Do You Think Your Passwords Are Secure?

Analyzing the Security of Android Password-Managers

Dominik Ziegler, Mattias Rauter, Christof Stromberger, Peter Teufl and Daniel Hein
Institute for Applied Information Processing and Communications
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

***Abstract*—Many systems rely on passwords for authentication. Due to numerous accounts for different services, users have to choose and remember a significant number of passwords. Password-Manager applications address this issue by storing the user's passwords. They are especially useful on mobile devices, because of the ubiquitous access to the account passwords.**

Password-Managers often use key derivation functions to convert a master password into a cryptographic key suitable for encrypting the list of passwords, thus protecting the passwords against unauthorized, off-line access. Therefore, design and implementation flaws in the key derivation function impact password security significantly. Design and implementation problems in the key derivation function can render the encryption on the password list useless, by for example allowing efficient brute-force attacks, or – even worse – direct decryption of the stored passwords.

In this paper, we analyze the key derivation functions of popular Android Password-Managers with often startling results. With this analysis, we want to raise the awareness of developers of security critical apps for security, and provide an overview about the current state of implementation security of security-critical applications.

I. Introduction

Researchers have recently shown, that the average user has around 25 online accounts [1] consisting of credentials for e-mail, online-shopping, social networks, finance, banking or any other online system that needs to verify the identity of a customer. With password based authentication being the most commonly used system, attacks such as phishing, dictionary-attacks, or brute-forcing, have become a lucrative business [2][3][4]. Attackers are further encouraged by a bad habit: Many users share passwords over multiple accounts [1]. As a consequence, once a password gets stolen, every associated account is compromised. Furthermore, users often choose passwords with a low entropy allowing for efficient brute-force or dictionary attacks. Different password policies try to address this issue by defining minimum password requirements. Examples are policies, such as "minimum-password-length", "different password for each account", "usage of special characters" and similar [5].

Unfortunately, increasing the password complexity, does not necessarily increase the security and most certainly decreases the usability: Users tend to forget long and hard to guess passwords [6][7], or even worse, store them unencrypted on their smartphone [8]. Furthermore, those users, who do use a strong password, might think themselves safe and thus use it on multiple sites.

So called *Password-Managers* [9][10] try to address this issue. Password-Managers offer the ability to securely save user credentials in an encrypted form. Many Password-Managers use a single master password, in order to protect their credentials [11]. Obviously, the security of the chosen master password significantly impacts the security of the stored credentials. The actual security is also dependent on the key derivation function used to convert the master password into a key suitable for encryption. A key derivation function typically uses a pseudo-random function such as an HMAC to derive a key from a password. Furthermore, a good key derivation function tries to amend for low entropy in the master password, while at the same time making the key derivation a cryptographically secure and protracted process. Key derivation functions use a random value, a so-called salt, to increase the entropy of the master password and thus prevent efficient dictionary attacks, where one dictionary can be used for all instances of the key derivation function. Furthermore, by making the key derivation a protracted process, trying all possible combinations of passwords (brute-forcing) requires an inordinate amount of time, and with newer key derivation functions also memory. However, for usability reasons, optimum key derivation times are a compromise between security and usability. A user would hardly be willing to wait a significant amount of time and close all other applications just to get access to her credential store. The actual amount of time depends on how fast the key derivation function evaluates on the target device. Therefore, security and usability depend on the specific use case and a general guideline cannot be given.

Due to the offered convenience, Password-Managers have gained a lot of popularity over the last couple of years [12]. The success of smartphones and tablets made Password-Managers ubiquitous. This in turn entailed serious consequences: Smartphones can be stolen or lost fairly easy [13]. A correct implementation of a mobile Password-Manager is therefore all the more important as the encrypted database might fall into wrong hands quite unexpectedly. Furthermore, with today's prevalence of cheap online storage, a Password-Managers password list can be stored in the cloud for backup and availability reasons. Users are often unaware how much trust they put into such a service, when they store their password list there.

Apart from the aforementioned Password-Manager appli-

cations, key derivation functions play a crucial role in all applications that cannot rely on platform encryption features[1]. Examples are container applications that are used within Bring-Your-Own-Device (BYOD) environments and security-relevant consumer applications, such as mobile banking applications.

Due to the importance of secure implementations for these security critical components, knowing the current state of the implementations is crucial for the accurate estimation of the risk factors involved in security relevant deployment scenarios. Since, Password-Managers are widely spread and are very important for the protection of sensitive data, they were chosen as target for our security analysis.

In this paper, we manually examine the key derivation functions of various Password-Managers for the Android platform. Many of the analysed applications suffer from significant design and/or implementation errors. Apart from the conducted security analysis, we also implement brute-force tools for estimating the brute-force costs, of password classes with different entropy, for each analysed application. Finally, we summarize common implementation mistakes and thereby aim to raise the awareness of the respective application developers. We adhere to the responsible disclosure guideline and have informed the developers of the analysed software about our results and refrain from disclosing the names of the analyzed applications here.

II. Background & Related Work

In this section we provide information about related work, as well as an introduction to password-based key derivation functions, such as PBKDF (defined in "PKCS #5: Password-Based Cryptography Specification" [14]), Bcrypt and Scrypt.

A. Related Work

Recently Zhao and Yue examined the built-in Browser Password-Managers of the most common Web browsers and showed that all of them suffered from serious vulnerabilities [15]. Additionally they proposed a cloud based Password-Manager design to securely store credentials.

In 2005 Halderman, Waters and Felten came up with a client-side Password-Manager design, to generate and store arbitrary long and secure passwords [11]. The credentials are secured through a single, easy to remember master password. To prevent brute-force attacks, the key derivation function uses a strengthened cryptographic hash function, respectively applying a hash function multiple times to increase the computation time.

Indeed, in the last couple of years, several studies have pointed out, that more than one long and complex password is difficult to remember. As a result users tend to write down or store their passwords somewhere else [1][6][7][9][16]. Apart from well-known brute-force or dictionary attacks on passcodes [3][4], other attacks, such as phishing, which directly target the user, have also become widely spread.

As a consequence, researchers have developed several alternatives like graphical passwords or hardware tokens, to replace password based authentication. They showed, that their methods can be resistant to brute-force or similar attacks [17][18][19][20][21].

B. Password-Based Key Derivation Functions

Most of the time, passwords cannot be used directly as a key. This is due to the fact, that common cryptographic algorithms need much longer key sizes. As a result, pseudo-random functions are used to extend passwords to the appropriate length.

The key-space for password-based key derivation functions directly depends on the allowed characters (n) and the length (k) of the used password, resulting in n^k possible keys. Short passwords, using only characters [a-z] would therefore decrease the key-space dramatically which in return would allow quite efficient brute-force attacks.

1) PBKDF: The Public-Key-Cryptography-Standard #5 [14] defines PBKDF2[2] as well as its predecessor PBKDF1 to circumvent the issues of the limited key-space. While PBKDF1 was only capable of generating key sizes up to *160bit*, PBKDF2 allows an arbitrary key length. According to [14] it is defined as follows:

$$DK = PBKDF2(P,\ S,\ c,\ dkLen), \tag{1}$$

where DK denotes the derived key, P the password, S the salt, c the iteration-count, and $dkLen$ the desired key length. PBKDF2 allows the specification of a pseudo random function, such as *HMAC* [22]. HMAC is an algorithm for message authentication using hash functions and a pre-shared key. To deal with a small key space c has to be chosen accordingly. For example, the NIST recommendation for Password-Based Key Deriviation [23] requires a minimum of 1000 iterations, but also states that "the number of iterations should be set as high as can be tolerated for the environment". If selected adequately, this increases the computation time for a brute-force attack dramatically. To prevent dictionary attacks, a random salt can be chosen.
On the Android Platform, a key derivation function based on PKCS#5 is provided through `PBEKeySpec`[3].

Brute-force attacks on the PBKFD2 function can be sped-up with dedicated hardware. Hence, different design proposals for key derivation functions have been made, including scrypt (Section: II-B2) which is based on memory-hard algorithms.

2) Scrypt: In 2010 Percival proposed scrypt [24][25], a key derivation function based on memory-hard algorithms, to deal with the vulnerability of PBKDF2 to brute force attacks on special hardware. While previous assumptions are based on the premise that attackers are limited to the same hardware as users, Percival points out, that by parallelism the cost of a brute force attack decreases each year. Scrypt is already deployed by the file-system encryption system of Android 4.4. However, we

[1] Platform encryption can easily be enforced for managed devices. Due to the tight operating system integration and additional hardware-support, platform encryption systems typically offer a high level of protection. Unfortunately, in non-managed environments their availability and correct configuration cannot be guaranteed.

[2] Password-Based-Key-Derivation Function v2.0

[3] http://developer.android.com/reference/javax/crypto/spec/PBEKeySpec.html

are not going into further details, because none of the analyzed applications utilize Scrypt.

III. App Analysis

In this chapter, we discuss the key derivation functions of several Android Password-Manager applications. However, the apps presented here only represent a digest of all available apps and have been manually selected based on their number of users, positive ratings, and ranking in the search results in the Google Play Market[4]. This choice can be seen as an empirical selection based on how a regular user would choose a Password-Manager, when searching on the Google Play Market. We manually analyzed 45 applications, of which we present 8 here. We disregarded the other 37, because 14 were obfuscated, 16 were paid apps, and 7 used the key derivation function in accordance with our requirements. For our manual analysis we used apktool[5], dex2jar[6], and JD-GUI[7] to analyze the mentioned applications.

A. Assumptions

The conducted analysis concentrates on the deployed key derivation functions and does not consider the implementation of cryptographic functions required to protect the stored passwords.

In our scenario we assume that an attacker can gain full access to a smartphone without platform encryption[8] either by theft, loss of the device, or any other sophisticated method. We further assume that the attacker has basic knowledge in cryptography and is capable of extracting the file-system and analyzing Android applications. Any person capable of developing Android applications and using widely available tools for analyzing the Dalvik code in the Android application packages fulfils these assumptions.

B. PM-App1

This Android application is a Password-Manager tool that allows the user to enter an alpha-numeric master-password, including some special characters, to protect the user's credentials. According to the Google Play Market this app was installed for 100,000 - 500,000 times (Jan. 2014). Additionally the application offers premium features to enhance the usability. According to the developer's website, these in-app-purchase options do not affect the implemented security mechanisms. Therefore, only the free version was subject of our analysis.

The application comes with a backup functionality to store a zip-file, including the encrypted database, on an SD-card. While this is very convenient for data recovery, an attacker can directly access the data on this card, without the need to gain access to the app's data stored on the internal storage device (e.g., by rooting the device).

[4]https://play.google.com

[5]https://code.google.com/p/android-apktool/

[6]https://code.google.com/p/dex2jar/

[7]http://jd.benow.ca

[8]This is a plausible assumption due to the need to manually activate the platform encryption system on Android.

The key derivation function used to obtain a key from the user's master-password is implemented as follows: If the password-length is smaller than 32 characters, the password is repeated until the length of the new password-string is greater than or equal to 32. The resulting string, which is at least 32 characters long, is UTF8-encoded and copied into a byte-array. The first 32 bytes of that array are used as 256 bit key.

```
set password to ... /* userinput */

while getLengthOf(password) < 32 do
  password = password + password;
end while

utf8_password = getUTF8EncodingOf(password)

key = getFirst32BytesOf(utf8_password)
```

Consequences: The used key derivation function does not have the security properties of a real key derivation function and allows very efficient brute-force-attacks. Furthermore, recurring patterns within the user-password are ignored. Assuming a password "abababab", the user-input "ab" would decrypt the data, because of the implemented repetition of too short passwords.

C. PM-App2

This application, which is installed 10,000 - 50,000 times (Jan. 2014), allows the user to add items with predefined fields (type, description, account, password...). Further it is possible to choose a privacy class, which defines the level of security (e.g., show passwords without entering the master-password). The items are protected by a user-defined master-password. The password can consist of letters, numbers and some special characters and is used to encrypt the private fields of the items. The user-data is stored in XML files saved on the SD-card.

As key derivation function, the built-in HMAC-method provided by the Android Platform was chosen, but the algorithm is applied only once. A static UTF-8-encoded string initializes the HMAC-function. The resulting key is then used to encrypt the database with the AES implementation of the BouncyCastle [26] provider.

```
set password to ... /* userinput */

utf8_password  = getUTF8EncodingOf(password)
mac_init_value = getUTF8EncodingOf(STATIC_STRING)

key = performHMACWithSHA256(utf8_password,
                            mac_init_value)
```

Consequences: There are two major problems with the implemented HMAC-based key derivation function: **(1)** The iteration count is set to 1, which allows highly efficient brute-force attacks. **(2)** A static salt value is used, which allows to pre-compute all possible keys. **(3)** Although HMACs are also deployed by the PBKDF2 function, using this function in a custom way comes with the increased likelihood of security flaws, as demonstrated here with the static initialization value and the low iteration count.

D. PM-App3

PM-App3 was installed 1,000,000 - 5,000,000 times (Jan. 2014) and provides the ability to save credentials based on predefined or custom groups. The database is a proprietary binary file which can be saved anywhere on the phone. To protect the accounts either a master-password, a keyfile or a combination of both can be chosen. For our analysis we only considered the former.

The program implements a custom key derivation function: **(1)** The chosen master-password is hashed, using SHA-256. Two random seeds are generated using Java `SecureRandom`. The hashed master password is then encrypted using AES/ECB/NoPadding with 300 iterations. The first seed is used as salt. **(2)** The result is again hashed using SHA-256. The resulting check-sum is once again hashed with SHA-256 together with the second seed. The output is then used as an encryption key.
To verify validity of the key, the database needs to be decrypted in AES/CBC/PKCS5 Padding mode with the previously computed key. If the decryption key is correct, the check-sum saved over the decrypted database and the reference value will match. Otherwise an exception is raised, indicating that the use key was not correct.

```
set password to ... /* userinput */

setPassword(password);
key = generateMasterKey(
            seed,
            transformationSeed,
            encryptionRounds);
```

Consequences: The iteration count is set to 300, which results in a basic level of security. Additionally, a random salt is generated in order to prevent a pre computation of all possible keys. Still, compared to the reference implementation (see Figure 1), a full brute-force attack takes on average only half as long. It is recommended to increase the iteration count, and use a well-known key derivation function to avoid implementation mistakes.

E. PM-App4

PM-App4 is a simple Passwort-Manager. According to the Google Play Market it was installed 100,000 - 500,000 times (Jan. 2014). It offers the ability to add credentials, notes or URLs. Additionally, passwords can be generated with a built-in password generator. The application saves the provided information in an SQLite database on the internal storage, but provides a possibility to export the database to another storage location (e.g. the SD card).

Similar to the PM-App2 mentioned in section III-C this application uses an HMAC as key derivation functions, which takes a static-key and a user chosen master password as input. To verify the derived master key, cyclic redundancy check (CRC32) has been implemented. The checksum is saved as a base64 string in the database. Since the implementation resembles the one of PM-App2, the resulting pseudo-code looks similar (see section III-C).

Consequences: In general, an HMAC, given enough iterations, fulfils the requirements of a secure key derivation function. However PM-App4 initializes the HMAC with a static salt, allowing the pre computation of all possible keys. Additionally only one round is utilized to derive the key. This allows for very efficient brute-force attacks.

F. PM-App5

In addition to storing credentials, PM-App5 allows to protect other sensitive data, such as credit card or bank account information. Also, online-sync via Dropbox can be setup as a backup solution. This app was installed 500,000 - 1,000,000 times (Jan. 2014).

The key is derived via the build in `PBEKeySpec2` function using 100 iterations and a random salt. In addition to that, PM-App5 saves a SHA-512 hash, including a salt, of the master password in its preferences file to validate the entered login information.

```
set password to ... /* userinput */

checksum = getSHA512Hash(password)
```

Consequences: PM-APP5 uses the built-in `PBEKeySpec2` function in order to derive a key, which in practice provides sufficient security. However by saving the hashed master password, an oracle is provided that allows the verification of an input password in a fraction of the time of the actual key derivation function. This significantly simplifies brute-force attacks. Also, the SHA512 hash operation is only performed on the passcode, which allows to pre-calculate password tables. Also, the iteration count of the deployed `PBEKeySpec2` should be increased to increase the time required for brute-force attacks.

G. PM-App6

PM-App6 ships with a Desktop client to automatically sync login information via different devices. This app was installed 100,000 - 500,000 (Jan. 2014). Credentials cannot be manually added on the device but rather have to be created on a computer. PM-App6 saves each retrieved login information in a file on the external storage of the device.

To display the login information, a previously chosen master password has to be entered. Conveniently this information gets stored in the preferences file together with the username and password of the account. The app stores the data in an obfuscated, but *not* encrypted representation. The obfuscation and de-obfuscation process is based on methods that can easily be extracted from the application code.

Consequences: The master password can be directly extracted by calling the methods used for de-obfuscation without the requirement for a brute-force attack.

H. PM-App7

This Password-Manager allows users to manage passwords only. According to Google's Play Store this application was

installed 100,000 - 500,000 (Jan. 2014) times. The application itself is protected by a four digit PIN code which can be chosen at the first startup. However, after minimizing the app the PIN code is not required anymore until the application is either manually closed via the task manager or the Android operating system removes it from the memory.

Our investigation has shown that this application does not use any key derivation or encryption at all. Furthermore, our analysis shows that this app is vulnerable to SQL injections. The following code shows how the master password is stored into the database:

```
set password to ... /* userinput */

query = "INSERT into TABLE_NAME
         values (1, '" + password + "');"
```

The unlock functionality in the application is implemented by comparing the plain text PIN code from the database with the entered PIN code at the startup of the application.

Consequences: By reading the database on the device the attacker can read the master password (PIN code) and all the stored passwords in plain text.

I. PM-App8

This application is installed 50,000 - 100,000 (Jan. 2014) times. The app provides the ability to store passwords, credit card information, account data and addtional information. Access protection is based on a digital master key that must be defined by the user when the app is started for the very first time. This master key is required when starting or resuming the application.

Our investigations reveal that the custom key derivation function consists of three variables, where one of them is the iteration count. Thus, the entropy for the key depends on the other two variables. Both of them are calculated by misleadingly named methods implemented in different classes. Taking a closer look at the implementations of these methods reveals, that both methods are based on shuffling and replacing bytes of a static initialization array. As a result these methods do not provide sufficient entropy.

Consequences: The key is derived from static values and the master password does not influence the encryption key. In fact, the master password is stored in the database and is encrypted with the aforementioned static key.

We were able to mount an attack by using the same statically defined variables on all encrypted information stored in the database. Thus we were able to decrypt the master password and all user stored passwords by just retrieving the database from the device and decrypting the values using the aforementioned approach.

IV. Brute Force Comparison

To compare the security level of the identified key derivation functions, we have implemented a brute-force tool. The Java-based tool analyzes the mean execution time by executing 1000 iterations of a given key derivation function. Additionally we have implemented a tool to recover the master passwords of the analyzed applications. To achieve comparable results and attach a price tag to the brute-force attacks, we used the Amazon EC2 infrastructure. The brute-force times were calculated by using one "EC2 Compute Unit"[9] (ECU), which – referring to Amazon's definition – is the equivalent of a 1.0 to 1.2 GHz 2007 Opteron (AMD) or 2007 Xeon (Intel) processor, as a measurement unit. This way, we can extrapolate the time needed to launch a full brute-force attack, as can be seen in Table 1, but it also allows us to calculate the costs for such an attack. The brute-force times and costs have been calculated for 1 ECU. The current price for one CPU hour on a small on-demand instance with 1 ECU is currently 0.06$. This price could be further reduced by reserving instances. However, the conducted calculations should only give a general estimate of costs and thus, further optimizations, such as dedicated hardware, the use of GPUs, or different pricing models have not been considered.

For ease of comparison, we developed a reference implementation, that uses PBKDF2 with HMAC-SHA1 and 1000 iterations, as shown by T. Johns [27]. The brute-force times in Table 1 have only been calculated for 1 ECU. However, due to the effective paralellization of the brute-force attacks, an arbitrary number of instances could be reserved (within the limitations set by the Amazon EC2 infrastructure), which would reduce the brute-force times accordingly (divided by the number of instances), without influencing the brute-force price.

V. Common Problems

While analyzing applications for this paper, we found many security issues and implementation errors. The most common problems can be summed up in four categories:

No Encryption: In some cases, no encryption functionality has been utilized to protect the sensitive data.

No Key Derivation Function: Some applications did not use a key derivation function at all. UTF8-encoding or similar functions are not suitable key derivation functions and allow either highly efficient brute-force attacks or – even worse – the direct extraction of the encryption keys or the protected data.

In case a key derivation function is implemented, the following common issues could be identified:

Low number of iterations: Even, if an adequate key derivation function was used, in many cases a low iteration count was chosen. Combined with the fact that rather short passcodes are used on mobile devices (due to usability), this leads to highly efficient brute-force attacks. Unfortunately, there is no universal answer to the question on how many iterations should be used. The number of iterations depends on the use case, the available hardware and the right balance of security and usability for the envisaged deployment scenario. A possible approach in solving this problem is the utilization of so-called calibration functions that calculate the number of iterations for the given key derivation time on the current device.

Static salt: In many cases the salt value has not been chosen randomly. Either a static salt has been used, or one

[9] http://aws.amazon.com/ec2/faqs/#What_is_an_EC2_Compute_Unit_and_why_did_you_introduce_it

	Numerical 10 numbers			**Alphanumerical** 10 numbers, 26 letters			**Alphanumerical** lower/uppercase letters and numbers 10 numbers, 52 letters			**Complex** lower/uppercase letters, numbers and symbols 10 numbers, 52 letters, 45 symbols		
Passcode length	4	6	8	4	6	8	4	6	8	4	6	8
Possible characters	10	10	10	36	36	36	62	62	62	107	107	107
Possible passcodes	1,00E+04	1,00E+06	1,00E+08	1,68E+06	2,18E+09	2,82E+12	1,48E+07	5,68E+10	2,18E+14	1,31E+08	1,50E+12	1,72E+16
Application	**Days to try out 100% of passcodes**											
Reference implementation Time/Costs	0,001	0,14	14	0,2	311	402.945	2	8.113	31.185.942	19	214.352	2.454.118.730
	$ 0,002	$ 0,21	$ 21	$ 0,3	$ 448	$ 580.240	$ 3	$ 11.683	$ 44.907.756	$ 27	$ 308.667	$ 3.533.930.971
PM-App 3 Time/Costs	0,000	0,05	5	0,1	107	138.596	0,7	2.790	10.726.632	6	73.728	844.112.058
	$ 0,001	$ 0,07	$ 7	$ 0,1	$ 154	$ 199.578	$ 1	$ 4.018	$ 15.446.351	$ 9	$ 106.168	$ 1.215.521.363
PM-App 1 Time/Costs	0,000	0,01	1	0,02	30	39.009	0,2	785	3.019.105	2	20.751	237.582.791
	$ 0,000	$ 0,02	$ 2	$ 0,03	$ 43	$ 56.173	$ 0,3	$ 1.131	$ 4.347.512	$ 3	$ 29.882	$ 342.119.218
PM-App 4 Time/Costs	0,000	0,01	1	0,01	11	14.367	0,1	289	1.111.917	1	7.643	87.500.222
	$ 0,00	$ 0,01	$ 1	$ 0,01	$ 16	$ 20.688	$ 0,1	$ 417	$ 1.601.161	$ 1	$ 11.005	$ 126.000.320
PM-App 2 Time/Costs	0,000	0,00	0	0,0	7	8.447	0,0	170	653.792	0	4.494	51.448.937
	$ 0,000	$ 0,00	$ 0	$ 0,01	$ 9	$ 12.164	$ 0,1	$ 245	$ 941.461	$ 1	$ 6.471	$ 74.086.470
PM-App 5 Time/Costs	0,000	0,00	0,1	0,0	1	1.512	9,0	30	117.014	0	804	9.208.205
	$ 0,00	$ 0,00	$ 0,08	$ 0,00	$ 2	$ 2.177	$ 13,0	$ 44	$ 168.500	$ 0,1	$ 1.158	$ 13.259.815
PM-App 6	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PM-App 7	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
PM-App 8	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0

Fig. 1: Brute-force times (days) and costs ($) for the described applications: The values are calculated based on the mean execution time (1000 iterations) of the key derivation function. 0.0 indicates, that the password can be recovered without a brute-force attack (PM-App 6, PM-App 7, PM-App 8). Passcodes with brute-force costs are marked in the folling way: 0$ to 99.999$ (black), 100.000$ to 199.999$ (grey), from 200.000$ (white). The thresholds for these colors-markings have been chosen arbitrarily and are intended to provide a better overview of the table. A categorization based on the risk level, depends on the deployment scenarios and the results of a carefully conducted risk analysis.

that can easily be derived from device-specific identifiers (e.g. the IMEI). Since, the main purpose of the salt is increasing the entropy and thereby eliminating the possibility to pre-calculate password tables, an inadequate choice facilitates attacks.

Implementation/Design errors: Our analysis reveals that in many cases implementation errors or weak design choices have a strong negative impact on the achieved security level. **(1)** Oracles: While application developers use existing key derivation functions in the right way, in some cases ”oracles” where created that allow a brute-force attack to circumvent the key derivation function. **(2)** Custom implementations: There are many sources that strictly recommend against the custom implementation of cryptographic functions due to the likelihood of implementation or design errors. Still, these custom implementations exist in security-critical applications and unfortunately, often include mistakes that significantly decrease the level of security. **(3)** Wrong parameters: Even if a proper key derivation function is deployed its parameters are often used in a wrong way. The aforementioned parameters ”salt” and ”number of iterations” are an example for such parameters. However, newer key derivation functions, such as Scrypt include additional parameters that need to be set correctly – depending on the deployment scenario – by the application developers.

VI. Conclusion

Since password-based authentication is still a commonly used system, choosing secure credentials and providing secure storage is a crucial aspect for account security. Password-Managers that are typically used to store such credentials mostly rely on key derivation functions that derive cryptographic keys from master passcodes. Unfortunately, the analysis of popular Android Password-Managers reveals, that despite the fact that secure key derivation functions are offered by the standard operating system APIs, many implementation mistakes are made by the developers. These errors significantly downgrade the level of security offered by the Password-Managers and result in very efficient brute-force attacks, or even worse, the direct extraction of the sensitive data. It should be noted at this point, that, even tough we analyzed only free and non-obfuscated applications, the outcome is not less valuable. Free applications should provide, up to a certain point, the same amount of security as commercial programs. Additionally some of the analyzed applications were available as a paid version as well. To some extend it is therefore possible to draw conclusions from its free counterpart. Based on our initial analysis we conclude that secure implementations – especially within a mobile context – are not yet the standard, even in security-critical applications.

Although the analysis in this paper has been limited to Password-Managers, many other applications, such as container applications deployed in BYOD scenarios or mobile banking applications, also require password encryption systems. In that sense, this paper only represents a first step in the direction of evaluating the security of password-based encryption functions in mobile applications. In ongoing and future work we aim to simplify the current manual analysis procedure by using automated static and dynamic analysis tools, and to extent the scope of our analysis to other applications and platforms.

References

[1] D. Florencio and C. Herley, "A large-scale study of web password habits," in *Proceedings of the 16th international conference on World Wide Web - WWW '07*. New York, New York, USA: ACM Press, 2007, pp. 657–666. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1242572.1242661

[2] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *Proceedings of the 12th ACM conference on Computer and communications security - CCS '05*. New York, New York, USA: ACM Press, 2005, pp. 364 – 372. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1102120.1102168

[3] B. Pinkas and T. Sander, "Securing passwords against dictionary attacks," in *Proceedings of the 9th ACM conference on Computer and communications security - CCS '02*. New York, New York, USA: ACM Press, 2002, pp. 161 – 170. [Online]. Available: http://portal.acm.org/citation.cfm?doid=586110.586133

[4] P. Soni, S. Firake, and B. B. Meshram, "A phishing analysis of web based systems," in *Proceedings of the 2011 International Conference on Communication, Computing & Security - ICCCS '11*. New York, New York, USA: ACM Press, 2011, pp. 527–530. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1947940.1948049

[5] A. Singer, W. Anderson, and R. Farrow, "Rethinking Password Policies," *USENIX*, vol. 38, no. 4, pp. 14–18, 2013.

[6] A. Adams and M. A. Sasse, "Users are not the enemy," *Communications of the ACM*, vol. 42, no. 12, pp. 40–46, Dec. 1999. [Online]. Available: http://dl.acm.org/citation.cfm?doid=322796.322806

[7] J. Yan, A. Blackwell, R. Anderson, and A. Grant, "Password memorability and security: empirical results," *IEEE Security & Privacy Magazine*, vol. 2, no. 5, pp. 25–31, Sep. 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1341406

[8] RTT Staff Writer, "Survey Shows 18% Of U.S. Users Store Secret Passwords On Their Smartphone," 2013. [Online]. Available: http://www.rttnews.com/2208747/survey-shows-18-of-u-s-users-store-secret-passwords-on-their-smartphone.aspx

[9] S. Gaw and E. W. Felten, "Password management strategies for online accounts," in *Proceedings of the second symposium on Usable privacy and security - SOUPS '06*. New York, New York, USA: ACM Press, 2006, pp. 44 – 55. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1143120.1143127

[10] D. McCarney, D. Barrera, J. Clark, S. Chiasson, and P. C. van Oorschot, "Tapas: design, implementation, and usability evaluation of a password manager," in *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12*. New York, New York, USA: ACM Press, 2012, pp. 89–98. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2420950.2420964

[11] J. A. Halderman, B. Waters, and E. W. Felten, "A convenient method for securely managing passwords," in *Proceedings of the 14th international conference on World Wide Web - WWW '05*. New York, New York, USA: ACM Press, 2005, pp. 471–479. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1060745.1060815

[12] A. Huth, M. Orlando, and L. Pesante, "Password Security, Protection, and Management," Carnegie Mellon University. Produced for US-CERT, a government organization., Tech. Rep., 2012.

[13] J. Mailley, R. Garcia, S. Whitehead, and G. Farrell, "Phone Theft Index," *Security Journal*, vol. 21, no. 3, pp. 212–227, Jul. 2008. [Online]. Available: http://www.palgrave-journals.com/doifinder/10.1057/palgrave.sj.8350055

[14] B. Kaliski, "RFC 2898: PKCS #5: Password-Based Cryptography Specification (Version 2.0)," RSA Laboratories, Tech. Rep., 2000. [Online]. Available: http://tools.ietf.org/html/rfc2898

[15] R. Zhao and C. Yue, "All your browser-saved passwords could belong to us," in *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY '13*. New York, New York, USA: ACM Press, 2013, p. 333. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2435349.2435397

[16] C. Herley and P. Van Oorschot, "A Research Agenda Acknowledging the Persistence of Passwords," *IEEE Security & Privacy Magazine*, vol. 10, no. 1, pp. 28–36, Jan. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6035662

[17] R. Biddle, S. Chiasson, and P. Van Oorschot, "Graphical passwords," *ACM Computing Surveys*, vol. 44, no. 4, pp. 1–41, Aug. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2333112.2333114

[18] M. Fukumitsu, T. Katoh, B. B. Bista, and T. Takata, "A Proposal of an Associating Image-Based Password Creating Method and a Development of a Password Creating Support System," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 2010, pp. 438–445. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5474736

[19] F. Schaub, M. Walch, B. Könings, and M. Weber, "Exploring the design space of graphical passwords on smartphones," in *Proceedings of the Ninth Symposium on Usable Privacy and Security - SOUPS '13*. New York, New York, USA: ACM Press, 2013, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2501604.2501615

[20] M. Shakir and A. A. Khan, "S3TFPAS: Scalable shoulder surfing resistant textual-formula base password authentication system," in *2010 3rd International Conference on Computer Science and Information Technology*. IEEE, Jul. 2010, pp. 12–14. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5564479

[21] F. Stajano, "Pico: no more passwords!" in *SP'11 Proceedings of the 19th international conference on Security Protocols*. Springer-Verlag Berlin, Heidelberg, 2011, pp. 49–81.

[22] H. Krawczyk, M. Bellare, and R. Canetti, "RFC 2104: HMAC: Keyed-Hashing for Message Authentication," 1997. [Online]. Available: http://tools.ietf.org/html/rfc2104

[23] M. S. Turan, E. B. Barker, W. E. Burr, and L. Chen, "Sp 800-132. recommendation for password-based key derivation: Part 1: Storage applications," Gaithersburg, MD, United States, Tech. Rep., 2010.

[24] C. Percival, “Stronger key derivation via sequential memory-hard functions,” Tech. Rep., 2009. [Online]. Available: http://www.tarsnap.com/scrypt/scrypt.pdf

[25] ——, “The scrypt Password-Based Key Derivation Function,” 2012. [Online]. Available: http://tools.ietf.org/html/draft-josefsson-scrypt-kdf-01

[26] Legion of the Bouncy Castle Inc., “The Legion of the Bouncy Castle,” 2013. [Online]. Available: http://www.bouncycastle.org

[27] T. Johns, “Using Cryptography to Store Credentials Safely,” 2013. [Online]. Available: http://android-developers.blogspot.co.at/2013/02/using-cryptography-to-store-credentials.html

[28] B. K. Marshall, “PasswordResearch.com,” 2013. [Online]. Available: http://www.passwordresearch.com

[29] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez, “Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms,” in *2012 IEEE Symposium on Security and Privacy*. IEEE, May 2012, pp. 523–537. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6234434

[30] M. S. Turan, E. B. Barker, W. E. Burr, and L. Chen, “Sp 800-132. recommendation for password-based key derivation: Part 1: Storage applications,” Gaithersburg, MD, United States, Tech. Rep., 2010.