1}{\binom{3279064312}{2}} = \frac{1}{5376131379476484516}$$
(11)
= 1.8600735908678218e⁻¹⁹

which is fairly small enough to thwart guessing attacks or dictionary attacks that motivate us why to consider the user ID as a secret word. Apparently, the two secret words can cause inconvenience to the users. However, we must prevent diverse attacks such as password database leakage. Thus, the inconvenience is justified. But it cannot ensure freedom from the domino effect. Moreover, it cannot protect the password database from being stolen. Therefore, we use a domain word to prevent such kinds of attacks.

2.6 Domain Word

A user needs to retrieve the domain word, which is shuffled with the user's ID and password. The domain word is public and available to all, including the adversary. For instance, the domain words are ieee.org, acm.org, usenix.org, iacr.org, ndss-symposium.org, etc. These words are mixed with the user ID and password to prevent diverse attacks.

2.7 Shuffling the strings

Alg	Algorithm 1 Computing an integer value for utilization.		
1:	procedure GetVALUE($\omega_1, \mathscr{L}_{\omega_1}, \omega_2, \mathscr{L}_{\omega_2}, \omega_3, \mathscr{L}_{\omega_3}, \mathcal{S}, \tau$)		
2:	for <i>i</i> : 1 <i>to</i> τ do		
3:	$S = PRIMARYHASH(\omega_1, \mathscr{L}_{\omega_1}, S)$		
4:	$S = PRIMARYHASH(\omega_2, \mathcal{L}_{\omega_2}, S)$		
5:	$S = PRIMARYHASH(\omega_3, \mathcal{L}_{\omega_3}, S)$		
6:	end for		
7:	return S		
8:	8: end procedure		
	-		

Algorithm 1 computes an integer value using a primary hash function. A primary hash function is a non-secure hash function that produces a hash value of β -bit (32 bits or 64 bits), for instance, the murmur hash function [1]. The function iterates τ times to alter the seed value. Initially, the seed value is public and converted into a private value using a user ID, password, and domain word.

The user often uses the same user ID and password over multiple domains; therefore, once the password is compromised, the adversary can access all other domains. Therefore, we shuffle the user ID and password with a domain word. The domain word is public; however, the legitimate user can correctly reproduce the shuffled word. Algorithm 2 portrays the shuffling between two words with a context (see Definition 2 for context), and the results of the shuffling are shown in Table 3. Algorithm 2 requires two strings to shuffle with the context of another word. The strings are ω_1 and ω_2 and the context is ω_3 . Our algorithm depends on a non-cryptography string hash function such as Murmur2 [1]. The integer value generation process (Algorithm 1) requires the invocation of a primary hash function τ times, which is dependent on the input string. The τ can be computed as $\tau = S\%\delta + \mu$ where we set $\delta = 1783$ for demonstration purpose. We can set δ to a larger prime number to make it harder to reproduce the shuffled string by the adversary. However, the larger δ value slows down the process. The shuffling algorithm is similar to the password-stretching algorithm. The low-entropy user

to c	ontext ω_3 .
1:	procedure SHUFFLE($\omega_1, \omega_2, \omega_3, S$)
2:	$\mathscr{L}_{\omega_1}, \mathscr{L}_{\omega_4} = \text{Length}(\omega_1)$
3:	$\mathscr{L}_{\omega_2}, \mathscr{L}_{\omega_5} = \text{Length}(\omega_2)$
4:	$\mathscr{L}_{\omega_3}, \mathscr{L}_{\omega_6} = \text{Length}(\omega_3)$
5:	$Copy(\omega_4, \omega_1), Copy(\omega_5, \omega_2), Copy(\omega_6, \omega_3)$
6:	$\tau = 16, \delta = 1783, \mu = 16$
7:	$\mathcal{S} = \text{GetValue}(\omega_3, \mathscr{L}_{\omega_3}, \omega_2, \mathscr{L}_{\omega_2}, \omega_1, \mathscr{L}_{\omega_1}, \mathcal{S}, \tau)$
8:	$ au = S\%\delta + \mu$
9:	while $\mathscr{L}_{\boldsymbol{\omega}_1} \neq 0$ and $\mathscr{L}_{\boldsymbol{\omega}_2} \neq 0$ do
10:	if $\mathcal{S} \wedge 1 = 0$ then
11:	$S = \text{GETVALUE}(\omega_1, \mathscr{L}_{\omega_1}, \omega_2, \mathscr{L}_{\omega_2}, \omega_3, \mathscr{L}_{\omega_3}, S, \tau)$
12:	$ au = \mathcal{S}\%\delta + \mu$
13:	$k = S \% \mathscr{L}_{\omega_4}$
14:	$buff[i++] = \omega_4[k]$
15:	REMOVEONECHAR $(\omega_4, k, \mathscr{L}_{\omega_4})$
16:	$\mathscr{L}_{\omega_4} = \mathscr{L}_{\omega_4} - 1$
17:	else
18:	$\mathcal{S} = \text{GETVALUE}(\boldsymbol{\omega}_3, \mathscr{L}_{\boldsymbol{\omega}_3}, \boldsymbol{\omega}_1, \mathscr{L}_{\boldsymbol{\omega}_1}, \boldsymbol{\omega}_2, \mathscr{L}_{\boldsymbol{\omega}_2}, \mathcal{S}, \boldsymbol{\tau})$
19:	$ au = S\%\delta + \mu$
20:	$k = S \% \mathscr{L}_{\omega_5}$
21:	$buff[i++] = \omega_5[k]$
22:	REMOVEONECHAR($\omega_5, k, \mathscr{L}_{\omega_5}$)
23:	$\mathscr{L}_{\omega_5} = \mathscr{L}_{\omega_5} - 1$
24:	end if
25:	end while
26:	for $k: 0$ to \mathcal{L}_{ω_4} do
27:	$\mathcal{S} = \text{GETVALUE}(\boldsymbol{\omega}_1, \mathscr{L}_{\boldsymbol{\omega}_1}, \boldsymbol{\omega}_3, \mathscr{L}_{\boldsymbol{\omega}_3}, \boldsymbol{\omega}_2, \mathscr{L}_{\boldsymbol{\omega}_2}, \mathcal{S}, \boldsymbol{\tau})$
28:	$\tau = \mathcal{S}\%\delta + \mu$
29:	pos = S%i, i = i + 1
30:	INSERTCHARAT($buff, i, \omega_4[k], pos$)
31:	end for
32:	for $k : 0$ to \mathcal{L}_{05} do
33:	$\begin{split} \mathcal{S} &= \text{GETVALUE}(\omega_2, \mathscr{L}_{\omega_2}, \omega_1, \mathscr{L}_{\omega_1}, \omega_3, \mathscr{L}_{\omega_3}, \mathcal{S}, \tau) \\ \tau &= \mathcal{S}\%\delta + \mu \end{split}$
34:	pos = S%i, i = i + 1
35:	
36:	INSERTCHARAT($buff, i, \omega_5[k], pos$)
37: 38:	end for SHA256(<i>buff</i> , <i>length</i> (<i>buff</i>),0)
	end procedure
	chu procedure

Algorithm 2 Shuffling two strings, ω_1 and ω_2 , with respect

ID and password are converted into a high-entropy string by shuffling the strings. The shuffling process depends on a pseudo-random number generation algorithm. The indexes are generated pseudo-randomly depending on the input string. Based on the generated index, the characters are placed in a buffer, and the selected character of the string is removed using REMOVEONECHAR($\omega_4, k, \mathscr{L}_{\omega_4}$) where it removes a character from *omega*₄ indexed at *k*. Finally, the remaining characters are inserted pseudo-randomly into the buffer using INSERTCHARAT(*buff*, *i*, $\omega_4[k]$, *pos*) where the character $\omega_4[k]$ is inserted into *buff* at index *pos*. The legitimate user can reproduce the shuffled strings, but it is hard for adversaries. Therefore, the adversary tries to shuffle by guessing a valid user ID and password. The valid user ID and the password must belong to a single user. Let us assume that the adversary guesses two valid strings: user ID and passwords, and they belong to different users. The adversary cannot gain authentication because the produced hash values will be different. For instance, shuffling the correct user ID and domain name requires the correct password (context) to produce the correct shuffled string. Therefore, partial correctness leads to a wrong hash value, as shown in Table 3. The adversary requires two secrets to produce the correct hash values.

Table 3 shows the example of shuffling two words using a context, and the inputs are taken as an example for demonstration purposes. We produce three different shuffled words and hash these three shuffled words using SHA256. In Test Case 1, the user ID is User@2023, the password is Pass@1975, and the domain word is mydomain.com for demonstration and understanding purposes. We want to produce a shuffled string for user ID, so user ID and domain word are shuffled using a content password. The shuffled string is e2roy3mmi@no.dcsaUm02, where the shuffling process is influenced by the context, but the context is not included in the shuffled string. A similar process is applied to the password and the key. In Test Case 2, we change the password from Pass@1975 to Abc@1975 to see the changes with respect to Test Case 1. Similarly, in Test Case 3, we change the domain word from mydomain.com to example.com to observe the changes with respect to Test Case 1. Likewise, we change the user ID from User@2023 to Xyz@2023 in Test Case 4 to observe the changes with respect to Test Case 1. However, our proposed work shuffles two hash values with a given context instead of the plaintext as given in Table 3 to make it harder to reproduce by adversaries. Thus, a tiny change leads to a drastic change in the hash value.

Therefore, we can conclude as follows-

- Different passwords with the same user ID and domain word translate into different shuffled user IDs and passwords.
- Different domain names with the same user ID and password translate into different shuffled user IDs and Passwords.
- Different user IDs with the same password and domain word translate into different shuffled user IDs and Passwords.

PassPro creates different user IDs and Passwords for different user IDs, passwords, and domain words. From Table 3, the adversary can easily reconstruct the user ID and password from the shuffled user ID and password by removing the characters from the domain words if the client does not hash the shuffled words. Also, the adversary can get the user IDs and passwords if the adversary is able to steal the entire database.

Test	Context	Shuffling	String	Input String	Shuffled string	256-bit hash value using SHA2-256
Case		for	type			
1	Pass@1975	User ID	User ID	User@2023	e2roy3mmi@no	7da56be963061fc99b598f0f7a103780
	Pass@1973		Domain	mydomain.com	.dcsaUm02	a49262a08b44588dad21fa57e627c62a
1	User@2023	Password	Password	Pass@1975	cynPmas9i@s.m	76ad4e9e6be754db4b77039732886788
			Domain	mydomain.com	50a0d7m1	0ddfec73270d7a75130ab89fc0b42f4f
[Pass@1975+	Key	User ID	User@2023	9.7m@m1r5so0mU	9379b269ff4bdbd097a78e702371df3a4
	User@2023		Merging	Pass@1975+ mydomain.com	sPa22icseon@yad3	124b7c59b500bce1c681cbceb056656
2			User ID	User@2023	3yammoro.sUcd0	eb57bbaef2f1b1223a43584e82ac6913b
Pass-	Abc@1975	User ID	Domain	mydomain.com	2me2ni@	5a815bc0ed7945d5c4ce7b689d03c39
word		-	Password	Abc@1975	7dc@9nabcmo.	52973688dad62031f7429f1a0023c6c0
Change	User@2023	Password	Domain	mydomain.com	momyi1A5	59d09cb8e6100103764dd58194a27bbc
	Abc@1975+	Key	User ID	User@2023	id1U59reoocca2@	2aa1e1b994ed322e54e0141f72b17702
			Merging	Abc@1975+ mydomain.com	mA@7mbn0.3yims2	83d15f3e1eff9eb96e7d69e4310c64c3
3	Pass@1975	User ID	User ID	User@2023	3opmx.mec@rae	042d567cf42b8242921b6c1d88fa22bf
Domain			Domain	example.com	02l2eUs	170461f9470fe34f75cefaf1640a111e
Name	Chair@2023	Password	Password	Pass@1975	ps.a175cmmoe9@	3353e4b7d6237992a5951b61686abcb
Change			Domain	example.com	lxeaPs	dc6dcf44f89c7b8173cc37f240cb0f728
ſ	Pass@1975+	Key	User ID	User@2023	@sx2pm5c3se92re	6d69ae422b83ef372bd21435a629219a
	User@2023	ксу	Merging	Pass@1975+ example.com	0@o7la1amePs.U	2572ade52400497a4b80ec651c3aed04
4	Pass@1975	User ID	User ID	Xyz@2023	.2@z3mnmd2yyo	1ca8b4dcc6836ed13fe8208df4d5716e
User ID			Domain	mydomain.com	i0cmaXo	318231f537c70d39715e65ade45091dd
	Xyz@2023	Password	Password	Pass@1975	i1y.Pd@oo7acms	26d4226853f5a229dda4ad690cde4d5c
			Domain	mydomain.com	5mmasn9	9550fbee0354f72693f23e405cc5865a
ſ	Pass@1975+ Xyz@2023	Key	User ID	Xyz@2023	@a5329@mmaiyos	6418887b91b45d0d1a258a9fb475b61
			Merging	Pass@1975+ mydomain.com	2donX07yPs1.cmz	330757d4927c3d30381f5ddbe274f05fa

Table 3: Example of shuffling two input strings with respect to a context.

Therefore, PassPro strongly discourages storing/transmitting the raw form of the user ID and password in the identity manager. Therefore, it demands client-side password hashing to prevent it, which is described in the next subsection.

2.8 Client-side hashing

PassPro does not transmit the raw password and user ID to store it 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. Secondly, the client of PassPro converts the shuffled user ID and password into two different hash values for transmission to the identity manager. We rely on the SHA256 hash function for hashing the shuffled strings. However, we can also use other variants of SHA, such as SHA512 or SHA3. The identity manager of PassPro stores the hash values of the user ID and password for future authentication by encrypting. The hash value of the same user ID and password is invalid for the different domains. Therefore, these hash values cannot be used in other domains to gain authentication by the adversary. We assume that the user's password is a vulnerable low-entropy string. Thus, we shuffle the strings with a domain word and hash the shuffled string. Consequently, the hash values become high-entropy strings. The legitimate user can reproduce the hash values, and it is hard to reproduce the hash values of a user by an adversary. For illustration, the \mathscr{U} , \mathscr{P} and \mathscr{K} are constructed by shuffling \mathcal{U} and \mathcal{P} using a domain word \mathcal{D} . We shuffled the strings, \mathcal{U}, \mathcal{P} and \mathcal{D} using (1) and produces \mathcal{U}, \mathcal{P} and \mathcal{K} . The shuffling process uses a context to shuffle the word, and context influences the placement of the characters in the shuffled word. The shuffled strings are hashed using (2) and produce $\mathcal{H}_{\mathcal{U}}, \mathcal{H}_{\mathcal{P}}$ and $\mathcal{H}_{\mathcal{K}}$. It conceals the original raw user ID and passwords. Moreover, the shuffling process mixes the domain words, and therefore, the hash values are different for different domains for the same user ID and password.

It is guaranteed that the original string is not possible to

reconstruct from $\mathcal{H}_{\mathcal{U}}$, $\mathcal{H}_{\mathcal{U}}$, and $\mathcal{H}_{\mathcal{K}}$. Thus, the hash values $\mathcal{H}_{\mathcal{U}}$, $\mathcal{H}_{\mathcal{U}}$, and $\mathcal{H}_{\mathcal{K}}$ are sent to the identity manager for either identity creation or authentication. The original string can never be reconstructed from the hash value because it uses the SHA256 algorithm. Moreover, the input is shuffled pseudorandomly, so the adversary can never reconstruct the sequence without knowing the input strings. Furthermore, the rainbow table attack becomes infeasible because the input strings are shuffled in another context. We show that two different users, let \mathbb{A} and \mathbb{B} , cannot produce the same hash values. There are two cases: a) two different user IDs and the same passwords, and b) two different passwords and the same user IDs. We take the first case by taking two different user IDs as $\mathcal{U}_1 \neq \mathcal{U}_2$, which are non-empty. The shuffling of the client \mathbb{A} 's user ID and password are given in Equation (12).

$$\begin{aligned} \mathcal{U}_{1} \cup \mathcal{D} \xrightarrow{\mathcal{P}} \mathscr{U}_{1} \\ \mathcal{P} \cup \mathcal{D} \xrightarrow{\mathcal{U}_{1}} \mathscr{P}_{1} \\ \mathcal{U}_{1} \cup (\mathcal{P} \mid\mid \mathcal{D}) \xrightarrow{\mathcal{P} \mid\mid \mathcal{U}_{1}} \mathscr{K}_{1} \end{aligned}$$
(12)

The hash values of the client \mathbb{A} 's user ID and password are given in Equation (13).

$$\begin{aligned} \mathcal{H}(\mathscr{U}_1) &\to \mathcal{H}_{\mathscr{U}_1} \\ \mathcal{H}(\mathscr{P}_1) &\to \mathcal{H}_{\mathscr{P}_1} \\ \mathcal{H}(\mathscr{K}_1) &\to \mathcal{H}_{\mathscr{K}_1} \end{aligned} \tag{13}$$

The shuffling of the client \mathbb{B} 's user ID and password are shown in Equation (14).

$$\begin{aligned} \mathcal{U}_2 \cup \mathcal{D} \xrightarrow{\mathcal{P}} \mathscr{U}_2 \\ \mathcal{P} \cup \mathcal{D} \xrightarrow{\mathcal{U}_2} \mathscr{P}_2 \\ \mathcal{U}_2 \cup (\mathcal{P} \mid\mid \mathcal{D}) \xrightarrow{\mathcal{P} \mid\mid \mathcal{U}_2} \mathscr{K}_2 \end{aligned}$$
(14)

The hash values of the client \mathbb{B} 's user ID and password are given in Equation (15).

$$\begin{aligned} \mathcal{H}(\mathscr{U}_2) &\to \mathcal{H}_{\mathscr{U}_2} \\ \mathcal{H}(\mathscr{P}_2) &\to \mathcal{H}_{\mathscr{P}_2} \\ \mathcal{H}(\mathscr{K}_2) &\to \mathcal{H}_{\mathscr{K}_2} \end{aligned}$$
(15)

From Equation (13) and (15), we can draw the Equation (16).

$$\begin{aligned} &\mathcal{H}_{\mathscr{U}_{1}} \neq \mathcal{H}_{\mathscr{U}_{2}} \\ &\mathcal{H}_{\mathscr{P}_{1}} \neq \mathcal{H}_{\mathscr{P}_{2}} \\ &\mathcal{H}_{\mathscr{K}_{1}} \neq \mathcal{H}_{\mathscr{K}_{2}} \end{aligned}$$
(16)

Therefore, two users with different user IDs and the same password always create different hash values. Similarly, two different users having the same user ID but different passwords also create different hash values. Let the password be $\mathcal{P}_1 \neq \mathcal{P}_2$, which are non-empty. The shuffled strings of the client A are given in Equation (17).

$$\begin{aligned} \mathcal{U} \cup \mathcal{D} &\xrightarrow{\mathcal{P}_1} \mathscr{U}_3 \\ \mathcal{P}_1 \cup \mathcal{D} &\xrightarrow{\mathcal{U}} \mathscr{P}_3 \\ \mathcal{U} \cup (\mathcal{P}_1 \mid\mid \mathcal{D}) &\xrightarrow{\mathcal{P}_1 \mid\mid \mathcal{U}} \mathscr{K}_3 \end{aligned}$$
(17)

The hash values of the client \mathbb{A} are created using Equation (18).

$$\begin{aligned} &\mathcal{H}(\mathscr{U}_{3}) \to \mathcal{H}_{\mathscr{U}_{3}} \\ &\mathcal{H}(\mathscr{P}_{3}) \to \mathcal{H}_{\mathscr{P}_{3}} \\ &\mathcal{H}(\mathscr{K}_{3}) \to \mathcal{H}_{\mathscr{K}_{3}} \end{aligned}$$
(18)

The shuffled strings of the client \mathbb{B} are given in Equation (19).

$$\begin{aligned} \mathcal{U} \cup \mathcal{D} \xrightarrow{\mathcal{U}_{2}} \mathscr{U}_{4} \\ \mathcal{P}_{2} \cup \mathcal{D} \xrightarrow{\mathcal{U}} \mathscr{P}_{4} \\ \mathcal{U} \cup (\mathcal{P}_{2} \mid\mid \mathcal{D}) \xrightarrow{\mathcal{P}_{2} \mid\mid \mathcal{U}} \mathscr{K}_{4} \end{aligned} \tag{19}$$

The hash values of the client \mathbb{B} are given in Equation (20).

$$\begin{aligned} \mathcal{H}(\mathscr{U}_4) &\to \mathcal{H}_{\mathscr{U}_4} \\ \mathcal{H}(\mathscr{P}_4) &\to \mathcal{H}_{\mathscr{P}_4} \\ \mathcal{H}(\mathscr{K}_4) &\to \mathcal{H}_{\mathscr{K}_4} \end{aligned}$$
(20)

From Equation (18) and (20), we conclude that no two users can produce the same hash values if the users' passwords are different, as shown in Equation (21), even if the user ID and the domain word are the same.

$$\begin{aligned} &\mathcal{H}_{\mathscr{V}_{3}} \neq \mathcal{H}_{\mathscr{V}_{4}} \\ &\mathcal{H}_{\mathscr{P}_{3}} \neq \mathcal{H}_{\mathscr{P}_{4}} \\ &\mathcal{H}_{\mathscr{K}_{3}} \neq \mathcal{H}_{\mathscr{K}_{4}} \end{aligned}$$
 (21)

Equation (16) and (21) ensure that two different users cannot produce the same hash values.

2.9 Prevention of the domino effect

The domino effect is a crucial effect that needs to be prevented in password-based authentication systems. We consider the domino effect, where the adversary derives a password from a leaked password database, and it occurs due to password reuse. A study estimates that about 43-51% of users reuse passwords in multiple domains [5]. PassPro deals with hash values with a shuffled user ID and password, which encourages password reuse. Our proposed approach is similar to Ross *et al.* [16]. 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 $\mathcal{H}_{\mathcal{U}}$, $\mathcal{H}_{\mathcal{U}}$, and $\mathcal{H}_{\mathcal{K}}$ if the adversary does not know the input strings except the domain word. This method ensures the prevention of the domino effect because different domain words create different shuffled hash values. Thus, it permits the reuse of user IDs and passwords in multiple domains. Let us assume a user \mathbb{A} uses the \mathcal{U} and \mathcal{P} at two domains, namely, the domain words are \mathcal{D}_1 and \mathcal{D}_2 where $\mathcal{D}_1 \neq \mathcal{D}_2$ which are non-empty. The shuffled strings for the domain \mathcal{D}_1 for user \mathbb{A} are shown in Equation (22).

$$\begin{aligned} \mathcal{U} \cup \mathcal{D}_{1} &\xrightarrow{\mathcal{P}} \mathscr{U}_{5} \\ \mathcal{P} \cup \mathcal{D}_{1} &\xrightarrow{\mathcal{U}} \mathscr{P}_{5} \\ \mathcal{U} \cup (\mathcal{P} \mid\mid \mathcal{D}_{1}) &\xrightarrow{\mathcal{P} \mid\mid \mathcal{U}} \mathscr{K}_{5} \end{aligned}$$
(22)

The shuffled strings for the domain \mathcal{D}_2 for user A are derived in Equation (23).

$$\begin{aligned} \mathcal{U} \cup \mathcal{D}_2 &\xrightarrow{\mathcal{P}} \mathscr{U}_6 \\ \mathcal{P} \cup \mathcal{D}_2 &\xrightarrow{\mathcal{U}} \mathscr{P}_6 \\ \mathcal{U} \cup (\mathcal{P} \mid\mid \mathcal{D}_2) &\xrightarrow{\mathcal{P} \mid\mid \mathcal{U}} \mathscr{K}_6 \end{aligned}$$
(23)

We know that $\mathcal{D}_1 \neq \mathcal{D}_2$ which is non-empty; therefore, it produces different shuffled strings. Hence, Equation (24) holds.

$$\begin{aligned} &\mathcal{U}_5 \neq \mathcal{U}_6 \\ &\mathcal{P}_5 \neq \mathcal{P}_6 \\ &\mathcal{K}_5 \neq \mathcal{K}_6 \end{aligned}$$

The hash values for the shuffled string with domain \mathcal{D}_1 for user \mathbb{A} are given in Equation (25).

$$\begin{aligned} \mathcal{H}(\mathcal{U}_5) &\to \mathcal{H}_{\mathcal{U}_5} \\ \mathcal{H}(\mathcal{P}_5) &\to \mathcal{H}_{\mathcal{P}_5} \\ \mathcal{H}(\mathcal{K}_5) &\to \mathcal{H}_{\mathcal{K}_5} \end{aligned}$$
(25)

Similar to Equation (25), the hash values for the domain \mathcal{D}_2 for the same user are demonstrated in Equation (26).

$$\begin{aligned} \mathcal{H}(\mathscr{U}_6) &\to \mathcal{H}_{\mathscr{U}_6} \\ \mathcal{H}(\mathscr{P}_6) &\to \mathcal{H}_{\mathscr{P}_6} \\ \mathcal{H}(\mathscr{K}_6) &\to \mathcal{H}_{\mathscr{K}_6} \end{aligned}$$
(26)

We can conclude that the hash values cannot be the same in Equation (25) and (26). Hence, Equation (27) shows the inequality.

$$\begin{aligned} &\mathcal{H}_{\mathscr{U}_{5}} \neq \mathcal{H}_{\mathscr{U}_{6}} \\ &\mathcal{H}_{\mathscr{P}_{5}} \neq \mathcal{H}_{\mathscr{P}_{6}} \\ &\mathcal{H}_{\mathscr{K}_{5}} \neq \mathcal{H}_{\mathscr{K}_{6}} \end{aligned}$$
 (27)

Equation (27) shows that the hash values cannot be the same for the same user ID and password for different domains. Thus, there is no domino effect from the stolen password database. Moreover, the identity manager and adversary do not know the original input strings of the user IDs and passwords.

2.10 Context

Suppose the client uses hashing of the password with a domain word similar to Ross *et al.* [16]. The adversary can guess the password and reproduce the hash value because the domain word is public. Thus, we need to use two secret words: user ID and password. Suppose the adversary guesses the password, but it also requires the user ID to gain authentication. Therefore, we use context in shuffling such that the adversary cannot get any partial success even if the adversary guesses either the correct password or user ID.

$$\mathcal{H}_{\mathscr{P}} \stackrel{?}{\not\leftarrow} \mathcal{H}(\mathcal{P} \cup \mathcal{D}) \tag{28}$$

The adversary does not know the context of Equation (28). Therefore, the adversary cannot correctly reproduce the hash value even if the adversary can guess the password correctly. Similarly, the user ID and domain are shuffled using a password context. Therefore, the adversary must correctly guess the two secret words to reproduce the correct hash values. For further illustration, we use two different contexts for the same password and domain name. Let the context be U_1 and U_2 where $U_1 \neq U_2$, then we get

$$\mathcal{P}_{1} \xleftarrow{\mathcal{U}_{1}} \mathcal{P} \cup \mathcal{D}$$

$$\mathcal{P}_{2} \xleftarrow{\mathcal{U}_{2}} \mathcal{P} \cup \mathcal{D}$$
(29)

where

$$\mathscr{P}_1 \neq \mathscr{P}_2$$
 (30)

It shows that the adversary cannot reproduce the shuffled string even if the adversary can get one secret word. It is hard to reconstruct the correct sequence of the shuffled word without knowing the secret. Therefore, the shuffling prevents rainbow attacks. Moreover, the collision attack does not apply due to multiple hash values.

2.11 Mutually reproducible secret word

The mutually reproducible secret word is a combination of the user's hash value and the identity manager's secret word. The user computes $\mathcal{H}_{\mathscr{H}}$ and sends this hash value to the identity manager. The identity manager produces \mathfrak{W} using Equations (3)-(6). The mutually reproducible secret word is used to encrypt or decrypt the hash values. The \mathfrak{W} restricts the unwanted decryption of passwords and user IDs from the password database.

Let us assume the adversary gets the master secret word \mathcal{W} . In this scenario, the adversary requires a hash value $\mathcal{H}_{\mathcal{K}}$, but the adversary could not get, and hence, the adversary uses a random hash value $\mathcal{H}'_{\mathcal{K}}$. The adversary computes $\mathcal{H}'_{\mathfrak{W}}$ where the $\mathfrak{W} \neq \mathfrak{W}'$ holds, and hence, the adversary cannot decrypt the stored user IDs and passwords. Thus, it is still secure even if the server makes the master secret word \mathcal{W} public. Moreover, we can assume that the *Salt* is leaked, but

still, PassPro is secure because the adversary cannot correctly reproduce the secret key \mathfrak{W} .

2.12 Encryption and decryption

We can also use the hashing method to store the user's credentials as follows-

$$C_{\mathscr{U}} = \operatorname{Argon2i}(\mathcal{H}_{\mathscr{U}}, \mathfrak{W})$$

$$C_{\mathscr{P}} = \operatorname{Argon2i}(\mathcal{H}_{\mathscr{P}}, \mathfrak{W})$$
(31)

We can use the above equation to store the user's credentials; however, we choose encryption or decryption for better security. Our encryption and decryption involve a mutually reproducible secret word, \mathfrak{W} . Users' hash values are encrypted and inserted into the identity manager's password database during identity creation. The identity manager decrypts the encrypted hash values from the identity manager's database for authentication. For cryptography, we have many options for key derivation: password-based encryption [2] without key derivation, password-based key derivation [13, 14], Argon2 hash function [3], and PBKDF2-HMAC-SHA512. However, we use SHA512 and Argon2i for key derivation. Moreover, we can use either AES or ECC encryption. However, we use AES for demonstration purposes for this paper.

2.12.1 Encryption

The identity manager encrypts the hash values of the user ID and password using an individual key. The identity manager requires a hash value to produce the secret word to encrypt. The identity manager stores $\mathcal{H}_{\mathcal{U}}$, and $\mathcal{H}_{\mathcal{P}}$ in its database by encrypting using individual mutually reproducible secret word, \mathfrak{W} . The \mathfrak{W} is produced by shuffling-then-hash using $\mathcal{H}_{\mathcal{K}}, \mathcal{H}_{\mathcal{W}}$ and \mathcal{W} as shown in Equations (3)-(6). In identity creation, the $\mathcal{H}_{\mathcal{U}}$, and $\mathcal{H}_{\mathcal{P}}$ are encrypted by the identity manager using \mathfrak{W} and inserted these encrypted hash values into the database. The hash values are encrypted as shown in Equation (7). The \mathfrak{W} is different for each user, i.e., the two users, A and \mathbb{B} produce $\mathfrak{W}_{\mathbb{A}}$ and $\mathfrak{W}_{\mathbb{B}}$, respectively, and $\mathfrak{W}_{\mathbb{A}} \neq \mathfrak{W}_{\mathbb{B}}$. The $C_{\mathcal{U}}$ and $C_{\mathcal{P}}$ are stored in the password database for future use. Retrieving these hash values requires a mutually reproducible secret word, without which it fails to decrypt. After inserting the encrypted hash values, the identity manager deletes $\mathcal{H}_{\mathcal{H}}, \mathcal{SW}, \mathcal{H}_{W}$ and \mathfrak{W} . Otherwise, the adversary can get the secret words to decrypt the stored hash values. Let us assume two different users, \mathbb{A} and \mathbb{B} , have a valid identity at the identity manager. The user A sends $\mathcal{H}_{\mathcal{U}_A}$, $\mathcal{H}_{\mathscr{P}_A}$, and $\mathcal{H}_{\mathscr{K}_{\mathbb{A}}}$ to the identity manager. Similarly, the user $\mathbb B$ also sends $\mathcal{H}_{\mathscr{U}_{\mathbb{R}}}, \mathcal{H}_{\mathscr{P}_{\mathbb{R}}}$, and $\mathcal{H}_{\mathscr{K}_{\mathbb{R}}}$ to the identity manager. Therefore, the identity manager produces two mutually reproducible secret words, $\mathfrak{W}_{\mathbb{A}}$ and $\mathfrak{W}_{\mathbb{B}}$, for the hash values of the user \mathbb{A} and \mathbb{B} , respectively. Let us assume that the email ID for the user $\mathbb A$ and \mathbb{B} are $\mathcal{E}_{\mathbb{A}}$ and $\mathcal{E}_{\mathbb{B}}$, respectively. Thus, the identity manager encrypts the hash values of \mathbb{A} and inserts the encrypted hash

values into the password database against the email ID $\mathcal{E}_{\mathbb{A}}$, as shown in Equation (32).

$$\mathcal{E}_{\mathbb{A}} \begin{cases} \mathcal{C}_{\mathscr{U}} \leftarrow AES.Enc(\mathcal{H}_{\mathscr{U}_{\mathbb{A}}}, \mathfrak{W}_{\mathbb{A}}) \\ \mathcal{C}_{\mathscr{P}} \leftarrow AES.Enc(\mathcal{H}_{\mathscr{P}_{\mathbb{A}}}, \mathfrak{W}_{\mathbb{A}}) \end{cases} \end{cases}$$
(32)

Similarly, the identity manager inserts the encrypted hash values of the user \mathbb{B} and inserts the encrypted hash values into the password database against the email ID $\mathcal{E}_{\mathbb{B}}$, as shown in Equation (33).

$$\mathcal{E}_{\mathbb{B}} \begin{cases} \mathcal{C}_{\mathscr{U}} \leftarrow AES.Enc(\mathcal{H}_{\mathscr{U}_{\mathbb{B}}}, \mathfrak{W}_{\mathbb{B}}) \\ \mathcal{C}_{\mathscr{P}} \leftarrow AES.Enc(\mathcal{H}_{\mathscr{P}_{\mathbb{B}}}, \mathfrak{W}_{\mathbb{B}}) \end{cases}$$
(33)

The $\mathcal{E}_{\mathbb{A}}.\mathcal{C}_{\mathcal{U}} \neq \mathcal{E}_{\mathbb{B}}.\mathcal{C}_{\mathcal{U}}$ and $\mathcal{E}_{\mathbb{A}}.\mathcal{C}_{\mathcal{P}} \neq \mathcal{E}_{\mathbb{B}}.\mathcal{C}_{\mathcal{P}}$ where dot (.) is membership operator for this representation. PassPro does not encrypt the user's email IDs because they are public. The email IDs are used to index the users' hash values in the password database.

2.12.2 Decryption

The identity manager receives $\mathcal{H}_{\mathscr{K}}$ to reproduce a mutually reproducible secret word, \mathfrak{W} , to retrieve $\mathcal{H}_{\mathscr{U}}$ and $\mathcal{H}_{\mathscr{P}}$ from the encrypted $\mathcal{C}_{\mathscr{U}}$ and $\mathcal{C}_{\mathscr{P}}$ from its database.

$$\mathcal{H}_{\mathscr{P}}^{decrypted} = AES.Dec(\mathcal{C}_{\mathscr{U}}, \mathfrak{W})$$
$$\mathcal{H}_{\mathscr{P}}^{decrypted} = AES.Dec(\mathcal{C}_{\mathscr{P}}, \mathfrak{W})$$
(34)

The decrypted hash values in Equation (34) are used to match the hash value sent from the user for authentication by a genuine user. Let an adversary produce a mutually reproducible secret word, say \mathfrak{W}' . The adversary decrypts the hash values as given in Equation (35).

$$\mathcal{AD}_{\mathscr{U}}^{decrypted} = AES.Dec(\mathcal{C}_{\mathscr{U}},\mathfrak{W}')$$
$$\mathcal{AD}_{\mathscr{P}}^{decrypted} = AES.Dec(\mathcal{C}_{\mathscr{P}},\mathfrak{W}')$$
(35)

where

$$\begin{aligned} \mathcal{H}_{\mathcal{U}}^{decrypted} &\neq \mathcal{AD}_{\mathcal{U}}^{decrypted} \\ \mathcal{H}_{\mathcal{U}}^{decrypted} &\neq \mathcal{AD}_{\mathcal{P}}^{decrypted} \end{aligned}$$
(36)

Therefore, the adversary cannot decrypt the stored hash values. The adversary cannot decrypt the hash values of the other users even if the adversary can decrypt the hash value of a particular user.

3 Dishonest Identity Manager

To elaborate on the domino effect and misuse of stolen users' credentials, we assume a dishonest identity manager, *DisIDM*. Let the purpose of the *DisIDM* be to gain authentication in

other honest identity managers by using its own clients' identities and passwords where the clients do not know about the identity manager. Thus, the *DisIDM* collects the user IDs and passwords. This assumption is significant in understanding the capability of our proposed method. Let a user \mathbb{A} create an account in the dishonest identity manager DisIDM. There are two cases: a) the *DisIDM* authenticates using the state-of-the-art authentication mechanism, and b) the DisIDM authenticates using PassPro. Firstly, the DisIDM can receive the user's credentials in raw form, and the received credentials are stored in different databases in raw form because the DisIDM is a dishonest authentication manager. After storing it, the *DisIDM* can store the passwords by applying the salt to them. Therefore, the objective of the DisIDM is successful. In PassPro, the user \mathbbm{A} produces three hash values, and these are $\mathcal{H}_{\mathcal{H}}^{DisIDM}$, $\mathcal{H}_{\mathcal{P}}^{DisIDM}$, and $\mathcal{H}_{\mathcal{K}}^{DisIDM}$. These hash values are sent to the DisIDM using the POST method. The DisIDM receives the three hash values and stores these three hash values separately because we assume that the DisIDM is a dishonest identity manager. Let the user \mathbb{A} create an account in an honest identity manager (*HoIDM*). The A produces three hash values $\mathcal{H}_{\mathscr{U}}^{HoIDM}$, $\mathcal{H}_{\mathscr{P}}^{HoIDM}$, and $\mathcal{H}_{\mathscr{K}}^{HoIDM}$, and sends these three hash values to the HoIDM using the POST method. The *HoIDM* stores the hash values in the password database by encrypting using a mutually reproducible secret word. The \mathbb{A} keeps the same user ID and passwords for both identity managers (DisIDM and HoIDM). PassPro shuffles the password and identity with a domain word, and the domain word for the DisIDM and HoIDM is different. Therefore, it is guaranteed that the hash values for two different domains will always be different. Particularly, it guarantees Equation (37).

$$\begin{aligned}
\mathcal{H}_{\mathscr{U}}^{DisIDM} &\neq \mathcal{H}_{\mathscr{U}}^{HoIDM} \\
\mathcal{H}_{\mathscr{P}}^{DisIDM} &\neq \mathcal{H}_{\mathscr{P}}^{HoIDM} \\
\mathcal{H}_{\mathscr{K}}^{DisIDM} &\neq \mathcal{H}_{\mathscr{K}}^{HoIDM}
\end{aligned} (37)$$

Therefore, the *DisIDM* cannot gain authentication in the *HoIDM* by using its own clients' credentials. This assumption shows that the adversary cannot use the credentials of stolen identities and passwords to gain authentication in another identity manager. The *DisIDM* can also try to get the raw form of a user ID and password by launching different types of attacks on the supplied hash values from the user, but it is computationally hard to reproduce the original strings from the hash values. The assumption of the dishonest identity manager proves the superiority of our proposed work over the state-of-the-art authentication system.

4 Password database leakage

Let us assume that the adversary evades the security of the identity manager and has stolen the password database. The adversary tries to remove the salts from the passwords in the state-of-the-art identity manager. However, the removal of

salt is hard and computationally infeasible. Therefore, the adversary performs rainbow table attacks to reveal the password. Often, the adversary becomes successful by the rainbow table in the state-of-the-art identity manager, for instance, COMB [9] and RockYou2021 [10]. Let us assume that the password database is given to the adversary. At first, the adversary requires two words: the $\mathcal{H}_{\mathscr{K}}$ from the user and the master secret word from the identity manager. For simplicity, we assume that the master secret word is only one throughout the password database. Let us assume that the adversary decoded the master secret word from the identity manager. Let the adversary try to decrypt the hash values of the user \mathbb{A} . The adversary cannot decrypt the hash values of \mathbb{A} without the $\mathcal{H}_{\mathcal{K}}$ even if the adversary has the master secret word. Let us assume that the adversary is able to reproduce the mutually reproducible secret word of \mathbb{A} , $\mathfrak{W}_{\mathbb{A}}$; then the adversary is able to decrypt the hash values of the \mathbb{A} . However, it does not imply that the adversary can decrypt the hash values of other users. PassPro encrypts the hash values of each user with an individual mutually reproducible secret word. The password database is intact even though the adversary evades the security of the identity manager. We assumed the $\mathcal{H}_{\mathcal{K}}$ has with the adversary, which is computationally hard.

5 Conclusion

In this paper, we presented a secure password-based authentication called PassPro. In PassPro, an adversary cannot use the derived credentials of a user from the leaked database in multiple domains to gain authentication. Thus, it encourages users to reuse the same user ID and password in multiple domains. It guarantees that the hash values in an identity manager are invalid in other identity managers. Moreover, it is computationally hard to retrieve the original user ID and password from the PassPro. Consequently, it prevents adversaries from publishing the passwords. We have also presented the idea that PassPro encrypts the users' hash values with a mutually reproducible secret word. The mutually reproducible secret word is produced or reproduced mutually by the user's hash value and the master secret word of the identity manager. Consequently, the adversary cannot decrypt the hash values even if the identity manager provides the adversary with the password database. Moreover, the adversary fails in the decryption process of the encrypted hash values even if it succeeds in retrieving the master secret word from the identity manager. Furthermore, our shuffling algorithm is designed based on a pseudo-random algorithm to shuffle the user ID, password, and domain word, where legitimate users can consistently reproduce the order of the shuffled string. The shuffling process creates unique strings even if the same user ID and password are used in multiple domains. Finally, we demonstrated the security of PassPro by assuming a dishonest identity manager, and this assumption leads to tighter security than the state-of-the-art password-based authentication methods.

Open Science Policy

The source code of the project will be made available on GitHub if the paper is accepted.

Acknowledgments

The research work of Dr. Ripon Patgiri is supported by the Science and Engineering Research Board (SERB), Government of India, under grant Number EEQ/2021/000694.

References

- [1] Austin Appleby. murmurhash. [Online], Retrieved on December 2022 from https://sites.google.com/ site/murmurhash.
- [2] Mihir Bellare and Laura Shea. Flexible password-based encryption: Securing cloud storage and provably resisting partitioning-oracle attacks. In Mike Rosulek, editor, *Topics in Cryptology – CT-RSA 2023*, pages 594–621, Cham, 2023. Springer International Publishing.
- [3] Alex Biryukov, Daniel Dinu, and Dmitry 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), pages 292–302, 2016.
- [4] Jeremiah Blocki and Anupam Datta. Cash: A cost asymmetric secure hash algorithm for optimal password protection. In 2016 IEEE 29th Computer Security Foundations Symposium (CSF), pages 371–386, 2016.
- [5] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In NDSS, volume 14, pages 23–26, 2014.
- [6] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. Passgan: A deep learning approach for password guessing. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 217– 237, Cham, 2019. Springer International Publishing.
- [7] Wenjian Luo, Yamin Hu, Hao Jiang, and Junteng Wang. Authentication by encrypted negative password. *IEEE Transactions on Information Forensics and Security*, 14(1):114–128, 2019.
- [8] William Melicher, Blase Ur, Sean M. Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In 25th USENIX Security Symposium (USENIX Security 16), pages 175–191, Austin, TX, August 2016. USENIX Association.

- [9] Bernard Meyer. COMB: over 3.2 Billion Email/Password Combinations Leaked | Cybernews. [Online], available at https: //cybernews.com/news/largest-compilationof-emails-and-passwords-leaked-free, July 2022.
- [10] Edvardas Mikalauskas. RockYou2021: Largest Ever Password Compilation Leaked | Cybernews, July 2022. [Online], Available at https://cybernews.com/ security/rockyou2021-alltime-largestpassword-compilation-leaked.
- [11] Dario Pasquini, Marco Cianfriglia, Giuseppe Ateniese, and Massimo Bernaschi. Reducing bias in modeling real-world password strength via deep learning and dynamic dictionaries. In 30th USENIX Security Symposium (USENIX Security 21), pages 821–838. USENIX Association, August 2021.
- [12] Dario Pasquini, Ankit Gangwal, Giuseppe Ateniese, Massimo Bernaschi, and Mauro Conti. Improving password guessing via representation learning. In 2021 IEEE Symposium on Security and Privacy (SP), pages 1382–1399, 2021.
- [13] C Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan The BSD Conference*, 2009.
- [14] C. Percival and S. Josefsson. The scrypt Password-Based Key Derivation Function, August 2016. [Online; accessed 16 April 2023].
- [15] Javier Rando, Fernando Perez-Cruz, and Briland Hitaj. Passgpt: Password modeling and (guided) generation with large language models, 2023.
- [16] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C Mitchell. Stronger password authentication using browser extensions. In 14th USENIX Security Symposium (USENIX Security 05), Baltimore, MD, July 2005. USENIX Association.
- [17] Emin Islam Tatli. Cracking more password hashes with patterns. *IEEE Transactions on Information Forensics and Security*, 10(8):1656–1665, 2015.
- [18] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1242–1254, New York, NY, USA, 2016. Association for Computing Machinery.
- [19] Ke Coby Wang and Michael K. Reiter. How to end password reuse on the web. In *Proceedings 2019 Network and Distributed System Security Symposium*, pages 1–15, 2019.

[20] Ming Xu, Chuanwang Wang, Jitao Yu, Junjie Zhang, Kai Zhang, and Weili Han. Chunk-level password guessing: Towards modeling refined password composition representations. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 5–20, New York, NY, USA, 2021. Association for Computing Machinery.

Appendix

Client-side password metering

PassPro does not store or deal with the raw form of user IDs and passwords. Therefore, PassPro cannot offer AJAX service to verify the strength of the password. Also, PassPro cannot suggest password vulnerabilities or recommend passwords. Instead, PassPro offers a client-side password-strength metering system to suggest the strength of the password to the user. Algorithm 3 imposes the rules of PassPro as described in

Algorithm 3 User ID or password strength checking algorithm. For password, $\mathcal{L} \ge 10$ and for user ID, $\mathcal{L} \ge 8$. The flag = 0 for user ID and flag = 1 for password.

```
1: procedure STRENGTHMETER(\omega, \mathcal{L}, flag)
        if flag = 1 then
                                            \triangleright Case for password.
 2:
 3:
            \ell = 10
                                               ⊳ Case for user ID.
 4:
        else
            \ell = 8
 5:
        end if
 6:
        if \mathcal{L} < \ell then
 7:
            Invalid length.
 8:
 9:
            Exit
10:
        end if
        if ISEMAIL(\boldsymbol{\omega}) = true then
11:
            An email ID cannot be the user ID or password.
12:
            Exit
13:
        end if
14 \cdot
        CC, SC, DC, SyC counts the capital letters, small let-
15:
    ters, digits, and symbols in \omega
        min = MINIMUM(CC, SC, DC, SyC)
16:
        if min < 1 then
17:
            Invalid
18:
19
        else if min = 1 then
            Weak
20.
        else if min = 2 then
21:
             Good
22:
        else if min = 3 then
23:
24:
            Strong
25:
        else
            Very strong
26:
        end if
27:
28: end procedure
```

subsection 2.3 and 2.4. Algorithm 3 checks the length of the user ID and password. The minimum length of the user ID is eight characters, and the minimum length of the password is ten characters. Also, it imposes restrictions on using an email ID as a user ID or password. Moreover, Algorithm 3 counts the capital letters, small letters, digits, and symbols used in the user ID and password. The CC, SC, DC, and SvC are the counters of capital letters, small letters, digits, and symbols present in the input string. The minimum value is used to decide the strength of the password or user ID. To qualify for user ID, the minimum value must be 1. Alternatively, the user ID must contain at least one small letter, one capital letter, one digit, and one symbol. Similarly, the minimum value for a password is also 1 to qualify as a password. Therefore, the password must contain at least a small letter, a capital letter, a digit, and a symbol.

ป	User ID, a secret word
\mathcal{P}	Password, a secret word
\mathcal{D}	A domain word which is public
$\mathcal{H}()$	A secure hash function, can be SHA256, SHA512 or SHA3
U	Shuffling of two words based on a context
$\xrightarrow{\mathcal{P}}$	The \mathcal{P} is a context during the shuffling process
	Concatenation of two strings
U	Shuffled word for user ID
P	Shuffled word for password
K	Shuffled word for a part of mutually reproducible secret key
$\mathcal{H}_{\mathcal{U}}$	Hash value of shuffled word of user ID
$\mathcal{H}_{\mathscr{P}}$	Hash value of shuffled word of password
$\mathcal{H}_{\mathscr{K}}$	Hash value of shuffled word of the part of the mutually reproducible secret key
\mathcal{W}	A master secret word from server, Identity manager
$\mathcal{H}_{\mathcal{W}}$	Hash value of the master secret word from the identity manager
SW	Shuffled word of the hash value of the master secret word of the identity manager and secret key
	supplied by the user
W	The secret key for encryption or decryption of the hash values of the user. It is a mutually reproducible
	secret word.
Cu	It is an encrypted ciphertext stored in the server's password database by encrypting the hash value of the user ID using \mathfrak{W} .
Cø	It is an encrypted ciphertext stored in the server's password database by encrypting the hash value of
	the password using \mathfrak{W} .
$\mathcal{H}^{decrypted}_{\mathscr{U}}$	A decrypted a hash value of user ID at the server side.
$\mathcal{AD}^{decrypted}_{\mathscr{U}}$	A decrypted a hash value of user ID by the adversary.
DisIDM	Assumed dishonest identity manager for analysis purposes.
$\mathcal{H}^{DisIDM}_{\mathscr{U}}$	An hash value of user ID by the assumed dishonest identity manager

Table 4: Symbols and their descriptions.