Security Risks in the Encryption of Database Connection Strings

Ross Rannells and James H. Hill
Luddy School of Informatics, Computing, and Engineering
Indianapolis, IN USA
Email: rrannell@iu.edu, hilljh@iupui.edu

***Abstract*—This article presents a novel approach to obfuscating database connection strings using Keyword Cipher, which is based on the Roman Caesar Cipher and Greek Scytale Cipher. This is an important and open problem because database connection strings typically have long substrings of identical and well-known character substrings. These known substrings in related database connection strings greatly increase the risk of their encryption key's being broken, in addition to having identical initial substrings in their encrypted versions. Our experience applying these two obfuscation techniques to database connection strings show that the simple and easily implemented string obfuscation functions effectively solve the problem of common initial substrings. It also greatly reduces risk of breaking the connections strings encryption keys by hiding the known substrings and making the number of possible string needed to search for grow geometrically. Lastly, the use of obfuscation functions completely eliminates all the commonality between related database connection strings.**

***Index Terms*—database connection strings, encryption, decryption, symmetric encryption, Scytale, AES, DES**

I. Introduction

Almost all modern software applications use some form of a persistent data store, such as a relational database. These applications connect to the database using a connection string, which is a value that contains pertinent information like the database's location on the network, any runtime connection parameters, and the database engine. Traditionally, database connection strings are stored in plain text. For example, you can search many open-source software projects and find plain-text database connection strings. The same holds true for proprietary software applications. In cases where the database connection string is stored in an environment variable to "increase" security [16], the database connection is still stored as plain-text.

Although using plain-text database connection strings is an acceptable norm, there is a problem with this approach. The primary problem is it makes it easy for malicious actors to access the information in database because they have direct access to all the necessary credentials to make the connection. To overcome the plain-text problem, it is possible to store an encrypted version of the database connection string in the source code, initialization file, or an environment variable. [10] The database connection string is then decrypted before it is used to connect with the database.

This approach offers an improved level of security, but it has one major problem. The problem is not with the encryption algorithm used, such as the Data Encryption Algorithm [7] (3DES), Advanced Encryption Standard [1] (AES) or any other modern encryption algorithm. The problem is database connection strings being encrypted have common attributes that appear in all database connection strings regardless of the application. For example, a Java Database Connection (JDBC) string will always have the word "java" in the connection string. [6] Because of this fact, it is possible to eventually learn the encryption key used to encrypt the database connection string. Consequently, when the same encryption key is used to encrypt other data on the system, a malicious actor can gain access to that data as well.

One solution a developer may use to overcome common attributes in a database connection string would be to simply encrypt the encrypted connection string a second time with a different encryption key. Unfortunately, this will not solve the problem as you would end up encrypting a different string of common characters, and you would end up with a new set of common characters. We therefore propose a solution approach that introduces randomness into the database connection string that would enable the cipher algorithm to hide the common characters in the database connection string. More specifically, we introduce the randomness using a pair of ancient message obfuscation functions: the Roman Keyword cipher [4] and the Greek Scytale cipher [17]. With this understanding, the main contributions of this paper are as follows:

- It showcases the security limitations of using encrypted database connection strings in production applications;
- It explores using two different obfuscation functions to introduce randomness into the database connection string to reduce a malicious actors ability to learn the encryption key; and
- It empirically evaluates using the obfuscation functions in conjunction with the database connection strings.

Lastly, the results of our work show that encrypting database connections string without applying any obfuscation confirms that the commonality within the plain text database connection strings were clearly reflected in the encrypted text strings. Also, when the keyword obfuscation function was applied to database connections strings, the commonality between the database connection strings was no longer visible in the encrypted strings. We were also able to achieve the same results with the Scytale obfuscation function. Both obfuscation

functions were therefore able to introduce a sufficient number of random characters into the connection strings to hide the commonality between them.

Article organization. The remainder of this article is organized as follows: Section II introduces a case study related the database connection string problem; Section **??** presents our experimental approach to the applying of the Scytale and Keyword obfuscation function to database connection string; Section III presents our experimental results; Section IV discusses the threats to validity; Section V compares our work to existing related works; and Section VI provides concluding remarks and lessons learned.

II. Background Information

Java database connections strings have a set format and contain some easily known string constants. The standard format begins with the four-character string `jdbc`, signifying that it is a Java database connection string. That is followed by a colon connector and the constant value representing the database engine being connected to. This is followed by another colon and possibly a pair of forward-slashes, depending on the target database engine. The slashes are then followed by the location of the database, its IP address, URL, or some other way to locate the database. The database port is preceded by another colon connector. Finally, a forward-slash separator is used followed by the name of the database being connected to. Additional qualifiers and optional name value pairs may also be included in the database connection string. Listing 1 gives an illustration of the Java Database Connection String, and its parts.

```
jdbc:[engine]://[host:port]/[name]
```

Listing 1. The format of the Java Database Connection String.

While not strictly part of the connection string format, many modern database engines have extensions that allow the target account/username and password to be included in the connection string. Another non-standard extension that has become common place is the use of qualifier values to indicate the application level for a database such as:

- **`dev`** – This qualifier is use for the development database;
- **`tst`** – This qualifier is use for the test database;
- **`int`** – This qualifier is use for the system integration testing database;
- **`uat`** – This qualifier is use for the user acceptance testing database; and
- **`prd`** – This qualifier is use for the production database.

These qualifiers can appear in either the address or name of the database depending on how the database and network are set up. Table I provides example of connection strings for different databases. As shown in this table, each has similar characteristics to the Java database connection string.

In each example, the database connection string begins with a preset number of known characters. The exact number differs with each type of database engine, but if an attacker knows the type of database being connected to, the number of known initial characters in the database connection string increases significantly. Additionally, with some of the more modern databases, the connection strings contain additional known key values that increase the number of known initial characters at the cost of a small increase in the number of possible initial strings. This therefore raises a security question: how do the known initial character strings affect the encryption of a database connection string?

Assuming that a development group has multiple different databases for the development life cycle. For example, let's assume the group uses the following key values to signify the various levels of databases: development (dev), testing (tst), integration (int), user acceptance (uat) and production (prd). This gives five different databases that an application will need to connect with as it moves through the development process. This also means that there are five different property files, each with its own slightly different database connection string stored in the code repository. To understand the problems associated with maintaining several different database connection strings, let's examine how each connection string gets encrypted using the symmetric encryption algorithms 3DES and AES—two of the most widely symmetric encryption algorithms.

3DES. For each example, we will use the following 3DES Encryption Key and Initialization Vector (IV) shown in Listing 2. The encryption key is 168 bits long and the initialization vecter is a set of 8, 8 bit integers.

```
Key: v5JKf8QfzUDWWOOAkmcVTMQlXiPpL/ib
IV:  124, -30, -66, 119, -102, 89, -41, 74
```

Listing 2. 3DES Encryption Key and Initialization Vector

Using the aforementioned encryption key and initialization vector to encrypt the example database connection strings in table II. The resulting encrypted connection strings all have the same initial 10 characters.

Based on the encrypted results provided in Table II, we can see that encrypted strings are not independent of each other. In each of the examples strings, the first 10 characters are identical in each of the encrypted strings. This is the result of the first 64 bits of the encrypted text being identical. Once those initial bits are Base 64 encoded, the first 10 characters of the encrypted string end up being identical. The key factor in this is that each database connection strings start with more than a dozen identical characters. Unfortunately, this is larger than the block size of the 3DES encryption algorithm—meaning each of the encrypted string have the same initial substring value.

AES. For each example, we will use the following AES Encryption Key and IV shown in Listing 3. The key is 256 bits in size and the initialization vector is a set of 16, 8 bit integers.

```
Key: 98PG7rAqhrJSD1x03sSKTmzNxr26DnLkeitsjg8NKYk=
IV: 27, 21, 28, 14, 47, -71, -78, -119, 65, -97,
    -72, 118, 122, -62, -64, 65
```

Listing 3. AES Encryption Key and Initialization Vector

Using 256-bit AES encryption algorithm with the aforementioned key and initialization vector on the example MYSQL

TABLE I
EXAMPLE DATABASE CONNECTIONS STRINGS

Database Engine	Eample Connection String
MySql	jdbc:mysql://dev.rolodex.abcd:3505/contacts
SQLite	jdbc:sqlite:/dev/apps/rolodex/db/contacts.db
MS SQL Server	jdbc:sqlserver://dev.rolodex.abcd:3505/devContacts
Maria	jdbc:mariadb:User=DevUser;Password=DevPswrd;Database=contacts;Server=dev.rolodex.abcd;Port=3505
Oracle	jdbc:oracle://dev.rolodex.abcd:3505/contacts
Postgres	jdbc:postgresql://dev.rolodex.abcd:3505/contacts
Sybase	jdbc:sybase:Tds:dev.rolodex.abcd:3505?ServiceName=contacts
DB2	jdbc:db2://dev.rolodex.abcd:3505/contacts;
Cassandra	jdbc:cassandra://dev.rolodex.abcd:3505/contacts:User=DevUser:Password:DevPswrd

TABLE II
MYSQL CONNECTIONS STRINGS WITH 3DES ENCRYPTION

Key	Plain Text	3DES Encrypted Text
dev	jdbc:mysql://dev.rolodex.abcd:3505/contacts	9I7YGlG+YvA7JLp+AYwYVJqcSrCWMhNsFesYvcQ4e2aA+TvXyUQLe9MZiRtu09jD
tst	jdbc:mysql://tst.rolodex.abcd:3515/contacts	9I7YGlG+YvCQp0c8HSehntTUfEw8cW5UU7JQ9bQBfNnlL274lTouK1K/3yfhb2xN
int	jdbc:mysql://int.rolodex.abcd:3525/contacts	9I7YGlG+YvBnW5u6TE/rgLKbH9l5FWVav9HWfMzgl8+mMheo9WoafNBSj7t/SEyU
uat	jdbc:mysql://uat.rolodex.abcd:3535/contacts	9I7YGlG+YvDwqXoxJjnPRxlSMshX7Of2ggADMwEPc4YBhQz/r8EMHzEdwjTLVTvv
prd	jdbc:mysql://prd.rolodex.abcd:3545/contacts	9I7YGlG+YvCoaRMHFefrbYqsmPW2bc5gP0lfFHOLpIlRVzmXeNBHHOiaBp3HiHyj

database connection strings results in the encrypted strings shown in Table III. The results show that the 128 bit block size of the AES algorithm is sufficient to remove the commonality seen in the 3DES encryption.

When applying 256-bit AES encryption key to a Microsoft SQL Server database connection strings, however, exposes the commonality of the encryption string in the encrypted text as shown in Table IV.

As shown in Table IV, even AES's 128 bit block size not sufficient enough to hide the fact that the initial 17 characters in a Microsoft SQL Server database connection string are identical. For example, the initial 128 bits of the encrypted text are identical, which translates to the first 21 Base 64 encoded characters being identical in the encrypted text.

While the 256-bit AES algorithm does appear to solve the problem for the MySQL, the problem still exists for longer connections stings like SQL Server. In fact, it becomes apparent that the problem is not in the algorithms themselves but in the block size that the algorithm uses in its encryption process. When the number of initial identical characters is smaller than the block size (128-bits or 16 characters), the AES algorithm was able to generate what appears to be a string of random characters. When the number of initial identical characters is equal to or greater than the algorithm's block size, then the encrypted text has identical initial substrings.

A. Prepending a random keyword

An early method for obfuscating text was the Roman Ceasar Cipher. This obfuscatation function would shift the characters in the alphabet a given number of characters the right or left. A shift of three characters to the right would map the letter A to the letter D, B would map to E, C to F and so on. A common variation on the Caesar Cipher is the Keyword cipher. Which takes a keyword and places it at the start of the alphabet. Thus the First letter of the keyword would be mapped to A, and the second letter in the keyword would map to B, the tird to C and so on until the all of the uniques letter in the keyward are mapped to a letter in the alphbet. The remaining unused letters of the alphabet are placed after the keyword to map them to the rest of the alphabet.

Adopting this approach to obfuscate a database connection string such that similar connection strings (as shown in Table V) are no longer similar after being encrypted? Table V shows the results of prepending a set length random string (keyword) to each of the example database connection strings from Table V.

Table VI, uses the aforementioned AES encryption key and IV to encrypt the Microsoft SQL Server connction shtrings. Once again prepending each of the database connection strings with a fixed length string of random characters gives us improved results. Each encrypted database connection string appears to be just a collection of random characters. More importantly, they do not have anything in common with any of the other similar database connection strings. For example, each of the strings was prepended with a different random keyword. If we had prepended the connection strings with the same random keyword, then the problem of having common characters would have still existed.[1]

B. Inserting random characters

Another early message obfuscating methodology used by the Greeks was the Scytale. It obfuscated a message by wrapping the message around a rod of a fixed diameter and writing the message along one side of the rod. Additional characters were then added to the message along other sides of the rod. The premise of this approach was the only way to remove obfuscation from (or decrypt) the message was to have a rod of the same diameter.

To apply the Scytale approach to encrpyting a database connection strings, a single random character is prepended to

[1]The encrypted database connection string was truncated at 70 characters because its encrypted string length was too long to include in the article.

TABLE III
MYSQL CONNECTIONS STRINGS WITH AES ENCRYPTION

Key	Plain Text	AES Encrypted Text
dev	jdbc:mysql://dev.rolodex.abcd:3505/contacts	de88jjyWHinV9143H2sQjCslzYlyf0PcRNQceTCQ7TjspGR8kSTuG1bycKvZglLy
tst	jdbc:mysql://tst.rolodex.abcd:3515/contacts	2WaVtXM+wNAtbzLD0OuaHx8iZspSaybehvE9t3ShPNRhx01nRv2kMP0+sffHVHiG
int	jdbc:mysql://int.rolodex.abcd:3525/contacts	UGif+NvGNz2daQxR9cIgtnE/KbkQOgWYJYPhVqeDUVVeBGtwDoGm5ttUZdm149Dn
uat	jdbc:mysql://uat.rolodex.abcd:3535/contacts	sDLi1zg2u6usqiFvodJXSi9Aq3ruykO+TezFZYozl8e+ZZCplnpTQyOfKaN1lqQj
prd	jdbc:mysql://prd.rolodex.abcd:3545/contacts	k7RRxlncgYQtsjATXoc3XeYIXSBJbzBGvKFBBk5sxbbiTa99AwBfrsikZXqCyB1k

TABLE IV
SQL SERVER CONNECTIONS STRINGS WITH AES ENCRYPTION

Key	Plain Text	AES Encrypted Text
dev	jdbc:sqlserver://dev.rolodex.abcd:3505/contacts	OzpuFGmb1p0UODW8QfTMr0tnaH1LzUbfoPghSUDWcsJdkY4iatflmOWwfaNDBou4
tst	jdbc:sqlserver://tst.rolodex.abcd:3515/contacts	OzpuFGmb1p0UODW8QfTMrxPfL44TcvMNsEeUq+f3xawWIvOCawLJTpZe0RnG84pk
int	jdbc:sqlserver://int.rolodex.abcd:3525/contacts	OzpuFGmb1p0UODW8QfTMr4JHo2NdW5Dc49rkj3ozFPu7K2qIxbnzw6RCI8p0icD6
uat	jdbc:sqlserver://uat.rolodex.abcd:3535/contacts	OzpuFGmb1p0UODW8QfTMr6QLphclAYDPdNk00LekdpIm5xXGi6FBHQBlRp3jE18q
prd	jdbc:sqlserver://prd.rolodex.abcd:3545/contacts	OzpuFGmb1p0UODW8QfTMr948BvFEBhC3EEVhWfJqTXizAOj/L8xXjzMriEz8T/Vd

TABLE V
KEYWORD OBFUSCATED MYSQL CONNECTION STRINGS WITH 3DES ENCRYPTION

Plain Text	3DES Encrypted Keyword Text
jdbc:mysql://dev.rolodex.abcd:3505/contacts	ukB+uuwS4sDGySZP0/Qd7PHuxK7f/Mtr3vJwXUAC702d/1Afr65X5kfqd5Bn3+ZjFcfggs6WfxE=
jdbc:mysql://tst.rolodex.abcd:3515/contacts	ukB+uuwS4sDGySZP0/Qd7PHuxK7f/Mtr3vJwXUAC702d/1Afr65X5kfqd5Bn3+ZjFcfggs6WfxE=
jdbc:mysql://int.rolodex.abcd:3525/contacts	RhndRDZOU1Aa6dUNiw65GSiDpLvHXUKN5YQ2m0asktGk2JBqXLEGhSNuHclcHS/xm011kj4qTps=
jdbc:mysql://uat.rolodex.abcd:3535/contacts	Bs9ltqNBCjRqg3CuOhtfJcFXoePBaHSgHRbD2TBCdXUP3ngoxm0p7+syD9V5vNsmZwLr6y/hSN4=
jdbc:mysql://prd.rolodex.abcd:3545/contacts	DADV/A79y+tabE2zc2CksPKJBJMGuYXdkV6/DQKZ5i7T4x683V4ASFetO7c2h6yRDykSClqqvWY=

TABLE VI
KEYWORD OBFUSCATED MICROSOFT SQL SERVER CONNECTION STRINGS WITH AES ENCRYPTION

Plain Text	Aes Encrypted Keyword Text (truncated at 70 chararacters)
jdbc:sqlserver://dev.rolodex.abcd:3505/contacts	5HAOS5P6CvjASJ03mh0mKE6lflcddqPTvg2R7A92J4ow5MXRKOT353Xpf+YyivNJMEjJ3I
jdbc:sqlserver://tst.rolodex.abcd:3515/contacts	A3LfiePD7e3o+xxHv1Mwr8+Q9r/l5u+WKII+eA//7TMnOXzxIxRpRuPcffLqIMTiXi6bnr
jdbc:sqlserver://int.rolodex.abcd:3525/contacts	h7WDDjSaekTUPik+O47vbVN8hfTcYhg2Zknh/LBW6yymYQVUeFSwEXJpGfiU7uQtDXEtNB
jdbc:sqlserver://uat.rolodex.abcd:3535/contacts	VL1Gxhz2pt5xwvl+juAq5it/1RCVWZOSfzBlsz8ITMOKREQA0pCABhkH3Z6nygIEwtw0Hg
jdbc:sqlserver://prd.rolodex.abcd:3545/contacts	ZBhNk5OxF/OaocCFuhMjxqNNFEwvvOsQ62Fgj9yurx4gmzreGm7k4mB032H58c7HCXp+jg

the front of the database connection string and then a random character is inserted after each character in the connection string. This approach will more than double the length of the database connection string to be encrypted. It will also add far more random characters to the connection string than the keyword obfuscation function does. Table VII shows the results of applying the Scytale obfuscation function to the MySQL database connection strings from Table VII.[2]

Likewise, Table VIII shows the results of applying the Scytale obfuscation function to the Microsoft SQL Server database connection strings from Table VIII.[3]

The use of the Scytale obfuscation function has the same effect on the resulting encrypted database connection strings as keyword obfuscation function. The encrypted connection strings appear to be a string of random characters with no commonality to any of the other database connections strings. Since both methodologies produce the desired results, the question now becomes which of the two methodologies is superior or are they equivalent? We will answer this question in the remainder of the article.

[2]The encrypted database connection string was truncated at 70 characters because its encrypted string length was too long to include in the article.

[3]The encrypted database connection string was truncated at 70 characters because its encrypted string length was too long to include in the article.

III. RESULTS

The Java application used to encrypt the various database connection strings in Table II through Table VIII used the Bouncy Castle encryption libraries. The application also allows for a user to enter a known symmetric cipher keys and initialization vectors that can be used to encrypt and decrypt the database connection strings. The Java application can generate encryption keys for 168-bit 3DES, 128-bit AES, 192-bit AES, and 256-bit AES cipher functions, but only the 168-bit 3DES and the 256-bit AES algorithms were used in the above testing. The application can also generate random initialization vectors for use with the 3DES and AES symmetric cipher algorithms. The application can also apply the two obfuscation functions tested above to the database connection strings. The first obfuscation function,the Keyword function, (see Table V and Table VI) prepends from one to ten random base 64 characters to the beginning of the connection string. The second obfuscation function, the Scytale function, (see Table VII and Table VIII), prepends a

TABLE VII
SCYTALE OBFUSCATED MYSQL CONNECTION STRINGS WITH 3DES ENCRYPTION

Plain Text	3DES Encrypted Scytale Text (truncated at 70 chararacters)
jdbc:mysql://dev.rolodex.abcd:3505/contacts	8l4zsIB4XG3K+zehVqZPHLR76h6GGJUncwo7WWvdfzJ3ahteOI7bid/CyDoyBg+JLCjqJT
jdbc:mysql://tst.rolodex.abcd:3515/contacts	vYeXtxSVdpvM4gHqwQY6tE+i8OfpTsBqkDn7F1yhAes+tE+KqCi/6FX2nLPzxCtWvKA1ly
jdbc:mysql://int.rolodex.abcd:3525/contacts	iZGSyCXkj9qLjfHCLg4kc7eV7fWhB7I7VAdoEmN5pqtZw6yMkRi9+n68mMCdGO7h/vnjHK
jdbc:mysql://uat.rolodex.abcd:3535/contacts	zp5fNcpprQ0HaB+M+b8pEqclklpEQ4arMIWt6zN75wKBAv90vT/li6MH6iHMWnG2jlKHxa
jdbc:mysql://prd.rolodex.abcd:3545/contacts	k952+Wa+l1inYtQmOfT2uhJ4Ki9lYosBXDOnGDQ67F7w37oCv9ZvT43G4M27QrYsnF2GKe

TABLE VIII
SCYTALE OBFUSCATED MICROSFT SQL SERVER CONNECTION STRINGS WITH AES ENCRYPTION

Plain Text	AES Encrypted Scytale Text (truncated at 70 chararacters)
jdbc:sqlserver://dev.rolodex.abcd:3505/contacts	ytJ9EpvKAoMrCMuLqiwnPBYIPOMUgadqrQ9PPNqnVr14IPzp+I8uYveNuf5Y4e/2dMitMe
jdbc:sqlserver://tst.rolodex.abcd:3515/contacts	F9hWo0XDSuxbKO07nZTuAMsc4+EwbgVKM19+b8qxDo1DbmR8hLAtVIdrYIYT4mxy/CIoHl
jdbc:sqlserver://int.rolodex.abcd:3525/contacts	ErEmAgeQ1hBz1SaCRaMffjTZPUeRpl0DEhnZu87osKD8+8T/myHuXLIo1PvbsriY1n7RZ8
jdbc:sqlserver://uat.rolodex.abcd:3535/contacts	SPowc8tFmwBCRevzKggi7qYOG/gi7G97Zxou9W6gaiCmRFJ3kIg6944BuLg/FLjJ97+6Tj
jdbc:sqlserver://prd.rolodex.abcd:3545/contacts	OZFv5B/EOpba0Ugt78GmDmpb+HkuBR5uzEaemuzWUAN7mjOsxxorWUDhbZtIQIiGzFM54f

single random base 64 character to the front of a connection string and an addition random base 64 character between each character in the connection string. A third obfuscation function is also available, which applies both the Scytale and then the Keyword obfuscation functions to the connection strings. The application can also decrypt the encrypted string and undo any of the applied obfuscation function.

Further testing of the encryption algorithms on the example connection strings with the available cipher algorithms (168-bit 3DES, 128-bit AES, 192-bit AES, and 256-bit AES) in the application yielded the following results: the 128-bit version of AES did show some improvement over the 3DES results. For a couple of the database connection strings the commonality in them was hidden by the AES algorithm while in others the commonality remained blatantly obvious. Unfortunately, the results from testing 192-bit AES did not show any improvement over the 128-bit version of the algorithm. The results of the testing with the 256-bit AES were similar to the results of 128-bit and 192-bit versions of the AES algorithm. In all three versions of the AES cipher function, the database connection strings that had their commonality hidden by the encryption process were the same and the database connections strings where the commonality remained obvious were also the same. This led to the preliminary conclusion that it was not the symmetric algorithm, nor the strength of their encryption that determined if a database connection string retained its inherent commonality in the encrypted string but the block size that the encryption algorithm used.

A. Database Engine Comparisons

From the testing done with the symmetric encryption algorithms (3DES and AES) when applied to various example plain text database connections strings. It becomes clear that it was not the algorithm or the strength of the key that is the deciding factor in determining if resulting encrypted strings appearance of independence from each other. That it was based on the bock size of the encryption algorithm and the number of preset characters in the connection string. Which means that the index of the first unique character in a database connection string is quintessential in knowing if a symmetric encryption algorithm will be able to generate encrypted text that will appear to be random character string. The second column of table IX shows the minimum length for of the initial character sub-string for nine popular database engines, note that due to SQLite′s dependence on the host operating system′s file structure there is a difference between Unix and Microsoft minimum length. The third column of table IX shows how many different initial connection strings can occur with the given database engine. While the fourth column of table IX shows how many known sub-strings can appear within a Java database connection string for the database engine.

From the previous testing, we know that if the initial sub-string is equal to or greater than the block size of the encryption algorithm, then there will be a clear correspondence in the encrypted versions of the database connection strings. In five (Postgres, MS SQL Server, Cassandra, Maria, and SQLite on an MS System) of the above cases not even AES′s 128-bit block size will be sufficient to hide commonality between database connections strings for those database engines. Add to that any even a small commonality in the server name or IP address and all but DB2 would likely cross the 128-bit block size boundary of the AES algorithm. Even DB2 can only handle 4 additional common characters in the initial part of the database connection string for AES to be able to conceal the commonality between the various database connection strings.

Once the preliminary conclusion that the cipher algorithm′s block size was the reason why the commonality within the database connections strings was being reflected in their encrypted strings. It became clear that there needed to be some way to introduce some easily removable random characters into the initial part of a database connection strings. This randomness would allow the symmetric encryption algorithm′s permutation and substitution functions to properly hide the commonality within the database connection strings. This is the reasoning that led to the decision to introduce random characters into the database connection strings by applying the Keyword function, Scytale function or both functions to

TABLE IX
DATABASE CONNECTIONS STRING STATISTICS

Database Engine	Preset	Options	Known Text
MySql	13	1	2
Oracle	14	1	2
Postgres	18	1	2
MS SQL Server	17	1	2
Sybase	14	1	2
DB2	11	1	2
SQLite (Unix)	13	1	2
SQLite (MS)	16	24	2
Cassandra	22	3	5
Maria	17	3	7

obfuscate the database connection strings.

B. Applying the Keyword Obfuscation Function

The keyword obfuscation function, as implemented in the test application, allows for the prepending of from one to ten random base 64 character to database connection string. The application allows the user to set the length of the string prepended, but it defaults to a string of five characters. The application randomly selects a different character string to be prepended to each of the database connection strings. This uniqueness of each of the prepended strings is necessary, since if the same string was used for all of the connection strings, then the commonality would still remain in the plain text and the commonality would be reflected in the encrypted text. The function only slightly increases the size of the connection string and does significantly increase the size of the resulting encrypted string. The keyword obfuscation function is a very simple to implement and can be easily undone.

From the results of further testing of the Keyword Obfuscation function, it became clear that the addition of five random characters to the front of any database connection string was sufficient to remove the commonality that exists in the plain text connection string from the connection string′s encrypted text representation. This is due to fact that the random character string introduces a likely to introduce a unique character into the part of the database connection string within the first five characters of the string. In fact, the chance of two randomly generated five-character base-64 strings being identical is one in 64 to the fifth, less than 1 in a billion. This is likely sufficiently secure for most situations but if a more secure string is needed, then increase the number of random characters in the prepended string. The only caveat is that the size of the prepended string cannot be multiple the cipher algorithm′s block size without losing its effectiveness. If 3DES is being used to encrypt the connection string, then the prepended random character string cannot be a multiple of eight.

Further testing also showed that prepending a single character to the front of the connection string was insufficient to safely hide the commonality in the database connection strings. The 1.5625chance of getting a duplicate character at the start of the database connection string was simply too high when encrypting large numbers of connection strings. Even a two random character string has a roughly 1 on 4000 chance of getting a duplicate random string value. The author would recommend using at least a three-character random string to prepend to the database connection string.

C. Applying the Scytale Obfuscation Function

The Scytale obfuscation function, as implemented in the test application, prepends a random character to the beginning of the database connection string and inserts another random character after each of the characters in the connection string. The characters that are added to the connection string are randomly selected. The Scytale function more than doubles the length of the connection string which greatly increases the length of the resulting encrypted string also. The Scytale function is more complex to implement and is more time consuming to both apply and remove from the database connection string than the keyword obfuscation function, but the increases should not significantly affect the run-time of any application making use of the Scytale obfuscation function.

The results of the testing of the Scytale obfuscation function above, much like the keyword obfuscation function, showed that it could easily remove the commonality in the database connection strings from the encrypted text strings. They were both able to produce encrypted strings that appeared to be a string of random characters. The key difference between the two obfuscation functions is that the Scytale function does not suffer from the problem of being ineffective when the number of random characters introduced into the connection string is a multiple of the cipher algorithm′s block size. The Scytale obfuscation function as implemented is able to remove the commonality in a database connection string without any limitations on the number of characters added to the connection string.

The key problem with the Scytale obfuscation function is that it more than doubles the length of the connection string. If a database connection string is 40 characters long, the Scytale obfuscated version of the plain text string would be 81 characters long, which would lead to an appreciable increase in the length of the encrypted string. While the length of the encrypted string may not be of paramount importance in most use cases, in those where the length does matter then the Scytale function may not be an appropriate solution.

D. Shortcomings of AES and 3DES

It is clear from the test results that neither 3DES nor the AES cipher algorithms are sufficient to safely encrypt most database connection strings. 3DES′ 64-bit block size is smaller than all of the connection strings minimum preset character lengths which means that 3DES cannot hide the commonality within any of the tested database connection strings. While the AES cipher algorithm did show some improvements over the 3DES algorithm. AES was able to hide the commonality of some of the database connections strings, so long as the first unique character appeared in the connection string within the first sixteen characters, the AES block size, of the connection string. The AES cipher algorithm was able to hide the commonalities of some database connection strings including MySql and SQLite. Unfortunately for AES, in the case of database connection string with longer preset initial strings like MS SQL Server and Maris, the inability to hide the commonality in the connection string still exists. In further testing, the fact that the problems persisted regardless of the strength of the cipher key showed that the problem is not with the encryption key or initialization vector but in the block size used by the cipher algorithm.

The application of both the Keyword and Scytale obfuscation functions before the encrypting of the plain text database connection string had the same effect on the resulting encrypted test string regardless of whether the 3DES or AES cipher algorithm were used to encrypt the obfuscated string. With both cipher algorithms, the resulting encrypted text appeared to be a string of random characters. While both algorithms were able to produce apparent random strings, they both have some disadvantages. In the case of the keyword obfuscation function, the end user must be cognizant of the block size of the cipher algorithm to be used to encrypt the obfuscated text and the number of random characters prepended to the database connection strings. On the other hand, the Scytale obfuscation function will greatly increase the length of the resulting encrypted text strings.

Additionally, note that one of the easiest ways to break a cipher key is to know some of the sub-string that produced the encrypted string. The problem of the known initial values in database connection strings is just a special case where not only do we know some of the sub-strings that produced the encrypted text, but we also know where those strings are located in the plain text, namely at the start of the plain text. With more modern databases, like Maria and Cassandra, the fact that they have embedded in them multiple name value pairs for information like database names, server addresses, ports listened on, accounts and passwords should not be overlooked. Knowing the sub-string such as "password", "server", and "database" are in the plain text, even if they are not at the beginning, can only make the breaking of the encryption key easier.

E. Results the Keyword Obfuscation Function

The keyword obfuscation function was very successful at hiding the commonality that exists in database connection strings. In testing a single character was sufficient to change the look of and encrypted connection string to one that looked like a collection of random characters. Unfortunately, a single character was not sufficiently safe when groups of connections strings are encrypted. Testing suggested that it would take a keyword string of three or more characters to be reasonably safe. Additionally, when the keyword is a multiple of the cipher algorithms block size, the keyword has no effect on the security of the encrypted database connection string.

While the use of the Keyword obfuscation function does make sure the beginning part of the encrypted strings look random, it does nothing to hide the known strings that already exist in the database connection string. The fact that the strings "dbc" and a designater for the database engine are at the start of every database connection string is not way changed by the use of the keyword obfuscation function. The fact that the keyword function only moves the known string over a few places in the string and fills in the gaps with random characters may make it harder to identify the encrypted string, but it does nothing to protect the encryption key from being broken by an intelligent attacker.

F. Results the Scytale Obfuscation Function

The Scytale obfuscation function was also successful at hiding the commonality that exists in database connection strings. It did just as good a job of obscuring the commonality of the database connections strings as the Keyword function but at the cost of greatly increasing the length of the encrypted text and at a slight increase in system resource use. Its ability to hide the commonality in the database connection string was also independent of the block size of the cipher algorithm.

An additional advantage that the Scytale obfuscation function has over the keyword function is that it not only obfuscates the database connection strings, it also obfuscates the known string within the database connection string. While the keyword function hides the known string "jdbc", the Scytale function replaces the string jdbc with a seven-character string that has a random character between each of the known characters in the string. An attacker who tries to break the encryption key using the fact that jdbc is somewhere in the database connection string would now have to check for more then a quarter of a million possible sub-string to account for all of the additional three randomly selected base-64 characters inserted into the four-character jdbc string by the Scytale obfuscation function.

The insertion of the random characters into every part of the database connection string rather than just at the start of the string increases the security of the encryption key. The Scytale obfuscation function not only hides the commonality of the connection strings, it also protects the encryption key use to encrypt the connection string by removing the all know name flag values from the string and replaces them with longer strings with random characters in them.

IV. Threats to Validity

One threat to the validity for this research is it is completely based on the generation of random numbers for the creation of the encryption keys, initialization vectors and the random characters used in both the Keyword and Scytale obfuscation functions. The Java random number generation function was initialized using the system time to minimize the chance of the random number generator function introducing any bias into the results observed in the various tests of the obfuscation functions.

For our analysis to be valid, the software application and the libraries used to generate the encryption keys, initialization vector and encrypted text must be accurate. We therefore used the older Bouncy Castle encryption libraries to mitigate the chance of errors in the encryption and decryption of the plain text.

Likewise, our testing was only applied to the Java database connection strings using the most popular symmetric cipher algorithms. Other cipher algorithms may show that applying Keyword or Scytale obfuscation functions to plain text database connection strings before encrypting the connection string may not yield similar results. Lastly, known limitations of both the Keyword and Scytale obfuscation functions, such as if the length of the keyword prepended to the text is a multiple of the encryption algorithm's block size then the keyword has no effect on the encrypted text. Additionally the length of the keyword prepended to the text directly effects the effort needed to break the encrytion key. A keyword of only a characters or two does not significantly increase the security of the database connection string, while the longer the keyword is, the more difficult it is to break the encryption key so long as it is not a multiple of the algorithms block size. One of the limitations of the Scytale obfuscation function is that it more them doubles the size of the text being encryted. Also, in the case of the Scytale obfuscation function, the length of the keyword directly effects how much more secure the obfuscated text is. The shorter the keyword, the fewer characters inserted inside of it. These limitations may impact the applicability of our approach in certain use cases.

V. Related Work

Sousi et al. [2] presents an in-dept examination of the AES symmetric cipher algorithm. Their analysis directly compares the AES encryption algorithm to DES and shows how AES's is a stronger, safer and, thus far, unbreakable encryption algorithm. Sousi discusses both the advantages and disadvantages of AES and lists several examples of failed attempts to crack AES encryption. Our work is similar to Sousi work in that we showed that AES was not as susceptible to vulnerabilities in database connection string encryption as 3DES was. Additionally, our experiments on encrypting the database connection strings fully supports Sousi's conclusion of AES's is a stronger encryption algorithm when compared to DES. For example, our results showed DES was far more susceptible to the problems with encrypting database connection strings.

Patel et al. [13] did an extensive comparison study of the DES, 3DES, AES, RSA and Blowfish encrytion algorithms. They compared and contrasted algorithms for various parameters including encryption/decryption time, memory use, its avalanche effect, and the algorithm's entropy effect. Our work is similar to their work because their evaluation of the entropy effect of the algorithms exemplifies what our experiments are attempting to affect. By adding random characters to the plain text before encryption, we increased the entropy effect on the plain text. Patel et al's work showed that AES and Blowfish had the highest entroy effects. Our work showed that the 64 bit (8 byte) block size, which Blowfish uses, was a major contributing factor to the commonality in encrypted database connection strings.

Qianru Gong [5] showed how combining both DES/3DES symmetric encryption algorithm with RSA asymmetric encryption algorithm can improve the overall security of electronic communications. By encrypting a message using a DES or 3DES cipher key and then encrypting that cipher key with the recipients public RSA key, a secure one-time pad can be created. The problem with Gong's proposed solution is the use of DES/3DEs. As our experiments show DES/3DES's reliance on a 64-bit block size leaves it vulnerable to long strings of identical characters. While our work does nothing to bring into question Gong's security solution, it does show that AES may be a better choice of encryption algorithm when encrypting message/data due to it larger block size.

Raed Abu Zitar and Muhammed J. Al-Muhammed [18] took a completely different approach to encrypting information by using a two-layer feed-forward neural net for the encryption function. Their approach trains the first layer of neural net to encrypt the plain text. While the second layer decrypts what the first layer encrypted, to recover the original text. The methodology also includes a recurrent neural network that generates a security code that maintains the integrity of the plain text. Our approach of obfuscating the text with random characters would not work well with such an encryption methodology. Due to the fact that the security code for maintaining the integrity of the encryption would be invalid as every time the obfuscated text is encrypted, a different integrity security code would be calculated.

VI. Conclusion

This paper presented our work on removal of commonalities within encrypted database connection by applying simple obfuscation functions. Based on our experimental results, we show that, even with the strongest of symmetric encryption keys, commonalities remain visible in encrypted database connection strings. Our experimental results also show that visible commonalities, however, disappear by adding a short character string to the front of a database connection string. Lastly, further experimentation with a Scytale obfuscation function also eliminated the visible commonalities. Based on our experimental results and research findings, we have learned following lessons:

- **Cipher algorithm block size impacts discernible commonalities.** Plain text database connection strings with no obfuscation function applied demonstrate there are easily discernible commonalities between related database connections strings. This commonality can be traced back to the block size of the cipher algorithm and the length of the preset section of the database connections string. t com In the case of the 3DES cipher algorithm, even the smallest of preset keyword sections of the connection strings were longer than the block size of the cipher algorithm. While AES's longer block size did solve the problem for some connection strings, it did not solve it for all of them. Moreover, failure to produce what appears to be a random character string after encryption increases the chance of the encryption key being cracked.
- **Simple obfuscation function eliminate the discernible commonalities.** Applying either the Keyword or Scytale obfuscation functions to the encrypted text causes the observed commonalities to disappear. Both the Keyword and Scytale obfuscation function had the effect of making the encrypted text appear to be random collections of characters. Only the Scytale obfuscation function manages to remove the known substrings within a database connection strings. A four-character substring would have three additional random characters inserted into it, giving more than a quarter of a million possible strings that would have to be checked when trying to break the encryption key used to encrypt the connection string.
  Therefore the advantages of the Scytale obfuscation function is not only that it makes the encrypted string look like a collection of random characters, it also makes it harder to use the known strings in a database connection string to crack the encryption key. The addition of random characters inserted into the known strings increases the number or strings that have to be searched for by a geometric factor. If using the Base 64 characters, each character inserted into a know string increases the number of string to search by a factor of 64. The keyword java would have over 262,000 possible resulting string after applying the scytale obfuscation function to it. The primary downside to the Scytale obfuscation function is that the resulting encryption string is more than double the length of what the encrypted string would be if the Scytale obfuscation function had not been applied to plain text connection string.
- **Goals achieved, but questions left unanswered?** Both the Keyword and Scytale obfuscation functions achieved the goals set forth for them in our experimental tests. They both removed the obvious commonality in database connection strings. With the Scytale obuscation function being viewed as the superior of the two obfuscation functions. Based on the experimental results we got from our tests, we believe tht that the following questions can be answered in future research efforts.
  The first of these questions is the obvious one, what other types of obfuscation functions could be applied to a database connections string prior to its encryption that will also hide the commonalities within related connection strings. Additionally what advantages and disadvantages might these alternative obfuscation function have over the Keyword and Scytale functions? Another question not dealt with in this article is how other symmetric and asymmetric encryption algorithms would deal with the commonality inherent in related database connection strings. Also would other encryption algorithms benefit from the application of an obfuscation function to the plain text before encrypting it. Another question that needs to be answered, but is beyond the scope of this paper, is just how much the addition of an obfuscation function such as scytale increases the entropy effect of an encryption algorithm over encypting text with no obfuscation function applied. Finally would the addition of more random charaters within the plain text characters significantly increase the entropy effect and would an obfuscation function inserting a small random sized string within the plain text characters further increase the entropy rating of an encryption algorithm?

REFERENCES

[1] "Advanced Encryption Standard", Accessed = 2023/06/03, https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[2] Ahmad-Loay Sousi, Dalia Yehya, Mohamad Joud, "AES Encryption: Study and Evaluation", *CCEE552: Cryptography and Network Security*, 2020, https://www.researchgate.net/publication/346446212_AES_Encryption_Study_Evaluatio

[3] "The Legion of the Bouncy Castle", Accessed = 2023/06/03, bcpg-jdk15on-156.jar and bcprov-ext-jdk15on-156.jar, https://www.bouncycastle.org.

[4] "Caesar Cipher", Accessed = 2023/06/03, https://en.wikipedia.org/wiki/Caesar_cipherx.

[5] Qianru Gong, "Application Research of Data Encryption Algorithm in Computer Security Management", *Wireless Communications and Mobile Computing*, 2022, https://doi.org/10.1155/2022/1463724.

[6] "Connection String Formats", Accessed = 2023/06/03, https://docs.oracle.com/en/database/other-databases/essbase/21/essoa/connection-string-formats.html.

[7] "DES Encryption Cipher", Accessed = 2023/06/03, https://en.wikipedia.org/wiki/Data_Encryption_Standard.

[8] "Cipher Disk", Accessed = 2023/06/03, https://docs.oracle.com/en/database/other-databases/essbase/21/essoa/connection-string-formats.html.

[9] "Enigma Machine", Accessed = 2023/06/03, https://en.wikipedia.org/wiki/Enigma_machine.

[10] Biplov, Apr 7, 2020, "Handling Passwords and Secret Keys using Environment Variables", Accessed = 2023/07/29, https://dev.to/biplov/handling-passwords-and-secret-keys-using-environment-variables-2ei0

[11] "Initialization Vector", Accessed = 2023/06/03, https://en.wikipedia.org/wiki/Initialization_vector.

[12] "Java SE 8 Archive Downloads (JDK 8u211 and later)", 06/03/2023, https://www.oracle.com/java/technologies/javase/javase8u211-later-archive-downloads.html Java SE Runtime Environment - build 1.8.0_221-b11.

[13] Priyadarshini Patil, Prashant Narayankar, Narayan D.G., Meena S.M., "A Comprehensive Evaluation of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish", *Procedia Computer Science, 1st International Conference on Information Security and Privacy 2015*, volume 78, pages 617-624, 2016.

[14] "Known-Plaintext Attack", Accessed = 2023/06/03, https://en.wikipedia.org/wiki/Known-plaintext_attack.

[15] "Type B Cipher Machine", Accessed = 2023/06/03, https://en.wikipedia.org/wiki/Type_B_Cipher_Machine.

[16] Mike Huls, ”Towards Data Science”, ”Keep your code secure by using environment variables and env files”, Nov 1 2021, Accessed = 2023/07/29, https://towardsdatascience.com/keep-your-code-secure-by-using-environment-variables-and-env-files-4688a70ea286
[17] ”Scytale”, Accessed = 2023/06/03, https://en.wikipedia.org/wiki/Scytale.
[18] Abu Zitar, Raed and Al-Muhammed, Muhammed J., ”Hybrid encryption technique: Integrating the neural network with distortion techniques”, *PLOS ONE, Public Library of Science*, volume 17, pages 1-27, 09/2022, https://doi.org/10.1371/journal.pone.0274947.