OPEN ACCESS
Indo-JC Indonesia Journal of Computing
ISSN 2460-9056
socj.telkomuniversity.ac.id/indojc

Ind. Journal on Computing
Vol. 7, Issue. 1, April 2022. pp. 15-30
doi:10.34818/indojc.2022.7.1.609

Column-Level Database Encryption Using Rijndael Algorithm and Dynamic Key on Learning Management System

Ariva Syam Mursalat [1], Ari Moesriami Barmawi [2], Prasti Eko Yunanto [3]

[#] *School of Computing, Telkom University*
Bandung, Indonesia

[1] arivasyam@student.telkomuniversity.ac.id

[2] mbarmawi@melsa.net.id
[3] gppras@telkomuniversity.ac.id

Abstract

The course management system's goal is to help learning activities. The system helps to manage tasks, the grading process, and user communications. To avoid unauthorized data access, the course management system needs a mechanism to protect the password that is used in the system's login process. Database encryption using Rijndael algorithm is proposed by Francis Onodueze et al. to protect the data. A key is needed for the encryption process, and the key has to be kept secret. Thus, when the key is static, it is vulnerable against key guessing attacks. To overcome the static key's drawback, a dynamic key generation using Hash Messages Authentication Code - Deterministic Random Bit Generator (HMAC-DRBG) is proposed because it can generate keys periodically. Based on the evaluation, the probability of success key guessing attack using the proposed method is less than using the previous method proposed by Francis Onodueze et al., while the time complexity of those methods is similar.

Keywords: Rijndael, HMAC-DRBG, Pseudorandom-bit.

Abstrak

Tujuan sistem manajemen pembelajaran adalah untuk membantu kegiatan belajar. Sistem membantu mengelola tugas, proses penilaian, dan komunikasi pengguna. Untuk menghindari akses data yang tidak sah, sistem manajemen pembelajaran membutuhkan mekanisme untuk melindungi password yang digunakan dalam proses login sistem. Enkripsi database menggunakan algoritma Rijndael diusulkan oleh Francis Onodueze dkk. untuk melindungi data. Sebuah kunci diperlukan untuk proses enkripsi, dan kunci tersebut harus tetap rahasia. Jadi, ketika kuncinya statis, ia rentan terhadap serangan menebak kunci. Untuk mengatasi kelemahan kunci statis, diusulkan pembuatan kunci dinamis menggunakan Hash Messages Authentication Code - Deterministic Random Bit Generator (HMAC-DRBG) karena dapat menghasilkan kunci secara berkala. Berdasarkan hasil evaluasi, peluang sukses terhadap penyerangan penebakan kunci dari metode yang diusulkan lebih kecil dibandingkan dengan metode yang diusulkan sebelumnya, dimana kompleksitas waktu yang dibutuhkan oleh kedua metode sama.

Kata Kunci: Rijndael, HMAC-DRBG, Pseudorandom-bit.

Received on Januari, 2022. Accepted on March, 2022

I. Introduction

THE course management system's goal is to help learning activities. The system helps to manage tasks, the grading process, and user communications. To be able to access the course management system, a user has to go through the registration and the login process, where the user has to insert their username and password into the system. The username and password combination denotes a unique identity of the user, such that the user can access the course management systems content. Thus, the password has to be protected to avoid unauthorized data access.

To avoid unauthorized data access, the course management system needs a mechanism to protect the username and password data combination in the database. There are many methods proposed in order to protect the data. One of the methods is the column-level database encryption method proposed by Francis Onodueze [1]. Francis Onodueze et al. proposed column-level database encryption using the Rijndael algorithm [1]. To do the encryption process, the Rijndael algorithm needs a key. The encryption process that was proposed by Francis Onodueze et al. has a drawback, where the key used in the encryption process is static. A static key would not change throughout the entire process, such that it is vulnerable against key guessing attacks. Therefore, the dynamic key generation is necessary.

The dynamic generation is conducted by generating random numbers. Several techniques have been proposed including True Random Generators (TRNG) and Deterministic Random Bit Generators (DRBG). TRNG has drawbacks that must be considered, where the hardware requires a lot of energy and inadequate output [2]. In addition to TRNG, there is another research on random numbers proposed [3]. This study uses Quantum Random Number Generator (QRNG) [3], which is not suitable to use in this study. This happens because the method uses bottleneck communication so that the cost required to implement this method is quite large, so it is not commensurate with the value of the data being secured [3]. In addition, there is also a method proposed by Liu et al [4], which uses randomization function in the C++ Library to generate the dynamic key. In the research proposed by Katherine ye et al. [5], it was explained that random values needed a cryptographic function to generate key, nonce, and initialization vectors. Most cryptographic devices rely on pseudorandom generators, one of which is the Deterministic Random Bit Generator (DRBG). DRBG is used to change the level of randomness of random numbers from small to large levels of randomness at the output of pseudorandom.

To overcome the static key's drawback, a dynamic key generation using Hash Message Authentication Code-Deterministic Random Bit Generator (HMAC-DRBG) is used. The selection of HMAC-DRBG is based on the ability of this method to generate random numbers periodically [6]. The random number is further used as the key. Thus, the key used in Rijndael is dynamic. HMAC-DRBG is used because it does not require high energy and cost as TRNG [2] and QRNG [3]. Meanwhile, the method proposed by Liu et al. [4] has a high probability of success in guessing the key.

II. Literature Review

In this section, the theories used in this research are discussed. The discussion is divided into three sub-sections, namely previous research [1] and theoretical basics.

A. *Rijndael Algorithm for Database Encryption on a Learning Management System [1]*

In 2017, Francis Onodueze et al. [1] proposed a method for database encryption on a learning (course) management system [1]). This research is used as our previous method. Francis Onodueze et al. proposed a column-level database encryption on learning management system using the Rijndael algorithm, where encryption is used in the registration and login processes. Encrypted data in the database is a password. In the registration process, the password field is filled with the password encryption result string. While in the login process, the password matching process is carried out by comparing the similarity between the data from the password entered by the user and the decrypted data (encrypted password that has been stored in the database). If the results are similar, the user will be considered as a legitimate user, while if they are not similar, then the user will be considered as an unauthorized user.

Francis Onodueze et al [1] implemented the database access security layer shown in figure 1. At each layer, there are rules defined for databases and applications, where these rules aim to ensure the security of user data.

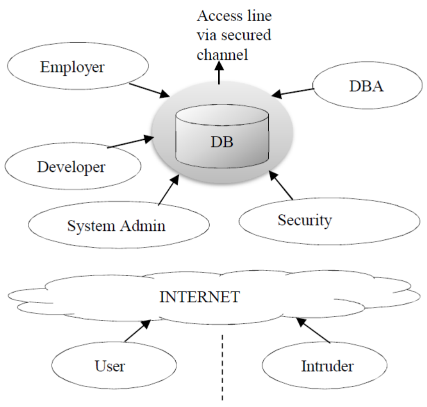

Fig. 1. Access layer database security implemented by Francis Onodueze et al. [1].

1) Encryption of Registration Data: The encryption process on the registration data aims to encrypt the password, where the password data (*plainText*) obtained from the registration process is sent to the hash function to generate a string called *clearText*. The encryption stage in the encryption function begins with the declaration of the encryption key stored in the *encryptionKey* variable. Then the *clearText* is encrypted using Rijndael algorithm and *encryptionKey*. The output in the form of *cipherText* is stored in the password column in the database. while the encryption key is stored in the database used in the login process. Algorithm 1 shows the stages of the encryption in the registration process.

Algorithm 1 Encryption of Registration Data

```
    Input : plainText      Output : cipherText, encryptionKey
1: $encryptionKey \leftarrow keyGenerator()$          ▷ random key generation
2: $clearText \leftarrow hashFunction(plainText)$
3: $cipherText \leftarrow rijndaelEncryption(encryptionKey, clearText)$
4: return cipherText, encryptionKey
```

2) Decryption of Login Data: During the login process, password decryption is performed to decrypt the encrypted *clearText* and further compare the similarity between the *clearText* used in the login phase and the *clearText* obtained by decrypting the *cipherText* stored in the database. The input from this process is the password string (login Password) which is obtained from the login page, while the data in the database is taken in the form of *cipherText* and encryption key. If the value of the *clearText* are similar, the login is successful, and otherwise, the login fails. Algorithm 2 shows the steps of decryption process on login data.

Algorithm 2 Decryption of Login Data

```
    Input : loginPassword        Output : loginStatus
1: $encryptionKey,\ cipherText \leftarrow getfromdatabase$
2: $plainText \leftarrow rijndaelDecryiption(encryptionKey, cipherText)$
3: $loginPassword \leftarrow hashFunction(loginPassword)$
4: if plainText == loginPassword then
5:     return "Login Success
6: else
7:     return "Login Failed"
8: end if
```

B. Dynamic Encryption Algorithm Based on Rijndael

In 2012, Zhiqiang Liu et al. proposed a dynamic encryption based on Rijndael [4]. The algorithm uses a key to encrypt plaintext twice, such that they have two ciphertext as the result. Zhiqiang Liu et al. introduces a function for generating the dynamic key based on random function in C++ library [4], where the probability of success key guessing attack is greater than $1/1024$. The detail of Liu's method is shown on Algorithm 3

Algorithm 3 Dynamic Encryption Algorithm Based on Rijndael

```
1: procedure $AES\_encryptkey(\ byte\ out[16],\ const\ byte\ key[32])$
2:     $i \leftarrow 0$
3:     $j \leftarrow 0$
4:     while $i < 16$ do
5:         $srand(unsigned)time(NULL)$
6:         $j \leftarrow rand()\ mod\ 96$                    ▷ Generating random numbers from
7:         $out[i] \leftarrow key[\ i+16\ ]\ \otimes\ sjbox[\ j\ ]$
8:         $key[\ i+16\ ] \leftarrow sjbox[\ j\ ]$
9:         $i \leftarrow i+1$
10:    end while
11: end procedure
```

C. Database Enryption

A database is a collection of data items that provides an organizational structure for information storage [7]. Information stored in databases is often considered as a valuable and important corporate resource [8]. Encryption is one of the methods to secure the information by changing the form of the data to maintain confidentiality, integrity, and authenticity [1]. Encryption on the database depends on the nature of the data stored (in this case, the data is sensitive or insensitive) [9]. Sensitive data is data that needs to be protected from the access of unrelated parties, while insensitive data is data that does not need special security protection. The more data that is encrypted, the more expensive the costs(such as time and computation) required. Database encryption is divided into 4 levels [1], which are column, row, extreme column level, and database level encryption.

The level of encryption used can be adjusted. The higher the value of the data, the more threats the system will face. According to [10], the 10 most common threats to databases are SQL injection, excessive privilege abuse, abuse of legitimate privileges, privilege escalation, exploitation of vulnerabilities in vulnerable or incorrectly configured databases, weakness of the native audit, denial of Service, vulnerabilities of database communication protocols, unauthorized copying of sensitive data, and exposure of backup data.

1) Column-level encryption: At the column level encryption, the encryption process is carried out to protect certain columns by using a key. Protected fields can be passwords, usernames, ID, credit card numbers, and other important data. The illustration of database encryption at the column level can be seen in table I.

TABLE I
EXAMPLE OF COLUMN-LEVEL DATABASE ENCRYPTION IN THE PASSWORD FIELD

Email	Name	Department	Password
Arvsyam13@gmail.com	Arivan Syam	informatics	xxxxxxxx
jaylaolshop@gmail.com	Jihan Lailatul	electro	xxxxxxxx

2) Row-level encryption: At the row-level encryption, the encryption process is carried out to protect every row in the database by using a key. This method is suitable for small databases and all of its contents are included in very important data. An illustration of row-level database encryption can be seen in table II.

TABLE II
SAMPLE ROW-LEVEL ENCRYPTION

Email	Name	Department	Password
xxxxxx	xxxxxx	xxxxxx	xxxxxxxx
xxxxxx	xxxxxx	xxxxxx	xxxxxxxx

3) Extreme column-level encryption: Extreme column-level encryption is a technique that encrypts specific columns using a different key for each encrypted column. The illustration of the extreme column-level database encryption in the email and password columns can be seen in table III.

TABLE III
EXAMPLE OF EXTREME COLUMN-LEVEL ENCRYPTION IN THE EMAIL AND PASSWORD FIELDS

	Email	Name	Major	Password
Key	Key2	ariva	informatics	Key1
Data	xxxxxxxx	Jihan Lailatul	electro	xxxxxxxx
Data	xxxxxxxx	Ariva Syam	informatics	xxxxxxxx

4) Database level encryption: Database level encryption is a technique that encrypts the framework of the database.

D. Rijndael Algorithm

The Rijndael algorithm is also known as Advanced Encryption Standard (AES) is one of the symmetric encryption methods which is considered a new block cipher that replaces the Data Encryption Standard (DES) [11]. This algorithm is capable of encrypting plaintext of 16 bytes or 128-bits. This algorithm also uses a key (128-bits, 192-bits, and 256-bits) of at least 128-bits and a maximum of 256-bits. In this encryption process, the number of rounds conducted by the process depends on the length of the encryption keys (10 rounds for a 128-bits key, 12 rounds for a 192-bits key, 14 rounds for a 256-bits key).

The first step of Rijndael algorithm is creating keys as many as the number of loop processes. To avoid using the same key in every round, Rijndael proposed to create a new key based on the key schedule. The key is represented in the form of a 4x4 byte matrix. To create a round key, the first step that needs to be done is to take the key byte value from the last column to be shifted up once, then the key byte in that column is substituted with the bytes taken based on the S-box. The column which is the result of the S-box substitution is XOR-ed with the first column of the key and R-con of each round to produce the first column of the round key. To generate The second to the fourth column of the round key, the values in the column and the round key in the previous column have to be XOR. For example, to get the round key of the third column, the second column of the round key is XOR-ed with the value of the third column.

In the Rijndael algorithm, the data to be encrypted is stored in a 4x4 matrix called state. The state will go through four main processes, and those are Sub Bytes, Shift Rows, Mix Column, and Add Round Key. The encryption process of the Rijndael algorithm is shown in Figure 2.

- **Sub Bytes :** Sub Bytes is a process where each byte of the state is substituted with a new byte that is taken based on the S-box.
- **Shift Rows :** Shift Rows is a process in which each byte of the second row in the state is shifted to the left by one column, the third row is shifted to the left by two columns, and the fourth row is shifted to the left by three columns. This matrix is known as the S matrix.
- **Mix Column :** Mix Column is a process where the state resulted from the shift rows process will be converted into a new matrix of 4x4 size. In the Mix Columns step, the cross product between the mix column matrix and the matrix resulted from the shift row process is conducted.
- **Add Round key :** Add Round Key is an XOR process between the round key and the results of the previous three processes.

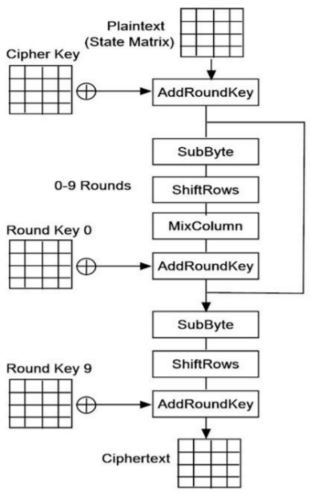

Fig. 2. **Rijndael algorithm encryption process [12]**

E. Hash Messages Authentication Code-Deterministic Random Bit Generator (HMAC-DRBG)

According to [6], DRBG is a mechanism that uses an algorithm that functions to generate a set of bits from an initial value determined by the seed. Seed is the forerunner of a random number that must have sufficient entropy, and the seed is the output of a randomness source (randomness source is a randomization function). Once the seed and initial values have been determined, the DRBG is considered to have been initiated and can be used to generate random numbers. The bits generated by DRBG are pseudorandom bits. To design a good DRBG algorithm, seeds must be kept secret because the output of DRBG must be unpredictable, such that the security strength of DRBG is getting higher. The functional model of DRBG based on [6] is shown in Figure 3.

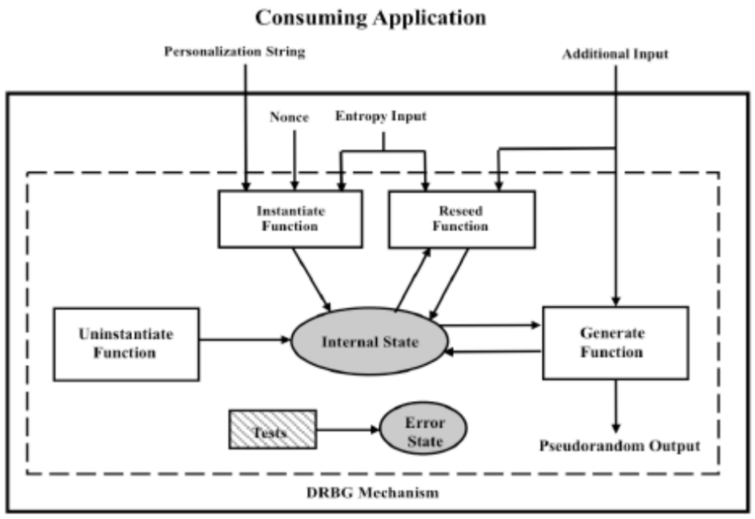

Fig. 3. **DRBG Functional Model [6]**

Based on Figure 3, the DRBG mechanism generally has several components, those are input entropy, additional input, personalization string, nonce, internal state, and the function of the DRBG mechanism.

- **Entropy Input**
  In the DRBG mechanism, the randomization source is used to generate input entropy which will then be used for seed generation where the input and seed entropy are secret. The secrecy of the input entropy is the basis of the security of the DRBG. The randomization source must provide input entropy that can support the security forces required in the DRBG mechanism.
- **Other Input**
  The DRBG mechanism has additional input, personalization strings, and nonce which do not need to be kept secret. Additional input consist of secret or public information. However, if the additional input contains secret private information, that information does not have to be protected at a higher security strength [6].
- **Internal State**
  Internal State is the memory of the DRBG that contains the required parameters and variables.
- **DRBG Mechanism Function**
  The functions that exist in the DRBG mechanism handle the internal state of the DRBG. The DRBG mechanism has five different functions, namely the instantiate, generate, reseed, uninstantiate, and DRBG quality test functions. The instantiate function aims to obtain input entropy for generating seeds. The generate function generates pseudorandom bits as needed using the current internal state and builds a new internal state for subsequent requests. The reseed function generates a new seed using the current internal state and internal state. The uninstantiate function aims to empty or delete the internal state. The function of the DRBG quality check aims is to find out whether the DRBG mechanism is working properly.

According to [6], there are two types of hash-based DRBG algorithms, those are hash-DRBG and HMAC-DRBG. The difference between the two algorithms is in the use of hashes to randomize certain variables. HMAC-DRBG uses a hash function for two variables, while HASH-DRBG only uses a hash function of one variable. Thus, HMAC-DRBG has a higher randomization complexity so that its security strength becomes higher.

In HMAC-DRBG there is an internal state which contains the values of V, K, and a *reseed counter*. The internal state will be updated when a new pseudorandom bit is generated. To generate pseudorandom bits, three processes are carried out, namely the HMAC-DRBG instantiate, generate, and reseed processes. In addition, each process has an HMAC-DRBG update function which is used to update the internal state of the HMAC-DRBG.

The first step that is executed is instantiate which aims to generate the first bit-string (first *seed*) or initial bit-string (initial *seed*), and initialize the values of *V*, *K*, and *reseed counter* which are in the internal state. The internal state is processed at the generate process to generate a pseudorandomrandom bit, then the internal state is updated to create a new random bit code. This stage is repeated based on the reseed-interval that has been determined at the beginning of generation process. Each iteration is recorded by the reseed counter, and if the reseed counter has reached the reseed-interval, the reseed process is carried out to produce a new seed. The new seed is used for the next generation process. At each of these stages, the HMAC-DRBG update function is always conducted to update the *V* and *K* values. The detail of each stage is described as follows:

- **Instantiate Process**
  The instantiate process in HMAC-DRBG aims to generate initial internal state values such as V, Key, and reseed counter. The process is carried out by entering the input in the form of entropy input, nonce, and personalization string. The three input values are concatenated and used as a seed. In addition, after the initialization of the *seed*, *V*, *K*, and *reseed counter* value then *V* and *K* values are updated using HMAC-DRBG update function and the *seed*.
  Based on the input value and initial value (*V* and *K*), this process begins with the creation of a seed by concatenating the input entropy, nonce, and personalization string. Then, initializing *K* with a hexadecimal value of 00 and *V* with a hexadecimal value of 01 as many as the pre-determined number of output bits. In the next step, *K* and *V* are updated by using the internal update function. The value of the *reseed-counter* is initialized with 1 to indicate the start of the new *seed* creation process. The resulting output is *V*, *K*, and *reseed counter*. The algorithm of the instantiate process is shown in algorithm 4.

Algorithm 4 HMAC_DRBG_Instantiate

```
    Input : entropy_input, nonce, personalization_string, security_strength
    Output : V, K, reseed_counter

1: $seed \leftarrow entropy\_input||nonce||personalization\_string$
2: $K \leftarrow 0X00...00$                                        ▷ Outlen bits
3: $V \leftarrow 0X01...01$                                        ▷ Outlen bits
4: $(K, V) \leftarrow HMAC\_DRBG\_Update(seed, K, V)$.
5: $Reseed\_counter \leftarrow 1$
6: return V, K, Reseed_counter
```

- **HMAC-DRBG Update Function**
  The HMAC-DRBG update function aims to update the value of *V* and the *K* of the HMAC-DRBG by hashing *V* using the *K* as a hash key. The inputs of this function are *V*, *K*, and *provided data*, where p*rovided data* is an additional input whose value depends on the stage being conducted. In the instantiate and reseed process, *provided data* contains *seed*, while the generated *provided data* process contains additional input.
  This process begins by checking whether the provided data is empty or not. If the provided data is empty then a hash of *V* and the f*lag* concatenation using the *K* as the hash key is conducted, where the flag is '0x00'. Then the first hash results are used as the hash key for the second hash process of *V*. If provided data is not empty then the first hash process is carried out after the concatenation of *V*, *flags*, and *provided data*. The resulting *K* is used as the hash key for the second hash process of *V*. After conducting the first two hashes, these two hash processes are repeated with the new *flag* value '0x01'. Finally, the HMAC-DRBG update function returns the new *V* value and *K*. The algorithm of the HMAC-DRBG update function can be seen in algorithm 5.

Algorithm 5 HMAC_DRBG_Update

```
Input : provided_data, K, V
Output : K, V

1: $flag \leftarrow' 0x00'$
2: if $isEmpty(provided\_data)$ then
3:     $K \leftarrow HMAC(K,\ V \ || \ flag)$
4:     $V \leftarrow HMAC(K,\ V)$
5: else
6:     $K \leftarrow HMAC(K,\ V \ || \ flag \ || \ provided\_data)$
7:     $V \leftarrow HMAC(K,\ V)$
8:     $flag \leftarrow' 0x01'$
9:     $K \leftarrow HMAC(K,\ V \ || \ flag \ || \ provided\_data)$
10:    $V \leftarrow HMAC(K,\ V)$
11: end if
12: return K, V
```

- **Generate Process**
  The generate process aims to create pseudorandom bits, where the input is *V*, *Key*, *reseed counter*, requested number of bits, and additional input. The first step of this process is to check whether reseed needs to be done or not. The reseed process is carried out if the reseed-counter value has exceeded the reseed interval. The second step of this process is to update the internal state if the additional input is not empty. In the third step , pseudorandom bits are generated by combining the values of *V* resulting from the hash process that is carried out repeatedly. The hash function is conducted on the value of *V* obtained in each iteration using the *K* as the hash key. This process stops when the length of the pseudorandom bit matches the requested number of bits. last step is to update the *V* and *K* values using HMAC-DRBG update function. *Reseed counter* increased by one. The algorithm from the generate process is shown in algorithm 6.

Algorithm 6 HMAC_DRBG_Generate

```
Input : V, Key, reseed_counter, requested_number_of_bits, additional_input
Output : pseudorandom_bits, Key, V, reseed_counter

1: if $reseed\_counter > reseed\_interval$ then
2:     $entropy\_input \leftarrow generateRandom(64bit)$
3:     $rc \leftarrow reseed\_counter$
4:     $ai \leftarrow addtional\_input$
5:     $(V, Key, reseed\_counter) \leftarrow HMAC\_DRBG\_Reseed(V,\ Key,\ rc,\ entropy\_input,\ ai)$
6: end if
7: if $isNotEmpty(additional\_input)$ then
8:     $(Key,\ V) \leftarrow HMAC\_DRBG\_Update(additional\_input,\ Key,\ V)$
9: end if
10: $temp \leftarrow empty()$
11: while $temp.length < requested\_number\_of\_bits$ do
12:     $V \leftarrow HMAC(Key,\ V)$
13:     $temp \leftarrow temp \ || \ V$
14: end while
15: $pseudorandom\_bits \leftarrow leftmost(temp, requested\_number\_of\_bits)$
16: $(Key, V) \leftarrow HMAC\_DRBG\_Update(ai,\ Key,\ V)$
17: $reseed\_counter \leftarrow rc + 1$
18: return pseudorandom_bits, Key, V, reseed_counter
```

- **Reseed Process**
  This stage is carried out for generating a new seed, where the inputs are *V*, *K*, *reseed counter*, entropy input, and additional input. The first, the construction of *seed* is carried out by concatenating the entropy input and additional input values. Then *V* and the *K* values are updated using the HMAC-DRBG update function based on the *seed*, *V*, and *K* values obtained from the input. Finally, the *reseed counter* is reset to 1. The algorithm of the reseed process is shown in algorithm 7.

Algorithm 7 HMAC_DRBG_Reseed

```
    Input : V, K, reseed_counter, entropy_input, additional_input
    Output : V, K, reseed_counter

1: $seed\_material \leftarrow entropy\_input||additional\_input$
2: $(K, V) \leftarrow HMAC\_DRBG\_update(seed\_material, K, V)$
3: $reseed\_counter \leftarrow 1$
4: return V, K, Reseed_counter
```

III. Research Method

The system built in this research consists of two sub-systems, namely the registration system and the login system. The registration system aims to register users on the system via email, password, and full name data and build an initial key, while the login system aims to provide access to the data contained in the learning management system by authenticating users. In the login system, a new key is also generated after a communication session is ended.

A. Registration Mechanism

During the registration process, users is asked to write their email, password, and full name on the registration page. In this case, an email can only be used for one user, then the system will check to make sure that the email used to register has not been used by another user. If the email is already used by another user then the registration process is canceled by the system. Meanwhile, if the email has never been used by another user, the registration process is continued by generating a key that is used to encrypt the password. The password encryption process is carried out using the Rijndael algorithm, while the key generation is carried out using the Hash Message Authentication Code - Deterministic Random Bit Generator (HMAC-DRBG) method. Finally, the email, the full name, the key for encrypting the password, and the encrypted password is stored in the learning management system database. The overall scheme of the registration process is shown in Figure 4.

The key generation used to encrypt the password is done using HMAC-DRBG instantiate and generate process. The process carried out is as follows:

- Generating Random Numbers using random module
- Calculating the entropy of a random number that has been generated. If the entropy is less than the determined security strength, then the random number cannot be used as a seed. If the entropy of the random number meets the requirements, then this random number can be used as a seed.
- Generating the *seed* can be done in two ways: by adding or without adding additional inputs.
- Initiating the value *K* = 0x00, *V*= 0x01, *reseed counter* = 0. Changing to a new value as a result of the update function (HMAC_DRBG_update(*seed*, *K*, *V*)) and changes the *reseed-counter* to 1.
- Generating the key using the function HMAC_DRBG_generate(*K*, *V*, *reseed counter*, *seed*, *requested number of bits* = 32, *additional input*). Furthermore, converting V and K to a new value as a result of the update function (HMAC_DRBG_update(*seed*, *K*, *V*)), and the *reseed counter* is increased by 1.

After the key is created, the password padding process is carried out by adding (128 - the length of the password in bit) bits of 0. The password encryption process is carried out using the Rijndael (AES)

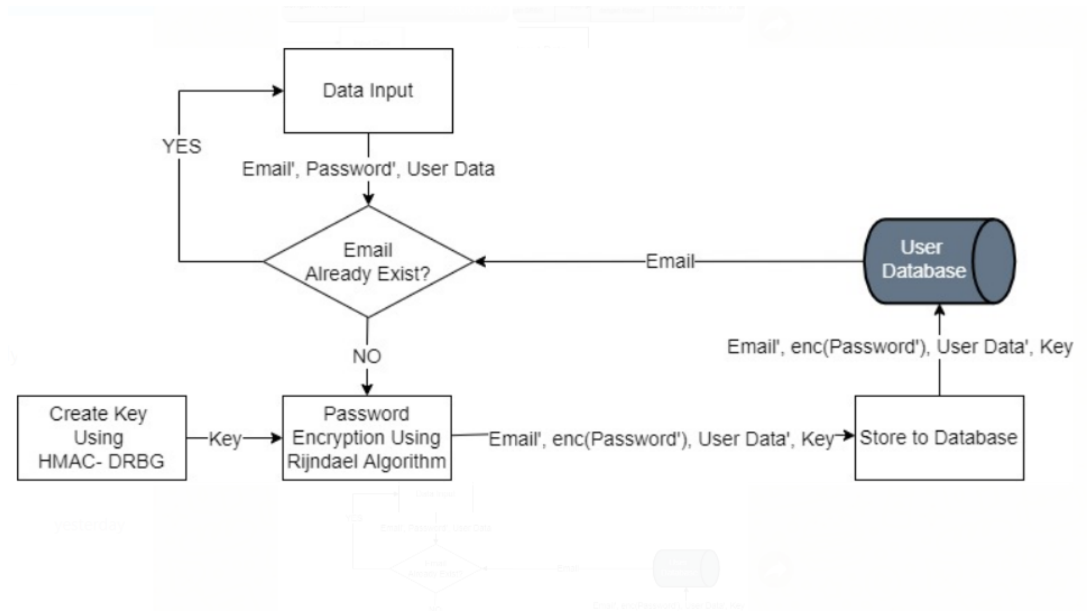
Data Input
YES
Email', Password', User Data
Email Already Exist?
Email
User Database
NO
Email', enc(Password'), User Data', Key
Create Key Using HMAC- DRBG
Key
Password Encryption Using Rijndael Algorithm
Email', enc(Password'), User Data', Key
Store to Database

Fig. 4. **User Registration Scheme**

algorithm as described in section 2.3. The encrypted password along with the email and full name is stored in the database. The complete registration algorithm is shown in algorithm 8.

Algorithm 8 Registration

```
    Input : email, password, name
 1: $user[\,][\,]$                    ▷ database user[email, password, name, encryption key]
 2: $found \leftarrow false$
 3: $i \leftarrow 0$
 4: while $i \le user.length$ do
 5:     if $user[\ i\ ][\ 0\ ] = email$ then
 6:         $found \leftarrow true$
 7:     end if
 8:     $i = i + 1$
 9: end while
10: if $found = false$ then
11:     $seed,\ K,\ V,\ reseed\_counter \leftarrow HMAC\_DRBG\_instantiate(entropy\_input)$
12:     $key,\ K,\ V,\ seed,\ reseed\_counter \leftarrow$
13:             $HMAC\_DRBG\_generate(K,\ V,\ reseed\_counter, seed,\ 32,\ additional\_input)$
14:     $encryptedPass \leftarrow AES(password,\ key)$
15: end if
```

B. Login Mechanism

During the login process, the user will be asked to enter his email and password on the login page. The system will check the existence of the user's email in the learning management system database. If the email is already stored in the database, it is necessary to check the validity of the password by comparing the results of the encrypted password written on the login page with the encrypted password stored in the database. If both are similar, then the user is allowed to access the data in the learning management system. If the email is not stored in the database, the user have to register first.

After accessing the data in the database, the system generates a new key using the HMAC_DRBG_generate function from the previous internal state. Finally, the password is encrypted using the new key and stored along with the name and email into the database, while the previous key and data will be deleted. The login scheme is shown in the image 5

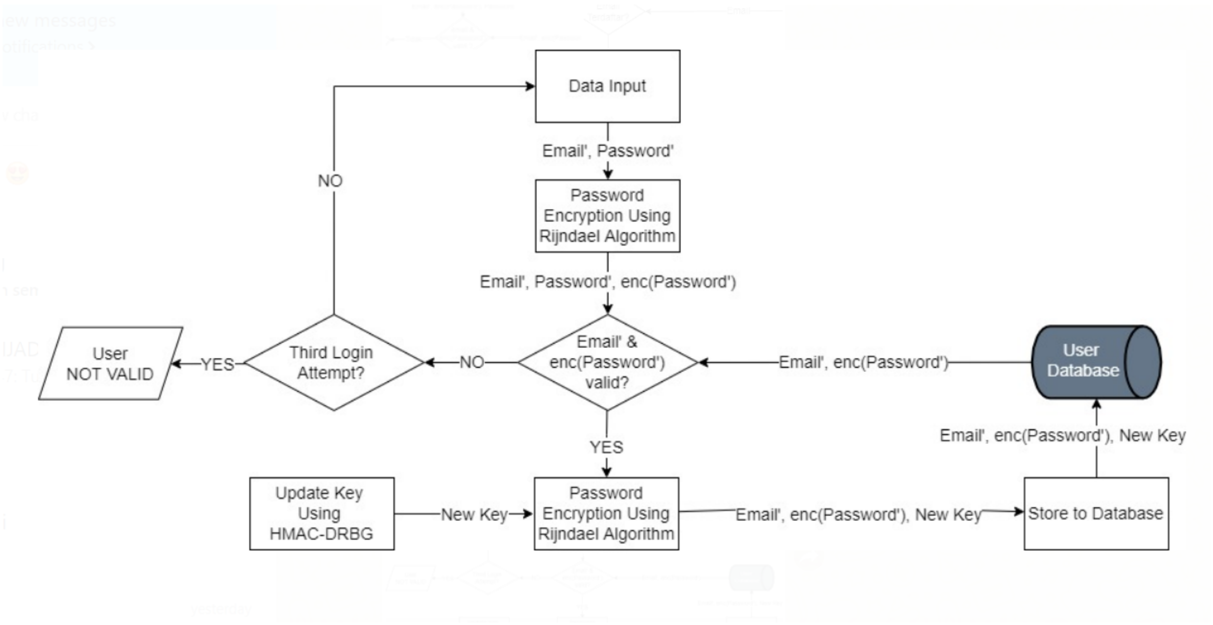
Data Input
Email', Password'
Password Encryption Using Rijndael Algorithm
Email', Password', enc(Password')
Email' & enc(Password') valid?
NO
Third Login Attempt?
YES
User NOT VALID
NO
Email', enc(Password')
User Database
YES
Update Key Using HMAC-DRBG
New Key
Password Encryption Using Rijndael Algorithm
Email', enc(Password'), New Key
Store to Database
Email', enc(Password'), New Key

Fig. 5. **User login Scheme**

IV. Results and Discussion

Evaluation of the proposed method is divided into two sub-sections, those are evaluation of performance and evaluation of security.

A. *Performance Analysis*

This section, discusses the time complexity required by the learning management system to encrypt passwords using the Rijndael algorithm and to generate keys that will be used in the encryption process. The time complexity required for this system is determined by the time complexity required for the password encryption process using the Rijndael algorithm and the time complexity required to generate the key using HMAC-DRBG. The time complexity required by DRBG depends on the time complexity required for SHA256, which is O(n). While the time complexity of password encryption using the Rijndael algorithm is O(n). Thus, the time complexity required by the learning management system to generate the key used in the password encryption process and to encrypt the password is O(n).

B. *Security Analysis*

In this section, discusses the resilience of the system against key guessing attacks. If the length of a key is m-bits, then there are 2^m possible keys that can be generated without any additional function. If an attacker attempts to attack this method by brute force key guessing for n times, then the probability of the attacker for getting the key can be calculated using equation (1).

$$P = \frac{n}{2^m} \tag{1}$$

Since this system uses a hash function,and the hash function has a collision, then there is a possibility of two different input values result in the same hash value. Thus, the probability of successful key guessing will depend on the hash function used. The hash function used is SHA256 which has a maximum limit of 64-bits input length, such that the probability of successful key guessing will depend on the number of experiments conducted and the maximum input length.

Collision attack is carried out for calculating a function h using x' in such a way that it can satisfy $h(x) = h(x')$, where $x'$is the guessed input. The attack method is done by to taking a random value of x' as much as q and then calculating $h(x')$. If the attacker tries to retrieve n random numbers of x'_i where $x'_i \in X$, $X = \{ x'_i \mid i = 1, 2, \ldots, n \}$, and the length of x' is y-bits. Then each value of x' will be used to calculate the function h. If $h(x) = h(x'_i)$ and $x \neq x'_i$, then x'_i is a collision of x. The stages of the attack are shown in algorithm 9.

Algorithm 9 BruteforceCollision

```
1: $result \leftarrow null$
2: $stop \leftarrow false$
3: $a_1 \leftarrow 0$
4: while $a_1 < 2$ do
5:     $a_2 \leftarrow 0$
6:     while $a_2 < 2$ do
7:         $a_3 \leftarrow 0$
8:         $\cdots$
9:         while $a_{n-1} < 2$ do
10:            $a_n \leftarrow 0$
11:            while $a_n < 2$ do
12:                $att \leftarrow a_1, a_2, \cdots, a_n$
13:                $hashAtt \leftarrow sha256(att)$
14:                if $key = hashAtt$ then
15:                    $stop \leftarrow true$
16:                    $result \leftarrow hashAtt$
17:                end if
18:                $a_n \leftarrow a_n + 1$
19:            end while
20:            $a_{n-1} \leftarrow a_{n-1} + 1$
21:        end while
22:        $\cdots$
23:        $a_2 \leftarrow a_2 + 1$
24:    end while
25:    $a_1 \leftarrow a_1 + 1$
26: end while
27: return $result$
```

By assuming that n is the number of experiments conducted by the attacker, and the input length of the hash function is $|x'|$, then the probability of succeed of the key guessing attack can be calculated using equation (2).

$$P_{prev} = \frac{n}{|x'|} \tag{2}$$

The proposed system uses the HMAC-DRBG method to build a dynamic key while the key depends on the seed used in HMAC-DRBG. Since the seed in the HMAC-DRBG changes periodically, the resilience of the system depends on the seed and the period of seed changes (reseed). If the seed changes period is q, the length of the seed x is m-bit, and there are n trials, then the probability to succeed guessing a seed (P_{prop}) that produces the original key can be calculated according to equation (3). The probability of guessing the seed is the same as the probability of guessing the key because the key generated by HMAC-DRBG is determined by the seed.

$$P_{prop} = \frac{n \mod q}{|x'|} \times \left(\frac{q}{|x'|}\right)^{\lfloor \frac{n}{q} \rfloor} \tag{3}$$

Based on equation (3), it is shown that the probability of success for guessing a key when using the proposed method is less than the probability of success for guessing a key when using the previous method proposed by Francis Onodueze et al. [1] (see equation (2)). Finally, it can be concluded that the use of HMAC-DRBG to generate keys in the Rijndael algorithm increases the security strength against key guessing attacks. The probability of success key guessing attack using the proposed method is also less than Liu's method, because the probability of success key guessing attack is greater than 1/1024.

Based on the attack method that has been previously described (brute force and collision attacks), the time complexity required for the attack will be calculated. The time complexity calculation is carried out using an attack algorithm which is shown further in Algorithm 9. Based on Algorithm 9, the time complexity required to attack the previous method proposed by Francis Onodueze et al. [1] is O(k^n), where n is the length of the input (*m*-bits) of the hash function and *k* is the number of possible character types used to represent the input. Meanwhile, the time complexity required to attack the proposed method is O($k^{m \cdot n}$) where *m* is the number of generated seeds. Finally, it can be concluded that the time complexity required to perform a successful key guessing attack when using the proposed method is greater than the time complexity required to perform a key guess attack when using the previous method that was proposed by Francis Onodueze et al. [1].

V. Conclusion

Learning management system serves to facilitate teaching and learning activities. User data stored in the learning management system database includes data that requires protection against access from unauthorized parties, such as cybercrime. To overcome this problem, an encryption process is used for data that is considered important. A password is used for securing and managing the access to the learning management system. Since the password is a sensitive user identity, the password needs to be encrypted. In the previous method proposed by Francis Onodueze et al. [1], Rijndael encryption was used. The key used to encrypt passwords using the Rijndael method is a static key such that the key guessing attacks can be easily carried out using brute force method. To increase the security against the key guessing attacks, a dynamic key generation method is proposed using HMAC-DRBG. Based on the results of the evaluation of the performance and safety of the previously proposed and currently proposed methods, it can be concluded that the time complexity for running the two methods is the same. Which is O(n). The time complexity to conduct a success key guessing attack when using the previous method is O(k^n), while the time complexity to conduct a success key guessing attack when using the proposed method is O(k^{mn}). The probability of success key guessing attack when using the previous method proposed by Francis Onodueze et al. [1] is greater than the probability of the success key guessing attack on the proposed method.

References

[1] Francis Onodueze and Sharad Sharma. Rijndael algorithm for database encryption on a course management system. *International Journal of Computers and their Applications, IJCA*, 24, 03 2017.

[2] Luca Baldanzi, Luca Crocetti, Francesco Falaschi, Matteo Bertolucci, Jacopo Belli, and Luca Fanucci. Cryptographically secure pseudo-random number generator ip-core based on sha2 algorithm. *Sensors*, 20:1869, 03 2020.

[3] Miguel Herrero-Collantes and Juan Carlos Garcia-Escartin. Quantum random number generators. *Reviews of Modern Physics*, 89, 04 2016.

[4] Zhi Liu and De Han. Dynamic encryption algorithm based on rijndael. *Advanced Materials Research*, 490-495:339–342, 03 2012.

[5] Katherine Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew Appel. Verified correctness and security of mbedtls hmac-drbg. 08 2017.

[6] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators (revised). *National Institute of Standards and Technology*, 01 2007.

[7] Mrs Yasmeen. Nosql database engines for big data management. *International Journal of Trend in Scientific Research and Development*, Volume-2:617–622, 10 2018.

[8] Than Myo Zaw, Min Thant, and S. V. Bezzateev. Database security with aes encryption, elliptic curve encryption and signature. In *2019 Wave Electronics and its Application in Information and Telecommunication Systems (WECONF)*, pages 1–6, 2019.

[9] AL-Saraireh J. An efficient approach for query processing over encrypted database. *Journal of Computer Science*, 13:548–557, 10 2017.

[10] Simanta Sarmah. Database security –threats prevention. *International Journal of Computer Trends and Technology*, 67:46–53, 05 2019.

[11] Neeraj Sharma and Mohammed Farik. A performance test on symmetric encryption algorithms-rc2 vs rijndael. *International Journal of Scientific Technology Research*, 6:292–294, 07 2017.

[12] Asmaa Ashoor. Enhancing performance of aes algorithm using concurrency and multithreading. 06 2019.