ISSN: 0974-8571 Vol.14 No. 1 June, 2022

International Journal of Computational Intelligence in Control

SecureNum: A Random Chunked PIN Resilient Against Offline Brute-Force Attack

Awais Ahmad[1] **Muhammad Asif**[2] **Isma Hamid**[3]

[1] 16ntu7003@student.ntu.edu.pk[2] asif@ntu.edu.pk ✉[3] ismahamid@ntu.edu.pk

Department of Computer Science

National Textile University, Faisalabad

Abstract

Habits of the password users do not match with this technology boost: still, the easy passwords are selected no matter the nature of use (sensitive vs insensitive). This problem invites the adversaries who brought modern methods of password cracking. In comparison, the security of passwords lies among encryption, hashing, salted passwords, key stretching and NDB (negative database). Now modern tools (hashcat, john-the-ripper)in combination with modern hardware (CPUs having more than 50 physical cores, GPUs with hundreds of cores) may assassinate current encryption and hashing methods within minutes. In the current situation, we have a dire need for a defence mechanism against these modern attacks. In this paper, we have proposed a two-layer defence method; 1st layer (login script) is resilient against *man-in-middle, replay, guessing* and *exhaustive* attacks, 2nd layer (custom encoding/decoding) a chunked PIN to tackle the brute-force, dictionary, lookup and rainbow attacks. The main idea is to segregate the PIN into several chunks and store it in multiple columns in the database table on a random basis. Our security analysis reveals how an 8-digit PIN may be expanded (to 51 alphabets) and have ***224.0532*** bits of entropy along with *2.7113403136065****E67***password space of plain secret. However, encoded variantsare sure to have higher values. Comparisons with state-of-the-art reveal SecureNum has comparable time complexity likeO(*n*), as opposed to the proposed brute-force attack (3 algorithms), which produces exponentially very high attack complexity(O(*n!*)^O(*n!*)^O(*n!*)) while requiring considerable time and cost.

Keywords: brute-force attack, dictionary attack, hashing, encryption

1. Introduction

PIN and Password authentication is considered to be low cost, easy to deploy and well manageable on every type of device e.g. Servers, PCs, Phones, IoT devices, etc. Scientists have conducted extensive research on the topic of password security, but human factors have remained the same to date[1]. Humans normally prefer to choose easy PINs and passwords due to the recall-ability problem. Furthermore, they have habitsof re-using the same password among various systems [2][3]. These human factors adversely invite impending threats through the potential leak holes. A prominent threat is the online brute-force[4], which was related to the third-person PC shootergame released in 2000.Their theme was to find and re-union the other reliable characters who have good strengths and capabilities. Brute-force attacks may be mainly categorized into dictionary and hybrid attacks[5].

Brute-force attacks may be flooded online or offline. Online attacks have a definite solution [4][6] in which the host may set a threshold of certain failed attempts for a definite time or to be unlocked only by the administrator.Offline brute-force attacks[5]are highly vulnerable because the attacker manages to snag the shadow copy or encrypted files from the target machine. After this, their next step is to obtain vulnerable datasets named precomputed wordlists, which are increasing tremendously and are considered helpful in cracking weak passwords in a few seconds. According to a resource [7], the rainbow table has TBs of data related to MD5 hashes.Rainbow tables are really helpful in cracking the password because adversaries generate the precomputed lookup table along with plain passwords. Next, they collect the offline hacked data and simply match the hashes of targeted data with a precomputed lookup table and obtain the secret credentials. Conferring to these hacked password files, literature is flooded [8][9][10][11][12][13] to modify (old) and design (new) sophisticated cracking tools i.e. hashcat [14], John the Ripper[15] by using the powerful arrays of GPUs [16] (having of thousands of cores) used for calculating and comparing each possible combination of particular alphabets. Brute-force may be considered more disastrous against dictionary passwords[17]. A modern GPU may crack 95% [3] of passwords in just a few days.

When first introduced for UNIX systems, passwords were stored in the backend database in plain text. Then in 1979, authors of [18] pointed out the potential threat of brute-force and concluded the encryption by DES (Data Encryption Standard) is too fast and secure against this attack. They also highlighted that technological advances in computing power pose a potential threat to password cracking but humans didn't learn the lesson and still choose the short and predictable secrets. After ten years, in the follow-up study [19], it was observed that cracking methods were improved a lot and the habits of humans were the same. They also proposed using strong user passwords as well as salts and most importantly, they suggested improvingthe entropy of passwords (detailed in the upcoming section).

Asa consequence, dictionaries of hacked passwords were augmented tremendously in the comparison of the same habits of humans which urges the researchers to invent certain methods of hashing and encryption. Hashing is the one-way process in which plaintext is scrambled into a message-digest using a hash function. MD5 is a message-digest algorithm which may take arbitrary input and generate 128-bit output. In comparison,a family of SHAs contains a cryptographic hash function with variable input and output lengths. MD5 and SHA1 are considered broken algorithms and are not suggested to use. In research of 2019 [20], a modified version of MD5 and SHA1 was proposed after applying a head and tail technique with the help of *fragmentation* and *concatenation.* (They borrowed this idea of fragmentation from research [21] published in 2017). They managed to produce a 512-bit hash after this modification. There is another secure hashing method specifically designed to encrypt passwords known as Bcrypt. It is a blowfish block cipher and is structured for 16 rounds. The input of Bcryptis 128-bit salt along with a password having 72 bytes max. Bcrypt is considered secure against brute-force but it is practically slow. In research [22], some low-power parallel devices to exploit Bcrypt peculiar.

On the other hand, encryption is a two-way cryptographic function that produces a cipher of variable length and the famous ones are RSA and AES. RSA is the oldest which applies a single round of encryption and has key sizes from 2048-4096 bits. AES is an advanced and highly adopted algorithm mostly by government and security agencies. It used 128, 192 and 256-bit same keys as input and output. In a research published in 2021 [23], the authors of the paper applied SHA as with RSA and AES to generate negative passwords against brute-force attacks.

1.1 Motivation to use hashing and fragmentation/concatenation

The key difference between hashing and encryption is related to the output, which is non-reversible in hashing and reversible in encryption. In password authentication, the practice of hashing is widely adopted in comparison to encryption. One reason is the non-reversible hash of hashing algorithms and the security of the same key (used for encryption and decryption) in the encryption algorithms is also a difficult task (because of brute-force). According to the idea of [24], traditional PBE (password-based encryption), which follows the standard of PKCS v.2.0 (public key cryptography standard) where practitioners mostly adopt PBKDF2 (Password-Based Key Derivation Function 2) to derive encryption and decryption keys, are highly susceptible to brute-force attack. Because at the time of decryption, unless the correct key is applied, there is no decryption. But in their proposal of honey encryption, the list of fake passwords with bogus accounts is generated after applying a non-valid key.

According to our observation, this idea may be good for online attacks but in offline attacks where the attacker may obtain hash files may set their criteria for cracking the secret. Thus in our proposal, we designed the backend system with automatic hashing with SHA3-224 having a salt of 64base. Later we further adopted the techniques of *fragmentation* and *concatenation*[20][21]detailed in the prior section. The proposal of [21] is costly because they used multiple servers and the limitation is that the servers are on the cloud. The work of [20] is upon securing the MD5 and SHA1 while there are more secure hashing available in the market i.e. SHA3.

In this paper, we have proposed a new custom encoding scheme named SecureNum, which distributes the secret into random hashed chunks and stores it in the backend database table again on a random basis. Some further add-ons (salt + random number generator) are also appended with those chunks. There are two contributions of this paper 1) introduction of a secure PIN protection method named SecureNum with the rotation + fragmentation + incremental swapping algorithms, 2) analysis and comparison of the attack complexity (time + cost) regarding salted + hashed + key stretching + SecureNum.

The paper is organized as follows. In the second section, we cover the related works and continue to the third section which introduces the proposed methodology. The fourth section explains the implementation of SecureNum while the fifth section elaboratesthe comprehensive analyses as discussions and comparisons. Lastly,the seventh section concludes the paper.

2. Related Works

Passwords are the primary methods considered for authentication purposes. In history, passwords were stored in plain format which inevitably were in the sight of attackers. There are a bundle of examples of password leakages. One solution is to change the password regularly. According to research [25], There are 11 websites which are Alexa's top 500 list which store passwords in plain text. They further investigated and found that 135 academic websites were also having the same practice. According to the studies of 2017 and 2018, computer science students do not bother about storing the password safely [26]. There are many techniques to store passwords as secure and the notable ones are password encryption, password hashing and key stretching.

Encrypted Passwords: In the early days, passwords were stored in encrypted format using RSA or AES algorithms. In these techniques, the problem of key management (used to decrypt)still exists[25].The key servers where the key is stored are also at stake.

Hashed Passwords: Password hashing is the most popular method to store passwords because hashes can't be decrypted with modern hardware machines. At the time of verification, hashes are matched for comparison purposes. There are many one-way cryptographic algorithms out there e.g. MD5, SHA family. The hashed passwords were considered safe in history but some cracking techniques like lookup tables or rainbow tables have becomea nightmare for hashed passwords [27]. As the processing power is being improved, the success of password cracking of hashed passwords is also increased [28].

Salted Passwords: Precomputation attacks may be rectified by the use of salted passwords [27]. Salt is a random string of plain or hashed data, concatenated with the password, after which a cryptographic hash function is applied to obtain the final hash. The size of the salt decides the strength of the hash. In a study of 2021 [29], 138 developers were asked to write code to secure the password, only 14% used the salts of whichmerely 7% utilized the random salts. Salt is an additional burden for developers to keep these as secure. Furthermore, dictionary attacks may infer devastating effects even on salted passwords.

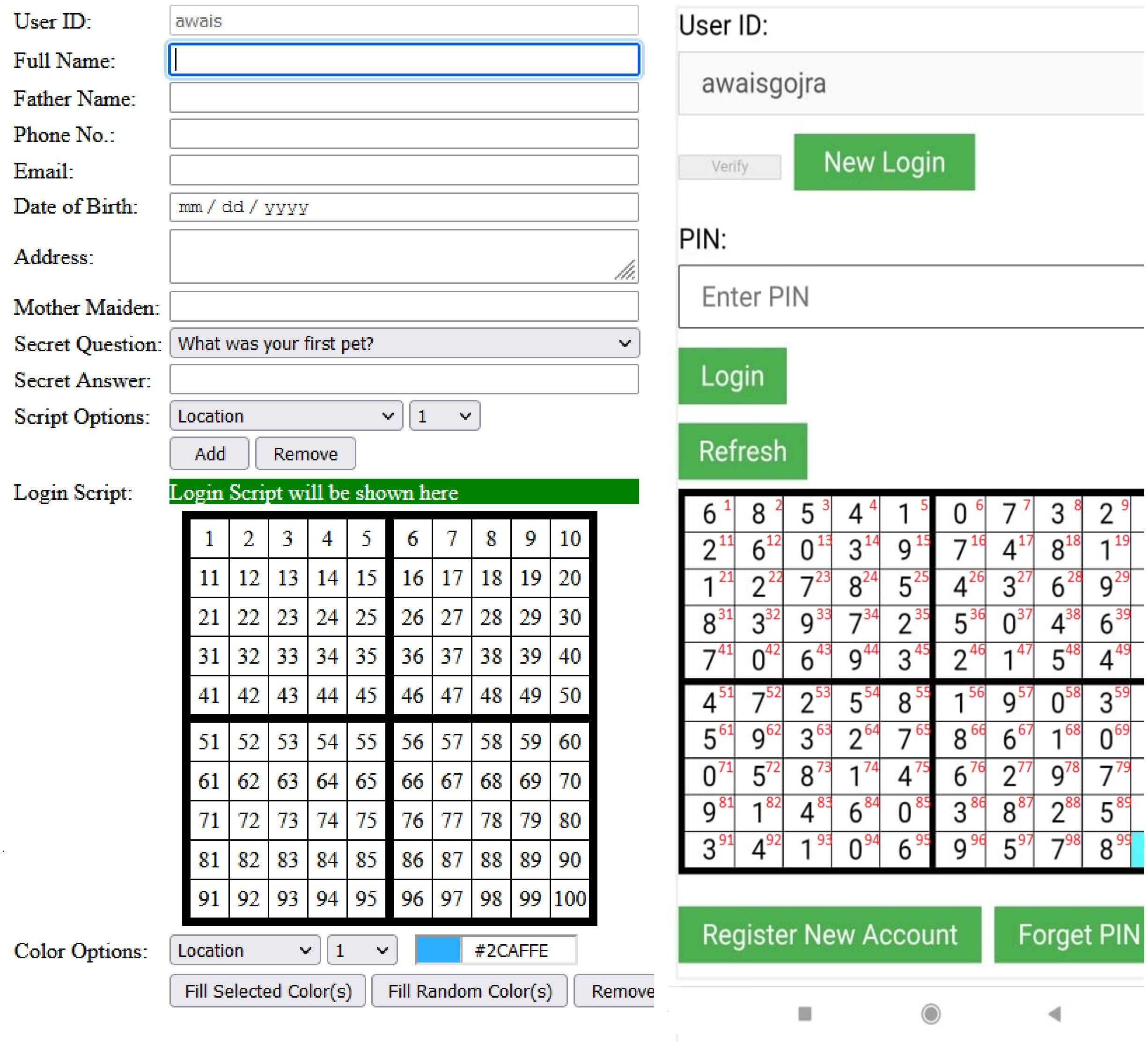
Key stretching: This technique may be used with hashed passwords or salted passwords. Key stretching applies multiple rounds on the hashes produced with the previous detailed methods. The common techniques are PBKDF2, Bcrypt and Scrypt. These techniques are CPU-intensive and considered to be slow. In a study [30], authors revealed that 50% of CPU power may be saved on cracking the PBKDF2 algorithm.

Negative Database: Abbreviated as NDB, this technique is considered to be safe against brute-force attacks when merged with the previously discussed scheme. NDB works on the replacement of bits where 0 and 1 may be replaced by the same but another symbol * may be replaced on both 0 and 1. It means the position of * is unspecified. In many studies [31][32][33], authors propose to secure the password using custom NDBs. These solutions have some demerits on the usability side like the liability for a particular user to generate the random numberor to select the custom hashing techniques on the run time. (not recommended for naïve users)

3. Proposed Method

The proposed scheme aka SecureNum, is a graphical PIN authentication method, equipped with a custom encoding mechanism for the secret being stored in a backend database. The PIN on SecureNum is dynamic because, on every new login session, users need to enter different PINs for successful authentication. This dynamic PIN is backed by a static login-script which is mapped to a grid of 100 cells (**Figure 1**).

Login-script: We named the secret of SecureNum as login-script which is an alphanumeric string. Login-script is generated automatically upon the selection of particular cells on the grid of SecureNum. It also offers arithmetic operations (plus, minus, multiplication, division) and fake numbers (any length of digits). There are two formats of the login-script (**Figure 2**), one for user view and the other one for backend use. E.g. A user sets the cells of 44, 45, 54 and 55 (four cells). S/he also uses the plus arithmetic operations (44+1, 45+2, 54+3, 55+4) on the respective cells along with fake numbers (1, 2, 3, 4). The final user viewable login-script is shown in **Figure 2.**Which is reconfigured for the backend system automatically in the shorter format. With the increase of GPU-based brute-force attacks, the length of this login-script (generated for only 4 cells) proved to be resilient in encoded format as well as in plain format. We will analyze its entropy and password space in the coming section.



User ID: awais
Full Name:
Father Name:
Phone No.:
Email:
Date of Birth: mm / dd / yyyy
Address:
Mother Maiden:
Secret Question: What was your first pet?
Secret Answer:
Script Options: Location 1
Add Remove
Login Script: Login Script will be shown here

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Color Options: Location 1 #2CAFFE
Fill Selected Color(s) Fill Random Color(s) Remove

(a)

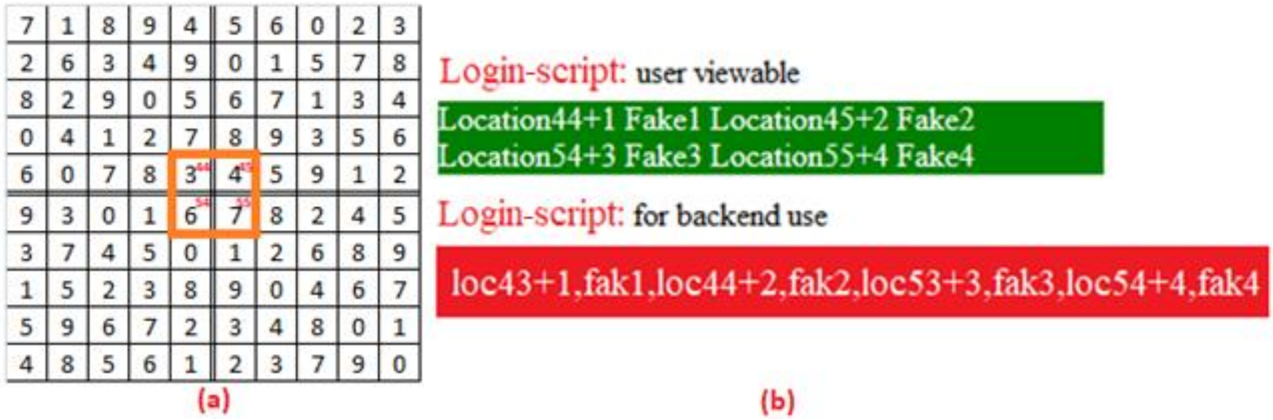
User ID:
awaisgojra
Verify New Login
PIN:
Enter PIN
Login
Refresh

6^{1}	8^{2}	5^{3}	4^{4}	1^{5}	0^{6}	7^{7}	3^{8}	2^{9}
2^{11}	6^{12}	0^{13}	3^{14}	9^{15}	7^{16}	4^{17}	8^{18}	1^{19}
1^{21}	2^{22}	7^{23}	8^{24}	5^{25}	4^{26}	3^{27}	6^{28}	9^{29}
8^{31}	3^{32}	9^{33}	7^{34}	2^{35}	5^{36}	0^{37}	4^{38}	6^{39}
7^{41}	0^{42}	6^{43}	9^{44}	3^{45}	2^{46}	1^{47}	5^{48}	4^{49}
4^{51}	7^{52}	2^{53}	5^{54}	8^{55}	1^{56}	9^{57}	0^{58}	3^{59}
5^{61}	9^{62}	3^{63}	2^{64}	7^{65}	8^{66}	6^{67}	1^{68}	0^{69}
0^{71}	5^{72}	8^{73}	1^{74}	4^{75}	6^{76}	2^{77}	9^{78}	7^{79}
9^{81}	1^{82}	4^{83}	6^{84}	0^{85}	3^{86}	8^{87}	2^{88}	5^{89}
3^{91}	4^{92}	1^{93}	0^{94}	6^{95}	9^{96}	5^{97}	7^{98}	8^{99}

Register New Account Forget PIN

(b)

Figure 1: Registration (a) and login (b) processes of the working prototype of SecureNum



7	1	8	9	4	5	6	0	2	3
2	6	3	4	9	0	1	5	7	8
8	2	9	0	5	6	7	1	3	4
0	4	1	2	7	8	9	3	5	6
6	0	7	8	3^{44}	4^{45}	5	9	1	2
9	3	0	1	6^{54}	7^{55}	8	2	4	5
3	7	4	5	0	1	2	6	8	9
1	5	2	3	8	9	0	4	6	7
5	9	6	7	2	3	4	8	0	1
4	8	5	6	1	2	3	7	9	0

Login-script: user viewable

Location44+1 Fake1 Location45+2 Fake2
Location54+3 Fake3 Location55+4 Fake4

Login-script: for backend use

loc43+1,fak1,loc44+2,fak2,loc53+3,fak3,loc54+4,fak4

Figure 2: Two types of login-scripts

Registration: SecureNum includes the processes of registration and login. New users need to register themselves by inputting the information shown in **Figure 1 (a).**After verifying the user ID, the user is needed to fill in their personal and security details. Upon filling in the complete information, the user is needed to generate the login-script by selecting the single or combination of cells. Next, the user simply taps/clicks the submit button.After this, all the information including login-script is *encoded* through the proposed mechanism and stored in the backend database table.Moreover, to further minimize the brute-force attempt, the metadata (column names of the table) is also hashed with the same method.

Login: Upon the authentication time**Figure 1 (b)**, the user verifies the user ID, enters the dynamic PIN (generates cognitively according to the stored login-script) and taps/clicks the login button. This time reverse processes (*decoding*) are executed with an additional process (mapping of dynamic PIN to login-script).

4. Implementation

In this section, we propose the implementation of SecureNum using the .NET libraries along with the SQL Server database. We used the HP Probook 450 laptop, having an I5 CPU and 16 GB RAM.

4.1 Overview

The backend design of SecureNum is handled by a custom-developed encoding/decoding module which has multiple layers of security. This module starts to work from metadata hashing through SHA3-224 (56 characters as output) in addition to salt. In comparison, user data (login-script) is highly secured with three layers. In the first layer, login-script is encoded in incremental format through a *rotation algorithm*. In the second layer, characters of login-script are distributed to 5 strings of repeating clockwise distribution format into the random positions (5-digit random numbers). In the last third layer, a *rotation algorithm* is applied to these 5 strings in addition to 5 different salts having 44 characters. Finally, all the user data is stored in this encoded format to the backend database relation.

4.1.1 Rotation algorithm

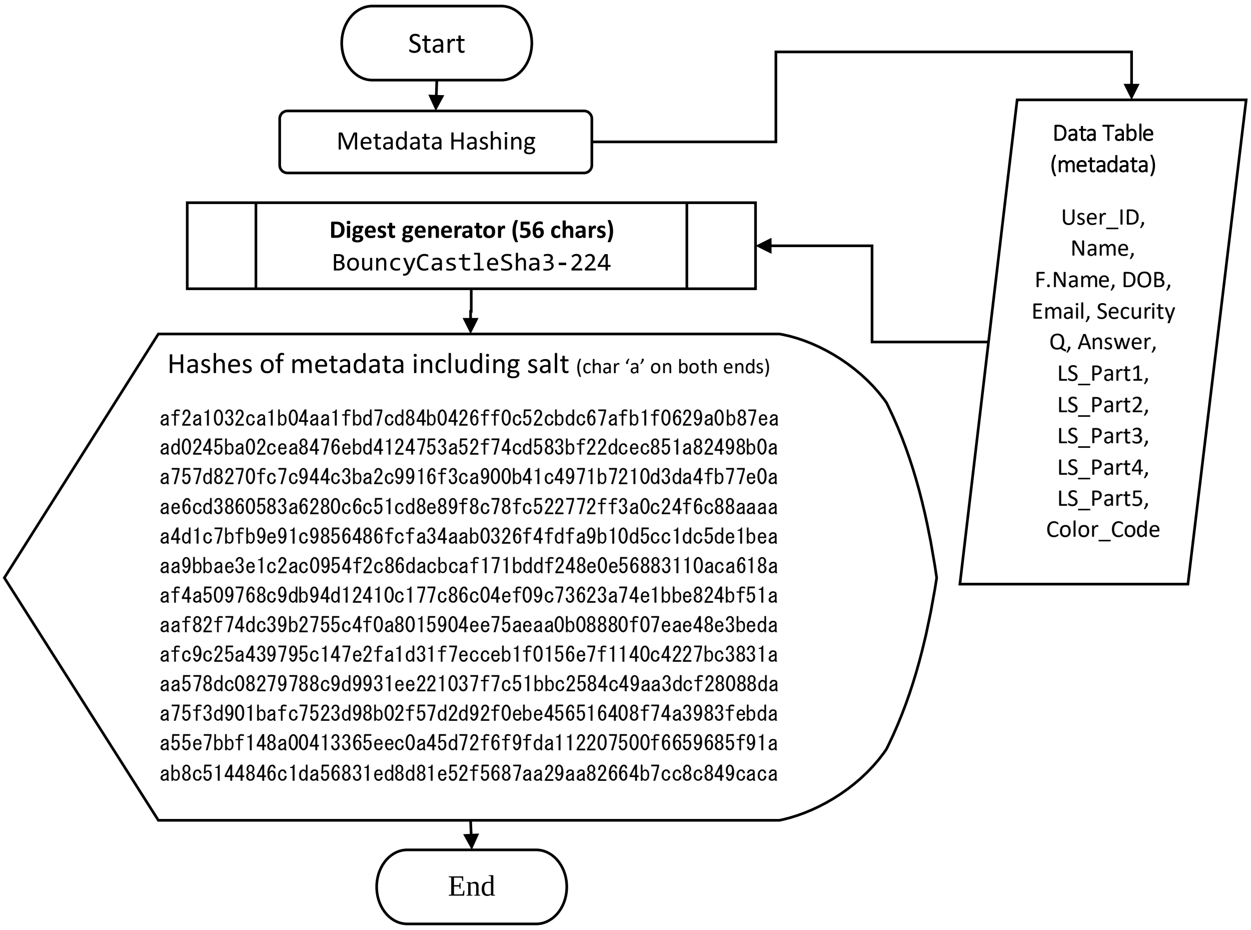
The rotation algorithm fills an empty string with rotated alphabets. It starts with generating, reversing and removing the duplicates from the salt (44 characters). After that, two strings named original_chars and shuffle_chars of the same 88 characters containing alphabets, numbers and special characters are used. In the first loop, the characters appearing as same in the salt are removed from the string shuffle_chars and salt is appended at the start of the same string. This step is ensured to generate further randomness. In the last step, the second loop is responsible for swapping the requested string from orginial_chars to shuffle_chars.

Rotation Algorithm: Character swapping

```
Input  :Empty string.
Output : Filled string of swapped characters
Initialization salt;
original_chars;
shuffle_chars;
stringtorotate;
rotatedPass;
salt←unique_rev_salt(salt);
original_chars←alphabets, numbers and special characters (88 characters);
shuffle_chars←Shuffled alphabets, numbers and special characters (88 characters);
for ( i=0 to length.salt) do
    if (indexof(shuffle_chars)>= 0 ) then
        shuffle_chars←indexof(remove(shuffle_chars);
    end
end
shuffle_chars = salt + shuffle_chars
for ( i=0 to length.stringtorotate) do
    if ( stringtorotate>= 0 ) then
        rotatedPass←rotatedPass+ shuffle_char;
    end
end
```

4.2 Working mechanism

SecureNum ensures two-fold security after applying encoding on both metadata and user data. Encoding of metadata is done once through the 56-character digest generator (SHA3-224) with the library of BouncyCastle. The whole process is self-explanatory and presented in**Figure 3**. The backend data table has fields of *user-id, name, f.name, dob, email, security q, answer, ls_part1, ls_part2, ls_part3, ls_part4, ls_part5 and color_code.* Hashes are also shown in the same figure. Before saving the hashed metadata into the physical data table, we faced a trouble that some of the hashes were commenced by digits which was not supported in SQL Server naming conventions. Thus we added the character 'a' on both ends of all hashes. These hashes were generated by using a salt value.


Start
Metadata Hashing
Data Table (metadata)
User_ID, Name, F.Name, DOB, Email, Security Q, Answer, LS_Part1, LS_Part2, LS_Part3, LS_Part4, LS_Part5, Color_Code
Digest generator (56 chars)
BouncyCastleSha3-224
Hashes of metadata including salt (char 'a' on both ends)
End

```
af2a1032ca1b04aa1fbd7cd84b0426ff0c52cbdc67afb1f0629a0b87ea
ad0245ba02cea8476ebd4124753a52f74cd583bf22dcec851a82498b0a
a757d8270fc7c944c3ba2c9916f3ca900b41c4971b7210d3da4fb77e0a
ae6cd3860583a6280c6c51cd8e89f8c78fc522772ff3a0c24f6c88aaaa
a4d1c7bfb9e91c9856486fcfa34aab0326f4fdfa9b10d5cc1dc5de1bea
aa9bbae3e1c2ac0954f2c86dacbcaf171bddf248e0e56883110aca618a
af4a509768c9db94d12410c177c86c04ef09c73623a74e1bbe824bf51a
aaf82f74dc39b2755c4f0a8015904ee75aeaa0b08880f07eae48e3beda
afc9c25a439795c147e2fa1d31f7ecceb1f0156e7f1140c4227bc3831a
aa578dc08279788c9d9931ee221037f7c51bbc2584c49aa3dcf28088da
a75f3d901bafc7523d98b02f57d2d92f0ebe456516408f74a3983febda
a55e7bbf148a00413365eec0a45d72f6f9fda112207500f6659685f91a
ab8c5144846c1da56831ed8d81e52f5687aa29aa82664b7cc8c849caca
```

Figure 3: Flow diagram of metadata hashing

User data hashing/encoding (**Figure 4**) is somewhat technical in which the main party is the *rotation algorithm.* This process is started by inputting the plain login-script (set by the particular user, at the time of registration) into the *fragmentation process 1.* This process splits the login-script into comma-separated chunks and pushes forward to the process of *incremental character swapping aka ICS (all the game lies here).* The ICS process has two lists of 88 characters, each from ASCII-95 characters. Space, back-quote, single-quote, double-quote, back-slash, greater-than and lesser-than are omitted due to the string manageability issue. List 1 is in ordered format while list 2 is dynamically random for each user. Its randomness is achieved with the two tasks, firstly, the reversed string of unique salt is generated then all the characters appearing in salt are removed from list 2. Secondly, salt (44 characters of base64) is received from the *salt generator* and is appended with the list 2. This way list 1 and List 2, both have the same set of characters but List 2 has randomized characters. Next, the actual process started by swapping the characters appearing in List 1 with List 2 of the first chunk received from *fragmentation process 1.* Next,the chunk is received from the same process and then these old swapped and new un-swapped chunks are forwarded to the *concatenation process* which again merges the received ones and forwards back to ICS. The swapping process is done again in ICS on these two merged chunks simultaneously. In this way, the incremental character swapping is done on all chunks of login-script and a single encoded login-script is generated.

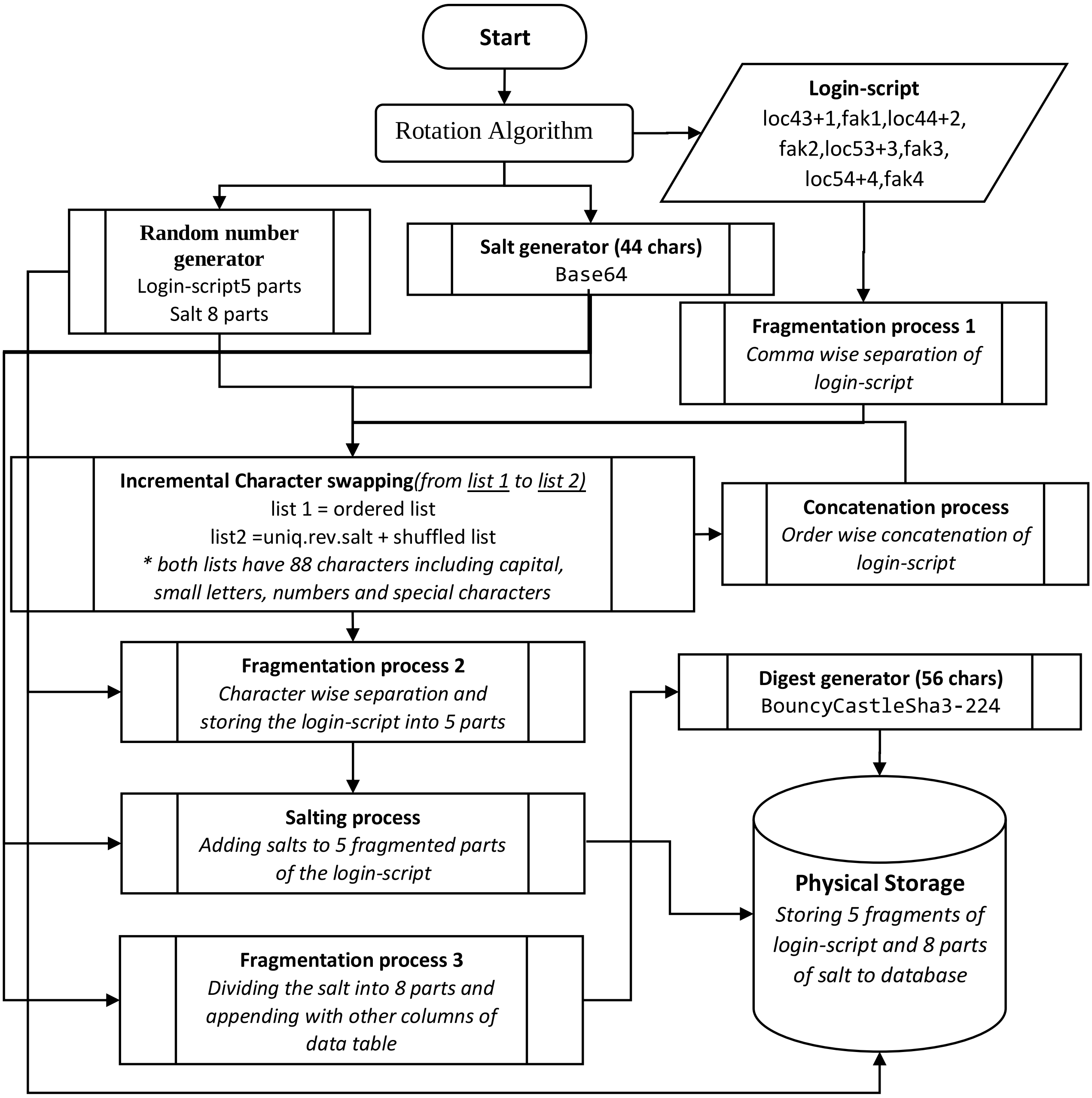
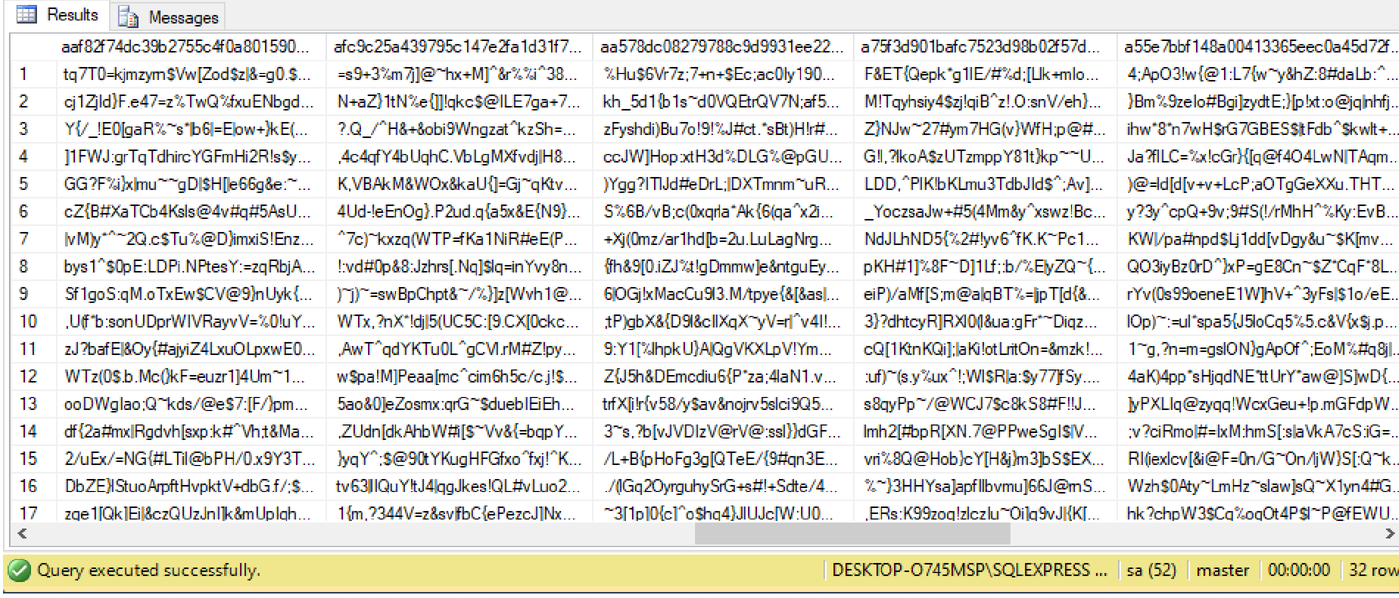

Start
Rotation Algorithm
Login-script
loc43+1,fak1,loc44+2,
fak2,loc53+3,fak3,
loc54+4,fak4
Random number generator
Login-script5 parts
Salt 8 parts
Salt generator (44 chars)
Base64
Fragmentation process 1
Comma wise separation of login-script
Incremental Character swapping(from list 1 to list 2)
list 1 = ordered list
list2 =uniq.rev.salt + shuffled list
* both lists have 88 characters including capital, small letters, numbers and special characters
Concatenation process
Order wise concatenation of login-script
Fragmentation process 2
Character wise separation and storing the login-script into 5 parts
Digest generator (56 chars)
BouncyCastleSha3-224
Salting process
Adding salts to 5 fragmented parts of the login-script
Physical Storage
Storing 5 fragments of login-script and 8 parts of salt to database
Fragmentation process 3
Dividing the salt into 8 parts and appending with other columns of data table

Figure 4: Secure hashed/custom encoded design against offline brute-force

Another duty of the ICS is to receive a string of 5 random digits (1 to 5) from a *random number generator* and maintain two copies of it, swapped and un-swapped. Afterwards, the encoded login-scriptwith an un-swapped random string is dispatched to *fragmentation process 2* which is responsible for shifting characters of login-script to 5 random parts (according to the random string received). This process is character-specific of the login-script and finally, data is pushed to the *salting process*. Lastly, 5 new unique salts are produced in the *salt generator* and sent out to the *salting process*. This process appended the final 5 parts of login-script with 5 unique salts separately and pushed to the data table (*physical storage*) to store this encoded data. The final shape of these 5 parts of login-script is shown in**Figure 5**. There is another process named *fragmentation process 3*, responsible for splitting out the original salt into 8 chunks, merging these chunks with the data of other columns of the datatable then forwarding it to the *digest generator* (56 characters with sha3-224) and finally destined to *physical storage.*



	aaf82f74dc39b2755c4f0a801590...	afc9c25a439795c147e2fa1d31f7...	aa578dc08279788c9d9931ee22...	a75f3d901bafc7523d98b02f57d...	a55e7bbf148a00413365eec0a45d72f...
1	tq7T0=kjmzym$Vw[Zod$z|&=g0.$...	=s9+3%m7j]@~hx+M]^&r%%i^38...	%Hu$6Vr7z;7+n+$Ec;ac0ly190...	F&ET{Qepk*g1lE/#%d;[Llk+mlo...	4;ApO3!w{@1:L7{w~y&hZ:8#daLb:^...
2	cj1Zj|d}F.e47=z%TwQ%fxuENbgd...	N+aZ}1tN%e{[]!qkc$@ILE7ga+7...	kh_5d1{b1s~d0VQEtrQV7N;af5...	M!Tqyhsiy4$zj!qiB^z!.O:snV/eh}...	}Bm%9zelo#Bgi]zydtE;}[p!xt:o@jq|nhfj...
3	Y{/_!E0[gaR%~s*|b6|=E|ow+}kE(...	?.Q_/^H&+&obi9Wngzat^kzSh=...	zFyshdi)Bu7o!9!%J#ct.*sBt)H!r#...	Z}NJw~27#ym7HG(v}WfH;p@#...	ihw*8*n7wH$rG7GBES$|tFdb^$kwlt+...
4	]1FWJ:grTqTdhircYGFmHi2R!s$y...	,4c4qfY4bUqhC.VbLgMXfvdj|H8...	ccJW]Hop:xtH3d%DLG%@pGU...	G!|,?lkoA$zUTzmppY81t}kp~~U...	Ja?flLC=%x!cGr}{[q@f4O4LwN|TAqm...
5	GG?F%i}x|mu~~gD|$H[|e66g&e:~...	K,VBAkM&WOx&kaU{]=Gj~qKtv...	)Ygg?ITlJd#eDrL;|DXTmnm~uR...	LDD,^PIK!bKLmu3TdbJld$^;Av]...	)@=ld[d[v+v+LcP;aOTgGeXXu.THT...
6	cZ{B#XaTCb4Ksls@4v#q#5AsU...	4Ud-!eEnOg}.P2ud.q{a5x&E{N9}...	S%6B/vB;c(0xqrla*Ak{6(qa^x2i...	_YoczsaJw+#5(4Mm&y^xswz!Bc...	y?3y^cpQ+9v;9#S(!/rMhH^%Ky:EvB...
7	|vM)y*^~2Q.c$Tu%@D}imxiS!Enz...	^7c)~kxzq(WTP=fKa1NiR#eE(P...	+Xj(0mz/ar1hd[b=2u.LuLagNrg...	NdJLhND5{%2#!yv6^fK.K~Pc1...	KW|/pa#npd$Lj1dd[vDgy&u~$K[mv...
8	bys1^$0pE:LDPi.NPtesY:=zqRbjA...	!:vd#0p&8:Jzhrs[.Nq]$lq=inYvy8n...	{fh&9[0.iZJ%t!gDmmw]e&ntguEy...	pKH#1]%8F~D]1Lf;:b/%ElyZQ~{...	QO3iyBz0rD^}xP=gE8Cn~$Z*CqF*8L...
9	Sf1goS:qM.oTxEw$CV@9}nUyk{...	)~j)~=swBpChpt&~/%}]z[Wvh1@...	6|OGj!xMacCu9l3.M/tpye{&[&as|...	eiP)/aMf[S;m@a|qBT%=|jpT[d{&...	rYv(0s99oeneE1W]hV+^3yFs|$1o/eE...
10	,U(f*b:sonUDprWIVRayvV=%0!uY...	WTx,?nX*!dj|5(UC5C:[9.CX[0ckc...	;tP)gbX&{D9l&cllXqX~yV=r|^v4!!...	3}?dhtcyR]RXl0(l&ua:gFr*~Diqz...	lOp)~:=ul*spa5{J5loCq5%5.c&V{x$j.p...
11	zJ?bafE|&Oy{#ajyiZ4LxuOLpxwE0...	,AwT^qdYKTu0L^gCVl.rM#Z!py...	9:Y1[%lhpkU}A|QgVKXLpV!Ym...	cQ[1KtnKQi];|aKi!otLritOn=&mzk!...	1~g,?n=m=gslON}gApOf^;EoM%#q8j|...
12	WTz(0$.b.Mc()kF=euzr1]4Um~1...	w$pa!M]Peaa[mc^cim6h5c/c.j!$...	Z{J5h&DEmcdiu6{P*za;4laN1.v...	:uf)~(s.y%ux^!;WI$R\|a:$y77]fSy....	4aK)4pp*sHjqdNE*ttUrY*aw@]S]wD{...
13	ooDWglao;Q~kds/@e$7:[F/}pm...	5ao&0]eZosmx:qrG~$duebIEiEh...	trfX[i!r{v58/y$av&nojrv5slci9Q5...	s8qyPp~/@WCJ7$c8kS8#F!!J...	]yPXLlq@zyqq!WcxGeu+!p.mGFdpW...
14	df{2a#mx|Rgdvh[sxp:k#^Vh;t&Ma...	,ZUdn[dkAhbW#i[$~Vv&{=bqpY...	3~s,?b[vJVDIzV@rV@:ssl}}dGF...	lmh2[#bpR[XN.7@PPweSgl$|V...	;v?ciRmo|#=lxM:hmS[:s|aVkA7cS:iG=...
15	2/uEx/=NG{#LTil@bPH/0.x9Y3T...	}yqY^:$@90tYKugHFGfxo^fxj!^K...	/L+B{pHoFg3g[QTeE/{9#qn3E...	vri%8Q@Hob}cY[H&j}m3]bS$EX...	Rl(iexlcv[&i@F=0n/G~On/|jW}S[:Q~k...
16	DbZE}|StuoArpftHvpktV+dbG.f/;$...	tv63||lQuY!tJ4|qgJkes!QL#vLuo2...	./(lGq2OyrguhySrG+s#!+Sdte/4...	%~}3HHYsa]apfllbvmu]66J@mS...	Wzh$0Aty~LmHz~slaw]sQ~X1yn4#G...
17	zqe1[Qk]Ei|&czQUzJnl]k&mUplqh...	1{m,?344V=z&sv|fbC{ePezcJ]Nx...	~3[1p]0{c]^o$hq4}JlUJc[W:U0...	,ERs:K99zoq!zlczlu~Oi]q9vJ|{K[...	hk?chpW3$Cq%oqOt4P$l~P@fEWU...

Query executed successfully. | DESKTOP-O745MSP\SQLEXPRESS ... | sa (52) | master | 00:00:00 | 32 row

Figure 5: Data stored in the backend table

5. Analyses

In this section, we evaluate the strength of the proposed method. Firstly we define the attack models then we assess the theoretical password space and entropy of our proposal and compare the strength level against these attack models.

5.1 Attack models

There are many types of attackswhich may divided into two categories: the first category includes lookup and rainbow table attacks. These two attacks use precomputation lists. The second category includes the brute-force, dictionary and reverse lookup table attacks. In brute-force, adversaries try every possible combination of passwordswhile the other two may be disastrous to the people who re-use the same password across many applications.

In our proposal, we have designed a login-script with a custom character-set which shouldn't be available in rainbow or lookup tables, moreover, we didn't use the dictionary words. Thus the only leftover option is the brute-force attack which may be simulated in our proposal.

5.2 Password space and entropy of SecureNum's plain login-script:

The security of any authentication method is directly proportional to the password space. According to the information theory of Claude Shannon [34][35], the maximum number of passwords that may be generated from a given set of alphabets may be known as *password space*, and how many guesses are needed to find the actual password is called *entropy* [13]. The entropy of any password may be calculated in bits [3]. As prepared by [34], password space and entropy calculations of all possible predefined character-sets (as well as custom character-sets) are defined in an Excel sheet [34]. The formula for entropy calculation is as under.

Although Shannon's entropy calculation criteria were challenged [8] only with minor differences (1-2 bits). In another comprehensive research [9] from the same authors, it was concluded that Shannon's formula may be applied where the possibilities of password creation are not arbitrary in size.

$$H = -\sum_{i=1}^{n} p_i \cdot \log_{2(}p_{i)} \quad \text{——} \quad \textbf{(i)}$$

The entropy of any character-set may be calculated by using the equation (i), e.g., a standard PIN has a total of 10 digits, and English letters are 26, A to Z (capital) and a to z (small). For standard PIN, the chances of guesses for each digit is "1 / 10 = 0.10" and for English, letters are "1 / 26 = 0.0384". After putting these values to the equation (i) for standard PIN, the value of H = - [(0.10log$_2$ x 0.10) + (0.10log$_2$ x 0.10) + (0.10log$_2$ x 0.10) … (up to 10)] = ***3.3219*** and for English letters H = - [(0.0384log$_2$ x 0.0384) + (0.0384log$_2$ x 0.0384) + (0.0384log$_2$ x 0.0384) … (up to 26)] = ***4.7004.*** It means that the 3.3219 and 4.7004 average number of questions needed to be asked to fetch a randomly selected PIN digit or English letter, respectively.

According to the example detailed above, firstly, it is needed to generate the character-set of the SecureNum login-script. In**Figure 2**, upon new registration, two login-scripts are shown to the user (green) and the other for backend purposes (red).User viewable login-script is easily understandable for the users, while the backend one is in a plain backend format. The character-set of SecureNum may be as *"acfklo+-*/1234567890,"* (for backend login-script) which consists of 21 alphabets (a subset of ASCII characters set) having ***4.3932*** (single alphabet entropy in bits). For obtaining the entropy of a particular length of PIN or password, these single character entropies must be multiplied by the total length of PIN or password, e.g., a 10-length standard PIN and textual password should have entropies ***33.219*** and ***47.004*** (in bits), respectively. Likewise, SecureNum should have an entropy of ***43.932*** bits for 10 lengths (alphabets), but interestingly this statement is false for SecureNum because, there is a twist behind the 10-length password of SecureNum, which is way longer than the standard 10-length PIN and Password because, in the latter ones, each character is directly comparable with associated characters

A security checkpoint lies behind the backend login-script shown in**Figure 2**, which are 8 processes (separated by a comma). Therefore, in comparing ***33.219*** and ***47.004*** bits of PIN and password, the ***4.3932*** should be multiplied by 8 processes (*51 alphabets*) to obtain its actual entropy. When ***4.3932*** is multiplied by 51, the actual entropy of SecureNum becomes ***224.0532*** bits which is nearly equal to *32 character password* having a full character-set of *ASCII table* which is practically not feasible for a normal human memory (normal password lengths are 8.18 (young) and 4.45 (old) [36] according to age groups). Consequently, the password space with this entropy of SecureNum is exponentially much higher, i.e., *2.7113403136065**E67**.*

5.3 Discussion

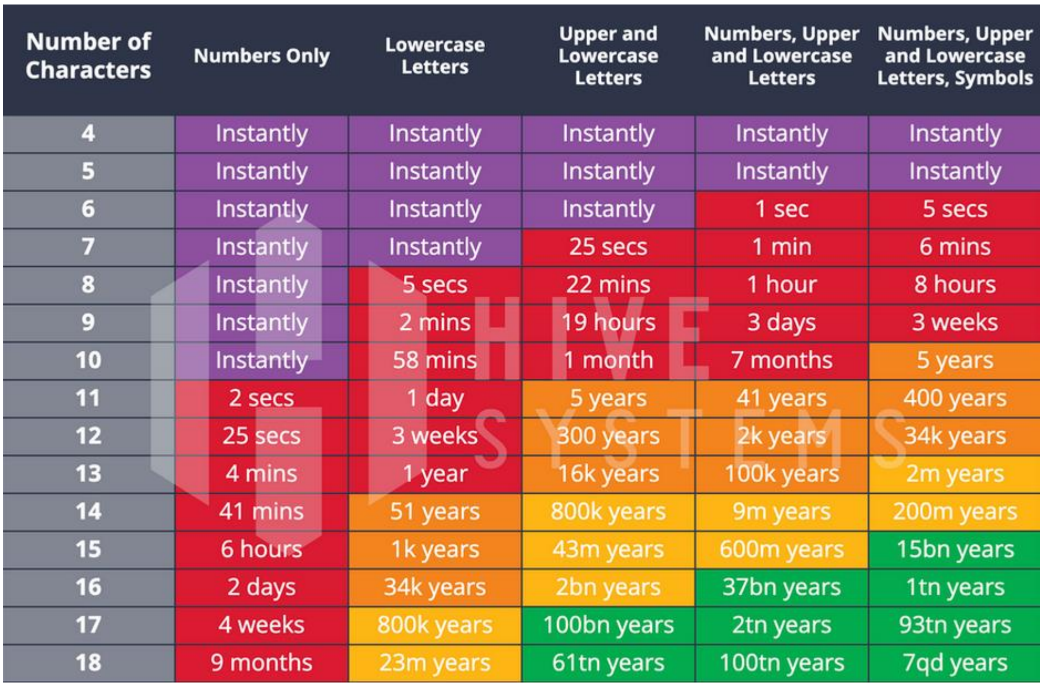
SecureNum works on the pattern of passphrases, whichare space-delimited sets of words chosen from natural language. These are used to balance security and usability and provide high entropy [3] when used with a large character-set. A large-scale study of 1476 online participants[37] was conducted to compare 3-4 character random passphrases with 5-6 character passwords (both were system assigned).

The results were quite astonishing because both passwords and passphrases were forgotten at similar rates. Login-script of SecureNum is motivated bypassphrases but with memory aids which does not impact memorability.

Let's first analyze the resilience of *plain login-script*(assumed it is also stored as straight (un-hashed) on the backend) of SecureNum against offline brute-force attacks. To develop our understating, there is a need to recall the discussion detailed in the password space and entropy section where we defined the entropy of SecureNum i.e. ***4.3932*** (single character) with a mathematical equation as well as the entropies of digit 0-9 and English letters are***3.3219*** and ***4.7004***respectively**.** Most interestingly the entropies of character-set of ASCII-95 is ***6.5699****.* These calculations are calculated in bits and are based on Shannon and NIST [34][35].Entropy calculations are based on (Shannon theory) character-set and we have also exposed the character-set of SecureNum which have 21 alphabets i.e. *"acfklo+-*/1234567890,"*.It is also explored that the length of the password string decides the entropy of the whole string.

According to hive systems [38] (**Figure 6**), a brute-force attack for 18 length password, chosen from the character-set of ASCII-95 (numbers, upper and lowercase letters and symbols) may take 26 trillion years. Most interestingly, the simpler and memorable 4-digit PIN with the meek 1,2,3,4 FNs (**Figure 2)** of SecureNum generates ***51*** length login-script with an entropy of ***224.0532*** (password having entropy of 70 bits of entropy may be cracked within 37 years with the speed of 1 trillion guesses per second[38]), equal to the ***34*** alphabet long from ASCII-95 i.e. ***6.5699*** bits. Still, the 34 length is nearly double for 18 length. Thus, on increasing only a single digit in SecureNum's PIN, the length may increase from 4 to 8 alphabets. i.e. 4 in case of "loc1" and 8 in case of "loc99+99". After this thorough analysis, it may be concluded that the *plain version* of login-script is highly secure for brute-force.

On the other hand, some researchers [9] conclude to modify the entropy calculation because the Shannon and NIST [34][35] propose the theoretical approaches, while there is a gap between the theoretical and practical distribution of password space. They introduced the idea because many traditional graphical password methods [39][40] mostly do not save the secret as straight as a textual password. While SecureNum generates the secret same as the textual password, so we may follow the benchmarked formulas [34][35]. Moreover, some known researchers[3] adopted the same entropy calculator.



Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	Instantly	Instantly	Instantly
6	Instantly	Instantly	Instantly	1 sec	5 secs
7	Instantly	Instantly	25 secs	1 min	6 mins
8	Instantly	5 secs	22 mins	1 hour	8 hours
9	Instantly	2 mins	19 hours	3 days	3 weeks
10	Instantly	58 mins	1 month	7 months	5 years
11	2 secs	1 day	5 years	41 years	400 years
12	25 secs	3 weeks	300 years	2k years	34k years
13	4 mins	1 year	16k years	100k years	2m years
14	41 mins	51 years	800k years	9m years	200m years
15	6 hours	1k years	43m years	600m years	15bn years
16	2 days	34k years	2bn years	37bn years	1tn years
17	4 weeks	800k years	100bn years	2tn years	93tn years
18	9 months	23m years	61tn years	100tn years	7qd years

Figure 6: Password table for 2021[38]

Now we come to the *hashed/encoded version* of SecureNum, as presented in**Figure 3** the metadata is hashed with SHA3-224 with salt. Moreover, we proposed a refined custom-encoded design for user data. The result is shown in**Figure 5**. Let's assume the attacker obtained the complete database file(**Figure 5**). Now, from where s/he will start there will be strong confusion to understand the dataset. Let's assume, the attacker knows our custom mapping like s/he knows the rules of *fragmentation*, *concatenation* and *character swapping*.S/he also knows the 5 fragments stored on 5 random columns of the data table. However, the actual problem for the attacker is randomness at each stage. We see this process step by step.

1. Identification of 5 fragments of login-scriptout of 13 columns/fields. $nPr = 13P5 = 154440$ (number of permutations required). If succeed, go to step 2.
2. Finding out the correct sort order of 5 fragments. Factorial of 5! = 120 possible permutations. If succeed, go to step 3.
3. Join the 5 fragments into a single string and obtain the actual characters of login-script randomly distributed in 150 x 5 = 750 characters (each fragment hasa size of 150 characters). If succeed, go to step 4.
4. Generate the 2 lists for de-swapping i.e. list 1, list 2, and both lists should be 88 characters each. The first list is straightforward but the second list is randomly distributed. If succeed, go to step 5.
5. Find the comma-separated chunks and apply ICS in reverse format using the lists found in the last step.

Before elevating the attack scenario, the attacker must know that on the login-script only custom encoding is applied, Hashing is only done on metadata.Now, we evaluate the success rate of the attacker. Each step needs the correct input of the last step which practically makesSecureNum's login-script unbreakable. The first step hasa maximum number of 120 permutations but in the second step, finding the correct length and correct characters of login-script is tough because login-script may be of any length. As described earlier, 1 digit of SecureNum's PIN may have 4 to 8 long alphabets at the backend. After finding the length, the actual trouble is to find the correct characters of that length from the 750-character list. In step 3, the difficulty is increased indefinitely, when the attacker needs to find the factorial of 88!, i.e. 1.854826422574E+134.

John-the-Ripper is an expert in cracking the hashes [24] by complying with the knowledge of bad habits of users (password reuse) but here even hashcat [3] may not help because these need custom configuration and sample files of hashed/un-hashed passwords. Researchers of [3] further added that improper configuration of brute-force applications i.e. hashcat may result inthe opposite of the expectation.The format of login-script is custom-defined which may not be found even in the up-to-date dataset[7].

5.4 Proposed brute-force algorithms

To check the strength of our proposal against brute-force attacks, we have proposed 3 algorithms to crack the defence mechanism of SecureNum. In the previous section, we have detailed the working mechanism of SecureNum and the final output is shown in **Figure 5**. The very first step, an adversary needs to take is to find out the 5 columns (among 13) from the backend table. Algorithm 1 is presented for the same purpose which is a recursive heap algorithm. Upon each iteration of finding out the 5 chucks, it calls the algorithm 2.

Algorithm 1: rec_Heap_Algo (find 5 chunks among 13 objects)

```
Input  : Elements for permutation
Output : Permuted items
Initialization gen_Permutations;
elements_2_Permute;
LenOfArr;
IfLenOfArr = = 1  then
else
rec_Haep_Algo ( elements_2_Permute, LenOfArr -1)
for ( i=0 to LenOfArr) do
if (LenOfArr is even) then
SwapElements(i, LenOfArr – 1);
    //////////////Call Algorithm 2/////////////////////
rec_Factorial_Algo
/////////////////////////////////////////////////////////////
else
SwapElements(0, LenOfArr – 1);
    //////////////Call Algorithm 2/////////////////////
rec_Factorial_Algo
/////////////////////////////////////////////////////////////
end
rec_Heap_Algo(gen_Permutations, elements_2_Permute, LenOfArr);
end
end
```

Algorithm 2 (recursive factorial algorithm) also works in a recursive pattern which generates the permutations of 5 chunks and after each result, it calls algorithm 3.

Algorithm 2: rec_Factorial_Algo (find the required order of 5 chunks)

```
Input  : Elements for factorial
Output : permuted items
Initialization gen_Factorial;
Ifgen_Factorial = = 0 or 1 then
~~~~~~~~finish~~~~~~~~
else
Return gen_Factorial * rec_Factorial_Algo (gen_Factorial – 1);
  //////////////Call Algorithm 3/////////////////////
rec_Heap_Algo_for_88
/////////////////////////////////////////////////////////////
end
```

Algorithm 3 (recursive heap algorithm for 88 characters) is also recursive because it also obtains the permutations of 88 alphanumeric. For each iteration (88 permutations), it runs the *Incremental Character Swapping* (ICS) algorithm which is responsible for finding out the original login-script after swapping the merged data of 5 chunks according to the *original* and *shuffled chars* (algorithm 3).

Algorithm 3: rec_Heap_Algo_for_88 (find each permutation for 88 and run ICS on each)

```
Input  : Elements for permutation and an empty string for the final result
Output : Permuted items and swapped characters
Initialization gen_Permutations;
               elements_2_Permute;
LenOfArr;
salt;
original_chars;
shuffle_chars;
stringtorotate;
rotatedPass;

IfLenOfArr = = 1  then
else
rec_Haep_Algo ( elements_2_Permute, LenOfArr -1)
for ( i=0 to LenOfArr) do
if (LenOfArr is even) then
SwapElements(i, LenOfArr – 1);
/////////////////////////////Run ICS on each permutation of 88 chars/////////////////////////////
salt←unique_rev_salt(salt);
original_chars← ASCII (88 characters);
```

```
shuffle_chars←ASCII (88 characters);
for ( i=0 to length.salt) do
    if (indexof(shuffle_chars)>= 0 ) then
        shuffle_chars←indexof(remove(shuffle_chars);
    end
end
shuffle_chars = salt + shuffle_chars
for ( i=0 to length.stringtorotate) do
    if ( stringtorotate>= 0 ) then
        rotatedPass←rotatedPass+ shuffle_char;
    end
end
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
else
SwapElements(0, LenOfArr – 1);
//////////////////////////////Run ICS on each permutation of 88 chars///////////////////////////////
salt←unique_rev_salt(salt);
original_chars← ASCII (88 characters);
shuffle_chars←ASCII (88 characters);
for ( i=0 to length.salt) do
    if (indexof(shuffle_chars)>= 0 ) then
        shuffle_chars←indexof(remove(shuffle_chars);
    end
end
shuffle_chars = salt + shuffle_chars
for ( i=0 to length.stringtorotate) do
    if ( stringtorotate>= 0 ) then
        rotatedPass←rotatedPass+ shuffle_char;
    end
end
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
end
rec_Heap_Algo(gen_Permutations, elements_2_Permute, LenOfArr);
end
end
```

6

2
7

2
8

2
9

3
0

3
1

3
2

3
3

3
4

3
5

3
6

3
7

3
8

3
9

4
0

4
1

4
2

4
3

4
4

4
5

4
6

4
7

4
8

4
9

5
0

5
1

5
2

5
3

5.4.1 Time Complexity Analysis

The time complexity of all three algorithms is *O(n!)*because of the permutations but there are some important points that algorithm 1 calls algorithm 2 within each permutation, similarly algorithm 2 also calls algorithm 3 within each permutation. Algorithm 3 also generates the permutations but in each iteration, it performs the swapping operation of one string with another string (based onthe current permutation). Thus, it may be concluded that the time complexity of all three algorithms is exponentially very high i.e. *O(n!)*^*O(n!)*^*O(n!)* which is surely beyond the limit of current hardware and require massive amount of time.

5.5 Comparisons

We have also compared our proposal with state-of-the-art in terms of time complexity regarding the brute-force attack (table 1). Most of the proposals have the same rating complexity as O(*n*). Nearly all proposals are using hash. "S/KEY"[41]has adopted a purely hash-based mechanism where the schemes of "Insure Networks [42]", "Smart Card based [43]" and "OTP for public key [44]" are based on the hard problems i.e. discrete logarithm problem. The studies of [32] and [31] are based on NDBs. ENPI and ENPII [31]are the only schemes whose complexity for the attack is O(*n!*) rather all the other ones have O(*n*). In comparison, the authentication algorithm of SecureNum also has the time complexity as O(*n*) but the proposed brute-force attack (having 3 algorithms) is as much exponentially complex in terms of time i.e. O(*n!*)^O(*n!*)^O(*n!*) and space (above the limit of certain memory limits) i.e. O(*n*)^O(*n*)^O(*n*) as well as the beyond the limit of current hardware (machine cost [16][17]) .

Table 1: Comparisons for the complexity of attack

Schemes	Security	Time Complexity
S/KEY [41][45]	*Hash*	O(*n*)
Insecure Networks [42]	*Discrete logarithm problem + Hash*	O(*n*)
Smart Card based [43]	*Discrete logarithm problem + Hash*	O(*n*)
OTP for public key [44]	*Discrete logarithm problem + Hash*	O(*n*)
OTP for NDB [32]	*NDB + Hash*	O(*n*)
ENPI + ENPII [31]	*NDB + Hash*	O(*n!*)
SecureNum (Our proposal)	*Custom Encoding/Decoding + Hash*	O(*n*)
Proposed brute-force attack over our proposed scheme (SecureNum)		O(*n!*)^O(*n!*)^O(*n!*)
Where 'n' denotes the number of elements		

5.6 Resistance against other attacks

SecureNum delivers security in a double layer. One layer is detailed in the last section, the other layer is the adaptation of its custom login-script. The verification process needs a dynamic PIN which should match the login-script. The dynamic PIN is the rectification for*man-in-the-middle* as well as *replay attacks.* Furthermore, attackers didn't verify the PIN whether itwas correct or not because there was no exact match available in the backend. Thus the *guessing and exhaustive attacks* go in vain.

6. Declarations

A. Ethical Approval

It was not required because, in this study, the identity of the human participants is not shown.

B. Competing interests

The authors declare that they have no conflict of interest regarding the study.

C. Authors' contributions

All authors equally contributed to this work.

D. Funding

No funding was received for this study.

E. Availability of data and materials

It will be available upon request.

7. Conclusion

In this paper, we proposed a custom encoding/decoding scheme, a chunked PIN method resistant to *brute-force, dictionary, lookup, and rainbow* attacks. Our analysis shows our proposal provides exponentially higher protection with industry-standard methods (5! First step, 750 characters 2nd step, 88! 3rd step). Even the modern tools, hashcat and john- the-ripper may not help due to the custom method of protection.We also compared our proposed algorithm with state-of-the-art and the results were quite astonishing because SecureNum has comparable time complexity as $O(n)$ but the proposed brute-force attack (3 algorithms) produced exponentially very high attack complexity i.e. $O(n!)$^$O(n!)$^$O(n!)$ in terms of time and cost.We further explained how our proposed custom login-script is capable of resisting*man-in-the-middle, replay, guessing* and *exhaustive attacks*. For future work, we have planned to extend our scheme by applying the SHA3-512 and comparing the performance of the existing with new implemented setup.

References

[1] D. Florencio and C. Herley, "A large-scale study of web password habits," *16th Int. World Wide Web Conf. WWW2007*, pp. 657–666, 2007.

[2] N. Woods and M. Siponen, "Improving password memorability, while not inconveniencing the user," *Int. J. Hum. Comput. Stud.*, vol. 128, no. February, pp. 61–71, 2019.

[3] L. Bošnjak, L. Bošnjak, J. Sreš, and B. Brumen, "Brute-force and dictionary attack on hashed real-world passwords," *ieeexplore.ieee.org*, 2018.

[4] K. D.-I. J. I. E. T. Brute-force and undefined 2013, "Brute-force attack 'seeking but distressing,'" *Citeseer*.

[5] T. Gautam and A. Jain, "Analysis of brute force attack using TG-Dataset," in *IntelliSys 2015 - Proceedings of 2015 SAI Intelligent Systems Conference*, 2015, pp. 984–988.

[6] M. M. Najafabadi, T. M. Khoshgoftaar, C. Calvert, and C. Kemp, "Detection of SSH brute force attacks using aggregated netflow data," *Proc. - 2015 IEEE 14th Int. Conf. Mach. Learn. Appl. ICMLA 2015*, pp. 283–288, Mar. 2016.

[7] "Free Rainbow Tables." [Online]. Available: https://freerainbowtables.com/. [Accessed: 21-Dec-2021].

[8] J. Bonneau, "Statistical metrics for individual password strength (Transcript of discussion)," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7622 LNCS, pp. 87–95, 2012.

[9] J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," *Proc. - IEEE Symp. Secur. Priv.*, pp. 538–552, 2012.

[10] P. G. Kelley *et al.*, "Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms," *Proc. - IEEE Symp. Secur. Priv.*, pp. 523–537, 2012.

[11] M. L. Mazurek *et al.*, "Measuring password guessability for an entire university," *Proc. ACM Conf. Comput. Commun. Secur.*, pp. 173–186, 2013.

[12] M. M. Taha, T. A. Alhaj, A. E. Moktar, A. H. Salim, and S. M. Abdullah, "On password strength measurements: Password entropy and password quality," *Proc. - 2013 Int. Conf. Comput. Electr. Electron. Eng.*

'Research Makes a Differ. ICCEEE 2013, pp. 497–501, 2013.

[13] C. Castelluccia, M. Dürmuth, and D. Perito, "Adaptive password-strength meters from markov models.," *ndss-symposium.org*.

[14] "hashcat - advanced password recovery." [Online]. Available: https://hashcat.net/hashcat/. [Accessed: 12-Mar-2024].

[15] "John the Ripper password cracker." [Online]. Available: https://www.openwall.com/john/. [Accessed: 12-Mar-2024].

[16] A. D. Vu, J. Il Han, H. A. Nguyen, Y. M. Kim, and E. J. Im, "A homogeneous parallel brute force cracking algorithm on the GPU," *2011 Int. Conf. ICT Converg. ICTC 2011*, pp. 561–564, 2011.

[17] Laatansa, R. Saputra, and B. Noranita, "Analysis of GPGPU-Based Brute-Force and Dictionary Attack on SHA-1 Password Hash," *ICICOS 2019 - 3rd Int. Conf. Informatics Comput. Sci. Accel. Informatics Comput. Res. Smarter Soc. Era Ind. 4.0, Proc.*, Oct. 2019.

[18] R. Morris and K. Thompson, "Password Security: A Case History," *Commun.* ACM, vol. 22, no. 11, pp. 594–597, Nov. 1979.

[19] D. C. Feldmeier and P. R. Karn, "UNIX password security - Ten years later," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 435 LNCS, pp. 44–63, 1990.

[20] M. A. D. Brogada, A. M. Sison, and R. P. Medina, "Head and tail technique for hashing passwords," *2019 IEEE 11th Int. Conf. Commun. Softw. Networks, ICCSN 2019*, pp. 805–810, Jun. 2019.

[21] H. Qiu, G. Memmi, and H. Noura, "An Efficient Secure Storage Scheme Based on Information Fragmentation," *Proc. - 4th IEEE Int. Conf. Cyber Secur. Cloud Comput. CSCloud 2017 3rd IEEE Int. Conf. Scalable Smart Cloud, SSC 2017*, pp. 108–113, Jul. 2017.

[22] K. Malvoni, ... J. K.-W. on O. T. (WOOT 14, and undefined 2014, "Are Your Passwords Safe:{Energy-Efficient} Bcrypt Cracking with {Low-Cost} Parallel Hardware," *usenix.org*.

[23] N. R. Sai, T. Cherukuri, B. Susmita, R. Keerthana, and Y. Anjali, "Encrypted Negative Password Identification Exploitation RSA Rule," *Proc. 6th Int. Conf. Inven. Comput. Technol. ICICT 2021*, Jan. 2021.

[24] A. Juels and T. Ristenpart, "Honey encryption: Encryption beyond the brute-force barrier," *IEEE Secur. Priv.*, vol. 12, no. 4, pp. 59–62, 2014.

[25] E. Bauman, Y. Lu, and Z. Lin, "Half a century of practice: Who is still storing plaintext passwords?," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9065, pp. 253–267, 2015.

[26] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why Do Developers get password storage wrong? a qualitative usability study," *Proc. ACM Conf. Comput. Commun. Secur.*, pp. 311–328, Oct. 2017.

[27] M. C. Ah Kioon, Z. S. Wang, and S. Deb Das, "Security Analysis of MD5 Algorithm in Password Storage," *Appl. Mech. Mater.*, vol. 347–350, pp. 2706–2711, 2013.

[28] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 2729, pp. 617–630, 2003.

[29] J. Hallett, N. Patnaik, B. Shreeve, and A. Rashid, "'Do this! Do that!, and nothing will happen' Do specifications lead to securely stored passwords?," *Proc. - Int. Conf. Softw. Eng.*, pp. 486–498, May 2021.

[30] A. Visconti, S. Bossi, H. Ragab, and A. Calò, "On the weaknesses of PBKDF2," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9476, pp. 119–126, 2015.

[31] W. Luo, Y. Hu, H. Jiang, and J. Wang, "Authentication by encrypted negative password," *IEEE Trans. Inf. Forensics Secur.*, vol. 14, no. 1, pp. 114–128, Jan. 2019.

[32] D. Zhao and W. Luo, "One-time password authentication scheme based on the negative database," *Eng. Appl. Artif. Intell.*, vol. 62, pp. 396–404, Jun. 2017.

[33] D. Zhao, W. Luo, R. Liu, and L. Yue, "A fine-grained algorithm for generating hard-toreverse negative databases," *Int. Work. Artif. Immune Syst. AIS 2015/ICSI3 2015 - Syst. Immunol. Immunoinformatics Immune-computation Immunol. without Borders, Proc.*, May 2016.

[34] "A Somewhat Brief Explanation of Password Entropy | IT Dojo." [Online]. Available: https://www.itdojo.com/a-somewhat-brief-explanation-of-password-entropy/. [Accessed: 12-Dec-2021].

[35] W. Burr, W. Polk, D. D.-N. S. Publication, and U. 2004, "Electronic Authentication," *NIST Spec. Publ.*, 2004.

[36] D. R. Pilar, A. Jaeger, C. F. A. Gomes, and L. M. Stein, "Passwords Usage and Human Memory Limitations: A Survey across Age and Educational Background," *PLoS One*, vol. 7, no. 12, p. 51067, Dec. 2012.

[37] R. Shay *et al.*, "Correct horse battery staple: Exploring the usability of system-assigned passphrases," *SOUPS 2012 - Proc. 8th Symp. Usable Priv. Secur.*, 2012.

[38] "How an 8-character password could be cracked in just a few minutes." [Online]. Available: https://www.techrepublic.com/article/how-an-8-character-password-could-be-cracked-in-less-than-an-hour/. [Accessed: 26-Dec-2021].

[39] P. Dunphy and J. Yan, "Do background images improve 'draw a secret' graphical passwords?," *Proc. ACM Conf. Comput. Commun. Secur.*, pp. 36–47, 2007.

[40] S. Wiedenbeck, J. Waters, J. C. Birget, A. Brodskiy, and N. Memon, "PassPoints: Design and longitudinal evaluation of a graphical password system," *Int. J. Hum. Comput. Stud.*, vol. 63, no. 1–2, pp. 102–127, Jul. 2005.

[41] N. Haller, "The S/KEY One-Time Password System," Feb. 1995.

[42] I. E. Liao, C. C. Lee, and M. S. Hwang, "A password authentication scheme over insecure networks," *J. Comput. Syst. Sci.*, vol. 72, no. 4, pp. 727–740, Jun. 2006.

[43] J. Xu, W. T. Zhu, and D. G. Feng, "An improved smart card based password authentication scheme with provable security," *Comput. Stand. Interfaces*, vol. 31, no. 4, pp. 723–728, Jun. 2009.

[44] H. C. Kim, H. W. Lee, K. S. Lee, and M. S. Jun, "A design of one-time password mechanism using public key infrastructure," *Proc. - 4th Int. Conf. Networked Comput. Adv. Inf. Manag. NCM 2008*, vol. 1, pp. 18–24, 2008.

[45] M. SANDIRIGAMA, A. SHIMIZU, and M.-T. NODA, "Simple and Secure Password Authentication Protocol (SAS)," *IEICE Trans. Commun.*, vol. E83-B, no. 6, pp. 1363–1365, Jun. 2000.

Authors Biography

Awais Ahmad is a Ph.D candidate in the department of Computer Science of National Textile University, Faisalabad. He earned **Gold Medal** in his MSCS degree program from the same university. His research interests are usable security in the authentication systems and blockchain technology. One of his published research regarding blockchain got 90+ citations publish in ScienceDirect.

Muhammad Asif is currently working as Director, Graduate Studies and Research and Tenured Associate Professor of Computer Science. He also served as Chairman of the Department of Computer Science at National Textile University, Faisalabad, till November 04, 2020. Before this, he was a research scholar in the Computer Science and Information Management Department at the Asian Institute of Technology, Thailand. He received his MS and Ph.D. from AIT in 2009 and 2012 on HEC foreign Scholarship. He enjoys more than 200 impact factors from his research publications in top-class computer science journals and allied domains.

Isma Hamid has a doctoral degree specialized in behavior analysis and visualization technology of social networks and her research interests are in the areas of behavior analysis, visualization, Image processing, Pattern recognition and Big Data Analysis . She has published a number of research papers in different EI and SCI indexed international journals and Conferences. She has nine years of teaching, research, and application development experience in reputed public and private sector universities of Pakistan.