ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 5, Issue 1, pp: (66-104), Month: January - March 2017, Available at: www.researchpublish.com

Encrypting Critical Data in Databases

NAULESH KUMAR

PRINCIPAL, VIDYA MEMORIAL INSTITUTE OF TECHNOLOGY, RANCHI, INDIA

Abstract: **Today, it is imperative for an enterprise/organization to protect critical data from both internal and external threats and ensure for security. As the incidence and severity of security breaches continues to grow, it is increasingly incumbent upon organizations to begin encrypting data inside the enterprise. Ingrian offers breakthrough solutions that make it practical to encrypt critical data, and ensure it's secured throughout an organization.**

The Thesis is focus on how organizations can implement Ingrian DataSecure Platforms and employ them to encrypt critical data inside a database. Here I have discuss how critical data is encrypted in data base step by step by creating a table CUSTOMER by using SQL Query. With Ingrian DataSecure Platforms, organizations can protect critical data from both internal and external threats, and ensure compliance with legislative and policy mandates for security. Secure key management, backup, and administration are a few of the core elements for achieving true data privacy. Because Ingrian DataSecure Platforms have been built from the ground up with security in mind, implementing these policies and procedures is fast and easy, utilizing our unique management interface. We encourage administrators to focus on developing key management and administrative policies for their organizations that will provide maximum security and hope that this thesis will serve as a guide in this effort. The thesis specifies two cryptographic algorithms, the Data Encryption Standard (DES) and the Triple Data Encryption Algorithm (TDEA) which may be used by Federal organizations to protect sensitive data. Protection of data during transmission or while in storage may be necessary to maintain the confidentiality and integrity of the information represented by the data. The algorithms uniquely define the mathematical steps required to transform data into a cryptographic cipher and also to transform the cipher back to the original form. The Data Encryption Standard is being made available for use by Federal agencies within the context of a total security program consisting of physical security procedures, good information management practices, and computer system/network access controls.

Keywords: **Data encryption, Key management, Critical Data, Data Security, Algorithm, SQL Query.**

1. INTRODUCTION

Today, it is imperative for enterprises to know which specific threats they are trying to protect against and take stringent measures to address those threats. Ingrian Networks brings complete data privacy to the enterprise. With Ingrian Data Secure Platforms, organizations can protect critical data from both internal and external threats, and ensure compliance with legislative and policy mandates for security. Data Secure features a dedicated security appliance and specialized software that enables organizations to encrypt critical data in applications and databases.

File level encryption within databases provides protection largely against media theft and loss. Column level encryption provides protection against media theft along with a broad range of logical threats. While protecting against identified threats, enterprises must consider the type of encryption to deploy to protect sensitive data in databases, while minimizing performance impact, ensuring strong, efficient AAA and key management, and minimizing the impact of batch type processes. Effectively addressing all these factors will ensure enterprises can protect sensitive data, while minimizing the overall cost to the business.

Storage level encryption within databases provides protection largely against media theft and loss. Column level encryption provides protection against media theft along with a broad range of logical threats. While protecting against identified threats enterprises must consider the type of encryption to deploy to protect databases, performance impact, AAA and key management, and impact to batch type processes. These considerations along with execution on best practices for key management and security will ensure enterprises that their sensitive data is well protected and controlled within their environment.

Ingrian is a privately held company backed by such investors as Globespan Capital Partners (formerly JAFCO Ventures), Prism Venture Partners, HighBAR Ventures, and Partech International.

Database Integration Process:

As the incidence and severity of security breaches continues to grow, it is increasingly incumbent upon organizations to begin encrypting data inside the enterprise. Ingrian offers breakthrough solutions that make it practical to encrypt critical data, and ensure it's secured throughout an organization. With Ingrian DataSecure Platforms, organizations can better ensure that they are compliant with legislative and policy mandates for security, and eliminate the risks of a breach. Ingrian DataSecure Platforms deliver comprehensive security capabilities:

- Encrypt critical data in Web servers, application servers, and databases.
- .Ensure all access to critical data is carefully managed, logged, and controlled.
- Administer keys and policies in a secure, centralized fashion.
- Ensure security processing is highly scalable and reliable.

DataSecure works seamlessly with leading databases—including IBM DB2, Microsoft SQL Server, and Oracle—delivering capabilities for securely and efficiently managing encryption. DataSecure encrypts data in the database at the column level, and can be used to secure such information as credit card numbers, social security numbers, passwords, account balances, and email addresses. DataSecure significantly streamlines the administrative tasks involved in database encryption—it automates much of the configuration and implementation process and it can be deployed without any disruption to the applications tied to the database. The Ingrian DataSecure Platform is comprised of three components:

- The Data Secure appliance, a dedicated hardware system,
- The Network-Attached Encryption™ (NAE) Server, which runs on the DataSecure appliance, and
- The Ingrian NAE Connector, software that is installed on the Web, application, or database server.

The Ingrian NAE Connector features standards-based cryptographic interfaces that allow the protection of user-defined data through integration of security functions at the business logic layer. These small software components are installed on each database that has a need to interface with the Ingrian appliance, and they initiate encrypt and decrypt operations..

1.1 How it Works:

Integrating the DataSecure platform in your existing database infrastructure is a straightforward, automated process. The NAE Connector component outlined above consists of complete code to manage a seamless interaction between the database and the DataSecure platform. The DataSecure appliance features a secure, Web-based user interface that walks administrators through a step-by-step configuration process and, once parameters have been set, even automates the installation of the required NAE Connector software on the database. Once installed and configured, the NAE Connector dynamically generates all the necessary stored procedures and functions to:

- Encrypt and decrypt data on demand from inside the database.
- Migrate data from plaintext to ciphertext and change the database schema to accommodate encrypted columns.
- Rotate cryptographic keys.
- Automate subsequent encrypt and decrypt operations.
- Authenticate users so that only authorized users are able to access sensitive data.

This transparent integration means that you can continue using your existing SQL statements without having to modify them. And, more importantly, you do not have to write any of the logic to perform encrypt or decrypt operations from your database. Following is a high-level diagram outlining how the solution is deployed.

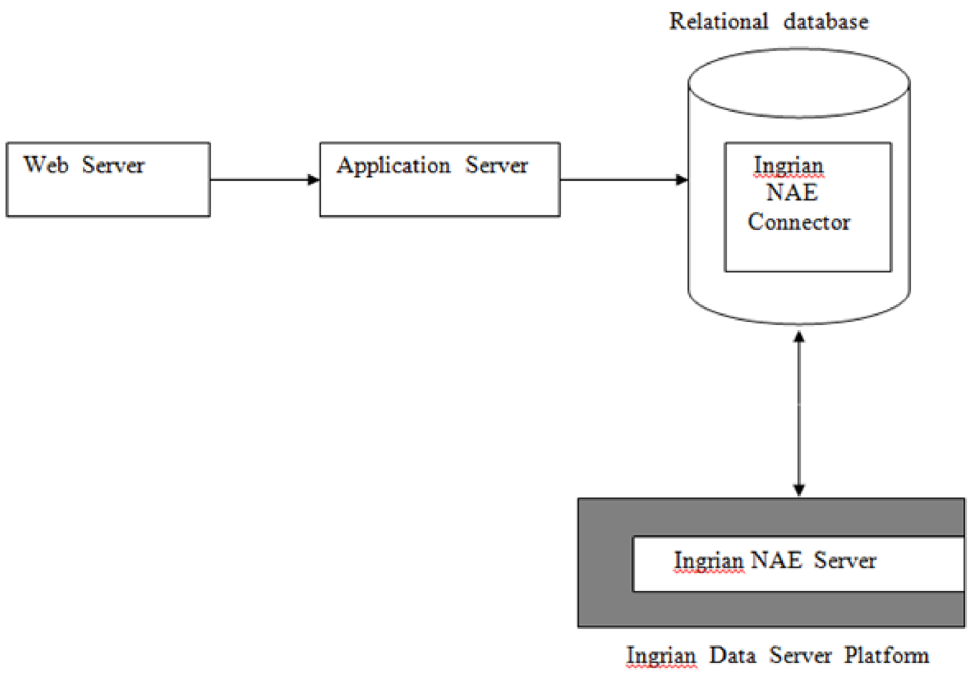


Figure 1 : High Level View of Implementation with Ingrian Data Secure Plateform

The diagram above shows a Web server accessing an application server, which is making a call to a database to access sensitive data. The NAE Connector is installed in the database and the sensitive data is stored in encrypted format within the database. The user who requests the sensitive data must have permission within the database to make a request to the NAE Connector. That user must also have access to the requested key on the NAE Server. If either of the conditions above is not met, then the user is not given access to the sensitive data. If the user is authorized to use the requested key for decryption, then the NAE Server performs the decrypt operation on the sensitive data. The decrypted data is then passed back to the database. The DataSecure appliance is the physical device where cryptographic operations and key management operations are performed. Different hardware platforms provide varying capabilities for performance and FIPS compliance. The NAE Server and all cryptographic keys reside on this hardened security system.

1.2 How Does Data Get Encrypted?

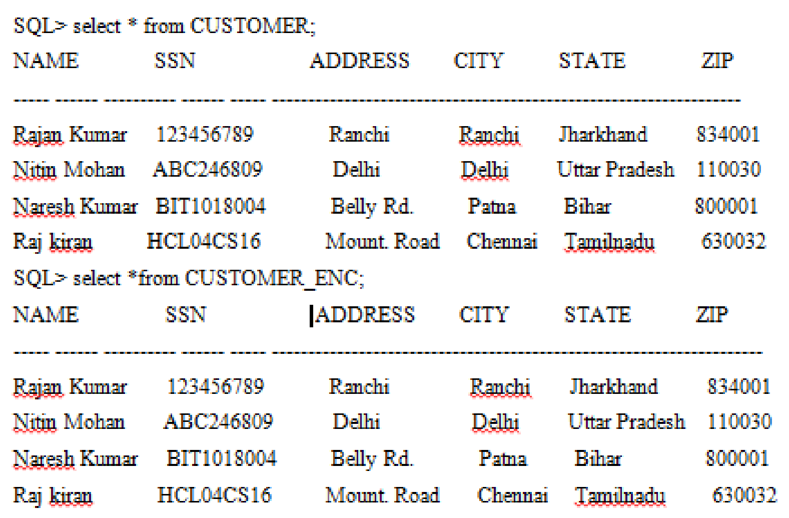
Before implementing DataSecure in your enterprise, your sensitive data is most likely being stored as plaintext, so the logical question is: how do you migrate plaintext data into encrypted format? The process is straightforward, and, as mentioned above, Ingrian automates this process through the DataSecure appliance's GUI. To illustrate the simplicity of the process, take social security numbers as an example. If you have a table called CUSTOMER that stores sensitive customer data like names, addresses, and social security numbers, you might want to encrypt the social security numbers. Your original CUSTOMER table might look like this:

Table 1.1: Sensitive Data Is Stored In the Column SSN

```
SQL> select * from CUSTOMER;
NAME          SSN           ADDRESS      CITY      STATE          ZIP
----- ------ ---------- ------ ----- ------------------------------------------------------------------
Rajan Kumar   123456789     Ranchi       Ranchi    Jharkhand      834001
Nitin Mohan   ABC246809     Delhi        Delhi     Uttar Pradesh  110030
Naresh Kumar  BIT1018004    Belly Road.  Patna     Bihar          800001
Raj kiran     HCL04CS16     Mount. Road  Chennai   Tamilnadu      630032
```

The first step in the process of securing your sensitive data is to identify what data you want to secure and where that data resides. In this example, social security numbers are stored in a column called SSN. Once you have identified the sensitive data, you can configure Ingrian to automate this data protection process. During the first step, Ingrian renames the table in which the sensitive data resides; the table must be renamed so that a view can be created later with the same name as the original table. Notice in Figure 3 that the table is renamed to CUSTOMER_ENC, but the SSN column is not yet changed.

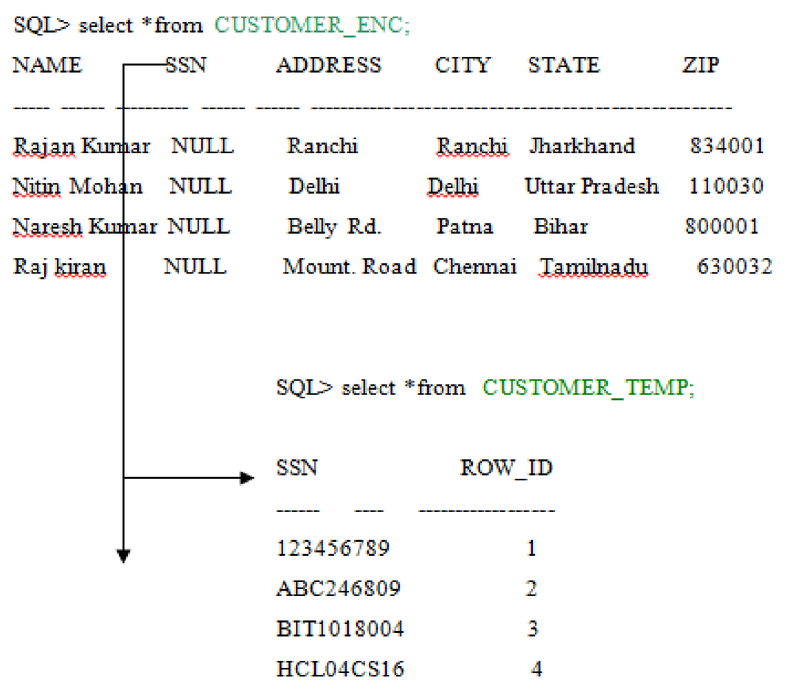
Table 1.2 : ***Rename CUSTOMER Table***



```
SQL> select * from CUSTOMER;
NAME          SSN            ADDRESS      CITY      STATE          ZIP
----- ------ ---------- ------ ----- -------------------------------------------------------------
Rajan Kumar   123456789      Ranchi       Ranchi    Jharkhand      834001
Nitin Mohan   ABC246809      Delhi        Delhi     Uttar Pradesh  110030
Naresh Kumar  BIT1018004     Belly Rd.    Patna     Bihar          800001
Raj kiran     HCL04CS16      Mount. Road  Chennai   Tamilnadu      630032
SQL> select *from CUSTOMER_ENC;
NAME          SSN            |ADDRESS     CITY      STATE          ZIP
----- ------ ---------- ------ ----- ----------------------------------------------------------------
Rajan Kumar   123456789      Ranchi       Ranchi    Jharkhand      834001
Nitin Mohan   ABC246809      Delhi        Delhi     Uttar Pradesh  110030
Naresh Kumar  BIT1018004     Belly Rd.    Patna     Bihar          800001
Raj kiran     HCL04CS16      Mount. Road  Chennai   Tamilnadu      630032
```

In the next step, Ingrian creates a temporary table and exports the sensitive data to that temporary table. Notice in Figure 4 below that SSN is the only column exported to the temporary table from the original table. The Row_ID column is added automatically and used later when returning the encrypted data back to the original table. The values in the column that held the sensitive data in the CUSTOMER_ENC table (remember—the CUSTOMER table was renamed) are set to null to avoid any data conversion issues that might arise when changing the data type in a later step.

Table 1.3: Export Sensitive Data to Temporary Table



```
SQL> select *from CUSTOMER_ENC;
NAME          SSN       ADDRESS      CITY      STATE          ZIP
----- ------ --------- ------ ------ -------------------------------------------------------
Rajan Kumar   NULL      Ranchi       Ranchi    Jharkhand      834001
Nitin Mohan   NULL      Delhi        Delhi     Uttar Pradesh  110030
Naresh Kumar  NULL      Belly Rd.    Patna     Bihar          800001
Raj kiran     NULL      Mount. Road  Chennai   Tamilnadu      630032


                SQL> select *from CUSTOMER_TEMP;

                SSN              ROW_ID
                ------  ----   ------------------
                123456789            1
                ABC246809            2
                BIT1018004           3
                HCL04CS16            4
```

Before Ingrian can populate the original column with encrypted data, it must modify the column size and data type because encrypted data is predictably larger than plaintext data, and most likely, your social security numbers are stored as some sort of integer or character data type. Although Ingrian gives you the option to store your encrypted data in Base64 encoded format, it is recommended that you store your encrypted data in binary (the default choice during configuration) because binary data is smaller than Base64 encoded data and there is less overhead with binary because the system does not have to encode and decode data with every encrypt or decrypt operation.

Table 1. 4: Modify Data Type and Column Size of the Encrypted Table

```
SQL> desc CUSTOMER;
Name                          Null?    Type
----------------------------- -------- ----
NAME                                   VARCHAR2(5)
SSN                                    CHAR(9)
ADDRESS                                VARCHAR2(10)
CITY                                   CHAR(6)
STATE                                  CHAR(5)
ZIP                                    NUMBER(6)
SQL>  desc CUSTOMER_ENC;
Name                          Null?    Type
----------------------------- -------- ----
NAME                                   VARCHAR2(5)
SSN                                    VARCHAR2(16)
ADDRESS                                VARCHAR2(10)
CITY                                   CHAR(6)
STATE                                  CHAR(5)
ZIP                                    NUMBER(6)
```

Once the column is modified, Ingrian can migrate the sensitive data back into the CUSTOMER_ENC table. Before the data is imported back into the CUSTOMER_ENC table, however, the NAE Connector sends the data to the NAE Server, where the data is encrypted. The NAE Server returns the encrypted data to the NAE Connector, which then inserts the encrypted data into the CUSTOMER_ENC table.

Table 1.5 : Encrypt Data and Insert into CUSTOMER_ENC Table

```
SQL> select *from  CUSTOMER_TEMP;
SSN          ROW_ID
------  ----  -----
123456789       1
ABC246809       2
 BIT1018004     3
HCL04CS16       4
```

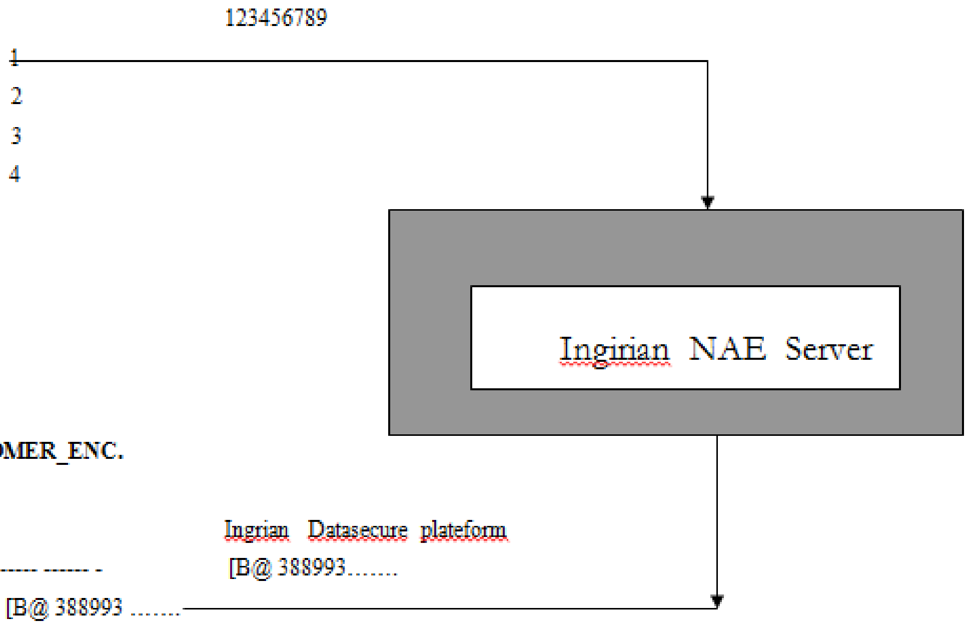


Table 1.6 : CUSTOMER_ENC.

```
NAME      SSN
----- ------ ---------- ------ ------ -
Rajan kumar     [B@ 388993 .......
Nitin Mohan     [B@ b8f82d  .......
Naresh Kumar    [B@ 1d04653  ......
Raj kiran       [B@b8f82d  ......
```

After encryption, the CUSTOMER_TEMP table is dropped and the CUSTOMER_ENC table might look something like this:

Table 1.7 : Sensitive Data Is Stored in Encrypted Format

SQL> select *from CUSTOMER_ENC;

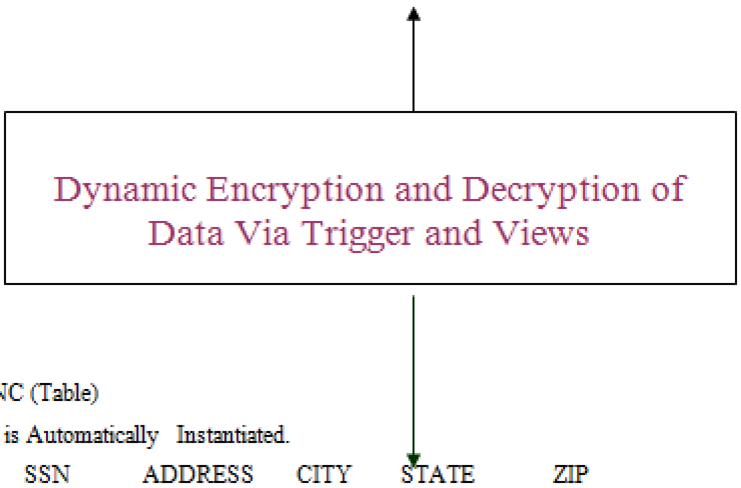
NAME	SSN	ADDRESS	CITY	STATE	ZIP
Rajan Kumar	[B@ 388993	Ranchi	Ranchi	Jharkhand	834001
Nitin Mohan	[B@ b8f82d	Delhi	Delhi	Uttar Pradesh	110030
Naresh Kumar	[B@ 1d04653	Belly Rd.	Patna	Bihar	800001
Raj kiran	[B@b8f82d	Mount. Road	Chennai	Tamilnadu	630032

1.3 How Do You Automate Subsequent Updates and Inserts?

Once your table and column are able to accommodate encrypted data, Ingrian automates encryption and decryption of data by creating a view, triggers, and stored procedures that are generated during configuration to work with the NAE Connector. In this way, properly authenticated applications outside the database can continue to query and update the same database tables as before. The NAE Connector remains transparent to outside applications, and, more importantly, the amount of code changes necessary to integrate the NAE Connector is minimal.

CUSTOMER (View)

NAME	SSN	ADDRESS	CITY	STATE	ZIP
Rajan kumar	123456789	Ranchi	Ranchi	Jharkhand	835220
Nitin Mohan	ABC246809	Delhi	Delhi	Uttar Pradesh	110030
Naresh Kumar	BIT1018-04	Belly Rd.	Patna	Bihar	800001
Raj kiran	HCL04CS16	Mount. Road	Chennai	Tamilnadu	630032


Dynamic Encryption and Decryption of Data Via Trigger and Views

CUSTOMER_ENC (Table)

Table 1.7 : View is Automatically Instantiated.

NAME	SSN	ADDRESS	CITY	STATE	ZIP
Ranjan Kumar	[B@ 388993	Ranchi	Ranchi	Jharkhand	835220
Nitin Mohan	[B@ b8f82d	Delhi	Delhi	Uttar Pradesh	110030
NareshKumar	[B@ 1d04653	Belly Rd.	Patna	Bihar	800001
Raj kiran	[B@b8f82d	kakinara	Andhra	Andhra	630032

As you can see from Figure 8 above, when sensitive data is accessed, the view is instantiated by the database and populated with decrypted data from the CUSTOMER_ENC table. Because the view has the same name as the original table, all SQL statements that reference the encrypted data can function regularly without modification. Likewise, triggers trap all the inserts and updates executed on the view. If an insert statement is detected, a new insert statement is generated based on the original insert values. The social security number in this case is encrypted before insertion into the base table. Similarly, when an update statement is executed, a new update statement is generated to update the base table (CUSTOMER_ENC) as opposed to the view referenced in the call from the outside application (CUSTOMER).

As was mentioned above, the process to migrate plaintext data to encrypted format is quite simple when using the NAE Connector; furthermore the process can be completely automated through the use of triggers, views, and stored procedures, all of which are created and

Installed by the DataSecure appliance during configuration. What's most important is that the integration is completely transparent to applications that interface with your sensitive data. Before deploying Data Secure, your sensitive data sits in the clear in your databases.

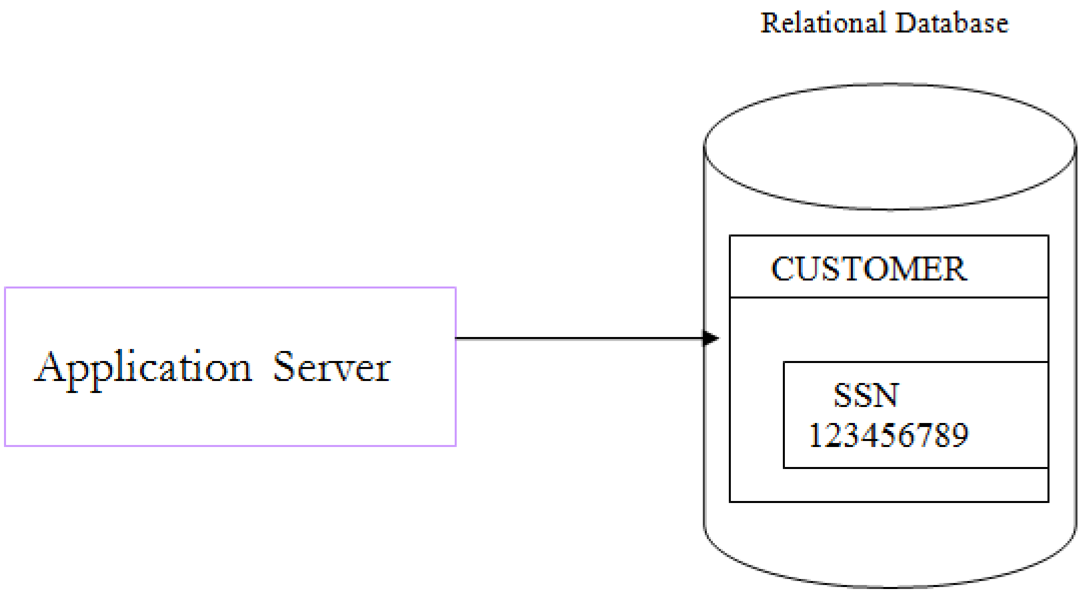


Figure 1.2: App Server Interacts with Plaintext Data Before Deploying DataSecure

After deploying DataSecure, your sensitive data is encrypted, and applications can continue interacting with sensitive data using the same SQL statements; however, instead of interacting directly with that sensitive data, the application servers are actually interacting with a view of the data.

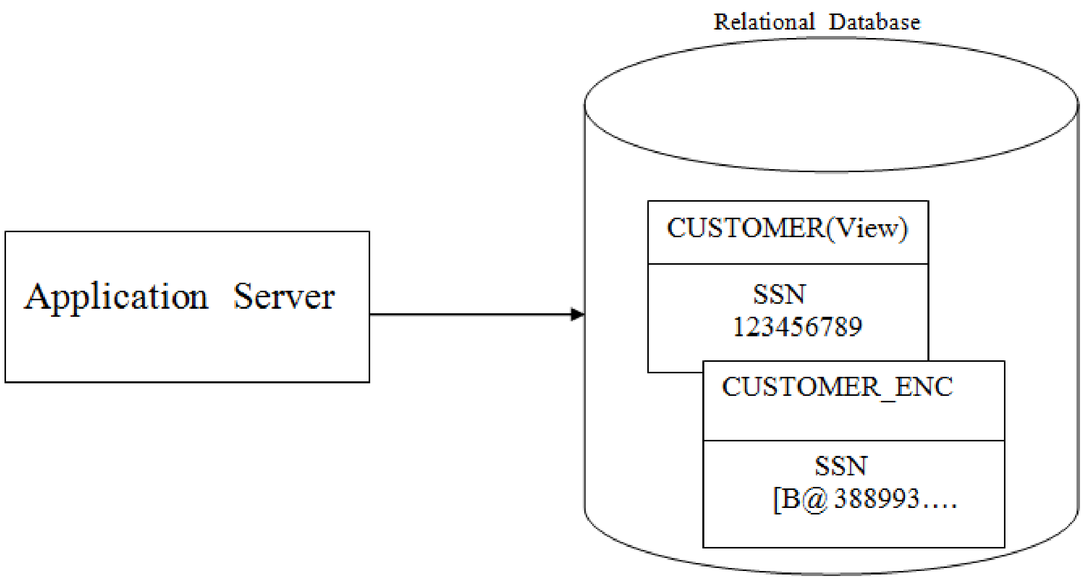
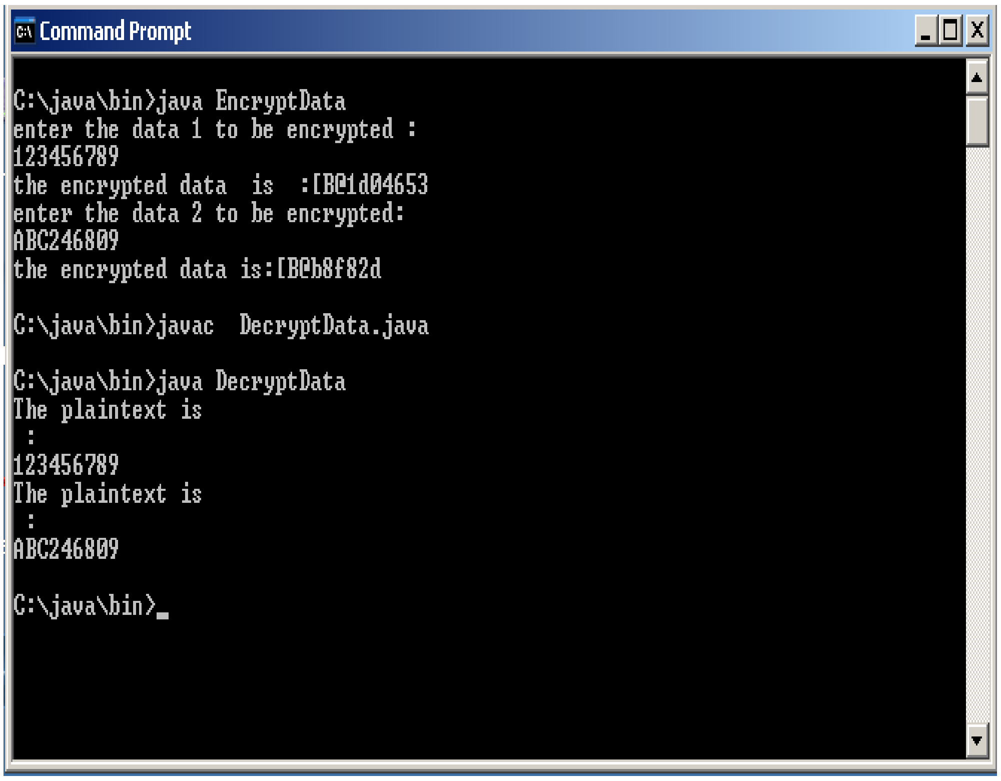


Figure 1.3 : App Server Interacts with Plaintext View of Encrypted Data After Deploying Data Secure.

1.4 Output of Java Programm:

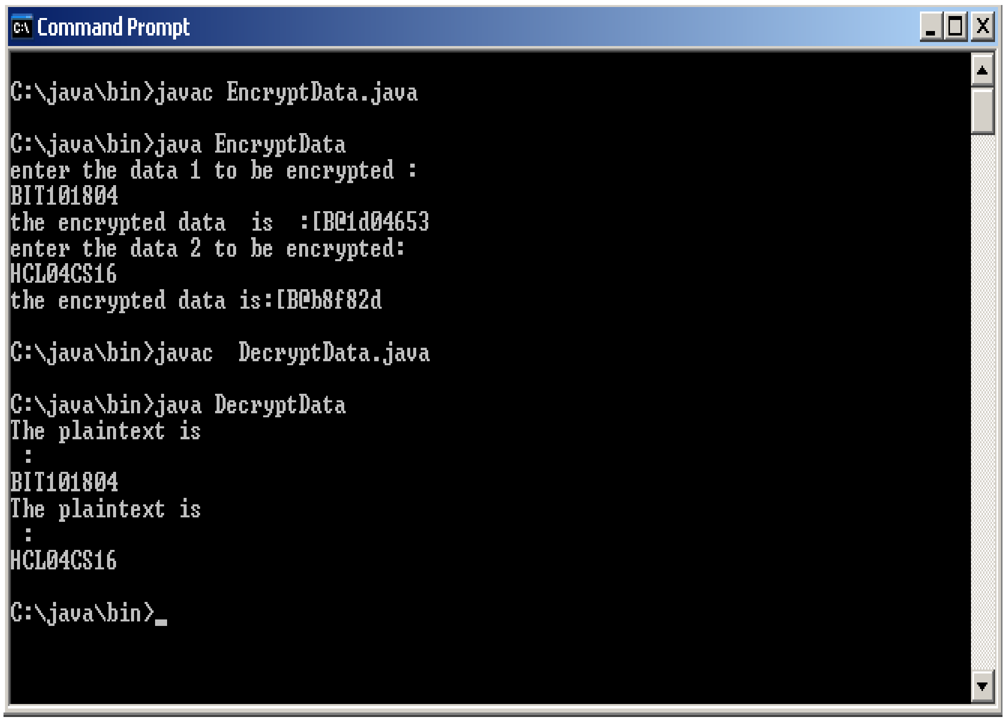
Output of the java program is shown bellow in Command Prompt which is written in Appendix B and Appendix C for Encryption and Decryption of any data.

Output of the Column SSN , Encrypted by DESede Encryptin Algorithm in CBC mode in shown on the Command Prompt screen .



```
C:\java\bin>java EncryptData
enter the data 1 to be encrypted :
123456789
the encrypted data  is  :[B@1d04653
enter the data 2 to be encrypted:
ABC246809
the encrypted data is:[B@b8f82d

C:\java\bin>javac  DecryptData.java

C:\java\bin>java DecryptData
The plaintext is
 :
123456789
The plaintext is
 :
ABC246809

C:\java\bin>_
```

Figure 1.4 a : Output of the data (SSN) in Encrypted and Decrypted form.



```
C:\java\bin>javac EncryptData.java

C:\java\bin>java EncryptData
enter the data 1 to be encrypted :
BIT101804
the encrypted data  is  :[B@1d04653
enter the data 2 to be encrypted:
HCL04CS16
the encrypted data is:[B@b8f82d

C:\java\bin>javac  DecryptData.java

C:\java\bin>java DecryptData
The plaintext is
 :
BIT101804
The plaintext is
 :
HCL04CS16

C:\java\bin>_
```

Figure 1.4 b : Output of the data (SSN) in Encrypted and Decrypted form.

2. DATA PRIVACY IN THE ENTERPRISE

For Implementing Encryption:

In spite of a range of security technologies being deployed, devastating thefts of sensitive data continue to occur. To address these threats, many organizations are looking to deploy data privacy solutions—solutions that ensure the security of data inside the enterprise. The thesis offers a detailed look at the best practices for achieving data privacy, outlining the key building blocks of a data privacy implementation, and the best practices for each of these areas.

Enterprises worldwide are spending approximately $40 billion per year on IT security, yet very costly breaches continue to occur. In large part, this is because security efforts have mainly been focused on network security rather than data privacy. Data privacy is the process of securing critical data as it is being stored, transmitted, and used within the enterprise. Failure to implement a data privacy solution can have a disastrous effect on an organization. Public disclosure of breaches can be catastrophic to an organization's brand, market capitalization, and consumer trust. Plus, privacy

legislation and the security policies of credit card issuers alike mandate disclosure of breaches, meaning organizations that try to keep a breach secret will be susceptible to civil litigation and steep fines.

A comprehensive data privacy solution must address security across the whole enterprise. Often, organizations deploy security systems to encrypt data in transport between machines, but then have the data stored in the clear. Likewise, many enterprises have already purchased multiple highly secure and fault tolerant systems, but security weaknesses can still be present due to the way these systems interface with each other. To achieve data privacy, it is important to think about the security of applications, databases, and the infrastructure that supports them as a whole. This document offers an overview of how to address data privacy in a comprehensive fashion, outlining the key building blocks of a data privacy implementation, and offering detailed guidance for each of these areas. According to Gartner, 75% of external-based attacks are tunneling through applications— and so go undetected by a range of traditional security mechanisms.

2.1 Encryption:

2.1.1 Cryptographic Algorithms:

This section provides a list of recommendations for choosing among the various cryptographic operations available in implementing a data privacy solution.

DES

Although DES has enjoyed widespread popularity for many years, it is no longer considered secure enough by today's standards. DES should generally not be used in production environments.

Key length : 64 bits as encoded; 56 bits excluding parity bits.

Block size : 8 bytes.

Security comment:

The fixed 56-bit effective key length is too short to prevent brute-force attacks

3DES

3DES is a more secure alternative to DES and is a widely used symmetric algorithm. If possible, AES should be used rather than 3DES due to the performance advantages offered by AES

DESede

Designers : Whitfield Diffie, Martin Hellman, Walt Tuchmann

Published : 1978-79

Key length **:** 128 or 192 bits, as encoded (112 or 168 bits excluding parity). The default key length depends on the name of the KeyGenerator: 128 bits for DES-EDE2, and 192 bits for DES-EDE3 .

The default key length for DESede and the other aliases is implemented inconsistently between different providers, and therefore if an application needs to create a specific length of DESede key in a way that is guaranteed to work across providers, it should explicitly create a SecretKeySpec.

Block size: 8 bytes.

Comments:

If the key length is 128 bits including parity (i.e. two-key triple DES), the first 8 bytes of the encoding represent the key used for the two outer DES operations, and the second 8 bytes represent the key used for the inner DES operation.

If the key length is 192 bits including parity (i.e. three-key triple DES), then three independent DES keys are represented, in the order in which they are used for encryption.

2.2 Additional Cryptographic Options:

2.2.1 Modes and Ivs:

Block encryption algorithms (such as DES and AES) can be used in a number of different modes, such as “electronic code book” (ECB) and “cipher block chaining” (CBC). In nearly all cases, CBC mode is recommended over ECB mode. ECB mode can be less secure because the same block of plaintext data always results in the same block of ciphertext, a property that can be used by an attacker to reveal information about the original data and to tamper with the encrypted data. Many modes, including CBC mode, require an “initialization vector” (IV), which is a sequence of random bytes used as input to the algorithm along with the plaintext The IV does not need to be secret, but it should be unpredictable.

ECB:

This is the electronic cookbook mode. ECB is the simplest of all modes; it takes a simple block of data (8 bytes in the SunJCE implementation, which is standard) and encrypts the entire block at once. No attempt is made to hide patterns in the data, and the blocks may be rearranged without affecting decryption (though the resulting plaintext will be out of order). Because of these limitations, ECB is recommended only for binary data; text or other data with patterns in it is not well-suited for this mode. ECB mode can only operate on full blocks of data, so it is generally used with a padding scheme. ECB mode does not require an initialization vector.

CBC:

This is the cipher block chaining mode. In this mode, input from one block of data is used to modify the encryption of the next block of data; this helps to hide patterns (although data that contains identical initial text--such as mail messages--will still show an initial pattern). As a result, this mode is suitable for text data. CBC mode can only operate on full blocks of data (8-byte blocks in the SunJCE implementation), so it is generally used with a padding scheme. CBC mode requires an initialization vector for decryption.

CFB:

This is the cipher-feedback mode. This mode is very similar to CBC, but its internal implementation is slightly different. CBC requires a full block (8 bytes) of data to begin its encryption, while CFB can begin encryption with a smaller amount of data. So this mode is suitable for encrypting text, especially when that text may need to be processed a character at a time. By default, CFB mode operates on 8-byte (64-bit) blocks, but you may append a number of bits after CFB (e.g., CFB8) to specify a different number of bits on which the mode should operate. This number must be a multiple of 8.

CFB requires that the data be padded so that it fills a complete block. Since that size may vary, the padding scheme that is used with it must vary as well. For CFB8, no padding is required, since data is always fed in an integral number of bytes. CFB mode requires an initialization vector for decryption.

OFB:

This is the output-feedback mode. This mode is also suitable for text; it is used most often when there is a possibility that bits of the encrypted data may be altered in transit (e.g., over a noisy modem). While a 1-bit error would cause an entire block of data to be lost in the other modes, it only causes a loss of 1 bit in this mode. By default, OFB mode operates on 8-byte (64-bit) blocks, but you may append a number of bits after OFB (e.g., OFB8) to specify a different number of bits on which the mode should operate. This number must be a multiple of 8. OFB requires that the data be padded so that it fills a complete block. Since that size may vary, the padding scheme that is used with it must vary as well. For OFB8, no padding is required, since data is always fed in an integral number of bytes. OFB mode requires an initialization vector for decryption..

Initialization Vectors:

When using CBC mode of a block encryption algorithm, a randomly generated initialization vector is used and must be stored for future use when the data is decrypted. Since the IV does not need to be kept secret it can be stored in the database. If the application requires having

an IV per column, which can be necessary to allow for searching within that column, the value can be stored in a separate table. For a more secure deployment, but with limited searching capabilities, an additional column can be added to the table and an IV can be generated per row. In the case where multiple columns are encrypted, but the table has space limitations, the same IV can be reused for each encrypted value in the row, even if the encryption keys for each column are different, provided the encryption algorithm and key size are the same.

Padding:

Symmetric block algorithms, such as DES and AES, have various padding options. Two common padding options are no padding and PKCS5. When no padding is used, data to be encrypted must be exactly a multiple of the block size. For example, if AES-128 is used, the data must be a multiple of 16 bytes. No padding can be advantageous in situations where the size of the data is fixed since the resulting ciphertext will be exactly the same size as the plaintext (i.e., there will be no additional space requirements to store the encrypted data). In situations where the size of the data can vary or is not a multiple of the block size, PKCS5 padding is recommended. This allows data of arbitrary size to be encrypted with the resulting ciphertext at most a block size larger than the plaintext. For example, with AES-128 if between 0 and 15 bytes of data need to be encrypted the resulting ciphertext will be 16 bytes. If between 16 and 31 bytes of data need to be encrypted, the resulting ciphertext will be 32 bytes.

Key Management:

Key management is a fundamental consideration when deploying a data privacy solution. If the keys used to protect sensitive data within an enterprise are not properly secured, attackers may be able to gain access to this data with relative ease. This section discusses some of the considerations when managing keys in the context of enterprise data privacy.

2.3 Centralizing Storage and Administration:

In a highly secure environment it is important to generate and manage keys in a centralized manner in which strict access privileges are enforced. For example, keys stored across multiple application server and database hard drives are significantly more difficult to manage and protect than keys stored on a centralized platform.

Hardware:

A specialized hardware device in which all cryptographic operations are performed securely and in which keys are never visible in the clear is highly recommended. This provides a significantly higher level of security over a pure software solution in which keys are managed and used in the clear. Some highly specialized hardware can provide a level of tamper resistance, so that, if an attempt is made to compromise the keys, the hardware will clear all information including the keys. This type of hardware solution is recommended for enterprises that require an extremely high level of security.

Import:

Importing a key into a secure key management system is not recommended since the system has no way of verifying where the key has existed prior to the import or even if the key has been compromised. If key import is a requirement, the history of the key should be well documented, and all copies of the key should be managed carefully.

Export:

Exporting a key from a secure management system is not recommended since the system will have no means of verifying how the key is used once it leaves the secure environment. If key export is a requirement, exported copies of the key should be managed carefully.

Rotation:

It is good practice to protect data with newly generated keys periodically. Re-encrypting data with a new key at least once a year is recommended. An important consideration when rotating keys is managing backups and archives. An enterprise must be able to ensure that sensitive data cannot be compromised through the use of old keys and archived data, while also being able to guarantee access to this data if necessary. If the keys used to protect sensitive data within an enterprise are not properly secured, attackers may be able to gain access to this data with relative ease.

2.4 Authentication, Authorization, and Auditing:

Enterprises need a secure way to identify people and entities that require access to sensitive data. In implementing a solution, administrators need to decide what data will be accessible and who will have access to it. Some methods of access control are passwords, client certificates, biometrics, and tokens.

Authentication:

Authentication ensures that an entity is really what or who it says it is. Methods of authentication can be classified as what you know, what you have, or what you are. The traditional "what you know" form of authentication, a username and password, is the least secure. Password can be given away, guessed, or stolen. "What you have" is called a token such as a client certificate or a smart card. "What you are" can be proven by recording and comparing a voiceprint, fingerprint, retinal scan, or even DNA. In general, the more factors used for authentication, the stronger the authentication is. For example, a system that uses a username and password along with a client certificate provides a higher level of security than one that only uses a username and password.

Authorization:

Authorization ensures that only the entities who should have access to resources obtain that access. In order for authorization to be granted to an entity, it must first be authenticated. Two of the authorization methods available are role based security and subject/object access control. Role-based security means that authorization is defined based on an entity's responsibilities. For example, an enterprise may choose to create an "application" role that only has authorization to encrypt credit card numbers and a "processing" role that only has authorization to decrypt the same information. In a subject/object access control system, every resource ("object") has an explicit list of entities ("subjects") that are allowed access. The least privilege security principle states that entities should only be granted the absolutely minimal set of privileges necessary to perform their tasks. Additional unnecessary authorization privileges only increase the vulnerability of the system to attack. It is important to be wary of "access creep", where employees are granted more and more authorization over time by special request. It is also critical to remember to change an employee's role when he or she changes jobs.

Auditing:

Auditing is an extremely important part of a data privacy solution. It allows the enterprise determine who did what at any given point in time, including when authentication and authorization were allowed or denied to an entity. A data privacy solution should offer robust logging capabilities and support log signing, in order to prevent an attacker from tampering with logs. Logs should be analyzed regularly to look for strange behavior that could potentially represent attacks on the enterprise.

2.5 Network and Transport:

Transport:

It is recommended to use SSL at all points in which sensitive data is in transit, both over the Internet and within the enterprise (such as between the application server and the database). A good data privacy solution will allow for secure transmission of sensitive data between all entities across an enterprise.

Firewall:

In general, all administrative ports should be blocked from external networks. Access control lists (ACLs) should be set up to restrict access to certain devices on the network. For example, if a network-based security appliance resides on the network, only those devices that require access for administrative or cryptographic operations should be granted access.

2.6 Encryption of Multiple Columns:

If multiple columns of a database table are encrypted, it is strongly recommended to use different encryption keys for each column. That way, even if an attacker manages to compromise a single key, the rest of the encrypted columns will remain secure. The only reason to use a single key to encrypt multiple columns is if the columns all contain values from the same set of data and the encrypted values have to be compared with each other to determine equality (such as when performing a join). The database schema and application logic should be designed so as to minimize situations where this is necessary. If multiple columns of a database table are encrypted, it is strongly recommended to use different encryption keys for each column.

Indexes:

Indexes are created to facilitate the search of a particular record or a set of records from a database table. Indexes are created on a specific column or a set of columns. When the database table is selected, and WHERE conditions are provided, the database will typically use the indexes to locate the records, avoiding the need to do a full table scan. In many cases searching on an encrypted column will require the database to perform a full table scan regardless of whether an index exists. For this reason, encrypting a column that is part of an index is not recommended.

Primary Key:

Encrypted columns can be a primary key or part of a primary key, since the encryption of a piece of data is stable (i.e., it always produces the same result), and no two distinct pieces of data will produce the same ciphertext, provided that the key and initialization vector used are consistent. However, when encrypting entire columns of an existing database, depending on the data migration method, database administrators might have to drop existing primary keys, as well as any other associated reference keys, and re-create them after the data is encrypted. For this reason, encrypting a column that is part of a primary key constraint is not recommended. Since primary keys are automatically indexed there are also performance considerations, as described above.

Foreign Key:

A foreign key constraint can be created on an encrypted column. However, special care must be given during migration. In order to convert an existing table to one that holds encrypted data, all the tables with which it has constraints must first be identified. All referenced tables have to be converted accordingly.

Searching:

Searching for an exact match of an encrypted value within a column is possible, provided that the same initialization vector is used for the entire column. On the other hand, searching for partial matches on encrypted data within a database can be challenging and can result in full table scans. One approach to performing partial searches, without prohibitive performance constraints and without revealing too much sensitive information, is to apply an HMAC to part of the sensitive data and store it in another column in the same row. For example, a table that stores encrypted customer email addresses could also store the HMAC of

3. DATA ENCRYPTION ALGORITHM

The DEA is used by TDEA to cryptographically protect blocks of data consisting of 64 bits under the control of a 64-bit key4. Subsequent processing of the protected data is accomplished using the same key as was used to protect the data. Each 64-bit key shall contain 56 bits that are randomly generated and used directly by the algorithm as key bits. The other eight bits, which are not used by the algorithm, may be used for error detection. The eight error detecting bits are set to make the parity of each 8-bit byte of the key odd. That is, there is an odd number of "1"s in each 8-bit byte.

During each application of the DEA engine, a block is subjected to an initial permutation ***IP***, then to a complex key-dependent computation and finally to a permutation that is the inverse of the initial permutation, IP^{-1}. The key-dependent computation can be simply defined in terms of a function ***f*** and a function ***KS***, called the **key schedule**. The DEA engine can be run in two directions - as a forward transformation and as an inverse transformation. The two directions differ only by the order in which the bits of the key are used. A description of the forward and inverse transformations are provided below, followed by a definition of the function ***f*** in terms of primitive functions called by the selection functions ***Si,*** and the permutation function ***P***. Values for ***Si***, ***P*** and ***KS*** of the engine are contained in Appendix A. The following notation is convenient: Given two blocks ***L*** and ***R*** of bits, ***LR*** denotes the block consisting of the bits of ***L*** followed by the bits of ***R***. Since concatenation is associative, ***B1B2...B8***, for example, denotes the block consisting of the bits of byte ***B1*** followed by the bits of byte ***B2***...followed by the bits of byte ***B8***.

3.1 DEA Forward Transformation:

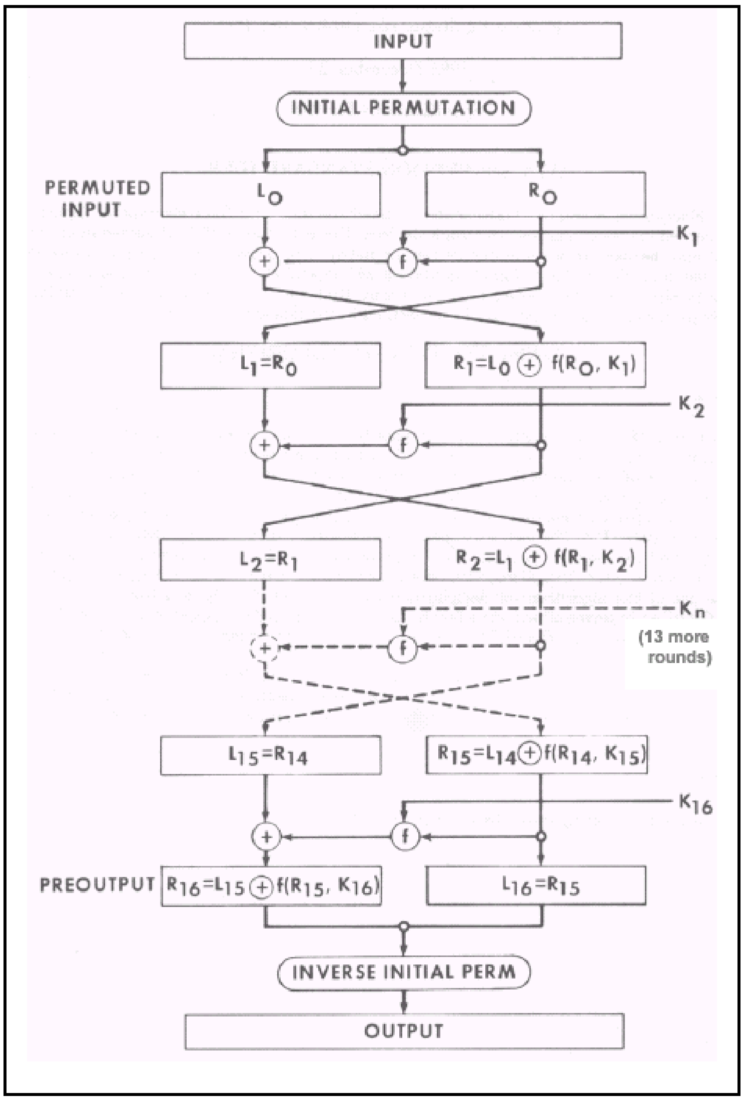


Figure 3.1. Forward Transformation of the DEA Cryptographic Engine

The 64 bits of the input block for the forward transformation are first subjected to the following permutation, called the initial permutation ***IP***:

IP

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

That is, the permuted input has bit 58 of the input as its first bit, bit 50 as its second bit, and so on, with bit 7 as its last bit. The permuted input block is then the input to a complex key dependent computation that is described below. The output of that computation, called the preoutputs, is then subjected to the following permutation that is the inverse of the initial permutation:

$\underline{IP^{-1}}$

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output. The key-dependent computation that uses the permuted input block as its input to produce the preoutput block consists, except for a final interchange of blocks, of 16 iterations of a calculation that is described below in terms of the function ***f***. This function operates on two blocks, one of 32 bits and one of 48 bits, to produce a block of 32 bits. Let the 64 bits of the input block to an iteration consist of a 32 bit block ***L,*** followed by a 32 bits block ***R***. Using the notation defined above, the input block is then ***LR***. Let ***K*** be a block of 48 bits chosen from the 64-bit key. Then the output ***L'R'*** of an iteration with input ***LR*** is defined by:

L'* = *R ----------------------------***(1)***

R'* = *L (+)f(R,K)

where (+)denotes bit-by-bit addition modulo 2 (also known as exclusive-or or XOR). As remarked before, the input of the first iteration of the calculation is the permuted input block. If ***L'R'*** is the output of the 16th iteration, then ***R'L'*** is the preoutput block. At each iteration, a different block ***K*** of key bits is chosen from the 64-bit key designated by ***KEY***. With more notation, the iterations of the computation can be described in more detail. Let ***KS*** be a function that takes an integer *n* in the range from 1 to 16 and a 64-bit block ***KEY*** as input. The output of ***KS*** is a 48-bit block ***Kn*** that is a permuted selection of bits from ***KEY***. That is:

Kn* = *KS* (*n*, *KEY) -----------------------------------***(2)***

with ***Kn*** determined by the bits in 48 distinct bit positions of ***KEY***. ***KS*** is called the key schedule because the block ***K*** used in the *n*'th iteration of (1) is the block ***Kn*** determined by (2). As before, let the permuted input block be ***LR***. Finally, let ***L()*** and ***R()*** be respectively ***L*** and ***R,*** and let ***Ln*** and ***Rn*** be respectively ***L'*** and ***R'*** of (1) when

L and ***R*** are respectively ***Ln-1*** and ***Rn-1***, and ***K*** is ***Kn***; that is, when *n* is in the range from 1 to 16,

Ln* = *Rn-1

Rn* = *Ln-1 (+) f (Rn-1,Kn) ----------------------------------***(3)***

The preoutput block is then ***R16L16***. The key schedule ***KS*** of the algorithm is described in detail in Appendix A. The key schedule Produces the 16 ***Kn*** that are required for the algorithm.

3.2 DEA Inverse Transformation

The permutation ***IP-1*** applied to the preoutput block is the inverse of the initial permutation ***IP*** applied to the input. Further, from (1) it follows that:

R* = *L'

L* = *R' (+) f (L', K) ---------------------------------- ***(4)***

Consequently, to apply the inverse transformation, it is only necessary to apply the very same algorithm to a block of the protected data produced by the forward transformation, taking care that at each iteration of the computation, the same block of key bits **K** is used during the inverse transformation as was used during the forward transformation. Using the notation of the previous section, this can be expressed by the equations:

Rn-1 = ***Ln***

Ln-1 = ***Rn (+) f (Ln, Kn)*** ------------------------------------*(5)*

Where ***R16L16*** is the permuted input block for the inverse transformation, and ***L0R0*** is the preoutput block. That is, for the inverse transformation with ***R16L16*** as the permuted input, ***K16*** is used in the first iteration, ***K15*** in the second, and so on, with ***K1*** used in the 16th iteration.

3.3 The Function f:

A sketch of the calculation of ***f(R,K)*** is given in Figure

Let ***E*** denote a function, which takes a block of 32 bits as input and yields a block of 48 bits as output. Let ***E*** be such that the 48 bits of its output, written as 8 blocks of 6 bits each, are obtained by selecting the bits in its inputs in order according to Table 1:

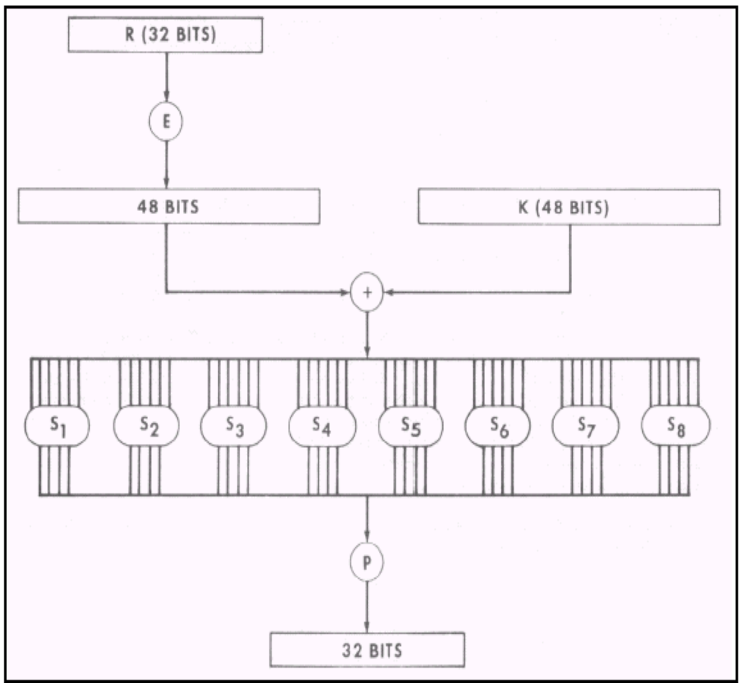


Figure 3.2 Calculation of f(*R*, *K*)

Table 1: *E* BIT-SELECTION TABLE

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Thus, the first three bits of ***E(R)*** are the bits in positions 32, 1 and 2 of ***R,*** while the last 2 bits of ***E(R)*** are the bits in positions 32 and 1.

Each of the unique selection functions ***S1, S2,..., S8***, takes a 6-bit block as input and yields a 4-bit block as output and is illustrated by using Table 3. Table 2 contains ***S1***:

Table 3: *S*1 Column Number

Row No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

If ***S1*** is the function defined in this table, and ***B*** is a block of 6 bits, then ***S1(B)*** is determined as follows: The first and last bits of ***B*** represent, in base 2, a number in the range 0 to 3. Let that number be *i*. The middle 4 bits of ***B*** represent, in base 2, a number in the range 0 to 15. Let that number be *j*. Using the table, look up the number in the *i*'th row and *j*'th column. It is a number in the range 0 to 15 and is uniquely represented by a 4-bit block. That block is the output ***S1(B)*** of ***S1*** for the input ***B***. For example, for input 011011 the row is 01 (i.e., row 1), and the column is determined by 1101 (i.e., column 13). The number 5 appears in row 1, column 13, so the output is 0101. Selection functions ***S1,S2,...,S8*** of the algorithm appear in Appendix A.

The permutation function ***P*** yields a 32-bit output from a 32-bit input by permuting the bits of the input block. Such a function is defined by Table 3.1 :

Table 3.1 : *P*

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

The output ***P(L)*** for the function ***P*** defined by this table is obtained from the input ***L*** by taking the 16th bit of ***L*** as the first bit of ***P(L)***, the 7th bit as the second bit of ***P(L)***, and so on until the 25th bit of ***L*** is taken as the 32nd bit of ***P(L)***. The permutation function on ***P*** of the algorithm is repeated in Appendix A. Now let ***S1,...,S8*** be eight distinct selection functions, let ***P*** be the permutation function, and let ***E*** be the function defined above.

To define ***f(R,K)***, let ***B1,...,B8*** be blocks of 6 bits each for which

B1B2...B8 = ***K*** (+) ***E(R)***

The block ***f(R, K)*** is then defined to be ---------------------------------(6)

P (S1 (B1) S2 (B2)...S8 (B8)) ---------------------------------*(7)*

Thus, ***K (+) E(R)*** is first divided into the 8 blocks as indicated in (6). Then each ***Bi*** is taken as an input to ***Si,*** and the 8 blocks ***S1(B1),S2(B2),...,S8(B8)*** of 4 bits each are consolidated into a single

block of 32 bits, which forms the input to ***P***. The resultt (7) is then the output of the function ***f*** for the inputs ***R*** and ***K***.

3.4 TRIPLE DATA ENCRYPTION ALGORITHM:

Let ***EK(I)*** and ***DK(I)*** represent the DES encryption and decryption of ***I*** using DES key ***K*** respectively. Each TDEA encryption/decryption operation (as specified in ANSI X9.52) is a compound operation of DES encryption and decryption operations. The following operations are used:

1. TDEA encryption operation: the transformation of a 64-bit block ***I*** into a 64-bit block ***O*** that is defined as follows:

O = EK3(DK2(EK1(I))).

2. TDEA decryption operation: the transformation of a 64-bit block ***I*** into a 64-bit block ***O*** that is defined as follows:

O = DK1(EK2(DK3(I)))

The standard specifies the following keying options for bundle ***(K1, K2, K3)***

1. Keying Option 1: ***K1, K2*** and ***K3*** are independent keys;

2. Keying Option 2: ***K1*** and ***K2*** are independent keys and ***K3 = K1***;

3. Keying Option 3: ***K1 = K2 = K3***.

A TDEA mode of operation is backward compatible with its single DES counterpart if, with compatible keying options for TDEA operation, 1. an encrypted plaintext computed using a single DES mode of operation can be decrypted correctly by a corresponding TDEA mode of operation; and 2. an encrypted plaintext computed using a TDEA mode of operation can be decrypted correctly by a corresponding single DES mode of operation. When using Keying Option 3 ***(K1 = K2 = K3)***, TECB, TCBC, TCFB and TOFB modes are backward compatible with single DES modes of operation ECB, CBC, CFB, OFB respectively. The diagram is shown bellow to illustrates TDEA encryption and TDEA decryption.

TRIPLE DES BLOCK DIAGRAM (ECB Mode)

TDEA Encryption Operation:

I → DES(EK1) → DES (DK2) → DES(EK3) → O

TDEA Decryption Operation:

I→ DES (DK3) → DES(EK2) → DES (DK1) → O

3.5 SUMMARY OF ALGORITHM:

3.5.1 DES and Triple DES:

The Data Encryption Standard (DES) algorithm is a Fiestel cipher which processes plaintext blocks of 64 bits, producing 64 bit cipher-text blocks. Details of the DES algorithm. Is given bellow. Triple DES is triple encipherment of the plain-text to produce the cipher-text. Figure 3 shows the process of encryption under Triple DES in EDE mode.

3.5.2 Encryption:

INPUT : plain-text m1,m2…..m64 ; 64-bit key K= k1,k2….k64 $ ___ (includes 8 parity bits)

OUTPUT: 64-bit cipher-text block C = c1,c2……..c64;

1. (key schedule) Compute sixteen 48-bit round keys ki __ from K

2. (L0,R0)←_ _ IP (m1,m2 …..m64), Split the result into left and right 32-bit halves ,_

and IP(x) _is the Initial Permutation.

3. (16 rounds) for i from 1 to 16, compute Li ,_ and Ri

Li = Ri-1

Ri = Li (+) f (Ri-1 , Ki) where f (Ri-1 , Ki) =P(S(E(Ri-1) (+) Ki))

computing f (Ri-1 , Ki) =P(S(E(Ri-1) (+) Ki)) _ as follows:

(a) Expand Ri-1 = r1,r2……r32; _ from 32 bits to 48 using the Expansion Permutation T ← E(Ri-1)

. _ ______

(b) Tv ← T (+) Ki Represent ._ as eight 6-bit character strings (B1,B2……B8) = Tv

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 5, Issue 1, pp: (66-104), Month: January - March 2017, Available at: www.researchpublish.com

(c) Tw ← (S (B1),S(B2).......S(B8)), S1......S8 represent the Substitution boxes.

(d) Tx ← P(Tw) P(x) is the Permutation function.

4. b1, b2.........b64 ← (R16 , L16)

5 C ← IP^{-1} (b1,b2,.......b64) _ Transpose Initial Permutation IP(x) to IP^{-1}.

3.5.3 Decryption:

DES Decryption uses the same algorithm as Encryption, however the Key Schedule is

reversed and , and _ are swapped around.

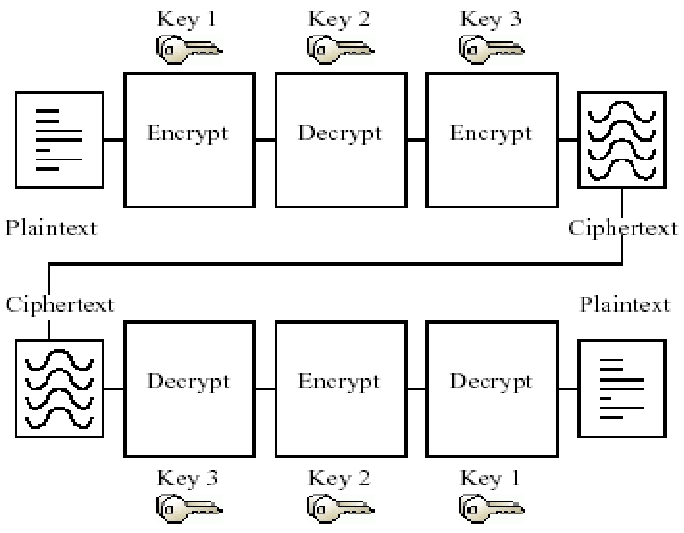


Figure 3.3: This is DES encryption where an encrypt operation is encrypt-decrypt-encrypt, and decrypt is decrypt-encrypt-decrypt. An implementation of Triple DES would simply be a wrapper around the three DES rounds.

4. ENCRYPTION STANDARDS

4.1 Block ciphers:

Block ciphers operate on fixed blocks of data at a time. They encrypt data and are building blocks for other functions such as message authentication, pseudorandom number generation, or hashing. Figure 1 in the main text (next page) illustrates the basic electronic codebook (ECB) mode of block cipher operations, which occurs under the *symmetric key*'s control, so called because the same key is used to both encrypt and decrypt data. An *m*-bit block of *plaintext* data and an *n*-bit key are input to the encryption algorithm, which generates an *m*-bit block of *ciphertext* output. The same key and the ciphertext block are input to the decryption algorithm to output the plaintext again. The secrets are the plaintext and the key. Although military algorithms are traditionally kept secret, this is infeasible for open commercial use, and everyone in the field knows the AES algorithm. Security depends only on the key's secrecy. The goal of a strong symmetric key encryption algorithm is that there is no way to decrypt the data except by knowledge of the key and no better way to find that key than key exhaustion. Put plainly, given infinite resources, key exhaustion always works. But, on average, key exhaustion takes 2*n*–1 operations, and if *n* is large enough, all the combined computer power in the world cannot accomplish the task in a human lifetime or even for centuries. The AES block size is 128 bits (before the AES, 64 bits was common) and the key sizes can be either 128, 192, or 256 bits. Every key defines a different codebook, mapping each plaintext value to a unique ciphertext value. We assume that an adversary can obtain some plaintext–ciphertext pairs (he or she might know the contents of some encrypted messages). The block size must, therefore, be large enough that an adversary cannot accumulate enough pairs to learn any appreciable fraction of the total codebook. In practice, we prevent this by using large blocks and by changing keys often enough that only a small fraction of the codebook is ever used. Obviously this constraint on key life imposed by a 128-bit block cipher such as the AES is much less of a limitation than for a 64-bit block cipher.

4.2 Modes of Operation:

Block ciphers are used to protect data in certain basic patterns called *modes of operation*. After NIST introduced the DES in 1977, it produced *FIPS 81, DES Modes of Operation*, which defines four encryption modes for the DES algorithm: the electronic codebook (ECB) mode, the cipher block chaining (CBC) mode, the cipher feedback (CFB) mode, and the output feedback (OFB) mode. These four modes, informally known as the "NIST modes," have been widely adopted by users of cryptography around the world. Although these modes are defined in FIPS 81 specifically for the 64-bit block and 56-bit key of DES, they can be extended to other block and key sizes. Of the modes, two include two full block cipher modes that directly encrypt entire blocks of data at a time**:**

ECB mod : as Figures 1 illustrates, is the mode in which a block of plaintext is simply encrypted directly to form the ciphertext, and is the basic operation from which all other modes are built. (See the "Block Ciphers" sidebar for an ECB mode explanation.) However, when ECB mode encrypts data, the same plaintext input results in the same ciphertext output, so patterns in the input are revealed to an eavesdropper. Therefore, ECB mode is insecure for many applications.

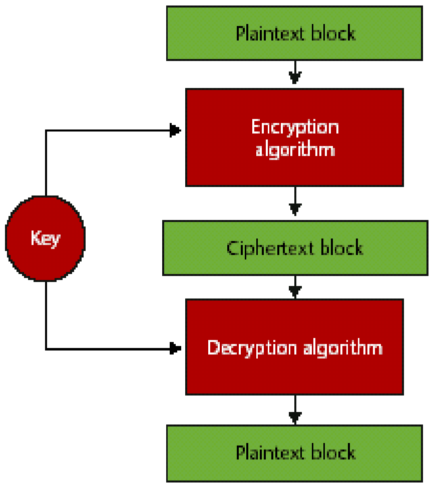


Figure 4.1. Block cipher encryption in Electronic Code Book (ECB) mode

CBC mode : illustrated in Figure 2, is the mode in which the ciphertext of block *i* is XORed with the plaintext of block *i*+1 before it is encrypted. The first block is XORed with an initialization vector (IV) that need not be kept secret. CBC mode hides data patterns in the ciphertext and is more secure than ECB mode. One problem with CBC mode, though, is that it is strictly block serial, so block *i* must be encrypted before block *i*+1, limiting its ability to perform multiple operations in parallel. A single bit error in a communications channel usually results in two full blocks of corrupt plaintext. FIPS 81 also includes two *stream cipher* modes, in which the block cipher generates a unique pseudorandom *keystream* that is determined by a key, an IV, and, in some cases, the plaintext itself. The sender then XORs the keystream with the plaintext to encrypt it, and the receiver then XORs the keystream with the ciphertext to decrypt it. The IV need not be kept secret but must be a *nonce*, a value that is never repeated during a cryptographic session period or until the key is changed. Only the block cipher encryption operation is needed; block cipher decryption is not used. Using stream ciphers has its pitfalls; in particular, the keystream must not be repeated or plaintext could be revealed, and it is sometimes possible for an attacker to invert predictable bits in the plaintext. In many cases, if bits are added to or lost from the ciphertext during transmission, cryptographic synchronization is lost. Using stream ciphers with noncryptographic checksums in protocols is perilous and could let an active attacker decrypt plaintext without learning the key. For example, an application that uses a stream cipher insecurely is the IEEE 802.11 WEP protocol, which protects wireless Ethernet. WEP falls victim to such a checksum attack and also more or less guarantees that cipher streams repeat after a short period of operation. You might ask, if stream ciphers come with all these pitfalls and perils, why do we use them? We use stream ciphers to protect data communications channels because the keystream can be generated in parallel with the rest of the channel processing. Therefore, stream ciphers only need to introduce one bit-time of additional latency to encrypt, plus one bit-time to decrypt.

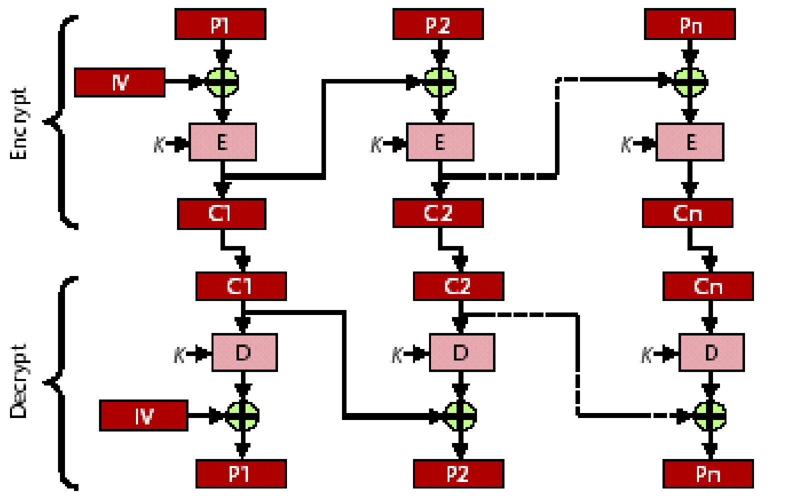


Figure 4.2 Block cipher encryption in Cipher Block Chaining (CBC) mode.

CBC or EBC mode, on the other hand, must accumulate a full block of data before encrypting or decrypting it, and also adds the time required for the encryption and decryption operations to the channel latency. This delay is intolerable in many communications applications. The two stream cipher modes of FIPS 81 are

OFB mode : As Figure 3 illustrates, in this mode, the encryption operations output is fed back into the input pitfalls and perils, why do we use them? We use stream and used to generate a keystream, which is then XORed with the data to generate the ciphertext. Unlike many modes, ciphertext bit transmission errors are not expanded in the received plaintext, but if bits are lost or inserted in the ciphertext, cryptographic synchronization is lost.

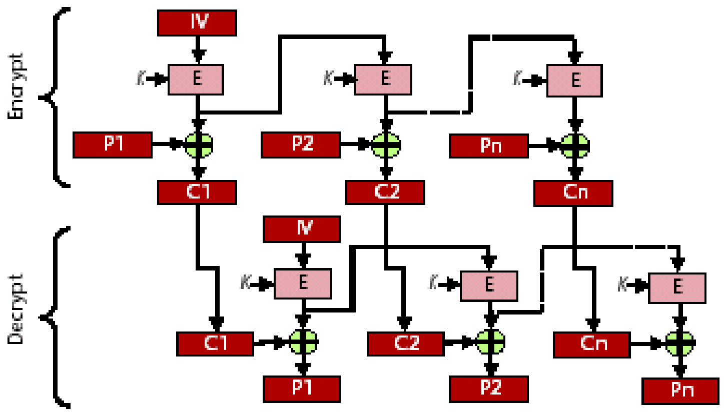


Figure 4. 3 Stream cipher encryption in Output Feedback (OFB) mode.

CFB mode : As Figure 4 illustrates, this is another stream cipher mode in which the block encryption output is XORed as a keystream with the plaintext, but the feedback term to the encryption algorithm is the ciphertext itself. If only one ciphertext bit at a time is fed back through a shift register into the block encryption operation, this mode can be made self-synchronizing. CFB transmission errors in the ciphertext are expanded in the plaintext output. In some cases, this error expansion can be a security advantage because it prevents an adversary from selectively inverting only one bit of plaintext. NIST knew that it would have to revise the AES's modes of operation to account for the block and key size differences from DES. Moreover, a major complaint about the existing modes (except ECB) was that they all involved feedback or chaining, making it impossible to improve their performance with parallel encryption operations. On the other hand, ECB is almost always insecure if it is used to encrypt a lot of data directly.

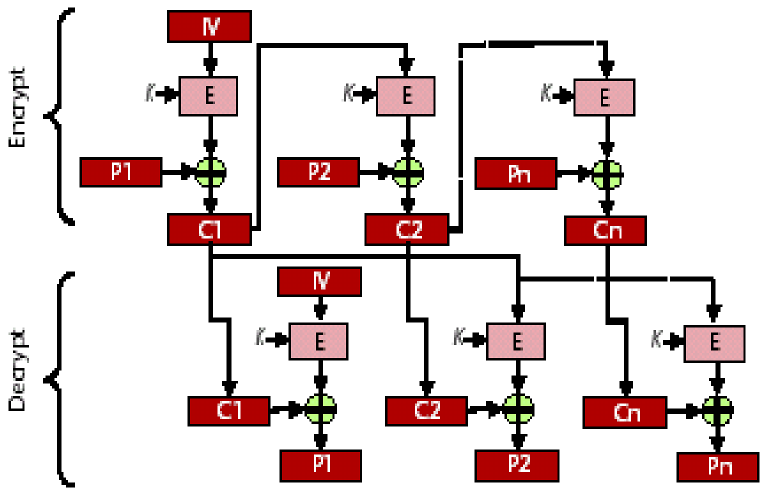


Figure 4. 4 Stream cipher encryption in Cipher Feedback (CFB) mode.

5. DESIGN AND IMPLEMENTATION OF ALGORITHM IN JAVA.

5.1 Encrypting and Decrypting Data:

Basically, to encrypt or decrypt data, we need a key and a cipher. We can use the SecretKeyFactory to generate new keys. We can either generate a random key or use an existing array of bytes for the key. In the latter case, we first create a key "spec" from the original bytes and then pass this spec to the key factory.

The following code snippet creates a key spec from an array of bytes and then creates a key. The key is a DESede key, which is commonly known as Triple-DES—a more secure version of the original Data Encryption Standard (DES) algorithm.

```
// Create an array to hold the key
    byte[] encryptKey = "This is a test DESede key".getBytes();
// Create a DESede key spec from the key
    DESedeKeySpec spec = new DESedeKeySpec(encryptKey);
// Get the secret key factor for generating DESede keys
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(
      "DESede");
// Generate a DESede SecretKey object
    SecretKey theKey = keyFactory.generateSecret(spec);
```

Once we have a key, we can create a cipher using the Cipher.getInstance method:

```
Cipher cipher =   Cipher.getInstance("DESede/CBC/PKCS5Padding");
```

In this case, we create a DESede (Triple-DES) cipher with some extra parameters. The second parameter (CBC) controls how we handle successive blocks of data. The simplest method of encrypting multiple blocks is called Electronic Code Book (ECB). When you use ECB, we simply encrypt one block at a time. For each block, we just take the original data, and encrypt it without any additional information. ECB is rather unsafe, however, because someone may be able to learn something about the content of the data by analyzing identical blocks. we can make your data safer by using Chained Block Cipher (CBC) mode. When we encrypt using CBC mode, we combine the data that we want to encrypt with a special array of data called an initialization vector (IV) and then encrypt the combined data. For the second block, however, we combine the encrypted first block with the data we want to encrypt (usually with a simple XOR) and then encrypt the combined block. This eliminates repetitive patterns in the encrypted data.

Two other modes, Output Feedback (OFB) and Cipher Feedback (CFB), create a stream cipher byte generator out of a block cipher; that is, they make a block cipher behave like a stream cipher.

The other parameter—PKCS5Padding, in this case—indicates how to handle incomplete blocks. Remember that a block cipher operates on blocks of a fixed size. By using one of the common padding algorithms, we include the block size in the encrypted data, ensuring that when we decrypt, we get the correct number of bytes back. We can also specify NoPadding if we don't want any padding.

When we use CBC mode, we must create an initialization vector before we encrypt or decrypt data. The contents of the IV don't matter much, but we *must* use the same IV to encrypt and decrypt.

```
IvParameterSpec IvParameters = new IvParameterSpec(
  new byte[] { 12, 34, 56, 78, 90, 87, 65, 43 });
```

The process of encryption and decryption is almost identical from a coding perspective. To decrypt data, you use Cipher.DECRYPT_MODE in the init method for the Cipher object. Beyond that, everything else is the same.

The cryptography libraries in Java 2 SDK 1.4 are pretty robust, and there are many options available. You can digitally sign or encrypt whole objects. You can also encrypt a stream of data without resorting to SSL. Take a look at the javax.crypto package to get an idea of some of the other available features.

6 . DATABASE ENCRYPTION

6.1 Evaluating File versus Column Level Approaches:

which approach to database encryption best meets their requirements and assessing a range of solutions available in the marketplace. Today, enterprises face an increasing number of attacks that are targeted at stealing sensitive information. This fact, coupled with a broad number of state, federal, and corporate compliance mandates, are driving enterprises towards database encryption. In doing so, enterprises face the challenge of determining Today all databases write their data in a structured format to underlying files. As a result, one way to implement database encryption is by encrypting the entire database file. While file level encryption is well suited for encrypting word documents and other sensitive files in their entirety, using this technology to encrypt database files in their entirety is rarely done or useful due to the intensive read and write nature of databases. Another approach to database encryption is granular, column-level encryption, which enables encryption of specific fields within a database. While this approach provides both flexibility and a high level of security, there are several factors that should be taken into consideration when implementing this type of solution.This paper is designed to help enterprises better understand both file and column level encryption in order to make informed decisions about which approaches and technologies to implement. When considering a database encryption solution, enterprises must consider several key areas that require evaluation to address some fundamental questions. These include the following:

1. Encryption—what type of solution is most effective when encrypting within database environments?

2. Performance—What are the performance implications of database encryption for each of the various approaches in the market?

3. ETL, backup, and replication requirements—How is data best protected when it is backed up, replicated, and loaded in and out of the database?

4. Authentication, authorization, and auditing—What solution offers the most effective and granular authentication and

auditing capability?

5. Key Management and Security—What is the most effective and secure way to store and manage cryptographic keys

Understanding the answers to these questions will help enterprises assess and determine the right solution for their organization.

6.2 Encryption:

When encrypting sensitive data within database tables one should consider the difference between encrypting at the column level versus encrypting an entire database file. File encryption techniques operate on entire files and offer access control and auditing capabilities on the entire file only. Column level encryption solutions provide much more granularity and enable access control, auditing, and security policies to be applied to specific columns within a database. Typically, the sensitive information that needs to be encrypted—such as credit cards numbers, social security numbers, and the like—only represents a very small percentage of all the information managed within a database. As a result, it is undesirable to incur the overhead of encrypting tables, stored procedures, and other objects for which there is no requirement for encryption. Some file level encryption solutions may require manipulation of table spaces and file groups so that tables with encrypted data can be grouped in a single file, but this can have several negative implications. First, it is extremely unlikely that the tables with encrypted data are already grouped in this manner. Consequently, organizations must move tables across table spaces and file groups, which ultimately breaks the applications associated with the database. Secondly, any tuning that may have been performed previously within the existing database will be affected by any changes to the underlying table space. Even if all of the encrypted tables can be simply grouped into a single file, there is still the possibility that there may be many columns within these tables and many other database objects that do not require encryption. Take for example a customer table that holds a credit card column but also has 40 other columns, which, if stolen in the event of a breach, would pose no threat to a business. In all of these cases, there is a significant penalty for encryption and decryption of non-sensitive data. Conversely, column-level encryption only impacts the data that needs to be encrypted, thereby eliminating any degradation to non-sensitive columns. Access to non-sensitive data continues through standard database queries and retrieves cleartext data while access to sensitive data is performed through similar queries but retrieves ciphertext and decrypts it for authorized database and application users. Column level encryption solutions can provide full transparency to the application by invoking a view/trigger model using stored procedures. While these schema changes are required as part of the implementation, most column level solutions on the market have automated these changes, and which can often be pushed out from a central management interface. Typically, the sensitive information that needs to be encrypted only represents a very small percentage of all the information managed within a database.

6.3 Performance:

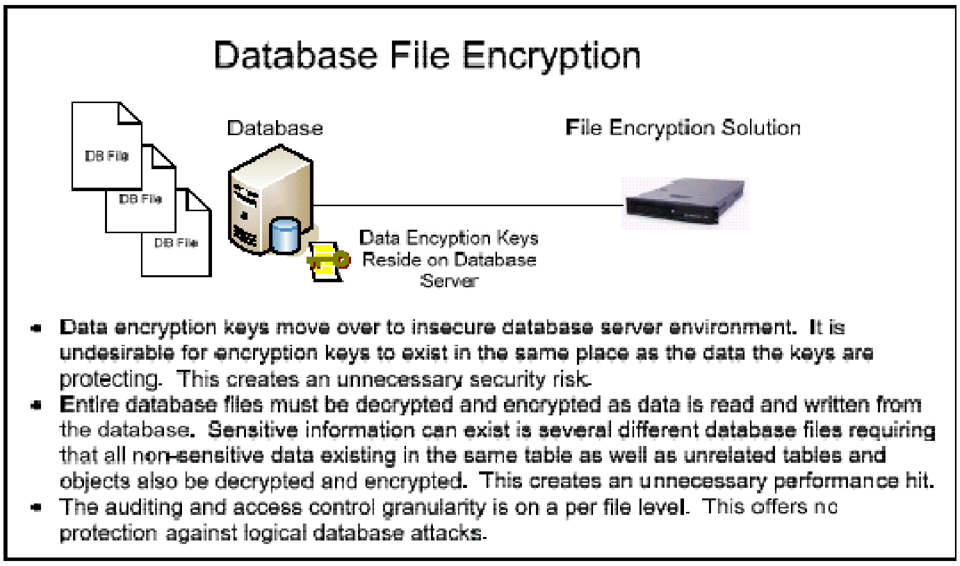
The performance of virtually any I.T. solution is paramount, and database encryption is no exception. With the high volumes of data processing being conducted in today’s businesses, it is necessary that encryption of sensitive data is achieved without severely affecting the performance of associated I.T. systems. Cryptography is without a doubt resource intensive; as a result enterprises will need to go to great lengths to minimize the overhead created by cryptography by only encrypting sensitive data. Why incur the overhead of encrypting and decrypting data that isn’t sensitive? While encryption of entire database files has advantages in terms of simplicity, it is important to consider a number of issues that arise when taking this approach. Databases at most enterprises are very large and growing every day; encrypting the entire database file will require cryptographic functions to be performed every time data is accessed. While encrypting entire database files may be feasible for very small databases, in enterprise environments the prospect of managing the overhead created by encrypting and decrypting every piece of data in large databases is unacceptable. Databases typically access data from underlying files through the native file system in data segments. When implementing file level encryption, this results in a data segments (which are typically 4Kb in size) being decrypted every time a single select statement requests a result. When deploying encryption of database files, encryption and decryption must be performed for each read operation that is invoked by the database to the underlying encrypted file. This can have a significant impact on enterprise databases. Column level encryption logically encrypts individual fields/columns within a table that are deemed sensitive. For example, if a customer wants to encrypt a credit card number, the ‘CC_NUM’ column is encrypted. As a result, a select statement requesting a particular user credit card number will only decrypt a 16 byte credit card number to retrieve the desired result set. A file level encryption solution would have to decrypt 250 times more data (assuming a 4Kb segment within the file where a credit card number resides versus a 16 byte card number). Furthermore, in column level encryption when non-sensitive data is accessed, no encryption or decryption takes place, so the original database performance is not affected. Column level encryption does pose some potential challenges when encrypting indexed columns. However, most advanced column level solutions in the market offer a variety of methods to address this problem by ensuring that full table scans are not performed to return a single equality query. Encrypting the entire database file will require cryptographic functions to be performed every time data is accessed. Column level encryption solutions can provide full transparency to the application by invoking a view/trigger model using stored procedures

6.4 Batch/ETL, Backup, and Database Replication:

It is a common requirement for customers to move data from one environment to another. For example, one may want to export credit card numbers out of an Oracle database into a flat file and then import the flat file into a SQL Server database. With a file encryption solution, a customer would have no choice but to decrypt all credit card numbers before writing to the flat file. When the flat file is imported into the SQL Server database, all of the credit card numbers must be encrypted again. However, when deploying column level encryption, a customer can export the credit card numbers to a flat file in an encrypted format and import the flat file into the SQL Server database in the encrypted format. The two main benefits of this latter approach are that no additional encryption/decryption is required and the data does not exist in the clear outside of the database as it would in a file encryption solution. Simply put, by encrypting at the column level, companies are able to successfully manage and minimize the processing overhead that is inherent in cryptography. File encryption also raises issues concerning backups, replication, and data exports. When cryptography is implemented at the file level, the data is always decrypted when accessed. When data is extracted, whether to be written to a backup file, replicated to another database, or written to a flat file, it must be decrypted as it is extracted, and then encrypted again when stored in the new medium. The additional resources needed to decrypt and encrypt data whenever it is exported will add an unacceptable amount of overhead to a variety of critical business processes. By increasing the number of instances in which sensitive data may exist in clear text presents additional security risks. Shouldn't a credit card number only be decrypted when processing a transaction associated with that credit card? Due to the nature of file level encryption, there is no way to control the database users who are actively accessing the data within the database.

6.5 Authentication, Authorization, and Auditing:

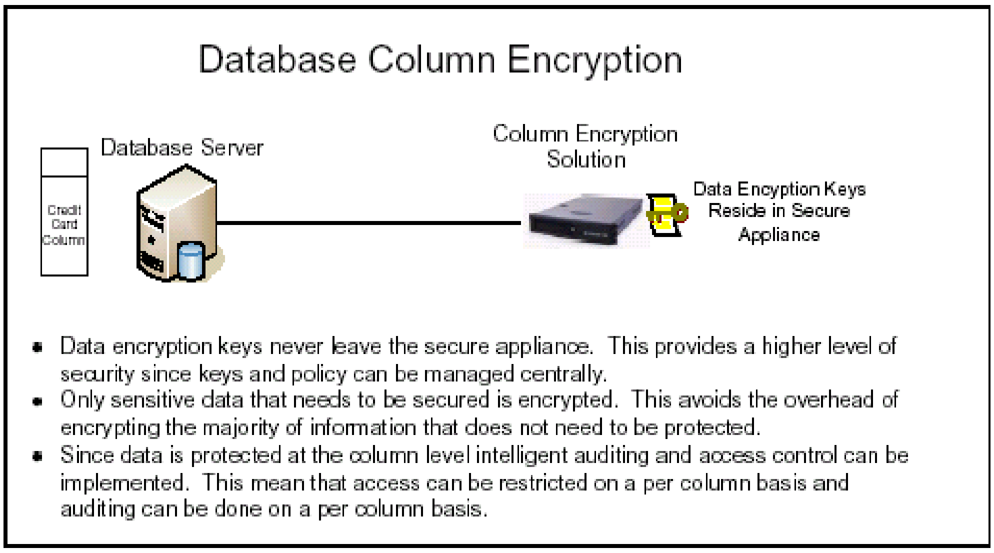
Encryption of sensitive data is only one piece of a data privacy solution. Without the proper authentication, authorization, and auditing mechanisms, encryption fails to provide a higher level of security. File level security and auditing is extremely valuable when dealing with files that are accessed directly by end users. However, that is not the case with database files. Data within a database is accessed by end users either through front end applications or database query tools that directly interface with the database server. The database files are accessed by the service id or service account used to run the database server. Encryption at the file level only provides the ability to control and audit the actions of the service account used to run the database server. There is no way to control the database users who are actively accessing the data within the database. While file encryption fails to even complement the native security mechanisms of a database server, column-level encryption takes existing security controls one step further. By implementing authentication, authorization, and auditing measures at the column level, it is possible to significantly increase security. For example, access can be granted at a more granular level; two database users may have the rights to execute select queries on the same table, but only one of the users can decrypt the encrypted columns in the table. Auditing goes further and allows any operation performed with sensitive data to be recorded. As a result, enterprises have the ability to know exactly how many credit cards were decrypted, by which database users, from which tables and databases, and at what times. Advanced authorization policies can be built around the audit data including rate and time limiting. Column-level encryption allows for monitoring of database users activities. If a user is authorized to decrypt social security numbers, you want to make sure they don't walk out the door with all of the social security numbers decrypted. Column-level encryption enables an enterprise to truly implement a fully functional data privacy solution and go beyond just encryption by controlling, monitoring, reporting, and intelligently managing your sensitive data.



- Data encryption keys move over to insecure database server environment. It is undesirable for encryption keys to exist in the same place as the data the keys are protecting. This creates an unnecessary security risk.
- Entire database files must be decrypted and encrypted as data is read and written from the database. Sensitive information can exist is several different database files requiring that all non-sensitive data existing in the same table as well as unrelated tables and objects also be decrypted and encrypted. This creates an unnecessary performance hit.
- The auditing and access control granularity is on a per file level. This offers no protection against logical database attacks.

6.6 Key Management:

When considering any encryption solution, key management is an extremely important factor. Strong considerations should be given to how keys are accessed, how they're managed, where they are located, how they can be backed up and recovered, and how integrity can be ensured. Most file encryption solutions today have some form of centralized key management, but move keys to the host for encrypting and decrypting database files. Storing the symmetric data encryption keys in the database significantly reduces the overall security of the solution. Most simple attacks on the host can compromise the keys and keys can be easily changed, altered, and deleted. Having the encryption keys reside in the same place as the data these keys are meant to protect is highly discouraged by most security professionals. In more advanced column level encryption solutions, the keys never leave the secure hardware platform. This allows security administrators to better manage keys and policy from a centralized location, creating a true security boundary. This also enables enterprises to enforce true separation of duties by enabling security administrators to control keys, users, groups, and policy while a database administrator is effectively a user of the encryption capabilities as a service. Today, enterprises face increasing threats. Therefore, it is imperative for enterprises to know which specific threats they are trying to protect against. File level encryption within databases provides protection largely against media theft and loss. Column level encryption provides protection against media theft along with a broad range of logical threats. While protecting against identified threats, enterprises must consider the type of encryption to deploy to protect sensitive data in databases, while minimizing performance impact, ensuring strong, efficient AAA and key management, and minimizing the impact of batch type processes. Effectively addressing all these factors will ensure enterprises can protect sensitive data, while minimizing the overall cost to the business.


Database Column Encryption
Database Server
Credit Card Column
Column Encryption Solution
Data Encyption Keys Reside in Secure Appliance

- Data encryption keys never leave the secure appliance. This provides a higher level of security since keys and policy can be managed centrally.
- Only sensitive data that needs to be secured is encrypted. This avoids the overhead of encrypting the majority of information that does not need to be protected.
- Since data is protected at the column level intelligent auditing and access control can be implemented. This mean that access can be restricted on a per column basis and auditing can be done on a per column basis.

7. EMPLOYING ENCRYPTION TO ACHIEVE MAXIMUM SECURITY

7.1 Introduction:

To secure sensitive data and ensure data privacy. Ingrian DataSecure Platforms offer an array of robust security features, including key management, backup, and auditing and logging—all designed to ensure the utmost security of sensitive data. This thesis offers a detailed look at these various features and provides detailed guidance into how they work and how they should be managed to optimize security.

7.2 Protecting Keys with Secure Key Management:

Keys are the foundation of all encryption-based security solutions. If a hacker, internal or external, gains access to your private keys, the security of all data formerly protected by encryption is gone. Not reduced—gone. That is a risk currently assumed by companies that store private keys used for data encryption in insecure locations, whether Web, application, or database servers. These servers are typically not secure because there are many people with access to them, they are often misconfigured and they often aren't kept up to date with the latest security patches. Additionally, keys areusually stored in an easily readable plaintext format. Even organizations that make efforts to protect private keys with passwords find that these passwords aren't protected properly, are chosen poorly, and usually must be shared between multiple administrators. Not protected properly, stored in a software environment, and exposed in server memory, keys are vulnerable to discovery. An intruder who compromises your keys can launch "eavesdropping" attacks using the stolen key to hack into vital data repositories. This could result in data theft, loss of privacy for your employees and customers, and damages to brand credibility and customer confidence. What's needed? The best protection against private key compromise is a superior combination of physical security and key management technology, including tamper-resistant hardware and the most stringent security standards throughout the private key lifecycle. With DataSecure, private keys remain private. In contrast to typical encryption configurations where private keys are stored insecurely on multiple servers, DataSecure protects the private keys in a specialized, centralized hardware appliance that features multi-level security perimeters. DataSecure uses secure key storage solutions to safeguards keys, in both hardware and software, against compromise throughout the entire lifecycle of sensitive data.

7.2.1 Secure Key Storage:

Stringent security defenses protect all data residing on Ingrian's products. With DataSecure, each sensitive element of the system is protected by its own unique, randomly generated key. Private keys are stored encrypted with several Triple-DES encryption keys that are nested within a hierarchy in which each key is protected by a parent key. This multi-layer hierarchy of keys ensures the highest level of protection against attack. DataSecure platforms are also available with an optional, tamper-resistant hardware security module (HSM). DataSecure HSM's are certified to FIPS 140-2 Level 3, the widely accepted standard of government-specified best practices for network security. Private keys are generated and stored in encrypted form within a tamper-resistant hardware security module (HSM). Keys stored in the HSM are protected from physical attacks and cannot be compromised even by stealing the Ingrian appliance itself. Any attempt to tamper with or probe the card will result in the immediate destruction of all private key data, making it virtually impossible for either external or internal hackers to access this vital information.

7.2.2 Secure Key Back Up:

A weak link in the security of many networks is the backup process. Often, private keys and certificates are archived along with configuration data from the backend servers. The backup key file may be stored in clear text or protected only by an admin istrative password. This password is often chosen poorly and/or shared between operators. To take advantage of this weak protection mechanism, hackers can simply launch a dictionary attack (a series of educated guesses based on dictionary words) to obtain private keys and associated certificates. Ingrian Networks has designed a secure backup system where:

- Private keys are never exported from the product in clear text.

- The backup file is password protected and then encrypted using an internal Ingrian key. When private keys are backed up from the DataSecure platform, they are encrypted twice, once using an administrative backup key and a second time with the internal Ingrian key. The Ingrian key makes it impossible for attackers to launch dictionary attacks and other password-guessing techniques aimed at exposing an administrative password and unlocking the backup file. Your private keys can never be exported in clear text and cannot be released without cracking several layers of triple-DES encryption, ensuring secure preservation of key data in all backup and storage activities.

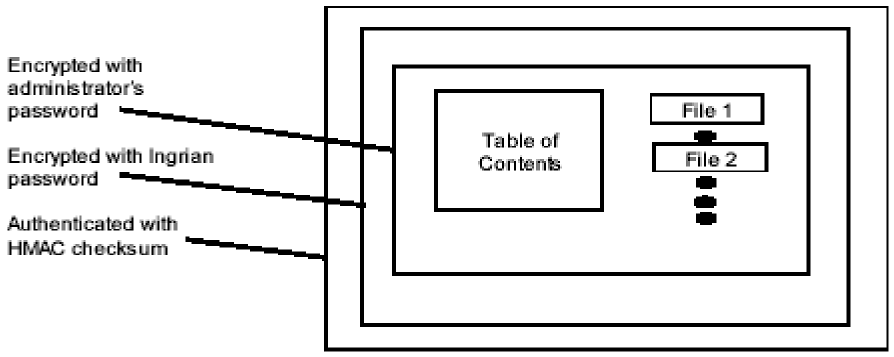

Encrypted with administrator's password
Encrypted with Ingrian password
Authenticated with HMAC checksum
Table of Contents
File 1
File 2

Figure 7. : Layers of protection for backup files

Tamper Resistance

In the DataSecure appliances that feature HSMs, key data is also protected by tamper-resistant hardware and smart cards. These products encrypt private keys using a group key that is divided between a small, predefined number of smart cards. Only Ingrian Networks products that are members of the same security group (i.e. have access to the same group key) can share the sensitive key information contained within the backup files. Group key information is transferred between DataSecure platforms using the appropriate number of smart cards.

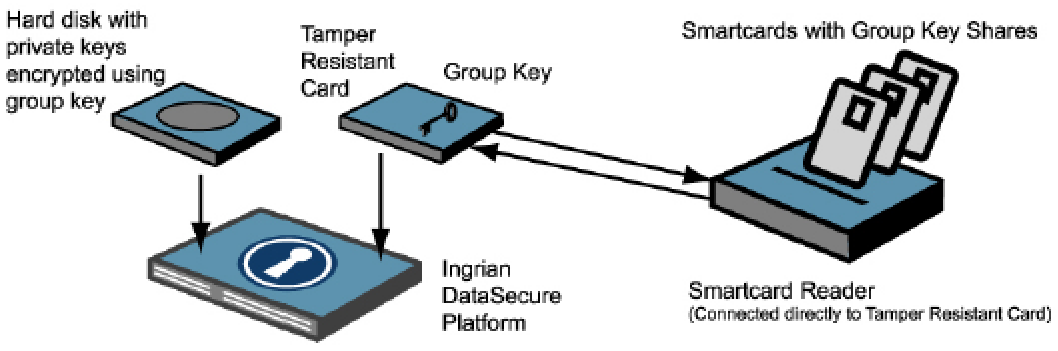

Hard disk with private keys encrypted using group key
Tamper Resistant Card
Group Key
Smartcards with Group Key Shares
Ingrian DataSecure Platform
Smartcard Reader
(Connected directly to Tamper Resistant Card)

Figure 7.1 : Private key security architecture

DataSecure supports k of n secret sharing for the group key for increased security. The group key information is distributed across a collection of smart cards (n) where group membership requires verification of a minimum number of smart cards (k). For example, organizations may elect to require three smart cards (k) and create five smart cards (n). Attempts to join a DataSecure appliance to the security group defined above with fewer than three smart cards will fail. Using a k of n schema ensures a high level of data safety. Even if k-1 smart cards are stolen, the thief will not be able to access the configuration data stored on the tamper-resistant card because they do not have enough cards to meet the defined k of n criteria.

7.3 Managing Keys for Maximum Security:

With any product or system, key management security is only as good as the policies and practices that support it. Ingrian supports the following recommendations to ensure that your administrative practices are as secure as the key management technology we employ.

Administrative Access:

The first step toward secure key management involves securing access to the DataSecure product. This can be done is several ways:

- Restricting allowed ad ministration methods. Ingrian products can be managed via a secure web-management console, a command line interface over SSH, and a console connection. To provide the desired level of security and convenience, these services can be started or stopped. In addition, the desired administration methods can be restricted on a user-by-user basis.

- Restricting administration access to an isolated or protected network. The administration of the Ingrian product can be limited to a particular interface and IP address. Ingrian recommends that this IP address exist on a network that is not routable and that the traffic be isolated through the use of VLANs or similar technologies.

- Properly configuring role-based access for each administrator using Ingrian's finegrained user access control. When creating a new user, the administrator selects the accesses allowed each user. DataSecure platforms have an extensive list of controls. For example, one user might only be given access to network configuration or backup settings, while another user might only be given access to key management functions.

- Enforcing two-factor authentication for Web administration is accomplished through the use of client certificates. The Ingrian products will recognize certificates signed by its list of trusted certificate authorities (CAs). You can also use the onboard CA to sign client certificates for use in administrative authentication.

7.4 Secure Configuration, Backup, and Restore:

The DataSecure platforms allow for all configuration data and keys to be securely backed up and restored. The backups are in the form of doubly encrypted backup files. The FIPS offering provides an additional level of security by requiring that smart cards are present in order to perform a restore of a system and adding a third level of encryption to the backup files. The Ingrian platform's backup mechanism provides two functions:

1. backing up information on the Ingrian device to be restored in case of a failure, and

2. copying configuration information between Ingrian appliances. Once an Ingrian appliance is fully configured with networking information, certificates, and user account, Ingrian recommends that the entire configuration be backed up. Every backup is protected using two keys: an Ingrian appliance internal key, and a password provided by the administrator. Because a backup file may contain sensitive information, such as user accounts and certificates, Ingrian recommends a reasonably long backup password. As mentioned above, the backup facility is also used to copy configuration information between DataSecure platforms. When creating a backup, you can choose which components of the Ingrian platform are to be backed up. Once the backup file is created, it can be uploaded onto another Ingrian platform. The information in the backup file is added to the other DataSecure platform's configuration database. Information in the backup file overrides the existing configuration on the Ingrian device.

7.5 Maintaining a Backup Password:

Every backup file is protected using two keys: an Ingrian internal key, and a password provided by the administrator at the time the backup is created. This password will be required when the backup file is restored. Therefore, it is imperative that each person who needs to restore a backup file must have access to the password that was specified when the file was created. If this password needs to be shared among administrators, it is recommended that the password

be written down by a trusted member of the staff and stored in a secure location. If possible, the password should be stored in a tamper evident container. In the event that this password needs to be used, a new file and new password should be created and stored in a similar fashion. The old backup file should be deleted or destroyed.

Creating a Security Group:

Ingrian Networks recommends the following practices for creation of a security group. Keep the following items in mind for your k of n scheme:

▪ Make sure that k is large enough to give sufficient protection to the group key. Choosing a security group in which k=1, for example, is usually a poor choice because it means that only one smart card needs to be compromised before the security of the group is at risk.

▪ Choose a k to n ratio that you are comfortable with. The larger the value of n, the more cards that are in existence as part of the group and could possibly be compromised. As a general rule, n should never be more the 3x of k. o Ensure that n – k is large enough to prevent loss of the group key information. Smart cards can be lost, damaged or stolen. Your k of n sharing scheme should be tolerant enough to allow for these unfortunate events. In addition, smart cards should be physically protected when they aren't in use. This means they should be stored in a secure location (such as a safe or vault). They should also be divided to protect against compromise and to ensure full disaster recovery.

7.6 Creating Backups:

The backup system works as follows:

1. An administrator chooses which parts of configuration data to backup. Alternatively, the administrator could choose to backup all configuration data on an Ingrian Networks product at once. It is recommended that a single configuration file be created for replication among all Ingrian products in a pool. This file should contain all components, save system information and key information. Backup files containing key information should be created individually and treated with special care.

2. The backup system then creates the encrypted backup file and downloads it onto the administrator's computer or transfers the file to the specified host using FTP/SCP. Ingrian recommends that SCP be used to transfer backup files to a secure server. These files should be named in a fashion that distinctly identifies them as well as the date the file was created. The description field should be used to add any other useful comments.Backup files containing key information can be transferring to a mobile storage media(such as a floppy disk) and stored in a physically secure location for added protection.

7.6.1 Creating a Backup Schedule:

A master backup file should be maintained for each DataSecure platform in your network. This backup file should contain the working configuration or each product and be updated each time this configuration changes. Backups containing key information should only be updated when additional keys are created.

7.6.2 Upgrades

Secure software upgrade mechanisms are a big differentiator between a secure networking product and a generic network product. Common network products often make little effort to secure the software upgrade mechanism, which makes such a device more vulnerable to security breaches. DataSecure platforms contain a secure software upgrade mechanism to avoid this problem. These products will only load upgrades signed by Ingrian. The product will reject all other upgrades.

7.7 Internal Backup List:

The Internal Backup List section of the Backup and Restore page provides a list of internal backup files. The list shows the date on which the backup was created. Using the Download button you can download an internal backup file to your browser. The Download button enables you to move a previously created internal backup file to a secondary system. Internal backups may also be deleted to remove them from the Ingrian device .

8. CONCLUSIONS AND FUTURE WORK

Conclusions:

The process to migrate plaintext data to encrypted format is quite simple when using the NAE Connector; furthermore the process can be completely automated. The most important is that the integration is completely transparent to applications that interface with our sensitive data. Before deploying DataSecure, our sensitive data sits in the clear in the databases. After deploying DataSecure, sensitive data is encrypted, and applications can continue interacting with sensitive data using the same SQL statements; however, instead of interacting directly with that sensitive data, the application servers are actually interacting with a view of the data.

In this thesis , I have used a DESede (Triple-DES) algorithm in CBC mode and implemented in Java . The result is shown in the encrypted form on command Prompt Screen in the thesis . With this any data, either Numeric, Alphanumeri or Strings can be encrypted . The simplest method of encrypting multiple blocks is called Electronic Code Book (ECB). When you use ECB, we simply encrypt one block at a time. For each block, we just take the original data, and encrypt it without any additional information. ECB is rather unsafe, however, because someone may be able to learn something about the content of the data by analyzing identical blocks. we can make your data safer by using Chained Block Cipher (CBC) mode. When we encrypt using CBC mode, we combine the data that we want to encrypt with a special array of data called an initialization vector (IV) and then encrypt the combined data. For the second block, however, we combine the encrypted first block with the data we want to encrypt (usually with a simple XOR) and then encrypt the combined block. This eliminates repetitive patterns in the encrypted data..

PKCS5Padding, in this case—indicates how to handle incomplete blocks. Remember that a block cipher operates on blocks of a fixed size. By using one of the common padding algorithms, we include the block size in the encrypted data, ensuring that when we decrypt, we get the correct number of bytes back. We can also specify NoPadding if we don't want any padding.

When we use CBC mode, we must create an initialization vector before we encrypt or decrypt data. The contents of the IV don't matter much, but we *must* use the same IV to encrypt and decrypt.

The percentage by which encrypted data is larger than plaintext data varies depending on the encryption algorithm that is used to perform the encrypt operation. In this thesis I used DESede cipher in CBC mode with PKCS5 padding ,as such, a nine digit number Expands to 16 bytes of binary data and the results is shown in byte code .

Advantages:

– Much faster than public key systems like RSA

– Easy to implement in both hardware and software

– Tested for 25 years and no logic flaws. Trusted cipher

Disadvantages:

– Transmission of secret key over public channel is tricky

– Not as fast as AES, RC6 or Blowfish which are newer.

– Keylength of DES is too small but 3DES solves this.

Applications and future Work:

Cryptography is utilized in various applications and environments. The specific utilization of encryption and the implementation of TDEAwill be based on many factors particular to the computer system and its associated components. In general, cryptography is used to protect data while it is being communicated between two points or while it is stored in a medium vulnerable to physical theft or technical intrusion (e.g., hacker attacks). In the first case, the key must be available at the transmitter and receiver simultaneously during communication. In the second case, the key must be maintained and accessible for the duration of the storage period. NIST Ingrian Networks brings complete data privacy to the enterprise. DataSecure features a dedicated security appliance and specialized software that enables organizations to encrypt critical data in applications and databases.

The Encryption technique (TDEA) that we used in this thesis can also be implemented in other modes like OFB and CFB mode using PKCS5 padding or No padding in Java or Some other language like C and C++.

With Ingrian DataSecure Platforms, organizations can protect critical data from both internal and external threats, and ensure compliance with legislative and policy mandates for security. Secure key management, backup, and administration are a few of the core elements for achieving true data privacy. Because Ingrian DataSecure Platforms have been built from the ground up with security in mind, implementing these policies and procedures is fast and easy, utilizing our unique management interface. We encourage administrators to focus on developing key management and administrative policies for their organizations that will provide maximum security and hope that this thesis will serve as a guide in this effort.

REFERENCES

Books and Publications.

[1] ”Cryptography Decrypted “ By H.X Mel & Doris Baker, Publication Addison – Wesley.

[2] “Fundamentals of Network Seccurity.”By Eric Maiwald, Dreamtech publication Edition –2004.

[3] “Crytography and Network Security Principle and Practice” By Stalling, William, 2ed edition, Pretice Hall, 1999 , ISBN)-13-869017-0. Practical Discussion of Cryptographic principle.

[4] “Computer Networks” By Tanenbaum , Andrews, Parentice Hall of India, New Delhi- 1987

[5] ”Data Communications and Networking”. By Behrouz A. Forouzans Tata McGraw-Hill Publishing Company Limited, Edition- 2000.

[6] “Cryptography in C & C++” By Michael WelschenbBach, translated by David Kramer.

[7] “SQL,PL/SQL Progamming Language of oracle.” By – Ivan Bayross.

[8] “ Starting Out with Oracle covering Database. “ By John Day craig Van Slyke

[9] "Beginning Oracle Programming" By Sean Dillon , Christopher Beck, Thomas Kyte, & Joel Kallman , Shroff Publishers & Distributors Pvt. Ltd.

[10] "Internet & JAVA Programming" By R.Krishnamoorthy, S.Prabhu, New Age International(p)Limited, Publishers, New Delhi – 2003.

[11] " Starting Out With JAVA" , By Tony Gaddis. , Published by Dreamtech , New Delhi-2004

[12] "ICSA Guide to Crytography",By Nicholos, Randy , McGraw – Hill, !999. Comprehensive Discussion of Historical to Modern –day Cryptograpgy.

Internet Resources.

[13] http://ieee.org

[14] http://csrc.nist.gov/encryption/aes/

[15] http://www.ingrian.com

[16] http://www.cs.org

[17] http://www.airdefense.net.

[18] http://www.google.com,

[19] http//www.bitpipe.com/rlist/datasecurity.html. http://www.esat.kuleuven.ac.be/~rijmen/rijndael/

[20] http://www.cis.comell.edu,

[21] http://isaac.cs.berkeley.edu/mobicom.pdf.

[22] http://www.computeruser.com/resources/dictionary/dictionary.html (reference for

[23] technical terms) .

[24] http://www.sciencedirect.com

[25] http://www.scirus.com

[26] http://wwwspringerlink.com

[27] http://www.computerworld.com (provides white papers, surveys, and reports) .

[28] http://www.informationweek.com (provides information on wireless networks, wireless communications, and security solutions in the form of articles and other documents).

[29] http://www.infosecuritymagazine.com (provides white papers, surveys, and reports).

[30] http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html(Universityof California at Berkeley.

[31] RSA Laboratories. PKCS#11 v2.11: Cryptographic Token Interface Standard, Revision 1, November 2001. ftp://ftp.rsasecurity.com/pub/pkcs/ pkcs-11/v211/pkcs-11v2-11r1.pdf Mark Russinovich. Inside Encrypting File System. Windows & .NET Magazine, June -July 1999.

[32] Sun Corporation. The Java^{TM} Cryptography Architecture API Speciation & Reference. http://java.sun.com/products/ jdk/1.2/docs/guide/ security/CryptoSpec.html (valid as of September 2002), December 1999.

[33] Articles and White paper Material.

[34] http://www.ieeexplre.ieee.org,

[35] http://www.ingrian.com

[36] http://www.airdedense.com,

[37] http:// www.itsecurity.com.

[38] http://www.infosecuritymagazine.com

APPENDIX A

PRIMITIVE FUNCTIONS FOR THE DATA ENCRYPTION ALGORITHM

The choice of the primitive functions ***KS, S1,...,S8*** and ***P*** is critical to the strength of the Transformations resulting from the algorithm. The tables below specify the functions ***S1,...,S8*** and ***P***. For the interpretation of the tables describing these functions.

The primitive functions ***S1...S8*** are:

S_1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S_2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S_3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S_4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S_5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S_6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S_7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S_8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

The primitive function ***P*** is :

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Recall that ***Kn***, for 1□*n*□16, is the block of 48 bits in (2) of the algorithm. Hence, to describe ***KS***, it is sufficient to describe the calculation of ***Kn*** from a key (***Keyi***) from the key bundle for *n* = 1, 2,..., 16. That calculation is illustrated in Figure 4. To complete the definition of ***KS***, it is therefore sufficient to describe the two permuted choices, as well as the schedule of left shifts. One bit in each 8-bit byte of ***Keyi*** may be utilized for error detection in key generation, distribution and storage. Bits 8, 16,..., 64 are for use in assuring that each byte is of odd parity. [Note that these eight parity bits have no effect on the operation of the algorithm.]

Permuted choice 1 is determined by the following table

PC-1

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

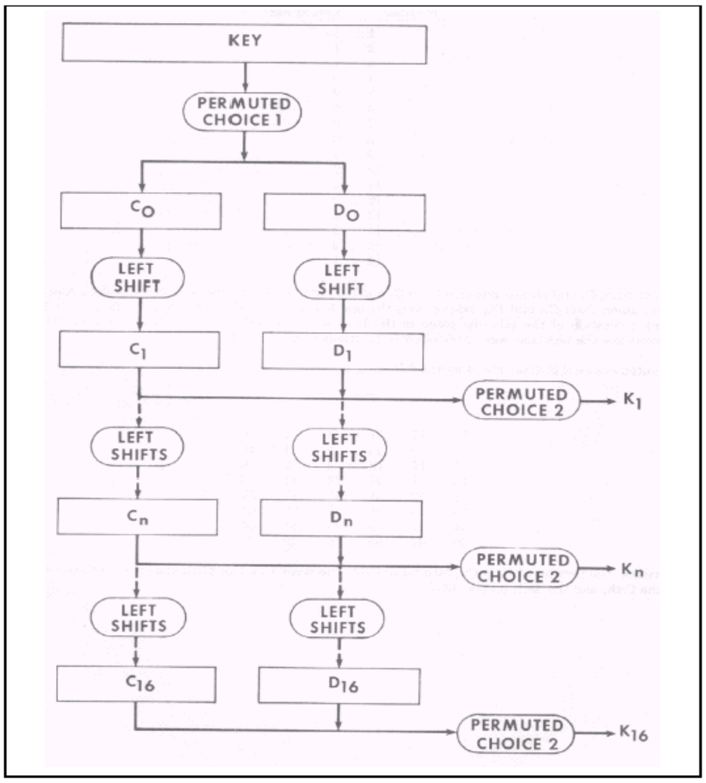


Figure A1: ***Key Schedule Calculation***

The table has been divided into two parts, with the first part determining how the bits of ***Co*** are chosen, and the second part determining how the bits of ***Do*** are chosen. The bits of ***Keyi*** are numbered 1 through 64. The bits of ***Co*** are respectively bits 57, 49, 41,..., 44 and 36 of ***Keyi***, with the bits of ***Do*** being bits 63, 55, 47,..., 12 and 4 of ***Keyi***. With ***Co*** and ***Do*** defined, the blocks ***Cn*** and ***Dn*** are obtained from the blocks ***Cn-1*** and ***Dn-1***, respectively, for *n* = 1, 2,..., 16, by adhering to the following schedule of left shifts of the individual blocks:

Iteration *Number*	Number of *Left Shifts*
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

For example, ***C3*** and ***D3*** are obtained from ***C2*** and ***D2***, respectively, by two left shifts, and ***C16*** and ***D16*** are obtained from ***C15*** and ***D15***, respectively, by one left shift. In all cases, by a single left Initial Public Draft shift is meant a rotation of the bits one place to the left, so that after one left shift the bits in the 28 positions are the bits that were previously in positions 2, 3,..., 28, 1.

Permuted choice 2 is determined by the following table :

PC-2

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Therefore, the first bit of ***Kn*** is the 14th bit of ***CnDn***, the second bit of ***Kn*** is the 17th bit of ***CnDn***, and so on, with the 47th bit of ***Kn*** as the 29th bit of ***CnDn***, and the 48th bit of ***Kn*** as the 32nd bit of ***CnDn***.

APPENDIX B

// Program for Encryption :

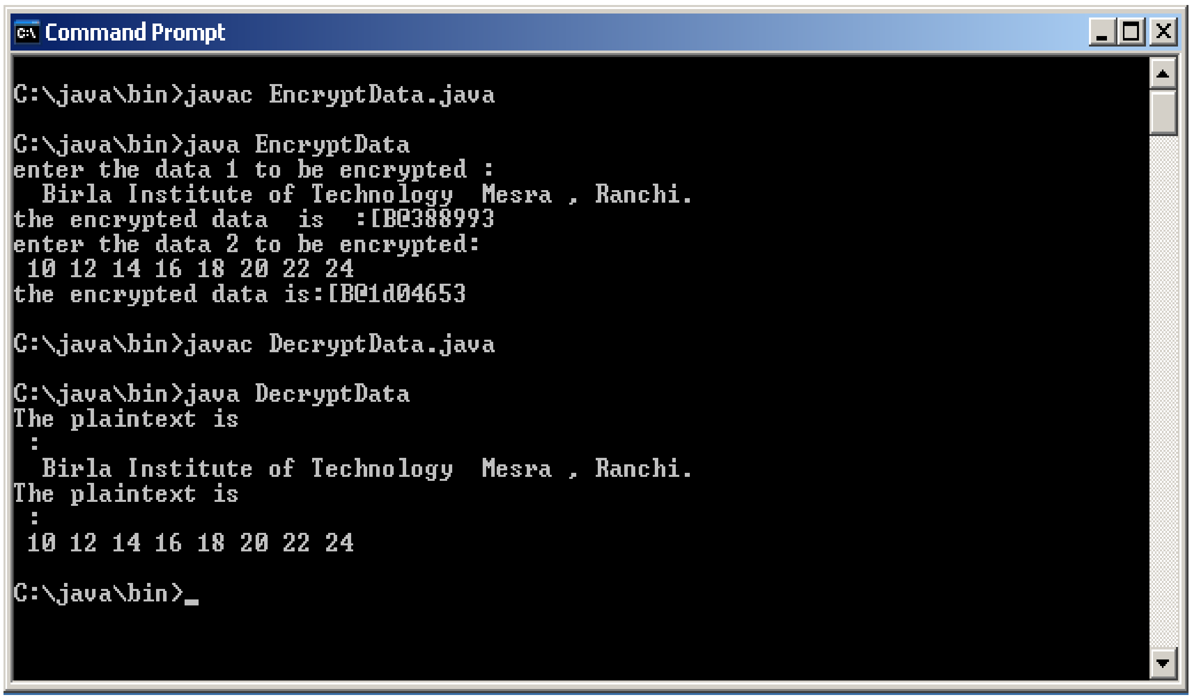
```
import java.io.*;

import javax.crypto.*;

import javax.crypto.spec.*;

public class EncryptData

{

public static void main(String[] args)

  {

    try

    {

            byte[] encryptKey = "This is a test DESede key".getBytes();

      DESedeKeySpec spec = new DESedeKeySpec(encryptKey);


      SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(

        "DESede");

      SecretKey theKey = keyFactory.generateSecret(spec);

      Cipher cipher = Cipher.getInstance("DESede/CBC/PKCS5Padding");

      IvParameterSpec IvParameters = new IvParameterSpec(

        new byte[] { 12, 34, 56, 78, 90, 87, 65, 43 });

      cipher.init(Cipher.ENCRYPT_MODE, theKey, IvParameters);

            System.out.println("enter the data 1 to be encrypted :");

            InputStreamReader dis1 = new InputStreamReader(System.in);

            BufferedReader br1 = new BufferedReader(dis1);

String encr1 = br1.readLine();

      byte[] plaintext1 =encr1.getBytes();

// Encrypt the data

      byte[] encrypted1 = cipher.doFinal(plaintext1);

// Write the data out to a file

System.out.print("the encrypted data  is  :");

            System.out.println(encrypted1);

      FileOutputStream out1 = new FileOutputStream("encrypted1.dat");

      out1.write(encrypted1);

      out1.close();

            System.out.println("enter the data 2 to be encrypted:");

            InputStreamReader dis2 = new InputStreamReader(System.in);
            BufferedReader br2 =new BufferedReader(dis2);
String encr2 = br2.readLine();
      byte[] plaintext2 =encr2.getBytes();
// Encrypt the data
      byte[] encrypted2 = cipher.doFinal(plaintext2);
// Write the data out to a file
System.out.print("the encrypted data is:");
            System.out.println(encrypted2);
      FileOutputStream out2 = new FileOutputStream("encrypted2.dat");
      out2.write(encrypted2);
      out2.close();
    }
    catch (Exception exc)
    {
      exc.printStackTrace();
    }
  }
}
```

APPENDIX C

// Program for Decryption :

```
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.security.*;
public class DecryptData
{
  public static void main(String[] args)
  {
    try
    {
      byte[] encryptKey = "This is a test DESede key".getBytes();
      DESedeKeySpec spec = new DESedeKeySpec(encryptKey);
      SecretKeyFactory keyFactory = SecretKeyFactory.getInstance( "DESede");
      SecretKey theKey = keyFactory.generateSecret(spec);
      Cipher cipher = Cipher.getInstance("DESede/CBC/PKCS5Padding");
      IvParameterSpec ivParameters = new IvParameterSpec(
        new byte[] { 12, 34, 56, 78, 90, 87, 65, 43 } );
      cipher.init(Cipher.DECRYPT_MODE, theKey, ivParameters);
      File encryptedFile1 = new File("encrypted1.dat");
      byte[] encryptedText1 = new byte[(int) encryptedFile1.length()];
      FileInputStream fileIn1 = new FileInputStream(encryptedFile1);
      fileIn1.read(encryptedText1);
      fileIn1.close();
    // Decrypt the data
   //////// ////added
 ////System.out.println("The encryped data is\n :");
 ////System.out.println( encryptedText1);
  //////// ////end added
      byte[] plaintext1 = cipher.doFinal(encryptedText1);
     String plaintextStr1 = new String(plaintext1);
      System.out.println("The plaintext is\n :");
      System.out.println(plaintextStr1);
/////second data
File encryptedFile2 = new File("encrypted2.dat");
    byte[] encryptedText2 = new byte[(int) encryptedFile2.length()];
   FileInputStream fileIn = new FileInputStream(encryptedFile2);
      fileIn.read(encryptedText2);
      fileIn.close();
// Decrypt the data
/////////////////added
///System.out.println("The encryped data is:");
///System.out.println(encryptedText2);
//////////////end added
    byte[] plaintext2 = cipher.doFinal(encryptedText2);
    String plaintextStr2 = new String(plaintext2);
      System.out.println("The plaintext is \n :");
      System.out.println(plaintextStr2);
//////end second data
    }
    catch (Exception exc)
    {
      exc.printStackTrace();
    }
  }
}
```

//// **Implementation of the thesis work in java . :**



```
C:\java\bin>javac EncryptData.java

C:\java\bin>java EncryptData
enter the data 1 to be encrypted :
  Birla Institute of Technology  Mesra , Ranchi.
the encrypted data  is  :[B@388993
enter the data 2 to be encrypted:
 10 12 14 16 18 20 22 24
the encrypted data is:[B@1d04653

C:\java\bin>javac DecryptData.java

C:\java\bin>java DecryptData
The plaintext is
 :
  Birla Institute of Technology  Mesra , Ranchi.
The plaintext is
 :
 10 12 14 16 18 20 22 24

C:\java\bin>_
```

Figure: AC - Output of the java Programm which is written above .