IMPLEMENTATION NOTES ON *bdes*(1)

Matt Bishop

Technical Report PCS-TR91-158

Implementation Notes on *bdes*(1)

Matt Bishop

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

ABSTRACT

This note describes the implementation of *bdes*, the file encryption program being distributed in the 4.4 release of the Berkeley Software Distribution. It implements all modes of the Data Encryption Standard program.

1. Introduction

The Data Encryption Standard is a standard endorsed by the federal government. It is considerably stronger than the algorithm used by the UNIX™ *crypt*(1) program, and therefore is a more suitable candidate for protecting information, especially information contained in ASCII files. The program *bdes*(1) implements the DES and all of its modes, including the two authentication modes.

Because others may wish to write software compatible with this program, this note presents the layout of the encrypted files produced by *bdes* as well as internal details relevant to the implementation. Whereever possible and appropriate, the description of the *des*(1) program given in [4] has been followed; thus, *bdes* is completely compatible with that program. However, *bdes* also offers several extensions to *des* that are not compatible, and these will be explicitly pointed out.

In this note, strings typed as shown will be in `Courier Roman font`, and strings to be chosen by the user will be in *`Courier Oblique font`*. The space character (ASCII <SP>, octal 40, decimal 32, hex 20) will be represented as "␣" and the newline character (ASCII <NL>, octal 12, decimal 10, hex a) as "↵". Because it is often more convenient to represent arbitrary characters as a sequence of hexadecimal digits, that representation will often be used; these digits will be in **`Courier Bold font`** with spaces often inserted for readability.

2. Overview and Use

Bdes is an implementation of the Data Encryption Standard. It implements the Data Encryption Standard algorithm in software, and enables the user to encrypt data using any of the four

---

This work is based on work funded by grant NAG2-680 from the National Aeronautics and Space Administration to Dartmouth College.
UNIX is a Registered Trademark of AT&T Bell Laboratories.

modes of operation of the DES (Electronic Code Book, Cipher Block Chaining, *k*-bit Cipher Feed Back, and *k*-bit Output Feed Back) as well as the Alternate *k*-bit Cipher Feed Back mode. Further, *bdes* supports message authentication code generation based on both the Cipher Block Chaining mode and the *k*-bit Cipher Feed Back mode.

By default, *bdes* encrypts an input file using Cipher Block Chaining mode, and is invoked as a filter. The key may be specified either on the command line or may be typed to the prompt. So, if the input file `inputfile` contains the message

```
a␣test␣message␤
```

then the following command encrypts it using the key `abcdefgh`:

```
bdes abcdefgh < inputfile > outputfile
```

Now `outputfile` contains

```
16 0e eb af 68 a0 d0 19 f1 a2 9b 31 8a 0d 01 c3
```

Other modes are specified using command-line options, as is control of the way the key is interpreted. The next sections contain several examples, and the Appendix has the manual page.

3. Keys and Parity

The key consists of 64 bits, and may be presented in any of hex, binary, or as a string of ASCII characters. If the key is given in hex or binary, it is used as is with no changes. However, if the key is given in ASCII, a delicate problem arises: by convention, the parity bit is usually set to 0. This high-order bit is generally ignored by applications; but the DES does not do so. Instead, it discards the low-order bit, effectively reducing the size of the space of possible keys from 2^{56} to 2^{48}.

To preserve the size of the key space, the value of the parity bit must be related to the value in the low-order bit, so the program sets the high-order bit to make each character in the key be of odd parity. (Note that the initial value of the parity bit is *not* used in this computation.) For example, if the key is `abcdefgh`, the actual key bits used are determined as follows:

ASCII key	a	b	c	d	e	f	g	h
ASCII key bits (hex)	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**
parity	odd	odd	even	odd	even	even	odd	odd
key bits used (hex)	**61**	**62**	**e3**	**64**	**e5**	**e6**	**67**	**68**

This convention (as opposed to requiring even parity, or simply copying the low-order bit to the high-order bit) was chosen to provide compatibility with the encryption program *des* distrib-

uted by Sun Microsystems, Inc. [4]. Whether the key is entered on the command line or on the keyboard, by default it is processed into the same key schedule generated by Sun's *des*, so files encrypted on a Sun can be decrypted using *bdes* (and vice versa).

If the user does not wish to use the Sun convention, the option **–p** will disable the parity bit changing; with it, the parity bit is that of the character typed. This is useful when the key is a known ASCII string and the file was encrypted on a system which does not alter parity bits.

A key may be represented as a bit vector, rather than an ASCII string, in one of two ways. It may be represented as a string of up to 16 hexadecimal digits; if fewer than 16 are given, the key is right filled with 0 bits. Or, it may be represented as a string of up to 64 binary digits, and again if fewer than 64 are given, the key is right-filled with 0 bits. Bit vector keys must be given on the command line, and must begin with the characters `0x` or `0X` (for hexadecimal) or `0b` or `0B` (for binary). For example, all of the following strings generate the same key schedule:

ASCII key	`abcdefgh`
hexadecimal key	`0x6162e364e5e66768`
binary key	`0b01100001011000101110001101101000111001011110000-110011001110110100 0`

Note that giving the key on the command line as `0x6162636465666768` will *not* reset the parity bits, because it is interpreted as a sequence of hex digits, not ASCII characters. The difference in interpretation is that here the user can specify all bits of the key exactly, whereas (on most terminals) it is not possible to control how the parity bit of ASCII characters is set. On some systems, it is possible to use a "Meta" key to set the parity bit for an ASCII character; should this be the case and the user desire *bdes* not to reset the parity bit, the option **–p** will force the parity bit to be used as typed.

4. Encryption Output Representation

All modes of the DES output ciphertext in blocks; the size of the block is 64 bits (8 bytes) for ECB and CBC modes, and *k* bits for the *k*-bit CFB and OFB modes, and there are as many output blocks as input blocks. However, as the length of the input is usually not a multiple of the block size, some padding is necessary; but as padding must be done by appending characters, these characters must be distinguished from the input characters somehow. The mechanism used is that the last character of the (decrypted) last block is the (integer) number of characters from the input in the last block.

For example, suppose *inputfile* contains "This␣is␣a␣test␣", and it is encrypted in CBC mode using the key "abcdef#@" and the initialization vector 0x0; the command is

```
bdes abcdef#@ < inputfile > outputfile
```

as CBC is the default encryption mode and 0x0 the default initialization vector:

text	T	h	i	s	␣	i	s	␣	a	␣	t	e	s	t	␣	
hex	**54**	**68**	**69**	**73**	**20**	**69**	**73**	**20**	**61**	**20**	**74**	**65**	**73**	**74**	**0a**	
input	**54**	**68**	**69**	**73**	**20**	**69**	**73**	**20**	**61**	**20**	**74**	**65**	**73**	**74**	**0a**	**07**
output	**a5**	**5f**	**81**	**53**	**51**	**98**	**47**	**02**	**db**	**5a**	**c5**	**fe**	**50**	**3d**	**40**	**ce**

Notice that the text is 15 characters long, so there are 7 bytes following the last full block. *Bdes* pads this to a full block by appending one byte containing the ASCII character with numeric value 7 (the ASCII character <BEL>). The result is then encrypted.

As another example, suppose *inputfile* contains "test", and it is encrypted in ECB mode using the key "abcdef#@"; the command is

```
bdes -e abcdef#@ < inputfile > outputfile
```

because the option **–e** signifies ECB mode:

text	t	e	s	t				
hex	**74**	**65**	**73**	**74**				
input	**74**	**65**	**73**	**74**	**00**	**00**	**00**	**04**
output	**0d**	**8a**	**6e**	**57**	**9c**	**8f**	**27**	**5d**

Finally, if the length of the message is indeed a multiple of the block size, an extra block of all 0 bits is added. Suppose *inputfile* contains "test␣", and it is encrypted in 40-bit CFB mode using the key "abcdef#@" and the initialization vector 0x0123456789abcdef; the command is

```
bdes -f 40 -v 0x0123456789abcdef abcdef#@ < inputfile > outputfile
```

because the option **–f 40** signifies 40-bit CFB mode, and -v 0x01234566789abcdef sets the initialization vector:

text	t	e	s	t	␣					
hex	**74**	**65**	**73**	**74**	**0a**					
input	**74**	**65**	**73**	**74**	**0a**	**00**	**00**	**00**	**00**	**00**
output	**e2**	**c2**	**69**	**a4**	**5b**	**3c**	**3d**	**b3**	**f5**	**3c**

Note here the block size is 40 bits (5 bytes), not 64 bits (8 bytes).

This technique allows complete compatibility with Sun's *des* program. In Sun's implementation, padding is done with random bytes rather than bytes containing all zero bits. Cryptographically, this makes no difference, as the DES is a sufficiently good random cipher to obscure the input (see for example [2], Chapter 6), and known plaintext attacks are very difficult [1].

5. Differences Between the Standard CFB and OFB Modes and *bdes*

The UNIX operating system treats all files as streams of 8-bit bytes. In order to implement the CFB and OFB modes properly, it would be necessary to read k bits from the file, where k is an integer between 1 and 64 inclusive. However, this would require considerable buffering and be quite inefficient and prohibitively slow. For these reasons, the current implementation of *bdes* requires that k be a multiple of 8, so that an integral number of bytes will always be read from the file. Other than this change, this mode is implemented as described in [3].

A similar observation holds for the alternate CFB mode described in [3]. Here, only the low 7 bits of each byte are significant, and hence the parameter k is an integer from 1 to 56 inclusive; bdes requires k to be a multiple of 7. The high-order bit is retained for encryption and decryption, but output (whether from encryption or decryption) always has the high-order bit set to zero.

6. Message Authentication Code Modes

The Data Encryption Standard provides two modes of authentication, each providing between 1 and 64 bits of authentication data. In both cases an n-bit message authentication code (MAC) is generated, where $1 \leq n \leq 64$. The first is based on the CBC encryption mode, and the second on CFB mode. Both work the same.

First, the file is padded to a multiple of the block size by appending enough zero bits. It is then encrypted using the standard CBC (or CFB) algorithm, but all encrypted text is discarded except for the last block. The n leading bits of the last block are used as the MAC. Note that the block size constrains the number of bits available as the MAC.

The implementation allows the user to specify that the MAC is to be computed in either CBC or CFB mode, and the user can specify any number of bits from 1 to 64 inclusive. However, because the UNIX operating system can only output bits in multiples of 8, if the number of bits of MAC is not a multiple of 8, the MAC will be right-padded with the minimum number of zero bits necessary to make the MAC length be a multiple of 8. However, note that as the standard ([3], Ap-

pendix F) requires an incomplete final block be right-padded with zeroes, the technique of forcing the last octet to contain the number of bytes in the message is *not* used here.

For example, suppose *inputfile* contains "This␣is␣a␣test␣↵", and a 64-bit MAC is to be generated using CBC mode, the key "abcdef#@" and the initialization vector 0x0; the command is

```
bdes -m 64 abcdef#@ < inputfile > outputfile
```

as CBC is the default encryption mode and 0x0 the default initialization vector:

text	T	h	i	s	␣	i	s	␣	a	␣	t	e	s	t	↵	
hex	**54**	**68**	**69**	**73**	**20**	**69**	**73**	**20**	**61**	**20**	**74**	**65**	**73**	**74**	**0a**	
input	**54**	**68**	**69**	**73**	**20**	**69**	**73**	**20**	**61**	**20**	**74**	**65**	**73**	**74**	**0a**	**00**
output	**43**	**18**	**de**	**74**	**24**	**a9**	**65**	**d1**								

Notice that the text is 15 characters long, so there are 7 bytes following the last full block. *Bdes* pads this to a full block by appending a zero-filled byte. The result is then encrypted and the last block of output is used as the MAC.

As another example, suppose we used the same text, and wanted a 36-bit MAC to be generated using 40-bit CFB mode, the key "abcdef#@" and the initialization vector 0x0123456789abcdef; the command is

```
bdes -m 36 -f 40 -v 0x0123456789abcdef < inputfile > outputfile
```

where **–m 36** is the option to generate a 36-bit MAC, **–f 40** indicates 40-bit CFB is to be used, and **–v 0x123456789abcdef** sets the initialization vector. Note that, as the key is not given on the command line, the user will be prompted for it; only ASCII keys can be obtained in this way. It gives:

text	T	h	i	s	␣	i	s	␣	a	␣	t	e	s	t	↵
hex	**54**	**68**	**69**	**73**	**20**	**69**	**73**	**20**	**61**	**20**	**74**	**65**	**73**	**74**	**0a**
input	**54**	**68**	**69**	**73**	**20**	**69**	**73**	**20**	**61**	**20**	**74**	**65**	**73**	**74**	**0a**
output	**2b**	**18**	**68**	**2d**	**60**										

Note that the MAC is padded on the right by four zero bits to produce five characters that can be output.

7. Differences Between *bdes* and Sun's DES Implementation

The program *bdes* is designed to be completely compatible with Sun Microsystems, Inc.'s implementation of the Data Encryption Standard, called *des* and described in [4]. Thus, files en-

crypted using *des* can be decrypted using *bdes*, and vice versa, provided modes common to both are used. However, the user interfaces are completely different (and incompatible); as the manual page to *bdes* is in the appendix, these differences will not be elaborated upon further.

Sun's *des* supports the use of special-purpose hardware to encrypt and decrypt. Although *bdes* does not directly support the use of such hardware, it uses the library routine *encrypt*(3), which may. Hardware support was not included directly to support as large a number of platforms as possible with installers needing to know as little about the hardware as possible.

Sun's *des* supports only the CBC and ECB encryption modes; *bdes* supports all modes described in [3] (although CFB and OFB are not completely supported) as well as both CBC-based and CFB-based MACs.

Although input with length not a multiple of the block size is handled in the same way by both *des* and *bdes*, different values of the padding bytes are used in all but the last byte of the input. Where *bdes* puts zero bytes, *des* puts bytes containing random values. The reason for Sun's doing so is to prevent a known plaintext attack on the file should an attacker determine that the input's length were a multiple of the block size. With *bdes*, the plaintext contents of the last block of input for such a file is known (a block with all bits zero). With *des*, the plaintext contents of that block are not known. Cryptanalytically, given the information about the strength of the DES currently known, it is widely believed that known plaintext attacks are infeasible (see for example [1]) and so initializing and invoking the pseudorandom number generator seems unnecessary. But this means that ciphertexts produced from a plaintext by *bdes* and *des* will differ in the last block.

References

[1] D. Denning, "The Data Encryption Standard: Fifteen Years of Public Scrutiny," *Proceedings of the Sixth Annual Computer Security Applications Conference* pp. x–xv (Dec. 1990).

[2] A. Konheim, *Cryptography: A Primer*, John Wiley and Sons, Inc., New York, NY (1981).

[3] *DES Modes of Operation*, Federal Information Processing Standards Publication 81, National Bureau of Standards, U.S. Department of Commerce, Washington, DC (Dec. 1980).

[4] *UNIX User's Manual*, Sun Microsystems Inc., Mountain View, CA (Mar. 1988).

Appendix. The UNIX System Manual Page for *bdes*

NAME

bdes - encrypt/decrypt using the Data Encryption Standard

SYNOPSIS

bdes [*options*] [*key*]

DESCRIPTION

Bdes reads from the standard input and writes on the standard output. It implements all DES modes of operation described in FIPS PUB 81 including alternative cipher feedback mode and both authentication modes. All modes but the electronic code book mode require an initialization vector; if none is supplied, the zero vector is used. To protect the key and initialization vector from being read by *ps*(1), *bdes* hides its arguments on entry. If no *key* is given, one is requested from the controlling terminal if that can be opened, or from the standard input if not.

The key and initialization vector are taken as sequences of ASCII characters which are then mapped into their bit representations. If either begins with '0x' or '0X', that one is taken as a sequence of hexadecimal digits indicating the bit pattern; if either begins with '0b' or '0B', that one is taken as a sequence of binary digits indicating the bit pattern. In either case, only the leading 64 bits of the key or initialization vector are used, and if fewer than 64 bits are provided, enough 0 bits are appended to pad the key to 64 bits. Note that if the key is not entered on the command line, it is assumed to be ASCII. This is due to limits of the password reading function *getpass*(3), which allows at most 8 characters to be entered.

According to the DES standard, the low-order bit of each character in the key string is deleted. Since most ASCII representations set the high-order bit to 0, simply deleting the low-order bit effectively reduces the size of the key space from 2^{56} to 2^{48} keys. To prevent this, the high-order bit must be a function depending in part upon the low-order bit; so, the high-order bit is set to whatever value gives odd parity. This preserves the key space size. Note this resetting of the parity bit is *not* done if the key is given in binary or hex.

By default, the standard input is encrypted using cipher block chaining mode and is written to the standard output. Using the same key for encryption and decryption preserves plaintext, so

bdes *key* < plaintext | bdes –i *key*

is a very expensive equivalent of *cat*(1).

Options are:

–a The key and initialization vector strings are to be taken as ASCII suppressing the special interpretation given to leading '0x', '0X', '0b', and '0B' characters. Note this flag applies to *both* the key and initialization vector.

–c Use cipher block chaining mode. This is the default.

–e Use electronic code book mode.

–f *b* Use *b*-bit cipher feedback mode. Currently *b* must be a multiple of 8 between 8 and 64 inclusive (this does not conform to the standard CFB mode specification).

–F *b* Use *b*-bit alternative cipher feedback mode. Currently *b* must be a multiple of 7 between 7 and 56 inclusive (this does not conform to the alternative CFB mode specification).

–i invert (decrypt) the input.

–m *b* Compute a message authentication code (MAC) of *b* bits on the input. *b* must be between 1 and 64 inclusive; if *b* is not a multiple of 8, enough 0 bits will be added to pad the MAC length to the nearest multiple of 8. Only the MAC is output. MACs are only available in cipher block chaining mode or in cipher feedback mode.

–o *b* Use *b*-bit output feedback mode. Currently *b* must be a multiple of 8 between 8 and 64 inclusive (this does not conform to the OFB mode specification).

–p Disable the resetting of the parity bit. This flag forces the parity bit of the key to be used as typed, rather than making each character be of odd parity. It is used only if the key is given in ASCII.

–v *v* Set the initialization vector to *v*; the vector is interpreted in the same way as the key. The vector is ignored in electronic codebook mode.

The DES is considered a very strong cryptosystem, and other than table lookup attacks, key search attacks, and Hellman's time-memory tradeoff (all of which are very expensive and time-consuming), no cryptanalytic methods for breaking the DES are known in the open literature. No doubt the choice of keys and key security are the most vulnerable aspect of *bdes*.

IMPLEMENTATION NOTES

For implementors wishing to write software compatible with this program, the following notes are provided. This software is completely compatible with the implementation of the data encryption standard distributed by Sun Microsystems, Inc.

In the ECB and CBC modes, plaintext is encrypted in units of 64 bits (8 bytes, also called a block). To ensure that the plaintext file is encrypted correctly, *bdes* will (internally) append from 1 to 8 bytes, the last byte containing an integer stating how many bytes of that final block are from the plaintext file, and encrypt the resulting block. Hence, when decrypting, the last block may contain from 0 to 7 characters present in the plaintext file, and the last byte tells how many. Note that if during decryption the last byte of the file does not contain an integer between 0 and 7, either the file has been corrupted or an incorrect key has been given. A similar mechanism is used for the OFB and CFB modes, except that those simply require the length of the input to be a multiple of the mode size, and the final byte contains an integer between 0 and one less than the number of bytes being used as the mode. (This was another reason that the mode size must be a multiple of 8 for those modes.)

Unlike Sun's implementation, unused bytes of that last block are not filled with random data, but instead contain what was in those byte positions in the preceding block. This is quicker and more portable, and does not weaken the encryption significantly (and even then, only in a few cases).

If the key is entered in ASCII, the parity bits of the key characters are set so that each key character is of odd parity. Unlike Sun's implementation, it is possible to enter binary or

hexadecimal keys on the command line, and if this is done, the parity bits are *not* reset. This allows testing using arbitrary bit patterns as keys.

The Sun implementation always uses an initialization vector of 0 (that is, all zeroes). By default, *bdes* does too, but this may be changed from the command line.

FILES

/dev/tty controlling terminal for typed key

SEE ALSO

crypt(1), *crypt*(3)

Data Encryption Standard, Federal Information Processing Standard #46, National Bureau of Standards, U.S. Department of Commerce, Washington DC (Jan. 1977).

DES Modes of Operation, Federal Information Processing Standard #81, National Bureau of Standards, U.S. Department of Commerce, Washington DC (Dec. 1980).

Dorothy Denning, *Cryptography and Data Security*, Addison-Wesley Publishing Co., Reading, MA ©1982.

Matt Bishop, "Implementation Notes on *bdes*(1)", Technical Report PCS-TR-91-158, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH (Apr. 1991).

BUGS

There is a controversy raging over whether the DES will still be secure in a few years. The advent of special-purpose hardware could reduce the cost of any of the methods of attack named above so that they are no longer computationally infeasible.

As the key or key schedule is kept in memory throughout the run of this program, the encryption can be compromised if memory is readable.

There is no warranty of merchantability nor any warranty of fitness for a particular purpose nor any other warranty, either express or implied, as to the accuracy of the enclosed materials or as to their suitability for any particular purpose.

Accordingly, the user assumes full responsibility for their use. Further, the author assumes no obligation to furnish any assistance of any kind whatsoever, or to furnish any additional information or documentation.

AUTHOR

Matt Bishop, Department of Mathematics and Computer Science, Bradley Hall, Dartmouth College, Hanover, NH 03755

Electronic mail addresses:

Internet: Matt.Bishop@dartmouth.edu

UUCP: decvax!dartvax!Matt.Bishop