SECRET SHARING SCHEME FOR CREATING MULTIPLE SECURE STORAGE DIMENSIONS FOR MOBILE APPLICATIONS

Michał Glet, Kamil Kaczyński

Military University of Technology, Faculty of Cybernetics,
Institute of Mathematics and Cryptology
e-mails: michal.glet@wat.edu.pl, kamil.kaczynski@wat.edu.pl
Poland

Abstract: Mobile devices store large amounts of their users' private data. Applications such as instant messengers, activity recorders or financial management applications are a particularly important source of sensitive data. The security features implemented by the mainstream mobile device manufacturers in many cases turn out to be insufficient, exposing users to data loss, e.g. in the event of the loss or theft of a device. This paper proposes a method using secret sharing scheme for protecting data stored on the device, especially in common containers such as databases in the way allowing creation of multiple virtual safe containers, thus giving app developers a tool to protect the data of any application. The paper also addresses the specifics of adapting mechanisms to take advantage of the special features of Android operating system. The benefits of its use as well as restrictions that are imposed on the end user are presented. A solution security analysis has also been presented, with particular emphasis on the level of cryptographic strength offered by the proposed mechanism.

Key words: secret sharing, cryptography, cybersecurity, mobile security, keystore

1. INTRODUCTION

1.1. Background

According to [1], there are almost 3.5 billion smartphone users worldwide. Each one of them uses at least several apps that store some sensitive data. Apps which store and process healthcare, financial and personal (e.g. instant messengers) data are especially sensitive and should be protected in a secure and efficient manner. In [2], the authors state that in the nearly 40% of Italy’s divorce cases WhatsApp messages are cited as evidence against unfaithful partners. The above indicates that mechanisms for protecting personal data in WhatsApp is not secure enough, thus making it easy to recover all the conversation history.

Even privacy-focused apps, such as Signal, do not protect their users' data efficiently. In [3] and [4], the authors described security flaws of respectively Android and iOS versions of the app and recovered the complete database contents. An attack was possible because Signal relies only on the operating system mechanisms for securing cryptographic keys, e.g. Android Keystore. There have been many papers describing methods of breaking into the Keystore. In [5], the authors show impact of using non-provably secure cryptographic schemes in complex Keystore architecture and its consequences. Especially, the authors have showed that the AE scheme used did not satisfy integrity and unforgettableness of cipher text, which lead to reduction of the length of symmetric keys.

Apart from the vulnerabilities, the OS-integrated mechanism is strictly dependent on the device master key which is derived from the user secret (lock pattern, password, PIN, biometrics). The process of revealing the user secret may be performed using forensic tools such as Cellebrite UFED Ultimate, thus making the process of recovering application data easy to perform. In [6], the authors analyse different secure key storage solutions, but none of them guarantees a sufficient security level for the secrets stored. Most of them meet only two out of three security requirements – app-binding, device-binding and user-consent required. App-binding means that the key can only be used by an instance of a certain application on a certain device. Device-binding means that the key can only be used on a certain device. The last but not the least is the requirement for user-consent. The key may be used only when the user wants to use the key and has given their explicit consent to do so. In this paper, we propose a mechanism for data at rest protection, which meets user-consent requirement and, combined with other techniques, may allow app-binding and device-binding of secret key protecting application secrets.

1.2. Our contribution

There is a lack of universal data protection mechanisms that may be easily adopted by mobile app developers. Existing solutions are rather OS-specific and often dependent on its current version – such as Keystore API in Android [7]. Our goal for this paper was to provide an easy-to-implement, universal method of protecting multiple app secrets stored in common container using key derivation functions (KDF) and a secret-sharing scheme. The approach presented is totally independent of the operating system being used and the hardware installed, thus it may be deployed even on older devices.

Based on [8], we created a generic mechanism which utilizes user-provided password for creating secure storage of app sensitive data. Our solution may be also used in other situations, e.g. for limiting access to several functionalities or data sets in a way which uses cryptography instead of app logic. The architecture proposed evaluates the user password, ensuring its high entropy, which is crucial for the overall system security. We have chosen symmetric encryption schemes that allowed us to improve overall cryptographic strength of the solution.

What is more, a proposed mechanism is a unique and different from other approaches, e.g. like described in [9] or [10]. Crucial part of our mechanism is a possibility of creation of many dimensions of secret data. From the user's perspective, the dimension is acting like a separate database containing some data. In every dimension user can store other data. Access to the data from one dimension is given only after providing the correct passcode securing selected dimension. Different dimensions are secured with a different passcodes.

2. PROTECTING DATA AT REST

2.1. Generic approach

Mobile applications store their data in SQLite databases or in files placed in the dedicated area of the device memory. Usually, this kind of data is stored in the dedicated area in the device memory. Unfortunately, if commands can be executed with superuser privileges or complete device data can be extracted, then all of the application files are available to the attacker. If there is no extra layer of protection – all the data stored in that files are compromised. Thus, it is necessary to provide a mechanism which will protect application data in a way that access to its content will need user consent. Such mechanisms should meet the following requirements:

- There must be user consent for accessing the protected data;
- The mechanism should be independent from the OS version of the device;
- The user has to provide a secret (e.g. passcode) to access the data;
- The mechanism should have a possibility to integrate with other OS-specific mechanisms;
- Security of the mechanism must rely on cryptography.

We propose generic mechanisms based on PBES2 and utilizing user-provided password for encryption and decryption of the application data. In general, the process of protecting application data requires encryption of the sensitive data using a key derived from the user-provided secret. All of the data stored on the device are integrity-protected and encrypted. When an application wants to access the data, it has to perform the reverse operation. The decryption needs a key derived from the user-provided secret. If the key provided is invalid, the decryption process will result in an error, thus the application will not be able to read the data and will not process the request. If the application is using a database file, then it may be externally or internally encrypted. In the event that the application is using data stored in another type of file, e.g. SharedPreferences XML file, then the encryption is applied only internally to the specific values.

The mechanisms proposed may be used for protecting access to the application if the protected data is crucial for its operation, e.g. database containing all the messages of an instant messaging app (Fig. 1). It can be also used for protecting access only to part of the application data, e.g. history of user activities in an activity tracking app. During the implementation of a generic mechanism in an application, it is crucial to follow the guidelines from [9].

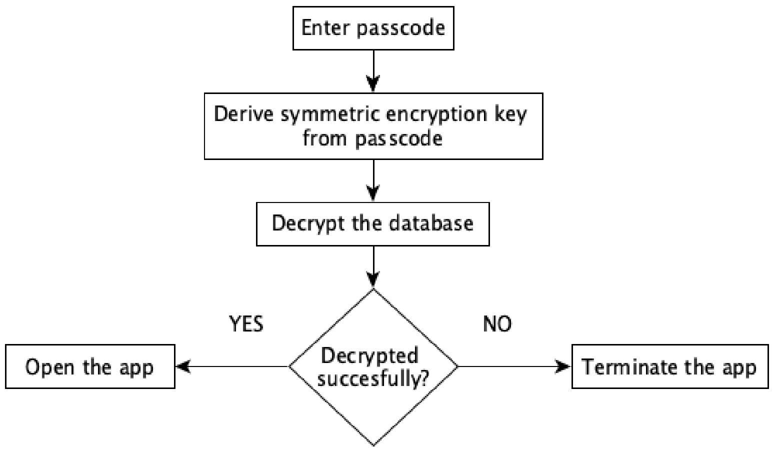

Fig. 1. Protecting access to the application using the proposed mechanism

Particularly, the implementation should meet usability requirements for memorized secrets, such as: limiting the maximum number of allowed attempts, providing clear and meaningful feedback about errors and remaining allowed attempts and support for delayed masking during text entry. When using it in a mobile environment, mechanisms for re-authenticating users must be implemented due to their inactivity and after a fixed period of time. The user must have a possibility for creating and changing memorized secrets. The application should inform the user about entropy of the password provided, thus giving him an opportunity to choose a sufficiently strong and easy to remember secret.

2.2. Android OS specific mechanisms

The Android keystore system [10] provides developers with a possibility of storing cryptographic keys in a container, thus making its extraction more difficult. Cryptographic operations can be executed with the key material which is non-exportable. The use of the Android keystore offers several advantages. First, it enables extraction prevention of key material by assuring that the key never enters the application process. Applications perform cryptographic operations using the key stored in the Android keystore, but the application process only uses plaintext or ciphertext in communication with the system process responsible for cryptographic operations. Thus, compromising the application process is not enough to compromise the app keys, because they are securely stored in keystore container. Secondly, the Android keystore system enables bounding key material to the secured hardware, e.g. Trusted Execution Environment. The Android keystore system allows the authorization of key material use. There are three categories of supported key use authorization:

- Cryptography – authorized key algorithm, operations and purpose, block modes, digests etc. which the key can be used for;
- Temporal validity interval – time during which the key is authorized for use;
- User authentication – the key can only be used if user authentication is recent enough.

Starting from API level 23, the Android operating system supports encryption using symmetric algorithms – especially AES in CBC, CTR, ECB and GCM mode. While the keystore is a great solution for protecting application key material, it also has one big disadvantage. When someone is able to pass through your screen lock (PIN, password, pattern, biometrics), they are considered an authenticated user and are capable of conducting all the cryptographic operations using the stored key material for a specific app. The same applies if someone can bypass the TEE security. In [11], the authors present main vulnerabilities to the TEE systems that may allow leakage of keying material.

The main drawback of using the Android keystore in an application is that there is no need for the intended user consent to decrypt the data, once the attacker is able to unlock the user's phone, then he can perform actions on his behalf. This problem can be resolved by using the mechanism proposed in 2.1. for generating the keying material together with the Android keystore. For instance, when an application uses the SQLCipher database, it needs to be initiated with a proper database secret – encryption key – each time the application is being opened. Such a database secret can be generated using the steps described in next sections.

The approach ensures that there is no possibility of decrypting the SQLCipher database without knowing the user memorized secret. In that way, an attacker cannot access app secrets without acquiring the key generated with user secret. Thus, together with the Android keystore, the mechanism proposed provides security features ensuring the binding of keying material to the app and device altogether with ensuring that user consent for decrypting data exists.

3. SECRET CHATS PROTOCOL

The idea of secret chats is very simple – enable users, by entering different passcodes, to see different data in the application. It can be e.g. different conversations, different contacts and other data that the user wants to keep secret in some circumstances. What is also important, the proposed Secret Chats Protocol allows the hiding of information in many dimensions – one user passcode gives access to only one dimension. Moreover, the protocol proposed hides the actual total number of dimensions used by a user, revealing only the boundary configured by the application developer.

Securing information in the Secret Chats Protocol is based on common idea of using SQLCipher to keep SQLite database in an encrypted form. Because of this, Secret Chats Protocol can be quite easily and with low cost integrated within existing application. Moreover, the Secret Chats Protocol can be effectively used also in environments which use other databases and database encryption methods – the Secret Chats Protocol proposed shows how to retrieve the database encryption key and chat information encryption key.

In the next sections, the following definition will be used:

- *Database Encryption Key* (DEK) – a common secret for all user's passcodes. It is mainly intended to be used as a symmetric key for decrypting and encrypting database e.g. with the SQLCipher mechanism. The Secret Chats Protocol uses only one database to store all information and that is why database encryption key is common for all secret chats (all dimensions). DEK is intended to be used by database encryption mechanisms.
- *Information Encryption Key* (IEK) – a symmetric key for decrypting and encrypting secret chat information – to retrieve only one dimension of secret information. This key is different for different passcodes and for different secret information dimensions. IEK is intended to be used internally by application e.g. to decrypt conversation from secret information dimension connected with provided passcode.
- *One dimension of secret information* – secret data stored in the application that is protected by user passcode. Access to this data is possible only after the correct retrieval of DEK and IEK.

3.1. The idea of the DEK

The idea of the database encryption key (DEK) is very simple and is widely used in many nowadays applications to store data in encrypted database. DEK is a secret data that contains key and other data necessary to retrieve encryption key and other secret parameters necessary to decrypt any data in an encrypted database. In many cases [3, 4] DEK is stored using OS specific security mechanisms, eg. in OS (mobile operation system) keystore. Access to the DEK data is given after first user authentication in the OS (eg. after first unlocking mobile device after reboot). In such case, security of DEK depends only on the security of mobile operating system, which for applications storing sensitive data should not be enough. After retrieval, DEK is used by Database Management System (DBMS) or some database engine extension (like popular SQLCiper solution) to decrypt the database. From now on the database is fully accessible for an application.

Secret Chats Protocol also utilise idea of the database encryption key (DEK) – all data used by an application is stored in the database encrypted with a DEK by internal mechanisms of DBMS or by the database engine extension like SQLCipher. Someone could ask why not use separate database files? It is all because of the secrecy. One of the purposes of using the Secret Chats Protocol is to hide total number of secret dimensions – no one could know how many secret dimensions are stored in an application. Using different databases for different dimensions will immediately give information how many dimensions are defined by the user. In such a case an attacker could force user to give him all passcodes to all dimensions. In case of the Secret Chats Protocol attacker do not know how many dimensions are and there is no simple way to know it.

3.2. The idea of the IEK

The idea of the information encryption key (IEK) is also very simple but it is not so widely used as the idea of DEK. This is mainly because of the ease of use – the idea of DEK assumes that all operations associated with database encryption and decryption is performed outside the application by a DBMS or a database engine extension. The application uses data in the same way as from unencrypted database – no special changes are required. In the opposition to this the idea of IEK requires that all encryption and decryption operations are performed directly by the application. Data is decrypted after retrieving it from the database, data is encrypted before storing it in the database. To encrypt and to decrypt data, the application use information encryption key (IEK) data. IEK is a secret data that contains key and other data necessary to retrieve encryption/decryption key and other secret parameters necessary to encrypt/decrypt data stored/retrieved from database. Main drawback of the IEK idea is a lack of database support in data manipulation. When application encrypt data stored in the database, database engine cannot e.g. perform effectively select queries – data is encrypted and restrictions in a SQL WHERE clause would not work. That is why the application that follow the IEK idea should be properly designed to work well. On the other hand one of the main pros of the IEK idea is the fact that sensitive key data does not leave application environment – developer do not need to trust third party systems like DBMS. It is quite likely that an application which follows the idea of the IEK, just like in the case of the DEK idea, will store data using a OS specific security mechanisms, eg. in OS keystore. Access to IEK data would be given after first user authentication in the OS (eg. after first unlocking mobile device after restart). In such cases security of IEK depends only on the security of a mobile operating system, which for applications storing sensitive data should not be enough. After retrieval, IEK is used by the application to encrypt and decrypt the data.

3.3. Idea of DEK and IEK in the Secrets Chats Protocol

The Secrets Chats Protocol incorporates both ideas – the DEK and IEK – and tries to emphasise their pros minimizing their cons. First of all, DEK and IEK data is stored in encapsulated form connected with the user passcode. DEK data is common for all dimensions of secret information – there is only one database for all dimensions of secret information. In the described above idea DEK data is used to retrieve key and other data required by DBMS encryption mechanism or database engine extension. IEK data is tightly connected with only one dimension of secret information. Different dimensions have different IEK data. In new applications data necessary to retrieve DEK and IEKs can be stored in OS keystore. Existing applications can store DEK and IEKs data in exactly this same way as currently store sensitive data. In both cases DEK and IEK will be saved in encapsulated form, which make retrieving infeasible without knowing passcode. Because of this security of DEK and IEK storage mechanism is not crucial. Every security dimension is secured by other passcode. Every passcode allows to retrieve this same DEK data and different IEK data. DEK data is used to decrypt/encrypt database by database mechanisms. IEK data is used to decrypt data read from the database and encrypt data send to the database. Only data connected with IEK's dimension of security information can be decrypted/encrypted with IEK data. For data from other dimensions decryption process will fail.

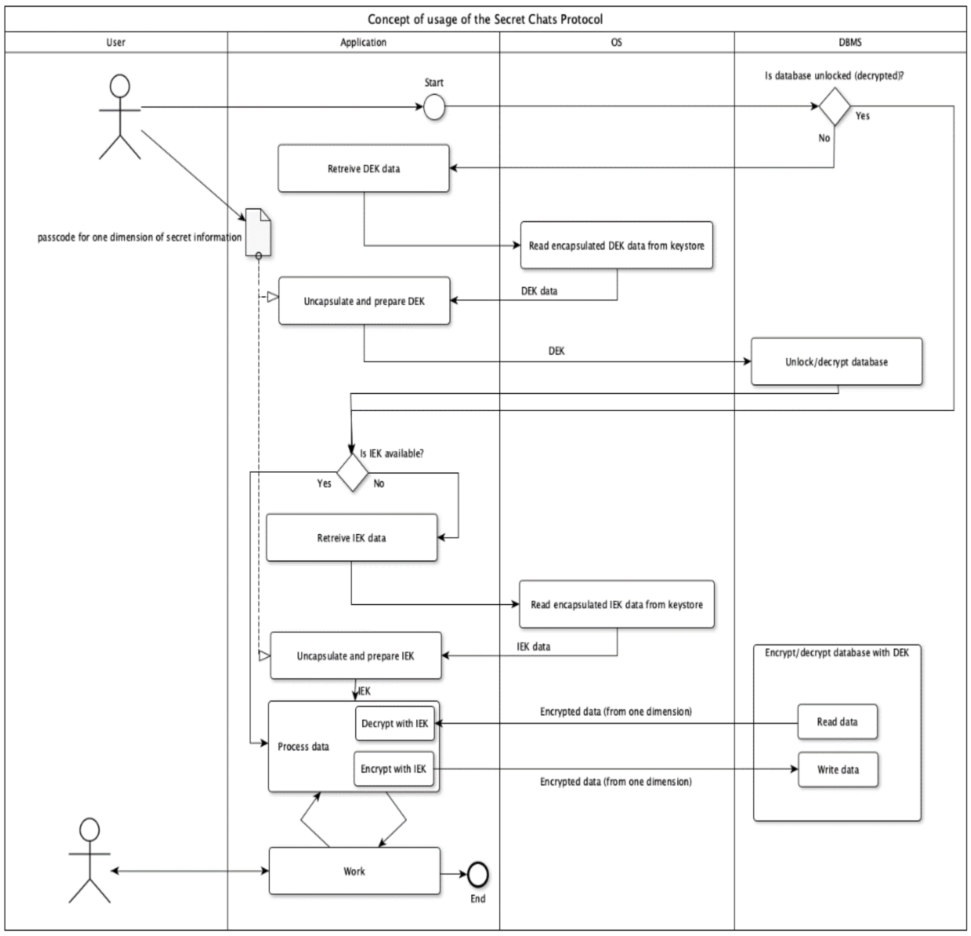
Concept of usage of the Secret Chats Protocol
User
Application
OS
DBMS
Start
Is database unlocked (decrypted)?
Yes
No
Retreive DEK data
passcode for one dimension of secret information
Read encapsulated DEK data from keystore
DEK data
Uncapsulate and prepare DEK
DEK
Unlock/decrypt database
Is IEK available?
Yes
No
Retreive IEK data
Read encapsulated IEK data from keystore
IEK data
Uncapsulate and prepare IEK
IEK
Process data
Decrypt with IEK
Encrypt with IEK
Encrypted data (from one dimension)
Encrypted data (from one dimension)
Encrypt/decrypt database with DEK
Read data
Write data
Work
End

Fig. 2. Basic idea of the Secret Chats Protocol.

4. SECRET CHATS WITH SHAMIR'S SS and CCA

4.1. Shamir's Secret Sharing with $(2, k)$ threshold

The Secret Chats Protocol is highly based on Shamir's Secret Sharing [11] with $(2, k)$ threshold scheme. It means that common secret, which is DEK data, is divided into k pieces in such way that:

1. knowledge of any 2 or more pieces allows to easy recover DEK data;
2. knowledge of 1 or 0 pieces makes recover of DEK data infeasible in terms of computational resources needed for this task (e.g. computational power, computations time) and as stated in [11], in this case all possible values of DEK data are equally likely;

Shamir's Secret Sharing scheme is based on polynomial interpolation and uses the fact, that having any 2 points defined in the 2-dimensional plane $(x_1, y_1), (x_2, y_2)$, where $x_1 \neq x_2$, one can construct one and only one polynomial $f(x)$ of degree 1 such that $f(x_1) = y_1$ and $f(x_2) = y_2$. Choosing at random polynomial $f(x) = (a_0 + a \cdot x)(mod\, p)$, where p is a prime number, and hiding value of DEK data in coefficient $a_0 = \tau(\text{DEK})$ Secret Chats Protocol divides secret into k parts by computing $(x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_{k-1}, f(x_{k-1})), (x_k, f(x_k))$. Having any 2 or more parts one can recover polynomial $f(x)$ and value of a_0, which gives the value of the DEK data.

Shamir's Secret Sharing scheme is a mechanism with a lot of potential areas of use. However, its construction gives secret shares in form of values that creates a point – this approach is not quite user friendly and directly does not allow user to select his own secrecy (e.g. passcode) that he can easily memorize. The Secret Chats Protocol deals with this problem– it uses Shamir's Secret Sharing $(2, k)$ threshold scheme, allows users to use their owns passcodes and does not require users to memorize any other data. In Secret Chats Protocol Shamir's shares (points) are recovered using passcode given by a user.

In the Secret Chats Protocol one share is always stored in the application data. Other shares are connected with dimensions of secret information. Every dimension has its own secret share. The application that stores m dimensions of secret information uses $m + 1$ secret shares – one for every dimension and one for the application. Having secret share computed from user passcode and its own share, application is able to retrieve DEK data.

4.2. DEK

As stated above, the database encryption key (DEK) is common for all dimensions of secret information. The DEK retrieval process uses Shamir's Secret Sharing $(2, k)$ threshold scheme, denoted as $SSS(2, k)$, and the Common Container Algorithm (CCA). The DEK data is a secret hidden with usage of $SSS(2, k)$. The Common Container Algorithm was invented to:

1. allow users to use their own passcodes in $SSS(2, k)$ schema;

2. keep parts of secret shares from the $SSS(2,k)$ schema in one place (one storage container);
3. be stored by the application;
4. easy allow users to add new dimensions of secret information – e.g. without need of re-computation of previously created secret shares.

Below is described an algorithm for Shamir's Secret Sharing $(2,k)$ threshold scheme, that the Secret Chats Protocol uses to hide the value of DEK.

Let DEK contain n-bits of data (e.g. n-bits encryption key). Let $\tau(\mathrm{DEK})$ be a positive integer representation of DEK.:

$$\begin{gathered}\mathrm{DEK} = (\mathrm{b}_0, \mathrm{b}_1, \mathrm{b}_2, \mathrm{b}_3, \dots, \mathrm{b}_{\mathrm{n}-1});\ \mathrm{b}_\mathrm{i} \in \{0,1\};\ \mathrm{i} = \overline{0, \mathrm{n}-1} \\ \tau(\mathrm{DEK}) = 2^0 \cdot \mathrm{b}_0 + 2^1 \cdot \mathrm{b}_1 + 2^2 \cdot \mathrm{b}_2 + 2^3 \cdot \mathrm{b}_3 + \cdots + 2^{\mathrm{n}-1} \cdot \mathrm{b}_{\mathrm{n}-1}\end{gathered} \quad (1)$$

Let P be a set containing all cryptographically secure prime numbers (e.g. satisfying requirements from [12] or from appendix B.3 in [13]), $p \in \mathrm{P}$ and $\tau(\mathrm{DEK}) < p$. Let Z_p^* be the multiplicative group modulo p. Let $\mathrm{a} \in Z_p^*$ be chosen randomly and let polynomial $f(x)$ of degree 1 over Z_p^* be constructed:

$$f(x) = (\tau(\mathrm{DEK}) + a \cdot x)(mod\ p) \quad (2)$$

According to the Shamir's Secret Sharing (2, k) threshold schema, 2 secret shares are necessary to rebuild polynomial of 1 degree. That is why the Secret Chats Protocol needs two different shares to reconstruct polynomial $f(x)$. Having polynomial $f(x)$, the database encryption key (DEK) can be retrieved:

$$f(0) = \tau(\mathrm{DEK}) \quad (3)$$

Having $\tau(\mathrm{DEK})$ application can easily reconstruct n-bits of DEK data (e.g. n-bits encryption key). Let $\left\lfloor \frac{a}{b} \right\rfloor$ be an integer part of division a by b. Then:

$$\begin{gathered}DEK = (b_0, b_1, b_2, b_3, \dots, b_{n-1}); \\ b_\mathrm{i} = \left\lfloor \frac{\tau(\mathrm{DEK})}{2^i} \right\rfloor (mod\ 2);\ i = \overline{0, n-1}\end{gathered} \quad (4)$$

4.3. IEK

As stated above, the information encryption key (IEK) is distinct for all dimensions of secret information – every dimension of secret information has different value of IEK data. Processes of creation and retrieval of IEK are describe as a part of algorithms presented in the next two sections.

4.4. Creating a new dimension of secret information

The procedure of creating a new dimension of secret information consists of the following steps:

The user chooses a unique $passcode$ for a new secret information dimension. The selected $passcode$ is expanded (or trimmed) to n-bits value by evaluating the function:

$$PASSCODE = expand(passcode) \quad (5)$$

As an *expand* function applications can use for example cryptographic hash function with *n*-bits output.

1. PASSCODE is converted to positive integer by computing the value of τ(PASSCODE) as stated in (1).
2. For the value of τ(PASSCODE), the value of $p_{passcode} \in \mathrm{P}$ is computed as the nearest prime number from P greater than or equal to τ(PASSCODE):

$$\begin{gathered} \tau(\mathrm{PASSCODE}) \leq p_{passcode} \\ \nexists p_k \in \mathrm{P}: \mathrm{p_k} \neq p_{passcode}\ AND\ \tau(\mathrm{PASSCODE}) \leq p_k \leq p_{passcode} \end{gathered} \tag{6}$$

3. Using formula (2), the new secret share is computed:

$$f\left(p_{passcode}\right) = \left(\tau(\mathrm{DEK}) + a \cdot p_{passcode}\right)(\mathrm{mod}\ p) \tag{7}$$

Point $(p_{passcode}, f\left(p_{passcode}\right))$ is a secret share for a new secret information dimension.

4. The value of the information encryption key (IEK) is computed by evaluating the Key Derivation Function [8] as follows:

$$IEK = KDF(PASSCODE) \tag{8}$$

Secret share needed by the $SSS(2, k)$ algorithm consist of two values – the value of $p_{passcode}$ which is computed from the user passcode and the value of $f(p_{passcode})$ (7). In classical approach the value of $f(p_{passcode})$ would be presented to the user and an application would ask the user to enter this value during the IEK and DEK retrieval process. However, this would be very infeasible and very problematic for the user – the value of $f(p_{passcode})$ would mainly be a very large positive integer. The proposed in this paper Secret Chats Protocol deals with this problem using the Common Container Algorithm (CCA). The next steps of the procedure of creating a new dimension are:

5. The value $m_{passcode}$ is computed as follows:

$$m_{passcode} = m \cdot p_{passcode} \tag{9}$$

where m is the value of multiplication of values (6) computed during process creation of other dimensions of secret information. For the first secret dimension $m = 1$.

6. The value $w_{passcode}$ is computed as follows:

$$\begin{gathered} A \equiv \frac{1}{f\left(p_{passcode}\right)}\ (\mathrm{mod}\ m) \\ w_{passcode} = \left(w \cdot f\left(p_{passcode}\right) \cdot A\right)\left(mod\ m_{passcode}\right) \end{gathered} \tag{10}$$

where w is a value computed during process creation of previous dimensions of secret information. For the first secret dimension $w = 1$.

The computation of A is not feasible when the greatest common divisor (GCD) [19] of $f\left(p_{passcode}\right)$ and m is not equal to 1. If this situation occurs, the application should ask the user to enter another passcode. However, as stated in next sections, the probability that such condition occurs goes asymptotically to 0.

7. The value of $r_{passcode}$ is computed:

$$\begin{gathered} B \equiv \frac{1}{A} \left(\bmod p_{passcode}\right) \\ C \equiv \frac{1}{w} \left(\bmod p_{passcode}\right); \\ r_{passcode} = B \cdot C \end{gathered} \tag{11}$$

The computation of B (or C) is not feasible when the greatest common divisor (GCD) [19] of A (or w) and $p_{passcode}$ is not equal to 1. However, $p_{passcode}$ is a prime number, so GCD of A (or w) and $p_{passcode}$ will not equal to 1 only in case when A (or w) is a multiplication of $p_{passcode}$. If this situation occurs, the application should ask the user to enter another passcode. However, as stated in next sections, the probability that such condition occurs goes asymptotically to 0.

8. The application should store new values of m, w and a value of $r_{passcode}$

$$\begin{gathered} m = m_{passcode} m = m_{passcode} \\ w = w_{passcode} \end{gathered} \tag{12}$$

4.5. Retrieving DEK and IEK from the passcode

The procedure of retrieving DEK and IEK for one dimension of secret information consists of the following steps:

1. The user enters a $passcode$ for one dimension of secret information. The given $passcode$ is expanded (or trimmed) to n-bits value by evaluating the function:

$$PASSCODE = expand(passcode) \tag{13}$$

2. PASSCODE is converted to positive integer by computing the value of τ(PASSCODE) as stated in (1).
3. For the value of τ(PASSCODE), the value of $p_{passcode} \in$ P is computed as the nearest prime number from P greater than or equal to τ(PASSCODE):

$$\begin{gathered} \tau(\text{PASSCODE}) \leq p_{passcode} \\ \nexists p \in \text{P: p} \neq \; p_{passcode} \; AND \; \tau(\text{PASSCODE}) \leq p \leq p_{passcode} \end{gathered} \tag{14}$$

4. The values of $r_{passcode}, m, w$ are retrieved from the application.
5. The value of $f(p_{passcode})$ is computed as follows:

$$\begin{gathered} v = \left(w \cdot r_{passcode}\right) (\bmod m) \\ f(p_{passcode}) = (v)\left(\bmod p_{passcode}\right) \end{gathered} \tag{15}$$

6. Point $P_1 = (p_{passcode}, f(p_{passcode}))$ is a computed secret share for the secret information dimension. Point $P_2 = (p_{app}, f(p_{app}))$ is a secret share stored by an application. Having points P_1 and P_2, the polynomial $f(x)$ is reconstructed using interpolation techniques like Lagrange Interpolating Polynomial [14]. The value of $f(0) = \tau(\text{DEK})$ is computed, and DEK data is retrieved.

7. The value of the information encryption key (IEK) is computed by evaluating the Key Derivation Function [8] as follow:

$$IEK = KDF(PASSCODE) \tag{16}$$

4.6. Why does it work?

The Secrets Chats Protocol uses the Common Container Algorithm to store (in hidden form) data that is used in Shamir's Secret Sharing scheme to reconstruct DEK data. All calculations are performed within multiplicative groups of integers modulo n, denoted as Z_n^*. Z_n^* is in fact a finite abelian group and is a multiplicative group of the ring of integers modulo n, denoted as Z/nZ. According to theorem 2 in [15], for Z/nZ, where n is a composite number $n = p_1 \cdot p_2 \cdot \ldots \cdot p_i$, where $p_1, p_2, \ldots, p_i$ are distinct prime numbers, there exists unique ring isomorphism between ring Z/nZ and direct product of rings of integers modulo prime factors of n.

$$\frac{Z}{nZ} \cong \frac{Z}{p_1Z} \times \frac{Z}{p_2Z} \times \ldots \times \frac{Z}{p_iZ} \tag{17}$$

Because of this doing arithmetic operations in Z/nZ (like multiplications in Z_n^*) we indirectly also do these same operations in each Z/p_jZ (multiplications in $Z_{p_j}^*$)) for $j = \overline{1, \iota}$. In other words, it means that having multiplication result in Z_n^* to get result from $Z_{p_j}^*$ we just need compute result modulo p_j.

$$\forall a, b \in Z_n^*; (ab)\left(\operatorname{mod} p_j\right) \equiv \left((ab)(\operatorname{mod} n)\right)\left(\operatorname{mod} p_j\right) \tag{18}$$

Proof: Lets $a, b \in Z_n^*,\ n = p_1 \cdot p_2 \cdot \ldots \cdot p_j \cdot \ldots \cdot p_i$. From left side we have $(ab)\left(\operatorname{mod} p_j\right) = L$ which means that $\exists h_j \in Z\ ab = h_j \cdot p_j + L$. From right side we have $(ab)(\operatorname{mod} n) = R$ which means that $\exists h \in Z\ ab = h \cdot n + R = h \cdot p_1 \cdot p_2 \cdot \ldots \cdot p_j \cdot \ldots \cdot p_i + R$ and $\left((ab)(\operatorname{mod} n)\right)\left(\operatorname{mod} p_j\right) = (h \cdot n + R)\left(\operatorname{mod} p_j\right) = \left(h \cdot p_1 \cdot p_2 \cdot \ldots \cdot p_j \cdot \ldots \cdot p_i + R\right)\left(\operatorname{mod} p_j\right) = R\ (\operatorname{mod} p_j)$. Going back to equation we have $h_j \cdot p_j + L = h \cdot p_1 \cdot p_2 \cdot \ldots \cdot p_j \cdot \ldots \cdot p_i + R$. Computing it modulo p_j we get $L \equiv R\left(\operatorname{mod} p_j\right)$ which ends our proof.

This property is highly used in the Secrets Chats Protocol calculations – calculations are done within multiplicative groups of integers modulo composite n, but to get any useful information (e.g. part of DEK secret share), one must know at least one prime factor of n. This prime number are created from passcode that is provided by a user.

In the Secrets Chats Protocol every dimension of a secret information has its own share, which is a point $\left(p_{passcode}, f\left(p_{passcode}\right)\right)$. Value of $p_{passcode}$ is calculated directly from passcode provided by a user (6), value of $f\left(p_{passcode}\right)$ is retrieved in the Common Container Algorithm using $p_{passcode}$ and data stored in the application. Application stores all values of $f(p_{passcode})$ hidden in the value of w. Value of $r_{passcode}$ and m are used to remove any other data (noise) not directly connected with $p_{passcode}$ and $f(p_{passcode})$ – data from other security dimensions. During creation process new value of w is constructed as in (10) using an old value of w, a value of $f(p_{passcode})$ and a value of special mask A.

$$w_{passcode} = (w \cdot f(p_{passcode}) \cdot A)(\operatorname{mod} m_{passcode}) \tag{19}$$

Mask value A is constructed as in (10) and is used to ensure that all previously stored values of $f(p_{passcode})$ from other dimensions will be still recoverable. This works because A is a just modular inversion of $f(p_{passcode})$ modulo previous value of m.

$$A \equiv \frac{1}{f(p_{passcode})} \ (\operatorname{mod} m) \tag{20}$$

That is why a new value of w equals previous value of w modulo previous value of m:

$$w_{passcode} = \left(w \cdot f(p_{passcode}) \cdot \frac{1}{f(p_{passcode})}\right)(\operatorname{mod} m) \equiv w \ (\operatorname{mod} m) \tag{21}$$

To compute $f(p_{passcode})$ application must use stored values m, w (12), $r_{passcode}$ (11) and a computed from users passcode value $p_{passcode}$. Then (some modulo operations omitted in terms of better readability):

$$\begin{gathered} f(p_{passcode}) = \big((w \cdot r_{passcode}) \ (\operatorname{mod} m)\big)(\operatorname{mod} p_{passcode}) = \\ \left(\left(w' \cdot f(p_{passcode}) \cdot A \cdot \frac{1}{A} \cdot \frac{1}{w'}\right) \ (\operatorname{mod} m)\right)(\operatorname{mod} p_{passcode}) = \\ \big(\big(f(p_{passcode})\big) \ (\operatorname{mod} m)\big)(\operatorname{mod} p_{passcode}) \equiv f(p_{passcode}) \ (\operatorname{mod} p_{passcode}) \end{gathered} \tag{22}$$

5. SECURITY

The Secret Chats Protocol is using Shamir's Secret Sharing $(2, k)$ threshold scheme, large prime numbers, arithmetic operations in multiplicative groups of integers modulo positive integer n. This section focuses on security aspects of this primitives and provides a kind of starting point for the discussion about the protocol and ways to make it better.

5.1. Shamir's Secret Sharing

Shamir's Secret Sharing $(\mathbf{2}, \boldsymbol{k})$ threshold scheme is using a 1-degree polynomial over finite field modulo prime number p.

$$f(x) = (\tau(\mathrm{DEK}) + a \cdot x)(\operatorname{mod} p) \tag{23}$$

Lemma 1: Probability of finding value of secret $\boldsymbol{\tau}(\mathbf{DEK})$ having less then 2 shares is equal to $\frac{\mathbf{1}}{\mathbf{p}}$.

Proof: Having two or more shares one can use interpolation techniques (e.g. Lagrange polynomial) to reconstruct $\boldsymbol{f(x)}$ and to compute secret $\boldsymbol{f(0)} = \boldsymbol{\tau}(\mathbf{DEK})$. Let now assume that an attacker has access to one piece of secret share – point $(\boldsymbol{x_a}, \boldsymbol{y_a})$. Value $\boldsymbol{\tau}(\mathbf{DEK})$ belongs to set $\boldsymbol{S} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \dots, \mathbf{p-2}, \mathbf{p-1}\}$. For each candidate value $\boldsymbol{\tau}'(\mathbf{DEK})$ in $\boldsymbol{S}$ an attacker can construct one and only one polynomial $\boldsymbol{f'(x)}$ of degree 1 that states: $\boldsymbol{f'(0)} = \boldsymbol{\tau}'(\mathbf{DEK})$ and $\boldsymbol{f'(x_a)} = \mathbf{y_a}$. Because of this every polynomial $\boldsymbol{f'(x)}$ is equally possible and every value $\boldsymbol{\tau}'(\mathbf{DEK})$ from $\boldsymbol{S}$ is equally possible. This yields $\mathbf{p}$ equally possible values and an attacker has gaining no real knowledge about correct value of $\boldsymbol{\tau}(\mathbf{DEK})$. That is why the probability of finding right polynomial $\boldsymbol{f(x)}$ and thus right value of $\boldsymbol{\tau}(\mathbf{DEK})$ is equal to $\frac{\mathbf{1}}{\mathbf{p}}$.

Lemma 1 shows that practical security of Shamir's Secret Sharing $(\mathbf{2}, \boldsymbol{k})$ threshold scheme used in the Secrets Chats Protocol highly depends on the value of prime number $\boldsymbol{p}$. Using larger prime number yields lower value of probability of finding $\boldsymbol{\tau}(\mathbf{DEK})$.

5.2. The Common Container Algorithm

The proposed Common Container Algorithm (CCA) is using prime numbers and arithmetic operations in multiplicative groups of integers modulo an positive integer $\boldsymbol{n}$. It is used within the Secret Chats Protocol to hide in common values different secret shares. The CCA shows a new way of hiding different information in a single common value.

The main security flaw is the simplicity of the algorithm. The CCA consists of a few very simple operations, which makes any possible attacks (e.g. brute-force) very efficient. This is why the value of $\boldsymbol{n}$ should be chosen correctly to make brute-force attacks practically impossible (lasting too long to succeed). Because of this, when choosing the right value of $\boldsymbol{n}$, the strength of the possible attacker (e.g. possible computational power) should be considered. On the other hand, this simplicity is also CCA's strength, because it can hide a lot of information associated with different prime numbers and even for a huge value of $\boldsymbol{m}$ (multiplication of all used prime numbers) the necessary computations can be done on devices with relatively low computational power.

Let us analyse the CCA retrieval steps of the hidden value $\boldsymbol{f(p_i)}$ for unknown $\boldsymbol{p_i}$. An attacker can get the values of $\boldsymbol{w}, \boldsymbol{m}, \boldsymbol{r_i}$ from the application. He can then, according to (15), compute the value of $\boldsymbol{v} = (\boldsymbol{w} \cdot \boldsymbol{r_i})\ (\boldsymbol{mod\ m})$. In the next step, an attack needs to solve equation (15) $\boldsymbol{f(p_i)} = (\boldsymbol{v})\ (\boldsymbol{mod\ p_i})$ in which he gets two unknown values – $\boldsymbol{f(p_i)}$ and $\boldsymbol{p_i}$. The attacker could try to get some information about $\boldsymbol{p_i}$ from value of $\boldsymbol{r_i}$ which was computed by evaluating the formula (11) $\boldsymbol{r_i} =$

$\boldsymbol{[(f(p_i))^{-1}_{m'}]^{-1}_{p_i} \cdot (w')^{-1}_{p_i}}$, where $\boldsymbol{w'}$ is the value of $\boldsymbol{w}$ before the value of $\boldsymbol{f(p_i)}$ is hidden in it, and $\boldsymbol{m'}$ is the value of $\boldsymbol{m}$ before it is multiplied by the value of $\boldsymbol{p_i}$. An attacker once again gets an equation with the multiplication of two unknown values that are different than in previous equations.

$$f(p_i) = (v)(mod\ p_i);\ r_i = [(f(p_i))^{-1}_{m'}]^{-1}_{p_i} \cdot (w')^{-1}_{p_i} \tag{24}$$

From (24) we do not see any way to effectively retrieve the value of $\boldsymbol{p_i}$ and $\boldsymbol{f(p_i)}$.

An attacker can try to break the CCA algorithm from a different side. In the CCA, the secrecy of prime numbers $\boldsymbol{p_i}$ is very important for the overall security. The knowledge of the $\boldsymbol{p_i}$ value allows for an easy reconstruction of the value of $\boldsymbol{f(p_i)}$. This is why an attacker can try to factorize the value of $\boldsymbol{m}$ and get information about all or some values of $\boldsymbol{p_i}$'s. When the value of $\boldsymbol{n}$ appropriate for the security of the application is chosen, this fact must be taken into account. The value of $\boldsymbol{m}$ in the simplest case of the CCA is the multiplication of two $\boldsymbol{n}$-bit prime numbers:

$$\begin{gathered} \boldsymbol{p_1 \in \mathbf{P}, p_2 \in \mathbf{P};} \\ \boldsymbol{m = p_1 \cdot p_2} \end{gathered} \tag{25}$$

The product $\boldsymbol{m}$ is a composite number containing $\boldsymbol{2 \cdot n}$-bits. The best publicly known (at the time of writing this paper) algorithm for integer factorization is the General Number Field Sieve (GNFS) [16]. Its asymptotic running time can be computed by evaluating the formula:

$$t(n) = e^{\left(((\frac{64}{9})^{\frac{1}{3}} + o(1)) \cdot (\ln(2^{2n}))^{\frac{1}{3}} \cdot (\ln(\ln(2^{2n})))^{\frac{2}{3}} \right)} \tag{26}$$

For the recommended value of $\boldsymbol{n = 768}$ we get:

$$t(2 * 768) = e^{\left(((\frac{64}{9})^{\frac{1}{3}} + o(1)) \cdot (\ln(2^{3072}))^{\frac{1}{3}} \cdot (\ln(\ln(2^{3072})))^{\frac{2}{3}} \right)} \approx e^{146{,}172} \tag{27}$$

which gives a security level of 146 bits. This calculation is based also on the assumption that the best algorithm for factorization of the value of $\boldsymbol{m}$ is a general-purpose algorithm GNFS and no special purpose algorithms (e.g. such as Fermat's factorization [17]) are suitable. Using cryptographically secure prime numbers, as defined in set **P** and formulas (8) (21), ensures that no special purpose algorithms are suitable. On the other hand, using cryptographically secure prime numbers lowers overall algorithm efficiency – finding secure prime numbers is more complex than finding prime numbers. Because of this, applications in which security is not critical could lower requirements for set **P** and use in (6) and (14) prime numbers instead of cryptographically secure prime numbers.

5.3. The Secret Chats Protocol

The security level of the Secret Chats Protocol depends on the quality and the size (in terms of number of bits in binary representation) used prime number. For the general-purpose usage, the recommended size is 768.

Lemma 2: Using prime numbers of 768-bits sets security level of the Secret Chats Protocol to 146 bits.

Proof: A user passcode will be expanded (or trimmed) in (5) to 768 bits value and $\boldsymbol{p_{passcode}}$ will be a prime number with binary representation of at least 768 bits. From user passcodes will be constructed value of $\boldsymbol{m}$. According to previous sections $\boldsymbol{m}$ must be a product of at least 2 primes. This gives at least 1536 bits of $\boldsymbol{m}$, which gives asymptotic running time of GNFS around $\boldsymbol{e^{146,172}}$, which gives 146 bits of security. Also, a prime number $\boldsymbol{p}$ from (23) used in Shamir's Secret Sharing $(\boldsymbol{2}, \boldsymbol{k})$ threshold scheme must be a prime number with binary representation of at least 768 bits. The total number of prime numbers having 768 bits be estimated by evaluating the formula [17]:

$$\Pi(x) \approx \frac{x}{lnx};$$
$$\Sigma(n) = \Pi(2^n) - \Pi(2^{n-1});\ \Sigma(768) = \Pi(2^{768}) - \Pi(2^{767}) \approx 2^{757,942} \qquad (28)$$

which gives approximately almost $\boldsymbol{2^{758}}$ different prime numbers which can be chosen. This yields a probability of finding right polynomial $\boldsymbol{f(x)}$ and thus right value of $\boldsymbol{\tau(\mathrm{DEK})}$ to $\boldsymbol{\frac{1}{2^{758}}}$. Taking into an account a birthday paradox [18], we must have at least $\boldsymbol{2^{379}}$ different 768-bits prime numbers to get a probability of finding right one greater than $\boldsymbol{\frac{1}{2}}$. This yields security level of 379 bits. So we have that the security level of the Secret Chats Protocol is 146 bits.

5.4. Known limitations

The first limitation of the CCA that affects the Secret Chats Protocol and its Shamir's Secret Sharing part is the maximum value of polynomial (2) for a given prime number $\boldsymbol{p_i}$. Because of formula (15) that is used to retrieve the value of $\boldsymbol{f(p_i)}$, the value of $\boldsymbol{f(p_i)}$ must be less than the value of $\boldsymbol{p_i}$. This is because computations in (15) are performed in $\boldsymbol{Z^*_{p_i}}$ and when $\boldsymbol{f(p_i)}$ is greater than $\boldsymbol{p_i}$, the value of $\boldsymbol{f(p_i)\ mod\ p_i}$ will be retrieved instead of the value of $\boldsymbol{f(p_i)}$.

The next limitation is associated with formula (10) and (11). In all those formulas the modular inversion is computed. However, this computation is only feasible when the greatest common divisor (GCD)[19] of given value and modular component is equal to 1. Because of the total number of prime numbers (36) which may be used in the CCA, the probability that such condition occurs goes asymptotically to 0. However, if this situation occurs, the application could ask the user to enter another passcode.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed mechanisms for improving the security level of protecting application data at rest. Our generic mechanism enables app developers to use both – operating system keystores and device secure hardware with a secret known only to the user. This approach ensures that the application user will always have to agree to use his cryptographic keys, thus making it much harder to recover application data. The Secret Chats Protocol provides new quality in protecting data stored in a common container. That kind of cryptographic protection may be used for apps that store their data in some common container – e.g. a database, but access to that data should be limited only to specific users. Nowadays, this kind of secret protection is often realized by application logic – a good example of such solution is the hidden chat option in the Viber messenger. With the secret chat protocol, it is possible to create virtual hidden containers without any limitations, and without the risk of being disclosed, e.g. by analysing application settings. Such type of secret chats can be extremely useful in situations where not all of the communication history can be presented.

We have presented a brief study for application recommendations, which guarantee a fair security level for commercial use of the secret chat protocol by application developers. The security analysis presented deals with the most important vectors of attack used for a secret sharing scheme and its mathematical principles. We have also described limitations known to us that application developers must be aware of.

In our future work, we plan to carry out performance testing on various hardware and software platforms, to ensure that efficiency of the secret chat protocol is enough for practical use. We plan to carry out an extended security analysis of the whole scheme, together with the analysis of optimization possibilities. During the optimization process, we plan to look at the possibility of using other mathematical primitives that will enable us to make the whole scheme resistant to quantum computing cryptanalysis algorithms.

REFERENCES

[1] Statista, *Number of smartphone users worldwide from 2016 to 2021*, https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide, last accessed 2020/02/17

[2] *The Times*, http://www.thetimes.co.uk/tto/news/world/europe/article4262527.ece, last accessed 2020/02/17

[3] K. Kaczyński, Security analysis of Signal Android database protection mechanisms, *International Journal on Information Technologies and Security*, **4** (vol. 11), Dec. 2019, pp. 63-70

[4] M. Glet, Security analysis of Signal data storage mechanisms in iOS version, *International Journal on Information Technologies and Security*, **4** (vol. 11), Dec. 2019, pp. 71-88

[5] Sabt, Mohamed, and Jacques Traoré. Breaking into the keystore: A practical forgery attack against the Android keystore. *European Symposium on Research in Computer Security*. Springer, Cham, 2016, pp. 531-548.

[6] Joeri de Ruiter Time Cooijmans and Erik Poll. Analysis of secure key storage solutions on Android. *Security and Privacy in Smartphones and Mobile Devices (SPSM'2014)*, 2014, pp. 11-20.

[7] *Android keystore system*, https://developer.android.com/training/articles/keystore, last accessed 02/18/2020

[8] Moriarty, Kathleen, Burt Kaliski, Andreas Rusch. Pkcs# 5: Password-based cryptography specification version 2.1. *Internet Engineering Task Force (IETF)*, 2017.

[9] Poonguzhali, P., Dhanokar, P., Chaithanya, M. K., & Patil, M. U. Secure storage of data on android based devices. *International Journal of Engineering and Technology*, **3** (vol. 8), 2016, p. 177.

[10] Boukayoua, F., Lapon, J., Decker, B. D., Naessens, V. *Improving secure storage of data in android*. KU Leuven – Internal Report., 2014, https://lirias. kuleuven. be/handle/123456789/446974,

[11] Shamir, Adi. How to share a secret. *Communications of the ACM*, 22.11, 1979, pp. 612-613.

[12] ANS X9.80, *Prime Number Generation, Primality Testing and Primality Certificates*, https://webstore.ansi.org/standards/ascx9/ansix9802005

[13] *FIPS PUB 186-4 – Digital Signature Standard (DSS)*, National Institute of Standards and Technology, USA, 2013

[14] Lagrange Interpolating Polynomial (last accessed 03/07/2020), https://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html

[15] Schwarzweller, C. The Chinese Remainder Theorem, its Proofs and its Generalizations in Mathematical Repositories. *Studies in Logic, Grammar and Rhetoric* 18(31), 2009, pp.103–119.

[16] Pomerance, Carl. A tale of two sieves. *Biscuits of Number Theory,* 85, 2008, p. 175.

[17] Dickson, Leonard Eugene. *History of the theory of numbers: Diophantine Analysis*. Vol. 2. Courier Corporation, 2013.

[18] *Birthday paradox*, https://en.wikipedia.org/wiki/Birthday_problem, last accessed 05/07/2020

[19] *GCD*, https://en.wikipedia.org/wiki/Greatest_common_divisor, last accessed 03/05/2020

Information about the authors:

Michał Glet – Military University of Technology, R&D assistant, Cryptography, Blockchain, Cryptanalysis, Cybersecurity, Mobile applications, Malware Analysis, Software Reverse Engineering, Software Development.

Kamil Kaczyński – Military University of Technology, R&D assistant, Cryptography, Steganography, Blockchain, Cryptanalysis, Steganalysis, Mobile applications, Internet of Things (IoT).

Manuscript received on 14 October 2020