Flexible Password-Based Encryption: Securing Cloud Storage and Provably Resisting Partitioning-Oracle Attacks - A Literature Review

Rishabh Gupta[1], Ujjwal Prakash[1], Prashant Kumar[1]

[1]International Institute of Information Technology, Hyderabad, India
https://www.iiit.ac.in/
{rishabh.g,ujjwal.prakash,prashant.k}@students.iiit.ac.in

Abstract. Flexible password-based encryption (FPBE) is a way to encrypt data using a password. It is designed to be used in modern applications such as end-to-end secure cloud storage. FPBE allows for the reuse of nonces, associated data, and salts, making it easier to use in practice. FPBE also adds an authenticity requirement to the usual privacy requirement, which is important because end-to-end security must protect against a malicious server. To achieve this, a specific FPBE scheme called DtE (Derive then Encrypt) is proposed. It has been proven to be secure and has good bounds. However, there is a challenge in circumventing partitioning-oracle attacks, which is done by leveraging key-robust encryption and a notion of authenticity with corruption. The key challenge is probably resisting partitioning-oracle attacks.

Keywords: DtE · FPBE · TPBE · Security.

1 Summary

FPBE is a method of encrypting data using a password, specifically designed for modern applications. It ensures that the data is not tampered with by anyone unauthorized, and the scheme is proven to be secure. However, there are still some challenges to overcome in order to make it practical and efficient.

1.1 Traditional PBE

The paper defines traditional password-based encryption (TPBE) which encrypts a message M using the password P, a random salt S, and a conventional symmetric encryption scheme SE (Symmetric Encryption). The key K is obtained by hashing the salt and password ($K \leftarrow H(P,S)$), and the ciphertext is (S, C) where C is the encryption of M under K ($C \leftarrow SE.Enc(K, M)$) using a conventional symmetric encryption scheme SE.

Limitations of TPBE: TPBE only provides message privacy under a **chosen-plaintext attack**, and it fails to define or prove authenticity. In short, traditional PBE is randomized privacy-only encryption with a fresh, per-message salt.

Chosen Plaintext attacks

- The attacker chooses n plaintexts and then sends these n plaintexts to the encryption oracle.
- The encryption oracle will then encrypt the attacker's plaintexts and send them back to the attacker.
- The attacker receives n ciphertexts back from the oracle, in such a way that the attacker knows which ciphertext corresponds to each plaintext.
- Based on the plaintext–ciphertext pairs, the attacker can attempt to extract the key used by the oracle to encode the plaintexts. Since the attacker in this type of attack is free to craft the plaintext to match his needs, the attack complexity may be reduced.

1.2 Flexible PBE

Flexible PBE has a new syntax and security definition. The key (password) can be any length and encryption is deterministic, taking the salt, nonce, and associated data as inputs. Decryption requires the salt, nonce, and associated data to be sent out of band.

Syntax:- In Flexible PBE, the key is a password (P) which can be any length and encryption is deterministic. The salt (S), nonce (N), and associated data (A) are also used as inputs for encryption (C $\leftarrow$ FPBE.Enc(P, S, N, M, A)). Decryption requires the salt, nonce, and associated data to be sent separately from the encrypted data (M $\leftarrow$ FPBE.Dec(P, S, N, C, A)).

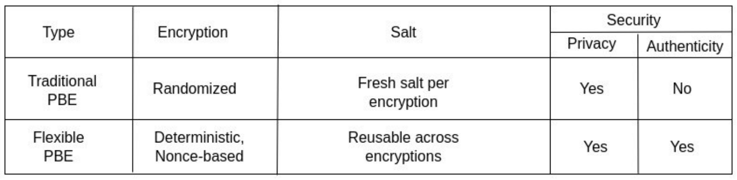
Security:- In a multi-user setting, the password of each user is denoted by P[i]. The distribution PD(vector of passwords) captures the strength of password choices and helps to define privacy and authenticity.
Flexible PBE has three security features:

1. **Privacy (PIND$):** It requires ciphertexts to be indistinguishable from random strings when the salt is honestly chosen, and a nonce is not repeated for a given salt.
2. **Authenticity (PAUTH):** It makes it infeasible to produce valid S, N, C, A (salt, nonce, ciphertext, and associated data) except in a trivial way.
3. **PAE:** It captures both privacy and authenticity in a single, integrated way.

The paper shows that if an FPBE scheme separately satisfies PIND$ and PAUTH, then it also satisfies PAE, i.e., **PIND$ + PAUTH $\rightarrow$ PAE**

Features:- Flexible PBE is a secure encryption method that permits salt to be reused for encrypting multiple messages, with the condition that the nonce is different each time. It enables authenticated but not encrypted associated data, such as metadata. Compared to TPBE, FPBE offers better privacy by requiring indistinguishability from random, providing a certain level of anonymity. However, FPBE's main advantage is authenticity, which TPBE lacks, and is essential for contemporary applications.

As a quick summary, we contrast Flexible PBE and Traditional PBE in Fig 1.



Type	Encryption	Salt	Security	
			Privacy	Authenticity
Traditional PBE	Randomized	Fresh salt per encryption	Yes	No
Flexible PBE	Deterministic, Nonce-based	Reusable across encryptions	Yes	Yes

Fig. 1. Comparison between TPBE and FPBE

2 Applications

- The first application is for securing data stored in the cloud. While most cloud storage providers offer some type of encryption, some services, such as MEGA and Box Cryptor, aim to provide end-to-end secure storage, where the encryption is under a key known only to the user. This means that even the service provider storing the encrypted file cannot decrypt it. However, traditional PBE is not a good fit for this task because it only provides privacy and not authenticity, which is necessary to ensure that the data has not been tampered with by a malicious server.
- The second application of FPBE is to reduce storage costs. With traditional PBE, each message is encrypted with a unique random salt, which adds a lot of storage overhead. However, with FPBE, only one random salt is used, along with a counter that can be short, which reduces the amount of storage needed.
- The third application is to modernize PBE and align it with authenticated encryption (AEAD), which is a more advanced form of symmetric encryption. FPBE adds support for nonces and associated data, and provides both privacy and authenticity, which is consistent with AEAD.

Overall, FPBE is a new type of PBE that offers enhanced security and reduced storage costs, while also aligning with modern cryptographic practices.

2.1 Partitioning-oracle attacks

The Traditional PBE encryption scheme uses a conventional symmetric encryption scheme called the base scheme. The upcoming DtE(Derive then Encrypt) flexible PBE scheme will also use the base scheme.
A base scheme is considered key-robust (or key-committing) if a ciphertext serves as a commitment to the key. This means that once a ciphertext is generated, it cannot be decrypted with any other key.
However, a new type of attack called partitioning-oracle attacks can exploit the lack of key robustness in the base scheme to speed up password recovery in the corresponding traditional PBE scheme.
The FPBE framework fills this gap by considering attacks that aim to violate authenticity, which includes partitioning-oracle attacks.
Assume that the attacker has access to the password's dictionary and that the server's password is also present in that dictionary.

Brute-force Attack

- Attacker takes the passwords one by one and sends it to the server after encrypting them. If the server responds with the decryption error then he continues on to the next password.
- This would take O(d) time, where d is the size of the password dictionary.

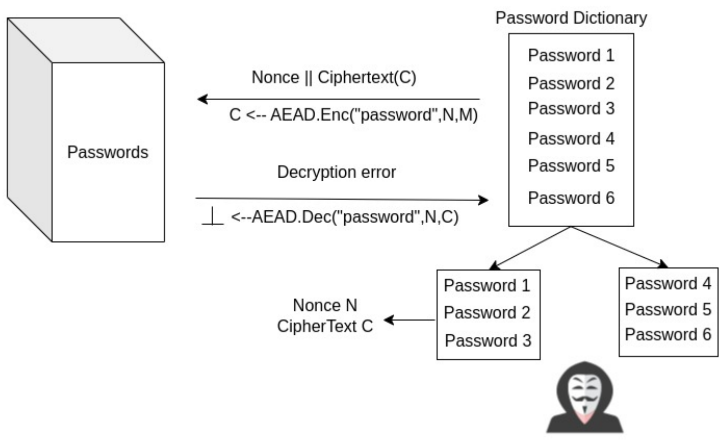


Fig. 2. Partitioning-Oracle Attacks

- Attacker splits the passwords into two sets and calculates the nonce of each set in such a way that every password of one set can decrypt the message while other sets cannot.

- Oracle-partitioning attacks eliminate half of the password set each time, thus reducing the complexity to O(log d).

3 The DtE Scheme

The DtE is a way to create a secure encryption scheme called FPBE. It has two main ingredients:

1. A conventional Authenticated encryption with associated data (AEAD) scheme, SE.
2. A password-based key-derivation function(PBKDF), F.

The DtE scheme defines FPBE scheme, FPBE = DtE[SE, F] as:
FPBE.Enc(P, S, N, M, A) derives K $\leftarrow$ F(P, S) and returns C $\leftarrow$ SE.Enc(K, N, M, A)

3.1 Password Strength

The security of the DtE scheme depends on the strength of the password used. The password needs to be strong, which means it should be hard to guess, in order to provide security for the PBE scheme.

3.2 Security of DtE

The scheme we analyze is FPBE = DtE[SE, F] with F(P, S) = H(P, S) where H is modeled as a random oracle. The security of the DtE scheme can be analyzed at two levels:

1. **Asymptotic (or Qualitative) -** At the asymptotic level, assuming the passwords is un-guessable.
   (a) If base scheme SE provides privacy, then FPBE meets our PIND$, definition of privacy.
   (b) If base scheme SE provides authenticity, then FPBE meets our PAUTH, definition of authenticity.
   (c) If base scheme SE provides both authenticity and key robustness, then FPBE again meets PAUTH.

   At this level, looks like redundancy; why do we add an extra assumption (key robustness) on SE to get the same result as in (a)

2. **Concrete (or Quantitative) -** At the concrete level, the security of the DtE scheme is analyzed by bounding the advantage of an adversary in violating the privacy or authenticity of FPBE. The bound depends on the number of guesses q that the adversary can make to guess the password. The lower the value of q, the better the result.

4 Salt vs. Nonces

- Salt is used to prevent pre-computation in brute-force attacks, forcing the attacker to do dictionary-size work.
- Salt makes each user's password unique, making it more difficult for an attacker to pre-compute a list of possible passwords.
- Nonces are used to make ciphertexts unique. However, nonces can be predictable and the same for different users, so they do not provide the same level of security as salts.
- Using both salts and nonces can have benefits, such as reducing storage costs in cloud encryption.

5 Storage Overhead In TPBE

Here we analyze the storage overhead in TPBE and how FPBE overcomes it.

- With traditional PBE, each message is encrypted with a per-message random salt, which needs to be stored along with the ciphertexts. This results in a storage overhead of sl * q (sl is the salt length, q is the number of messages).
- With flexible PBE, the user picks a single random salt S and encrypts each message M with salt S and a nonce $\langle i \rangle c$, where i is the index of the ciphertext in the list and c is a small encoding of i.
- This results in a storage overhead of sl + qc, which is lower than sl * q.
- In fact, with FPBE, only the single salt needs to be stored since the nonce can be derived from the index i of the ciphertext. This results in a storage overhead of just sl.

6 The tool: Symmetric Encryption

The paper builds the Flexible PBE scheme from the symmetric encryption scheme. Symmetric encryption is also called secret key encryption as the key must be kept secret from third parties.

6.1 SE syntax

The asymmetric encryption scheme specifies a key length SE.kl $\in$ N, nonce space SE.NS, message space SE.MS and associated data space SE.AS. The scheme has deterministic encryption and decryption algorithms and decryption correctness requires that the decryption of ciphertext with the corresponding nonce and key returns the plaintext. The encryption algorithm is SE.Enc : $\{0,1\}^{SE.kl}$ x SE.NS x SE.MS x SE.AS $\rightarrow$ $\{0,1\}^*$ returns a ciphertext C $\leftarrow$ SE.Enc(K, N, M, A). The decryption algorithm is SE.Dec : $\{0,1\}^{SE.kl}$ x SE.NIS x $\{0,1\}^*$ x SE.AS $\rightarrow$ SE.MS $\cup$ $\{\perp\}$ returns an output M $\leftarrow$ SE.Dec(K, I, C, A) that is either a string in SE.MS or is $\perp$ where SE.NIS is the nonce-information space. The nonce-information function allows for the recovery of other encryption syntaxes as special cases and enables decryption with partial information about the nonce.

6.2 Security Games and Adversary Classes

The concept of security games and adversary classes is used to define and achieve two levels of security - basic and advanced. The basic level requires that an encryption nonce not be re-used by a particular user, while the advanced level drops this condition. To achieve this level of security, three security goals are considered: privacy (PIND$), authenticity (AUTH), and joint privacy+authenticity (AE).

6.3 SE Privacy

```
– INIT:
    1. d ←{0,1} ; un ← true
    2. For i = 1...u do
    3. $K_i \leftarrow \{0,1\}^{SE.kl}$

– ENC(i, N, M, A):
    1. Require CT[$i, N, M, A$] = ⊥
    2. if ($N \in UN_i$) then un ← false
    3. $UN_i \leftarrow UN_i \cup$ {N}
    4. $C_1 \leftarrow$ SE.Enc($k_i, N, M, A$)
    5. $C_0 \leftarrow \{0,1\}^{SE.cl(|M|)}$
    6. CT[i, N, M, A] ← $C_d$
    7. Return $C_d$

– FIN(d'):
    1. Return (d' = d)
```

This code is a simplified version of a cryptographic protocol that is called the "Gind$ SE protocol", which provides privacy-preserving encryption of data. The protocol is executed by a set of u parties (indexed from 1 to u), each of which holds a secret key K_i, and has access to a common source of randomness. The protocol consists of two main procedures: Init and Enc.

- The Init procedure initializes the protocol by generating a random bit d and a set of uninitialized sets UN_i for each party i. The bit d is used as a secret shared between all parties, and the sets UN_i are used to keep track of the set of nonce values that have been used by each party.
- The Enc procedure takes as input a message M, a nonce value N, and an additional parameter A, and returns a ciphertext CT[i, N, M, A] that can be decrypted by the party i that holds the secret key, K_i. The procedure works as follows:
- Initializes the ciphertext value to an undefined value.
- Checks if the nonce N has already been used by the party i. If it has, then the party sets a flag un to false to indicate that the protocol has been compromised.

- Adds the nonce N to the set of used nonce values UNi for party i.
- Encrypts the message M using the secret key K_i, the nonce value N, and the additional parameter A, to obtain a ciphertext C1.
- Generates a random bit C0.
- Combines the encrypted message C1 and the random bit C0 to form the final ciphertext value Cd, and stores it in the corresponding entry in the table CT[i, N, M, A].
- Returns the ciphertext Cd.

The Fin procedure simply returns the value of the secret bit d. In terms of privacy, the SE protocol provides a strong form of privacy called semantic security. This means that an attacker who observes the ciphertexts generated by the protocol cannot learn anything about the underlying plaintexts, except for what can be inferred from the length and format of the ciphertexts themselves. The protocol achieves this by using a combination of randomization (in the form of the random bit C0) and encryption (in the form of the ciphertext C1) to obscure the underlying plaintext. Additionally, the use of nonce values helps to prevent replay attacks and other forms of cryptographic attacks.

6.4 SE Authenticity

```
– INIT:
    1. un ← true;
    2. For i = 1...u do
    3. $K_i \leftarrow \{0,1\}^{SE.kl}$

– ENC(i, N, M, A)
    1. if($N \in UN_i$) then un ← false
    2. $UN_i \leftarrow UN_i \cup$ {N}
    3. C ← SE.Enc($K_i$, N, M, A)
    4. MT[i, SE.NI(N), C, A] ← M
    5. Return C

– VERIFY(i, I, C, A):
    1. if($MT[i, I, C, A] \neq \perp$) then return $\perp$
    2. M ← SE.Dec($K_i$,I,C,A)
    3. if($M \neq \perp$) then win ← true
    4. return (M $\neq \perp$)

– FIN:
    1. Return win
```

The code implements a game called Game G_{AUTH} using a symmetric-key encryption scheme. The game is designed to check the authenticity of messages being transmitted between two parties. The game has two players: the sender (S) and the receiver (R).

- The initialization stage generates a random bit string K_i of length u for each player i.
- Each player also maintains a set UN_i of unique nonces received so far. A nonce is a one-time-use random number that helps to prevent replay attacks.
- The encryption function Enc takes as input a nonce N, a message M, and some additional data A.
- If the nonce N is already in the set UNi, then the player cannot use it again to avoid replay attacks, so the function returns an error value.
- Otherwise, the function encrypts the message M using the key Ki, the nonce N, and the additional data A, and stores the encrypted message C in a matrix MT.
- The function returns the encrypted message C.
- The verification function Verify takes as input the player's index i, the nonce I, the encrypted message C, and the additional data A.
- If the player has not received the nonce I before, then the function returns an error value.
- Otherwise, the function decrypts the message C using the key Ki, the nonce I, and the additional data A, and checks whether the decrypted message matches the original message M.
- If the decrypted message matches the original message, then the function returns true to indicate authenticity otherwise, it returns false.
- In the FIN step, the game is won by the player who receives a valid message from the other player first. The function returns true if the player wins otherwise, it returns false.

In summary, the given code implements a simple authentication protocol using symmetric-key encryption and nonces to prevent replay attacks. Each player generates a random key and maintains a set of unique nonces received so far. The sender encrypts the message using the key and a nonce, and the receiver verifies the authenticity of the message by decrypting it using the same key and nonce. The game is won by the player who receives a valid message first.

7 The Goal: Flexible password-based encryption

The paper defines both privacy and authenticity, as well as joint authenticated encryption(PAE).

7.1 FPBE syntax

The FPBE scheme consists of a keyspace FPBE.KS = $\{0, 1\}^*$, salt space FPBE.SS, nonce space FPBE.NS, associated data(header) FPBE.AS and message space FPBE.MS. These spaces are assumed to be length-closed. The deterministic encryption algorithm is defined as FPBE.ENC: FPBE.KS x FPBE.SS x FPBE.NS x FPBE.MS x FPBE.AS $\rightarrow \{0, 1\}^*$ and returns a ciphertext C $\leftarrow$ FPBE.ENC(P, S, N, M, A). The deterministic decryption algorithm is defined as FPBE.DEC:

FPBE.KS x FPBE.SS x FPBE.NIS x $\{0,1\}^*$ x FPBE.AS $\rightarrow$ FPBE.MS $\cup \perp$ returns an output M $\leftarrow$ FPBE.DEC(P, S, I, C, A) i.e. either a string in FPBE.MS or is $\perp$.

7.2 Security Games and Adversary Classes

The paper aims to improve the security of Password-Based Encryption(PBE) by treating both basic and advanced security. To achieve this, we use a concept called "security games" which involves an attacker (or adversary) trying to break the encryption scheme, and a defender trying to keep it secure. It describes the usage of a single game per security goal, which is restricted to adversary classes denoted A_b (basic) or A_a (advanced). The security goals considered are privacy(PIND\$), authenticity(PAUTH) and joint privacy and authenticity(PAE).

7.3 FPBE Privacy

```
– INIT:
    1. d $\leftarrow$\{0,1\} ; un $\leftarrow$ true
    2. P $\leftarrow$ PD // u-vector of passwords

– ENC(i, N, M, A):
    1. Require s(i) $\neq$ 0
    2. Require CT[$i$, $s(i)$, $N$, $M$, $A$] = $\perp$
    3. if (N $\in UN_{i,s(i)}$) then un $\leftarrow$ false
    4. $UN_{i,s(i)} \leftarrow UN_{i,s(i)} \cup$ \{N\}
    5. $C_1 \leftarrow$ FPBE.Enc($P[i]$, $S_i$, $N$, $M$, $A$)
    6. $C_0 \leftarrow \{0,1\}^{FPBE.cl(|M|)}$
    7. CT[i, s(i), N, M, A] $\leftarrow C_d$
    8. Return $C_d$

– SALT(i):
    1. s(i) $\leftarrow$ s(i) + 1 ; $S_i \leftarrow$ FPBE.SS
    2. Return $S_i$

– FIN(d'):
    1. Return (d' = d)
```

- **Init:** The initialization step randomly assigns a value of 0 or 1 to the variable d, which is a secret known only to the encryption system. This helps to maintain the privacy of the encryption system's internal state. The variable un is also set to true, which indicates that no unauthorized access has been detected yet. The passwords P are randomly generated and stored in a vector PD, which should be kept private to prevent unauthorized access.

- **Enc(i, N, M, A):** The encryption function takes as input the index i of the password to be used, the attribute set N, the message M to be encrypted, and any additional attribute A. The function first checks that the salt value s(i) for the password is not zero, which ensures that the password has been initialized before use. It also checks that the ciphertext CT[i, s(i), N, M, A] has not already been computed, which helps to prevent unauthorized access.
- The function then checks whether the attribute set N is in the user-specific set UNi,s(i), which tracks the attribute sets used for encryption by user i. If N is already in the set, it sets the variable un to false, indicating that unauthorized access has been detected. Otherwise, it adds N to the set and proceeds with the encryption.
- The function generates a ciphertext C1 using the FPBE encryption function, which encrypts the message M using the password P[i] and the salt value Si associated with the password. The function also generates a random bit C0, which is used to conceal the length of the message during encryption. Finally, the function updates the ciphertext array CT with the new ciphertext Cd and returns it.
- **Salt(i):** The salt function generates a new salt value Si for the password with index i and increments the salt index s(i) by 1. The new salt value ensures that the same password will generate a different encryption key each time it is used, helping to maintain the privacy of user data.
- **Fin(d'):** The final function simply returns a boolean value indicating whether the new internal state d' is equal to the original state d. This function is not directly related to privacy but may be useful for verifying the correctness of the encryption system.

Overall, the FPBE scheme implemented by the code provided can help to protect the privacy of user data by allowing users to encrypt messages using passwords and attributes while keeping the passwords and underlying data private from the encryption system. The use of salt values and user-specific sets helps to prevent attacks and maintain the privacy of user data.

7.4 FPBE Authenticity

```
– INIT:
    1. un $\leftarrow$ true;
    2. P $\leftarrow$ PD // u-vector of passwords

– ENC(i, N, M, A)
    1. Require s(i) $\neq$ 0
    2. if (N $\in UN_{i,s(i)}$) then un $\leftarrow$ false
    3. $UN_{i,s(i)} \leftarrow UN_{i,s(i)} \cup$ {N}
    4. C $\leftarrow$ FPBE.Enc(P[i], $S_i$, N, M, A)
    5. MT[i, $S_i$, FPBE.NI(N), C, A] $\leftarrow$ M
    6. Return C
```

```
– VERIFY(i, S, I, C, A):
    1. if(MT[i,S,I,C,A] $\neq \perp$) then return $\perp$
    2. M $\leftarrow$ FPBE.Dec(P[i], S, I, C, A)
    3. if(M $\neq \perp$) then win $\leftarrow$ true
    4. return (M $\neq \perp$)

– SALT(i):
    1. s(i) $\leftarrow$ s(i) + 1 ; $S_i \leftarrow$ FPBE.SS
    2. Return $S_i$

– FIN:
    1. Return win
```

- **Initialization (Init):** The initialization step sets a boolean variable un to true, indicating that the game has not yet been won by any player. It also sets the vector P to be the u-vector of passwords that are provided as input.
- **Encryption (Enc):** The encryption step takes as input the player i, a nonce N, a message M, and an additional parameter A. It first checks if the nonce N is already present in the set UNi,s(i), where s(i) is a salt value that is associated with the player i. If the nonce is already present, then the player cannot use it again and the un variable is set to false, indicating that the game cannot be won by that player.
- If the nonce is not already present, then it is added to the set UNi,s(i) and the player encrypts the message M using the password P[i], the salt Si associated with the player i, the nonce N, the message M, and the additional parameter A. The resulting ciphertext C is then stored in the MT (message table) data structure, indexed by the player i, the salt Si, the nonce NI(N), the ciphertext C, and the additional parameter A.
- The Verification step takes as input the player's index i, the nonce I, the encrypted message C, and the additional data A. If the player has not received the nonce I before, then the function returns an error value. Otherwise, the function decrypts the message C using the key Ki, the nonce I, and the additional data A and checks whether the decrypted message matches the original message M. If the decrypted message matches the original message, then the function returns true to indicate authenticity, otherwise, it returns false.

7.5 FPBE PAE

```
– INIT:
    1. d $\leftarrow$ {0,1} ; un $\leftarrow$ true
    2. P $\leftarrow$ PD // u-vector of passwords

– ENC(i, N, M, A)
    1. Require s(i) $\neq$ 0
    2. Require CT[$i$, $s(i)$, $N$, $M$, $A$] $= \perp$
    3. if (N $\in UN_{i,s(i)}$) then un $\leftarrow$ false
    4. $UN_{i,s(i)} \leftarrow UN_{i,s(i)} \cup \{N\}$
    5. $C_1 \leftarrow$ FPBE.Enc($P[i]$, $S_i$, $N$, $M$, $A$)
    6. $C_0 \leftarrow \{0,1\}^{FPBE.cl(|M|)}$
    7. CT[i, s(i), N, M, A] $\leftarrow C_d$
    8. MT[i, $S_i$, FPBE.NI(N), $C_d$, A] $\leftarrow$ M
    9. Return $C_d$

– DEC(i, S, I, C, A):
    1. if(MT[i, S, I, C, A] $\neq \perp$) then return MT[i, S, I, C, A]
    2. (d = 0) then return $\perp$
    3. M $\leftarrow$ FPBE.Dec(P[i], S, I, C, A)
    4. return M

– SALT(i):
    1. s(i) $\leftarrow$ s(i) + 1 ; $S_i \leftarrow$ FPBE.SS
    2. Return $S_i$

– FIN (d'):
    1. Return (d' = d)
```

The above code implements a cryptographic protocol for secure password-protected storage and retrieval of messages, using a variant of the functional encryption scheme called FPBE (Function Private Bi-Encryption). The protocol ensures privacy and authentication of the stored messages for a set of users indexed by i.

Privacy: The protocol ensures that only authorized users can access the stored messages, and other users cannot learn any information about the messages or the passwords used by the authorized users. This is achieved through the use of FPBE encryption, where each user i has a secret password P[i] and a set of unique salts Si, and the messages are encrypted with a key Cd that is derived from the combination of the password and the salt.

Authentication: The protocol ensures that the stored messages are not tampered with or replaced by unauthorized parties, and only authorized users can modify or delete their own messages. This is achieved through the use of a unique identifier NI(N) for each message N, and the storage of the encrypted message and its identifier in a hash table MT indexed by i, Si, NI(N), Cd, and an auxiliary data structure CT indexed by i, s(i), N, M, A. The hash table and the auxiliary data structure are updated only by authorized users who have the correct password and salt, and the updates are protected by FPBE encryption. The protocol works as follows:

- **Init:** Initialize the parameters of the protocol, including the vector of passwords PD, a flag d that indicates whether the protocol is active or not, and a flag un that indicates whether any new messages have been added to the hash table. Each user i has a salted password P[i] and an empty set UNi,s(i) of used message identifiers.

- **Enc:** Encrypt a new message M for a user i, using the password P[i], the current salt Si, a new unique identifier N, and a set of auxiliary data A. The protocol checks that the identifier is not already used by the user, and updates the set of used identifiers UNi,s(i) accordingly. The protocol also sets the flag un to false if the identifier is new. The message is encrypted using the FPBE.Enc function, which returns two ciphertexts C1 and C0, where C1 is used to protect the confidentiality of the message, and C0 is used to protect the authenticity of the ciphertext. The protocol stores the ciphertext C1 and the identifier NI(N) in the hash table MT, indexed by i, Si, NI(N), Cd, and A. The protocol also stores the ciphertext C1, the identifier N, the message M, and the auxiliary data A in the auxiliary data structure CT, indexed by i, s(i), N, M, A. The protocol returns the ciphertext Cd.
- **Dec:** Decrypt an existing message C for user i, using the password P[i], the salt S, the identifier I, and the auxiliary data A. The protocol checks if the message is already in the hash table MT, and returns it if so. Otherwise, the protocol decrypts the ciphertext C using the FPBE.Dec function, which returns the message M. The protocol checks the authenticity of the ciphertext by verifying that C0 is either 0 or 1. The protocol also checks that the identifier I matches the stored identifier NI(N) in the hash table, and that the auxiliary data A matches the stored data in the auxiliary data structure CT. If all checks pass, the protocol stores the decrypted message M in the hash table MT and returns it. Otherwise, the protocol returns $\perp$ (null).
- **Salt:** Generate a new salt Si for user i, and update the corresponding salted password P[i] with the new salt. The protocol increments the salt index s(i) by 1.

7.6 Theorem

Let FPBE be an FPBE scheme over u $\geq$ 1 users, password distributions PD, and salt length sl $\geq$ 1 with access to a random oracle H: D $\rightarrow$ R. Let y $\in$ {b, a}. Suppose A $\in A_y$ is an adversary making q_s SALT queries, q_e ENC queries, q_d DEC queries and q_h H queries in the $G^{pae}_{FPBE,PD,u}$ game in the ROM. Then we can construct adversaries $A_{pind\$} \in A_y$ and $A_{pauth} \in A_y \cap A_{seq}$ in the ROM such that:

$$Adv^{pae}_{FPBE,PD,u}(\mathrm{A}) \leq Adv^{pind\$}_{FPBE,PD,u}(A_{pind\$}) + 2 \cdot Adv^{pauth}_{FPBE,PD,u}(A_{pauth\$})$$

In the above proof, H refers to a random oracle. A random oracle is an idealized cryptographic primitive that provides a black-box interface for a function. It takes inputs and returns outputs, but the way it computes those outputs is completely random and unpredictable as if it were an oracle that has access to all possible outputs. In the context of the theorem, H is used to model a cryptographic hash function that generates salt values for the password-based encryption scheme. The use of a random oracle is a standard assumption in many cryptographic constructions, as it simplifies the analysis of security properties and provides a high degree of assurance that the scheme is secure.

The theorem shows that if there is an adversary (A) trying to break the security of the scheme by making various types of queries (Salt, Enc, Dec, H), then we can create two new adversaries ($A_{pind\$}$ and A_{pauth}) who are specialized in certain types of queries, and use them to prove the security of the scheme.

To be more specific, $A_{pind\$}$ is designed to handle Salt and Enc queries, while Apauth deals with Salt, Enc, and Verify queries. Both adversaries also make H queries. The theorem states that if we can prove the security of $A_{pind\$}$ and A_{pauth}, then we can conclude that the original adversary A has limited power and is unlikely to break the scheme.

In the above inequality, the factor of 2 in $2 \cdot Adv^{pauth}_{FPBE,PD,u}(A_{pauth\$})$ is important because it accounts for the fact that A_{pauth} makes both encryption and verification queries.

Specifically, Adv_{pauth} measures the advantage that A_{pauth} has in breaking the authentication security of the scheme, which involves verifying that the decryption of a ciphertext produces the correct plaintext. In order to do this, A_{pauth} needs to make both encryption and verification queries, which means it needs to be able to encrypt messages using the secret key and then verify that the corresponding ciphertexts decrypt to the correct plaintexts. This requires more computational effort than just making encryption or verification queries alone.

By including a factor of $2 \cdot Adv_{pauth}$ in the inequality, the theorem takes into account the extra computational effort that A_{pauth} needs to expend to break the authentication security of the scheme, and provides a more accurate estimate of the overall security of the scheme.

The theorem is useful because it allows us to simplify the proofs of security for the overall scheme. We can focus on proving the security of $A_{pind\$}$ and A_{pauth} separately, rather than trying to prove the security of the entire scheme all at once. This makes the proofs easier to manage and less prone to errors.

The theorem also highlights the importance of sequential adversaries, which are adversaries that make queries in a particular order. A_{pauth} is always sequential, but A can be non-sequential. However, the theorem shows that we can still use sequential adversaries to simplify the proofs.

Finally, the theorem notes that the statement holds in both the random oracle model and the standard model, which are two different ways of analyzing the security of cryptographic schemes. The theorem also points out that the reverse direction of the statement (PAE $\rightarrow$ PIND\$ + PAUTH) is also true, meaning that if an adversary breaks either PIND\$ or PAUTH, they can also break PAE with little extra effort.

8 Conclusion

- The paper discusses different ways of defining authenticated encryption, which is a way of encrypting data securely while also ensuring that it hasn't been tampered with.
- It describes how we can use a unique nonce while encryption through an encryption scheme called FPBE to attain both privacy and authenticity.
- It also discusses how we can resist partitioning oracle attacks with the help of the above-mentioned scheme.

References

[1.] J. Len, P. Grubbs, and T. Ristenpart. Partitioning oracle attacks. In M. Bailey and R. Greenstadt, editors, 30th USENIX Security Symposium. USENIX Association, 2021.

[2.] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, Sep. 2000.

[3.] M. Bellare, R. Ng, and B. Tackmann. Nonces are noticed: AEAD revisited. In A. Boldyreva and D. Micciancio, editors, CRYPTO 2019, Part I, volume 11692 of LNCS, pages 235–265. Springer, Heidelberg, Aug. 2019.

[4.] Boxcryptor. Technical overview. https://www.boxcryptor.com/en/technical-overview/, visited on October 17, 2022.

[5.] M. Bellare, R. Ng, and B. Tackmann. Nonces are noticed: AEAD revisited. In A. Boldyreva and D. Micciancio, editors, CRYPTO 2019, Part I, volume 11692 of LNCS, pages 235–265. Springer, Heidelberg, Aug. 2019.