Sybase® IQ 中的高级安全性

文档 ID：DC01150-01-1520-01

最后修订日期：2010 年 4 月

本文档介绍 Sybase IQ “高级安全性”选项。

Sybase IQ 中的高级安全性

Sybase IQ “高级安全性”选项提供附加安全性机制，可确保最高级别的数据和用户保护。如果数据为用于业务的货币，则确保宝贵数据资产的安全应是优先级最高事项之一。

Sybase IQ “高级安全性”选项的安全业务智能功能包括列加密、针对美国联邦信息处理标准 (FIPS) 认可的加密技术的网络加密支持，以及针对数据库连接及操作系统登录和网络登录的 Kerberos 鉴定。

Sybase IQ “高级安全性”选项提供的增强安全性功能确保符合 FIPS 标准和法规。

“高级安全性”选项是单独许可的 Sybase IQ 选项。

Sybase IQ 中的 FIPS 支持

Sybase IQ 对经美国联邦信息处理标准 (FIPS) 认可的加密技术进行了增强。在 Sybase IQ 支持的所有平台上都支持 FIPS。

FIPS 支持对 Sybase IQ 的主要影响是加密可以是非确定性的，这在目前是缺省行为。非确定性算法是一种相同的输入每次都产生不同输出值的算法。这意味着当使用密钥对字符串加密时，加密字符串每次都不相同。不过，该算法仍可以使用密钥对非确定性结果进行解密。此功能使得对加密算法的分析更加困难，从而使加密更安全。

FIPS 支持是单独许可的 Sybase IQ “高级安全性”选项的组成部分。

Sybase IQ 同时提供 RSA 和 FIPS 安全性。 RSA 加密无需单独的库，但 FIPS 需要以下两个可选库：*dbfips11.dll* 和 *sbgse2.dll*。 *sbgse2.dll* 库是由 Certicom 提供的。这两种安全模型都需要证书。 rsaserver 证书已从 *rsaserver.crt* 重命名为 *rsaserver.id*。

FIPS 还需要以下注册表设置，该设置由 Sybase IQ 安装实用程序自动设置：

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Certicom\libsb]
"expectedtag"=hex:5b,0f,4f,a6,e2,4a,ef,3b,44,07,05,2e,
b0,49,02,71,1f,d9,91,b6
```

有关使用 FIPS 和 RSA 加密的详细信息，请参见 “SQL Anywhere 11.0.1” > 《SQL Anywhere Server — 数据库管理》 > “安全性”中的“传输层安全性”和 “保证数据的安全性”。

Sybase IQ 中的 Kerberos 鉴定支持

Sybase IQ 支持 Kerberos 鉴定，它是一种允许您对数据库连接以及操作系统登录和网络登录维护单一用户 ID 和口令的登录功能。使用 Kerberos 证书，无需指定用户 ID 或口令即可连接到数据库。

Kerberos 鉴定是单独许可的 Sybase IQ “高级安全性”选项的组成部分。

有关使用 Kerberos 鉴定的详细信息，请参见 “SQL Anywhere 11.0.1> 《SQL Anywhere Server — 数据库管理》 > “启动和连接到数据库” > “SQL Anywhere 数据库连接”中的 “Kerberos 鉴定”一节。

Sybase IQ 中的列加密

Sybase IQ 数据库文件的强加密使用 128 位算法和安全密钥。如果没有密钥，数据将不可读取，而且实际上不可破译。美国联邦信息处理标准中有关高级加密标准的内容 (FIPS-197) 说明了受支持的算法。

Sybase IQ 增加了 AES_ENCRYPT 和 AES_DECRYPT 函数以及 LOAD TABLE ENCRYPTED 子句，以支持用户加密列。这些函数允许通过应用程序调用来显式加密和解密列数据。加密和解密的密钥管理由应用程序负责。

用户必须得到专门许可才能使用本产品文档中介绍的 Sybase IQ “高级安全性”选项的加密列功能。

某些数据库选项会影响列的加密。使用此功能前，请阅读第 20 页的“为列加密设置数据库选项”。

定义

介绍对存储数据的加密时会用到以下术语：

明文 以可读懂的原始形式存在的数据。明文并不仅限于字符串数据，而是用来描述任何以原始表示形式存在的数据。

密文 以难以读懂的形式存在，对明文形式的信息内容起保护作用的数据。

加密 将数据从明文变为密文的可逆转换。也称为加密。

解密 将密文变回明文的可逆转换。也称为解密。

密钥 用于对数据进行加密或解密的数字。对称密钥加密系统使用相同的密钥进行加密和解密。非对称密钥系统使用一个密钥进行加密，使用另一个（但在数学上是相关的）密钥进行解密。 Sybase IQ 界面接受使用字符串作为密钥。

Rijndael 发音为 “reign dahl”。支持各种密钥和块大小的特定加密算法。设计此算法的初衷在于使用简单的整字节操作，因此在软件中这种算法相对容易实现。

AES 即高级加密标准，是一种经过 FIPS 批准的加密算法，用于保护敏感 （但未分类）的电子数据。 AES 采用限制块大小和密钥长度的 Rijndael 算法。 AES 是 Sybase IQ 支持的算法。

适用于加密列的数据类型

本节列出了加密列支持和不支持的数据类型，并讨论了对加密列的原始数据类型的保护情况。

支持的数据类型

AES_ENCRYPT 函数的第一个参数必须为以下受支持的数据类型之一：

CHAR	NUMERIC
VARCHAR	FLOAT
TINYINT	REAL
SMALLINT	DOUBLE
INTEGER	DECIMAL
BIGINT	DATE
BIT	TIM
BINARY	DATETIME
VARBINARY	TIMESTAMP
UNSIGNED INT	SMALLDATETIME
UNSIGNED BIGINT	

LOB 数据类型目前不受 Sybase IQ 列加密支持。

保护数据类型

Sybase IQ 可确保在对数据进行解密时保留明文的原始数据类型，前提是将该数据类型作为参数提供给 AES_DECRYPT 函数，或者此函数位于 CAST 函数内。 Sybase IQ 会将 CAST 的目标数据类型与原始加密数据的数据类型进行比较。如果这两个数据类型不匹配，则会返回 -1001064 错误，其中包含有关原始和目标数据类型的详细信息。

例如，假定有一个经过加密的 VARCHAR(1) 值及以下有效解密语句：

```
SELECT AES_DECRYPT ( thecolumn, 'theKey',
VARCHAR(1) ) FROM thetable
```

如果尝试使用以下语句对数据进行解密：

```
SELECT AES_DECRYPT ( thecolumn, 'theKey',
SMALLINT ) FROM thetable
```

则返回的错误如下：

```
Decryption error: Incorrect CAST type smallint(5,0)
for decrypt data of type varchar(1,0).
```

这种数据类型检查仅在提供时执行。如果没有 CAST 或数据类型参数，查询将以二进制数据的形式返回密文。

注释 当对文字常量使用 AES_ENCRYPT 函数时 （如以下语句中所示）：

```
INSERT INTO t (cipherCol) VALUES (AES_ENCRYPT (1,
'key'))
```

请注意，数据类型 1 是不明确的。数据类型 1 可以是 TINYINT、SMALLINT、 INTEGER、 UNSIGNED INT、 BIGINT 或 UNSIGNED BIGINT，也可能是其它数据类型。

Sybase 建议以显式方式使用 CAST 函数来消除任何潜在的不明确性，如以下语句中所示：

```
INSERT INTO t (cipherCol)
VALUES ( AES_ENCRYPT (CAST (1 AS UNSIGNED INTEGER),
'key'))
```

在加密数据时通过使用 CAST 函数以显式方式转换数据类型，可以防止在解密数据时出现与使用 CAST 函数有关的问题。

如果要加密的数据来自于列，或加密数据是使用 LOAD TABLE 插入的，则不存在不明确性问题。

AES_ENCRYPT 函数 [String]

函数 使用所提供的加密密钥对指定值进行加密，并返回 VARBINARY 或 LONG VARBINARY。

语法 **AES_ENCRYPT**(*string-expression*, *key*)

参数

string-expression 要加密的数据。有关所支持数据类型的列表，请参见第 4 页的 “适用于加密列的数据类型”。此外，也可以将二进制值传递给 AES_ENCRYPT。此参数区分大小写，即使在不区分大小写的数据库中也是如此。

key 用来对 *string-expression* 加密的加密密钥。若要获取原始值，还必须使用同一密钥对值进行解密。此参数区分大小写，即使在不区分大小写的数据库中也是如此。

与使用大多数口令一样，最好选择不易猜出的密钥值。Sybase 建议为密钥选择的值长度至少为 16 个字符，同时包含大写和小写字母，且包括数字和特殊字符。每次要对数据进行解密时，都需要使用此密钥。

警告！请保护好您的密钥；将密钥副本存储在安全位置。如果丢失了密钥，则加密数据将完全无法访问且无法恢复。

用法

AES_ENCRYPT 返回一个 VARBINARY 类型的值，其长度最多比输入的 *string-expression* 长 31 个字节。该函数返回的值为密文，是人无法读懂的。可以使用 AES_DECRYPT 函数为使用 AES_ENCRYPT 函数加密的*字符串表达式*解密。为了成功对 *string-expression* 解密，请使用对数据加密时所使用的加密密钥和算法。如果您指定了不正确的加密密钥，将会产生错误。

如果将加密值存储在表中，则列的数据类型应为 VARBINARY 或 VARCHAR 且应大于或等于 32 字节，以便不对数据执行字符集转换。（字符集转换会阻止数据解密的进行。）如果 VARBINARY 或 VARCHAR 列的长度小于 32 字节，则 AES_DECRYPT 函数返回错误。

AES_ENCRYPT 函数的结果数据类型可能为 LONG VARBINARY。如果在 SELECT INTO 语句中使用 AES_ENCRYPT，您必须具有非结构化数据分析选件许可证，或者使用 CAST 并将 AES_ENCRYPT 设置为正确的数据类型和大小。

有关其它信息，请参见《参考：构件块、表和过程》的第 4 章 “SQL 函数”中的 “REPLACE 函数 String]”。

标准和兼容性

- **SQL** ISO/ANSI SQL 语法的供应商扩展
- **Sybase** 不受 Adaptive Server Enterprise 支持

另请参见

第 7 页的 “AES_DECRYPT 函数 [String]”

第 8 页的 “LOAD TABLE ENCRYPTED 子句”

示例

有关使用 AES_ENCRYPT 函数的示例，请参见第 10 页的 “加密和解密示例”。

AES_DECRYPT 函数 [String]

函数

使用所提供的密钥对字符串进行解密，并返回 VARBINARY 或 LONG VARBINARY （在缺省条件下），或者返回原始明文类型。

语法

AES_DECRYPT(*string-expression*, *key* [, *data-type*])

参数

string-expression 要解密的字符串。也可以将二进制值传递给此函数。此参数区分大小写，即使在不区分大小写的数据库中也是如此。

key 需要有加密密钥才能对 *string-expression* 解密。若要获取加密的原始值，密钥必须是用于加密 *string-expression* 的加密密钥。此参数区分大小写，即使在不区分大小写的数据库中也是如此。

警告！请保护好您的密钥；将密钥副本存储在安全位置。如果丢失了密钥，则加密数据将完全无法访问且无法恢复。

---

data-type 该可选参数指定解密的 *string-expression* 的数据类型，其值必须为原始明文的数据类型。

如果在使用 AES_ENCRYPT 函数插入数据时没有使用 CAST 语句，则可以通过将 VARCHAR 作为 *data-type* 传递来使用 AES_DECRYPT 函数查看相同数据。如果没有将 *data-type* 传递给 AES_DECRYPT，则返回 VARBINARY 数据类型。

用法

可以使用 AES_DECRYPT 函数为使用 AES_ENCRYPT 函数加密的 *字符串表达式* 解密。如果不指定数据类型，此函数返回与输入字符串具有相同字节数的 VARBINARY 或 LONG VARBINARY 值。否则将返回指定的数据类型。

为了成功对 *string-expression* 进行解密，必须使用用于加密数据的加密密钥。使用不正确的加密密钥将导致返回错误。

标准和兼容性

- **SQL** ISO/ANSI SQL 语法的供应商扩展
- **Sybase** 不受 Adaptive Server Enterprise 支持

另请参见

- 第 5 页的 “AES_ENCRYPT 函数 [String]”
- 第 10 页的 “加密和解密示例”
- 第 8 页的 “LOAD TABLE ENCRYPTED 子句”

示例

以下示例对 user_info 表中的用户口令进行解密：

```
SELECT AES_DECRYPT(user_pwd, '8U3dkA', CHAR(100))
FROM user_info;
```

LOAD TABLE ENCRYPTED 子句

LOAD TABLE 语句支持 column-spec 关键字 ENCRYPTED。 *column-specs* 必须按以下顺序跟在 LOAD TABLE 语句中列名的后面：

- *format-specs*
- *null-specs*
- *encrypted-specs*

请参见第 9 页的 “示例”。

有关完整语法，请参见 《参考：语句和选项》的第 1 章 “SQL 语句” 中的 “LOAD TABLE 语句”

语法

| **ENCRYPTED**(*data-type* ‘*key-string*’ [, ‘*algorithm-string*’])

参数

data-type 输入文件字段应转换到的目标数据类型，作为 AES_ENCRYPT 函数的输入。有关支持的数据类型，请参见第 4 页的 “适用于加密列的数据类型”。 *data-type* 的数据类型应当与 AES_DECRYPT 函数的输出的数据类型相同。请参见第 7 页的 “AES_DECRYPT 函数 [String]”。

key-string 用于对数据进行加密的加密密钥。此密钥必须为字符串文字。若要获取原始值，必须使用同一密钥对值进行解密。此参数区分大小写，即使在不区分大小写的数据库中也是如此。

与使用大多数口令一样，最好选择不容易忘记的密钥值。 Sybase 建议为密钥选择的值长度至少为 16 个字符，同时包含大写和小写字母，且包括数字和特殊字符。每次要对数据进行解密时，都需要使用此密钥。

警告！ 请保护好您的密钥；将密钥副本存储在安全位置。如果丢失了密钥，将导致完全无法访问加密数据，这是无法进行恢复的。

---

algorithm-string 用于对数据进行加密的算法。该参数为可选的，但数据的加密和解密算法必须相同。当前 AES 是缺省值，因为它是唯一受支持的算法。 AES 是一种数据块加密算法，美国国家标准与技术协会 (NIST) 选择它作为新的数据块密码高级加密标准 (AES)。

用法

使用 ENCRYPTED 列规范可以指定加密密钥，也可以选择指定对载入列中的数据进行加密时所使用的算法。这种装载操作的目标列应为 VARBINARY 类型。指定其它数据类型将返回错误。

另请参见

- 第 5 页的 “AES_ENCRYPT 函数 [String]”
- 第 7 页的 “AES_DECRYPT 函数 [String]”
- 第 10 页的 “加密和解密示例”

示例

```
LOAD TABLE table_name
(
plaintext_column_name,
a_ciphertext_column_name
NULL('nil')
ENCRYPTED(varchar(6),'tHefiRstkEy') ,
another_encrypted_column
ENCRYPTED(bigint,'thEseconDkeY','AES')
)
FROM '/path/to/the/input/file'
FORMAT ascii
DELIMITED BY ';'
ROW DELIMITED BY '\0xa'
QUOTES OFF
ESCAPES OFF
```

其中 LOAD TABLE 语句的输入文件的格式为：

```
a;b;c;
d;e;f;
g;h;i;
```

处理加密列

本节介绍如何处理加密列，并提供了一些示例。

对加密文本进行字符串比较

如果数据不区分大小写，或者使用 ISO_BINENG 以外的归类，则必须对密文列进行解密，以便执行字符串比较。

当对字符串执行比较时，对于诸多归类而言，等同字符串和相同字符串之间的区别非常重要，并且这取决于 CREATE DATABASE 的 CASE 选项。在设置为 CASE RESPECT 且采用 ISO_BINENG 归类的数据库中，将以相同方式解决 Sybase IQ 的缺省值、等同性和相同性问题。

相同字符串始终是等同的，但等同字符串有可能不相同。仅当字符串使用相同字节值表示时，它们才是等同的。当数据不区分大小写或使用必须将多个字符视为等同的归类时，等同性和相同性之间的区别则非常重要。 ISO1LATIN1 就是一个这样的归类。

例如，字符串 “ABC”和 “abc”在不区分大小写的数据库中不相同，但二者等同。在区分大小写的数据库中，上述字符串既不相同也不等同。

Sybase 加密函数创建的密文保留相同性，但不保留等同性。换句话说，“ABC”和 “abc”密文永远不会等同。

若要在归类或 CASE 设置不允许等同性比较的情况下对密文执行等同性比较，应用程序必须将该列中的值修改为某种规范形式，即任何相等值都是等同值。例如，如果数据库是使用 CASE IGNORE 和 ISO_BINENG 归类创建的，且应用程序在将所有输入值放入列中之前向所有输入值应用了 UCASE，则所有等同值都是相同的。

加密和解密示例

示例 1

下面这个有关 AES_ENCRYPT 和 AES_DECRYPT 函数的示例是用带注释的 SQL 编写的。

```
--   This example of aes_encrypt and aes_decrypt function use is presented
in three parts:
--
-- Part I: Preliminary description of target tables and users as DDL
-- Part II: Example schema changes motivated by introduction of encryption
-- Part III: Use of views and stored procedures to protect encryption keys
--


--   Part I: Define target tables and users

--   Assume two classes of user, represented here by the instances
--   PrivUser and NonPrivUser, assigned to groups reflecting differing
--   privileges.

--   The initial state reflects the schema prior to the introduction
--   of encryption.

--   Set up the starting context: There are two tables with a common key.
--   Some columns contain sensitive data, the remaining columns do not.
--   The usual join column for these tables is sensitiveA.
--   There is a key and a unique index.

   grant connect to PrivUser identified by 'verytrusted' ;
   grant connect to NonPrivUser identified by 'lesstrusted' ;

   grant connect to high_privileges_group ;
   grant group to high_privileges_group ;
   grant membership in group high_privileges_group to PrivUser ;

   grant connect to low_privileges_group ;
   grant group to low_privileges_group ;
```

```
    grant membership in group low_privileges_group to NonPrivUser ;

    create table DBA.first_table
                 (sensitiveA char(16) primary key
                 ,sensitiveB numeric(10,0)
                 ,publicC    varchar(255)
                 ,publicD    date
                 )  ;


--   There is an implicit unique HG (HighGroup) index enforcing the primary
key.

    create table second_table
                 (sensitiveA char(16)
                 ,publicP integer
                 ,publicQ tinyint
                 ,publicR varchar(64)
                 )  ;

    create hg index second_A_HG on second_table ( sensitiveA ) ;

--    TRUSTED users can see the sensitive columns.

    grant select ( sensitiveA, sensitiveB, publicC, publicD )
       on DBA.first_table to PrivUser ;
    grant select ( sensitiveA, publicP, publicQ, publicR )
       on DBA.second_table to PrivUser ;

--    Non-TRUSTED users in existing schema need to see sensitiveA to be
--    able to do joins, even though they should not see sensitiveB.

    grant select ( sensitiveA, publicC, publicD )
       on DBA.first_table to NonPrivUser ;
    grant select ( sensitiveA, publicP, publicQ, publicR )
       on DBA.second_table to NonPrivUser ;

--    Non-TRUSTED users can execute queries such as

    select I.publicC, 3*II.publicQ+1
    from DBA.first_table I, DBA.second_table II
    where I.sensitiveA = II.sensitiveA and I.publicD IN ( '2006-01-11' ) ;

--    and

    select count(*)
```

```
    from DBA.first_table I, DBA.second_table II
    where I.sensitiveA = II.sensitiveA and SUBSTR(I.sensitiveA,4,3)
    BETWEEN '345' AND '456' ;

--   But only TRUSTED users can execute the query

    select I.sensitiveB, 3*II.publicQ+1
    from DBA.first_table I, DBA.second_table II
    where I.sensitiveA = II.sensitiveA and I.publicD IN ( '2006-01-11' ) ;



--   Part II: Change the schema in preparation for encryption
--
--    The DBA introduces encryption as follows:
--
--    For applicable tables, the DBA changes the schema, adjusts access
--    permissions, and updates existing data. The encryption
--    keys used are hidden in a subsequent step.


--   DataLength comparison for length of varbinary encryption result
--   (units are Bytes):
--
--    PlainText CipherText    Corresponding Numeric Precisions
--
--            0         16
--      1 -  16         32      numeric(1,0)   - numeric(20,0)
--     17 -  32         48      numeric(21,0)  - numeric(52,0)
--     33 -  48         64      numeric(53,0)  - numeric(84,0)
--     49 -  64         80      numeric(85,0)  - numeric(116,0)
--     65 -  80         96      numeric(117,0) - numeric(128,0)
--     81 -  96        112
--     97 - 112        128
--    113 - 128        144
--    129 - 144        160
--    145 - 160        176
--    161 - 176        192
--    177 - 192        208
--    193 - 208        224
--    209 - 224        240

--   The integer data types tinyint, small int, integer, and bigint
--   are varbinary(32) ciphertext.

--   The exact relationship is
```

```
--   DATALENGTH(ciphertext) =
--   (((DATALENGTH(plaintext)+ 15) / 16) + 1) * 16

--  For the first table, the DBA chooses to preserve both the plaintext and
--   ciphertext forms. This is not typical and should only be done if the
--   database files are also encrypted.

--   Take away NonPrivUser's access to column sensitiveA and transfer
--   access to the ciphertext version.

--   Put a unique index on the ciphertext column. The ciphertext
--   itself is indexed.

--   NonPrivUser can select the ciphertext and use it.

--   PrivUser can still select either form (without paying decrypt costs).

   revoke select ( sensitiveA ) on DBA.first_table from NonPrivUser ;
   alter table DBA.first_table add encryptedA varbinary(32) ;
   grant select ( encryptedA ) on DBA.first_table to PrivUser ;
   grant select ( encryptedA ) on DBA.first_table to NonPrivUser ;
   create unique hg index first_A_unique on first_table ( encryptedA ) ;
   update DBA.first_table
      set encryptedA = aes_encrypt(sensitiveA, 'seCr3t')
      where encryptedA is null ;
   commit

--   Now change column sensitiveB.

   alter table DBA.first_table add encryptedB varbinary(32) ;
   grant select ( encryptedB ) on DBA.first_table to PrivUser ;
   create unique hg index first_B_unique on first_table ( encryptedB ) ;
   update DBA.first_table
      set encryptedB = aes_encrypt(sensitiveB,
      'givethiskeytonoone') where encryptedB is null ;
   commit

--   For the second table, the DBA chooses to keep only the ciphertext.
--   This is more typical and encrypting the database files is not required.

   revoke select ( sensitiveA ) on DBA.second_table from NonPrivUser ;
   revoke select ( sensitiveA ) on DBA.second_table from PrivUser ;
   alter table DBA.second_table add encryptedA varbinary(32) ;
   grant select ( encryptedA ) on DBA.second_table to PrivUser ;
   grant select ( encryptedA ) on DBA.second_table to NonPrivUser ;
   create unique hg index second_A_unique on second_table ( encryptedA ) ;
```

```
    update DBA.second_table
       set encryptedA = aes_encrypt(sensitiveA, 'seCr3t')
       where encryptedA is null ;
    commit
    alter table DBA.second_table drop sensitiveA ;




--   The following types of queries are permitted at this point, before
--   changes are made for key protection:

--   Non-TRUSTED users can equi-join on ciphertext; they can also select
--   the binary, but have no way to interpret it.

    select I.publicC, 3*II.publicQ+1
    from DBA.first_table I, DBA.second_table II
    where I.encryptedA = II.encryptedA and I.publicD IN ( '2006-01-11' ) ;

--   Ciphertext-only access rules out general predicates and expressions.
--   The following query does not return meaningful results.
--
--   NOTE: These four predicates can be used on the varbinary containing
--   ciphertext:
--  = (equality)
--  <> (inequality)
--  IS NULL
--  IS NOT NULL

    select count(*)
    from DBA.first_table I, DBA.second_table II
    where I.encryptedA = II.encryptedA and SUBSTR(I.encryptedA,4,3)
       BETWEEN '345' AND '456' ;

--   The TRUSTED user still has access to the plaintext columns that
--   were retained. Therefore, this user does not need to call
--   aes_decrypt and does not need the key.

    select count(*)
    from DBA.first_table I, DBA.second_table II
    where I.encryptedA = II.encryptedA and SUBSTR(I.sensitiveA,4,3)
       BETWEEN '345' AND '456' ;
```

```
--   Part III: Protect the encryption keys

--   This section illustrates how to grant access to the plaintext, but
--   still protect the keys.

--   For the first table, the DBA elected to retain the plaintext columns.
--   Therefore, the following view has the same capabilities as the trusted
--   user above.
--   Assume group_member is being used for additional access control.

--   NOTE: In this example, NonPrivUser still has access to the ciphertext
--   encrypted in the base table.

   create view DBA.a_first_view (sensitiveA, publicC, publicD)
      as
         select
           IF group_member('high_privileges_group',user_name()) = 1
             THEN sensitiveA
             ELSE NULL
           ENDIF,
           publicC,
           publicD
         from first_table ;

   grant select on DBA.a_first_view to PrivUser ;
   grant select on DBA.a_first_view to NonPrivUser ;

--   For the second table, the DBA did not keep the plaintext.
--   Therefore, aes_decrypt calls must be used in the view.
--   IMPORTANT: Hide the view definition with ALTER VIEW, so that no one
--   can discover the key.

   create view DBA.a_second_view (sensitiveA,publicP,publicQ,publicR)
      as
         select
           IF group_member('high_privileges_group',user_name()) = 1
             THEN aes_decrypt(encryptedA,'seCr3t', char(16))
             ELSE NULL
           ENDIF,
           publicP,
           publicQ,
           publicR
         from second_table ;
```

```
    alter view DBA.a_second_view set hidden ;
    grant select on DBA.a_second_view to PrivUser ;
    grant select on DBA.a_second_view to NonPrivUser ;

--   Likewise, the key used for loading can be protected in a stored
     procedure.
--   By hiding the procedure (just as the view is hidden), no one can see
--   the keys.

    create procedure load_first_proc(@inputFileName varchar(255),
                                     @colDelim varchar(4) default '$',
                                     @rowDelim varchar(4) default '\n')
       begin
          execute immediate with quotes
             'load table DBA.second_table
             (encryptedA encrypted(char(16),' ||
             '''' || 'seCr3t' || '''' || '),publicP,publicQ,publicR) ' ||
             ' from ' || '''' || @inputFileName || '''' ||
             ' delimited by ' || '''' || @colDelim || '''' ||
             ' row delimited by ' || '''' || @rowDelim || '''' ||
             ' quotes off escapes off' ;
       end
    ;

    alter procedure DBA.load_first_proc set hidden ;

--   Call the load procedure using the following syntax:

    call load_first_proc('/dev/null', '$', '\n') ;

--   Below is a comparison of several techniques for protecting the
--   encryption keys by using user-defined functions (UDFs), other views,
--   or both. The first and the last alternatives offer maximum performance.

--   The second_table is secured as defined earlier.
```

```
--   Alternative 1:
--   This baseline approach relies on restricting access to the entire view.

     create view
          DBA.second_baseline_view(sensitiveA,publicP,publicQ,publicR)
       as
          select
            IF group_member('high_privileges_group',user_name()) = 1
              THEN aes_decrypt(encryptedA,'seCr3t', char(16))
              ELSE NULL
            ENDIF,
            publicP,
            publicQ,
            publicR
          from DBA.second_table ;

     alter view DBA.second_baseline_view set hidden ;
     grant select on DBA.second_baseline_view to NonPrivUser ;
     grant select on DBA.second_baseline_view to PrivUser ;


--   Alternative 2:
--   Place the encryption function invocation within a user-defined
--    function (UDF).
-- Hide the definition of the UDF. Restrict the UDF permissions.
-- Use the UDF in a view that handles the remainder of the security
--     and business logic.
-- Note: The view itself does not need to be hidden.

     create function DBA.second_decrypt_function(IN datum varbinary(32))
          RETURNS char(16) DETERMINISTIC
          BEGIN
             RETURN aes_decrypt(datum,'seCr3t', char(16));
          END  ;

     grant execute on DBA.second_decrypt_function to PrivUser ;
     alter function DBA.second_decrypt_function set hidden ;
```

```
     create view
          DBA.second_decrypt_view(sensitiveA,publicP,publicQ,publicR)
       as
              select
                IF group_member('high_privileges_group',user_name()) = 1
                  THEN second_decrypt_function(encryptedA)
                  ELSE NULL
                ENDIF,
                publicP,
                publicQ,
                publicR
              from DBA.second_table ;

     grant select on DBA.second_decrypt_view to NonPrivUser ;
     grant select on DBA.second_decrypt_view to PrivUser ;

--  Alternative 3:
--  Sequester only the key selection in a user-defined function.
--  This function could be extended to support selection of any
--   number of keys.
--  This UDF is also hidden and has restricted execute privileges.
--  Note: Any view that uses this UDF therefore does not compromise
-- the key values.

     create function DBA.second_key_function()
            RETURNS varchar(32) DETERMINISTIC
            BEGIN
             return 'seCr3t' ;
            END

     grant execute on DBA.second_key_function to PrivUser ;
     alter function DBA.second_key_function set hidden ;
```

```
    create view DBA.second_key_view(sensitiveA,publicP,publicQ,publicR)
            as
               select
                 IF group_member('high_privileges_group',user_name()) = 1
                      THEN aes_decrypt(encryptedA,second_key_function(),
                      char(16))
                    ELSE NULL
                 ENDIF,
                 publicP,
                 publicQ,
                 publicR
               from DBA.second_table ;

    grant select on DBA.second_key_view to NonPrivUser ;
    grant select on DBA.second_key_view to PrivUser ;

--  Alternative 4:
--  The recommended alternative is to separate the security logic
--  from the business logic by dividing the concerns into two views.
--  Only the security logic view needs to be hidden.
--  Note: The performance of this approach is similar to that of the first
-- alternative.

    create view
      DBA.second_SecurityLogic_view(sensitiveA,publicP,publicQ,publicR)
          as
             select
               IF group_member('high_privileges_group',user_name()) = 1
                 THEN aes_decrypt(encryptedA,'seCr3t', char(16))
                 ELSE NULL
               ENDIF,
               publicP,
               publicQ,
               publicR
             from DBA.second_table ;

    alter view DBA.second_SecurityLogic_view set hidden ;
```

```
        create view
          DBA.second_BusinessLogic_view(sensitiveA,publicP,publicQ,publicR)
              as
                  select
                    sensitiveA,
                    publicP,
                    publicQ,
                    publicR
                  from DBA.second_SecurityLogic_view ;

        grant select on DBA.second_BusinessLogic_view to NonPrivUser ;
        grant select on DBA.second_BusinessLogic_view to PrivUser ;

    -- End of encryption example
```

示例 2

如果给定相同的输入值和密钥，但采用两种不同的数据类型，则 AES_ENCRYPT 生成的密文不同。因此，两个分别保存两种不同数据类型加密值的密文列连接后，可能不会返回相同的结果。

例如，假定：

```
CREATE TABLE tablea(c1 int, c2 smallint);
INSERT INTO tablea VALUES (100,100);
```

值 `AES_ENCRYPT(c1, 'key')` 与 `AES_ENCRYPT(c2,'key')` 不同，并且值 `AES_ENCRYPT(c1,'key')` 也与 `AES_ENCRYPT(100,'key')` 不同。

若要解决此问题，请将 AES_ENCRYPT 的输入强制转换为同一数据类型。例如，以下代码段的结果是相同的：

```
AES_ENCRYPT(c1, 'key');
AES_ENCRYPT(CAST(c2 AS INT), 'key');
AES_ENCRYPT(CAST(100 AS INT), 'key');
```

为列加密设置数据库选项

某些 Sybase IQ 数据库选项设置会影响列的加密和解密。请在使用 AES_ENCRYPT 或 AES_DECRYPT 之前检查本节中提及的选项，因为缺省设置对于大多数列加密操作而言并非最佳设置。

防止密文数据意外截断

若要防止加密函数的密文输出出现意外截断 （或防止任何其它字符或二进制字符串出现意外截断），请按如下所示设置数据库选项：

```
SET OPTION STRING_RTRUNCATION = 'ON'
```

当 STRING_RTRUNCATION 设置为 ON （缺省值）时，只要在装载、插入、更新或 SELECT INTO 操作期间字符串将被截断，引擎就会引发错误。这是 ISO/ANSI SQL 行为，也是推荐做法。

当需要执行显式截断时，请使用诸如 LEFT、 SUBSTRING 或 CAST 之类的字符串表达式。

将 STRING_RTRUNCATION 设置为 OFF 会对字符串强制执行无提示截断。

AES_DECRYPT 函数也会检查输入密文的数据长度是否有效，并检验文本输出以确认所得数据的长度以及所提供密钥的正确性。（如果提供了数据类型参数，则还会检查数据类型。）

保护密文的完整性

若要保护密文的完整性，请设置以下数据库选项：

```
SET OPTION ASE_BINARY_DISPLAY = 'OFF'
```

当 ASE_BINARY_DISPLAY 设置为 OFF （缺省值）时，系统将不修改二进制数据，保持其原始二进制形式不变。

当 ASE_BINARY_DISPLAY 设置为 ON 时，系统会将二进制数据转换为其十六进制字符串显示表示形式。只有在需要向最终用户显示数据或者需要将数据导出至另一个外部系统，且在数据传送过程中原始二进制数据可能会遭到更改的情况下，才可临时将该选项设置为 ON。

防止误用密文

CONVERSION_MODE 数据库选项限制在各种操作过程中在二进制数据类型 （BINARY、 VARBINARY 和 LONG BINARY）与其它非二进制数据类型（BIT、 TINYINT、 SMALLINT、 INT、 UNSIGNED INT、 BIGINT、 UNSIGNED BIGINT、 CHAR、 VARCHAR 和 LONG VARCHAR）之间进行隐式转换。使用 CONVERSION_MODE 可防止对加密数据进行隐式数据类型转换 （这种转换可导致在语义上无意义的操作）：

```
SET TEMPORARY OPTION CONVERSION_MODE = 1
```

将 CONVERSION_MODE 设置为 1，可防止在执行 INSERT 和 UPDATE 命令以及查询时将二进制数据类型隐式转换为任何其它非二进制数据类型。这种限制二进制转换模式还适用于 LOAD TABLE 缺省值和 CHECK 约束。

CONVERSION_MODE 选项采用缺省值 0 时，可保留 Sybase IQ 12.7 版之前的二进制数据类型隐式转换行为。

请参见 《参考：语句和选项》的第 2 章 “数据库选项”中的“CONVERSION_MODE 选项”。