VectorLite Encryption

Version 5M

User's Manual

Warning

This software is provided for algorithmic proof of concept testing and demonstration use only.

This software should not to be used to protect files of value or need.

It is recommended to contain the software within an independent folder or directory separate from useful valuable files. Test files should be ***copied*** into a test folder – leaving original files in place. Plain-text test files are provided, in addition to a program to create test patterned files.

No Warranty or Guarantee is Expressed or Implied.

VectorLite Encryption has not been peer reviewed. Please test / demonstrate with caution. Version 5M testing has improved, but bugs may / likely exist.

Export Restrictions

The software and executable programs may be subject to United States export regulation. Please comply with all regulatory laws and governance.

Version 5M Document Revision History

July 28, 2021 Initial Release

Additions to this document will be made as time permits.

Author

Robert J. Miller

E-Mail: robtjmiller4249@gmail.com

bob@secretware.org

Notes

*This document is intended for all users, including the **patient** technically advanced*

Version 5M

Version 5M is a substantial upgrade, with emphasis upon:

1. Multi threaded performance for the encrypt and decrypt programs
2. Improved I/O performance, buffering input / output up to 100,000 bytes at a time
3. Elimination / simplification of command line options no longer relevant
4. A new ***random*** program to better handle C run-time library pseudo-random issues
5. Numerous clean-ups, bug fixes, and so forth.
6. The inclusion of two new analytical utility programs.
   a) ***file-stats*** Display mean, std dev, and co-var (cv) of a file's histogram of byte values
   b) ***check-proximity*** Display histogram of average distance between common byte values

Known Limitations

1. Plain-text file sizes are limited by the standard 32 bit C library I/O functions. The maximum plain-text input file size to ***encrypt*** is approximately 1.2 giga-bytes, and is actually limited by the larger output cipher-text file which must later be read as input by ***decrypt***. Cipher-text files are typically 60 % larger than the plain-text input file size. A later release may implement the 64 bit IO to remove this limit.
2. Several counters and index variables may also be 32 bit limited at this time.

Known Risks

1. ***file-stats*** displays a file's byte value statistics based on the histogram, not the individual bytes. Determination if the two computations result in the same or different results will be performed soon. The intended purposed of ***file-stats*** is to determine input file suitability to ***random*** – that is all.
2. Software generated random numbers will always be an issue – the new ***random*** program lessons the C language run-time library's ***srand*** and ***rand*** functions predictability. This is by introducing an additional unknown (***random's*** input file), and multiple human provided random seed values spread across a wide range of accepted values (in place of platform clock time).

Version 5L *(October 2020)*

For testing and development:

1	encrypt	A key-table usage trace-file output option with command line specified filename
2	encrypt	A random number usage trace-file output option, also with cmd line filename
3	key-trace	A program to analyze the results of the encrypt program's key usage trace file
4	key-line-col	A program to display a key-table page's line and row values

For Better Encryption:

1. The addition of an "Alpha Bias Array" of 1024 random values to pre-modify plain-text
2. The addition of a "Displacement Bias Array" of 1024 random values to post modify traverse results
3. The addition of a "Re-Vector Bias Array" of 1024 random values to post modify re-vector key x, y.

These three new bias arrays are initially populated and included within the ***buildkey*** key-table output files. The arrays are subsequently modified during run-time as the encryption / decryption takes place, never using the same entry twice. The arrays are updated at random (non repeating) points near buffer exhaustion, each differently. The encrypt and decrypt programs track and update the arrays in the same manner via a method described later.

Utility Programs:

1	key-summary	Quickly display 1st key-table page's alpha and vector table's properties
2	key-details	More detail than key-summary
3	key-dump	Even more detail than key-details – each value dumped out
4	key-line-col	Display an individual line and column key-table page's data values
5	key-Trace	Histogram and detailed info on key-table traversing history and usage
6	compare-files	Compare two files, providing info on 1st point of variance, if diff
7	check-pattern-bytes-1	Counts repetitive patterns of bytes within a file, inclusive counts
8	check-pattern-bytes-2	Counts repetitive patterns of bytes within a file, exclusive counts
9	histogram-bytes	Output a histogram of byte values within a file
10	create-file	Create a file of repetitive binary values or character strings

The utility programs have not been peer reviewed and validated. The programs are the author's best attempts given time available to work on them.

Introduction

The VectorLite Encryption software provides three basic programs: "***buildkey***", "***encrypt***", & "***decrypt***". Version 5M introduces ***"random"*** to assist in creation of a "randomization file" for use with ***buildkey***.

The ***buildkey*** program generates a key-table file, which is used by the ***encrypt*** and ***decrypt*** programs to encipher and decipher files.

The encryption and decryption end points require possession of the same key-table file. It is a symmetrical cryptography system.

The ***cipher-text*** files typically expand from the plain-text file's size by 60 % or more. ***Cipher-text*** files include enciphered false data at the beginning, end, and various points in between. Key-tables randomly self modify during execution when a plain-text file exceeds a minimum size.

The key-table files are rather large for a modern-day cryptographic system – the minimum size is about 132 kilo-bytes (1 mega-bits). Key tables files may include a maximum of 512 "pages", for a maximum crypto-key file size of 33 mega-bytes (268 mega-bits).

As with any other software, modest intelligent usage would be required for the system to be secure. VectorLite makes little attempt to secure the key-table files on the host platforms. Responsibility to do so is left up the end-user, and beyond current scope.

Typical Usage

The VectorLite software package includes three (3) programs for normal operation: ***buildkey, encrypt,*** & ***decrypt***. Use of the ***random*** program is recommended, but optional. All programs are provided in shell / command-line versions without a GUI interface. The general command line syntax of each is:

prompt> **random** <input file> <output randomization file>

prompt> **buildkey** <randomization file> <key-table file>

prompt> **encrypt** <plain-text file> <cipher-text file> <key-table file>

prompt> **decrypt** <cipher-text file> <plain-text file> <key-table file>

All files within the brackets are required. Several non-required command line options exist for each program.

For testing, the following file name and type conventions are recommended:

1	Randomization file type	".ran"
2	Key-table file type	".vec"
3	Plain-text file type	".ptext"
4	Cipher-text file type	".ctext"
5	Decrypted files	< original file type, but ***NOT*** the file name >

File names beginning with "test" are suggested:

prompt >	**random**	lake.jpg	test-1.ran		
prompt >	**buildkey**	test-1.ran	test-1.vec		
prompt>	**encrypt**	lake.jpg	test-1.ctext	test-1.vec	
prompt>	**decrypt**	test-1.ctext	test-1.jpg	test-1.vec	*(note use of the original file type)*

The sequence above preserves the original file by not over-writing it later via ***decrypt.*** The common file-name of "test-1" relates files together. JPG files are convenient for testing, as are MP3 (or other format) music files. When folder contents are displayed in Windows File Explorer, with the preview option enabled, – the JPG file's preview image quickly verifies a correct ***encrypt → decrypt*** cycle, or alternatively displays an interesting and very noticeable preview image corruption in the event ot an error that needs investigation.

The files "test-1.ran" and "test-1.vec" would necessarily be secured and protected as secret keys in the real world. It may be wise to secure the file lake.jpg, or not make it known as the source of input to ***random***.

Files may be encrypted with multiple passes of ***encrypt*** using the same or different key-table files. The inverse of this process must be performed with the ***decrypt*** program, using key-table files in reverse order if different.

The Random Program

Usage: ***random*** < variable byte file > < randomized byte file >

Where: < variable byte file > = File to extract randomized byte values from

< randomized byte file > = Output file of randomized values 0 to 255.

Options: /h help

/q quiet mode

The ***random*** program generates a file of randomized bytes, based upon an extraction of byte values from the input "variable byte file", and subsequently scrambling them. The resultant output file often appears to make a more suitable file for input to the ***buildkey*** program.

The input file may by any type of file which meets 3 minimum qualifications:

(1) The file must contain at least 225 of the 256 byte values from 0 to 255

(2) The byte value quantities coefficient variable (cv) must be 0.5 or less

(3) The file must be 10,000 bytes or larger.

An input file size greater than 100,000 bytes is recommended. The output file of randomized values is currently hard fixed at 100,000 bytes. This may change in the future. The author uses JPG files for testing.

After the input file qualifications are checked, ***random*** will ask for a minimum of 5 integer values to use as independent seeds to the C language runtime library srand() function. A maximum of 200 values may be input. An input value of zero (0) terminates the input.

The algorithm used to generate the random numbers scans the input file randomly to obtain one byte at a time, slowing increasing the limit of common byte values obtained. The number of common values allowed to accumulate increases with subsequent randomized passes through.

The first integer input seeds the srand () library call to obtain random indexes to scan the input file. All subsequent integers re-seed srand () to scramble the byte's order they were obtained in.

Preliminary analysis of the output files indicates relatively uniform byte value quantities, spacing, with the minimal byte patterns which must exist statistically as predicted by probability theory.

The Buildkey Program

Usage: **buildkey** < randomization file > < key-table file >

Where: < randomization file > = Input file

<key-table file > = Key-table output file to create

Options:			
/a	<value>	alpha key-table page count	(1 to 255)
/A	<value>	alpha element variation	(10 to 50)
/de	<value>	debug output level	(1 to 3)
/h		help	
/p	<passwd>	encrypt key-table w/ passwd	(chars: 8-20)
/q		quiet mode - no banner out	
/r	<value>	Random number seed value	(0 to 9000000)
/s		Append suffix bytes at EOF	
/v	<value>	vector key-table page count	(1 to 255)

Buildkey constructs a VectorLite key-table file. The program requires a "*randomization*" input file. This randomization file may be of any type; however, output files produced by ***random*** are recommended. The utility program ***file-stats*** provides statistics on a file's suitability for use.

The randomization and key-table file-name arguments are required. Options following are not.

Use of the "/r" option is highly recommended, and suggested to always differ.

Buildkey randomly indexes into the "randomization file" to obtain random numbers.

Previously existing key-table file-names are over-written without warning. Key-table files generated with the same "randomization file" should be different upon each creation, with intelligent use of the "/r" option.

Best Practice:

Before using a "*randomization file*", run the ***file-stats*** and ***histogram-bytes*** utility programs to help determine suitability.

The "/a" and /"v Alpha & Vector Page Options

The "/a <n> " and "/v <m> " options may be used to add additional alpha (/a) and vector (/v) key-tables. By default, key-tables are constructed with 1 each. The encrypt program will randomly select additional key-tables different from the 1st of each after a minimum plain-text file size, if they exist.

The "/A" Alpha-Variation Option

Basic alpha key-tables contain 256 elements of each byte value – unless the "/A" option is specified. "/A" enables ***buildkey*** to randomly vary the quantity of each value +/- within the number range provided. This provides many more alpha key-table possible permutations. For example: If the number 20 is used – byte quantities of any given value may vary from 236 to 276.

The "/de" Debug Option

"/de <n>" sets the output developer debugging text verbosity level. Output is written to STDOUT.

The "/h" and "/q" Help / Quiet Options - Self Explanatory

The "/p" Password Option

A key-table file may be password "protected" with "/p". The string provided is used to crudely encrypt the key-table file. It is not intended to protect the file's content, but simply to confound searches for key-table files based via the known VectorLite key-table file "MAGIC ID" prefix.

The "/r" Random Seed Option

The "/r" option causes the ***buildkey*** custom random number generator seed the C language rand () function with value provided, rather than with the current system clock time.

The "/s" Suffix Option

The "/s" option appends a random quantity of "suffix" bytes at EOF, to disguise the content length. This makes file system searches for a key-table file via a file's fixed length more difficult.

Key-table files contain a Unix / Linux style MAGIC-ID byte string at beginning and file end. The VectorLite byte sequence is 57-62-19-16.

The Encrypt Program

Usage: **encrypt** < plain-text file > < cipher-text file > < key-table file >

Where: < plain-text file > = Input file (p-text) to be encrypted
< cipher-text file > = Output file (c-text) to create as output
< key-table file > = VectorLite key-table file to be use

Options:			
/de	< value >	debug output level	(1 to 3)
/h		help	
/kt	< filename >	enable key-table trace	
/p	< string >	key-table password	(8-20 chars)
/q		quiet mode	
/r	< value >	Random Number Seed	(0 to 9000000)
/rt	< filename >	enable random number trace	
/t	< value >	Number of concurrent threads	(1 to 8)

Encrypt translates a ***plain-text*** input file to produce a ***cipher-text*** output file, using a VectorLite key-table file as a secret key.

Previously existing output ***cipher-text*** files will be over-written without warning. The key-table file supplied to ***encrypt*** must be used by ***decrypt*** to re-create the plain-text file's original content. ***Encrypt*** verifies use of version 5M key-table files. ***Cipher-text*** output files have no identification information contained within.

File names within brackets are required. Use of any option is just that – none are required.

Incorrect syntax causes the usage information above to be displayed.

Encrypt creates as many threads are there are cores on the platform found, plus 1, by default (8 max).

The "/de" Debug Option

"/de <n>" enables debug output and sets the verbosity. Debug files have the file-name root of the cipher-text, one file per thread, with a "_en_x.txt" suffix and file type. "x" is the thread number. Debug files are NOT merged upon completion. "x" = "m" is for the main thread. "x" = 0 for the table of contents thread.

The "/h" and "/q" Help and Quiet Options - Self Explanatory

The "/kt" Key Trace Option

"/kt < file-name >" enables the key-table trace output file. Fixed length binary records of each key-table cell usage is written. There are one or more records per plain-text file byte. Trace files are temp written to the cipher-text file-name root, with a "_x.kt" file-name suffix and type. "x" is the thread number. Temp files are merged at the end and deleted. Key trace files may be analyzed with the ***key-trace*** utility program.

The "/p" Password Option

The password option is used to provide the password string if the key-table file was constructed with "/p".

The "/r" Random Seed Option

"/r < value >" enables one to enter a random number seed value for the C library srand () call. If not specified, ***encrypt*** will use the current system time as the seed value. An approximation of the current system time used is given away by the cipher-text file creation date. Hence, use of this option is recommended, with variation to not repeat. Encrypt uses random numbers to continually vary the pre and post enciphering bias arrays, false data generation (which is enciphered), and the key-table self modification scrambling process.

The "/rt" Random Number Trace Option

"/rt < file-name >" enables the random number use trace file output. One byte is written per random number use. Trace files are temporarily written to the same file-name root of the output cipher-text, one file per execution threat, with a "_x.rt" file-name suffix and file type, where "x" is the thread number. At the end of execution, these files are merged to a final output file specified in the command line. The ***file-stats, histogram-bytes ,*** and ***check-pattern-bytes*** utility programs may provide insightful information afterwards.

The "/t" Thread Count Optional

"/t < n >" is a developer option to specify the number of threads to create for encryption. Values from 1 to 8 are permitted, and is trimmed back to the number of platform cores if less. A main thread always runs 1st, followed by <n> threads, and upon termination of all <n> threads – a final Table of Contents thread.

The Decrypt Program

Usage: ***decrypt*** <cipher-text file> <plain-text file> <key-table file>

Where: <cipher-text file> = Input file-name of encrypted data
<plain-text file> = Output file-name to create
<key-table file> = Key-table file to use for decryption

Options:				
	/de	<value>	debug output level	(1 or 2)
	/h		help	
	/p	<passwd>	key-table file passwd	(chars: 8-20)
	/q		quiet mode	

Decrypt transforms a VectorLite created ***cipher-text*** file back to the original ***plain-text*** file contents, when used with same key-table file the original was enciphered with. Previously existing files are over-written without warning. ***Decrypt*** verifies a version 5M key-table is specified. VectorLite ***cipher-text*** files contain no identification and hence there is no check for a valid cipher-text file as input. It is also not possible to determine if the correct key-table file is specified. Garbage in → garbage out, as the saying goes. An incorrect key-table file, or a file which is not a VectorLite cipher-text file may cause erratic program behavior.

The "/de" Debug Option

"/de < value >" enables developer debug output and sets the verbosity level. Debug files are created in the same manner as with ***encrypt***, but with the two characters of "de" replacing "en".

The "/p" Password Option

This option is required if the key-table file was password protected when created by ***buildkey***.

The "/h" and "/q" Help and Quiet Options - Are self-explanatory.

Utility Programs

Several utility programs are included within the download zip file**.** They reside within the sub folders “test” and “tools”. They are unverified work in progress programs.

The ***histogram-bytes*** program generates a histogram of the byte values within a file to the STDOUT. It is useful to determine if a *randomization file* to the ***buildkey*** program contains a wide and equal distribution of byte values. It also provides information on input plain-text and output cipher-text files.

The ***key-summary***, ***key-details, key-line-col,*** and ***key-dump*** programs display various aspects of a key-table file. The author typically redirects the STDOUT to a text file to read with a text editor.

The ***key-trace*** program produces key-table file use history from an ***encrypt*** program execution. The program has three functions, which may warrant separation into 3 individual programs later. Histograms of key-table cell usage, and byte values the various stages of encryption may be displayed. ASCII histogram values may be output to STDOUT as well, and binary histogram values can directed to an output file (so as to be input to spreadsheet software for charting).

The ***create-pattern-file*** program is used to generate plain-text input files consisting of various repetitive binary byte values or ASCII string patterns of a specific size.

The ***compare-files*** program is used to compare ***decrypt*** program output files match the original **plain-text** files enciphered by the ***encrypt*** program.

The ***check-pattern-bytes-1*** and ***check-pattern-bytes-2*** programs are used to determine the duplicate repetitive byte patterns with a file. The 1st program generates inclusive results, meaning a duplicate pattern of 4 bytes has also incremented the duplicate pattern counts of 2 and 3. The 2nd program is exclusive and counts of only the largest. The programs may not handle the end of file case perfectly for the last few checks.

The ***check-proximity*** program reports the average spacing distance between common byte values of a file.

The ***file-stats*** program reports the mean, standard deviation, and coefficient of variance of the histogram of a file’s byte values.

Basic Encryption Algorythm

The VectorLite ***encrypt*** program transforms an input plain-text file into a randomized "displacement-domain" cipher-text file of randomized values. ***Encrypt*** translates 1 byte at a time into the displacement / distance value to an entry within a row or column who's cell entry matches within a 2 dimensional 256 x 256 element "alpha" key-table. The 1st matching value is used, should multiple exist. The "alpha" key-table is traversed in the direction (x,y) and (+, -) as determined by a parallel indexed and tracked a "vector" key-table. The (x,y) coordinates of each are identical at all times.

The ***encrypt*** program is much more complex than the basic method described above. Plain-text bytes are both pre and post "biased" with singly used random number pools to eliminate or reduce input file patterns. Blocks of zeroes and patterns are common with computer files. In addition, input files such as text files make use of a sub-set or unequal use of values from 0 to 255. The pre-bias eliminates these input stream problems. The post-translation bias eliminates a property of the displacement translation to favor smaller values statistically.

The ***encrypt*** program may not find a plain-text byte value searched for within the current row or column the "alpha" key-table state is currently at (x,y wise). The miss rate is expected to be ~38 %. When not found, ***encrypt*** searches a "re-vector flag" within the "vector" key-table within the current row or column, traversing the same direction as the failed plain-text byte search. The resulting "vector" table displacement is output to the cipher text file (also after post biasing + re-vector bias). Re-vector operations are guaranteed to change the searching direction of the next "alpha" table scan so as to prevent looping. ***Encrypt*** starts the search anew. With an observed 38 % miss rate, 10 consecutive misses become probable after a certain input file size. When such happens, a special "evasive maneuver" action is taken to "jump" elsewhere to a random location with the current row / column in a different direction as well.

A random amount of encrypted false data is always inserting into the beginning and end of cipher-text files. Input plain-text files beyond a small size will cause false encrypted random data to be inserted midway. The larger the plain-text file – the larger the quantity and number of mid-stream insertions.

The "alpha" key-tables randomly self modify / scramble 4 kilo-byte segments of themselves once a minimal plain-text file size is exceeded. The segment scrambled may start randomly from key-table starting table byte 0 to the end minus 4 Kbytes. Self modification does not cross "alpha" key-table "page" boundaries. A "page" is an individual "alpha" 256 x 256 byte key-table (65,536 bytes, or 524,288 bits). There may be 256 of these.

Bias arrays start in known states from the ***buildkey*** key-table file. Each bias array element is used 1 time only. Upon a random threshold near the array buffer pool end, the arrays are updated to a new set of values and a different sequence.

5 addition "vector" table flagged cells define actions to synchronize ***encrypt*** & ***decrypt***. (1) Change "alpha" page; (2) Change "vector" page; (3) Toggle false data on / off; (4) Alpha table self-mod; and (5) bias array update. Each flag type is guaranteed 1 per row / column min, with the "revector" type at 2 min. More than 1 or 2 may exist as column guarantees are made after 1 or 2 per row first.

Key-Table Structure(s)

VectorLite Encryption Ver 5M utilizes a 2-dimensional “alpha” key-table, as illustrated in Figure 1. The dimensions of the table are as wide as the range of possible data values to be encrypted at a time – 8 bits.

VectorLite 5M encrypts one 8 bit byte at a time, so the alpha key-tables are 256 x 256 in the (x,y) dimensions (using Cartesian coordinates). The table element size (depth) matches the depth encrypted at a time: one byte.

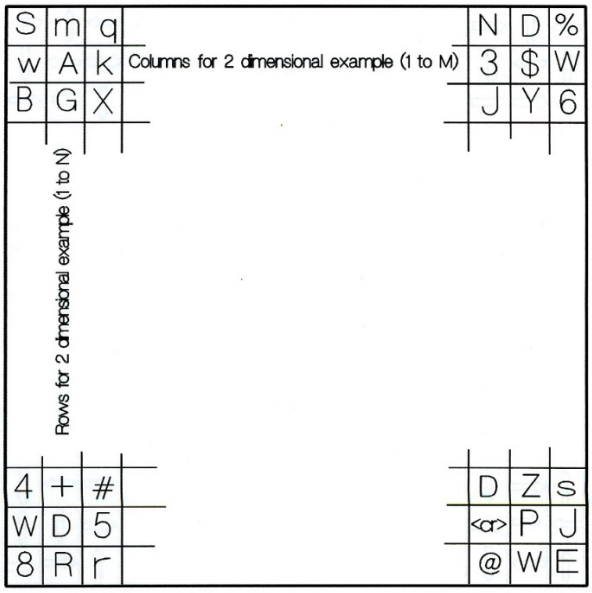

Figure 1 – Alpha Key-Table “Alpha” Elements

During ***buildkey*** program creation, alpha key-tables are populated as follows:

First, a simple table of all possible values within a ***plain-text*** file (0 to 255) are placed into the table in equal quantities - 256. Like values are set +1 column per row, to ensure each row and column has every value. Next, the equal population of 256 values is modified if the “/A” option was specified. The value following the “/A” option is the quantity range each value may randomly vary +/- from 256. A final adjustment to a few value quantities is performed at the end to compensate for any +/- accumulations.

Next and final is a series of random table scrambles, exchanging elements 0 to 65,535 sequentially with a randomly indexed element at some other table location. This is repeated a large random number of times.

After scrambling, the alpha key-table is written to an output file. When multiple alpha key-tables are specified via “/a” – the process repeats 1 to 254 times. When the table count is 256, the resultant key-tables are like a 3-dim key-table, but performance is better using 1 table (x,y) plain at a time.

Figure 2 illustrates a portion of a “vector” (aka “attribute”) key-table. The table is of the same dimensions and depth as the alpha key-table.

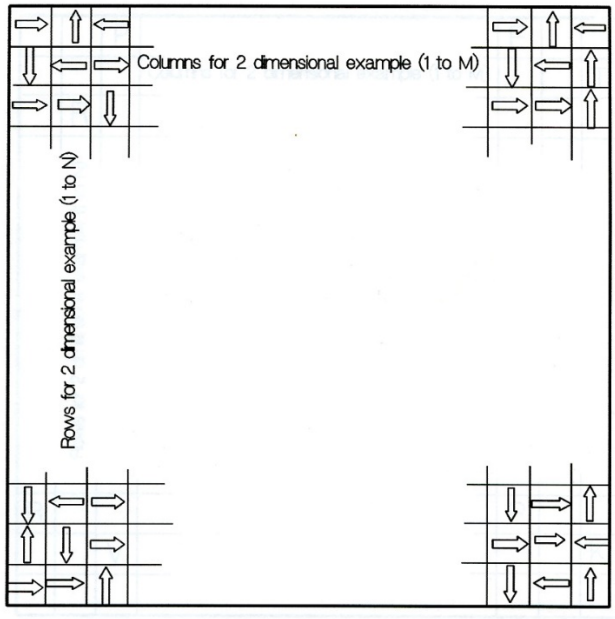

Figure 2, Vector Key-Table “vectors”

A “vector” table contains the next value alpha table search directional guidance (i.e. vectors, hence the name), and flag bits. Flags synchronize ***encrypt*** and ***decrypt*** actions: such as “re-vectoring” when an alpha search fails; or changing the active key-table page number. Figure 2 shows only the “vectors”.

Two dimension tables have four possible traverses, and the software uses literals representing N, E, S, & W for simplicity. These literal values of 0 to 3 consume the 2 LSBs of every table element byte.

Version 5M defines 6 bit flags from the remaining MSB bits. They are: “ReVector”, “False Data On / Off Toggle”, “Self-Mod”, “A-Page”, “V-Page”, and “Bias Array Update”. Only one flag bit may be defined per table element. The corresponding (x,y) “alpha” element of a flagged “vector” table cell is ineligible as a displacement translation result. Landing on a flagged “vector table” cell always indicates an action.

Construction of a “vector” key-table is performed by populating the table with an ordered known set of values 0 to 3 sequentially. The table is then randomized by scrambling the entire table a large number of times.

Flags are inserted after the vector direction element values are scrambled. Flags have a requirement to be present with each row & column and cannot be randomly scrambled and missing. Flagged cells make ~3 % the alpha table entries ineligible for use.

Key-Table Files

Vectorlite Encryption enciphers plain-text files one 8-bit byte at a time. Basically, a plain-text input value is translated to the distance value where the input is found within "alpha" key-table. Plain-text bytes are biased before the search to eliminate patterns and statistically equalize the values search for. The displacement results are also biased to eliminate 1st found value favoritism.

Key-File contents and structure:

1	File Header	8 bytes to identify VectorLite file type, version, sub-version
2	A Tables	Alpha key-tables, quantity 1 to 256, each at 64 KB size
3	V Tables	Vector key-tables, quantity 1 to 256, each at 64 KB size
4	A Bias Array	Alpha bias array of 1024 random bytes values
5	D Bias Array	Displacement bias array of 1024 random bytes value
6	R Bias Array	Re-Vector bias array of 1024 random bytes values
7	Suffix Bytes	Optional, a random number of random bytes to deceive file size based searches
8	File Header	8 bytes same as (1), but at EOF used to verify correct password unlocking

Key-table files may vary in size from approximately 132 kilo-bytes to 32 mega-bytes, dependent upon the number of alpha and vector tables. At the maximum , there are 256 of each for a 512 total. The maximum size approximates the disfavored 3-D table VectorLite version 6 demo release. 3-D tables are not cache or memory bandwidth PC or server friendly, but 2-D table plains are with today's CPU cache sizes.

The active Alpha and Vector table pages, should more than 1 exist, maintaining common key index locations relative to the start of each. In other words, the (x, y) coordinate location for active "alpha" page is always identical to the (x, y) coordinate of the active "vector" page. The relative page numbers active will vary.

Password protected key-table files generated with the "/p" option are verified by ***encrypt*** and ***decrypt*** via the proper file header ID and software version numbers at the start and end of file.

When ***encrypt*** finds multiple "alpha" or "vector" pages within a key-table file, "alpha" and "vector" key-tables are "paged" at different random thresholds. An input plain-text file need be sufficient minimal size to warrant a change. The "alpha" and "vector" active page table numbers often differ. ***Encrypt*** signals these changes to ***decrypt*** by purposely landing on either an A-page or V-page flagged "vector" key-table cell. The page to switch into is the placed into the cipher-text immediately afterwards, and post-biased so as to randomize it's file value and hence weakly encrypt it.

Synchronization between ***encrypt*** and ***decrypt*** is maintained via purposely landing on flagged "vector" key-table cells throughout the entire encryption process, when special actions are desired. The placement of "sign-post" values for these special actions is eliminated by using this process. The values to use in the special actions are within the cipher-text, but are post-biased to weakly encrypt and randomize them all – hidden in plain sight among the true and false encrypted plain-text → cipher-text data.

Encryption Method / Algorithm

The Basic Encrypt Program Algorithm

1 Unlock key-table file if password protected, and verify the file was constructed via buildkey current version

2 Read all “alpha” and “vector” Key-Tables into program memory (1 to 256 of each).
Set key-tables to the first page of each to start (Page 0 in the C language).

3 Read the 3 bias arrays of 1,024 random numbers each, from the key-table file (A, D, and R bias arrays).

4 Set the first key-table traversing “Direction Vector” to that of the vector table origin [0,0].

5 Read the first or next 8-bit byte of the input plain-text file (version 5M buffers up to 100k)

6 Change the value of the plain-text byte using the alpha bias array value at the current index, and increment the index. This create the “alpha” key-table search value. The operation is (PT + A-Bias) % 256.

7 Search the “alpha key-table,” in the direction of (4), for the value of calculated (6).

8 If (7) is unsuccessful at locating (6) – search for 1st Re-Vector flag cell in direction (4). Apply R and D bias to the returned value with math as (6). Output result into the cipher-text file. Repeat steps (7) – (8) a max of 10 times if not found continues on. If 10th not found occurs - executing an evasive “jump” by placing a binary 0 + a jump value distance to randomly change “alpha” / “vector” table (x,y) . Re-start steps (7) and (8) anew.

9 If step (6) was successful, apply a D bias to the distance value returned from the “alpha” table search, and output the ***cipher-text*** value. Increment the D bias array index.

10 Update the new “vector direction” to the “vector” key-table cell value (9).

11 Update the key-table indexes for the active “alpha” and “vector” key-tables to the location (9)

12 < rinse and repeat – i.e. go to (5) –> (until plain-text file EOF is reached)

Crossing key-table row or column edge boundaries uses circular wrap-around: (255 + 1 = 0), and (0 – 1 = 255).

The alpha search probable miss rate is 38 % and consequently cipher-text files typically >= 60 % in size due to consecutive misses in (8). The probability of consecutive alpha search misses is 0.38 ** N. The ineligible flagged vector cell alpha table elements contribute ~ 3 % to the miss rate – less than 10% of the total misses.

Deviations From The Basic Encrypt Steps:

Encrypted false data is inserted at the beginning and end of every file. The quantity of false data is random and proportional to the plain-text input file size. When the plain-text file size crosses various thresholds, additional randomly placed and larger randomized quantities are placed mid way in batches. The larger the file the larger the number of insertions and the larger the quantity – up to a reasonable maximum.

False data start and end points are signaled from ***encrypt*** to ***decrypt*** by insertion of the biased distance to a "false data" flag in the "vector" key-table. These flags are on / off toggles.

Key-table displacement values of zero (0) are not permitted. When pre-biased input plain-text creates consecutive identical values, use of the current (x,y) location in the "alpha" table is not permitted. A new location must be found. **C*ipher-text*** files will not contain zeroes, except under the circumstance of a ReVector deadlock jump, in step (8) after 10 unsuccessful search attempts.

Before an attempt is made to to encrypt a false or plain-text data item, thresholds are checked to determine if a time a special action – such as bias array update, key-table self-mod, or table page change.

Multi-Threading

Version 5M ***encrypt*** and ***decrypt*** is multi threaded. There is a main program to prepare the encryption threads, followed upon completion by a Table-Of-Contents (TOC) thread, followed by the main program's assembly of the final cipher-text file from individual cipher-text temp-files created.

A TOC segment resides at near the start of cipher-text files. The segment begins with a random quantity of enciphered false data before the true TOC biased data is encountered. Each thread cipher-text segment begins with different randomized quantities and values of encrypted false data, and ends similarly. Each thread segment also inserts enciphered biased false data mid stream in proportion to segment plain-text input size assigned by the main program.

Multi threads are created if the plain-text input file size warrants their creation for a performance gain. Plain-text input files are divided approximately in equal size but with a randomized variance.

A 4 core, 8 thread Intel I-7 4700 cpu has thread performance gains significant up to 3 or 4 cores, with diminishing returns there after. The number of encryption threads is bounded by the core count found on the CPU and a current upper limit of 8 for demonstration purposes. The desktop PC used to develop the software observes decreasing marginal gain after 3 threads, likely due to home desktop memory bandwidth limits. Servers likely will have a higher performance yield.

Future Thread Enabled Features

Multiple threads create independent self contained thread cipher-text files. These files can be independently deciphered in a self contained manner. This enables two desirable potential future features to be added:

1. Transmission of cipher-text segments via distinct communication channels, reducing whole file interception
2. Randomized scrambling / interleaving of the independent segment transmissions, reducing segment interception.

The Basic Decrypt Program Algorithm

Deciphering the Cipher-Text back into original Plain-Text requires the same key-table file used by ***encrypt***.

The cipher-text file is processed 1 byte sequentially at a time, using an I/O buffer up to 100 KB

1 Read in Key Tables and bias arrays from the key-table file into program memory

2 Set the current table traversing "Direction Vector" to the Vector Table origin [0,0].

3 Read the first or next 8-bit byte value of ***cipher-text*** from the input file.

4 Compensate for the D-bias of the input byte. Increment the D bias array index.

5 Traverse the alpha and vector key-tables by the value (4) in the direction of (2)

6 Check for a < revector or other flag > within vector table - if flagged, handle the special operation cases:

- 6.a If a re-vector flag, update key-index by R bias
- 6.b If a bias flag, read the update values from cipher-text and update the arrays
- 6.c If and alpha or vector key-table page change, read new page value update page number
- 6.d If a false flag, toggle false data processing from prior state
- 6.e If a self-mod flag, read the update parameter update the 4 KB alpha key-table segment.

7 If (6) is not a flagged cell , reverse the pre-bias. Increment the alpha key-table index.

8 If false data toggle state is ON – ignore (7) result. If false data state is OFF – pwrite (7) to output file.

9 Set next key-table traverse direction to new "vector" key-table value at new index.

10 If EOF - <close file>; else - if not EOF <goto 3>

Notes and Comments

Key-table Sizes: The key-tables are large for an encryption system, but with today's large secondary storage capacities, solid state disk drives, and computer main memory sizes , complaints of this should be mute. On an industrial scale, key-table libraries or vaults would be larger, but storage is massive continues to grow.

Cipher-Text File Sizes: The same argument is made concerning the minimum 60 % cipher-text file size expansion above the plain-text input file size.

Secret Key Type: The encryption implements symmetric keys. Investigation into an asymmetric implementation or enhancement is taking place - time permitting. Investigation into use within protocols like SSH is underway.

Nested Encryption / Decryption: multiple pass encryption via the same or different key-files tests successfully. Additional encryption passes causes the 60 % file size expansion each. Key-files must be used in reverse order during the decryption if different.

Encryption Method: The basic enciphering is primitive in complexity compared to those using polynomials and complex mathematical algorithms. The end goal is protection of the data, not status in the complexity to do so. VectorLite attempts overwhelm brute force or statistical analysis with massive possible key-tables permutations, and attempts to ensure every aspect of the process is randomized. This simple method was not practical until modern small computers became common place with sufficient memory and cache 15 years prior.

Hardware: Computer CPUs & ALUs do need nor require the recent AES additions. KISS.

Working as Desired: The ***encrypt*** "key-trace" option generates a key-table usage log during the encrypt program execution. The files provide a history of plain-text, biased plain-text alpha table search values, alpha search displacement results, final biased cipher-text values, key-table page and indexes used, in addition to a few more items. Data from analysis of these files appears to indicate VectorLite uses the key-tables in the desired statistical random manner and does not favor individual cells. A bell shaped use curve is observed that flattens as larger plain-text files are encrypted.

Cipher-Text – Prior Version Pattern Recognition: The use of the 3 bias arrays starting in version 5L appears to have virtually eliminated the minor cipher-text duplicate byte sequence patterns detected in prior versions above which probability theory states should exist.

Cipher-Text Value Pattern Probability Distributions: Significant effort has been made to create the nearly 100% random cipher-text results from any plain-text input file type. Test results are published at the end of this document. Specific goals were to "flatten the cipher-text output value quantities" statistically and eliminate duplicated byte value patterns above the minimal due to random chance. The average distance between like byte values also appears to be successfully randomized statistically.

Hence, the testing results thus far indicate near best case pure random output may have been achieved.

Of Sign Posts & Sign Post Values: The encryption process technically uses what can be categorized as "***s****ign-posts*" to signal special actions and synchronize to decryption. Both *sign-post indicators*, and *sign-post values* exist***.*** *Sign-Post indicators* are NOT written into cipher-text; they are signaled ***encrypt*** to ***decrypt*** by landing on falgged "vector" key-table cells. Biased *sign-post values* are written into the cipher-text immediately following the landing on a special flag cell for 4 special actions:

Item	Flag type / Purpose	Bytes	Value Range	Purpose in Encrypt / Decrypt
1	Alpha Key-Table Self Modification	1	rand 0 to 255	Used in scrambling algorithm
2	Bias Array Updates:	3	rand 0 to 255	Value changes & scrambling
3	Alpha Key-table page change	1	rand 0 to 255	Page change value
4	Vector Key-table page change	1	rand 0 to 255	Page change value

Sign-post values are randomized (in most cases within bounds) and biased – making detection challenging.

Buildkey "/A" Alpha-Variability: This option to vary the count / quantity of each value in the alpha key-table could be expanded upon to include variants for ASCII / UTF text document character sets to more heavily favor alpha-numeric & punctuation character set byte values, language. A future version may implement this if investigation into the subject warrants the effort.

Random Number Usage: Obtaining truly random number sequences for any computer program is problematic without specialized hardware. The C language run-time library rand() function is notoriously weak, implemented as pseudo-random. Sequences are identical when seeded the same value, and a database of some type consisting of all possible sequences may actually exist somewhee.

Hence, a custom "get_rand ()" function was created to introduce additional complexity and randomness into the ***buildkey*** and ***encrypt*** programs. The ***random*** program was created to further break library dependence.

Duplicate Functions Among Programs: No attempt was made to generate a link library of common functions or procedures between different programs. Configuration management has not been a development issue.

Coding Style: Simplicity and readability is took precedence over any source code brevity or optimization. Better the compiler optimize performance than someone go bald. White space is cheap, while people's time is priceless.