**The Golang AES File Encryption Project**

By: Jordan Murray

Department of Computer Science, Marist College, Poughkeepsie, New York, USA

**Abstract** - Files are constantly being transported, making them susceptible to data breaches, fraudulent data alterations, as well as other kinds of hacks. Thus, proving the importance of file encryption. The Golang AES File Encryption Project explores the methodology behind file encryption with a Golang server using gorilla mux. This serves a basic html page with a file selector that sends a text file to the backend, using a simple API. Once the file is on the backend, the contents of the file will be encrypted using the Advanced Encryption Standard Algorithm in Golang and the newly encrypted file is returned to the user in hexadecimal format. A comparison of the AES algorithm runtime in Golang as opposed to java, is also a captivating analysis, due to Golang’s growing popularity.

1. **Introduction**

Data privacy and protection is a very prevalent issue in cybersecurity. Files with sensitive data are transferred every day and are susceptible to many kinds of attacks. This served as motivation and inspiration for researching the Advanced Encryption Standard algorithm and how it can be applied to files.

This work is relevant and similar to work at my internship. We have a basic web

application set up as well, where files are sent to the back end and stored in a database. Thus, learning how to encrypt the files before storing them is useful knowledge as a programmer on this application. This is the only background and related work listed for this project.

Another motivation for this research is to compare the runtime of Golang on the AES standard, to the runtime of java. This

analyzation will provide fascinating data on which language is more efficient. Throughout my collegiate and professional career, I have seen Golang grow in popularity both in the classroom and the workplace. Hence, seeing this transition is a big motivation to compare the two languages I have the most experience with.

1. **Methodology**

The Golang Encryption project is a Golang server that spins up a basic webpage using gorilla mux. To set this up, Golang can be installed using homebrew for MacOS.

brew install go

A go directory was created under the prj folder in the repository and added go to my $PATH variable with the following commands:

export GOPATH=/Users/jordanmurray/go

export PATH=$PATH:$GOPATH

export PATH=$PATH:$GOPATH/bin

After the Golang setup, Gorilla Mux was installed using the commands:

go get -u github.com/gorilla/mux

go install github.com/gorilla/mux

The custom html page is served by running the commands:

go build main.go

go run main.go

because this is the file that spins up the webserver. The html page can be seen in a web browser and going to localhost:9000.

The initial setup of this page was two text boxes, one for plaintext hex and one for key hex, and a submit button.

Once the form was submitted with values, it gets sent back to the Golang server which performs the encryption. The AES algorithm is under its own file, AESCipher.go. The Aes method is called from main.go, where it passes in the values received from the front-end and performs the encryption. To run the full application with the encryption, the AESCipher.go file has to be added to the run command as such:

go build main.go AESCipher.go

go run main.go AESCipher.go

To program the AES algorithm in Golang, the same technique in the MSCS 630 java lab was used, with simple plaintext represented in 16-byte hex. This allowed the re-use of the data from the labs as test cases, to ensure the algorithm was working properly and providing the correct output. When the algorithm successfully completed, the output is returned and printed on a new page since a new route is called on the API.

Once the text encryption was working, the next step was to refactor the application for file encryption instead of plaintext. Golang has the ability to read in files as bytes. This allowed an easy conversion of the byte array to a hex string. Because of this feature, there was minimal refactoring needed to the AES cipher.

Only two methods were added AESCipher.go: parseFile and randomHex. The parseFile method has a signature of parseFile(content string) = string and the randomHex method has a signature of randomHex(n int) = string.

The parseFile method takes in a string value which is the bytes of the file read in converted to a hex string. This method breaks the hex into 16 byte sections and then calls the main encryption method. It encrypts the file 16 bytes at a time and loops, so the same key is used throughout the file.

If the number of bytes in the file is not divisible by 16, the remainder of the bytes were padded with 20 which is the ASCII character for space.

Once the encryption is done on the whole file, the parseFile method concatenates the output together and returns the string of the encrypted file represented in hexadecimal.

The randomHex method takes in an integer which is the length of the hex string to be generated (16 bytes) and returns the hex string to be used as the encryption key. Although the algorithm as written only handles 16 bytes at a time, having the ability to pass in the length needed, ensures the application is scalable. The cipher was tested without the API first, by creating a main method in the go file and manually passing in a file.

After ensuring the file encryption was working successfully, the client side of the application needed to be refactored. The form was refactored so that the input was a file selector and not text boxes. Main.go also had to be altered, so that it could handle file input and not just a string of text. This involved moving the go file reader into main.go instead of in the parseFile method in AESCipher.go. Once the file was passed to the backend and read into a byte array, the byte stream was converted to hex. This is the string passed into parseFile method in AESCipher.go

The output returned to the user is the hex string concatenated together. Hence, when the user clicks submit on the form, they are taken to the route that performs the encryption and the hex output is displayed on the screen.

1. **Experiments**

There were two main tests that were performed on the AES Cipher. The first was on the plaintext to ensure the algorithm was working properly and to compare the runtime of the algorithm to the runtime in Java.

A total of 10 test cases were run which can be seen in figure 1 below. These test cases came from lab 5 in MSCS 630. The test cases and the expected output are all listed in the table in figure 1, along with the runtimes for each test case in Java and Golang. The averages for these times are computed at the bottom. The time is measured in microseconds.

The second set of testing that was performed on the AES algorithm in Golang is for the file encryption. Testing was performed on multiple different file types, to ensure that the algorithm works. The file types tested were text file, word doc, pdf, and a jpeg image. These results can be seen in figure 2 below. Sample output was provided for the text file, but not the others because the output was too large.

1. **Analyzing Results**

The results in figure 1 are incredibly captivating. When running the test cases in both languages, the time was measured in microseconds. As seen below, the Java algorithm averaged at 31695.4802μs and the Golang algorithm averaged at 221.6128μs.

Comparatively, the Java algorithm took over 30,000 microseconds longer.

Contrastingly, the AESCipher.java file has about 500 lines of code, while the AESCipher.go has about 750 lines of code. It should be noted that the lines of code calculation for the go program does take into account the methods added for the file encryption portion. Looking at the lines of code makes the analysis even more captivating because the Golang algorithm is severely more efficient, despite having almost 250 more lines of code. Thus, providing concrete evidence for its capability and support for its growing popularity.

The results found in figure 2 are equally as fascinating. The expectation was that for different file types, the algorithm would have to be altered, but Golang’s ability to read in the bytes made the encryption much more versatile.

Figure 2 states the size of each of the files that are encrypted. This is important to note because the more content a file holds, the larger the file size, and the longer and harder it is to encrypt. Even a single page word document, produces a large output. The multi-page page pdf as seen in the table is even larger, hence provides even larger output.

The jpeg image was the largest file that was tested, which was surprising. A single image was not expected to be the largest or hardest file to encrypt yet proved true in testing. The explanation for this is the image has to generate bytes for all of the pixels, so the hex stream is very large since the image is a 2880 × 1140 grid.

1. **Conclusion**

After analyzing the results, it is evident that the demand and transition to Golang is warranted. This research shows that Golang provides a very efficient, versatile, and robust implement of the Advanced Encryption Standard algorithm.

Moving forward, I would like to implement a better UI for providing the encrypted file back to the user. Golang also provides libraries for creating files and storing data in them. Hence, the cipher that is returned could be placed back in a file of the same type and returned to the user.  
 From here, the file would be downloadable, so the user could take this and store the file in a database, such as at my internship or could be downloaded and transported however needed, such as through email.

Additionally, to further this research, it would be interesting to learn about the process of decryption using the AES algorithm. To extend upon this application for decryption, the user could send the encrypted file and the key used to encrypt and have it return the plaintext version of the file. Knowing the key is important for the user if they wanted to decrypt the file. Returning the key is necessary since the key is randomly generated on the server and is different every time. One good test case for this would be to just re-enter the file that was returned to see if the full encryption/decryption process is correct.

**Resources:**

“Documentation.” *Go*, golang.org/doc/.

Gorilla. “Gorilla/Mux.” *GitHub*, github.com/gorilla/mux#serving-single-page-applications.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Plaintext hex input | key hex input | Output String | Java Runtime | Golang Runtime |
| 54776F204F6E65204E696E652054776F | 5468617473206D79204B756E67204675 | 29C3505F571420F6402299B31A02D73A | 38059.981 | 241.131 |
| 43617365207769746820616C6C203173 | 11111111111111111111111111111111 | 6566BFDAE144226EDD0B9F31ABDA3298 | 36321.066 | 210.146 |
| 43617365207769746820616C6C204673 | FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF | 7C24B741D2BA535BE0A87138EFCBFABA | 27162.776 | 194.194 |
| 4361736520776974682030312E2E4673 | 0123456789ABCDEF0123456789ABCDEF | 42696B1B7E557B3D37B7A7F8A53B56AB | 26163.854 | 230.384 |
| 616C7465726E6174696E672030312773 | 01010101010101010101010101010101 | 32FF6A8E23EBA2B97CA47314DA9E9F56 | 26245.006 | 219.603 |
| 7261706964207461626C657320342075 | 48656C6C6F20576F726C64202121210A | 85233B96585F65C7F365981DE21E0B45 | 36955.239 | 210.719 |
| 64657363202D20696E202D20616E732E | 57656C636F6D65746F4D534353363330 | FCA3DF67558D4CCE57DCAB362FFE09AA | 36629.793 | 217.445 |
| 61697368776172796120706167616C61 | 44722E205061626C6F2052697661732E | 94E4929435E2D32FBC2B6D58A24AA416 | 27149.917 | 213.656 |
| 4E61676120416E757261672042656572 | 41534349492032204865782074657874 | 5BF1F1DDB0A3B8011F3C7013E222A70B | 31279.199 | 233.897 |
| 57696C6C2041492072756C652075733F | 4D617269737420436F6C6C6567652E2C | BCFA99B9F1EB90132E96DC6114D4BB08 | 30987.971 | 244.953 |
|  |  |  |  |  |
|  |  | Average Times: | 31695.4802 | 221.6128 |

Figure 1:

Figure 2:

|  |  |  |  |
| --- | --- | --- | --- |
| File Name | File Type | Number of Bytes | Sample Output |
| test.txt | text | 40 | 5E469E957B4D03BB94A1B18CB3A76735427DA80F4C7ADCA1A0655E81380238DF47289744BEE7269D4952C4FC50E12D70 |
| My Resume.docx | word doc | 23353 | Too Large |
| project-1.pdf | pdf | 71510 | Too Large |
| marist-college.jpeg | jpeg image | 1132892 | Too Large |