LEDA-128 - A 128-Bit Linear Encryption and Decryption Algorithm

****

Mandeep Parihar, Kyle Siu

SAIT - Information System Security

Calgary, Alberta

April 24, 2022

Table of Contents

[Executive Summary](#_heading=h.30j0zll) 3

[Linear Feedback Shift Registers in Cryptography](#_heading=h.3znysh7) 4

[Attributes of Linear Feedback Shift Registers](#_heading=h.2et92p0) 4

[Simplified Mathematical Proof for Maximal Length of Linear Feedback Shift Registers](#_heading=h.tyjcwt) 4

[Cryptanalysis Deterrence Methods](#_heading=h.17dp8vu) 6

[Linear Encryption-Decryption Algorithm 128-bit (LEDA-128)](#_heading=h.3rdcrjn) 8

[Front-End](#_heading=h.26in1rg) 8

[Header File](#_heading=h.lnxbz9) 10

[Main Function](#_heading=h.35nkun2) 11

[LFSR Initialization](#_heading=h.1ksv4uv) 13

[Seed Initialization](#_heading=h.44sinio) 13

[LFSR Initialization Using Seed](#_heading=h.2jxsxqh) 15

[File Mapping](#_heading=h.z337ya) 18

[Recursive Mapper](#_heading=h.3j2qqm3) 19

[Encryption and Decryption](#_heading=h.1y810tw) 22

[File Extension Reading](#_heading=h.4i7ojhp) 23

[Initializing the Encrypter/Decrypyer](#_heading=h.2xcytpi) 29

[Encrypting/Decrypting Files](#_heading=h.ixvctdzgx46u) 30

[Strength of the Initialized 128bit LFSRs and the Reversibility of the Files](#_heading=h.r2b1yz226gyy) 36

[Testing](#_heading=h.gpf9fikt51t4) 38

[Problems, Future Improvements, and Recommendations](#_heading=h.qqxutkk1xeh9) 40

[Conclusion](#_heading=h.yq14f6ykorb1) 42

[References](#_heading=h.qsh70q) 43

[Appendix](#_heading=h.5oraewqgg6z8) 45

[Keywords](#_heading=h.5fzsyqyykv4y) 45

# Executive Summary

LEDA-128 is a custom portable symmetric stream cipher encryption algorithm meant to be used on a USB device. The program is meant to be used in scenarios such as protecting the data inside a USB that is in transit – should it be lost; the data is secured by LEDA-128 and the data is rendered inaccessible. The encryption is launched on a computer, and secured via password using a front-end application. The program is initialized using a SHA-256 hash of the passwords and all files within a folder are mapped. This information is given to the encrypter that will encrypt or decrypt the files contents depending on which option the user chose in the front-end. Some recommendations include performance and operational improvements, as well as security improvements for the program. In conclusion, LEDA-128 is a functioning minimum viable product that, though contains some issues, works to secure the contents of a folder, regardless of file type, using a password.

# Linear Feedback Shift Registers in Cryptography

Linear feedback shift registers (LFSRs) are commonly seen in many things such as radio jammers, random number generators, or stream ciphers, but can also be used for cryptography. Their ability to easily produce pseudo-random numbers makes them a viable candidate for building an encryption algorithm around. LFSRs are stream ciphers, meaning that they use a symmetric key cipher where plaintext and a pseudo-random keystream are combined via XOR to create an obfuscated output. Given the mathematics behind LFSRs, they can be manipulated to be very complex or to perform better. They are seemingly simple in structure, but capable of producing complicated outputs. In a cryptographic sense, LFSRs can be used, if built properly, to provide a complex solution that is also relatively fast for securing data, with buildable anti-cryptanalysis techniques.

## Attributes of Linear Feedback Shift Registers

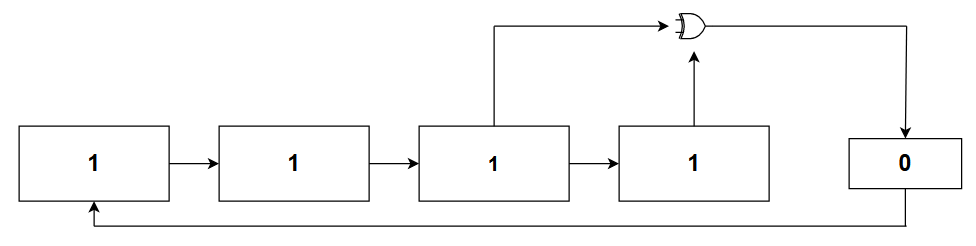
To create a cryptographic encryption algorithm using LFSRs, their basic nature must be understood. An LFSR is a series of binary bits that is known as the state. The least significant bit in the state is shifted out. Because the size of the state can be determined, and will always contain a number of bits equal to its size, the shifted bit will be linearly fed back into the LFSR to create a new state. This is done by using defined rules; wherein taps, which are selected positional bits, such as bits two and three in an LFSR that is 5-bits in length, are selected and assigned logical operations. The logical operation that is used is often XOR, this is because it is unknown what values that were used to produce the resulting bit value. From a security standpoint, the less information that someone has to work with, the harder it is to reverse and break into.

An additional attribute of LFSRs is that the thoroughness of their pseudo-random nature is largely dependent on the value that was given to it as the seed. The more complex and random the seed, the less noticeable any patterns in the LFSRs would be to find. This means that there is a correlated ratio between the pseudo-randomness of the seed and the pseudo-randomness of the initialized LFSR. This highly pseudo-random LFSR can be used to create complex patterns, making the process of finding and abusing them more challenging.

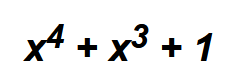
A final aspect of LFSRs to mention is that the state cannot be all zeroes. Should this occur, the LFSR will never leave the state and become stuck. This applies to both the initial state and the state after shifting has occurred. Though an all zero state is not common, it can still occur. This can be remedied by checking the LFSRs current state, and, if it is all zero, the program will commence a reversal or recovery of the work done. Alternatively, the program can inject a deterministic value into the state to remove the all zero, allowing work to continue again. The conditions for the all zero state would be consistent, so the injection will always be done in the same way at the same time during the process.

## Simplified Mathematical Proof for Maximal Length of Linear Feedback Shift Registers

The maximal-length of an LFSR can be determined by using a feedback polynomial that is primitive, which, in mathematics, are values that cannot be simplified further than what they are. This is done by utilizing optimal tap selection to produce an irreducible polynomial. The taps are based on the overall size of the binary state that is desired, and whether or not two (LFSR-2) or four (LFSR-4) taps are desired; though, not all n-bit sized LFSRs support both. A simple example of some of the aforementioned ideas can be demonstrated with a 4-bit LFSR. A maximal-length 4-bit LFSR only supports LFSR-2, at bits three and four. Using an XOR on the taps, and an initial state of all 1’s, the LFSR would look like this:



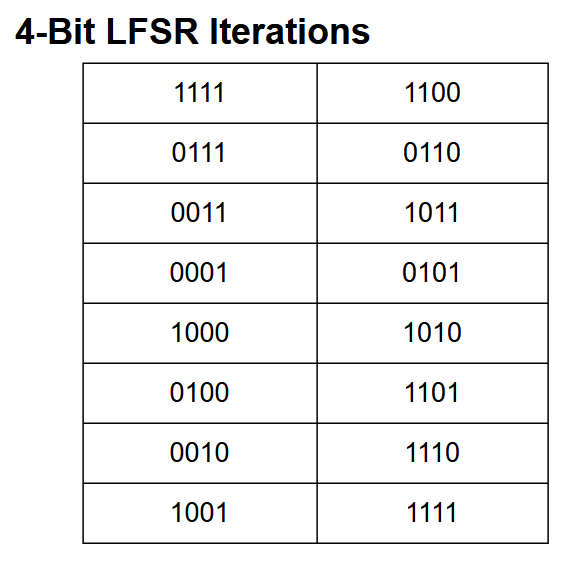
For the 4-bit LFSR example we have chosen, the maximal-polynomial expression can be written out using the taps selected as xn, where n is the value of the tap, equal to the number of taps excluding the “+ 1,” a correspondence to the input to the first bit, at the end of the polynomial equation:

****

The shifted off bit can also be mathematically derived. Where b1 is the bit in the first position, which will be filled in by the shifted off bit, and b’3 and b’4 are the bits chosen as taps for the XOR, the equation used for determining the shifted off bit is:



Looking at the example value used prior, the XOR placed on bits three and four, they will produce the value 0. When everything is shifted to the right by one bit, the XOR’d bit can be fed back into the bit 1’s position to fill in the shifted-out value. This process can be continued for as long as the maximum number of iterations allow. Following the primitive polynomial expression through, using the state depicted in the example above, would yield 16 iterations before returning to the original state’s value.



It can be seen that the initial state reappears at the end, but only at the end. This is the reason that the LFSR needs to be composed of the correct taps, so that they can be given to the primitive polynomial expression to produce the most iterations possible. It is an undesirable outcome to have the taps yield less than 16 iterations. That would make recognizing a pattern much easier, and would be a foot in the door for reversal.

## Cryptanalysis Deterrence Methods

Once the initial state reappears during the shifting, the LFSR is easier to find a pattern in and break. The size of the LFSR can be determined, and it can be assumed that the taps are chosen to provide maximal length. Though that is not always the case, it is cryptographically advantageous to do so. Knowing this, cryptanalysis becomes easier. To make aspects of the process difficult to reverse, it is possible to apply additional shifts, or decimations, to the state during the process to create a seemingly unpredictable output and shift value. When the information required for pattern recognition is altered, the cryptanalysis is impaired.

Further impairment is found by removing or limiting what information is given out. If a seed value is generated, then the seed should be hidden. LFSR output streams are deterministic, meaning that knowing the XOR gates and the current state can allow someone to predict all following outcomes.

The size of the LFSR should not be information that is given out, nor should any logical operations occurring in the program. These are rather basic, but they are some of the mandatory things to consider when attempting to ensure the integrity of the cryptographic LFSR.

A more advanced approach is to try and protect the inherent faults of LFSRs. Given the scenario where two different messages that were produced using the same key are intercepted, it is possible to recover the XOR of one value with another, and reverse the process. An attacker can decrypt both messages and will now have a reusable key to easily decipher any further messages. The simplest method of avoiding this is to never use the same key twice. If the same key is never used, then the entire cracking process has to be done. This increases the effort and skills required of a cryptanalyst to reverse the encryption.

Bit-flipping attacks can be done on partially or fully reversed messages even if the key is not known. This occurs by shifting some of the bits of the message to alter it, yielding inauthentic messages. An example of this attack would be flipping bits to alter an integer value that corresponds to a money amount. Preventing this attack is more involved, but using a message authentication code (MAC), the message can be tagged with an authenticity code that, when read, can be compared to the file to verify that it is unaltered. The MAC is conceptually similar, though not the same, to a verifiable hash provided during the download of an executable or ISO file.

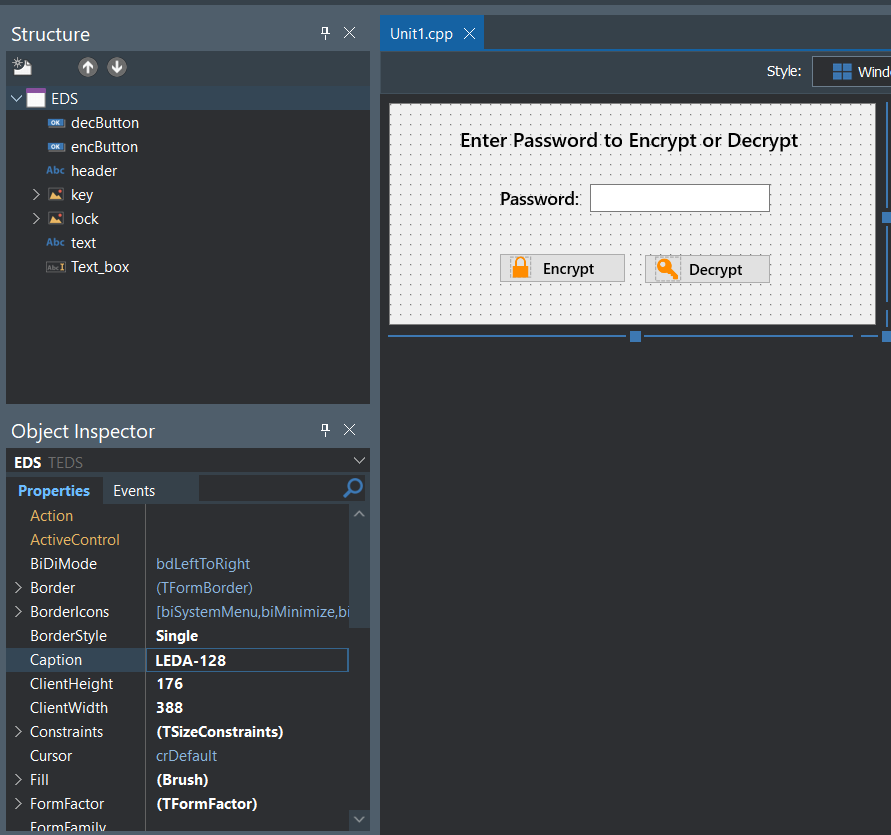
There is much more, both mathematically and conceptually, that is happening with LFSRs; however, those aspects will push the report out of the project's scope. For simplicity's sake, the information required to understand the created encryption algorithm has been covered, and will only be expanded where relevant.

# Linear Encryption-Decryption Algorithm 128-bit (LEDA-128)

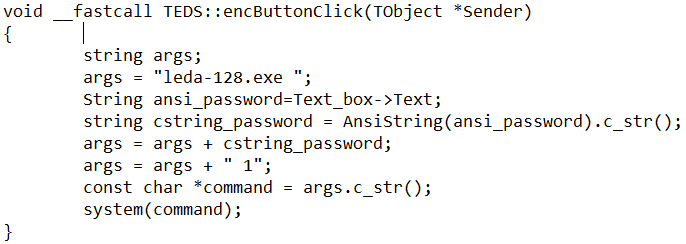
As explored in the previous section, the custom encryption algorithm will be utilizing the elements of LFSRs. To create a complex solution, there will be more than a single LFSR used for the encryption and decryption process, alongside custom methods of obfuscation to deter any cryptanalysts. Lastly, to add complexity, all LFSRs are created to be 128-bits in length, allowing for a high number of iterations due to maximal-length tap selection. The main program is executed via a front-end, which passes the password and which button was pressed to the program. This causes the main program to run. The process of the program can be viewed as operating in three distinct stages, where one is required for the next, and information is created or passed from one stage to the next. These stages are the initialization stage, wherein the seed is generated and the LFSRs are initialized with values. The next stage is the file mapping, used to find all files and folders within the folder that is being encrypted, and the final stage is the encryption or decryption of the binary values of the discovered files. Code execution will require moving between interwoven functions, so for simplicity's sake, the functionality of the program will be explored close to the code execution, and not in order of code appearance. This should create a relatively linear map of how the process is happening.

## Front-End

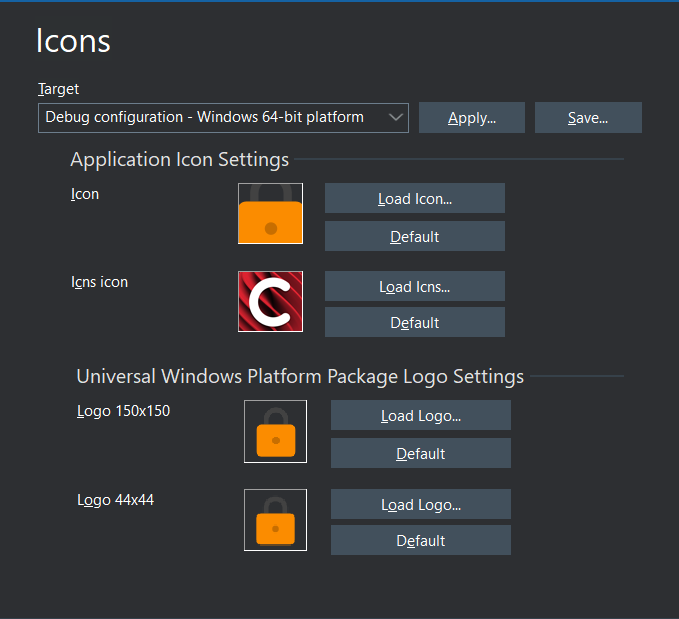
The front-end of the application was developed in a C++ builder called RAD Studio, which was created by Embarcadero. Most of the code was boilerplate, being generated based on what was done inside of the design window:



The code, outside of what was auto-generated, that was written, was as follows:



This is the function for when the encrypt button is pressed. The function begins by initializing a string, for the arguments that will be passed to the command line. Then a String will be initialized and given the value of the user's password from the Text\_box, Text element, as a ansi string. In the next step we will need to add the password and the encryption flag to the args variable but for that to work we will first need to change our ansi string to a c string so it can be manipulated. Once the cstring\_password variable has been assigned the value from the ansi\_password variable we can now append the continents of the string to the args string, resulting in “leda-128.exe password”, lastly we will append the encryption flag which in this case is 1. Once the flag has been added to the string we will have to create a const char variable type, as the system function only takes const char variables. Once the string has been converted it can then be used by the system function. The decryption function is very similar to the encrypt function, the only difference is the name of the function and the flag that is being passed, in the case of decryption the flag is 0.



Above is an image of an option provided by Embarcadero that allows us to change the icon of our application. It changes its appearance in the task bar as well as task manager. This was a simple change but we believe that it really adds some quality to the application.

## Header File

The header file, encrypter.h, is used to establish necessary header imports, macros, and structs that will be used for the primary code file. To understand the main code file, the key components of the header file must be understood.

The datatype uint64\_t is used for many different variables to store 64-bit integer values, but they must also be unsigned to prevent any LFSR from going below 0, unless by error such as an overflow. To make use of that, the stdint.h header file is required:

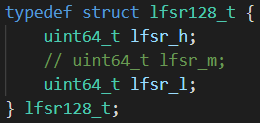


After the required datatype is made available, a macro is required to establish just how many bytes of data can be read from a file at a time. For ease of operation, a total of 4096 bytes was chosen:

****

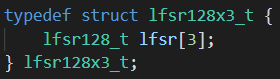
This amount would allow for most smaller files to be read easily as a single chunk, but also allow for sizable chunks to be read from files quickly. Should any data corruption occur, only 4096 bytes of the file is damaged. Though this is not ideal or desired behavior, any unexpected behavior should be kept small scale. For this, 4096 bytes is a viable size.

There are two structs made, of which, one is woven into the other. The one woven into is a struct that initializes two different uint64\_t variables, which will be the first 64-bits and the last 64-bits of the 128-bit LFSR. These are created in a struct named lfsr128\_t, meaning that these are for an LFSR of 128-bits. The “\_t” is a simple identifier that the struct contains data types that are uint64\_t:



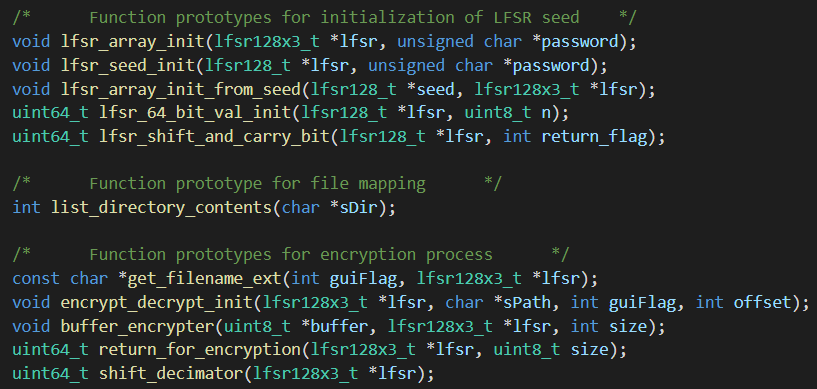
In the code, it can be seen that the variables are named in accordance to their position within the 128-bits. The variable lfsr\_h is used to store bits one to 64, and lfsr\_l is used to store the remaining bits from 65 to 128. The commented out variable lfsr\_m is part of the groundwork that would allow the program to go from a 128-bit encryption algorithm to a 192-bit encryption algorithm. Though it is out of scope, it is partially created, but not implemented in any way. There will be other instances of this code, which is commented out as well.

The final struct contains a single variable of the datatype of the previous struct. This variable is used to for an array with a size of 3:



Because the previous struct is used as the datatype for this struct, each element of the array will have its own lfsr\_h and lfsr\_l. This means that every element of the array will house a single 128-bit LFSR, of which there will be three in total. These LFSR will be used for encrypting and decrypting, except for a single LFSR that is used for obfuscation via LFSR state decimation for both LFSRs. That is an implementation that will be explored later. The name of the struct is identical in meaning to the first, except for the “x3,” which is only used to easily identify that the data type will be used for referencing the LFSRs.

The remainder of the header file is used to create function prototypes:



These functions will be explored in greater detail in the below sections of this report.

## Main Function

The main() will be called from the front-end program when either the encrypt or decrypt button is pressed. Upon being pressed, the aforementioned Command Prompt parameters will be passed to main(). These entries are read into main() as parameters:



Both int argc and char \*argv[] are used for parsing the Command Prompt, and int gui\_flag is used to store the value of the button that was pressed. These parameters are parsed in the order given on the Command Prompt terminal.

Within main(), the following declarations and initializations are made. Firstly, the lfsr variable is created using the datatype of the struct lfsr128x3\_t:



This variable is used to access the specific elements of the struct using an arrow operator. The next variable is the name of the folder that is being targeted as the root folder for the encrypter/decrypter:



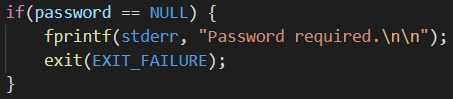
For this program, the folder “SecureFolder” is used. Because this folder is to be in the same folder as the front-end’s executable file, alongside the executable for the encryption algorithm, the name “SecureFolder” does not require anything additional. The final declaration is for a variable that will store the user entered password:



The variable is initialized as NULL to avoid any data from being present. Any value stored in it would affect the intended password, likely rendering all files irrecoverable because the same password may not have the same prepended or appended data. When the password is parsed from the Command Prompt terminal, there is a discrepancy with the datatype, so it is converted to something usable:



Additionally, if the password were somehow lost or considered empty, the program will exit to prevent any damage coming to the files:



With the required variables for all stages of the program established, the remainder of main() is used to call the functions required for LFSR initialization, file mapping, and encryption/decryption. The first stage is the initialization, started by lfsr\_array\_init():

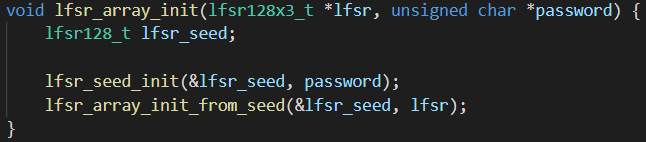


## LFSR Initialization

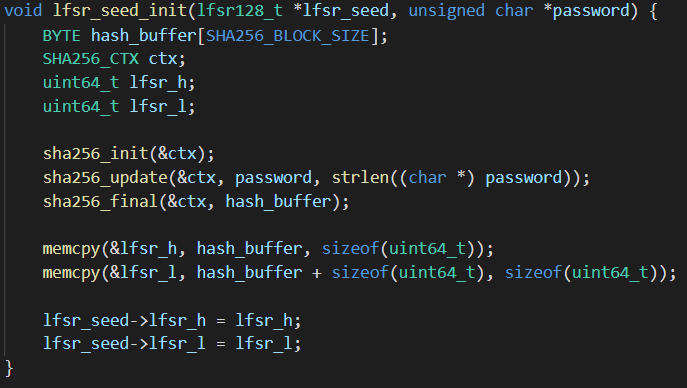
The first stage requires the initialization of a seed value. This seed is created by generating a SHA-256 hash from the user entered password. This seed, when made, is then used to initialize the values of each lfsr\_h and lfsr\_l for each LFSR for the program. After the seed and the password have served their purpose, they are discarded. This means that the values are irrecoverable, adding to the difficulty of reversing the encryption algorithm. This process occurs over five different functions.

### Seed Initialization

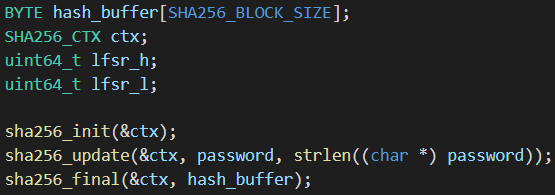
In the first function used for initialization, a variable is created to store the seed for the LFSRs, alongside two different function calls:



Within the first function call, the seed storage variable is passed alongside the password. The seed variable has its address passed because this seed is required for both functions, and is needed as it is. This will be a reoccurring factor for aspects related to the LFSR. Within lfsr\_seed\_init(), the following code is found:



The function begins with variable declarations.



The first and second declarations are used for the SHA-256 generation, in addition to sha256\_init, sha256\_update, and sha256\_final. This is the only time the password is used in the program. The password is sensitive information so to ensure its integrity and secrecy, it is used for sha256\_update(), and then never used again. This means that it is effectively discarded so there is less information for a cryptanalyst to try and use against the program to reverse it.

The way the SHA-256 hash is created will not be explored in detail, but the simplified version of it is that the user entered password is given to the program to create a hash out of. The SHA-256 is created using code files downloaded from Brad Conte’s GitHub account. The generated SHA-256 is stored in the hash\_buffer variable, which has a size of SHA256\_BLOCK\_SIZE, something declared in the header file of Conte’s program:



The macro is a 32-byte digest. There are separate instances of lfsr\_h and lfsr\_l in the function, as well. Due to the datatype of the seed, these duplicates are used to hold onto 64-bits of the created SHA-256 for initializing the seed. To get the values into the lfsr\_h and lfs\_l variables, a memcpy() is utilized, one that copies the first 64-bits of the hash into lfsr\_h and the second 64-bits into lfsr\_l:



To put both initialized 64-bit values into the seed, the arrow operator is required, seen in the last two lines of the function. The right arrow operator is used to access specific parts of a struct. In this case, because the seed is a datatype of lfsr128\_t, there is both a lfsr\_h and lfsr\_l for the seed. The explanation in words can be cumbersome to follow, so an isolated image of the command will be provided with an explanation below it to describe what is happening.



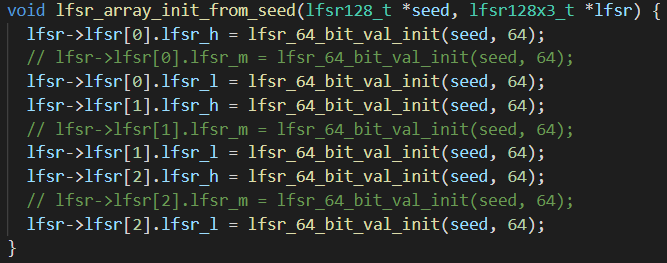
To copy the stored values from the hash into the seed properly, the following command structure is used: the variable that has a datatype of a struct is listed first, which is lfsr\_seed. When followed by the right arrow operator, “->”, it means that part of the struct is going to be accessed. In this case, both lfsr\_h and lfsr\_l are part of the struct for lfsr\_seed. Following that is the equals sign, equating the variable in the struct to the desired value. In this case, the intent is to make the structs lfsr\_h and lfsr\_l equal to the lfsr\_h and lfsr\_l created in this function, storing 64-bits of the SHA-256 hash inside of them. It is important to understand the arrow operator as this is not the only appearance of it in the code.

### LFSR Initialization Using Seed

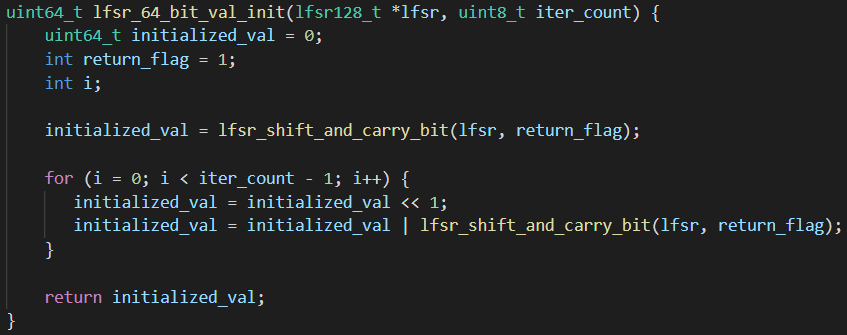
Back in the calling function, lfsr\_array\_init(), the next function call takes place, this time passing both the now initialized seed, and the lfsr variable declared in main():



Additional groundwork for a 192-bit implementation is present, but it is also commented out as to not affect the program. The intention of the function is to use the arrow operator to access the structs for each LFSR, and equate their lfsr\_h and lfsr\_l variables to appropriate values:

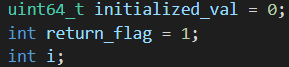


These equating values are gathered from the function lfsr\_64\_bit\_val\_init(), which takes both the now initialized seed and the integer value 64. Inside the function, the following code is present:



This function is reliant on other functions, and, as stated earlier, will be followed.

The code is being expressed in order of its execution, not appearance. The intent of this function is to use the seed that was passed in to initialize a 64-bit value that will be returned to the calling struct member from the previous function. This process begins with necessary initializations.

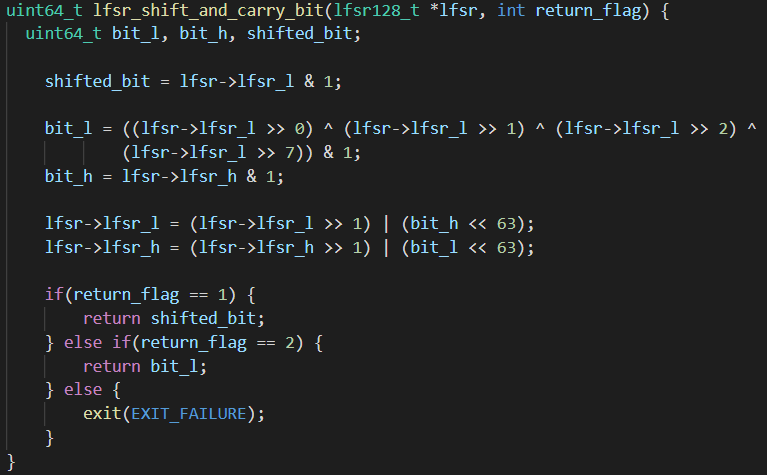


A variable is required to house the created 64-bit integer value that is being returned, so initialized\_val is declared and initialized. The next initialization is for the return\_flag variable. The function that uses return\_flag as a parameter is called upon by more than one function to do similar tasks. To differentiate the calling functions, and the desired return value, a simple flag is used. This will be explored further in the appropriate function. Lastly, is the variable i, only used for a for().

After the declarations is a call to a function. The returning value from lfsr\_shift\_and\_carry\_bit() is stored in the variable initialized\_val. The function is also passed the lfsr and return\_flag, which is equal to 1:



This function contains the following code:



The intention of this function is to programmatically create the functionality of a linear feedback shift register. This means using the XOR on the selected taps, and then shifting all values over linearly before copying the shifted-out bit into the first bit’s position of the 64-bit binary sequence. Depending on the flag value that was used, either the shifted bit is returned or the carry bit, which is stored in bit\_l is returned. This function begins with the necessary declarations. Afterwards, the low part of the register is logically AND’d with 1:



At this point, the lower half of the LFSRs are almost entirely empty, so lfsr\_l would be equal to 63 zeroes and a single one at the end. That means that shifted\_bit is equal to 1, avoiding an all zero state from appearing. The value of shifted\_bit, because of the logical AND, will be the last bit of lfsr\_l, the one that would have been shifted out. The following line is meant to utilize the taps for the LFSR:



The best taps for a 128-bit LFSR are bits 121, 126, 127, and 128. Though the LFSRs are 128-bits in size, they have to be broken into two halves because this code is created on a 64-bit architecture. Additionally, this is code that is made for the Windows 10 operating system, which uses little endian. Considering both of these points, lfsr\_l holds bits 65 to 128, but in reverse order, meaning the first bit that appears is the 128th bit. This is important because the bits are assigned as taps based on that knowledge. If the first tap is to be the 128th bit, then a shift to the right by zero will get that. To get the 127th bit, a shift to the right by one is desired, a shift of two for the 126th bit, and a shift to the right of seven to get the 121st bit. Afterwards, to get the last bit of the lower half of the LFSR, a logical AND can be used. The output of this operation will store carry bits from the pseudo-random LFSR output into the variable bit\_l.

A similar logic is used on lfsr\_h to get the carry bit for it:



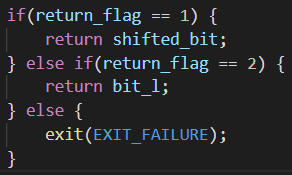
Because the taps do not exist in the bits that are held by the high half of the LFSR, the taps cannot be used. Despite that, the high half can still be logically AND’d. This will get the same value of the carry bit into the bit\_h variable.

The next two lines are used to do the shifting of the entire LFSR, and then put the carried bit back into it.



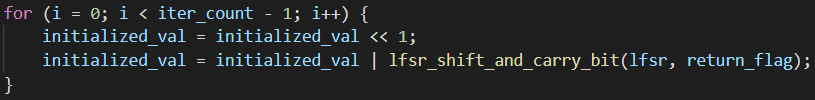
The LFSR has to be broken into two 64-bit halves, so they need to be shifted independently. Looking at only lfsr\_l in the struct, the value of lfsr\_l is shifted to the right once. A logical OR is run against it, using the value of bit\_h shifted to the left 63 times. The value of bit\_h after shifting would be the current value of bit\_h, but with 63 padded zeros added to it. This leaves the value in bit\_h at the start, and the 63 zeroes come after it. The end result of this operation would put the same value as what is present at the end of lfsr\_h into the correct position in lfsr\_l, creating the shift to the right without losing a value and padding by zero. This same logic is applied to lfsr\_h.

The remainder of the function is using the return\_flag to return the appropriate information to the calling function:



In this case, because the return\_flag was equal to 1, the shifted\_bit variable will be returned.

Back in the calling function lfsr\_64\_bit\_val\_init() a for() is run that will run 64 times, the number of times equal to the number of bits in a half of the LFSR. This is also the value that was passed from the previous function, lfsr\_array\_init\_from\_seed().



When this loop runs, a value in initialized\_val will be shifted to the left once, and then initialized\_val’s value will be logically OR’d with the shifted\_bit from the function lfsr\_shift\_and\_carry\_bit(), which is the same as it was before. Because the value in iter\_count is 64, the loop will run 64 times, which means that initialized\_val will, at the end of the loop, have a 64-digit unsigned integer value stored in it. This is the proper length for a half of the LFSR. After the for() is complete, the function ends by returning initialized\_val to the calling function:



The program is being followed as it functions, so that means, in this case, the value for lfsr[0]->lfsr\_h has been initialized to a 64-bit unsigned integer value. This process is repeated for every 64-bit section for each LFSR. Once complete, all LFSRs have been initialized successfully. The seed is no longer required so it is never used again, so to effectively throw it away and reduce the amount of information that someone would have to try and reverse the LFSRs with.

With this, the initialization of the LFSRs is complete.

## File Mapping

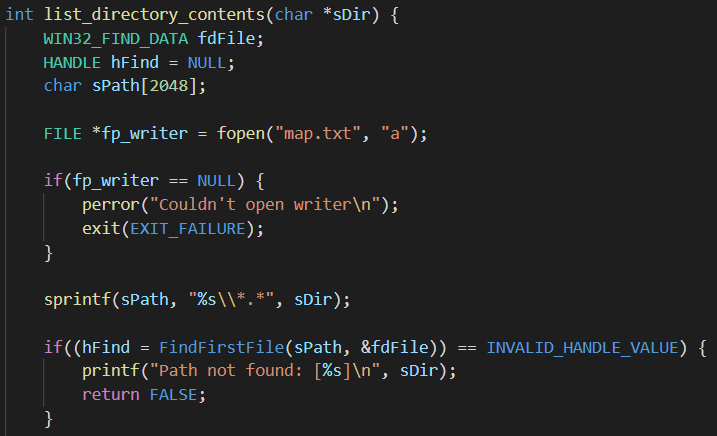
To ensure everything that is being encrypted or decrypted is known, a map of the files needs to be created. The most reliable method for doing this was to create a function that calls itself recursively. Due to the recursive nature of the function, there is no need for any other functions so long as the currently discovered files and folders are tracked, and the current folder and the files and folders inside of it are known. With that, the end of the program is reached when no further files or folders exist to read. This mapper function will take advantage of Windows specific features to read the file attribute to discern if what is being looked at is a file or a folder.

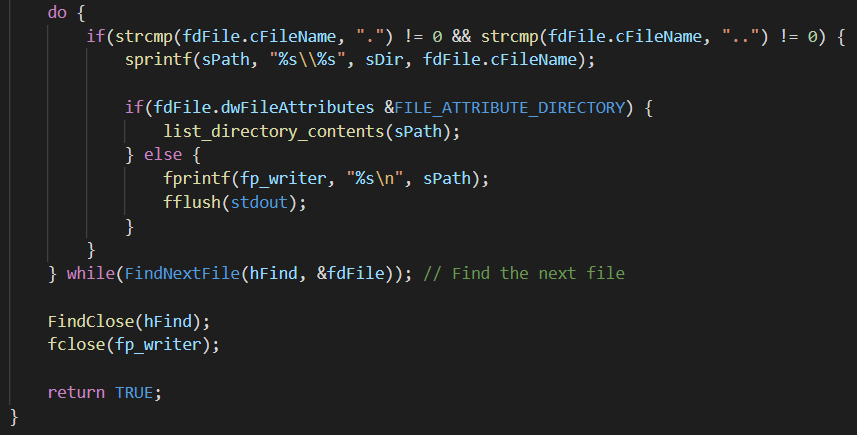
### Recursive Mapper

There exists a single call within main() that calls the singular function used for mapping the structure of the root folder.



The code for the function is as follows:



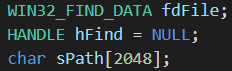


The operations of this code repeat the number of times equal to the number of directories that exist within the root folder. This is done recursively, utilizing the current directory’s file path to map out what files and folders exist within it until the last one has been reached.

This process begins with declarations, but because this program is meant to work on Windows, two Windows specific header files are required:



These header files will allow for Windows specific information to be checked to determine whether a file or folder is being returned recursively. The declarations for the function are:



The WIN32\_FIND\_DATA data type is a struct that is used for reading system information about a file. In this case, the variable using it is called fdFile. The next declaration is for a handler that will store references to specific system resources or objects. sPath is a simple array to store the complete path of the file or directory. Considering the length of a file path when a very nested folder or file is found, 2048 was chosen as an adequate size of the array.

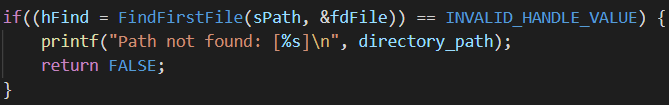
To circumvent an early design flaw, which could not be properly corrected due to time constraints, a text file is used to store all files and their respective absolute paths.



The problem in question was the decryption could, under certain circumstances, go in a slightly different order than the encryption took place in. This meant the states of the LFSRs were not the same, meaning that files that should have been decrypted were still entirely encrypted or partially encrypted. The map.txt is used to create a linear list that is read and enforces an order on the decryption, such that it will follow the trail left by the encrypter itself. This patchwork solution will introduce additional complications into the program, but those will be discussed when they appear.

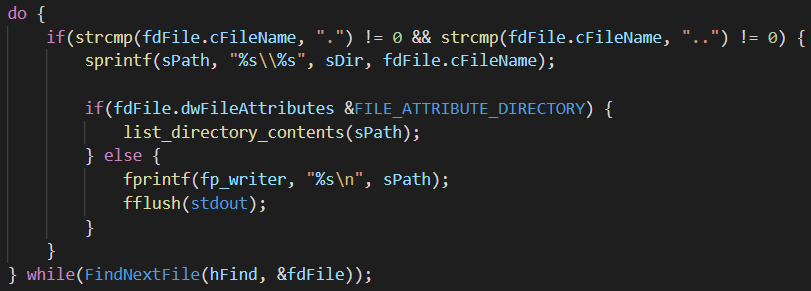
The file path of the root directory is stored in sPath by using an sprintf().

When copying from directory\_path, a format file mask of “\*.\*” is used to get everything, regardless of width or precision of the formatted string. Two slashes are used because one has the escape the other, and this should be appended to the end of the directory\_path value before adding anything else to it. With a safe copy of the current directory path, it is then checked to see if it exists:



The handler hFind is made equal to the WIN32\_FIND\_DATA’s feature FindFirstFile. The name of the function is self-explanatory, but it will use the current directory, in this case it is SecureFolder, stored in sPath and the pointer to fdFile which has information about the root folder. Should the file or directory exist, hFind will be equal to the search handle that can be used for further searching. If not, then hFind will be equal to INVALID\_HANDLE\_VALUE, and the program should exit.

At this point, a do-while() is used. A do-while() is necessary because the loop must execute at least once before moving on with the recursion or the remainder of the function.



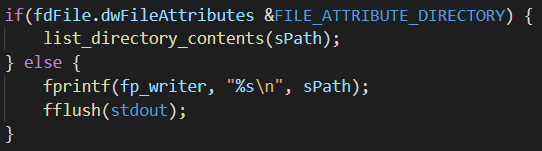
The if() contains two string comparisons.



These are checking if the filename of the file is equal to “.” or “..” respectively. This is done by accessing the struct of file data and accessing the cFileName member to get only the filename. Should their names not match, the contents of the statement can be run. Inside of the if(), another sprintf() is used. In this if() the program is made in a way to ignore folders that are named “.” and “..” as they are hard links to other file structures inside that folder, they are important to the file structure and cannot and should not be altered.



This one will copy the contents of directory\_path, followed by a single slash, and then the filename of the file or directory to it before copying that into sPath. Inside the WIN32\_FIND\_DATA struct, there is a way to check if the information is for a file or folder.



If the file attribute is that of a folder, the current sPath is returned back to list\_directory\_contents() so it can be utilized for the recursion, else a file was found. This name of the file is written to the created text file, and then stdout is flushed. If a file was found, this means that the end of the directory may not have been found yet, so the while statement will use the WIN32\_FIND\_DATA’s FindNextFile() to get the next file in the current directory.



Once there are no more directories to cause recursive calls, and the files have all been read, the while() will exit. At this point, the handler needs to be closed. As does the file pointer. With the entire structure traversed and mapped, the program returns to main().

## Encryption and Decryption

The encryption and decryption are handled through multiple functions. The first part of this phase is to read the last three letters of the filename. If the last three letters of the file are “enc,” then that file has been encrypted already. If the “enc” extension is missing, then the file is safe to encrypt. The goal with this is to avoid double encryption of a singular file. In the case of a file being encrypted, the “.enc” extension is added to the file. If the file is decrypted, the “.enc” extension is removed from the file. The actual encryption itself is done by writing 4096 bytes worth of encrypted information into the file. If the file is being decrypted, the same LFSR initialization values are present, so when the encryption is run over encrypted data, it will decrypt it instead. The functions used for encryption and decryption are the same with no changes between them for this reason.

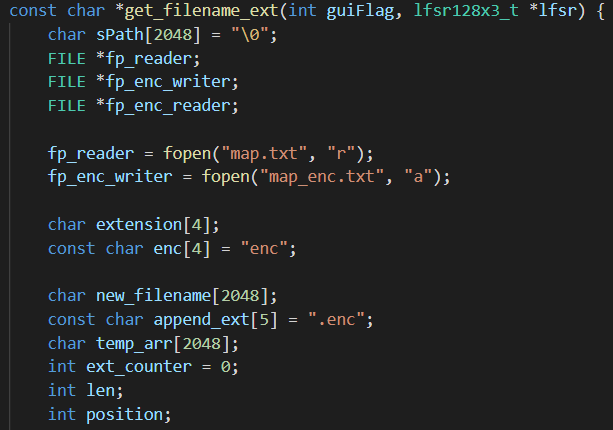
### File Extension Reading

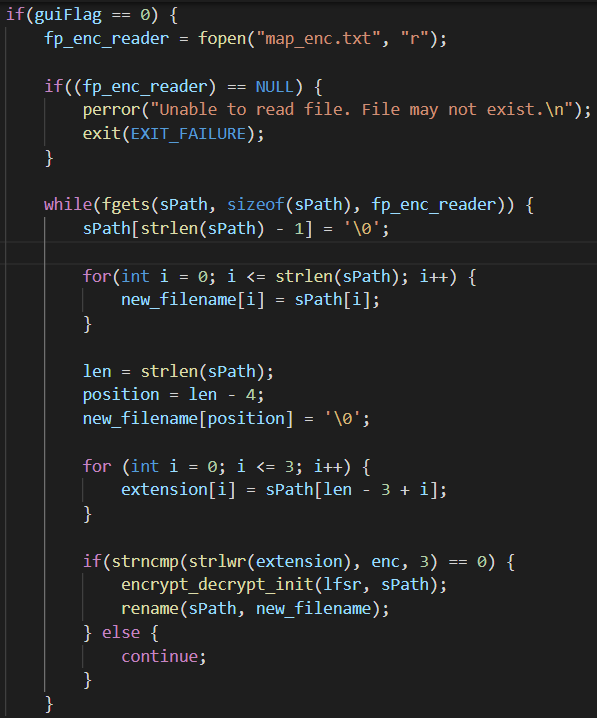
The encryption/decryption process is initiated by a call from main():

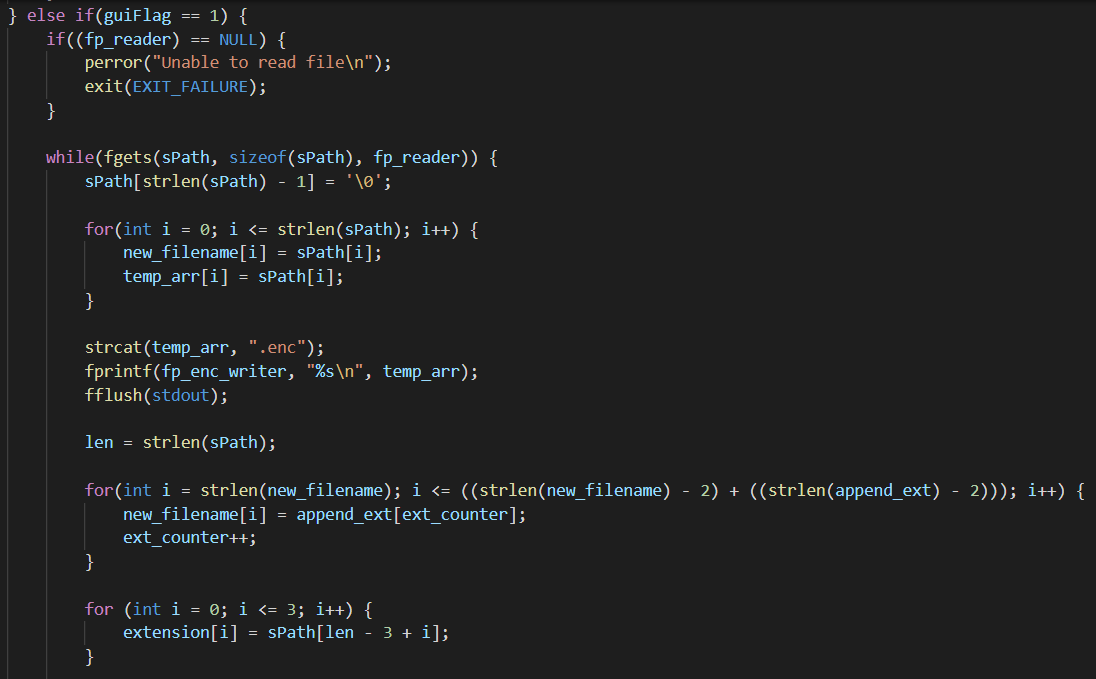


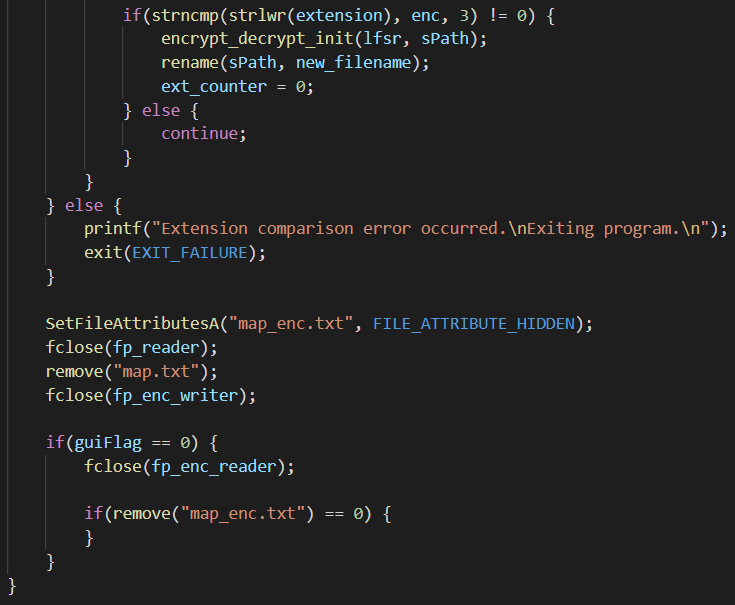
The gui\_flag, storing either 1 or 0, where 1 means the user pressed the encrypt button in the GUI or 0 meaning the user pressed the decrypt button, is passed alongside the address of the current state of the LFSR. The intent of get\_filename\_ext() is to read the filenames extension. If the last three characters of the extension end with “enc,” then the file has been encrypted already. This is important to ensure that double encryptions don’t happen. This also means that a decryption isn’t done on a non-encrypted file, as well. Both the encryption and decryption are done using the same functions with no differences between them. This means that a decryption on a non-encrypted file would end up encrypting it. This needs to be avoided, so the “.enc” extension is added onto any file that has been encrypted by the program for easy, automated identification.

Within get\_filename\_ext(), the following code is found:

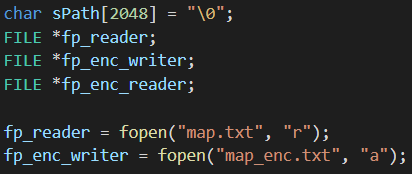






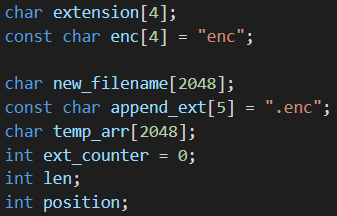


Due to the amount of code present in the function, the code will be reviewed in chunks. The function begins with needed declarations and creating file pointers:



sPath is used to store the current file path without altering the original one. The fp\_reader is for parsing the map.txt, and fp\_enc\_writer is for creating a map for the encrypted files.

Further variable declarations occur.

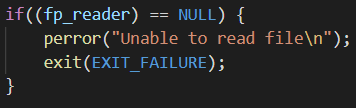


These are used for the appending of “.enc” to the end of encrypted files, or the removal of it from decrypted files.

The function uses an if-else if-else statement that operates based on the flag value passed by the front-end. For this report, we will assume they chose the encrypt option. The check for this is in the else-if:



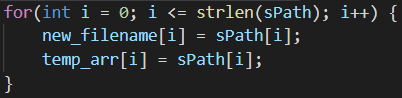
This starts by checking the map.txt reader:



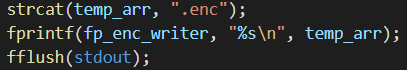
If there is an error with reading the file, an exit should take place now to ensure the integrity of the files. The next line is the start of a reader that will read every line of the map.txt file so that it can ultimately be given to the encrypter:



The contents of the line that is read from the file is stored into sPath, which is then copied into new\_filename and temp\_arr:

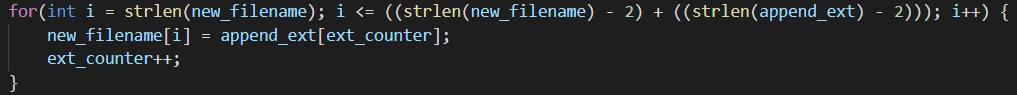


temp\_arr is used to hold a safe version of the original file path with the “.enc” extension at the end, which is done below the above for():



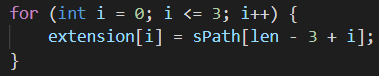
After appending “.enc,” the full path is written into fp\_enc\_writer, and standard output is flushed.

The following for() is, putting simply, used to get to the end of new\_filename, where the newline is, and then overwrite that with “.enc” and the newline operator of the append\_ext array:

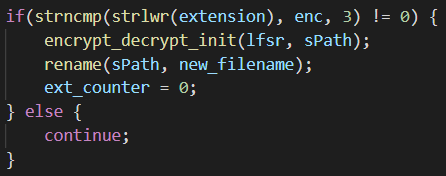


To iterate and write the extension one character at a time, a counter is used in the format of ext\_counter.

At this point, the extension of the existing filename within sPath is copied into the extension variable:



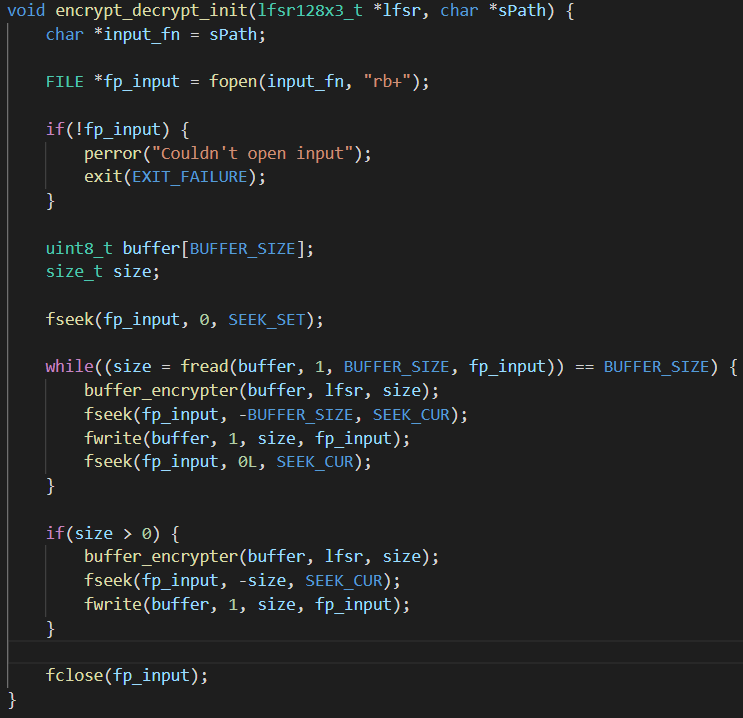
This is done for easy comparisons. The remainder of the else-if is doing the comparison in lowercase to ensure the extension is not equal to “enc,” so the encryption process can be initialized.



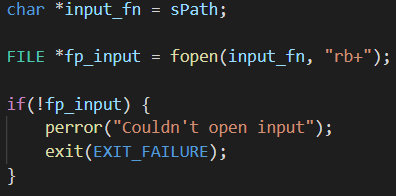
After initialization, the “.enc” extension is appended to the file path, and ext\_counter is reset. If the extension is equal to “enc,” the process will run the while() once more to get the next file in map.txt. The LFSR and the current sPath from the file is passed to encrypt\_decrypt\_init, and then, afterwards, sPath is renamed to the string stored in new\_filename.

### Initializing the Encrypter/Decrypyer

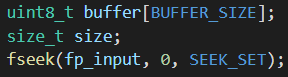
This function is used to establish the data required for the remaining functions that handle the encryption or decryption. The code for this file is as follows:



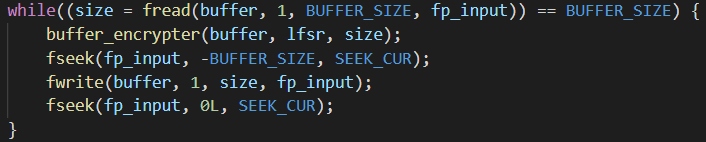
The process begins by taking a copy of sPath and putting it into a variable that can be used for opening a binary file reader and writer, which is followed by an error checker for the file pointer:



Before any of the real work can be done, some final declarations must be done.



The variable buffer uses a size that is from a definition created in the header file. For this program, BUFFER\_SIZE is equal to 4096. The next variable, size, is not initialized, but it is declared as a size\_t variable, allowing for the largest data storage the system can support. Lastly, an fseek() is used to ensure the cursor is at the beginning of the file. With everything set up, the encrypter/decrypter is ready to work:



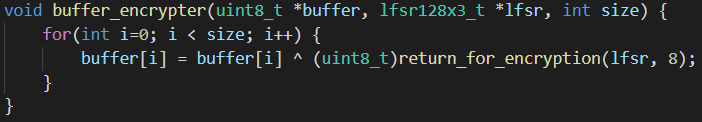
The above block of code starts with the while():



This while() will run so long as the variable size is equal to BUFFER\_SIZE, so 4096. The size of the size variable is determined by the amount of data read using fread(), which will store the information read into buffer, reading a single bit at a time for a total of 4096 bits, and using the fp\_input file pointer to do it, which is for the opened file from the start of the function. The nature of fread() is to iterate the cursor for an fseek() on its own equal to the amount being read, which is 1 bit,, so no cursor position manipulation is required. The first line in the while() is a call to buffer\_encrypter(), which takes the buffer, LFSR, and the size.

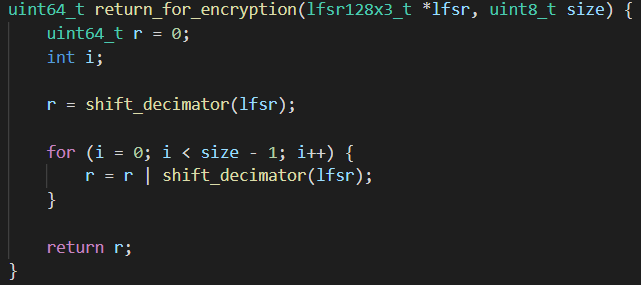
### Encrypting/Decrypting Files

The design of buffer\_encrypter() is to write into the buffer array the encrypted data that is equivalent to the original data within it. The code within buffer\_encrypter() is as follows:



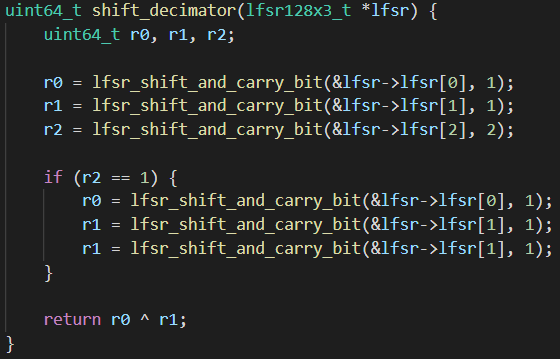
The for() will run for 4096 iterations. Each iteration will take the current character in the buffer array and replace it with its encrypted equivalent by using a logical OR on that character with the returned value from return\_for\_encryption(), which is passed the LFSR and the integer 8. This integer is used in a for() in the function.

The code for return\_for\_encryption() is as follows:

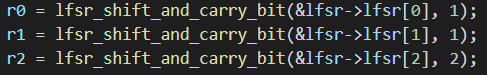


This function utilizes the shifting of the LFSRs to get bits returned from shift\_decimator() to get an obfuscated result. This obfuscation happens because of the multiple logical operations occurring. The function starts with a variable initialization with a uint64\_t datatype, and a generic int. The variable r is then made equal to the return value of shift\_decimator(), which is passed the LFSR.

Looking into shift\_decimator(), the code is rather succinct:



The start of the function is to declare three different variables as uint64\_t. They will then be initialized using lfsr\_shift\_and\_carry\_bit(), which is passed the address of an entire LFSR and a flag value, which was discussed earlier.



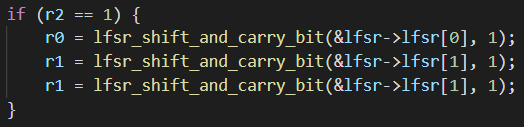
This flag will be used to determine which value is returned. In this case, LFSR1 and LFSR2 will be used to return the value of 63 zeroes and the bit that would have been shifted off after being logically AND’d, and the value of the return carry bit of LFSR3 is returned for r2 based on the XOR’d values of the selected taps, and that value being logically AND’d with one. The code for the shifted bit from lfsr\_shift\_and\_carry\_bit() is:



And the code that handles the return carry bit is:



The remainder of the function will handle the decimation of LFSR1 and LFSR2 depending on the value that was returned and stored in LFSR3:



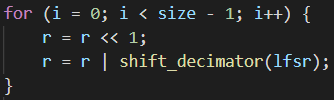
If the returned bit to r2 is equal to one, then r0 will shift past a single bit, and r2 will shift over two bits. The reason the shifting is not the same is to destroy patterns in the LFSRs states. Patterns are one of the primary ways of reversing LFSRs, contiguous binary states can be used against the program to find the initializing values, letting the algorithm be broken. By shifting past bits, the decimation achieves a simple manner of breaking the pattern, and by avoiding uniform shifting, each LFSR has to be repaired in a way that is not quite challenging, and different. These short few lines of code inject a great deal of complexity into the program, causing the reversal process to become extremely difficult.

The remainder of the code in the function is for the return statement:



The states of the LFSRs stored in r0 and r1, whether decimated or not, are XOR’d together and returned to the calling function.

Inside of the calling function, return\_for\_encryption(), the code continues with its execution.



This for() will iterate eight times, which was the integer passed to the function as a parameter from buffer\_encrypter(). Each iteration of the loop will shift the value of r, which is equal to the return from shift\_decimator(), once to the left. The value of the r is now logically OR’d with a new return value from shift\_decimator(). This is done to further confuse the state of the LFSR, and make reversal even more challenging. The ultimate goal of these shifts is to try and defeat algorithms such as the Berlekamp Massey algorithm, which is used to take discovered patterns of the LFSR and reverse it in the shortest manner possible. By decimating and then doing manual shifts, with logical OR’s, AND’s, and XOR’s, snippets of the LFSR cannot be fed into the Berlekamp Massey algorithm to find the initializing value, which is the password that is protecting all of the files. The function ends by returning the value of r:



The returned value is used in an XOR with the character in the buffer array within the buffer\_encrypter():



That XOR’d value is stored in the buffer array. This process repeats until all iterations of the for() are complete. After that, the calling function is returned to, which is the while() inside of encrypt\_decrypt\_init().

Because the while() uses fread, which automatically moves the cursor by the amount read from the file, and the desire of the program is to not create a separate encrypted version of the file; rather, write the encrypted data into the file itself, the cursor needs to be moved.



The cursor is moved by a negative BUFFER\_SIZE amount, meaning to move it back by 4096, the amount read from the file at this time. With the cursor in the correct position, the content can be written into the file:

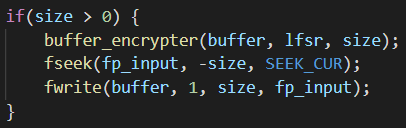


fwrite() has the same cursor moving property as fread, meaning the cursor will be moved forward by 4096. At this point, the while() can repeat; however, due to a quirk in C, an fwrite() and an fread(), or vise versa, cannot follow one another. To prevent complications in the program, an fseek() is used that does nothing with the cursor:



All the fseek() line means is that, from the current cursor position, the cursor should be moved by 0L amount, which is equal to zero. No changes take place, the cursor remains rooted, but the issue with fwrite() and fread() is circumvented.

The while() will only run so long as there are 4096 bits to read from the file, but it is likely that there will be a remainder that is below that. To encrypt the remaining bits, a similar statement is used, but one that will only grab what is left:

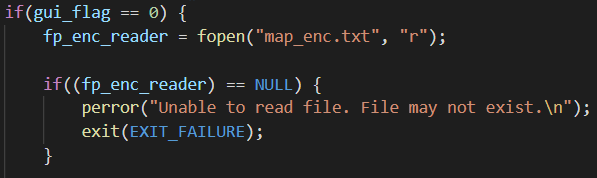


The functionality of the if() is similar to the while(); however, it only encrypts what is left in a single go.

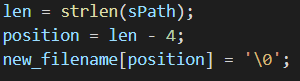
Before this function ends, the file pointer that was opened is closed:



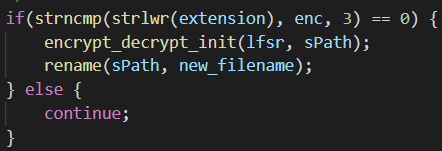
This returns the program to the previous, calling function, which is get\_filename\_ext(). The last part of this function deletes the map.txt file after closing the file stream for reading it, alongside any other opened file readers, and hiding the map\_enc.txt file, which is used for decryption. As there is no better place to transition to it, the decryption will be looked at here. Because an encryption has been run before, a hidden map\_enc.txt exists. This file will be read to prevent the aforementioned issue with the decrypter moving out of the path set by the encrypter, causing files to be partially or not decrypted at all. Considering the code is extremely similar to the encryption code, the differences will be looked at exclusively. They start by reading the map\_enc.txt file, and by doing a check on for the map\_enc.txt file:



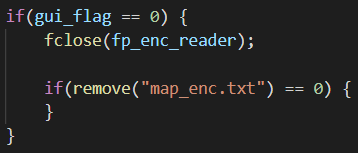
The variable new\_filename is given a copy of the value in sPath, just as before, but this time the aim is to remove the “.enc” at the end of the filename. This is done by replacing the start of the extension with a null terminator:



The last difference is to check if the extension of the file read from map\_enc.txt matches with “enc”:



If the match happens, the decryption is done with the same process as previously, and the files are renamed to remove the “.enc” part of their filename. To decrypt, the gui\_flag is equal to 0, so the end of the get\_filename\_ext() will use that value to close the reader for map\_enc.txt, and to delete map\_enc.txt:



With get\_filename\_ext() complete, main() is returned to. The only code left in the program to execute is a success:



# Strength of the Initialized 128bit LFSRs and the Reversibility of the Files

The chosen taps for a 128-bit LFSR will create a maximal-length LFSR that will iterate 2128 times, which is equal to 340,282,366,920,938,463,463,374,607,431,768,211,456. That is 3.4028237e+38, a 39-digit value, or 340 undecillion iterations. To drive the point home of how secure this program can be, some demonstrative math will be done. If a computer is capable of doing one million operations per second, the computer would be capable of generating 86,400,000,000 bits of the LFSR in a day.

**(60 \* 60 \* 24) \* 1,000,000 = 86,400,000,000**

To find how many days would be required to get through the 340 undecillion iterations, the total number of possible iterations can be divided by the previous value, producing the value 393,845,332,084,419,517,897.42431415713.

**340,282,366,920,938,463,463,374,607,431,768,211,456 / 86,400,000,000 = 393,845,332,084,419,517,897.42**

This value can be converted from days to years by dividing it by 365 to get 1,079,028,279,683,341,144.92.

**393,845,332,084,419,517,897.42 / 365 = 1,079,028,279,683,341,144.92**

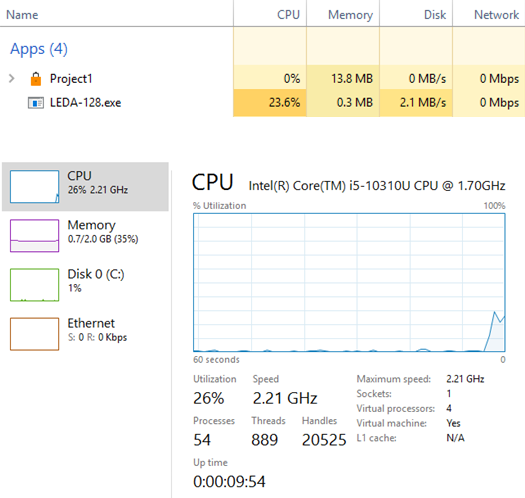
To give context to a number so high, the sun in our solar system is projected to live for about 4.6 billion years, and the universe in its entirety is expected to live 14 billion years from the time you are reading this. If a snapshot of the current state and age of the universe was taken, assuming it lives the projected 14 billion years, it would take over 77,073 snapshots of our universe before the initial state of the LFSR ever repeats.

The absurdity of these numbers means that the core code for the LFRSs can be used to create a virtually infeasible to crack one-time pad, or a potentially irreversible ransomware encrypter. It is not realistic to brute force, nor, in the case of this program, a good idea to guess random passwords. The possibilities are too many, and an incorrect password will generate a decryption that does not match the LFSR initialization that took place for the encryption, rendering all data irrecoverable unless decrypted with the same password and then the correct one to restore the files to their normal forms. From a security perspective, the consequences are almost absolute, meaning that the program cannot be played with unless in a controlled manner.

# 

# Testing

Testing was done inside of a Windows 10 virtual machine using VirtualBox. This VM sported 4 cores, and 2GB of RAM. During the encrypting or decrypting process, the program used roughly 25% of the VMs CPU, and had a nominal impact on the RAM, using about 20MB’s.



The testing times, across 10 tests for each category, was gathered on encryption and decryption runs on 867MB worth of files spread throughout the SecuredFolder. These tests were done on both USB 2.0 and USB 3.0. The average of the speeds in minutes are as follows:

* USB 2.0 encryption timings: 8.9, 8, 8.6, 8.3, 8.1, 7.9, 8.9, 8.4, 8.4, 8.5
  + USB 2.0 encryption time average: 84/10 = 8.4 minutes
* USB 2.0 decryption timings: 4.8, 5.2, 5.5, 4.2, 3.8, 4.8, 5.4, 4.8, 4.4, 4.6
  + USB 2.0 decryption time average: 47.5/10 = 4.75 minutes
* USB 3.0 encryption timings: 6, 6.3, 5.5, 5.1, 6.1, 6.4, 6.2, 6.9, 5.5, 6
  + USB 3.0 encryption time average: 60/10 = 6 minutes
* USB 3.0 decryption timings: 5, 5.2, 4.9, 4.2, 3.3, 4.6, 3.4, 4.2, 4.4, 4.2
  + USB 3.0 decryption time average: 43.4/10 = 4.3 minutes

The decryption speeds are lower than the encryption speeds. Through research, we were unable to determine why this is the case, but a consistent answer is that some parts of the process are precomputed. Which parts, we are unable to define, but that was the answer that was consistent and applied to LEDA-128.

# 

# Problems, Future Improvements, and Recommendations

Creating an encryption algorithm is complex and difficult. Due to this, and our blossoming understanding of encryption as a topic of study, there are a number of issues, improvements, and overall recommendations that can be made for LEDA-128.

Firstly, the Berlekamp-Massey algorithm can be used to find the shortest LFSR from a binary output, meaning that, should the decimations not occur, the entire encryption algorithm is completely broken. Though it is something we actively tried to combat, it may be possible that we made a mistake and LEDA-128 is vulnerable to it. We do not believe this to be the case, however.

On rare occasions, the state of an LFSR can become all zero, which breaks the program. This can be checked for and prevented, but, due to its rarity, was not a priority during development.

LEDA-128 lacks any anti-debugging measures. A conscious choice was made to not include simple anti-debuggers. Though something is better than nothing, simplistic anti-debugging measures are easily identified and reversed. They offer no more than a minor inconvenience to the person reversing the program. A more advanced measure is required, but they are not simple to implement. Due to this, there is no anti-debugging in the code, but it can be added.

Functionally speaking, there exists three requirements of an encryption algorithm, and one of them is speed. LEDA-128 fails that front. The focus on strengthening the encryption has made it slower than intended. When considering that this is not a commercial product, or one offered to others as a solution to a problem, the issue is rather small, but if any further development on this encryption algorithm is pursued, the poor speed is something that needs to be addressed.

The program lacks any multi-threading, meaning the operations are slower than anyone would want. The security of the program is theoretically high, but a good encryption algorithm requires both speed and strength. LEDA-128, unfortunately, only meets half of the requirements.

There are no restrictions on the password that the user chooses. This means that a password as simple as a single character can be chosen. Should there have been more time, this is something that could have been addressed, but the inherent weakness of the program still stands.

LFSRs are susceptible to stream cipher attacks. There are measures that can be incorporated into the code to combat these kinds of attacks, but the current version of the code does not contain those features.

Speaking more broadly, further optimization can be made to improve the program. There are also areas where the code can be written in a better way, such as get\_filename\_ext(), where the contents of the encryption and the decryption should be moved to their own functions, alongside some rewriting to improve the readability of the code.

It is possible to eventually brute force the password. When the files are encrypted, it is often rather obvious when they are decrypted because the content of the file is likely in something that is human readable. If the incorrect password is entered, the encryption taking place can be reversed by doing the same action with the same password, returning to the initially encrypted state. This means a brute force can be done by attempting a password, and, should the contents of the file still appear encrypted, reverse it and try again. Eventually, especially because there is no password enforcement on the program at the time of writing this, the password can be guessed or broken with the assistance of the popular RockYou password file and a Python script that volleys the passwords into the GUI, and checks a file to see if it is human readable again. The successful password can be stored and likely used again because, statistically speaking, people tend to use the same passwords across different accounts and mediums.

# 

# Conclusion

We believe that nothing is perfect, and LEDA-128 is no exception. It has room to grow and develop; however, what is there is a very strong custom encryption algorithm that is designed to combat some of the specific reversal measures meant for an algorithm like it. As it stands, the program functions simply, and operates seamlessly. It can be run on any 64-bit Windows 10 USB, and will secure any file within a specified folder. The files that are secured are difficult to break in the ways that protection was integrated into it. Through a five step process, files can be secured: the user enters a password into the front-end application, and then selects their method, whether to encrypt or decrypt files, and then the program will handle the next three phases. These phases are the initialization, file mapping, and encrypting/decrypting. This results in a secure encryption of the contents of the files in the secure folder. Though the application has many strengths, it does have its flaws. Some of these flaws are that there is a lack of multi-threading, making both encryption and decryption times less than ideal, a lack of debugging protection integrated into the program, and the lack of password requirements. Despite these flaws, the overall program is functional, and achieves the goals set out.

# References

[1]T. Gibson, *Cryptography: Linear Feedback Shift Register*. 2019. [Accessed: 19- Jan- 2022].

[2]B. Zhu, "An Online Calculator of Berlekamp-Massey Algorithm", *Bma.bozhu.me*, 2022. [Online]. Available: https://bma.bozhu.me/. [Accessed: 20- Jan- 2022].

[3]H. Ayguen, "GitHub - hayguen/mlpolygen: a maximal-length polynomial generator for linear feedback shift registers", *GitHub*, 2021. [Online]. Available: https://github.com/hayguen/mlpolygen. [Accessed: 20- Jan- 2022].

[4]B. Conte, "GitHub - B-Con/crypto-algorithms: Basic implementations of standard cryptography algorithms, like AES and SHA-1.", *GitHub*, 2012. [Online]. Available: https://github.com/B-Con/crypto-algorithms. [Accessed: 22- Jan- 2022].

[5]C. Romine, *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION*. Gaithersburg: US Department of Commerce, 2015. [Accessed: 28- Jan- 2022].

[6]K. Post, "GitHub - klauspost/mman-win32: Automatically exported from code.google.com/p/mman-win32", *GitHub*, 2022. [Online]. Available: https://github.com/klauspost/mman-win32. [Accessed: 03- Feb- 2022].

[7]D. Ritchie, *C Reference Manual*. Murray Hill: Bell Telephone Laboratories, 2002. [Accessed: 04- Feb- 2022].

[8]R. Ward and T. Molteno, *Table of Linear Feedback Shift Registers*. Dunedin, 2007. [Accessed: 08- Feb- 2022].

[9]S. Nurak, *Build C++ GUI apps FAST! Your first C++ GUI app!*. 2021.[Accessed: 09- Feb- 2022].

[10]E. Weisstein, "Primitive Polynomial -- from Wolfram MathWorld", *Mathworld.wolfram.com*, 2022. [Online]. Available: https://mathworld.wolfram.com/PrimitivePolynomial.html. [Accessed: 10- Feb- 2022].

[11]J. Weber, "Password Strength/Entropy: Characters vs. Words", *Weberblog.net*, 2013. [Online]. Available: https://weberblog.net/password-strengthentropy-characters-vs-words/. [Accessed: 16- Feb- 2022].

[12]"Linear Feedback Shift Register", *ScienceDirect*, 2022. [Online]. Available: https://www.sciencedirect.com/topics/mathematics/linear-feedback-shift-register. [Accessed: 22- Feb- 2022].

[13]G. Banegas, *Attacks in Stream Ciphers: A Survey*. São José, 2014. [Accessed: 26-Feb- 2022].

[14]G. Banegas, *Attacks in Stream Ciphers: A Survey*. São José, 2014. [Accessed: 26-Feb- 2022].

[15]T. Krovetz, *Fastcrypto.org*, 2007. [Online]. Available: https://www.fastcrypto.org/vmac/draft-krovetz-vmac-01.txt. [Accessed: 28- Feb- 2022].

[16]D. Singer, *SECRETS OF LINEAR FEEDBACK SHIFT REGISTERS*. 2020. [Accessed: 28- Feb- 2022].

[17]E. Barkan, E. Biham and N. Keller, *Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication?*. Haifa: Israel Institute of Technology, 2006. [Accessed: 29- Feb- 2022].

[18]R. Anderson, *On Fibonacci Keystream Generators*. Cambridge, 2006. [Accessed: 29- Feb- 2022].

[19]B. Toulas, "Ten notorious ransomware strains put to the encryption speed test", *BleepingComputer*, 2022. [Online]. Available: https://www.bleepingcomputer.com/news/security/ten-notorious-ransomware-strains-put-to-the-encryption-speed-test/. [Accessed: 16- Mar- 2022].

[20]P. Lab, "General Format // Purdue Writing Lab", *Purdue Writing Lab*, 2022. [Online]. Available: https://owl.purdue.edu/owl/research\_and\_citation/ieee\_style/ieee\_general\_format.html. [Accessed: 28- Mar- 2022].

[21]P. Lab, "General Format // Purdue Writing Lab", *Purdue Writing Lab*, 2022. [Online]. Available: https://owl.purdue.edu/owl/research\_and\_citation/ieee\_style/ieee\_general\_format.html. [Accessed: 28- Mar- 2022].

# 

# Appendix

## Keywords

All-zero - When every binary bit of an LFSRs state is zero

Ansi-string - A single-byte string

Bit-flipping attack - Flipping the bit value of ciphertext so that that, when decrypted, the plaintext value will be changed, resulting in unintended outcomes

C-String - One dimensional array or sequence of characters that are terminated using a null terminator

Cryptanalysis - The work of reversing an encryption algorithm to break it

Decimate - Shifting past bits of an LFSR

Decrypt - Return ciphered text into their original characters

Encrypt - Convert characters into ciphered text to hide their original value

Little endian - The order in which the least significant bit of the sequence is stored first

Logical AND - Boolean operands that returns true if, and only if, all the operands are true

Logical OR - returns the boolean value true if either or both operands is true, and returns false otherwise

Maximal-strength - The highest level of strength that can be achieved

One-time pad - An unbreakable encryption, provided certain conditions are met

Pseudo-random - Random appearing values that were produced by a deterministic input value

Ransomware - Malware that usually encrypts the victim’s data and ransoms them for payment to have their data returned to them

SHA-256 hash - A hash function that will create a pseudo random 256-bit output

Seed - Pseudo-random initializing value for an LFSR

State - All binary bits for an LFSR that compose it

XOR - A bitwise operation also known as exclusive or, returns true if the two input values are different