Journal of Theoretical and Applied Information Technology
28th February 2018. Vol.96. No 4
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

PREVENTING SECURITY ATTACKS ON MOBILE PATTERN PASSWORDS

[1]Bh PADMA, [2]GVS RAJKUMAR

[1]Gayatri Vidya Parishad, Rushikonda Beach Road, Visakhapatnam-45, INDIA

[2]GITAM, Rushikonda Beach Road, Visakhapatnam-45, INDIA

E-mail: [1]padma.bhogaraju@gmail.com, [2]gvsrajkumar@gmail.com

ABSTRACT

An Android Smartphone is a personal device, which keeps many of our personal files and data, such as photos, videos, messages, bank account information etc. Keeping these files safe from outsiders may be troublesome especially when they try to speculate the device passwords. So Android authentication processes should always pursue robust security enhancements to preserve the security of the sensitive data stored in the mobiles. For pattern locking systems of Android, older versions such as Kit Kat and Lollipop make use of authentication systems which rely on SHA-1 and MD5 unsalted hashes, but the latest versions such as Android Marshmallow employ Gatekeeper Mechanism and store the passwords and authenticate the users in a trusted execution environment and are more secured from brute-forcing. The former methods are vulnerable to dictionary and rainbow table attacks since they are unsalted hashes, whereas the later Android hashing schemes such as HMAC or Scrypt hashes use salts for hashing but cannot flee from hacker's forensic tools that crack the passwords and they do need an additional hardware support. Therefore this paper presents two substitute methodologies that suggest a new approach to enhance the basic SHA-1 hashing scheme using Elliptic Curves to prevent pre-computation attacks such as dictionaries, rainbow tables and brute forcing on pattern password scheme. These proposed methods seem to be simple and secure without employing a complex hardware-backed environment such as Trusted Execution Environment (TEE). This paper also presents a comparison among the proposed schemes with respect to Strict Avalanche Effect and CPU Execution Times after the implementation.

Keywords: *Android, Smartphone, SHA-1, Brute- Force, Dictionaries, TEE.*

1. INTRODUCTION

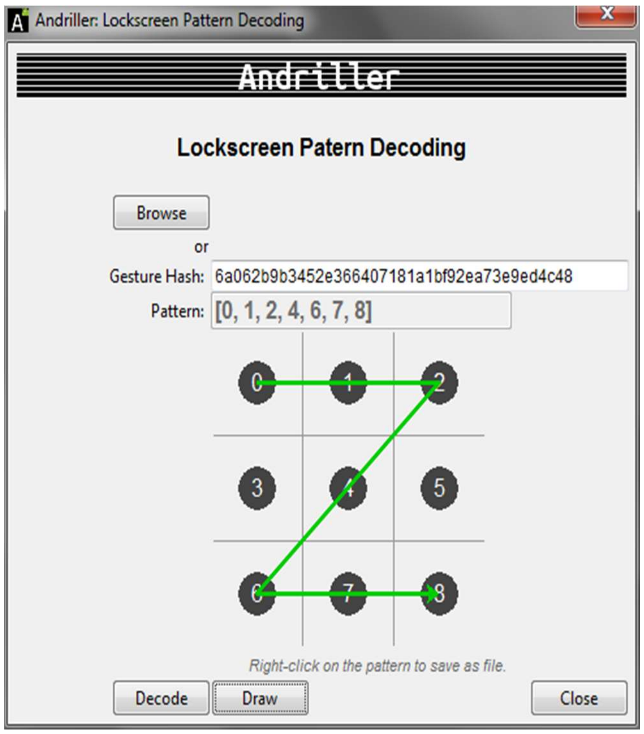
Mobile devices are more powerful than they were in the past, and these devices have a relatively huge storage capacity. Furthermore, mobile devices have gone from being vague novelties to becoming mainstream technologies. Providing security to mobile phones using robust cryptographic authentication techniques is very important now a days, because they protect confidential data. Especially for pattern unlock systems of Android, there is a lack of awareness in the people about various security breaches.

Android offers many types of password lock protection schemes. Among them pattern locking system is a graphical password authentication system[17], which is a mix of lines drawn by the phone owner linking the points on a matrix for unlocking his phone. But these systems seem to be insecure enough to be cracked using Smudge Attacks[16] and particularly suffer from pre-computation attacks in the lower versions of Android such as Kit Kat because they generate SHA-1 unsalted hashes for user authentication[2]. The pattern selected by the user must have at least four points where each point only can be used once. Statistically, it is very simple to experiment all the combinations between 0123 and 876543210 using either dictionaries or rainbow tables and many mobile unlocking methods are also available in the web. But here the focus is on gaining the password file attempting passive attacks. When your mobile is rooted and USB enabled, people can capture the gesture.key[8] file in which the pattern password i.e. SHA-1 hash of the pattern, is stored in "/data/system/" folder. If you have full admission to a mobile, you can just take away or replace the file that contains the SHA-1[13] pattern password. Figure 1 and Figure 2 shows how an attacker uses forensic tools[7] such as Andriller to achieve the password using a dictionary that can be downloaded and using SQLite browser. He easily finds the original pattern by running the query,

"Select * from Rainbow Table where hash = "**2c3422d33fb9dd9cde87657408e48f4e635713cb**".

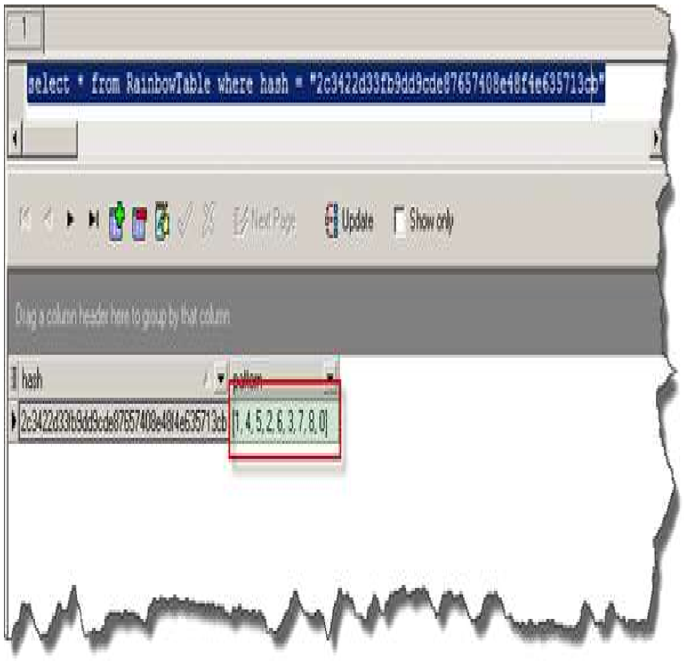
In this situation, the password storage file could be accessed, and the pattern password could be cracked. However, the attacker just needs to remove the key storage file directly, and the unlock pattern will fail. But if the attacker attacks passively, he silently gets the password using dictionary attacks through which he gains the access to the device a number of times. After Kit Kat 4.4 Android brought some changes in the authentication systems, which include salts for hashing particularly in Lollipop and android Marshmallow[9] versions. But salts will not solve the problem completely because we store them in database, if compromised the attacker may brute-force the password using the salt value. Once the salt is compromised, the attacker finds some way to gain the password using pre-computations, because having 100% security[21] for any security system is not achievable.

In this paper we concentrate on how to get rid of pre-computation on patterns. We need to have a solution to modify pattern authentication systems so that they can withstand to dictionaries. The proposed methodologies change the representation of the pattern entirely different and the pattern representation changes from user to user depending on his identities. We have included the application of elliptic curves in Android pattern representation, as the small key sizes of Elliptic Curve Crytography make it very smart for devices with restricted storage and processing power. So ECC is supposed to be the best candidate for mobile devices. After the pattern is selected by the user, it is transformed into another form which is unpredictable using dictionaries and rainbow tables. Also we enhance this idea to generate a dynamic salt value for the hash[11] to be stored in the databases. As the salt need not be stored along with the hash, it makes the pattern password difficult to guess using brute-forcing and obviously salts always prevent dictionary attacks.


Andriller: Lockscreen Pattern Decoding
Andriller
Lockscreen Patern Decoding
Browse
or
0
1
2
3
4
5
6
7
8
Right-click on the pattern to save as file.
Decode
Draw
Close

Gesture Hash: 6a062b9b3452e366407181a1bf92ea73e9ed4c48
Pattern: [0, 1, 2, 4, 6, 7, 8]

Figure 1 : Andriller tool



```
select * from RainbowTable where hash = "2c3422d33fb9dd9cde87657408e48f4e635713cb"
```

hash	pattern
2c3422d33fb9dd9cde87657408e48f4e635713cb	[1, 4, 5, 2, 6, 3, 7, 8, 0]

Figure 2: Using SQLite Browser

2. SALTS

The best way to safeguard passwords is to make use of salted password hashing[23]. There are lot of conflicting ideas and false impressions on how to do password hashing perfectly. Ahead of hashing, we can randomize the hashes either by appending or prepending a string called a salt. This causes the similar password hash into an entirely dissimilar string each time. Salting" is a security practice of attaching random data (a "salt") to a password sooner than hashing it and storing the hashed value.

The hashes are deterministic and present a problem with unsalted passwords. In the simplest case, if two people select the same password, then their hashed passwords are also same. And more importantly, if we are trying to crack a number of unsalted passwords, any result could strike any of the passwords. By salting, a successful attack can be practical to only one password at a time, and it is not easy to tell whether two passwords (with different salts) are same.

2.1 Salts and Dictionaries

In fact, there's a general strategy based on this plan is called a dictionary. A dictionary is a list of common passwords. If we can somehow get the passwords file, then with a dictionary, you can find users using common passwords. Salting passwords resolves this problem of dictionary attacks. In a salting scheme, we don't find the hash of a user's password by combining it with some additional random data, and later hashing that combination or concatenated strings. The additional information is known as the salt. We can use lots of methods to generate salts. There's a complex set of dealings in the correct salting strategy, which are beyond the scope of this research. We can choose a fixed salt string. This salt is added to each password, or we can make salts unique to each password. These salts are added along with user password hashed in the system.

If each user has a different salt that means that any attempt to breach the system needs to look at each user separately. If a salt is added, the hackers can't compute the hash code for a given common password once, and if each user has a different salt, then even if you've got terrible passwords, a thief needs to do a lot of labor to break the system. The attackers have to recompute the dictionaries and rainbow tables for each possible salt value or user.

Salts effectively increase the amount of effort needed to crack the passwords. If we add 12 bits of salt, then a rainbow table requires 4096 times more entries to discover common passwords. If the salt is long enough, then it will not be possible to create a rainbow table at all. If they try to attack you without a rainbow table, a 12 bit salt means that the attacker needs to attack the passwords of each of the user's separately. Even if they know the value of the salt, you've made it much harder for them to violate your security.

A public salt will not make dictionary attacks harder when cracking a single password. The attacker has admission into both the hashed password and the salt value, so when operating the dictionary attack, the attacker can simply employ the known salt when attempting to crack the password. So a static salt stored in a device/system database is always having a security breach to be hacked at any time.

2.2 Brute forcing with Salts in Android Authentication

The salt is stored in plaintext in the server or device database. To validate the user whenever he logs in, we need to ensure the password is correct, and we need the salt value, which usually is stored in the systems directory beside with the password or it is a part of the hash string. As the attacker won't be able to find out the salt value in advance, they can't pre-compute a lookup table or rainbow table. If each user's password is hashed by means of a different salt, each time he logs in, the reverse lookup table attack will not work either.

For example, SHA-1 hash of

(1234+QXLUF1bgIAde)

= 73a4b911081c1689f037e98b65ed6d95ba53f25.

But salted hashes are susceptible to brute-force attacks if they are not protected well. Even the most current versions of Android employ salts to generate the hashes such as Marshmallow, they are always liable to security vulnerabilities because hackers always try to attack the password files and salt values using certain tools to access the root directory and hack the salts.

3. MOTIVATION OF THE RESEARCH

Android latest versions such as Marshmallow, utilize Gatekeeper mechanism and needs a special hardware support such as TEE (Trusted Execution Environment). But there is a lack of research, in exploring how to make pattern authentication systems, stronger against pre-computations without using a special hardware support. So our motivation in this research is to invent a simple, enhanced design of the existing SHA-1 systems for pattern passwords against dictionaries and brute-forcing.

4. ELLIPTIC CURVES

An elliptic curve is a curve and is also a group. Elliptic curves[6] come into view in many areas of mathematics, starting from number theory to complex analysis, and from cryptography to mathematical physics. An Elliptic Curve equation is of the structure E: $y^2=x^3 + ax + c \pmod p$. The sum of two points P=(x1, y1) and Q=(x2, y2) is (x3, y3) where $x3 = \lambda^2 – x1 – x2$, and
$y3 = \lambda (x1 – x3) – y1$, with $\lambda = (y2 – y1)/(x2 – x1)$ if P!=Q then $x1 = \lambda^2 – 2x$, $y1 = \lambda (x – x1) – y$, and $\lambda = (3x^2 + a)/2y$ if P = Q. Elliptic curve cryptosystems are more extensive in everyday-life applications. The core operation of elliptic curve cryptosystems is the scalar multiplication[4] which multiplies some point on an elliptic curve by some (usually secret) scalar i.e. if P is a point on the curve, nP=P+P+…+n times. For any two points P and Q, it is computationally difficult to find an integer n such that P=nQ. This is called as **discrete logarithm problem**[10] which is difficult to solve and makes elliptic curve cryptosystems stronger to withstand the attacks.

Elliptic Curve[5] Cryptographic algorithms have the advantage of offering an equal level of security of RSA, using smaller key sizes. But while implementing elliptic curve protocols, people face difficulty of mapping a message on to the elliptic curve. Koblitz[1] proposed a technique to convert a message to a point on the curve. According to Koblitz, to map a message point m on to the curve, choose a parameter, for example k. For each number mk, obtain x=mk + 1 and attempt to find a solution for y in the elliptic curve equation. If we are unable to find such y that satisfies the curve equation for given x value, try x = mk +2 and then x = mk +3 till you can solve for y. Generally, we will find such a y before we get x = mk + k - 1. Then take the point (x, y) to represent m. For reverse mapping, the point (x, y) is set to m where m is the greatest integer less than (x-1)/k.

5. PROPOSED METHODOLOGIES

Hackers can capture the gesture.key file when people leave their mobiles rooted and USB enabled. If someone gains access to your mobile device, they can just remove or replace the file containing the SHA-1 hash of the pattern password. Android brought some modifications in the authentication systems, which include salts for hashing i.e. in Lollipop and Marshmallow versions. But even though salts are added to the passwords, the hackers may still gain the passwords using brute-force attacks, as salts are stored in the device databases and they can be compromised. There was no research regarding how to use salts efficiently so that they cannot be brute-forced. In this paper we are trying to invent a simple and novel method where it safeguards passwords those are being authenticated using salts.

One of the proposed systems generates a salt which is dynamically created depending on the user's password pattern, and unique identities such as Gmail-Id and Device-Id. This system do not store the salt in the system's directory securely and thus eliminates the pre-computation attacks such as dictionaries and rainbow tables and brute forcing. The way the salt value is used will be distinctive to each password. As a result, a more protected password storing technique can be accomplished.

The earlier versions of Android such as Kit Kat password systems are not using salted hashes. Salted hash has an advantage that even though the hash is cracked you cannot get the password. Android pattern locks are not salted hashes. Android Kit Kat pattern locks and Android Lollipop earlier versions are not using salted hashes. But the latest versions of Android like Marshmallow authentication systems used salted hashes generated by Scrypt algorithm and are somehow strong against dictionaries and rainbow tables but still stored salts are susceptible to brute-force. Salts will not solve the problem completely because we store them in database, if compromised the attacker may brute force the password using the salt value. Once the salt is compromised, because having 100% security for any security system is not achievable, the attacker finds some way to gain the salt value and he can still try the brute force attack, to gain the password. A salted hash has an advantage that even though the hash is cracked you cannot get the password.

This paper presents a novel idea to prevent pre-computations on patterns, based on the device identity and user identity that are used to represent the pattern using elliptic curve points, so that we can transform the input of the pattern to an alternative message from which a different hash [22] can be generated, to make it impossible to predict the pattern hash using dictionaries and rainbow tables. This method can be enhanced to generate a salt dynamically which need not be stored in the database so that system becomes resistant to brute-force and also dictionaries. Let us consider the elliptic curve i.e. $y^2 = x^3 + 9x + 17$ over F_{23}, the figure 3 shows how a pattern can be represented using the points on the curve.

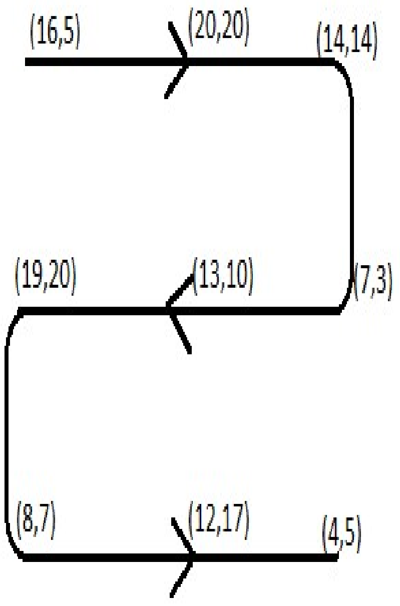


Figure 3: pattern representation using elliptic curve

Table 1: Elliptic Curve Representation of characters

Algorithm for Pattern Representation Using Elliptic curves:

Step 1*: Choose elliptic curve parameters a,b,p. (The application may select this triplet from a list of legal elliptic curve parameters and these values are stored somewhere in the root directory of the device.)*
Step 2*: Concatenate the user's Gmail-Id with the Android Device-Id.(these two values can be obtained programmatically in Android).*
Step 3 *Represent each character in the above string as a point on the given elliptic curve using Koblitz's method.*
Step 4*: Choose any 9 points among these points.*
Step 5*: Represent the pattern grid with these points*

Table 2 : Algorithm Enhanced SHA-1

Proposed Algorithm-1 (Enhanced SHA-1):

Step 1*: Represent the pattern using the given Pattern Representation algorithm.*
Step 2*: Generate an integer n by performing series of XOR operations on the character of the Device-Id.*
Step 3*: Choose a user pattern p to authenticate.*
Step 4*: Now all the points on this chosen pattern are multiplied by n using Scalar Multiplication[12] giving different points on the same curve.*
Step 5*: Now concatenate these points after converting them into hexadecimals to represent a message.*
Step 6*: Break this message into two halves and XOR with each other.*
Step 7*: Perform step 2 twice to produce an intermediate message m.*
Step 8*: Generate SHA-1 hash of the intermediate message m.*
Step 9*: Store this hash value in the device root directory to authenticate the user.*

ISSN: **1992-8645** www.jatit.org E-ISSN: **1817-3195**

Table 3: Algorithm Salted SHA-1

Proposed Algorithm-2 (Salted SHA-1):

***Step 1**: Perform the steps of algorithm 1(Enhanced SHA-1) from 1 through 7.*
***Step 2**: To make this message a 64-bit value, reverse the two halves and concatenate them to pad the string.*
***Step 3**: Mark this message as a Salt value for SHA-1.*
***Step 4**: Concatenate the salt with the original pattern selection i.e. p.*
***Step 5**: Generate SHA-1 hash of this message.*
***Step 6**: Store this hash value in the device root directory to authenticate the user.*

6. ALGORITHM-1 EXAMPLE(ENHANCED SHA-1)

1) Choose the elliptic curve parameters[19] as a=9, b=7,p=2011.
2) Let us say Gmail-Id is 'my_name@gmail.com' and Device-Id is 'f07a13984f6d116a'. After concatenation we get the string 'f07a13984f6d116amy_name@gmail.com'.
(note: We can take any identity number of the device).
3) Represent each character with corresponding point on the elliptic curve using Koblitz's encoding method. The elliptic curve representation of each character is shown in the below in table 4.

Table:4 Representing characters with points of the Elliptic Curve

f (1441,30)	**0 (363,173)**	7 (502,661)	**a (1341,250)**
1 (381,554)	**3 (421,149)**	9 (543,689)	8 (521,487)
4 (441,445)	**f (1441,30)**	**6 (481,91)**	d (1401,672)
1 (381,554)	1 (381,554)	6 (481,91)	a (1341,250)
m (1587,865)	y (1823,889)	**_ (1301,136)**	n (1603,905)
a (1341,250)	m (1587,865)	e (1421,830)	**@ (681,446)**
g (1462,123)	m- (1587,865)	**a (1341,250)**	i (1502,557)
l (1561,975)	. (321,525)	c (1382,758)	0 (1624,862)
m (1587,865)			

4) Now arrange them in ascending order of the y-coordinate and choose the first 9 points eliminating the duplicates. (we can adopt any policy to choose 9 points. They are highlighted in table 4).

5) The points that represent the grid are shown in the Figure 4.

6) Now generate an integer using Device-Id by performing series of XOR operations i.e.
f $\oplus$ 0 $\oplus$ 7 $\oplus$ a $\oplus$ 1 $\oplus$ 3 $\oplus$ 9 $\oplus$ 8 $\oplus$ 4 $\oplus$ f $\oplus$ 6 $\oplus$ d $\oplus$ 1 $\oplus$ 1 $\oplus$ 6 $\oplus$ a = 84.

7) Lets say the user's selected pattern is **1235789.**

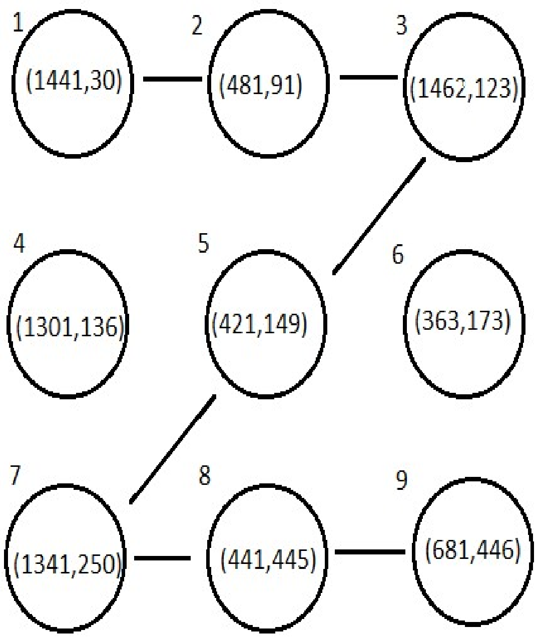


Figure:4 Pattern representations using given elliptic curve

8) The points that represent these numbers are : (1441,30), (481,91), (1462,123), (1301,136), (421,149), (363,173), (1341,250), (441,445), (681,446). Figure 4 shows the selected pattern.

9) Now perform Scalar Multiplication of the selected pattern points with the scalar generated i.e. 84.
84*(1441, 30) =(454,986)
84*(481,91)= (852,278)
84*(1462,123)=(1184,1759)
84*(421,149)=(1937,1868)
84*(1341,250)=(556,722)

84*(441,445)=(343,761)
84*(681,446) =(1837,1058)

10) Convert these points into hexadecimal and concatenate, we end up with the string

'1C63DA3541164A06DF79174C22C2D21572F97 2D422'.

11) Divide this string into 2 strings and XOR with each. Repeat this step twice. (pad with 1's if the length is odd).

1C63DA3541164A06DF79
$\oplus$
174C22C2D21572F972D422
= **67775176217160416791762 3B3**

6777517621716 $\oplus$ 04167917623B3
= **63612861434735.**

12) The highlighted string is the input for the SHA-1 Algorithm to generate a pattern hash, so the SHA-1 hash of
63612861434735 is
"A9A567A0F8959A62F694ADDCB710E9CA98 EEC576"

This hash is stored in the device memory instead of storing the hash of the original input 1235789.

7. ALGORITHM-2 EXAMPLE (SALTED SHA-1)

1) Follow the steps from 1 through 11 of the Algorithm-1 Example to generate a string i.e. 63612861434735.

2) But to make this string a 64-bit value, reverse the strings and concatenate them to pad the salt. So, 6821636 and 5374341 are concatenated to get a string 68216365374341.

3) So pad the message 63612861434735 with 68 to make it a 64-bit value. Mark the message '6361286143473568' as a **salt** value for the SHA-1 hash of the selected pattern.

4) Concatenate this salt value to the pattern which is an input for the SHA-1 Algorithm. So the input is

6361286143473568:1235789.

5) The SHA-1 hash of the input
6361286143473568:1235789 is **'12097AB19EC250499F4D545C22FC281A2CD1 33FF'**.

6) This value is stored in the device folder securely to authenticate the user whenever he logs in.

8. SECURITY ANALYSIS

The proposed algorithms dynamically generate the pattern grid representation using elliptic curve points depending on the user's identity and device identity. The existing schemes represent the grid with 1-9 integer values and pattern combination of these numbers is an input as a password[15] but the combinations are limited and known, consequently attacker can exploit the universal dictionaries easily. In the proposed schemes, these points are not fixed and change from user to user. Enhanced SHA-1 scheme makes the grid representation dynamic and generates an intermediate input, so that the attacker cannot get the hashes of input patterns using SHA-1[3] dictionaries and rainbow tables. The advantage of using salts in hashing is 'same passwords produce different hashes'. Particularly salts save the hashes from brute-forcing. A salt[14] is simply appended to make a common password uncommon. It is always safer to use different passwords for different users, and it is very common to store the salts in databases, but protecting salts is not 100% possible.

So there is always a chance for the attacker to obtain the salt value and brute-force the password value using Android forensic tools. To avoid such risks the later scheme i.e. salted SHA-1 uses the dynamic representation of the grid to generate a dynamic salt value which need not be stored in the databases to avoid brute-forcing. It is quite impossible for the eavesdroppers to generate dictionary or a rainbow table unique for each user. Both the schemes use Koblitz's encoding method which provides more security to the system and obviously discrete logs are always difficult to solve.

9. PERFORMANCE ANALYSIS

By implementing algorithm-1 for pattern authentication, we avoid the dictionary attacks and rainbow tables from the eavesdroppers since it is difficult to gain the intermediate message generated by the algorithm, whereas by implementing the salted hash i.e. algorithm-2 the attacker cannot brute-force the password, because the salt is not stored in the database, rather it is dynamically generated per user depending on the user Gmail-Id and Device-Id. This scheme saves the passwords from dictionaries. Both methods are trustworthy to avoid pre-computation attacks such as dictionaries, rainbow tables and brute-forcing. Now we may have a look into the performance evaluation of the two proposed methods.

9.1 Strict Avalanche Effect

The performance of any hash algorithm can be mainly considered with respect to the number of Collisions and Avalanche Effect. We collected observations to calculate the Avalanche Effect to evaluate the performance because finding out the number of Collisions is not feasible. Avalanche Effect is the quality of a hashing scheme that shows for a small change in input should cause a large change in the output. We have randomly taken 100 pairs of inputs where each pair differ by one bit, and recorded the number of bits changed in the output hashes, using both the methods. We have also observed whether these schemes have met the SAC (Strict Avalanche Effect). A hash function is said to satisfy the strict avalanche criterion if, a single input bit is changed, each of the output bits should alter with a probability of 50%.

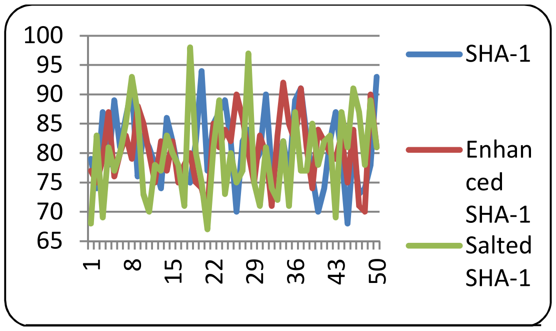


Figure:5 4-DOT PATTERNSS

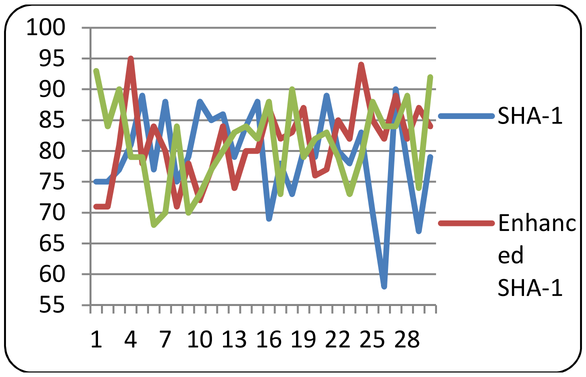


Figure:6 5-DOT PATTERNSS

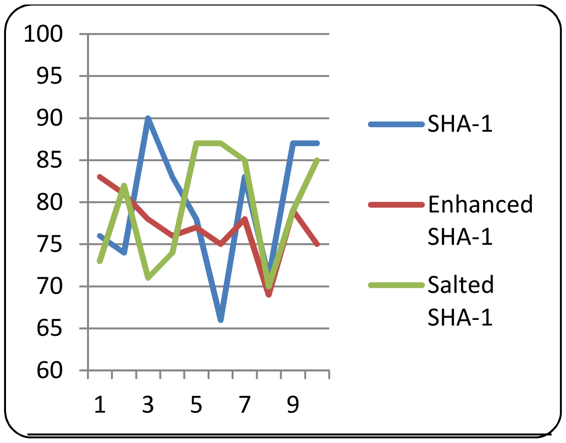


Figure:7 6-DOT PATTERNSS

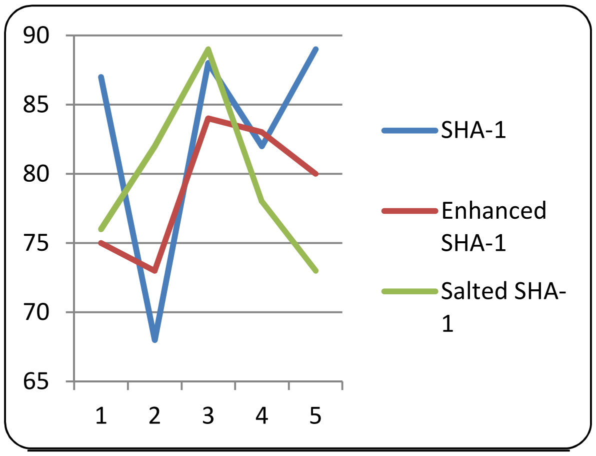


Figure:8 7-DOT PATTERNSS

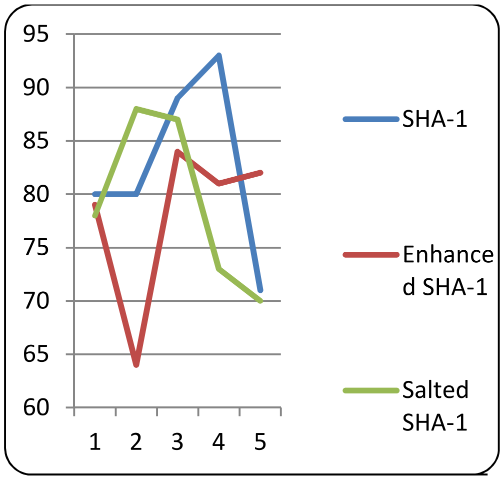


Figure:9 8-DOT PATTERNSS

The graphs 5-9 above show the Avalanche Effect shown by the two proposed methods which are almost nearer to that of the original SHA-1 hashing scheme. The X-axis represents the random input pair number and the Y-axis represent the number of bits changed for the input pair. (both inputs are differed by 1 bit).

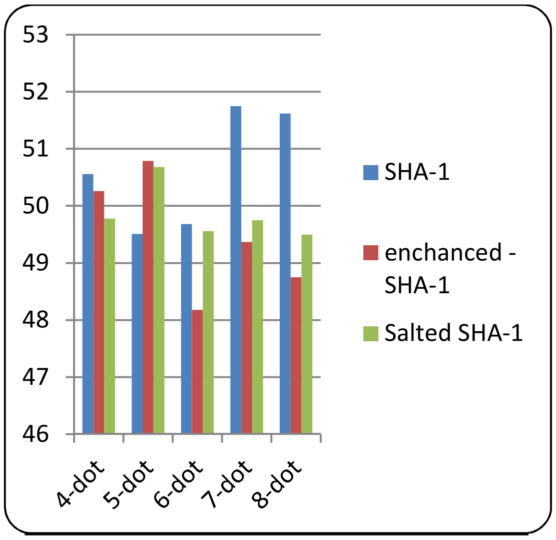


Figure 10: SAC of all the 3 method

Table 5: SAC of the 3 Methods

Pattern Scheme	*4-dot*	*5-dot*	*6-dot*	*7-dot*	*8-dot*	*Avg*
SHA-1	**50.66**	**49.5**	**49.7**	**51.7**	**51.6**	**50.6**
Enhanced - SHA-1	**50.36**	**50.8**	**48.2**	**49.4**	**48.7**	**49.5**
Salted SHA-1	**49.8**	**50.7**	**49.6**	**49.7**	**49.5**	**49.8**

Table 5 and Figure 10 show the average avalanche effect for the respective patterns and it can be observed that the two proposed schemes exhibit the SAC. Strict Avalanche Criterion of the two methods is observed by considering a number of input pairs and the average Avalanche Effect shown by each method, as shown above in the Table 5. According to the plotted graphs above and average SAC's observed in figure 10, it is obvious that the two proposed schemes show SAC with slight variation when compared with the original method SHA-1.

9.2 Time Complexities

Time complexity becomes an extremely vital issue when the scale of an application increases. Time Complexity analysis facilitates optimizing and improving the efficiency of code. Here we observed the efficiency of the proposed methods with respect to time taken by the processor to generate the hash. As we have examined here that the two methods are extensions to the existing method, it trivial to observe the time complexity compared to the original method. There will be obviously increase in the time complexity, as we have extended the existing system. These outputs are taken with a computer with Pentium-4 processor and 3GB RAM and 1.6GHz speed. We have taken observations to find out the executions times for all the pattern lengths which are shown in the following figures starting from figure 11 to Figure 15. The time periods are shown in milliseconds along y-axis. The x-axis shows the input patterns.

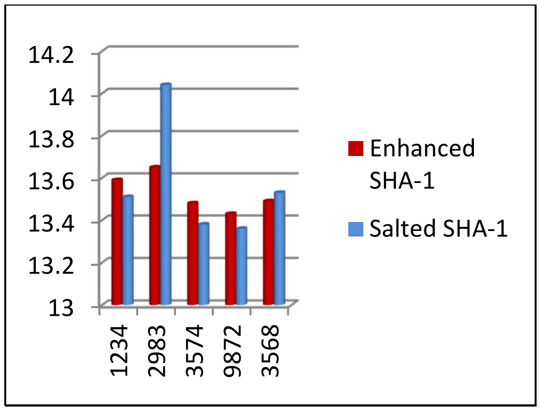


Figure 11: 4-dot pattern Execution Times

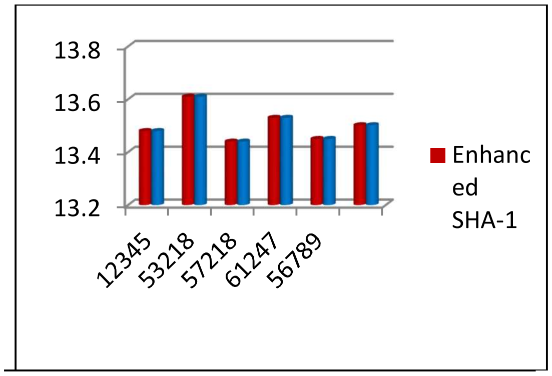


Figure 12: 5-dot pattern Execution Times

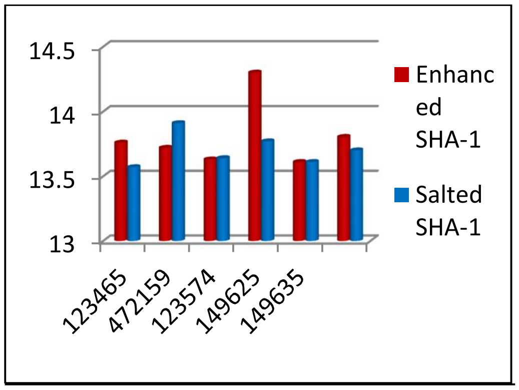


Figure 13 : 6-dot pattern Execution Times

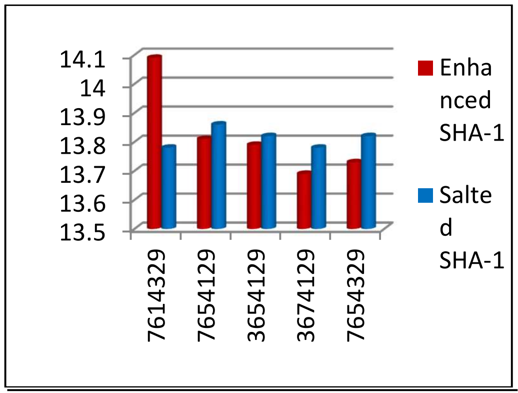


Figure 14 : 7-dot pattern Execution Times

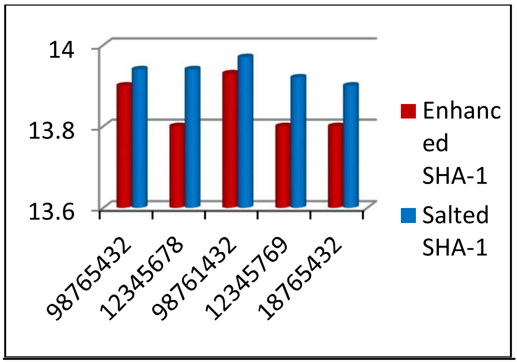


Figure 15 : 8-dot pattern Execution Times

By observing these figures 11-15 , it is apparent that there are no significant differences in the execution times of the two proposed methods.

Table 6: Average Execution Times

Avg Execution Times in millisecs	4-dot	5-dot	6-dot	7-dot	8-dot
Enhanced SHA-1	13.528	13.502	13.804	13.822	13.846
Salted SHA-1	13.564	13.502	13.7	13.812	13.934

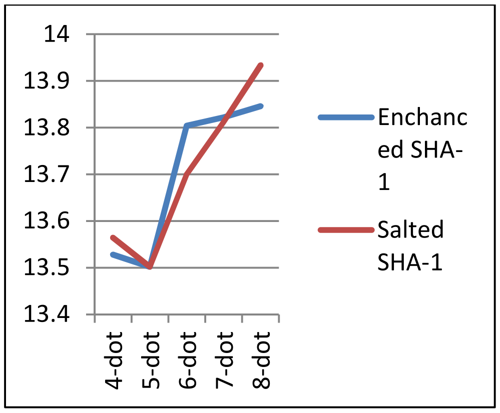


Figure 16 : Average Execution Times

By observing the above statistics and graph i.e. Figure 16 and table 6, we can deduce that the proposed methodologies show a slight increase in the time periods to calculate the hash as there is increase in the pattern size.

Since there is an added functionality in the design of these two schemes, increase in the time complexities is inevitable. There is no significant difference observed with respect to execution times in the two proposed schemes on an average.

There were no attempts by the researchers and forensic experts to improve the SHA-1 security by using a dynamic salt generation to make the system resistant to pre-computations. Our proposed methods are simple and need no hardware backend to build the system stronger against pre-computations and it generates the grid using elliptic curves.

10. CONCLUSION

Unauthorized admission to data and information by attackers, is an enormous problem with mobile security. The pattern passwords are always vulnerable to dictionaries. We need new security approaches that keep away from undesired taps on the mobiles and presents better authorization schemes than the existing one with respect to rainbow and dictionary attacks. In this paper we represented the pattern grid with points of an elliptic curve.
The smaller key sizes of ECC[18] are potentially suitable for devices such as mobile computers and smart cards and embedded systems for secure data transmissions.

This paper presents alternative methods to protect Android pattern password schemes against pre-computation attacks. The pattern grid representation using elliptic curves transforms the finite number of inputs to infinite. Enhanced
SHA-1 scheme makes it impossible for the attacker to employ dictionary attacks where as the Salted SHA-1 scheme uses the same technique to produce the a dynamic salt value which is not stored in the device database, and makes the pattern passwords difficult to crack using brute-forcing [20].

This paper compares the performance characteristics of the two proposed methods in terms of SAC (Strict Avalanche Criterion) and execution times while producing the hashes. According to the results obtained, we conclude that the two scheme exhibits SAC, and a slight increase in time complexities is observed when the size of the pattern increases.

11. LIMITATIONS AND ASSUMPTIONS

The proposed methodologies give the impression that they prevent the pre-computations on pattern lock passwords. But these methods also have certain limitations. The only limitation of these methods is we need to keep the design of this algorithms secret. Maintaining the algorithm secret is obviously un-trivial.

Even the algorithm is revealed, the system is secure because it involves few secret values such as elliptic curve parameters, device identification values and mail-id of the user. If the attacker gains these values, the password becomes vulnerable tobrute-forcing[26].

As far as the Salted SHA-1 scheme is concerned, without knowing the application design if the cryptanalyst knows that a salt is added to the password, brute-forcing becomes difficult because the salt is not stored in the database. But here also we assume that the algorithm is kept secret because the salt[27] generated is an application secret.

12. FUTURE WORK

Other Android Password Authentication systems such as PIN passwords[25] and Alphanumeric Password Systems also suffer from pre-computation attacks even though they employ salt values to generate the hashes. If the mobiles are rooted and USB mode is enabled, the hacker can use mobile forensic tools[24] to gain the password hashes. So we need to chase for a better and alternative security system to defense against these attacks. Here the two proposed methodologies can also be applicable to these authentication schemes also.

REFERENCES

[1] Bh.Padma, "Encoding and Decoding of a message in the implementation of Elliptic curve Cryptography using Koblitz" Method", International Journal On Computer Science and Engineering (IJCSE), volume-2 issue:5 , 2010 pp 1904-1907, ISSN: 0975-3397.

[2] Bh.Padma and GVS Raj Kumar, "A Review on Android Authentication system vulnerabilities", International Journal of Modern Trends in Engineering and Research(IJMTER), volume 3, Issue 8, 2016 pp 118-123, ISSN: 2349.

[3] Stephane Manuel, Classification and generation of disturbance vectors for collision attacks against SHA-1 , Des. Codes Cryptography 59 (2011), no. 1-3, 247–263.

[4] Bh Padma, Efficient Computation of Point Multiplication in the Implementation of Elliptic Curve Cryptography, ,E - Commerce for Future &Trends,STM Journals, Volume 1 , Issue 1.

[5] N. Koblitz, "Elliptic Curve Cryptosystems", Mathematics of Computation, 48, pp. 203-209, 1987.

[6] V. Miller, "Uses of elliptic curves in cryptography", Advances in Cryptology: proceedings of Crypto'85, Lecture Notes in Computer Science, vol. 218. New York: Springer-Verlag, 1986, pp. 417-426.

[7] "Cellphone Forensic Tools: An overview and Analysis "available at http://csrc.nist.gov/publications/nistir/nistir-7250.pdf.

[8] Android Forensics:How To Bypass The Android Phone Pattern Lock http://niiconsulting.com/checkmate.2014/04/how-to-bypass-the-android-phone-pattern-lock/

[9] "Android Explorations" available at https://nelenkov.blogspot.in/.

[10] G.V.S.Raju and Rehan Akbani,2003, "Elliptic Curve Cryptosystem and its Applications",2003,TheUniversity of Texas at san Atonio.

[11] B.V. Rompay, "Analysis and Design of Cryptographic Hash functions, MAC algorithms and Block Ciphers", Ph.D. thesis, Electrical Engineering Department, KatholiekeUniversiteit,Leuven, Belgium, 2004.

[12] Miller, V. (1986) 'Uses of elliptic curves in cryptography', in Advances in Cryptology: Proceedings of Crypto'85, Lecture Notes in Computer Science, Vol. 218, Springer-Verlag, New York, pp.417–426.

[13] Bh.Padma(2010), "Implementation of Optimizing point Multiplication in Elliptic Curve Cryptography using Binary Method", Journal Of Computer Science(JCS),Sep-Oct2010,vol-4,issue: 6.http://www.karpagamjcs.in/index.php/abstracts/articles/volume_4_issue_6_article_6

[14] Cryptographic Hash Functions: A Review Rajeev Sobti1 , G.Geetha2 IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 2, No 2, March 2012 ISSN (Online):1694-0814 www.IJCSI.org.

[15] Dynamic Salt Generation and Placement for Secure Password Storing Sirapat Boonkrong and Chaowalit Somboonpattanakit, IAENG International Journal of Computer Science, 43:1,IJCS_43_1_04. http://www.iaeng.org/IJCS/issues_v43/issue_1/IJCS_43_1_04.pdf.

[16] Tao, H. and Adams, C. 2008. Pass-Go: A proposal to improve the usability of graphical passwords, International Journal of Network Security

[17] Aviv, A. J., Gibson, K., Mossop, E., Blaze, M., and Smith, J. M. 2010. Smudge attacks on Smartphonetouch screens. In USENIX 4th Workshop on Offensive Technologies.

[18] S. Wiedenbeck, J. Waters, J. C. Birget, A. Brodskiy, and N.Memon, "Authentication Using Graphical Passwords: Basic Results", In Human-Computer Interaction International (HCII 2005), Las Vegas, NV, 2005.

[19] Darrel Hankerson, Julio Lopez Hernandez, Alfred Menezes, Software Implementation of Elliptic Curve Cryptography over Binary Fields, 2000, Available at http://citeseer.ist.psu.edu/hankerson00software.html

[20] Certicom, Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0, September 2000, Available at http://www.secg.org/download/aid-386/sec2_final.pdf

[21] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone,Handbook of Applied Cryptography, CRC Press, 1996.

[22] William Stallings, Cryptography and Network Security, Principles and Practice.ed.,Prentice Hall,New Jersey,2003.

[23] N. Koblitz. A Course in Number Theory and Cryptography, Springer-Verlag, second edition, 1994.

[24] http://android-forensics.com/android-forensics-study-of-password-and-pattern-lock-protection/143.

[25] http://forensics.spreitzenbarth.de/2012/02/28/cracking-the-pattern-lock-on-android.

[26] Padma, Bh., Raj Kumar, GVS., 2016. Design And Analysis of An Enhanced SHA-1 Hash Generation Scheme for Android Mobile Computers. International Journal of Applied Engineering Research (IJAER). ISSN 0973-4562 Volume 11, Number 4, pp 2359-2363.

[27] Padma, Bh., Raj Kumar, GVS., 2017. Dynamic salt generation for mobile data security using elliptic curves against precomputation attacks. Int. J. Image Mining. Vol. 2, Nos. 3/4, pp 179-194.